Team 18 — Design Document

Amul Chaulagain, Logan Kulinski, Nick Pappas, Patrick Crowne, Siddharth Madapati

Index

Purpose

AnarQ is a platform geared towards streamlining the music selection process at gatherings, events and venues.

There will be two sides to this platform, the client side, and the server side. The client side is what most of our users will end up using, a web based (or possibly even a standalone) application that allows them to connect to a host, and submit songs of their choosing. Users can also vote on music that other people have submitted, allowing the playlist to be organized based on greatest user preference.

The server side will generally be used by the minority of our users. This is reserved for those who want to host an event using our service. The server will be connected to a streaming service of the server operator's choice, and will handle the music queue created by the clients. Additionally, the server application will have moderation features, choosing who can connect, what kind of music can be played, and have the ability to remove music that is deemed inappropriate or unwanted.

Together, these components will create AnarQ, the solution to the ever aging question "What music should we play?"

We are designing a system that allows for people at gatherings, small or large, to create a music playlist based on requests, with a voting system to push desired songs to the top of the list, and undesired songs to the bottom.

Functional Requirements

Users can access their Spotify account

- As a user, I would like to log in with my Spotify account
- As a user, I would like to access my liked songs
- As a user, I would like to access my saved playlists
- As a user, I would like to save the AnarQ playlist to my saved Spotify playlists

Users can access song information

- As a user, I would like to see the artist(s) associated with a song
- As a user, I would like to see the album that a song belongs to
- As a user, I would like to see if the song features explicit content

Users can create an account with our application

- As a user, I would like to create an account with a username/password

- As a user, I would like to create an account with Google (if time allows)
- As a user, I would like to add a profile picture to my account
- As a user, I would like to biographical information to my account
- As a user, I would like to update my account's username/password
- As a user, I would like to update my account's avatar/bio
- As a user, I would like to log out of my account
- As a user, I would like to temporarily disable or permanently delete my account

Users can vote on songs for the playlist

- As a user, I would like to search for a Spotify song and get its info
- As a user, I would like to add a song to the queue
- As a user, I would like to 'like' a song to increase its vote count
- As a user, I would like to 'dislike' a song to decrease its vote count
- As a user, I would like to report a song for a legitimate reason
- As a user, I would like to report a user for a legitimate reason
- As a user, I would like to remove a song that I have requested
- As a user, I would like to request to appeal a song that doesn't match the session filters

Users can host parties with a required Spotify login

- As a host, I would like to host a music session by linking my Spotify account
- As a host, I would like to add and vote on songs in my own music session
- As a host, I would like to pause the song playing in my music session
- As a host, I would like to send out a condensed link to my music session

These hosts can have music/user restrictions

- As a host, I would like to set a genre filter on the music in my session
- As a host, I would like to filter explicit content on the music in my session
- As a host, I would like to delete a song in my music session for a legitimate reason
- As a host, I would like to view and act on reported users/songs
- As a host, I would like to restrict the actions of certain users for a legitimate reason
- As a host, I would like to implement BPM filters to keep music at a similar pace
- As a host, I would like to approve/deny appeals for songs that don't match session filters
- As a host, I would like to set a cooldown period for song repetition
- As a host, I would like to set a cooldown period for song requests per user

Non-Functional Requirements

Architecture and Scalability

- As a developer, I would like to implement the backend in Java as a Spring framework
- As a developer, I would like to ensure that this application implements modular design
- As a developer, I would like to implement a frontend framework such as AngularJS
- As a developer, I would like to ensure long-term stability and scalability
- As a developer, I would like a distinct interface between server hosts and listeners

Performance

- As a developer, I would like to service single requests in less than two seconds
- As a developer, I would like to support thousands of concurrent sessions

Security

- As a developer, I would like to obtain an SSL certificate from a valid issuer
- As a developer, I would like to use secure sockets
- As a developer, I would like to encrypt requests and user data with RSA and Triple DES
- As a developer, I would like to use the OAuth process from Spotify's API to protect users' Spotify account information

Usability

- As a developer, I would like to implement intuitive user design
- As a developer, I would like to implement a Dark UI option
- As a developer, I would like to implement a simplistic, sleek, and tidy layout

Deployment

- As a developer, I would like to make this easily and readily available through a link
- As a developer, I would like to use modular design to prevent downtime during updates
- As a developer, I would like to notify users ahead of an update for any potential server issues

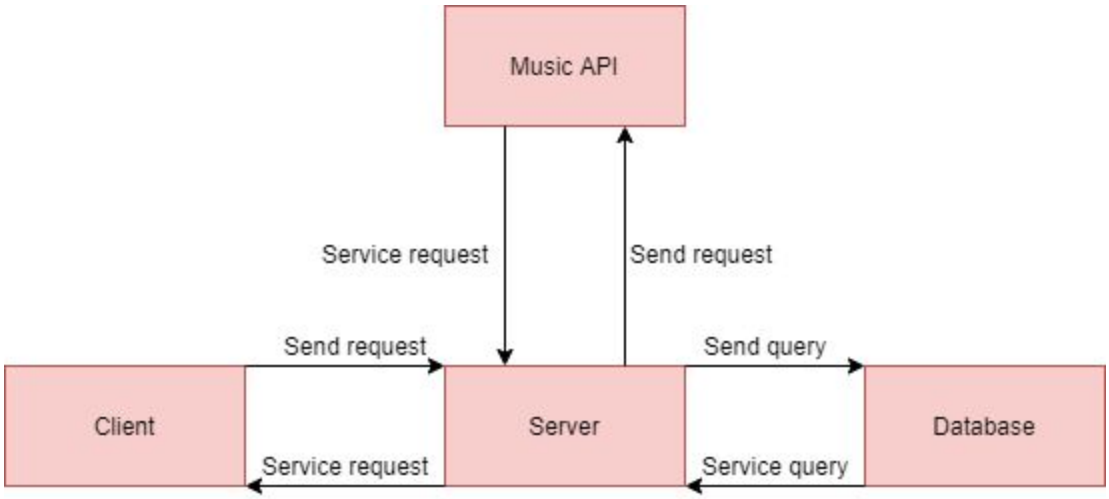
Design Outline

High Level Overview

We are going to be using a client-server model, where the server is heavy, handling all of the computation and queue organization, while the client is light, only submitting requests and votes.

