



**DUBLIN INSTITUTE
of TECHNOLOGY**
Institiúid Teicneolaíochta Bhaile Átha Cliath

Boppable – A Hybrid Mobile Application for Group Music Selection and Voting Final Year Project Report

DT228

BSc in Computer Science

Emmet Doyle

Bryan Duggan

School of Computing

Dublin Institute of Technology

15/04/2018



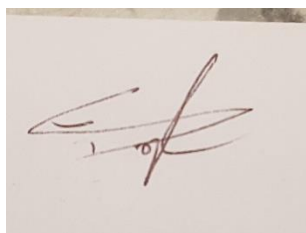
Abstract

This project aims to provide all attendants of a party or social event a means of having a say in what music gets played. Music at house parties typically comes from a playlist on the party host's phone or MP3 player. As the event progresses, guests of the party often begin to argue over what kind of music should be playing or insist on playing specific songs. This bickering can leave guests uncomfortable and frustrated which can in turn create a hostile environment and distract from the party's original purpose. This project aims to study how these arguments can be prevented and create an application based on the findings. A hybrid mobile application was created to allow the host of a party set up a room that the party's guests can join. Each guest can join the room on their own device and request songs to be played at the party via Spotify. Guests can also see every song that has been requested and upvote songs they like and downvote song they do not like. The song with the highest vote percentage gets played next, assuring the host that if a song is playing, most guests wanted to hear it.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

A handwritten signature in dark ink, appearing to be 'Emmet Doyle', written on a light-colored surface.

Emmet Doyle

15/04/2018

Acknowledgements

Firstly, I would like to sincerely thank my project supervisor, Bryan Duggan. Not only was his guidance and support incredibly helpful during my time working on the project, his enthusiasm for teaching in general has kept motivated and inspired during my entire time in DIT. He is a key reason I stayed in college when I was having trouble with the course, and I am eternally grateful.

I would like to thank my parents for believing in me and giving me the opportunity to continue my college career when things were not going as expected. Special thanks also to my twin brother, Kevin, for assuring me of my capabilities, and my sister, Sharon, for the support. I would also like to thank all the friends I have made during my time in college.

I would, lastly, like to thank Ash, whose emotional support and reassurance during my time working on this project proved critical.

Table of Contents

Abstract	3
Declaration	4
Acknowledgements	5
1 Introduction.....	9
1.1 Project Overview and Background.....	9
1.2 Project Objectives	9
1.3 Project Challenges.....	9
1.3.1 Full Stack Development	9
1.3.2 Client Application Challenges	9
1.3.3 Server Application Challenges	10
1.4 Structure of the Document	10
2 Research	11
2.1 Background Research.....	11
2.1.1 Typical Party Environment.....	11
2.1.2 DIT Snow Sports Party Case Study	11
2.2 Alternative Existing Solutions	12
2.2.1 crowdDJ	12
2.2.2 Jukestar.....	16
2.3 Music Sources Researched.....	19
2.3.1 Locally Stored Music	19
2.3.2 YouTube	19
2.3.3 Spotify.....	20
2.4 Other Relevant Research	20
2.4.1 Room Creation.....	20
2.4.2 Voting	22
2.5 Resultant Findings and Requirements Definition	23
2.5.1 Feature Selection	23
2.5.2 Requirements	24
2.6 Server Application Technologies Researched	24
2.6.1 Django.....	24
2.6.2 Deployment	24
2.7 Client Application Technologies Researched	25
2.7.1 Native Android Application.....	25
2.7.2 Cordova.....	25

2.7.3	React Native.....	26
3	Design	27
3.1	Methodologies	27
3.1.1	Waterfall	27
3.1.2	Scrum	27
3.2	Use Cases and Functionality Overview	29
3.2.1	Use Case Diagram	29
3.3	User Interface and User Experience.....	29
3.3.1	Choosing a Name	30
3.3.2	Style	30
3.3.3	Navigation.....	30
3.3.4	Screens.....	31
3.4	Server Application	40
4	Development & Architecture	41
4.1	System Architecture.....	41
4.2	React Native Basics	42
4.2.1	Flexbox.....	42
4.2.2	State and Props.....	43
4.2.3	Lifecycle Methods.....	43
4.2.4	AsyncStorage	43
4.3	UI Components	44
4.3.1	Party.....	44
4.3.2	Search Screen	44
4.3.3	Vote Screen.....	44
4.4	Party Lifecycle	44
4.4.1	Playing Songs	44
4.4.2	Updating Party.....	44
4.5	Server Application	45
4.5.1	Architecture and Database Models	45
4.5.2	View Methods.....	46
4.5.3	URL Endpoints.....	46
4.5.4	Deployment and Hosting	46
5	System Validation	47
5.1	Testing.....	47
5.1.1	Early Client Testing	47
5.1.2	Early Server Testing	47
5.1.3	API Testing	47
5.1.4	User Testing	47

5.2	Demonstration	48
6	Project Plan.....	51
6.1	Initial Proposal	51
6.2	Changes to the Proposal	51
7	Conclusion	52
7.1	Client Application	52
7.2	Server Application	52
7.3	Personal Reflection	52
7.4	Future Work	52
8	Bibliography.....	54

1 Introduction

1.1 Project Overview and Background

Music is a common element to most house parties. Regardless of the occasion, it is likely that some kind of music will be playing in the background. As these events develop, a common sight is various guests either requesting different music or changing the music source completely. Oftentimes, this leads to guests arguing over the music and creating an uncomfortable social environment. Sometimes guests would not be enjoying the music but not feel comfortable addressing it with the host.

This project will attempt to find a software-driven solution to the problems mentioned above. In moving through a full software project development lifecycle it will analyse various elements to parties to uncover what exactly leads to these hostile environments and what party attendants would need in order to prevent them. Based on the feedback from party guests, a mobile application allowing all guests to request and decide on the music being played will be designed, implemented, and tested with the same users.

1.2 Project Objectives

The main objective of this project is to design and develop a cross-platform mobile application with features that assist in a smooth music selection process based on the requirements of surveyed users. These requirements are discovered in detail in the research section, but a high-level description of the project's requirements can be seen below:

- Study party environments to identify what elements cause hostile environments and what elements lead to happy environments and increased guest engagement
- Build a mobile client application that lets users either start a party or join one
- Allow all guests to request songs to be played as well as vote on already requested songs
- Build a server application that handles the creation of these parties, as well as handling song requests and tabulating votes
- Deploy and host the server application

1.3 Project Challenges

1.3.1 Full Stack Development

This application requires a client application, a server application, and means of allowing them to communicate with each other. I have a small amount of experience doing this on a small scale, for example, a library website using PHP and MySQL. However, I have never done this on such a large scale. Research into the most relevant technologies to achieve this must be conducted before development can begin.

1.3.2 Client Application Challenges

To ensure that most guests of a party will be able to request songs, the client application needs to be available on both Android and iOS platforms. With no experience in iOS development, research into hybrid and cross-platform app development technologies must be performed. Once a technology is chosen, its features and methods of operation need to be learned rapidly so that they can be applied to the client application as soon as possible. Development on the client application needs to begin early. A skeleton of the app should be constructed as soon as it is learned how to do so, with the core features being implemented iteratively while delving further into the technology's capabilities. This poses a large

challenge to me because, with the allotted timeframe, learning how to use the selected technology and developing the application will need to happen concurrently.

1.3.3 Server Application Challenges

Building an application that handles and tabulates votes seems like a manageable task. However, the act of building a server with a RESTful API is new to me. Research will need to be conducted into various web framework solutions such as Django and Spring. The chosen technology will need to be learned and understood before work on the API can begin. There is also the challenge of creating the API endpoints and structuring network requests.

This server application will need to be accessible to all clients and, as such, will need to be deployed and hosted on the cloud. This poses a challenge because, while I have had a small experience with cloud computing solution, AWS, I have never deployed anything of this scale. The database will also need to be hosted somewhere, which is new territory for me.

1.4 Structure of the Document

Chapter 2 – Research

This chapter contains the research the conducted into elements of party environments. The identification of the problem to solve is discovered, as well as various reasons for why these problems occur in the first place. Alternate existing solutions to these problems are discovered and critiqued. The projects requirements are identified. Finally, the technologies necessary to achieve these requirements are researched.

Chapter 3 – Design

This chapter introduces the methodologies used when developing this application. The app's main use cases are identified and described. Finally, the app's screens and user interface components are designed in detail with an emphasis on creating a better user experience than the existing alternatives.

Chapter 4 – Architecture & Development

This chapter identifies the system's architect and then discusses how both the client and server application were developed to realize the designs laid out in Chapter 3

Chapter 5 – System Validation

This chapter discusses what testing was done to ensure that the client application, server application, and API were working as intended. It also discusses hosting an event for users to test the application and what insights were gained.

Chapter 6 – Project Plan

This chapter discusses how the project changed from initial proposal to final product.

Chapter 7 – Conclusion

This chapter reflects on what the project achieved, its successes, its failures, what would be done differently, and what can be done moving forward.

Chapter 8 – Bibliography

The bibliography for the sourced referenced throughout the report.

2 Research

2.1 Background Research

2.1.1 Typical Party Environment

A common element of a small house party or social event is music playing in the background. Depending on the occasion, music can be used to set the tone of an event, avoid awkward silences, or invite guests of a party up to dance. Before an event begins, the host usually finds an existing playlist or curates their own with songs they believe that their guests will enjoy. This can be done on a mobile device or computer using either their device's local music library or, more commonly, via a third-party music playing solution such as YouTube or Spotify. The device with the playlist is connected to a speaker, normally via Bluetooth or an auxiliary cable. Once guests begin to arrive, the playlist is started, and the party begins.

While attending many of these events over the past few years, the same problems with this way of music selection presented themselves. There was almost never an event attended in which the playlist went from beginning to end uninterrupted. As the party progressed, guests would begin to adamantly request specific songs to be played. The reasons for this can differ greatly. Sometimes it would be in a positive manner; a guest would hear the song that is currently playing, get reminded of another song that they like, and suggest that it should be played next because both songs go well together. Other times, it was more negative; a guest would not be enjoying any of the songs playing and insist that songs from other genres would be played.

Regardless of the reasoning, many guests would take it upon themselves to change the music. Sometimes they would open the host's device and play their desired song immediately. With software such as Spotify, this is a manageable task as songs can be added to a queue, get played next and return to the original playlist once the queue is empty. However, guests often didn't know of this feature, so the host would have to return to the playlist manually. This means that the songs would get re-shuffled and many songs that already played will be played again. When this happens, it often led to guests being frustrated that songs were repeating and would, in turn, lead more guests to request new songs. Another less frequent but more noticeable issue is when guests would disconnect the host's device from the speaker entirely and connect their own device to play their own playlists. This did not happen as often but would almost always lead to arguments over the music playing and heavily distract from the original purpose of the event.

It was clear that an alternate solution to playing music at a party was needed. Preferably one that allowed all guests to request music without disturbing the flow of the event. Further research was needed to determine what this solution would be and what features it would have. The first iteration of this idea was a mobile or web application that allowed guests to send song requests to the host seemed to be the best fit as this would ensure that the host's device remain untouched and the majority of guests would be able to send requests via their own mobile devices.

2.1.2 DIT Snow Sports Party Case Study

DIT Snow Sports is a popular sporting club within DIT. Up to 150 students attend their annual ski trip abroad. In preparation for the trip, ski and snowboarding lessons are held at the Ski Centre in Sandymount industrial estate every week of the first semester. To celebrate the launch of a new academic year and to welcome new members, a party is hosted within the Ski Centre immediately after the first lessons of the year. Being a member of this club for the previous year, I identified this party as a great opportunity to study in depth how guests react to the music being played.

The party was attended by 35 to 45 students, many of them in their first year and there by themselves. As expected, the party began with pop and chart music playing on one of the club's committee member's phone via a large speaker. This style of music seemed to sit well with the guests; they were upbeat, talkative, and excited to meet new people. Not too long after this, four non-committee members gathered around the speaker and connected one of their own devices to it. They began playing loud, fast, techno music; a very large contrast to the previous style of music playing. With these four men remaining around the speakers, it was impossible for anyone else to change the music.

It was at this point that other members of the party were surveyed about the music. The first attendants surveyed were the club's four committee members. They expressed that none of them liked the music that was playing, and they were not happy that these four people they did not know very well were in control of it. After being asked if they wanted to change the music, the consensus was that they would prefer it was not playing but they did not want to go through the hassle of confronting them to change it, especially because none of the other guests of the party have complained about it or seemed to care. With this response, other guests of the party were surveyed to see what they thought of the music that was playing. The majority of guests surveyed expressed that they did not particularly enjoy the music but did not mind that is was playing. When asked if they would choose to play different music if given the opportunity, the majority of guests said yes.

Later into the evening, the four people by the speaker playing the music left the party. From there, a different device was connected and a playlist containing pop and dance songs began playing. The mood of the party visibly increased once the music changed. Before long, guests began singing along to the music and occasionally even standing up in groups to dance. It was observed that if most guests knew the song that was playing, it allowed for increased interaction between guests.

After attending this event, the need for an application that allows all guests of a party to select songs was solidified. Also, after observing the increased interaction between guests when a song that most guests knew was playing, a feature to allow guests to see what songs have been requested and vote for the ones they like should also be a core feature of this application.

2.2 Alternative Existing Solutions

2.2.1 crowdDJ

crowdDJ is a group music selection application made by Australian live music provider Nightlife (1). The goal of this application is to allow attendees of nightclubs, pubs, restaurants etc, select the songs that play at that venue. The app acts as a modern jukebox. This product was researched to find benefits and aspects to avoid while designing the application.

Critiques

Upon opening this application, the user is brought to a list view showing available venues to join in order of closest to furthest away from the user. This immediately shows two issues to avoid during this project: the user can only join predetermined locations and not host their own event, and users that are not physically present at the venue can still join the rooms successfully. This app was tested in Dublin, Ireland, to which the closest venue was "The Sun & 13 Cantons", a restaurant in London, England. Despite using the application in an entirely different nation, check-in to the venue was successful. This stands out as a design flaw because it opens up the possibility of people not physically present requesting bad songs as a prank. This highlights the need for a feature that restricts access to a party to guests that are present at the event.

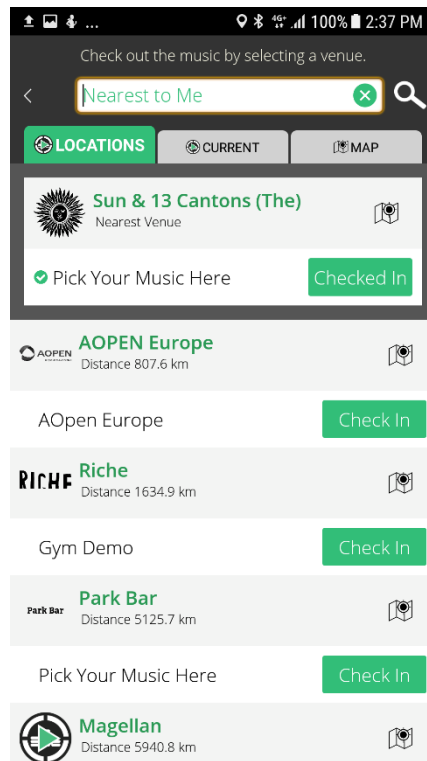


FIGURE 1 - CROWDDJ ALLOWING A USER TO JOIN A ROOM FROM ANYWHERE

Once in the room, the user can add songs to the playlist as they please. After choosing a song, the song gets added to the bottom of the playlist and will be played once the songs above it have finished playing. This highlights a major distinction between crowdDJ and this project. Once a song gets added to a playlist in crowdDJ, it is guaranteed to be played. However, in this project, adding a song to the playlist is not a guarantee that it will be played. It is merely adding the song to the voting pool; it will still need to receive votes from other users before it gets played.

