# Online Collaborative Playlist

School of Engineering | Santa Clara University

**Sara Cassella**
**Alexander Polatnick**
**Kristen Ronhovde**

Santa Clara, California
December 2nd, 2016

# Abstract

Currently there is no effective and widely used musical platform that allows guests at a music-based event to express their opinions on what should be played without distracting the host or DJ. The idea of this project was to capture the mood or vibe of an event or party in order to cater to the musical tastes of everyone involved. A platform that allows for this type of expression will allow the DJ to pick the best possible music for the given event. The system that will solve this issue is a client-server web application that will allow users to request songs and upvote or downvote songs on the current playlist in order to express their opinion for the DJ and the rest of the event to see. The web application will be easy to use and will require little authentication and no downloads for the guest user.

# Table of Contents

# List of Figures

# List of Tables

# Introduction

Many social and professional events involve music where the DJ ultimately decides which songs to play. However, we believe that every attendee, not just the DJ, should have an opportunity to choose the songs they hear. Currently other similar platforms do not perform well, and do not provide an effective way for an attendee to express his or her music choices, or for the DJ to get an accurate feel for the music opinions of the group as a whole. If an event attendee wants to suggest a particular song, they can only do so by approaching the DJ directly, which is not only inconvenient for the attendees themselves but can also distract the DJ. Further, often times the venue itself might not allow attendees to speak directly to the DJ. As a result, DJs often lack feedback on their music. Consequently, they cannot cater to the musical taste of their particular audiences.

Currently, no software platform on the market allows an audience to quickly and easily give their opinions and feedback about what songs they want to listen to at a particular event. Although popular web applications such as Spotify, Pandora, Soundcloud, and Youtube have implemented algorithms that form playlists to cater to listeners based on a history of past song choices, they do not take a live feed of data from an audience during an event. Because of this, our market is lacking an efficient way for the DJ to tap into the “mood” or “vibe” of his or her audience.

Our client-server, web-based solution allows an audience to suggest specific songs for the DJ to play, and allows audience members to give feedback on songs suggested by other users. An attendee can sign into the event, recommend songs to the DJ, and upvote or downvote songs found on a live feed. A DJ has the ultimate authority over what music is played, but they may find it helpful to have a list of suggested songs from his or her audience. Instead of speaking with multiple event attendees and noting down their song requests, a method that can get chaotic, a DJ can be more efficient with his or her time and quickly refer to a list to get song ideas that people want to hear, along with feedback to the songs that he or she has selected. Therefore, the DJ can cater to his or her specific audience and play songs accordingly. The DJ and audience can pick from a variety of songs, which can be found on Spotify or his or her iTunes library. Many people own smartphones that will be compatible with this application making it available to large audiences and eliminating the need to provide any hardware for users. This mobile application allows for an ease of expressing musical tastes or song choices using a platform that will suggest songs that will please the majority of the audience, and make the music at an event memorable by all.

# Requirements

Here we explain the requirements that outline our solution. They have been broken into 3 categories: functional requirements, non-functional requirements, and design constraints. Functional requirements describe characteristics of the system that are absolutely mandatory in order to meet the demands of the customer. Non-functional requirements are requirements that are not mandatory to meet the customer request, but would provide a more functional solution. Design constraints are the limits surrounding our ability to create the system.

**Functional Requirements:**
Critical:

- Host must be able to play music off of his/her computer through the web application
- Host must be able to create a playlist on the web application
- Host has last say in what song is actually played at any given moment
- Host must be able to set up an event on the application
- Guests must be able to recommend songs as well as upvotes and downvote them
- Guest must be able to authenticate themselves in event with one simple password

Recommended:

- Host is able to authenticate Soundcloud account through the web application
- System will allow for guests to search through all Spotify libraries that are available in addition to the host's libraries
- System has have an easy to use user interface

Suggested:

- Users can take part in a chat including all attendees who have been authenticated
- The web application will have a setting that allows for music to be automatically chosen based off of popular choice of guests, rather than DJ selection
- The host can remotely log into their event using his or her mobile device

**Non-Functional:**
Critical:

- The system will be able to handle many users at a time
- The system must be mobile responsive and have a guest friendly user interface
- The system must be responsive and update user data regularly
- The user interface will be self explanatory and simple

Recommended:

- User data is updated within 5 seconds of user action

**Design Constraints:**

- Must be a web application
- Must run on Safari, Google Chrome, and Firefox
- Must work on iPhones and Android devices
- Host must be able to authenticate different music providers through the application

# Use Cases

Uses cases of the system note the different roles that users or actors of the system take on. Each use case documents the actor, pre-conditions, post-conditions, steps, and exceptions of each scenario. Figure 1.1 illustrates which actors are involved in each case.