The server will be the host, playing all the music through a connected music streaming service account. **The clients** will be the users, submitting requests and voting on different songs that have been requested.

Clients will send requests to the server either in the form of a song request, or a vote for a song. Servers will send confirmation of these requests back to clients, and will notify users of any issues with their request(s).



Design Issues

What kind of interface will be used to allow people to connect to the service?
Option A: Downloadable Application
Option B: Web-Based Application
Option C: Standalone Hardware Solution

Choice: Option B (Maybe also Option A)

Justification
A web based application is incredibly accessible, requires no downloading, and can be run on almost any system with a common web browser. Since accessibility is an incredibly important pillar of this project, having the most people being able to connect to the service at any given time would fulfill that mission.

Making users download an application would require us to develop an app for multiple different operating systems, at the very least Android and IOS, which takes up a lot of time and resources. Being able to develop a solution that works on all web browsers is significantly easier and more time effective.

This isn't to say there aren't benefits to having a downloadable application. A well made downloadable application can be far more reliable and stable than even the best web applications.

Where will the primary execution of the software lie?
Option A: Server-Side
Option B: Client-Side
Option C: Central Server-Side

Choice: Option A

Justification
While having a central server handle all the requests would make the computation on a server's device a lot simpler, it would increase server costs on our side, leading to a less cost efficient program. Having the localized server handle the execution means that we can offload these calculations to each host, allowing for less memory dedicated to our servers, which would just have to direct requests back to the local servers.

Operating on the client side would derivate this workload more, but since our choice is to go web based, it would lead to performance issues, and maybe even security issues if not implemented correctly. It is both easier, more efficient and less expensive to host execution on the server side.

How will music be played?
Option A: Licensed music to be played
Option B: Use a Local media library
Option C: Sync the application with an existing streaming service

Choice: Option C

Justification
Historically, licensing music has always been an expensive process. Since millions of dollars is one of the many things we do not have access to for this project, it would be in our best interest to not use Option A. Option B would limit the music choices to what the user has downloaded onto their device, a practice that has fallen out of style in recent years. This method would also limit the AutoDJ features, since downloaded files usually lack critical metadata that could aid in the creation of seamless playlists. This leaves Option C, where we connect to an existing streaming service, with millions of hours of already licensed music, complete with accurate metadata. This choice provides the most flexibility for our service, and allows the most seamless way to playback music.

How will this application connect to your Spotify account?
Option A: Spotify Web API with Javascript
Option B: Spotify Web API Python Wrapper
Option C: Spotify Web API Java Wrapper

Choice: Option C

Justification
Thankfully, Spotify has recently provided to the public a very useful and efficient web API , allowing us to foster a connection between our application and a user's Spotify account. This will make it more efficient for a user to access their attributes such as songs, playlists, explicit filter settings, and much more, all on our application.

Option A, using their pure Web API with Javascript would definitely result in faster response time between the clients and the server. However, using Javascript to program this server would result in much more time spent attempting to work around unintuitive syntax, poor debugging software, as well as a lack of Javascript experience to be able to efficiently program all aspects of

this software. Option B, using their Spotify API Python Wrapper, would provide efficient programming on our end, with more intuitive syntax, better debugging software, and more experience. However, using Python to execute many requests on the server-side would be more likely to lead to more time elapsed between executions.

Option C acts as a perfect mix of the two previous options, as it balances the more intuitive syntax of Java that all members of the team are more comfortable with, as well as the speed of the execution being much better than that of Python.

How will the AutoDJ determine what song should play next?
Option A: Random Choice
Option B: Pseudo-Random Choice (Random, with exceptions)
Option C: artist-similarity-based music selection
Option D: Timbre based music selection

Choice: Option D, Option C

Justification
Using the timbre of the music that was recently played to select a new song allows for the same energy (or, as people in our generation call it, the "vibe") to stay rather consistent, not jumping wildly from a 60's romantic ballad, to Skrillex's latest electronic assault on the ears. A purely random choice, or even a pseudo-random choice, would lead to this situation frequently, not to mention it's already popular use in many music streaming services out there, that we intend to build upon, not emulate.

This timbre matching would also preserve any conceptual music albums, albums intended to be listened to in a specific order, from being scrambled, or abruptly cut off towards a sudden change near the end of the composition.

Choosing based on artist similarity is also a great way to preserve a certain energy, since most artists tend to make music that is stylistically similar to the rest of their music, leading to pleasing results.

How will user data be handled?
Option A: Require both users and admins to have an account
Option B: Require only users to have an account
Option C: Require only admins to have an account
Option D: Don't require accounts at all

Choice: Option D, but in the future, Option C

Justification

Having either users or admins sign up to have an account adds another barrier of entry to people wanting to use our software, managing everything through temporary accounts means that connecting to the service is quick and easy for the user. An easy to use software is a software people are much more willing to use. Differentiating users is as easy as gathering their IP address, which can be used for Blacklisting/Whitelisting users without even needing an account.

Having an account for admins would be useful down the line to allow for premium features to make this service somewhat profitable, but for now, keeping the barrier of entry low is the most important thing to give people a reason to want to use this software.