Benefits

Once a venue is selected, the user is brought to a welcome page where they receive “1 free song credit” and are informed that they will earn a new song credit every 15 seconds. This is a valuable idea as it stops the user from flooding the playlist. This could be used in the project as a feature that allows the host to set a certain amount of time to wait before a guest can select another song.



FIGURE 2 - CROWDDJ INTRODUCTION THE IDEA OF CREDITS

After the user clicks “Check In and Continue”, they are brought to the playlist view. A song is currently playing, and a list of upcoming songs is shown. These songs seem to be selected by the application itself as there were no other users in the room. This addresses the problem of what to do if the playlist of user-selected songs has run out. Something similar could be used in the project.

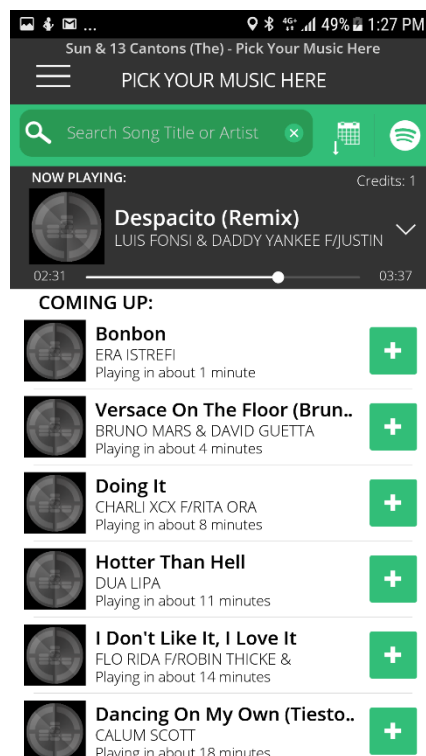


FIGURE 3 - CROWDDJ ADDRESSING THE PROBLEM OF AN EMPTY PLAYLIST

Selecting a song is very intuitive. The user searches for a song title or artist using the search bar the top of the screen and is shown the results. The user then simply taps the '+' icon beside their desired song and it automatically gets added to the playlist. Keeping song requesting as simple and intuitive as possible will be very important for this project. In a party environment, a user's attention will be hard to keep for a long time so it is important to design the UI in such a way that it doesn't require a large amount of thought or focus from the user.

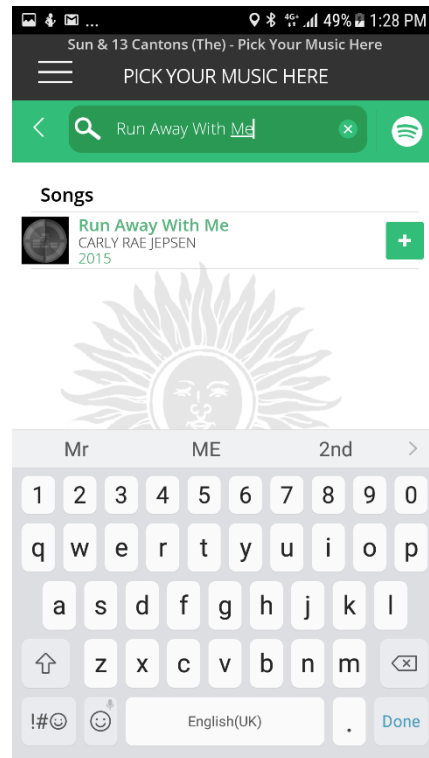


FIGURE 4 -SEARCHING FOR SONGS WITH CROWDDJ

Another feature of the searching system is that certain, seemingly common songs and artists are do not show up in search results. This would appear as though the songs are simply not in the system's library, however, the songs seem to be played via Spotify which has a music library of over 30 million songs (2). It is likely that this venue has certain restrictions on the songs that can and cannot be played. For example, any song by the heavy metal band Slipknot cannot be requested to be played. This makes sense as heavy metal music would not be appropriate at a fine dining restaurant. This would be a valuable feature in the project. A host should have the option to stop certain songs, artists, or genres from being played while setting up the room.

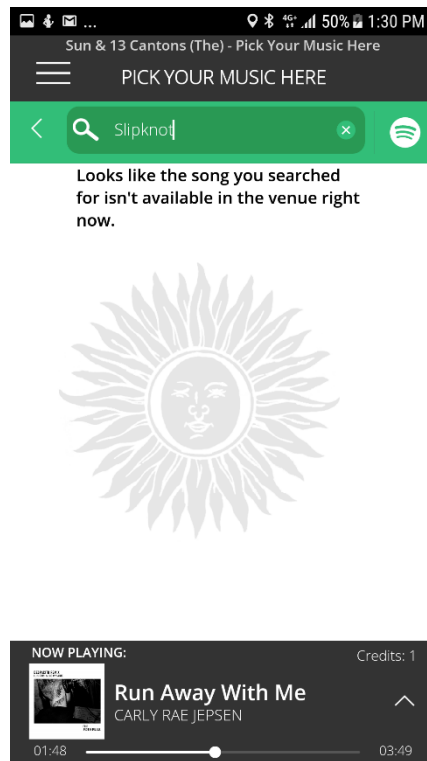


FIGURE 5 - CROWDDJ RESTRICTING CERTAIN SONGS FROM BEING REQUESTED

2.2.2 Jukestar

crowdDJ introduced the idea of multiple users selecting songs to be played at a venue via their mobile devices. However, with no voting for songs or hosting events, the overall premise is still very different from what this project aims to achieve. An example that is closer to the goals of this project is Jukestar, a cross-platform music selection application that allows users host or join rooms and vote on what songs to play (3).

Critiques

At a quick glance, Jukestar is a near perfect example of what this project should be. Anyone who downloads the application (and has a Spotify Premium account) can host a party and anyone else can join, request songs, and vote on what gets played. However, after testing out this app, some problems were identified. This application has an incredibly poor user interface (UI) and user experience (UX).

After setting up a room, the host is brought to the following screen:

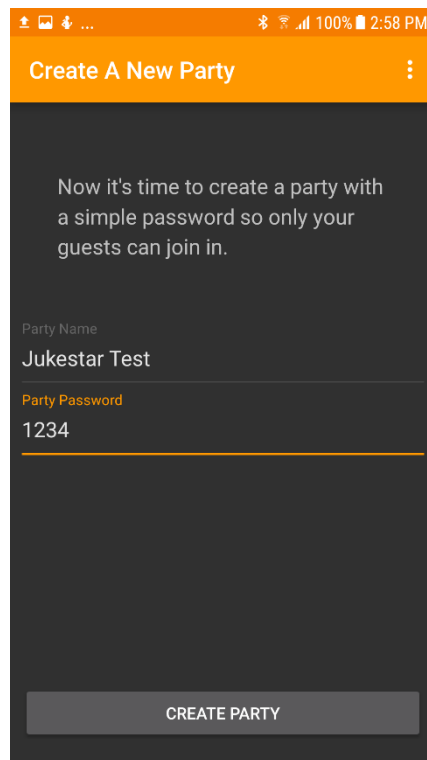


FIGURE 6 - STARTING A PARTY WITH JUKESTAR

This screen contains a lot of information and is not very intuitive. Once a party is started, the next logical step would be to start adding songs to the queue. However, no search bar for song searching or button marked “Add songs” etc. is visible. The host, in fact, cannot select any music via the mobile app. They, instead, must open the guest web application, log into it as a guest, and select songs from there. The mobile application seems to only ever be used to create a room and start/stop the music which can be very confusing to new users.

From the web app, guests join a party and get brought to the party screen. The user experience increases slightly as opposed to the mobile app. Tabs for Party, Request, and History are visible at the bottom of the screen and a user can intuit that these tabs are for seeing what is playing at the party, requesting songs, and seeing the history of songs that have already played respectively. While it does not look incredibly aesthetically pleasing, it gives the user all the information they need.

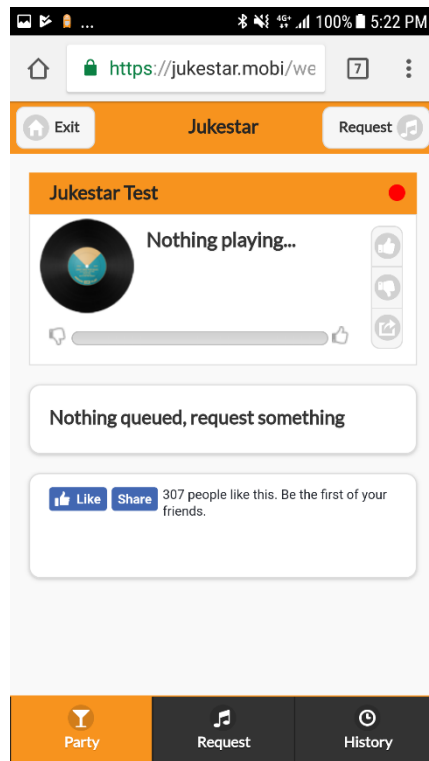


FIGURE 7 - JUKESTAR'S WEBAPP WITH AN EMPTY MUSIC QUEUE

Requesting a song is more complicated than it needs to be. A user searches for a song by typing the name of an artist or song into the search bar at the top of the page, similar to crowdDJ. The search returns a list of songs. If the user wants to request one, they must tap the small arrow at the side of the list entry which opens a pop-up menu from which the user can select the “Request Song” button. This is not good design and should be avoided in the project. A user should only have to click one button to request a song.

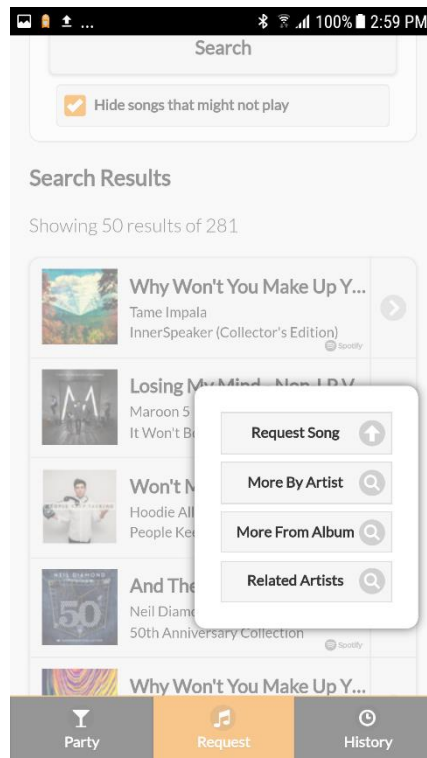


FIGURE 8 - JUKESTAR'S COMPLICATED VOTING METHOD

Benefits

Because of the web application, guests of a party can select and vote on music without ever needing to download the mobile app. This is an interesting feature that was not considered during the conception of the project. While song requesting and voting via the mobile application will remain as core features in this project, a web application alternative will also be explored.

2.3 Music Sources Researched

2.3.1 Locally Stored Music

If a host is to allow guests to request songs to play via the application, the application needs a library of music from which the guests can select. This first music source considered for this purpose was the host's device's native music library. Most smartphones contain the ability to store and play music locally. However, it was quickly discovered that this would not be an optimal solution for this application because there is no guarantee that the host has any music on their phone. Even if the host does have music stored locally, there is still a chance that the collection of songs is limited and that guests would like to hear something else. Because of this, other music sources need to be considered.

2.3.2 YouTube

YouTube is a popular video streaming website. While the site hosts videos of any kind, a common usage of the site is streaming music videos. Most popular songs come with a music video which gets hosted on YouTube. YouTube also has a playlist feature, meaning that videos can be organized and played one after another. Oftentimes, at parties, the music will be coming from a YouTube playlist instead of a solely audio-based alternative. Because of YouTube's popularity and the fact that it's free to all users, it was considered as the music source for the app but was decided against because there's no strict naming convention between videos. It would be beyond the scope of this project to read a video's title

and abstract the song's title and artist name. Also, because not all videos on YouTube are songs, it would be undesirable to allow guests to play non-musical audio at the party.

2.3.3 Spotify

Spotify is a music streaming service that gives users the ability to stream songs on computers, mobile devices, and tablets etc (2). It is available in 61 countries and has over 140 million active users. With a library of over 30 million songs, it is likely that a typical user would be able to find most of the songs that they would like to play. Spotify addresses some of the concerns with using YouTube as the music source: all songs follow the same naming convention, and only music can be played.

Spotify is a "freemium" service, meaning that most of the app's functionality is free to users with additional features available for an additional fee (4). In Spotify's case, the mobile application allows free-tier users to create and organize playlists and play songs but limits the songs to only be played from a playlist in shuffle mode. Songs cannot be played directly. For example, if a user wants to play an album, they cannot play it from first song to last and must, instead, listen the album's songs in a random order. To gain the ability to play specific songs on demand or in a desired order, users must pay to upgrade to the premium level. For the core functionality of this project's application to be achieved, songs must be played directly. Because of this, if Spotify is to be used as the app's music source, the host of a party would need to have a premium Spotify account, as the songs will be played through their device. This is undesirable, as it means that a user would need to be a paying subscriber of another application if they wanted to host a party.

Spotify has numerous developer tools and resources including a large Web API (5) and software development kits (SDKs) for both Android (6) and iOS (7). An SDK for both iOS and Android means that making a hybrid application with Spotify should be possible. It also has a Web API Console that interactively shows developers how to structure various API calls and the structure of the data returned. The availability of these tools and the in-depth documentation behind it, along with the large user-base and massive music library makes Spotify a fruitful music source for this application.

While experimenting with requests on the Spotify Web API Console, it was discovered that, in order to make requests, a valid OAuth token must be supplied. This means that every user of the application must have a Spotify account. Guests should connect with Spotify from within the application so that requests can be sent to the Web API successfully. While users need a premium Spotify account to be able to host parties, standard Spotify users will still be able to join parties and make requests.

2.4 Other Relevant Research

2.4.1 Room Creation

Upon opening the app, users will need to navigate to the correct party before they can start requesting songs. Different methods of how a user could achieve this were researched.

Manual passcode entry

The first solution observed was manual searching and passcode entry. In Jukestar, users join a party by searching for the party's name and clicking it. From there they are asked to type in the party's password. This meets the requirements of this feature but requires a lot of user interaction and data input.