**Create pin**
Actor: Host
Pre-Conditions: Event does not exist, no one can access it.
Steps: -Host authenticates with music providers.
-Host sets the location of the event.
-Host creates the initial playlist.
-Host chooses event PIN.
Post-Conditions: Event is created, Host can give out the pin number to guests.
Exceptions: N/A

**Enter pin**
Actor: Guest
Pre-Conditions: Guest cannot access event.
Steps: -Guest uses PIN to authenticate event.
Post-Conditions: Guest can access event and perform all actions.
Exceptions: N/A

**Search and Request a song**
Actors: Guest and Host
Pre-Conditions: Song is not requested.
Steps: -Guest searches for a song.
-Guest selects song.
-Song is added to request queue.
Post-Conditions: Song is found and requested.
Exceptions: Song is not provided by the music providers used by the host, so the song cannot be requested.

**Approve or Reject Song**
Actor: Host
Pre-Conditions: Song is populated into an “Add to Playlist?” list, the song is not yet added to the main queue.
Steps: -Host removes song from playlist.
-Host allows song to be added to “Up Next” playlist.
Post-Conditions: Song is either added into the main queue to be played or is rejected and deleted.
Exceptions: Request mode - the playlist is created only by user requests.

**Upvote or downvote songs**
Actors: Guest and Host
Pre-Conditions: Song entry has a zero. No upvotes or downvotes.
Steps: -Guest upvotes a song they like.
-Guest downvotes a song they do not want to hear.
Post-Conditions: Song moves up in the queue if it is upvoted more than others songs in the queue. The song is deleted if it receives more than 5 downvotes.
Exceptions: N/A

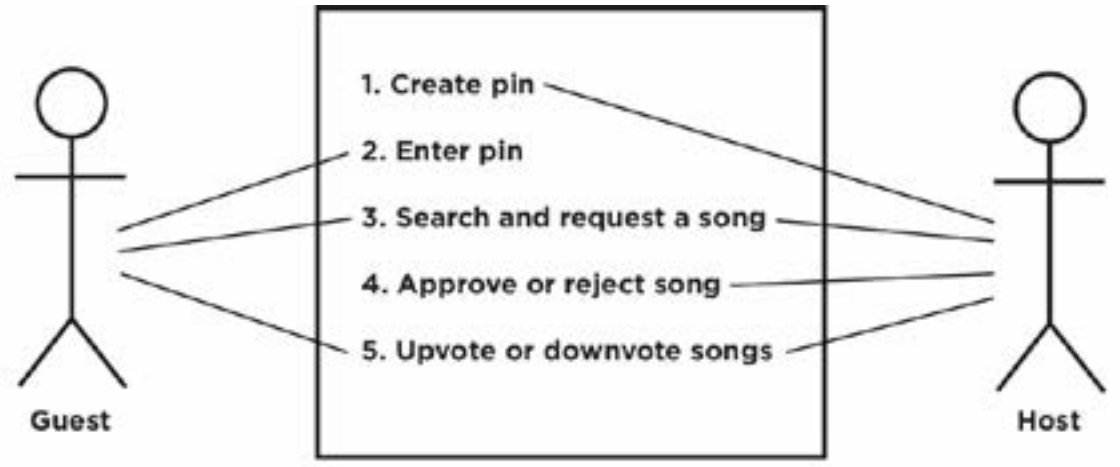


Figure 1.1. Use cases.