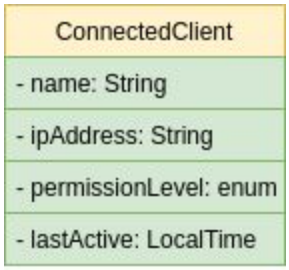
Design Details

Server Class

Duties

- Master class for the server, handles interactions between the other components of the server application.
- Only one instance per server application.
- Each server will have an associated ID used to connect
- Keeps track of clients through an associated ClientManager
- Contains a whitelist/blacklist of clients that can/cannot connect
- Contains a rule-set of what kind of music can be submitted for playback

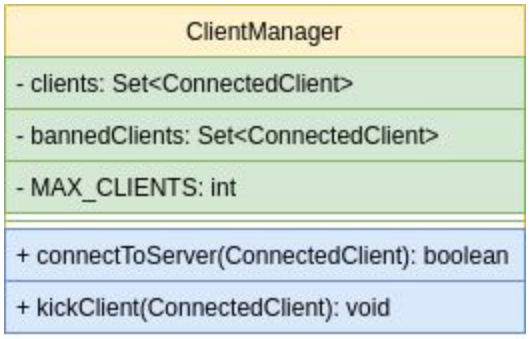
ConnectedClient Class



Duties

- Contains information about a specific connected client
- Instance is created upon the connection of a new client
- Contains the IP Address of the client connected
- Contains the nickname of the client connected
- Has the ability to send and receive information via a socket
- Upon connection, IP is checked against the list of blacklisted IPs
- If the ip is present on the IP blacklist, it will not be able to submit songs or vote
- Contains a permission level that grants certain privileges, for instance, being able to vote or being able to request songs
- Will keep track of the last time the client was active

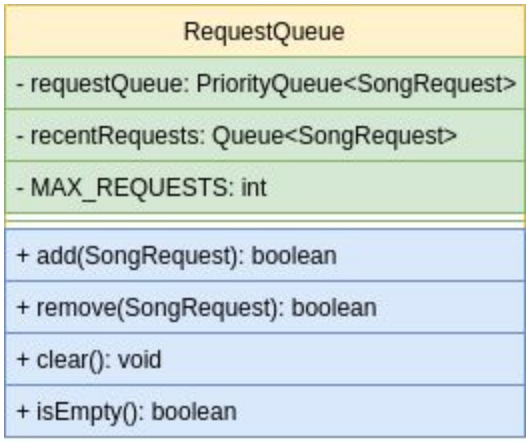
ClientManager Class



Duties

- Handles all of the connected clients
- When a user connects, they will be added to the array of connected Clients
- When a user disconnects, they will be removed from the array of connected clients
- Has the ability to remove a client forcibly from the server
- Grants clients permissions to perform tasks, like requesting a song or voting on a song
- If a client is unresponsive for a long time, it will be automatically removed

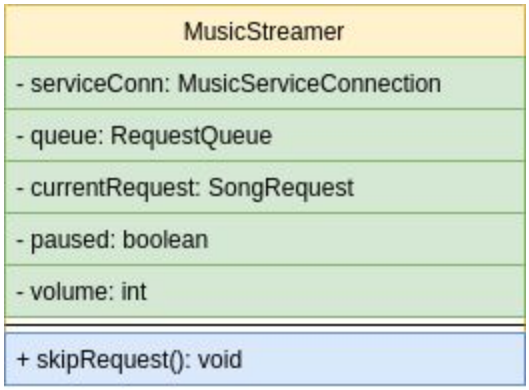
RequestQueue Class



Duties

- Manages the queue of requests
- Has the ability to add a new request
- Has the ability to remove an existing request
- Contains a weighted priority queue system to decide which order the music requested is to be played in
- Will autoplay each request at the top of the priority queue until there are none left.
- Will refresh frequently to keep the priority queue in order
- Has the ability to update an existing song with a new vote either for or against it
- Has the ability to clear all requests from the queue
- Keeps track of what songs were recently played

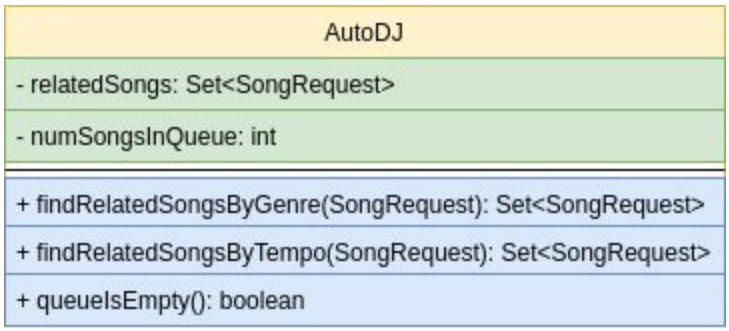
MusicStreamer Class



Duties

- Manages the execution of the request queue
- Connects to an associated music streaming service
- Pulls from the request queue and plays the music on the streaming service
- Keeps track of if the music is paused
- Keeps track of the volume
- Keeps track of the current metadata of the song being played
- Has the ability to skip to the next song

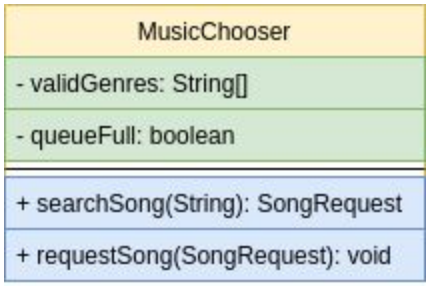
AutoDJ Class



Duties

- Discovers similar music to be played after the queue has ended
- Works in association with the music analyzer to create a playlist of similar music
- Has the ability to match tempo of songs and seamlessly splice them together

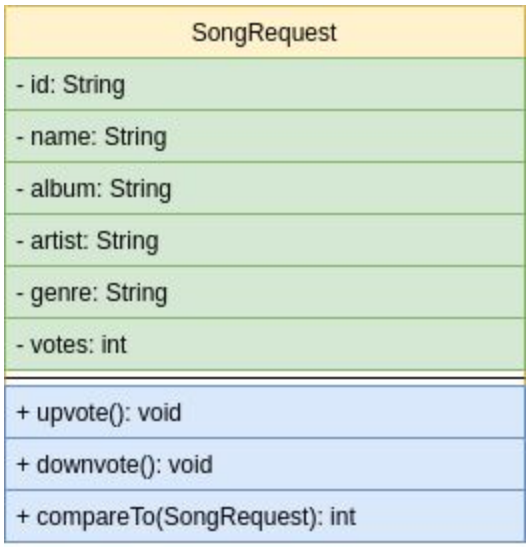
MusicChooser Class



Duties

- Manages how and what music the user can choose
- Allows the user to search for songs
- Allows the user to request songs