Automated passcode generation

It would be better if the application could come up with a unique room identifier instead. An application that does this is Jackbox. Jackbox Games is a video game development studio responsible for popular games such as Quplash, Drawful, and Fibbage (8). The premise for all of their games is that prompts or questions will show up on a large screen like a television or laptop, and guests answer the prompts via their smartphones. For example, in Quplash, a sentence will appear on screen with a missing word or phrase and players will type in the funniest thing they can think of to complete the sentence. In order

to join any of these games, players navigate to the URL <http://www.jackbox.tv> on the browser of their smartphone and enter in the four-letter room code they see onscreen. Once they enter the unique room code, they are brought into the game with no further verification needed.



FIGURE 9 - A JACKBOX GAME WITH THE GAME'S ACCESS CODE IN THE CORNER

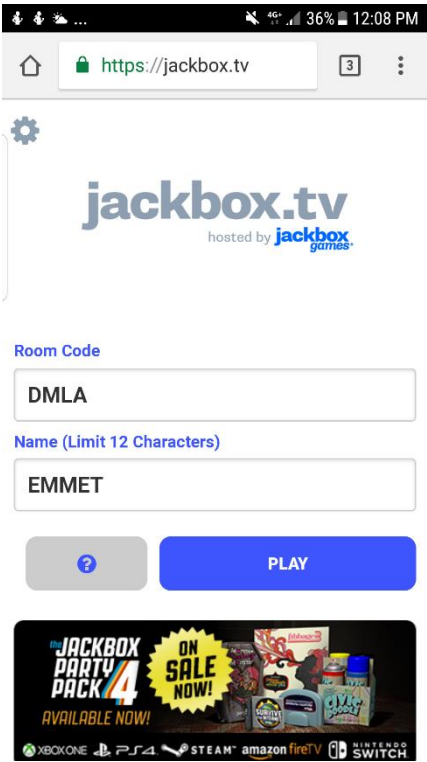


FIGURE 10 - JOINING A GAME ROOM WITH JACKBOX

A means of generating unique passcodes for each party seems like a much better way of controlling how users join parties and will be considered when designing the app.

2.4.2 Voting

While testing Jukestar, users expressed that it took a lot of focus to vote for song. With mobile devices being so small, and with party environments being so busy, it is important to find a way to allow users to vote on tracks without needing to focus very hard. A popular existing mobile application that involves users voting on things is Tinder.

Tinder is a dating application that allows a user to see other users nearby and decide if they like them or not (9). If both users like each other, a chat dialog between the two users is opened up and they can get to know each other. The user is shown other users one at a time.

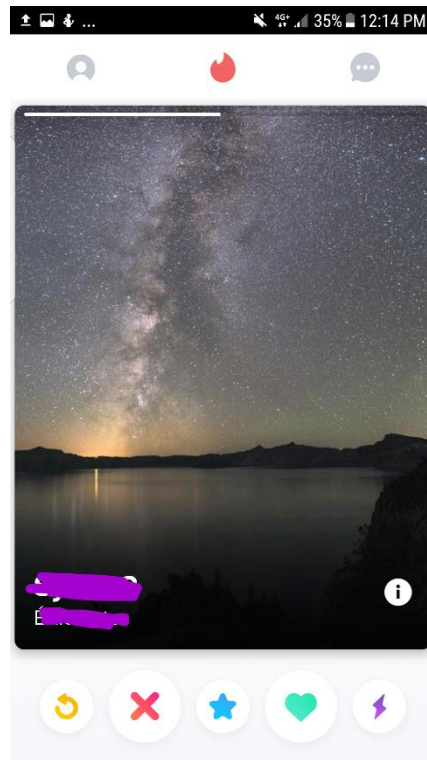


FIGURE 11 - TINDER DISPLAYING USERS ONE AT A TIME

They can then “like” the user by either tapping the green heart or swiping the image to the right. “Disliking” can be achieved by tapping the red X or by swiping left.

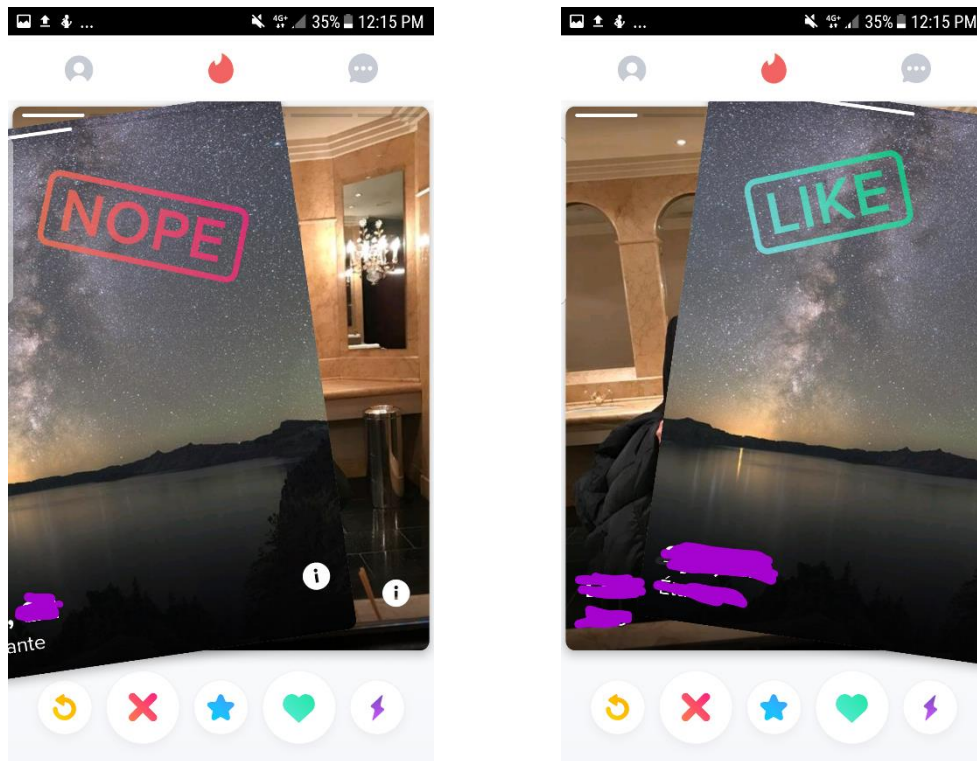


FIGURE 12 - VOTING FOR OR AGAINST WITH TINDER

Tinder's user interface is very intuitive and easy to use. It also makes the act of voting very fun. Studies have been made into why Tinder is so popular and part of the reasoning why is that the act of swiping makes the voting process feel like a game (10). A Tinder style voting system for songs could be a good addition to this app. Users will be able to swipe left and right with ease while devoting too much attention to the app.

2.5 Resultant Findings and Requirements Definition

2.5.1 Feature Selection

From the above research, various elements of how music gets played at parties that lead to arguments over the songs have been discovered. Also observed, were elements of parties that lead to further interaction between guests and happier environments. These elements have been considered and a list of features for application to help achieve a smoother music playing experience at parties have been identified.

This application should be mobile app and should allow users to:

- Connect with Spotify
- Start a party
- Join a party
- Request songs
- Upvote songs
- Downvote songs

Each party should have a playlist, comprising of the requested songs of all guests. Guests should be able to see the songs in the playlist and, for each song, be able to upvote the song if they want it to be played and downvote the song if they want it not to be played. Songs should be played through the hosts device only. The list of songs in the playlist should be organized in such a way that the song with the

highest number of votes is first in the list, second highest number of votes second in the list and so on. When a song is finished playing, the top song in the playlist should be played next.

2.5.2 Requirements

Client Application

The client application's main purpose is to allow hosts of a party create a space that their guests can join and allow guests request and vote for songs to be played. Party environments tend to be very busy with many of the guests talking to each other and socializing. To keep this environment as intended, the client application should not distract the user from the party and, as such, users of the app should not have to focus on the app incredibly hard to be able to use it. The UI should be incredibly simple and intuitive. As typing on a phone is a heavy distraction from conversations, data entry should be kept to a minimum. Users should only have to type something into the app when joining a party or searching for a song. Even at that, searching for a song should be quick and easy; a search request should be automatically sent to the Spotify Web API whenever a new character is entered into the search bar. Users should see new results after every character they type and hopefully find the song they are looking for before they finish typing.

Server Application

The primary purpose of the server application is to make network requests to the client application. It will accept requests from users to create or join a party and send party data back to the client. The server application will also be responsible for the tabulation of votes for all track in a party's playlist. It should arrange the list of songs in order of highest number of votes to lowest. The server will make this data available through a RESTful API.

2.6 Server Application Technologies Researched

2.6.1 Django

Django is a web-framework built with Python (11). It is primarily used in the building of web applications. It is based around its Model Template View (MTV) framework which is an extension of Model View Controller (MVC) architecture. Models are constructed as Python classes from which database tables can be made using Django's object-relational mapper (ORM); each attribute of a model class corresponds to a column of a database table (12). The class's fields can contain arguments to specify what each column can and cannot contain, for example what data type it is and if it can be null. Using Django for the server application means that the functionality for building the database and the functionality for creating parties and tabulating votes etc. can be done in the same place using a simple, familiar, high-level, and powerful object-oriented language.

Django is open source and, as such, has many community-made additional libraries available. One library that this project will utilize is Django REST Framework, a tool used for building RESTful APIs out of the models made in Django (13). Using this toolkit, data from the models can be serialized in JSON objects that be sent to the client application. Various URL endpoints can be constructed for the client application to call to get the serialized data.

2.6.2 Deployment

Database

Both the server application and the database will need to be deployed somewhere that is accessible to all client applications. Amazon Web Services (AWS) (14), a popular cloud computing platform was the first solution researched. While researching, it was discovered that AWS Relational Database Service (RDS) fulfilled the need for hosting the PostgreSQL database. It had a free tier, was simple to set up, and was able to be integrated into the server applications settings relatively easily. Because of all this, no other solutions needed to be researched.

Server Application

In order to deploy the server application successfully, it was discovered that it needed to be containerised with all its necessary requirements. Docker stood out as the best solution for this (15). Docker describes itself as a “platform for developers to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud” (15). Docker was considered from the beginning as Django applications come with a Dockerfile. From this file, the instructions for how to build the image and its requirements are read.

AWS Elastic Cloud Computing (EC2) was considered as the tool to host the server application. Setting this up proved unintuitive and difficult so other methods were researched. Digital Ocean is another cloud computing service provider (16). They provide a similar solution the AWS EC2, Droplets. Droplets proved to have easier integration with Docker images and was chosen as the servers hosting solutions as a result.

2.7 Client Application Technologies Researched

2.7.1 Native Android Application

With Spotify as the selected music source, some work needed to be done to ensure that the SDKs could be used to make a mobile application that allows songs to be played on the device via Spotify. To get accustomed with the SDKs before moving onto hybrid solutions, Spotify’s tutorial for a native Android application was followed and recreated (17). This tutorial walks the developer through installing the Android SDK, making use of Spotify’s playback and authentication libraries. It also informs the reader that it is necessary to register the application with Spotify to obtain a Client ID. This is a necessary step to get allow the application to play songs.

After some configuration, the tutorial was followed and completed. The end result was an application that opened a Spotify login dialog and, on successful login, navigated to a screen that played a single song that was hardcoded in the program using Spotify’s playURI method. This tutorial introduced the concept of registering an application with Spotify and some examples of Spotify methods. It also served as a proof of concept, showing that making an app that plays songs via Spotify is an achievable goal.

2.7.2 Cordova

Android makes up for about 85% of the global smartphone market. Apple is second with 14.7% with various other operating systems making up the final 0.3% (18). However, in Ireland, the market is a lot more competitive with 55% of smartphone users using Android devices and 42% using iPhones (19). This means that iPhone cannot be overlooked during this project if it is to be deployed and user-tested. With the time allotted for this project, it seemed unrealistic to design and implement two separate native apps in their respective languages on time. A native Android app would need to be made with Java and XML in Android Studio. A native iPhone app would need to be made with Swift in XCode. With a limited amount of time, and no previous experience in iPhone technologies, various cross-platform technologies needed to be researched so that both the iPhone and Android applications could be developed at the same time.

The standout solution to this problem was Cordova. Apache Cordova describes itself as an “open-source mobile development framework” (20). It allows developers to write their application as a web page, using standard web technologies such as HTML5, CSS3, and Javascript and renders it into a WebView which is what the mobile application will run. A benefit of using Cordova was that the application could be designed using the Spotify Web API instead of the mobile SDKs and be rendered as a WebView for both iOS and Android. Because apps are developed as web pages, this opened up the opportunity to build the application using popular advanced web technologies such as Angular or React (21), as well as the possibility for users to host events through their laptops instead of through their phones.

To research how Cordova works, a toy app was made using HTML, CSS, and Javascript and was pulled into an Android app using Cordova. Because this app ran successfully and was built relatively quickly, a sprint began with the goal of recreating the native Android tutorial using the Web API instead of the Android SDK. The logic behind this sprint being, if a web app can be made to played music via Spotify, this app could be pulled into both an iOS and Android application and a hybrid app could be achieved. Unfortunately, early into this sprint, it became clear that Web API does not allow the playing of songs and is instead intended as a tool for the development of analytical apps, playlist organizers etc. Because of this, Cordova was not an eligible candidate for making a hybrid app that plays music through Spotify.

2.7.3 React Native

If this application is going to be a hybrid app, a tool that lets developers design an app for both platforms in one place while also giving the developer the option to edit the native code of each platform is needed. React Native was found as a result of researching these requirements. React Native is a tool built by Facebook that allows developers to design and implement mobile applications for Android, iOS, and Windows using Javascript (22). It makes use of React, a Javascript library also built by Facebook that is used for designing powerful user interfaces. A distinction between React Native and Cordova is that React Native builds actual native applications as opposed to Cordova's solution of an app that opens a WebView showing a web app. When built, an Android app made using React Native would be indistinguishable from an Android app built with Java. With actual native applications being built, the Spotify Android and iOS SDKs can be used. React Native will be used when developing this application.

While researching React Native and seeing if it could work with the Spotify SDKs, a Node module, react-native-spotify (23) was found. This is a library that contains the relevant code for adding both the Spotify Android SDK and iOS SDK to a React Native project. Android applications are built using Java while iOS applications are built using Swift. Because of this, the commands used to play songs via Spotify etc are different for both platforms. This library allows the developer to write these commands in Javascript. When the application is built, the command will be converted into the relevant language. This library will be used while developing the client application.