# Activity Diagram

Our Activity Diagram is broken into two different actors. For the Host actor, which is shown in figure 2.1, the host will first log in to the system and create a playlist for the party or event that is being hosted. Once the playlist is created, the host has the option to either continue managing the playlist, or leave the queue to be determined by the guests. If the host chooses to let the guests determine the playlist, they no longer have to do anything, and the application will sort the queue based on user requests.

The second actor in the Activity Diagram is the guest to the party or event, shown in figure 2.2. The guest will log in using the same login interface as the host, however they will have a seperate PIN that will take them to the guest user interface where they can either search for a song to add it to the playlist, or they can look at the songs already in the queue and upvote and downvote the songs to determine the order for the next song to be played.

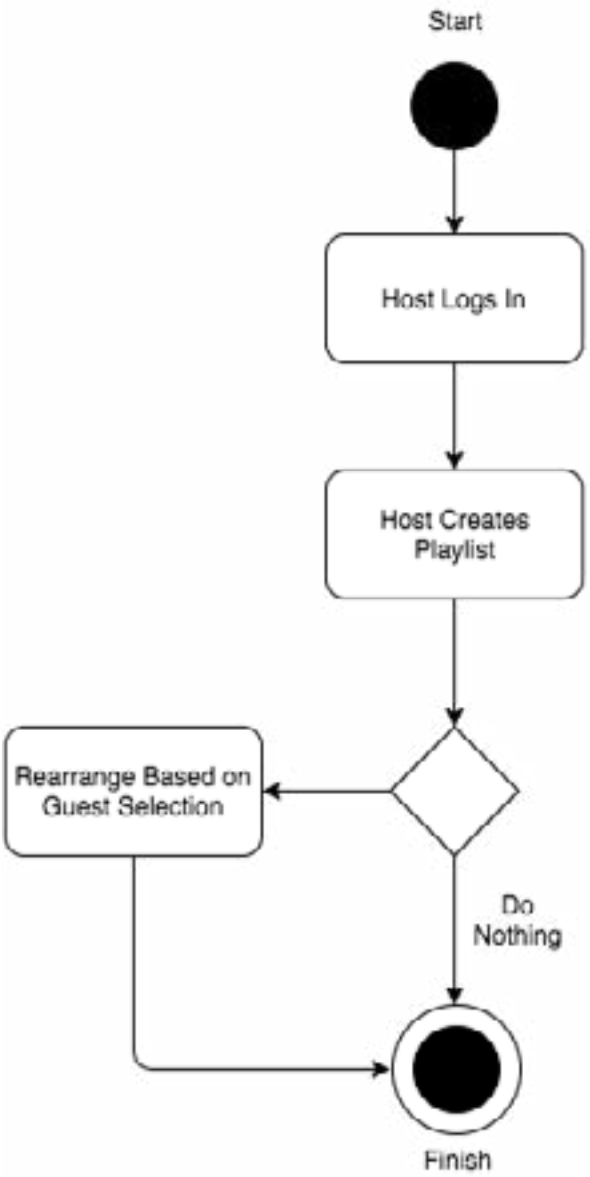


Figure 2.1. Host Activity Diagram

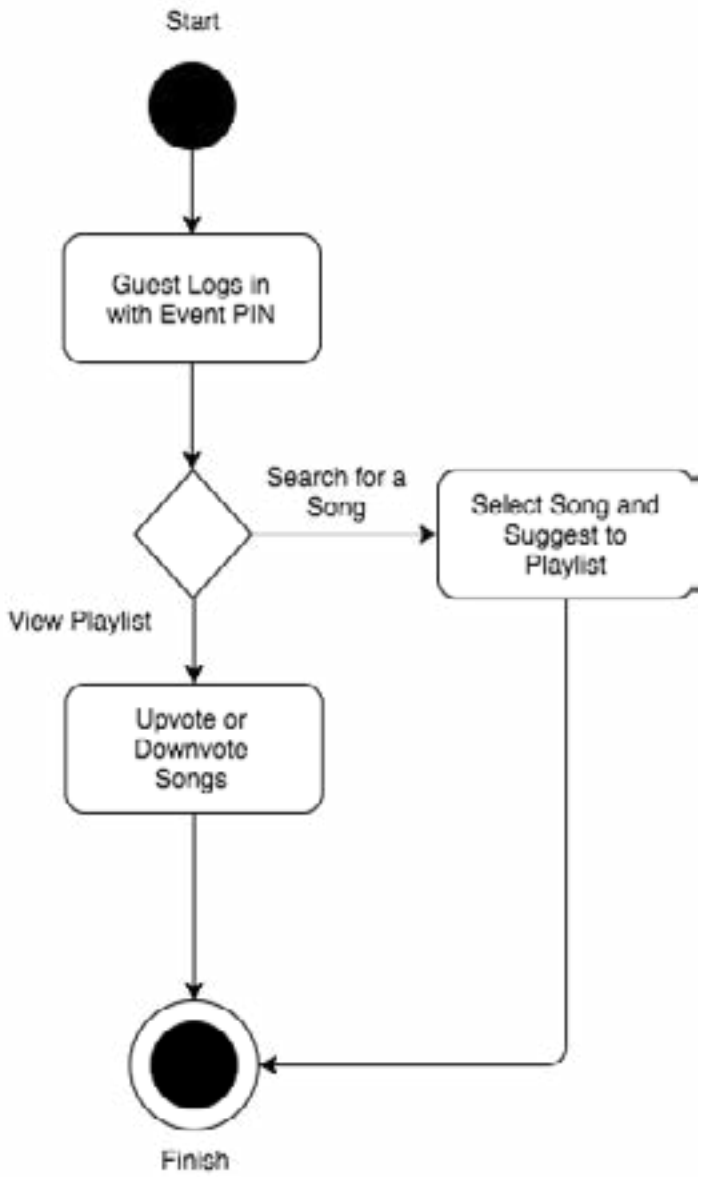


Figure 2.2. Guest Activity Diagram

# Conceptual Model

We want to keep our user interface very simple. Firstly, we wanted to keep the homepage very concise by only including a pin number to enter. Once a pin is entered, the main tasks that we want users to perform include searching for songs, requesting songs, upvoting and downvoting them, then adding them to the queue. Because searching and requesting songs both rely on a search bar, we have placed the search bar front and center on the page and made it white to improve contrast. We have also alternated shades of grey in between songs on the playlist to include contrast and readability. There are numbers next to each song on the playlist so that it is clear which song is first on the list and which one is last. We have selected a very dark interface because it is more aesthetically pleasing in low light situations, such as parties or concerts. The first four figures are the mobile version of our web application, and the last figure is the desktop version. We want to have two different viewports depending on what device the user is using to access the application.