SongRequest Class

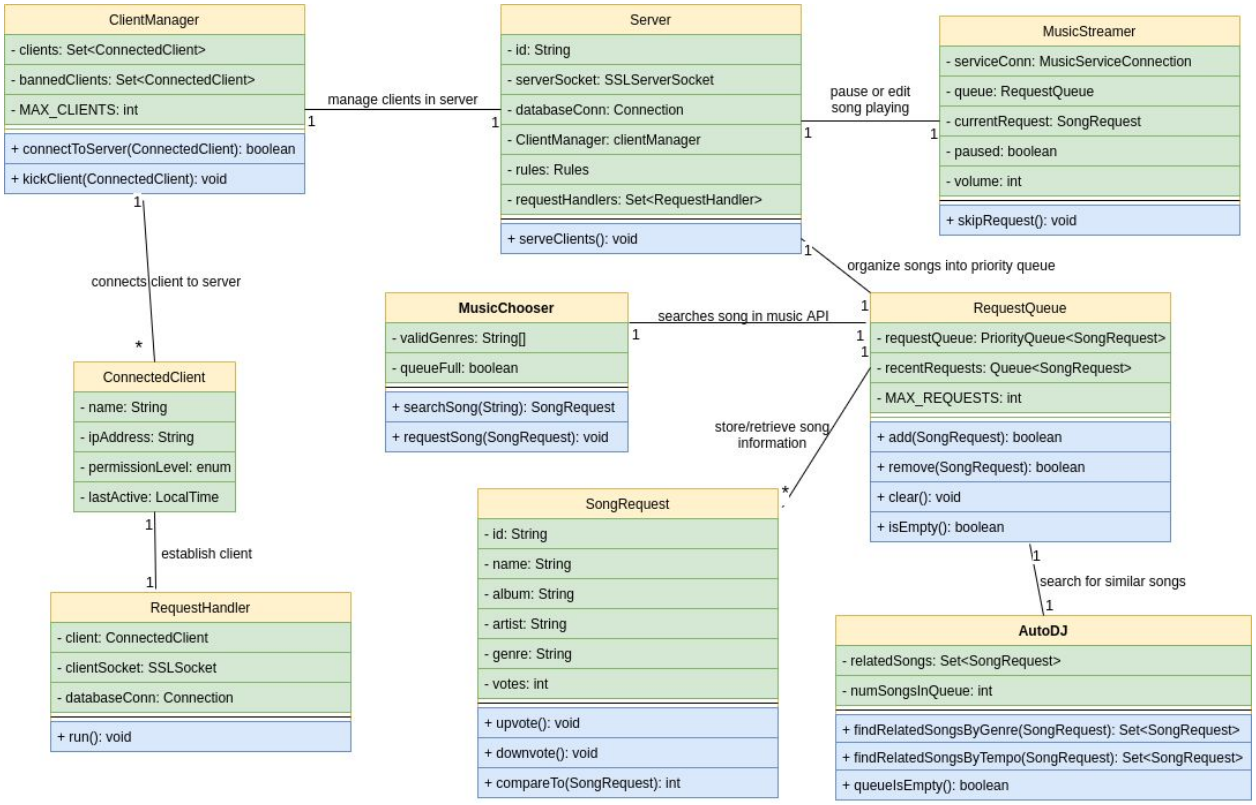


Duties

- Contains information about a specific song request
- Includes the ability to upvote and downvote a song which will send the data to the queue to be updated
- Can compare total vote score against another song