3 Design

3.1 Methodologies

3.1.1 Waterfall

The first software design process considered was the Waterfall model (24). This process is plan-driven, in that all activities are planned and scheduled before any development begins. It adheres to the following stages:

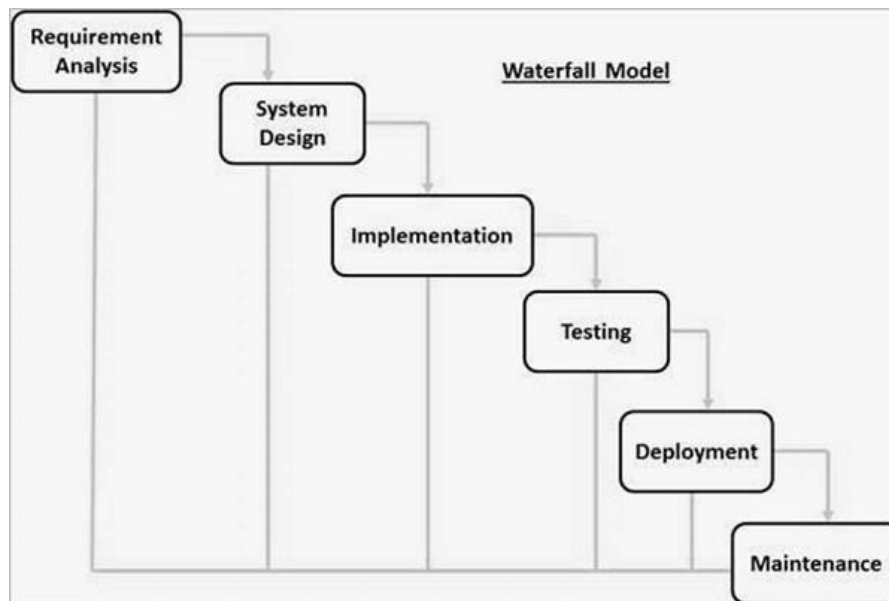


FIGURE 13 -WATERFALL MODEL (25)

Typically, with Waterfall, each phase must be completed and thoroughly documented before the next phase can begin. This means that a lot of the time at the beginning of the project will be spent researching and designing before any implementing can begin. This could be followed as the feature requirements have already been listing. However, during the implementation phase, problems with the design may found and new feature requirements could be discovered. With a strict deadline for this project in place, returning to the requirements phase and starting the project over will not be possible. Because of this, a linear model such as Waterfall is not suitable for this project and some agile methods will be considered instead.

3.1.2 Scrum

Most software products and applications are developed using agile methodologies (24). According to the agile manifesto, agile development values *“working software over comprehensive documentation”* and *“responding to change over following a plan”*. With agile, a code can be written early, working software can be shown quickly to users, and goals for the project can change easily. One agile method that has stood out is Scrum.

Scrum Framework

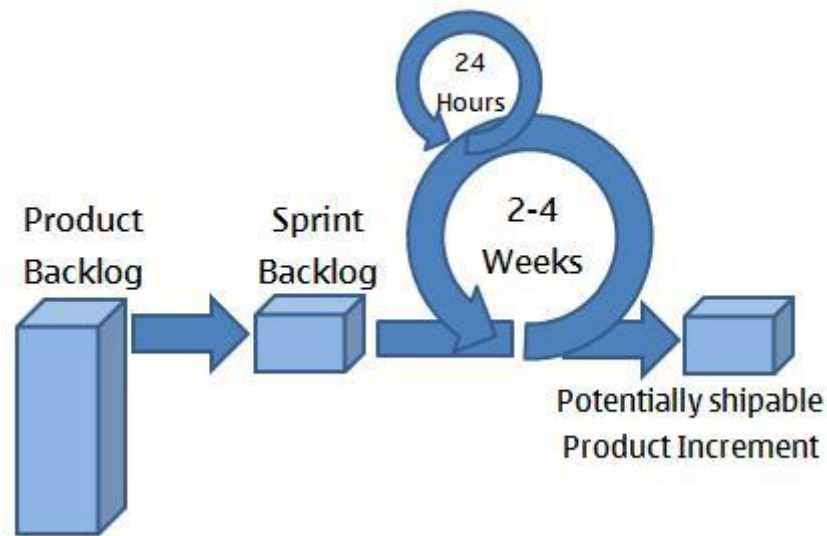


FIGURE 14 - SCRUM LIFECYCLE (26)

Scrum is the most widely used agile methodology in industry (24). It is based around a concept that has been mentioned a few times in the research section: sprints. A Scrum sprint-cycle begins with a product backlog. This is the list of features, requirements, etc. that need to be worked on in order for the product to be functional and ready for shipping. From the product backlog, sprints are planned. Sprints are small periods of development with a goal of reaching the desired functionality by the end of the period. In industry, sprints normally last between two and four weeks but, for this project, sprints should normally last one week and never last more than two weeks. Scrum allows the developer to focus on one feature or part of a feature at a time. It also frequently produces deliverables which can be presented and tested by users. This allows for rapid changing of requirements or goals depending on the feedback of the users. With a features list already defined that can act as the product backlog, and the possibility of new features being added later, Scrum will be used as the methodology for this project.

3.2 Use Cases and Functionality Overview

3.2.1 Use Case Diagram

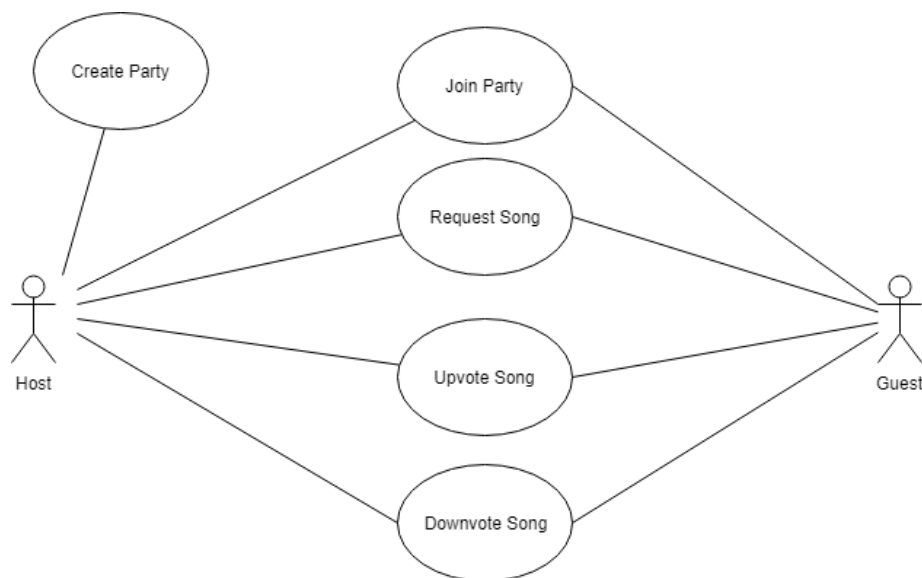


FIGURE 15 - USE CASE DIAGRAM

The figure above shows the core use cases for the application. It covers essentially every interaction with the app that a user can have.

Create Party

Provided they have a Spotify premium account, a user will be able to create their own party room that guests can join. In creating a party, the user takes on the role of the host.

Join Party

All users will have the ability to join a host's party provided they enter the party's identifying passcode. All non-host users in a party take on the role of the guest.

Request Song

Both hosts and guests will have the ability to search Spotify's music library for a song and add it to the party's playlist.

Upvote Song and Downvote Song

Hosts and guests will be able to see the party's playlist which is made up of every song requested by a user. They will be able to upvote the song if they want the song to be played and downvote it if they do not.

3.3 User Interface and User Experience

When developing an application that is so user-focused, user experience (UX) becomes incredibly important (27). In a party environment, users will have their attention split between many different things and, as such, it is unlikely to expect a user to give the application their undivided attention. The app needs to feed information to the user in such a way that does not require much thinking or decision making from them. One of the main criticisms observed while user-testing Jukestar was that too much information was being fed to the user at once which meant that the app required a lot of focus and attention from the user. This caused users to be distracted from the party and, in turn, caused the exact problem that the application was initially trying to solve. Users would spend too much time figuring out

how to navigate the application and little to no time enjoying the music that was requested. These observations showed that ease of use was of utmost importance while designing the client application. The criticisms of Jukestar were kept in mind while designing each screen and the navigation between them.

3.3.1 Choosing a Name

To build a reliable user base, an application like this needs to be recognizable, memorable, and easy to use. With user recognition in mind, some time was spent coming up with a name and a logo for the application. With this app being so intrinsically related to Spotify, some names with the “ify” suffix were considered. However, the name chosen for the app ended up being “Boppable”. Bop is a slang term for a good song and “able” is common suffix among web and mobile applications.

3.3.2 Style

If the host of a party decides to use Boppable as the method of playing music, it is likely that they will have to explain to most of their guests what the app is and what it does. This means that most Boppable users will have their first experience with the app while already attending a party. With this in mind, it is important not to give the user more information than necessary. To get used to the app quickly, it needs to feel familiar and not like an entirely new thing.

In order to use this app, the user must have a Spotify account. So, it can be assumed that the user has at least some experience with Spotify. Because of this, the app should be styled to look very similar to Spotify: the same colour scheme etc. Navigation bars will be coloured with the recognizable Spotify green, song components will be dark like Spotify’s one and use the same hex code values for the colour. This should give the app a familiar feel and leave the user comfortable knowing that this is a Spotify app and something that they are confident in using.

3.3.3 Navigation

Navigation between screens should be kept as simple as possible. Users will spend most of their time using the app while inside a party. The three screens that are available to users while in a party are the Party screen, the Request screen and the Vote screen. These screens shall be grouped together in a tab navigator. Users should be able to swipe between the screens. This keeps them readily available to the user and much more easily accessible than other navigation alternatives such as a sliding drawer sub menu on the side of the screen. Also, this addresses a concern with hybrid app design. User like apps that feel native (28). Using a navigation system that is common to both iOS and Android like tabs and swiping provides a familiar UI for both sets of users. The active tab should have a white indicator below it to show the user where they are.



FIGURE 16 - TAB BAR DESIGN

Navigation needs to be slightly different for host and guests respectively. Hosts need to be able to pause and play the music or stop the party entirely. Guests should not have these options but should have the option to leave the party and return to the Join Party screen. A header navigation bar needs to be designed for both scenarios.

The host’s header will have a button with a cross on it indicating that that is how to close the party entirely. Far on the other side of the navbar, a play/pause button will be there to allow the host to toggle the music’s playstate. The party’s room code will be placed between these buttons. This will keep both buttons separate and also allow the party’s room code to be visible at all times.

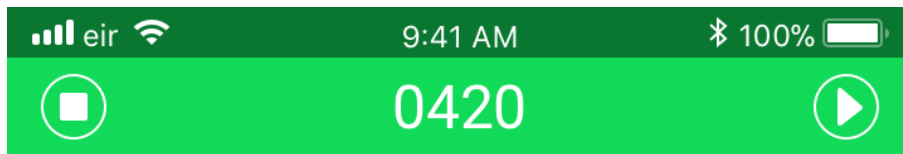


FIGURE 17 - HOST HEADER DESIGN

The guest's header will also contain the party's room code to keep them informed about what room they are in. It will also have a button with left-facing arrow indicating to the guest that that is what to click in order to leave the party and join a different one. To fill in empty space, the app's logo will be shown between these two elements.

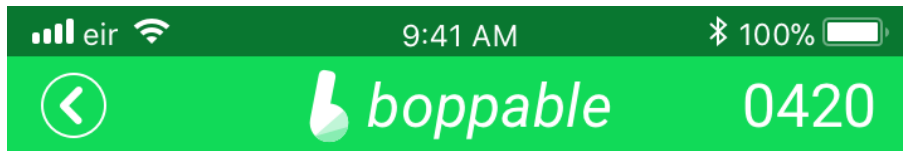


FIGURE 18 - GUEST HEADER DESIGN

3.3.4 Screens

Initial Menu

As identified earlier, Spotify requires a user's OAuth token in order to make certain requests to its API. This means that all users of Boppable, hosts and guests alike, will need to be logged into their respective Spotify account in order to use the app. With this in mind, the first thing any user should see upon opening the app is a single button prompting them to connect with their Spotify account. Anything else would merely be a distraction for the user. Clicking this button will open a Spotify login dialog that comes with the iOS and Android Spotify SDKs. If the user is already logged into Spotify on their device, they can click a button to connect their account with Boppable. If not, the user will see the option to manually log into Spotify instead.

Upon logging into Spotify, the client application will make a call to the Spotify Web API to get the user's information. This will return a JSON object containing information about the user. From this, the app can get the user's name instead of having the user enter their name manually. The ease of use of this screen is a large improvement on Jukestar's initial screen, shown side by side in the figure below.

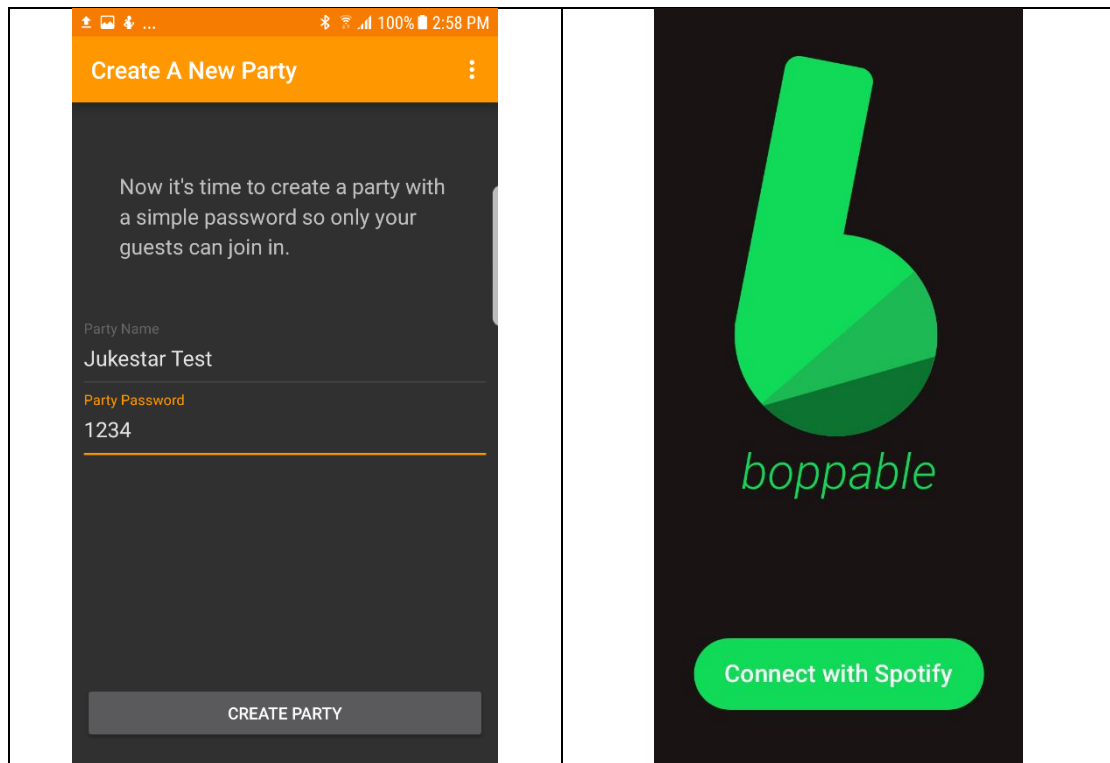


FIGURE 19 - INITIAL SCREEN COMPARISON

The app will also use the user information object to determine if the user has a premium or standard Spotify membership before navigating to the next screen. Streaming specific songs on demand is a feature only available to Spotify Premium account holders, which means that these are the only type of users that will be able to host parties. Because of this, only premium Spotify users should ever see the option to host a party. Upon successful login to Spotify, the app should navigate to the Premium Menu screen if they have a premium membership. If not, it should navigate to the Join a party screen.