**Figure 3.1**. Log in page (Mobile)

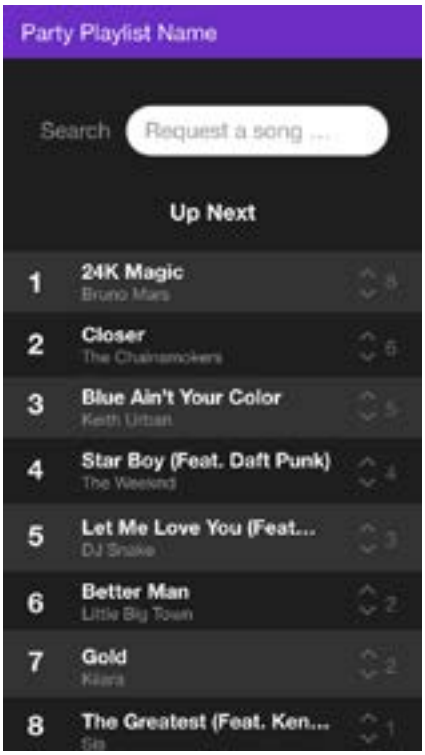


**Figure 3.2**. Guest Home page (Mobile)

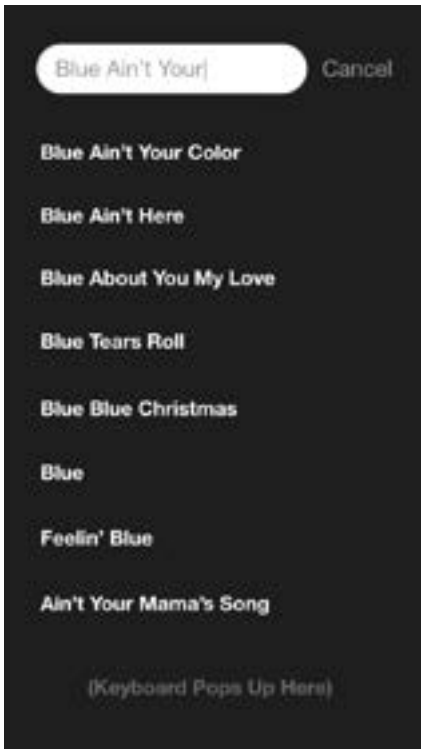


**Figure 3.3**. Search song title (Mobile)

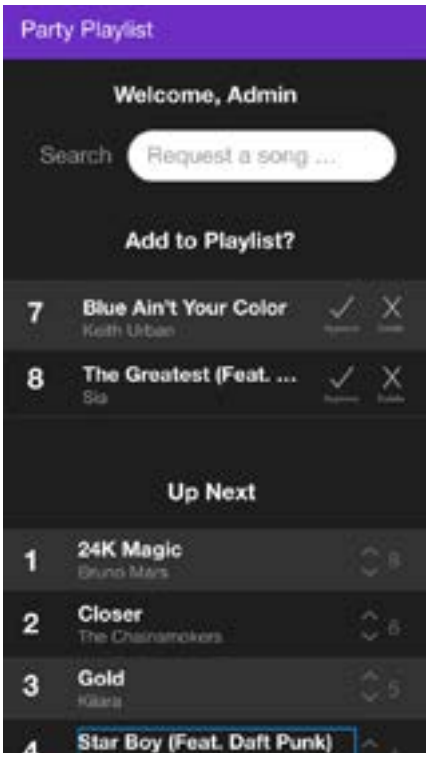


**Figure 3.4**. Admin approve or reject song (Mobile)

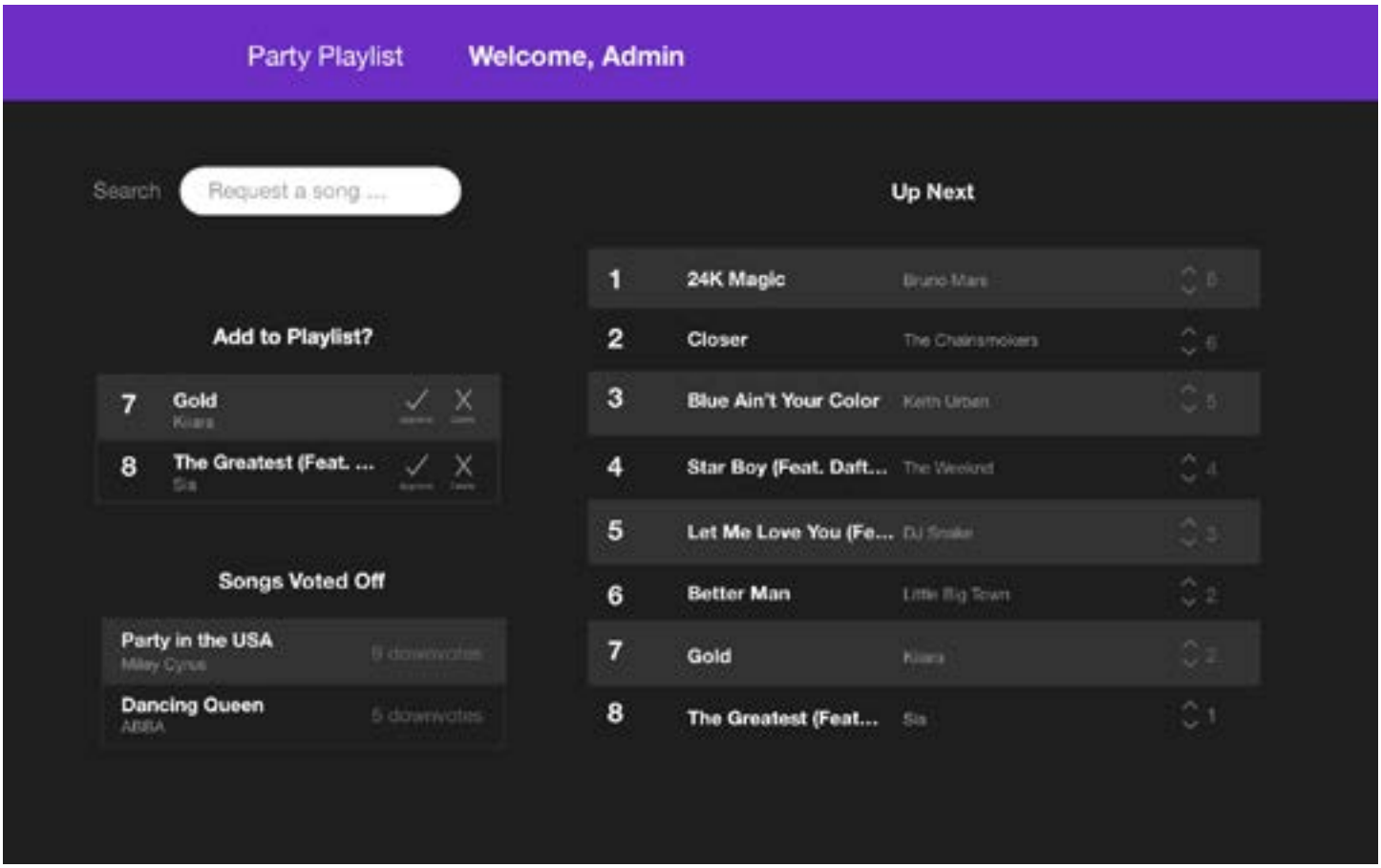


**Figure 3.5**. Desktop View

# Technologies Used

The following programming languages and technologies will be used to create and develop our system:

**HTML**

A markup language used to help define key foundational elements for our webpage such as organizational and content aspects.

**CSS**

A styling language that will be used to control the design and presentation features of our webpage.

**Javascript**

A programming language that controls the behavior of our webpage, such as process the user's entered form information.

**Bootstrap**

A front-end web programming language used to make web pages responsive, increase the aesthetics features of a webpage using vanilla CSS and other features used to create a fluid flow within a web application.

**Adobe Experience Design**

An application that allows UX designers to design and prototype web applications and mobile apps easily.

**Design Center Web Server**

To host our Web System.

# Architectual Diagram

The system will implement a client-server architecture. This architecture consists of a computer server in order to host the web application. This will start with the use of the Santa Clara University Design Center computer server but may need to be setup by hosts in order to conduct their own events. The clients will typically be mobile devices with access to the internet. More specifically, the system has a blackboard client server architecture, which means that the clients, or mobile devices in this case, are “dumb” or merely interact with the server in order to request data or to change it. The server is considered “smart” because it does all the updating and communication of data for all of the relevant clients. This architecture suits the system the best because it lays the burden of a user only upon the host who will potentially be hosting the server. As a guest of an event, any phone that can merely access the internet can be used as a functioning client with ease. This also simplifies the authentication process from the standpoint of the guests. They merely need to sign into the ID of the event they are attending rather than needing to authenticate with one or potentially multiple music providers.