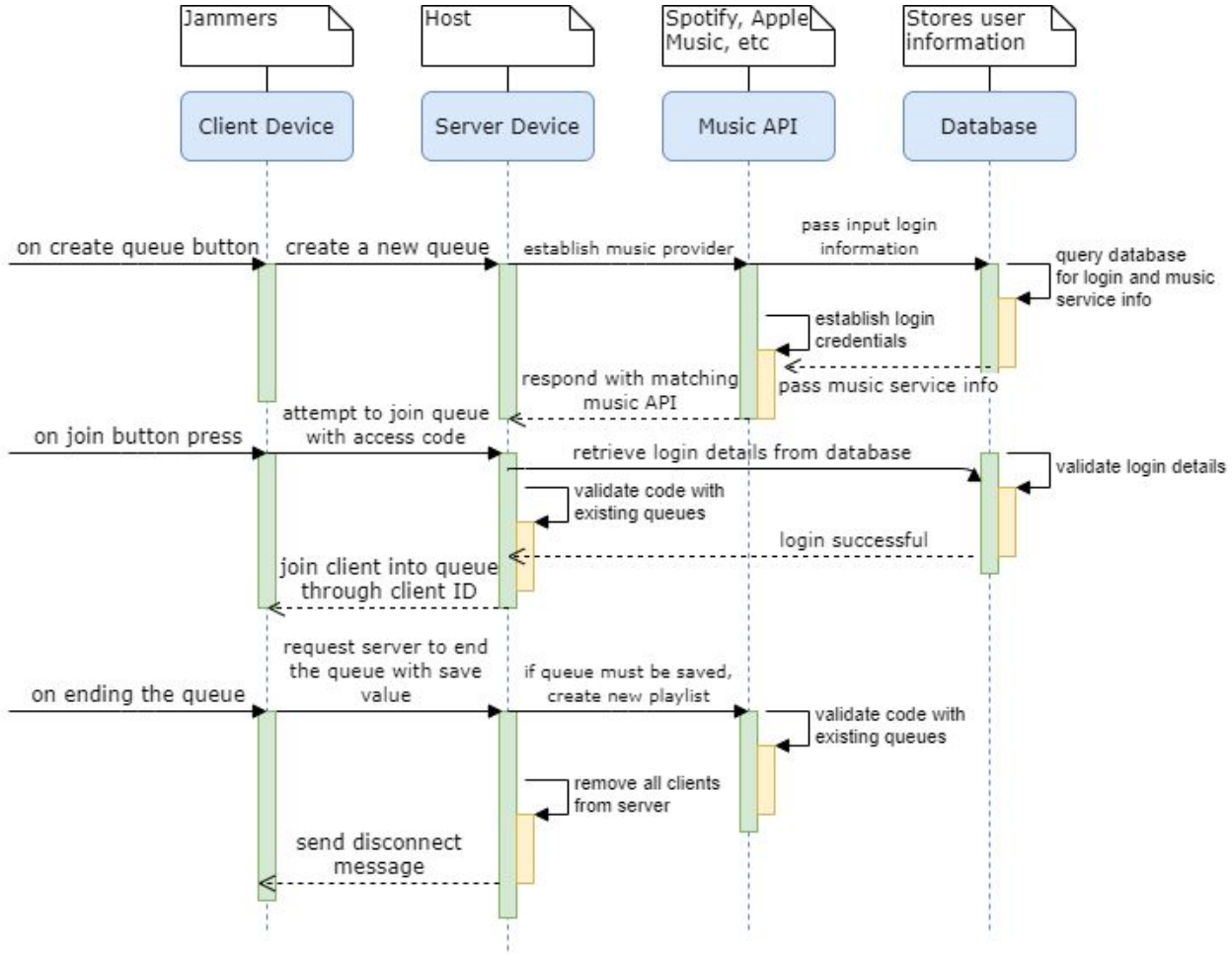
Diagrams

Class Diagram



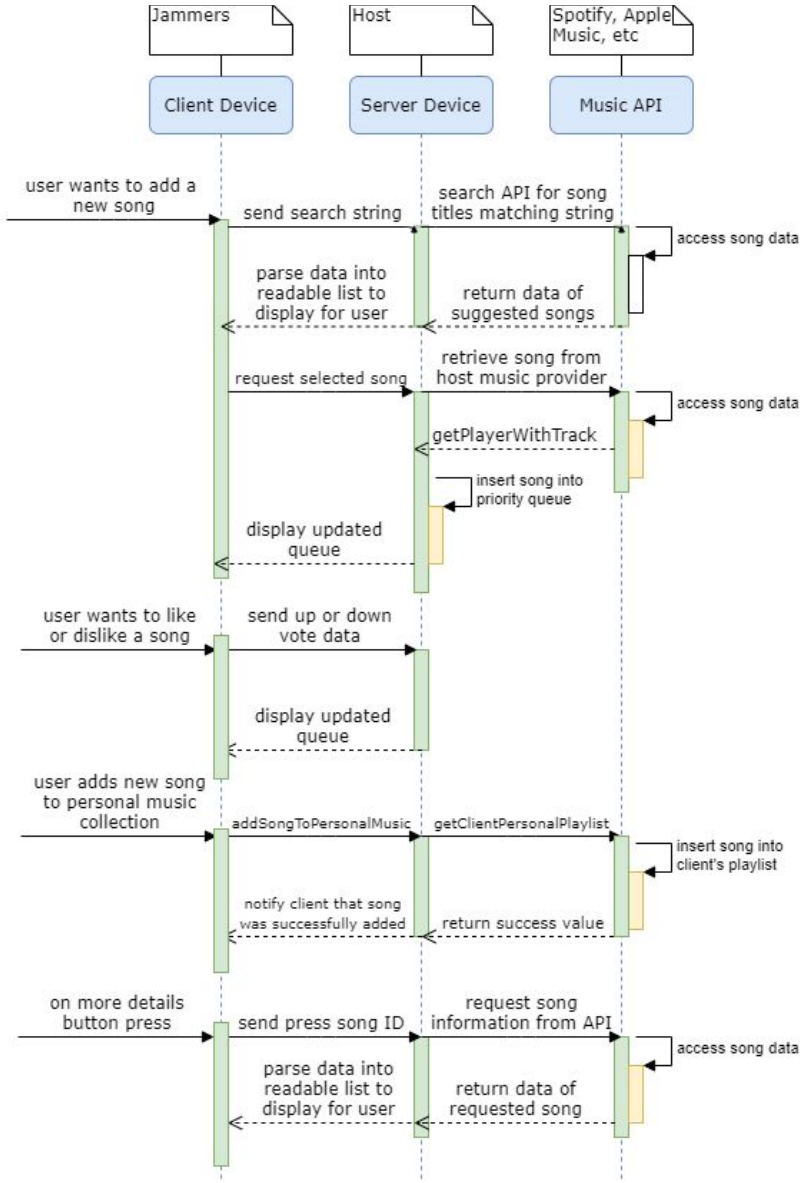
Creating, joining and leaving a queue

This diagram shows the sequence of events that take place when the user opens the application. Upon first loading, the user is presented with two buttons, join and host. When the create queue button is pressed, a request to the server is made to create a new queue. After establishing which music service the host uses and verifies their credentials, the queue is generated. When the join button is pressed, the server will verify the login credentials and queue code with the database. If they are correct, the client will be placed into the lobby. Finally, when the queue is ended, a request will be made to the server to disconnect all the clients. It will also save the queue that was created to the host's personal music service if necessary.



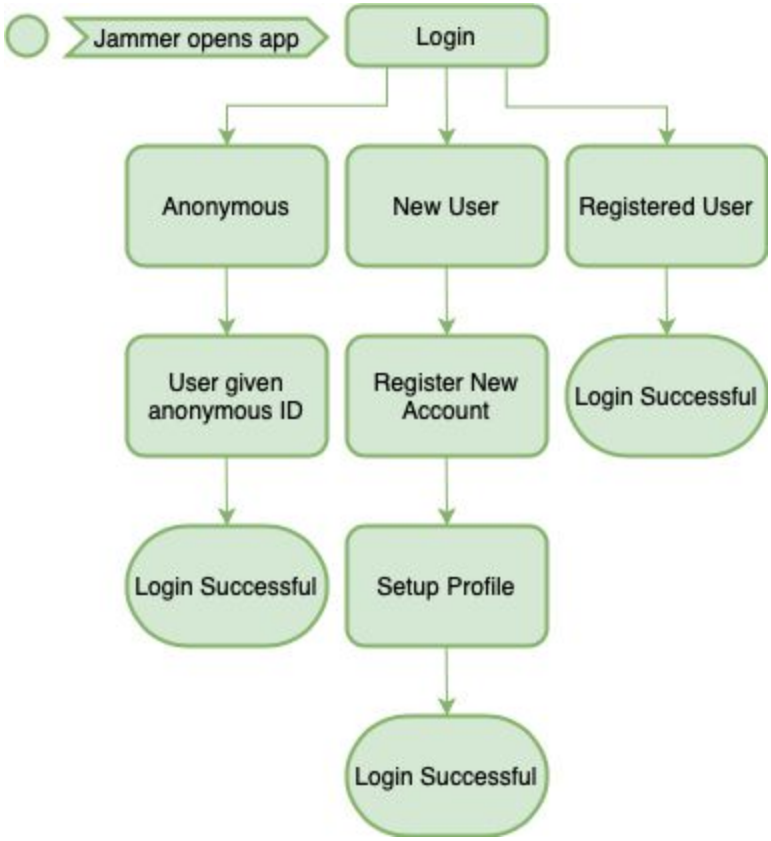
Editing and retrieving information from the queue