Premium Menu

If not currently in a party, premium members will have the option to either start a party or join one. With these being the only two options, the user should not see anything other than two buttons which will direct them to either screen.

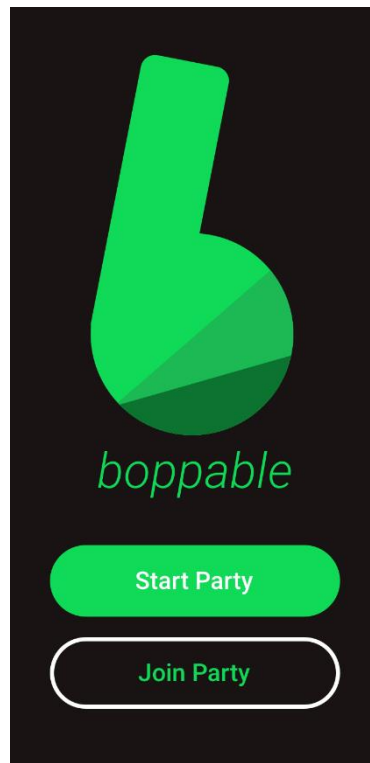


FIGURE 20 - PREMIUM MENU DESIGN

Start Party

In Jukestar, starting a party requires the user to give the party a name and set a password. This means that the user needs to manually type in two different strings of text before their party can begin. It also means that guests have to search for the party's name, click it, and then type in the correct password before being able to join the party. This is a lot of unnecessary work, especially in a party environment. With Boppable, each party will automatically unique passcode upon creation, similar to starting a game in Jackbox. This passcode will serve as the both the room identifier and password. Because the passcode gets generated automatically, when a user clicks "Start Party", there will be no need for any other input from the user and they will be brought straight to the Party screen.

Join Party

When joining a party, the user is required to enter the party's passcode, similar to what was observed when researching Jackbox. A difference between Jackbox and Boppable is that with Jackbox, three to eight people have decided to play a game together and that game has their undivided attention. With Boppable, however, the app will be used in parties and should not expect the user to focus on it for too long. To keep the user from focusing too hard while joining a party, the passcode should be four digits as opposed to Jackbox's four letter codes. This will allow the app to use a numerical keyboard instead of the standard alphabetical one. With less buttons on screen, data entry is kept as simple as possible and the user can join a party with ease.

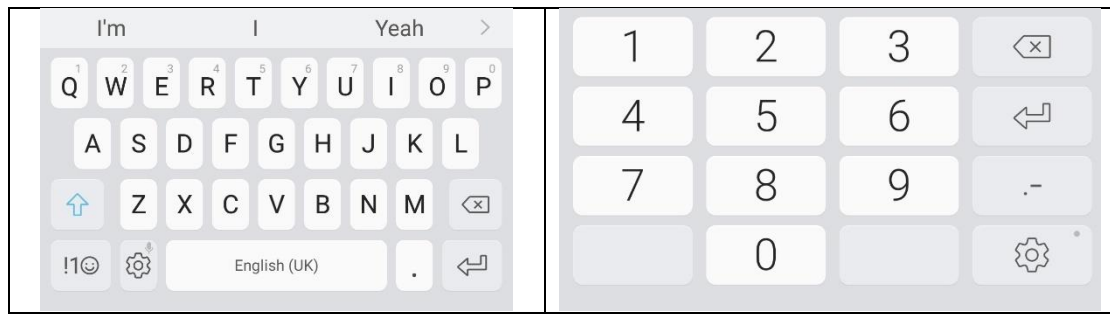


FIGURE 21 - KEYBOARD COMPARISON

Party

The party screen should be the first screen a user sees upon successfully joining a party. It consists of two main elements: the player, and the request list.

The player's goal is to show the user the song that is currently playing and who requested it. Following the criticism of Jukestar's player component, various decisions need to be made when designing this component. Jukestar's player component has five distinct elements: an image of the song's album art, some text showing the song's name and artist, some text showing the user who requested the song, a horizontal bar showing the state of how many votes the song has, and a vertical icon button menu allowing users to upvote or downvote the track or open a sub-menu showing more options relating to the song. This small component often overwhelmed users during testing due to how busy it is. For example, the song title font, the artist's name font, and the requesting user font are all different sizes which makes it difficult to read when they are so close together. Also, the bar showing vote progress and the buttons menu heavily distract from the information that shows what song is playing.



FIGURE 22 - JUKESTAR'S BUSY PLAYER UI

With the user's limited attention in mind, Boppable's player needs to be much simpler. For starters, the player will not contain any buttons or touchables. As discussed earlier, Boppable will not allow users to vote on a song if it is already playing, so there is no need for vote buttons or a vote progress bar. This leaves us with the album art image, song title, artist name, and suggester text. From user-testing Jukestar, it was discovered that most users would have preferred that the album art was bigger so that it could be more recognizable. Because of this, the cover art image should take up the whole left half of the player. The right half will contain the remaining text components. For readability purposes, the song title and artist name text components should be grouped together and placed far away from the suggester text component. The song title should be large and bold as it is the most important text component. The artist text should be the same font size as the song title, but it should not be bold. This will distinguish the two elements but also inform the user that they are grouped together and related to one another. The suggester text should have a much smaller font size so that it's not to be confused with the song details.

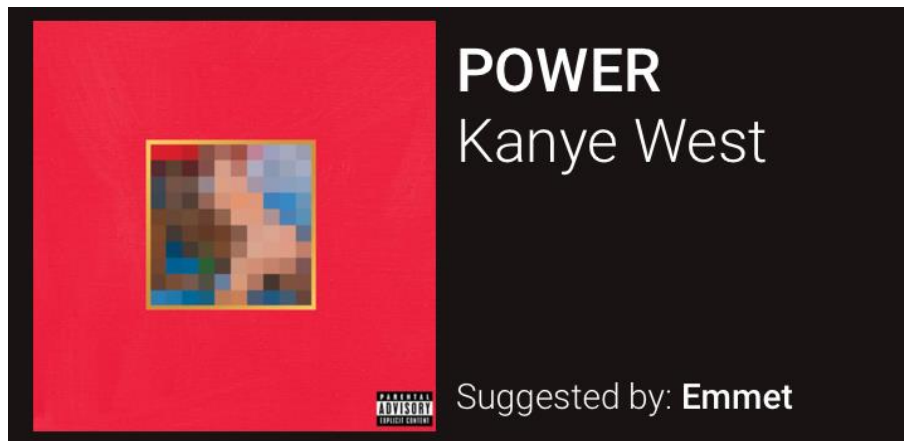


FIGURE 23 - BOPPABLE'S SIMPLER PLAYER DESIGN

The request list's goal is to show the user the list of songs that all guests requested in order of most popular to least popular. The list should be a list of request components: one for each song request. These components aim to do three things:

1. Let the user know the song that has been requested
2. Let the user know how many votes the song has
3. Give the user a means of voting the song up or down

Jukestar's version of a request component achieves these three goals but does so in a way caused confusion among guests. At a quick glance, the first two requirements are easy to spot. Cover art, song titles, and the artist are clearly visible. The component for the number of votes is also visible, however, it is very small. Also, the difference in colour behind the number makes it look like it's a touchable component despite it not being so. The third requirement, voting, is undesirable. Users must click the small arrow icon on the far right of the component which opens a small sub-menu. From there, the user can choose "Up Vote" or "Veto". While testing, users expressed their frustration with having to go through so many steps to upvote a song and stated that voting buttons directly on the request component would be preferable.

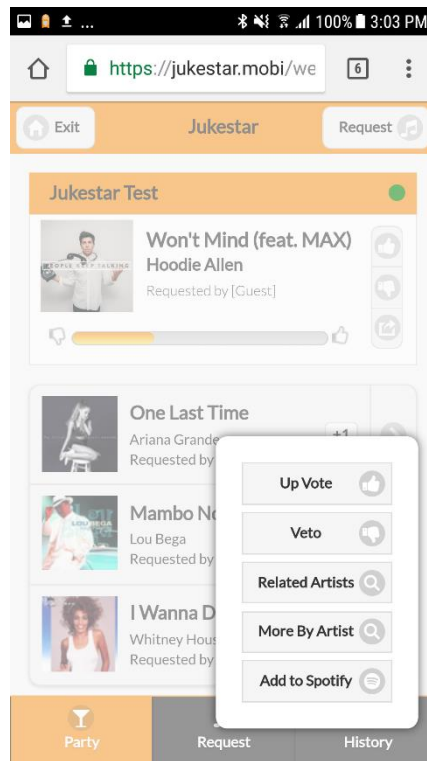


FIGURE 24 - VOTING FOR A SONG WITH JUKESTAR

As a result of this feedback, Boppable's request component will have an upvote button on one end of the component, a downvote button on the opposite side, and the remaining song details in between. This will keep both buttons as far away from each other so accidental clicks can be avoided. Because there is already a lot of text in the component, these buttons should just contain coloured arrow icons: red for down, green for up. The text components should follow the same style as the player component.

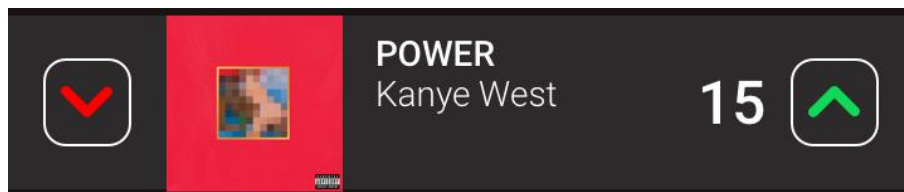


FIGURE 25 - BOPPABLE REQUEST COMPONENT DESIGN

Users should not be able to upvote or downvote a song more than once. To let users know that their vote went through, the vote icon should change colour. If a vote button is pressed while coloured, the vote will be undone and the button returns to normal.

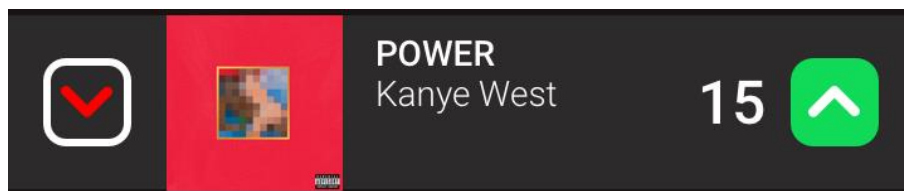


FIGURE 26 - UPVOTED REQUEST COMPONENT DESIGN

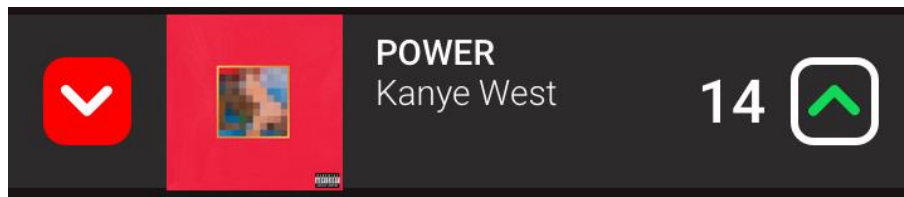


FIGURE 27 - A DOWNVOTED REQUEST COMPONENT DESIGN

The overall design for the Party screen will look like this:

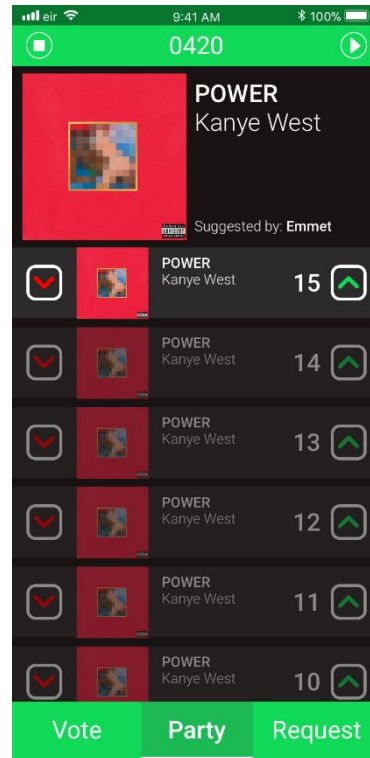


FIGURE 28 - PARTY SCREEN DESIGN

Request

The request screen is where users can search for songs on Spotify and add them to the party playlist. It contains two main components: the search bar and the result list.

To search for a song in Jukestar, the user must type in their search query in full and then press the “Search” button. This is a rather old way of searching. React Native allows developers to automatically send a new search request as soon as a new character is added or taken away from the search bar. This means that the user will get results as soon as they start typing, which is a lot easier on the user than manually retyping and sending new requests. As well as this, the search bar should be styled in such a way that makes the text large and visible so users know exactly where to type.

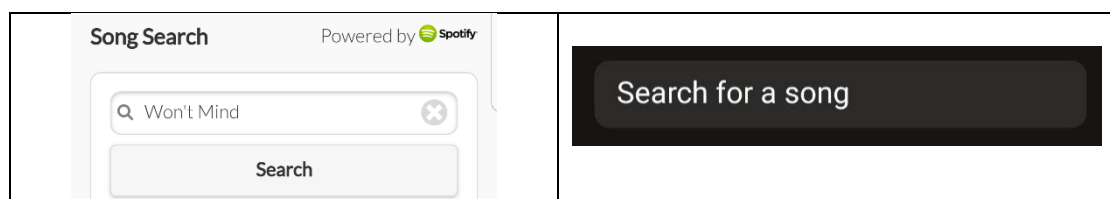


FIGURE 29 - SEARCH BAR DESIGN COMPARISON

The result list will show the user the most relevant songs based on their search query. Each song will be contained in request component. Similar to the request component in the player, this component will be a simplified version of its Jukestar counterpart. It should look essentially the same as the party's

request component except without the voting buttons and the number of votes component. The component should also avoid Jukestar’s sub menu solution for adding the song to the playlist. Instead, a button should be on the right side of the component that fulfils this requirement. To keep in theme with the name of the app, adding a song to the playlist will be called “bop”.