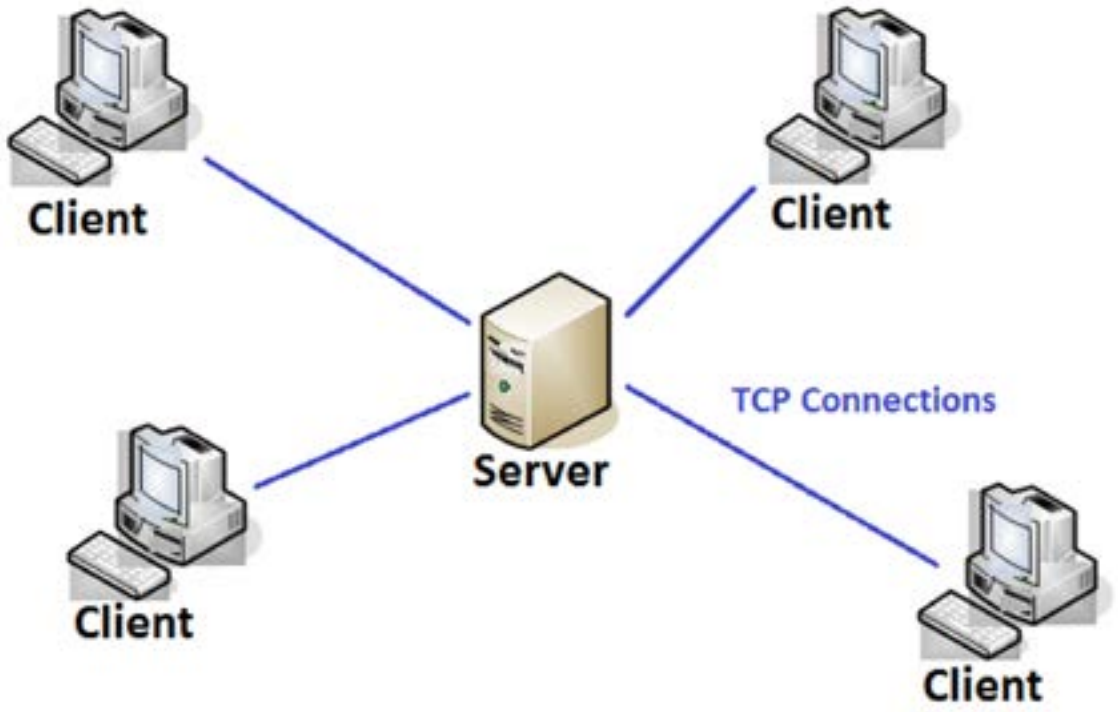


Figure 7.1: Architectural Diagram

# Design Rationale

The rationale behind making the system a web application relates to the idea that the system should be easy to access and use for the guest or attendee at an event. The guest will be using his or her smart-phone in order to access the system, so there were a couple of options for implementing our system. Instead of having it be a web application the system could be a native application to the phone. We decided not to choose this option because when a guest shows up to an event they are usually not interested in having to download the application and wait to authenticate. Instead they would more likely attend the event as they would without using the system at all, maybe suggesting songs to the DJ in person. The idea of having the web application is that users do not have to download anything, rather they can quickly access the web application online and authenticate with one simple password, chosen by the host of the event.

The negative of this design choice is that the host must be running on a server as well as authenticate the different music providers he or she is planning to pull from during the event. Although this can be slightly tedious, it is a standard procedure that the host sets up an event or party to make it easy on the guest for their enjoyment. This ease of use for the guests is why the system has a client server architecture with “dumb” clients. In order to make it as easy as possible for the guest to get started using the system, the clients in this architecture simply message the server in order to access its data. The clients send feedback to the server such as suggesting songs from the host’s library but does not save any data itself nor can it communicate directly with other clients. This architecture is more specifically called a blackboard client server.

The system is also an object-oriented architecture. Creating classes allows the system to abstract data so that it can be used in the user interface seamlessly. Because the system is pulling from multiple music providers, the way we extract the data from them is apt to different. The system needs a way to take in all this data and present it in a similar fashion. By using classes to interpret data from each provider we can have functions that if interpreted by another class can abstract the data that is being extracted. Say we have a “song” class. This class would be the parent class of a “SpotifySong” class and a “SoundcloudSong” class. This means that the two later classes would inherit the functions and variables from the “song” class. For example, we might have the functions “getName” and “getLength” in the song class. Each of the later two functions would have those functions, but the way they went about receiving that information about a song will be different. Therefore the “song” class is left vague and each of the child classes will define these functions in their own ways to appropriately return the correct data. The “song” class is an

abstraction of the underlying data which allows the user interface to not have to worry about any of the underlying classes, promoting unity of code and ease of implementation.