This diagram shows the various interactions the user can make with the queue itself. If the user wants to add a new song to the queue, a request will be made to the server. It will then send the search string to the API to look up the song. If found, the song selected by the user will be added to the queue by the server. If a user wants to like or dislike a song, a request will be made to the server and the queue will be updated accordingly. If a client wants to add a song to their personal playlists, a request will be made to the server which will make a call from the API to do so. The server will notify the user if this action succeeded. Lastly, if the user wants more details on a song, the server will request details form the API and display them for the user in a readable way.



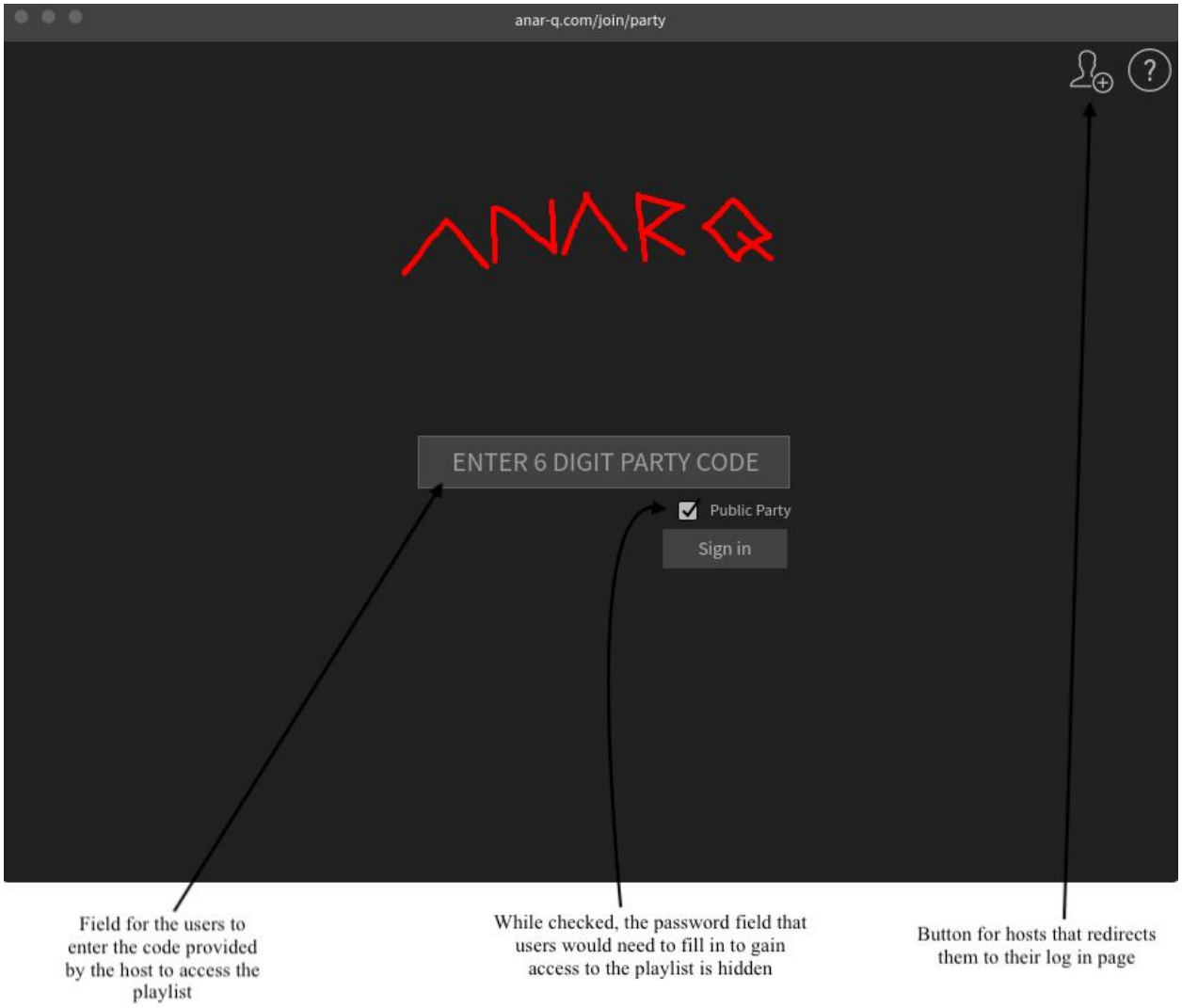
Logging Jammers into the application

This diagram shows the three ways Jammers can log into the application. If the Jammer does not want to create an account, they will not be forced to. If they decided to login anonymously, they will be given a unique anonymous ID. If the user is new and would like to register for a new account, they will be taken through the process and can set up a profile consisting of a profile picture, biography, etc. If the user is already registered, they can log in with their username and password. In all three scenarios, the user can log in successfully and begin to use the application.