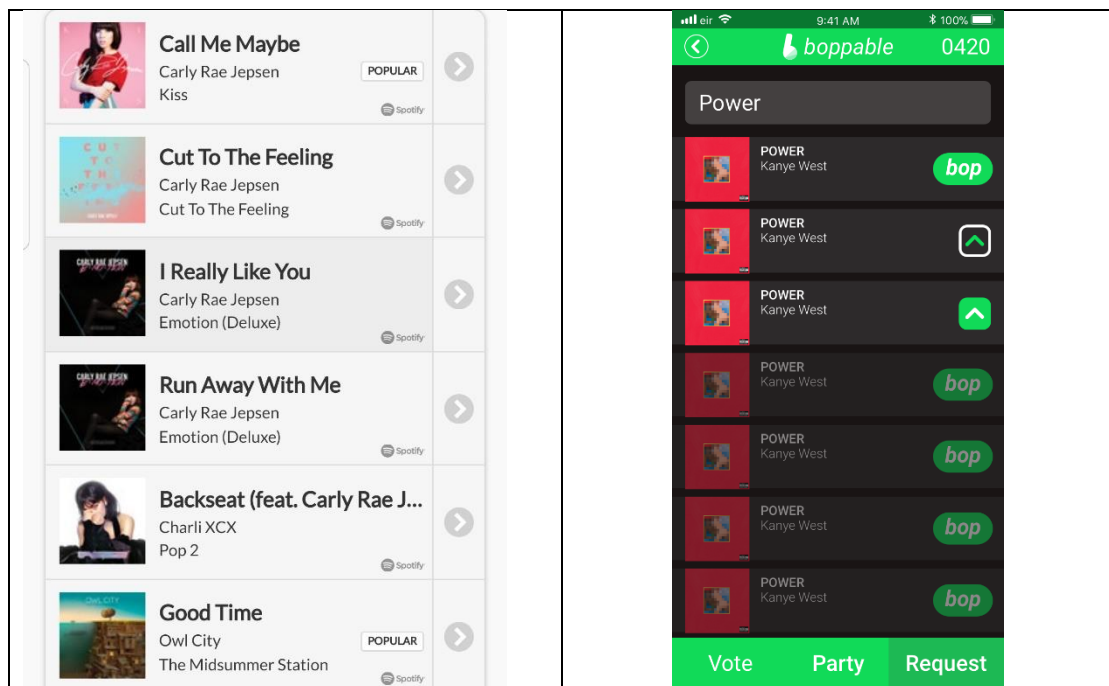


FIGURE 30 - SEARCH SCREEN COMPARISON

A user should not be able to request a song if it is already in the playlist. To combat this, when the result list populates, the component will make a call to the server application to see if the track is already in the playlist. If it already exists, instead of a “bop” button, the component should have an upvote button.

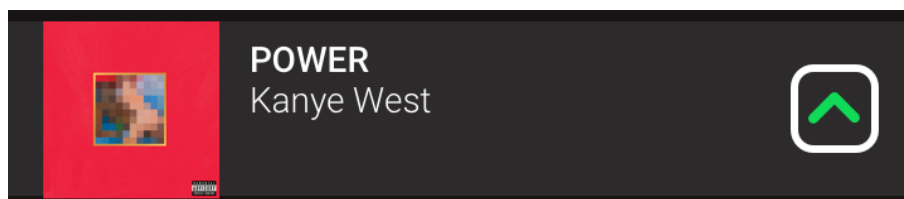


FIGURE 31 - A REQUESTED SEARCH RESULT THAT CAN BE UPVOTED

If the user has already upvoted the song, or pressed “bop”, a clicked upvote button will be rendered.

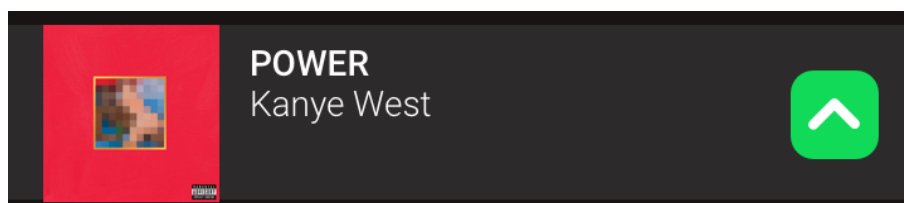
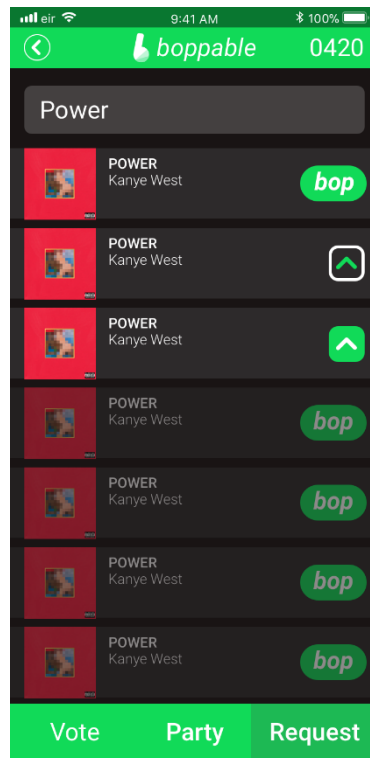


FIGURE 32 - A REQUESTED SEARCH RESULT THAT CANNOT BE UPVOTED

The final design for the Request screen looks like this:



Vote

One major element during the testing of Jukestar was the discovery that scrolling through a list of small track components requires a lot of attention from the user. The vote screen will address this problem by containing a Tinder style focused voting game. The user will be shown one song at a time and they will have the option to vote it up, vote it down, or skip to the next song without voting for it. The details for the song in question should take up most of the screen. Towards the bottom of the screen, the three voting buttons can be seen. The upvote button shall be a large green button that says “bop”. The downvote button shall be a large red button that says “flop”. The button to skip the current song and vote on the next one will be a smaller grey button in between the other two that says “next”. Like Tinder, users will also be able to swipe the cover art image to the right to upvote it and to the left to downvote it.

The overall design for the Vote screen looks like this:

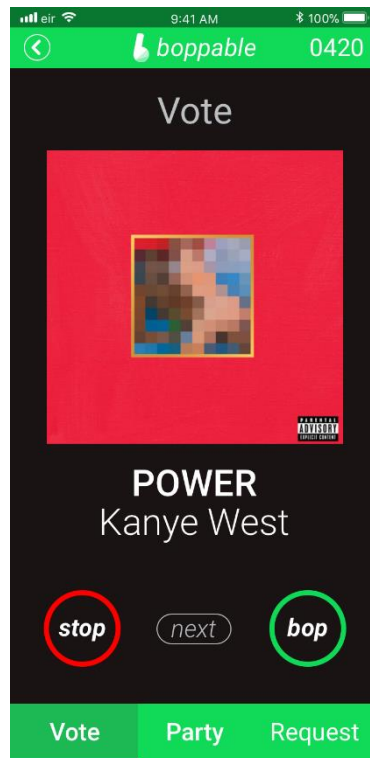


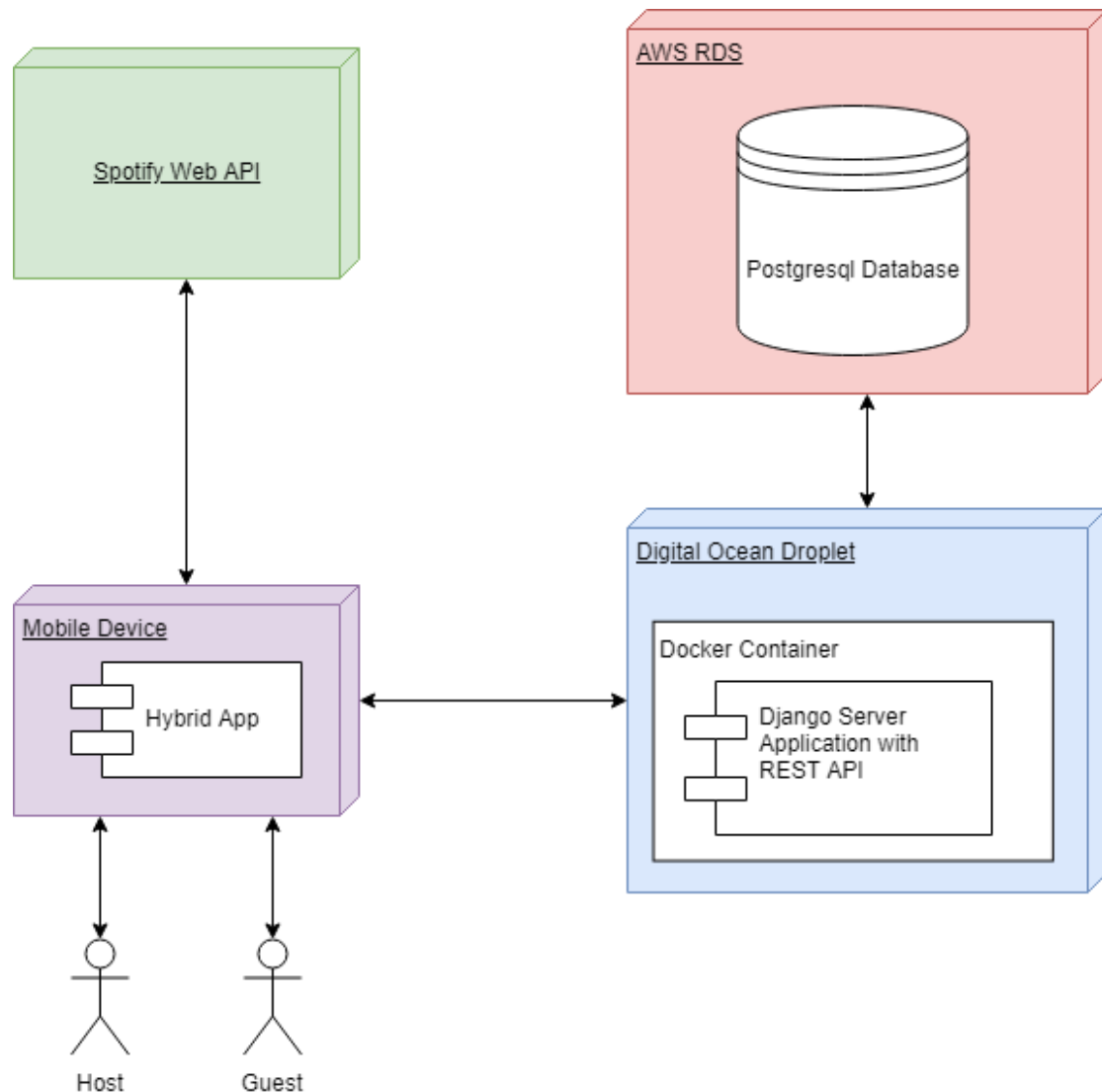
FIGURE 33 - VOTE SCREEN DESIGN

3.4 Server Application

The client application will be what gets installed on a user's device. With most mobile phones having limited storage, the client application need to be kept light. Because of this, the client application should not be doing much other than interfacing with the users and Spotify. The functionality for creating parties, handling song request, tabulating votes etc, needs to be done of the server side.

4 Development & Architecture

4.1 System Architecture



The figure above shows a high-level description of the system's architecture. The client application is hosted on the user's mobile device. It makes HTTP requests to both the Spotify Web API and the Django server application. From the Spotify API, the client application fetches song data which will both be used to play songs on host's device and also sent in turn to the Django server application. The Django application is contained in a Docker container which is hosted on a Digital Ocean droplet. This server is used to create new parties, handle song requests, tabulate votes etc. It communicates with a PostgreSQL database that is hosted on an instance of Amazon Web Services (AWS) Relational Database Services (RDS).

4.2 React Native Basics

4.2.1 Flexbox

In order to explain how the components in the client application work, an explanation of some key React Native concepts is required. First is Flexbox. Flexbox is a tool designed to give developers a means of keeping layouts consistent across all screen sizes. Each React component can have a style object, and one of the attributes this object can have is "flex". A component's flex value lets the app know how much of the screen the component should take up.

If an app only has one `<View>` component and that component has a flex of 1, it will take up the entire screen. If the app has three `<View>` components and they all have a flex of 1, the screen will be divided equally between them. Finally, if one of the views has a flex of 3, while the others still have a flex of 1, that view will take up three times the amount of space as the other two views. It works like a ratio.

Components can be nested and the flex values will still apply. For example, say a screen has two views both with a flex of 1. The views will take up half the screen each. If we then nest three more views within the bottom view and give them all a flex of 1, these three child views will take up an equal amount of space within the parent view.

This allows the developer to design complex components using ratios instead of pixel values, meaning that the component should look relatively the same across all devices. Flex values are used constantly throughout the app's components. For example, the Vote screen contains three parent components: the vote text container, the vote card container, and the vote buttons container. These are arranged vertically with a flex of 1, 5, and 2, respectively. It is clear that the vote text component takes up $1/8^{\text{th}}$ of the screen, the vote card $5/8^{\text{th}}$, and the vote buttons $2/8^{\text{th}}$. An example of nesting can be seen in the vote buttons component. This component contains three sub-components, the flop button, next button, and bop button. They are arranged horizontally and have a flex of 1 each, meaning that they take up an equal amount of space within the parent component.



FIGURE 34 - VOTE SCREEN WITH FLEX RATIOS

4.2.2 State and Props

React Native components come with two data types that control them: state and props. Props allows a developer to construct a component that changes appearance depending on the information passed to it. This passed down information is call the components props. For example, in Boppable, each request component contains an upvote and downvote button. However, these are actually the same VoteButton component just styled differently. One button gets rendered with upvote as a prop, and the other gets rendered with downvote as a prop.

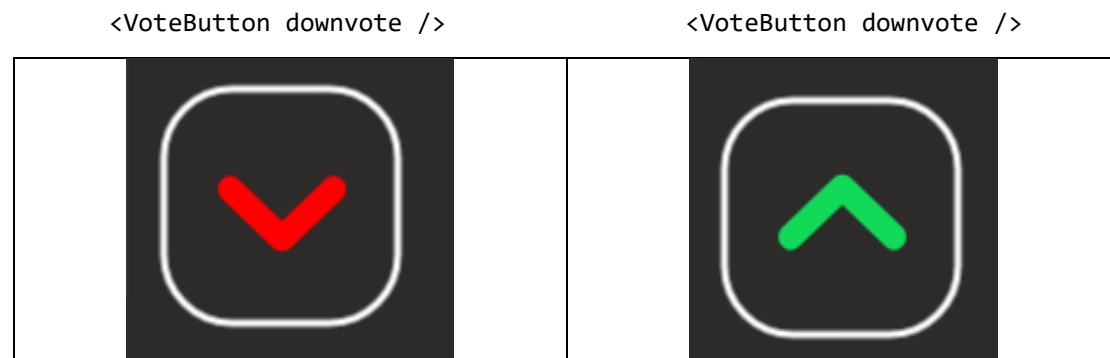


FIGURE 35 - A VOTEBUTTON COMPONENT BEING STYLED WITH DIFFERENT PROPS

A components props will stay the same for the entirety of the component's lifetime. To change data from within a component, state must be used. State stores data about a component. A component's initial state is set in its constructor. It can be updated from anywhere within the component by calling `setState()`. A parent components state values are often used as props for child components.