In terms of how the system will work, we decided to cater the system to the user's requirements. The system allows users to suggest songs into a suggested queue (separately from the selected playlist queue) because event attendees want to have a say in the music that is playing, but we do not want to take away the right of the DJ to select the songs that he or she decides on playing. Say a guest suggests an inappropriate song for the setting. That song should not be automatically in line to be played, and the DJ should have the right to remove the song from the list altogether. This is also why the system includes upvotes and downvotes. Not only can the DJ get an understanding on which songs should be played, if enough people downvote a particular song, it will be removed. The system may also include a setting where the DJ can allow the most upvoted songs to automatically be at the top of the playlist queue in order to appeal to the largest group of guests.

The system takes from multiple music providers. The system should not limit the user to selecting songs from one or another because alone they are limited. The system will ideally use Spotify, Soundcloud, and iTunes library as a starting point because together they amount to enough available music. The negative to using multiple providers is that it will be difficult to implement and possibly authenticate.
During the entire building process of our system we will be using github for version control and documentation as it helps us save and keep track of our progress, as well as back track if there was a mistake that causes us to take a step back.

# Development Timeline

The development timeline indicates when each task of completing the system will be done by and which team member(s) will do it. Each team member has a specific color that indicates what they are assigned and when they will start and finish each assignment. The development timeline helps the team stay on track and meet the appropriate deadlines in order to successfully complete the system.


Table 1: Development Timeline

# Risk Analysis Table

Table 2 refers to the different risks the team runs while attempting to complete the project. Each row indicates a certain risk, a description of what the risk entails, probability of the risk happening from 0 to 1, the severity of the risk which ranks from 1 to 10, the impact of the risk which is the probability times the severity, and finally two mitigation strategies to avoid the occurrence of the risk altogether.

| Risk | Description | P | S | I | Mitigation Strategies |
|---|---|---|---|---|---|
| Lack of Coding Knowlegde | Team takes on tasks that are out of scope of understanding or too difficult to perform | .7 | 8 | 5.6 | Research the languages we will be writing the system in.<br>Work together on each section to help teammates along with implementation. Meet with advisor consistently. |
| Lack of Time Management | Team does not leave enough time to complete time consuming tasks | .5 | 7 | 3.5 | Stay in constant contact with one another through group texting.<br>Hold each other accountable for deadlines. |
| Illness | One or more of the teammates gets sick and cannot work on the project for a certain time period | .6 | 4 | 2.4 | Stay Hydrated and get a lot of sleep.<br>What hands regularly and eat well. |
| Lack of Cooperation from outside APIs | APIs from music providers do not fit well with system or are too difficult to implement | .5 | 5 | 2.5 | Do a lot of research on each plausible API.<br>Choose to implement usable APIs. |
| File Loss | Loss of progress in code and documentation of system | .1 | 10 | 1 | Use github for versioning and saving.<br>Communicate with teammates to organize files appropriately. |
| Ambiguous Requirements | Either a lack of understand about what the customer wants out of the system or a failure to meet the requirements we have set out for ourselves | .3 | 7 | 2.1 | Communicate with the customer consistently.<br>Organize and checklist to make sure each requirement is successfully met. |
| Miscommunication | Lack of communication between teammates causing less time to meet up or aspects of the project to be incomplete | .4 | 7 | 2.8 | Communicate with team consistently through group chat and meetings.<br>Meet with advisor consistently. |

Table 2: Risk Analysis

# Test Plan

The testing of the system will be done incrementally as the system is being implemented and then continued heavily after the implementation stage. The structure of the system is being put together (making the classes or establishing a connection from the client to a server for example) each aspect of the system will be white-box tested so to move on with a stable foundation. As the user interface is created it will be looked over by not only the teammate but a select chosen group of people with a knowledge of computer science such as our advisor Darren Atkinson to receive feedback. After receiving and implementing feedback, we will show a newly made user interface to friends or potential customers of the system to receive their feedback.

Once the system is for the most part implemented, we will enter a testing phase of the system building process. Here the team will start with alpha testing of the system, which inquires testing as many use cases as possible so to make sure the system is in check. Any errors found will have to be fixed and re-implemented. After the alpha testing there will be beta testing where the customer is testing out the full system. Ideally we will implement this at a large party of users so to see how the system works with many users at once. Once receiving input from the customers as the team will once again go back to re-implement anything that needs change.