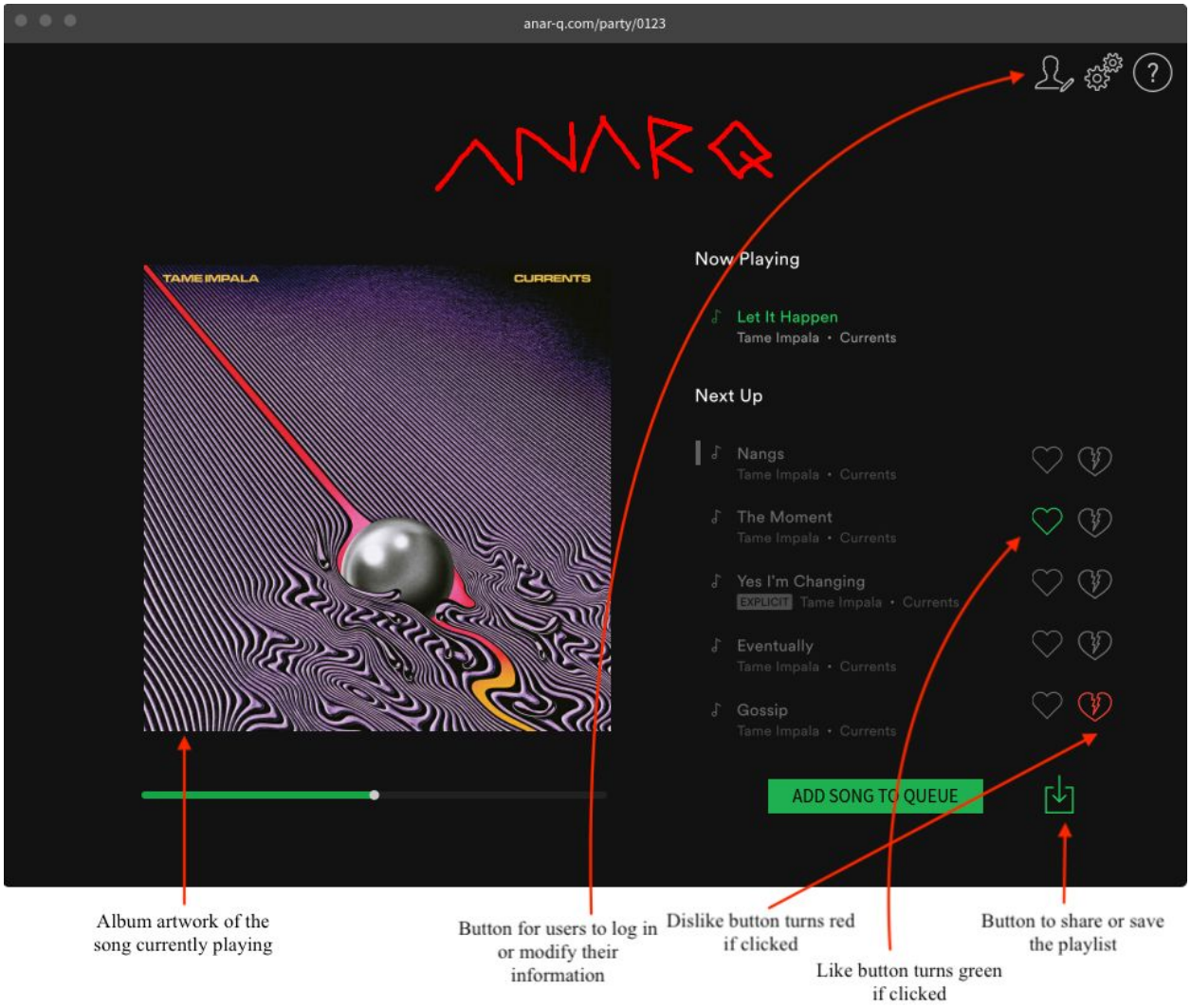
User Interface Mockups

Homepage for Jammers



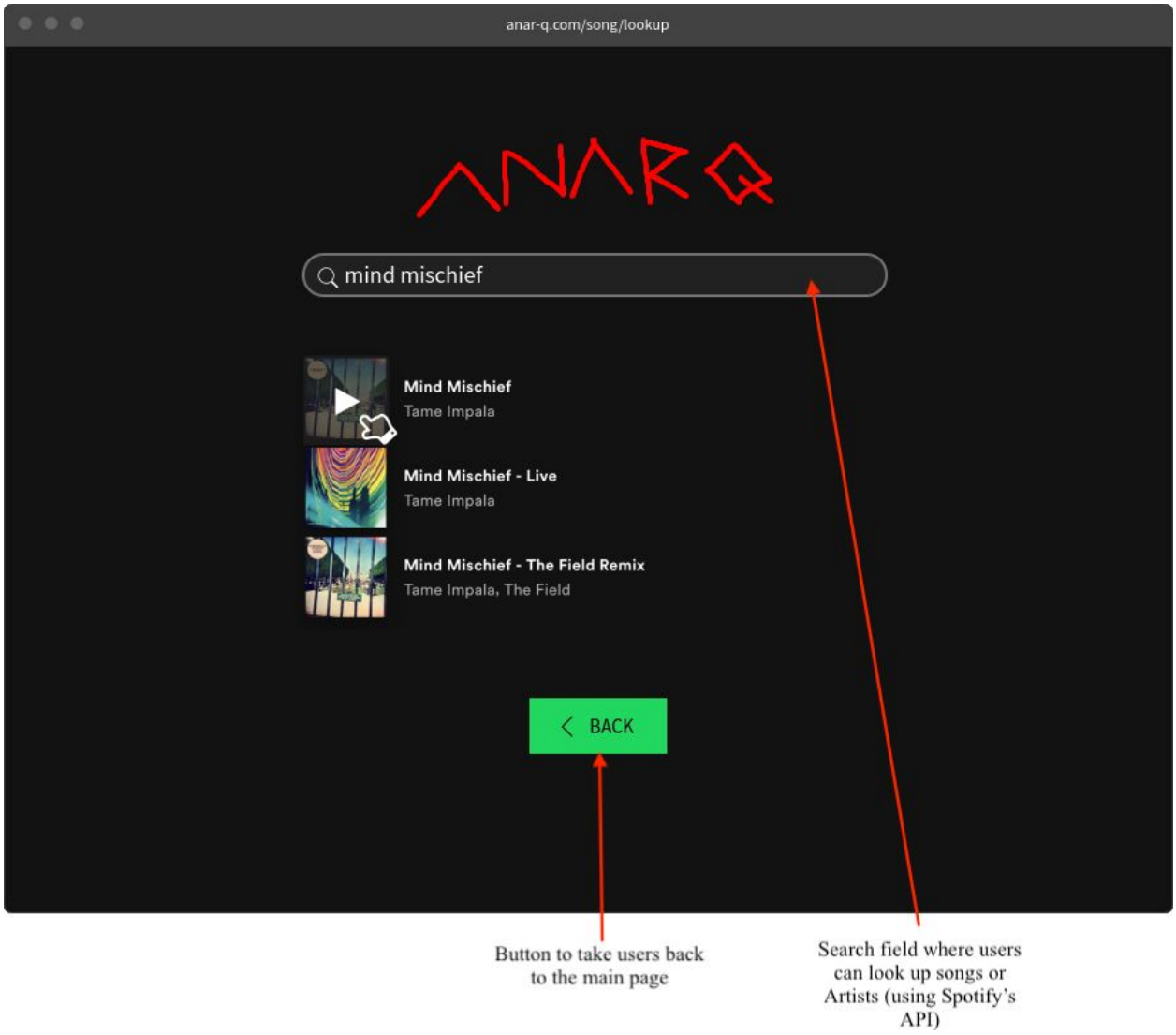
This screen mockup represents the basic design that we will implement for the initial greeting page for those looking to join a session (Jammers).