4.2.3 Lifecycle Methods

Every component in React Native has several lifecycle methods that can be overridden to run certain code or functions at specific times in the process. The most important of these are `componentDidMount()` and `componentDidUpdate()`.

componentDidMount()

This method gets called when the component successfully mounts on the screen. It is often used a the place in a component to make API calls. For example, in this application, the PlayerContainer is mounted with a track's ID as a prop. The PlayerContainer needs to get the name of the song with this ID, the cover art image URL, and the name of the artist before rendering the Player component. When the component mounts, a `getTrack` call to Spotify is made from within the `componentDidMount()` method. The relevant data is returned and then the Player component can be rendered with this data as props.

componentDidUpdate()

This method gets called whenever there a change to a component's props or state. It gets used in similar ways to `componentDidMount()` but can be called multiple times throughout a component's life and not just when it mounts. An example of this method being utilised is song searching. Whenever a new character is typed into the search box, the search screen's state updates with a new search string. This change of state triggers `componentDidUpdate()` and a new search request to Spotify is called.

4.2.4 AsyncStorage

AsyncStorage allows for the persistence of key-value data to the client's device. This gets used during room creation and song playing. If a user creates a room, they will be saved in AsyncStorage as a host. During the lifecycle of playing a song, the PlayerContainer checks to see if the user is a host. If not, songs will not play on that device.

4.3 UI Components

4.3.1 Party

The Party screen is the most complex screen in the application. It renders a `PlayerContainer` which, in turn renders a `Player`. Below that, it renders a `RequestList` which is a list view that renders a `Request` for each element in the list.

4.3.2 Search Screen

The Search screen renders a `SearchBar` component at the top of the screen and renders a `SearchResultList` below it. The `SearchResultList` is another list view that renders a `SearchResult` component for each element in the list.

4.3.3 Vote Screen

The Vote screen is to be the screen that allows for Tinder style swipe voting. Multiple open-source React Native swipe card libraries were tested when implementing this feature. This led to the discovery that performance was a major issue with this application. Swipe cards were implemented easily but were incredibly slow when responding to swipes to the point of being unusable. It could be up to ten seconds before the card followed the path of a finger swiping across the screen. With this being such a visible issue, the swipe card functionality was removed from the app. However, because a lot of research was put into the design of a screen like this, the UI for this screen was still developed.

4.4 Party Lifecycle

4.4.1 Playing Songs

As established when researching Spotify and `react-native-spotify`, playing a song through the app was a manageable task. While developing the `Player`, however, it was discovered that there was no way to call an event when the playing song ends. This meant that a custom solution needed to be constructed. This is the solution that ended up being implemented:

In the `PlayerContainer`, when a new song is played, the song's duration is fetched from Spotify and saved in the component's state. Next, the song's end time is calculated by getting the current time in milliseconds and adding the song's duration to it. The parent component, `Party`'s `UpdateEndTime()` method is called with this new end time as a parameter. This method sets the component's state with the new end time. Finally, the `Party` object will fetch the `Party` from the server every five seconds. If the current time is greater than the state's end time, a request is sent to the server to delete the first track from the playlist. This lifecycle will continue, the next song will be played, and a new end time will be calculated.

This is clearly not the most efficient way to handle song playing on this app, but with time constraints and other elements of the project needing attention, a better solution such as Django channels could not be explored or implemented.

4.4.2 Updating Party

Initially, a goal of this app was to send a request to the server every time an upvote, downvote, or bop button was pressed on any device so that the server would know to update the other users' devices. However, because each client app is already fetching the party information every five seconds, it seemed wasteful to add in more requests. If a user clicks the upvote button, they should see the number of votes increase within a few seconds.

4.5 Server Application

4.5.1 Architecture and Database Models

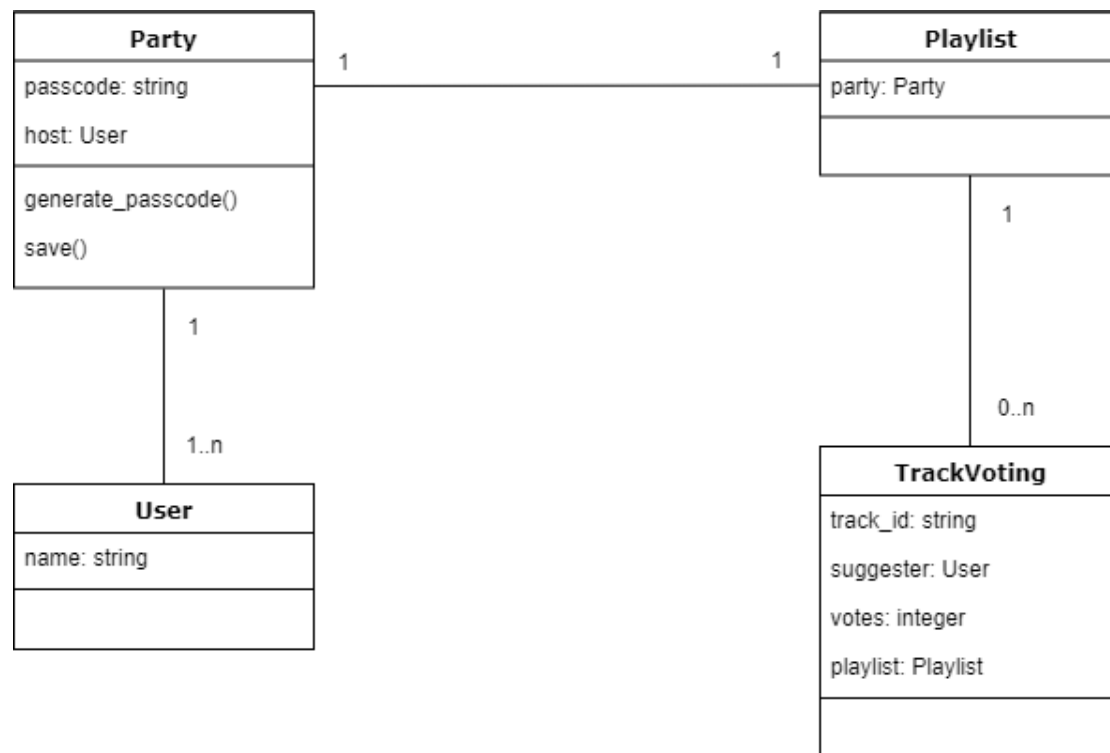


FIGURE 36 - CLASS DIAGRAM

Party

passcode: string

A four-digit party identifier

host: User

The ID of the user that is the host of the party.

generate_passcode()

Generates a unique, four-digit passcode that is not already in use by another party

save()

Saves the party to the database

Playlist

party: Party

The party to which the playlist belongs

TrackVoting

track_id: String

Spotify's track identifier

suggester: User

The user who suggested the song

votes: Integer

The number of votes the track has

playlist: Playlist

The playlist that the track belongs too.

User**name: String**

The user's name

4.5.2 View Methods

track_voting_upvote()

Adds 1 to relevant track's votes field and saves to database.

track_voting_downvote()

Subtracts 1 from relevant track's votes field and saves to database.

does_track_voting_exist()

Returns true if relevant track is in the playlist already.

4.5.3 URL Endpoints

Below are URL endpoints available for the client application to call:

Method	Endpoint	Description
GET	/	API Root
POST	/users/	Create User
POST	/parties/	Create Party
GET	/parties/{party id}/	Get Party
POST	/trackvoting/	Create TrackVoting Object
POST	/trackvoting/exists/	Check if TrackVoting Object Exists
GET	/trackvoting/{trackvoting id}/upvote	Upvote Track
GET	/trackvoting/{trackvoting id}/downvote	Downvote Track
DELETE	/trackvoting/{trackvoting id}/	Delete Track From Playlist

4.5.4 Deployment and Hosting

This server application is deployed using Digital Ocean. The Django application is containerized within a Docker container which stored on a Digital Ocean Droplet. Server solutions such as Apache and Nginx were considered but, for the scope of this project, simply running Django's manage.py program was sufficient. The Django models are stored on a PostgreSQL database which is being run on an AWS RDS instance.

5 System Validation

5.1 Testing

5.1.1 Early Client Testing

Testing the client application was done on a largely visual basis. Developing with React Native made it very easy to see changes made to the app in an instant, even on a phone. All development for this app was done on my own device. The app could be run on my device with ease and it made more sense to see progress be made on an actual device instead of on a virtual device on a computer. React Native's development server allowed for the live reloading of the application every time a change was made to the code. This meant that new visual components could be seen instantly and any changes that needed to be made to then became very apparent.

5.1.2 Early Server Testing

When developing the server, the Django Admin portal was used heavily to ensure that everything worked correctly. Once models were designed and the necessary functionality to create, update, and delete instances of these models was added, the Django Admin portal was used to ensure that these actions were possible and working as planned. Parties, playlists, users, and track-voting objects could be created and deleted through the portal. It was a goal to design unit tests for the server application so that these features could be tested automatically but they were never made due to timing issues.

5.1.3 API Testing

When making the endpoints for the server application, each endpoint needed to be tested to ensure that the server was returning the correct responses to different requests. To do this, the API development software Postman was used. With Postman, I was able to construct different HTTP requests to the server. For example, when testing the song exists check, a POST request would be constructed via Postman with a JSON body containing the track ID and the ID of the playlist it should belong to. The request would be sent to the server's URL with the correct endpoint and the response would show up in Postman. This allowed networking development on the client application begin with assurance that any errors with network requests would be because of how the request was constructed on the client side and not how they were handled on the server side.

5.1.4 User Testing

On Saturday, the 31st of March 2018, a user testing session to test out Boppable in a party scenario. Eight guests attended, all within the 18-24-year-old demographic. Some guests had backgrounds in technology, some in design, and some with experience in neither. This provided a good array people with different backgrounds and would likely provide an array of different kinds of feedback. The people with technology backgrounds were expected to have larger opinions on UI and performance. The people with backgrounds in design were expected to have larger opinions on User Experience. However, the guests with no tech or design backgrounds were the most valued testers as they expected to deliver opinions on the app as a whole without knowing exactly what to look for and without knowing how applications like these work in the background.

A party was started on my device and guests were invited to connect. The first thing that was observed was that all guests picked up how to use the app very quickly. When asked how they knew what to do, the consensus was that it felt very familiar and "looks like Spotify". Songs began flooding into the party's playlist. It was clear that song requesting was easy and intuitive. Voting seemed to be very intuitive too.

No guest needed to ask how to vote a song up or down. When asked how they knew how to vote for songs, guests expressed that the up and down arrows were easy to find and distinguish. One guest said: “it is clear that green means good and red means bad”. With regards to UI, the user testing session proved many of the design decisions a success.

Over time, however, many issues began to present themselves. One key issue was to do with performance. It started to take a long time for button presses to register with the app. A user would click the upvote button and be waiting seconds before the app would alert the user that the vote went through. This began to get very frustrating for the users. Related to this, it was discovered that sometimes a vote for one song would register with another song instead. This happened because the button press would register too late, and songs may have switched around in the list because the votes changed. In order to see if this was a client-side or server-side issue, I opened the Django Admin on a web browser and sent requests to the party manually. Changes would happen instantly on the Admin portal which indicated that it was a client-side issue. This made sense as the client app was getting the party data from the server every five seconds. This likely caused performance to drop dramatically.

Another very fatal design flaw was discovered. When the party started, the first song played on my phone via the speaker. When that song ended, the next song began playing on every guests’ phone. It was clear that there was an error made when working on the feature to only play the songs via the host’s device. Looking through the client-side code, the error was discovered: the app only checks if the user is a host when the component mounts. If the user is a host, the `playSong()` function is called. However, the `playSong()` function also gets called in the `componentDidUpdate()` function. There is no user type check in this function, so every device will play a song every time the Player component updates. Updates happen when state or props value change. The first time this happens for the Player component is when the first song ends and gets deleted from the database. That explains why no guest device began playing songs until the first song ended. This issue signified a major design flaw and since then, a check to see if the user is a host or not has been added to the `playSong()` function, where it should have been all along.

Some non-technical issues were also observed during this session. Due to the novelty of this app, guests began to play with it in undesirable ways. Two guests would flood the playlist with songs that they knew other users would not like. It made other guests feel the need to constantly downvote anything that they requested which, in turn, caused guests to be frustrated instead of enjoying the music. This highlighted the possible need add functionality to allow the host to ban some guests or veto some song requests. There is a chance that this may not be necessary because in an actual party environment, users would be more likely to take the app seriously. A host can always threaten to use a premade playlist instead which would entice guests to behave and only request songs they like. Regardless, these are features worth looking into in the future.

5.2 Demonstration

Below is a collection of screenshots showcasing the lifecycle of a party in Boppable.