Music Session Mockup



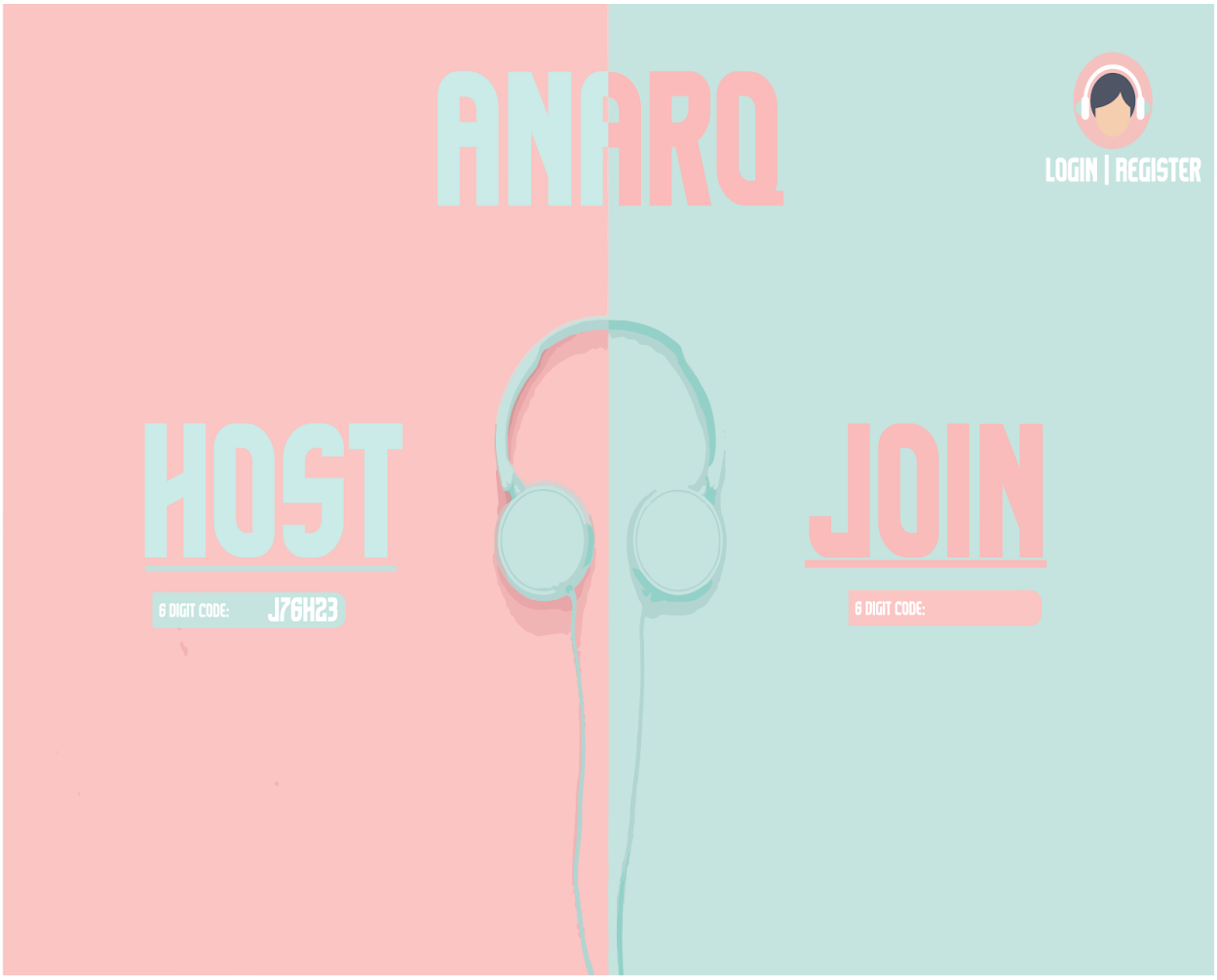
The above image is an example of what the users view of the main page would look like with explicit music allowed and the number of votes hidden.

Music Lookup Mockup



This screen is a mockup of a potential search screen that the user would be shown when they try to add songs to the queue.

Homepage Mockup



This mockup uses a sleek, modern design for the homepage, and gives users three choices — Host, Join, Login/Register. DJs can create a session, then broadcast the six digit code. Jammers can join the session using that code. While it is not required, users have the option to login or register with the service.

Music Session Mockup

This mockup gives an example of what a typical music session would look like. A “Now Playing” section is displayed consisting of the album art, song name, volume control, and what is currently up next. The opposite side of the page shows the listening party. It also allows users to search for a song and upvote or downvote songs that have been added to the queue.