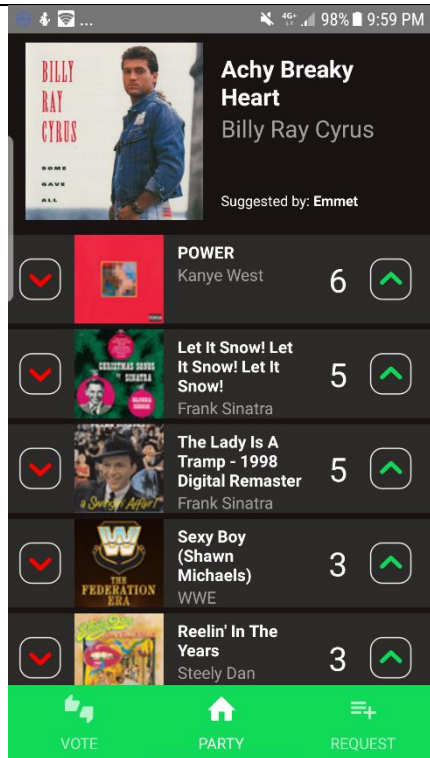


FIGURE 37 - A PARTY IN ACTION

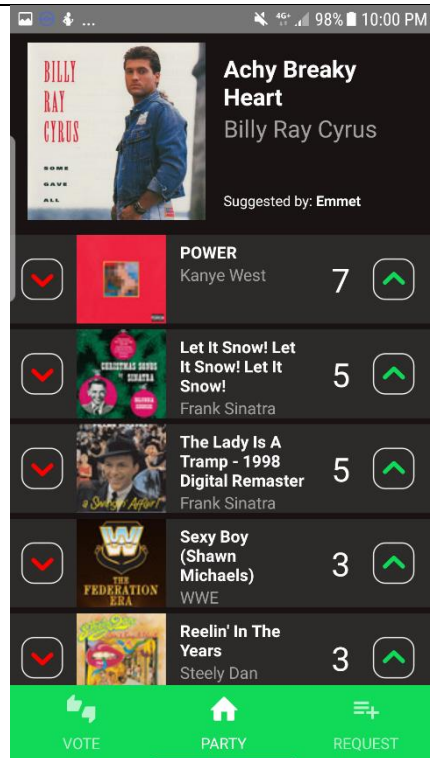


FIGURE 38 - THE TOP SONG GETS VOTED UP

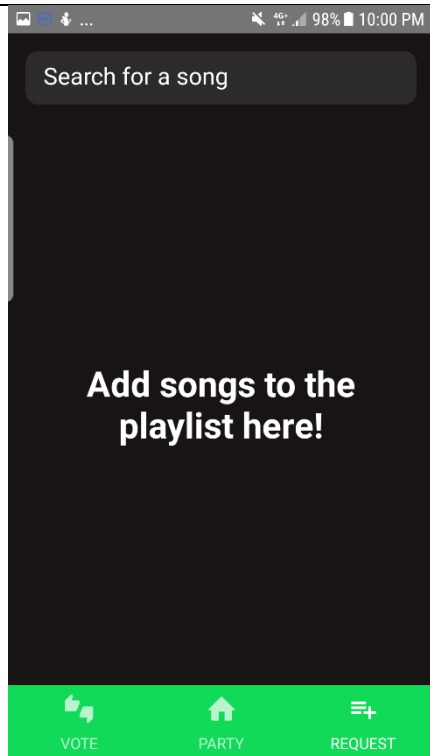


FIGURE 39 - THE SEARCH SCREEN BEFORE TYPING

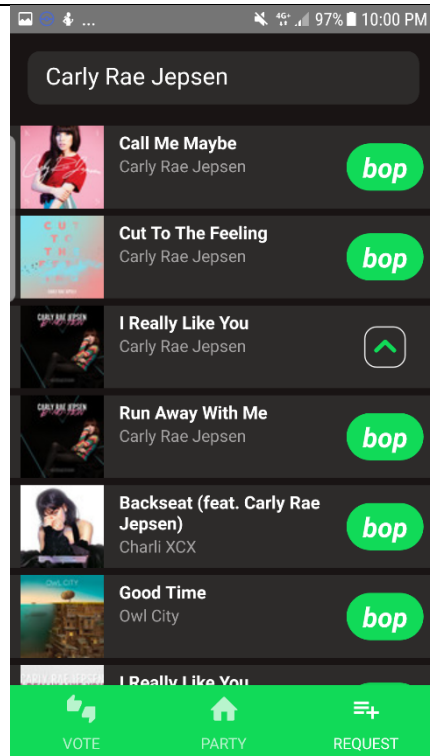


FIGURE 40 - THE SEARCH SCREEN AFTER TYPING

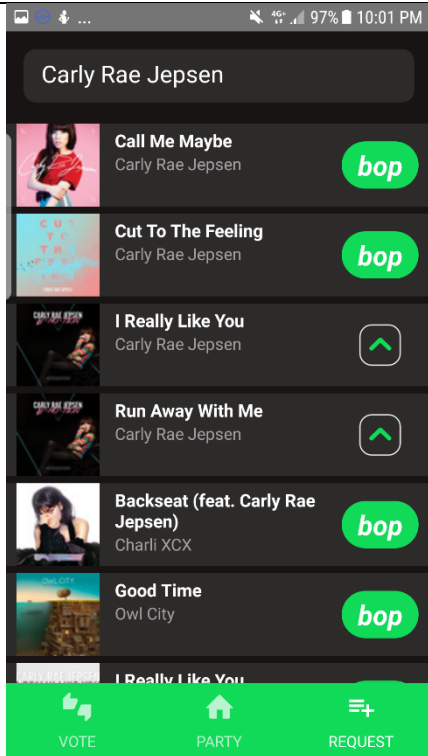


FIGURE 41 - "RUN AWAY WITH ME" GETS REQUESTED

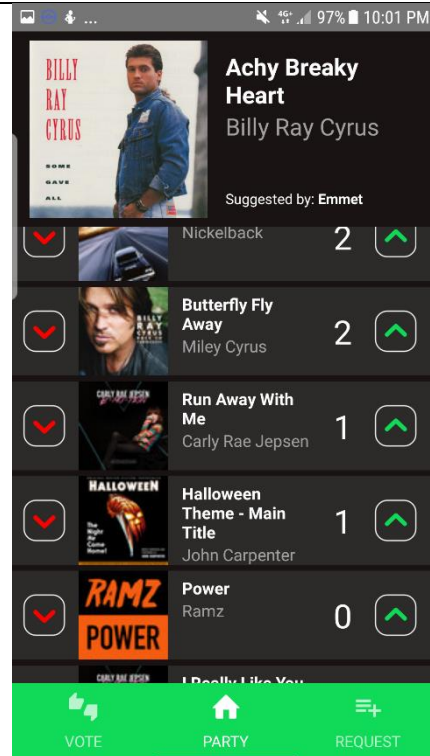


FIGURE 42 - REQUESTED SONG SHOWS UP IN PLAYLIST

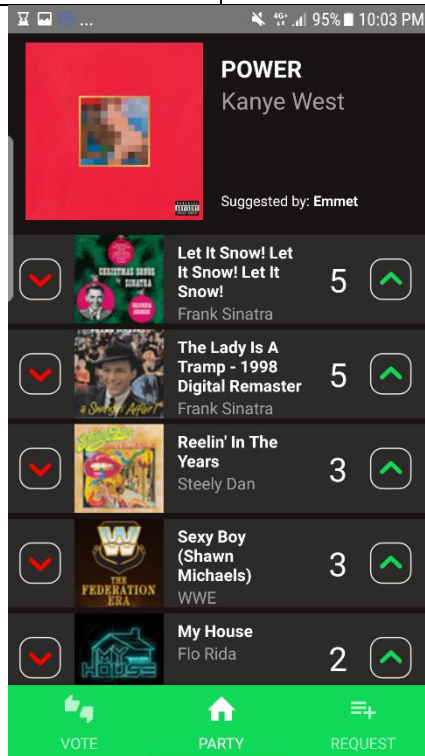


FIGURE 43 - HIGHEST VOTED SONG GETTING PLAYED NEXT

6 Project Plan

6.1 Initial Proposal

The initial proposal for this project is very different to what the project ended up being. The core functionality of the project was deemed to not be complex enough and other functionality needed to be proposed. The summary of the initial proposal can be seen below:

The goal of this project is to create and deploy an application to allow guests of a party have a say in what songs are getting played.

This project will allow a user to either set up or join a room. When in a room, users will have the option to add a song to the queue. Users will also have the option to see all the songs currently in the queue and, for each song, vote on whether or not they would also like it to be played. When the current song is finished, the song in the queue with the highest number of upvotes will get played next.

This application will come with a recommender system. This will be used to suggest songs to users to add to the queue based on the type of songs that have been playing. It will also be used to play similar songs to what has been playing if there are no more songs left in the queue.

Another feature of this application will be the option to use a “beatfinder”. This will analyse the song and identify the beats. It will start the song at the first beat and end it at the last beat so that long introductions to songs can be skipped in a party environment.

The first two paragraphs of this summary are still incredibly relevant to the project. The functionality described in these paragraphs have been achieved. The features described in the other paragraphs were proposed for complexity’s sake and were not core features. The first of which, the recommender system, was deemed a helpful feature because it would quickly suggest new songs for users to request which would save the user time in searching for new songs. The second feature, the beatfinder, was intended to identify when the first and last beat of a song occurs so that long intros or outros could be avoided.

6.2 Changes to the Proposal

Early into the project, it became apparent that the core functionality was more complex than originally thought. Understanding new technologies such as Django and React Native required lots of learning and time. Being so new to React Native required me to develop the client application as I learned. This forced a very iterative process. Very basic components would be built and revisited when better ways to develop them were discovered. Understanding Spotify’s API and implementing it into the client application also required lots of time.

It was discovered that additional features such as a recommender system or a music analyser were far beyond the scope of this project. If these features were to be explored, they would warrant being the sole focus of the project.

Another change to the proposal was that the project focused on UI/UX a lot more than originally intended. This came as a result of the feedback given from user testing Jukestar. It became clear that party environments are quite hectic and expecting a user to dedicate their entire undivided attention to the app was unrealistic and further research into UI components than expected was needed.

7 Conclusion

7.1 Client Application

The client application was carefully designed to allow users create and join parties, request and vote on songs. This was done with an emphasis on being intuitive and easy to use. All core features were successfully implemented, while secondary features such as swipe voting still need implementation. Performance issues also need to be addressed. User-testing was conducted and proved that the app was easy to understand and not too distracting from the party. The host of the party is required to have a Spotify premium account but this is not too discouraging as there is a chance that a guest of the party will have one if the host does not.

7.2 Server Application

The server application was designed to communicate with the client application via HTTP. It can create new parties, handle song requests, tabulate votes and persist it all to a PostgreSQL database. All functionality works as intended and, when testing its API, it proved that responses were given in a quick and reliable manner.

7.3 Personal Reflection

Over the course of this project, I learned a lot about popular technologies. Specifically, React Native and Django. React Native is currently in high demand in industry and I feel that spending the last six months learning has proved fruitful for prospective employment opportunities. While working on this project, I also discovered that I have an affinity for visual-based application development. It has cemented that a career in UI/UX design is what I will pursue after I graduate.

If I was to do this project again, I would heavily consider using a non-agile methodology. Agile methodologies are meant for large teams. Various team members will take on different responsibilities. Sometimes, a team will only be responsible for one class, with each member of the team working on a different aspect of the class. Scrum does not necessarily apply well to a one-person development team. The concept of sprints was useful, but, in practice, most of the other elements of Scrum were ignored as they did not feel necessary.

I believe the project was a success and achieved most of what I wanted it to. The idea for this project came to me out of frustration with how often arguments over music would happen at events I attended. I am proud of what I made and believe it has a lot of value. While showing my peers my progress, they would express excitement for when it's complete and emphasize that it is something they would use often. It felt good to design an application with purpose and a group of users in mind instead of simply picking a more complex idea with no real-world application.

7.4 Future Work

The project is in a state that allows for plenty of future work. An alternative to the current method of playing music needs to be made to improve performance. An event manager could be implemented so

that each client application only updates when necessary. Also, with performance improved, the Tinder style swipe voting feature can be implemented in full.

User testing also pointed out a possible need to give the host a suite of additional options so that they have slightly more control over the music playing than the guests. If this was to be pursued as a business, these features could be added as a premium option that users pay for.

The stretch-goal features mentioned in the initial proposal could also be pursued now that there are no deadlines in place.

8 Bibliography

1. Music N. crowdDJ - Nightlife Music - Everyone wants to be a DJ – Now you can! [Internet]. Nightlife Music. [cited 2017 Nov 27]. Available from: <https://nightlife.com.au/crowddj>
2. About [Internet]. Press. [cited 2017 Nov 27]. Available from: <https://press.spotify.com/us/about/>
3. Social Jukebox App for Spotify [Internet]. Jukestar. [cited 2017 Sep 27]. Available from: <https://jukestar.mobi/>
4. Schenck BF. Freemium: Is the Price Right for Your Company? [Internet]. Entrepreneur. 2011 [cited 2017 Nov 27]. Available from: <https://www.entrepreneur.com/article/218107>
5. Spotify Web API - Spotify Developer [Internet]. [cited 2017 Nov 27]. Available from: <https://developer.spotify.com/web-api/>
6. Spotify Android SDK - Spotify Developer [Internet]. [cited 2017 Nov 27]. Available from: <https://developer.spotify.com/technologies/spotify-android-sdk/>
7. Spotify iOS SDK - Spotify Developer [Internet]. [cited 2017 Nov 27]. Available from: <https://developer.spotify.com/technologies/spotify-ios-sdk/>
8. Jackbox Games [Internet]. Jackbox Games. [cited 2018 Apr 15]. Available from: <https://jackboxgames.com>
9. Tinder | Swipe. Match. Chat. [Internet]. Tinder. [cited 2018 Apr 15]. Available from: <https://tinder.com>
10. Purvis J. Why Tinder is so “evilly satisfying” [Internet]. The Conversation. [cited 2018 Apr 15]. Available from: <http://theconversation.com/why-tinder-is-so-evilly-satisfying-72177>
11. Pinkham A. Django Unleashed. SAMS Publishing; 2015.
12. Models | Django documentation | Django [Internet]. [cited 2017 Nov 28]. Available from: <https://docs.djangoproject.com/en/1.11/topics/db/models/>
13. Home - Django REST framework [Internet]. [cited 2018 Apr 15]. Available from: <http://www.django-rest-framework.org/>
14. Why AWS has such a big lead in the cloud | TechCrunch [Internet]. [cited 2018 Apr 15]. Available from: <https://techcrunch.com/2017/02/13/why-aws-has-such-a-big-lead-in-the-cloud/>
15. Why Is Docker So Popular? Explaining the Rise of Containers and Docker [Internet]. Channel Futures. 2017 [cited 2018 Apr 15]. Available from: <http://www.channelfutures.com/open-source/why-docker-so-popular-explaining-rise-containers-and-docker>
16. Five Reasons Why Developers Love DigitalOcean [Internet]. [cited 2018 Apr 15]. Available from: <https://www.forbes.com/sites/janakirammsv/2015/09/10/five-reasons-why-developers-love-digitalocean/#482262c844ef>
17. Spotify Android SDK Tutorial - Spotify Developer [Internet]. [cited 2017 Nov 27]. Available from: <https://developer.spotify.com/technologies/spotify-android-sdk/tutorial/>

18. IDC: Smartphone OS Market Share [Internet]. IDC: The premier global market intelligence company. [cited 2018 Apr 15]. Available from: <https://www.idc.com/promo/smartphone-market-share>
19. Mobile Operating System Market Share Ireland [Internet]. StatCounter Global Stats. [cited 2018 Apr 15]. Available from: <http://gs.statcounter.com/os-market-share/mobile/ireland/2016>
20. Architectural overview of Cordova platform - Apache Cordova [Internet]. [cited 2017 Nov 27]. Available from: <https://cordova.apache.org/docs/en/latest/guide/overview/index.html>
21. Elliott E. Top JavaScript Libraries & Tech to Learn in 2018 [Internet]. JavaScript Scene. 2017 [cited 2018 Apr 15]. Available from: <https://medium.com/javascript-scene/top-javascript-libraries-tech-to-learn-in-2018-c38028e028e6>
22. 7 Reasons Why React Native Is the Future of Hybrid Mobile Apps [Internet]. [cited 2018 Apr 15]. Available from: <https://www.upwork.com/hiring/mobile/react-native-hybrid-app-development/>
23. Finke L. react-native-spotify: A react native module for the Spotify SDK [Internet]. 2017 [cited 2017 Nov 27]. Available from: <https://github.com/lufinkey/react-native-spotify>
24. Sommerville I. Software Engineering. Tenth. Pearson; 2016.
25. sdlc_waterfall_model.jpg (600×401) [Internet]. [cited 2017 Dec 3]. Available from: https://www.tutorialspoint.com/sdlc/images/sdlc_waterfall_model.jpg
26. scrum-framework-diagram.jpg (415×308) [Internet]. [cited 2017 Dec 3]. Available from: <https://www.expertprogrammanagement.com/wp-content/uploads/2010/08/scrum-framework-diagram.jpg>
27. InfoSystem H. Importance of UI/UX In Mobile App Design Services [Internet]. Hyperlink InfoSystem. 2017 [cited 2018 Apr 15]. Available from: <https://medium.com/@hyperlinkinfosystem/importance-of-ui-ux-in-mobile-app-design-services-7f15683842e1>
28. Hybrid vs Native Mobile Apps - The Answer is Clear [Internet]. Y Media Labs. 2016 [cited 2018 Apr 15]. Available from: <https://ymedialabs.com/hybrid-vs-native-mobile-apps-the-answer-is-clear/>