Essence

Intelligent Playlist Generation for Groups

FRANCESCO MACAGNO and CARLY ROBISON, California Institute of Technology

Essence is a service intended to fill the use case of a group of people all wanting to listen to music from the same system. Existing systems are not designed to keep track of multiple people's preferences.Essence does this by allowing people to request specific songs to play, then dsigning a playlist to cater to everyone's needs. This involves ensuring that everyone gets to hear music of their preference periodically, while also creating a cohesive queue of songs which makes sense and sounds appealing.

CCS Concepts: • **Information systems** → **Multimedia streaming**;

ACM Reference format:
Francesco Macagno and Carly Robison. 2017. Essence. 1, 1, Article 1 (June 2017), 5 pages.
https://doi.org/0000001.0000001

1 PROBLEM

This basis for the need for this program comes from the lack of existing group playlist generation and players. Classically, a dj is employed to cater the music for a party. This consists of reading the crowd, determining the music which best fits the situation and then playing it. The dj needs to select music that people enjoy and fits the situation, but also music that makes sense when played together. Hiring a dj however is expensive, not to mention unmerited for every day circumstances. Thus, Essence is intended to fill a similar role to that of a dj, but instead of for parties, for everyday use. The music during parties is a limited set of songs when considering the entire basis of written music. It is generally not music one would listen to on a regular basis or for all kinds of events, such as simply working or relaxing. Thus we wanted to build a system that allows groups to play music for situations where a dj isn't merited or relevant.
Thus, what we wanted to create a service with certain features:

- Music by Request:
  Our system is based on people making requests for specific tracks to be played by the system. This method operates on the assumption that the user best know what music they would like to hear at any given moment. Thus, the system allows its user to designate specific tracks which they would like to hear, and then use this data to create a playlist. This playlist does not have to contain any of the suggested tracks: music can both be removed and added by the system to achieve the best end result, but it gives the system a basis for determining who is currently listening and the mood they are currently in.

- Web-Based:
  An important part of all programs these day is being able to function on a variety of platforms. The easiest way to do this is to build a web-app, a service based on a website frontend and a backend running on a server. This is advantageous because it can easily be purchased as a cloud service, or hosted on a personal or organizational server. It could even be hosted locally for just a single home network, and this can all be done using the exact same implementation, avoiding the problem of managing separate code bases. We also wanted it to play music by streaming it to the user from the server. The benefit of this is that this allows multiple different systems to simultaneously play the same track, allowing for distributed speakers across an area. It also puts the minimum requirement on the client, and allows them to play the stream using whatever service they want to.
- User Based:
  In order to adequately play music that everyone will enjoy at one point or another it is necessary to keep track of who requests which songs. The system has to have some sort of user system, but the specific implementation doesn't matter, as all that is necessary is that a request can be tied to a user.
- Playback Control:
  Another important aspect of this system is that it needs to be controllable from the web interface. Since the system may be set up remotely, we want a user to be able to control speakers in a different area. Thus it should have controls allowing the user to skip songs, change the volume, and start or stop the stream.
- Intelligent Playlist Generation:
  The most important aspect of this system is that it be capable of doing more than simply adding tracks to a queue in the order they are requested. The system needs to be able to create an ordering which is fair to everyone listening, and curate a listening experience which is well ordered and makes sense from a artistic point of view.
- Historical Playlist Generation:
  Having to manually request every song is difficult, so the system needs to be able to generate playlists from past history. BY mixing this with intelligent generation it is possible to create a playlist which matches requests and suggests tracks to go between them, varying the number added based on how many tracks are specifically requested.
- Track Storage:
  The system should store all the tracks that are uploaded so that they can be used again without the user having to go find them on their computer on youtube.

2 IMPLEMENTATION

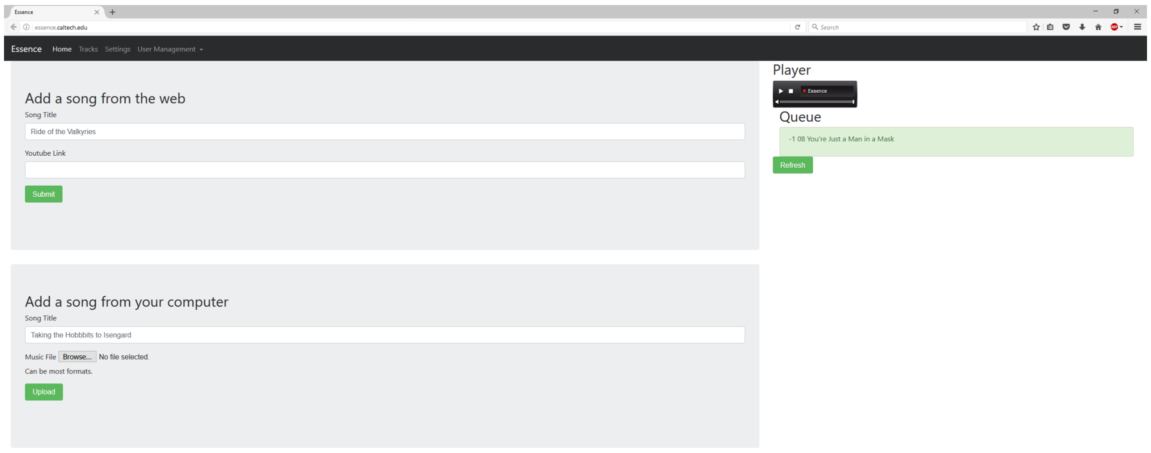
2.1 Frontend

The frontend is built using standard html and javascript, using Bootstrap for visuals. Bootstrap is a visual framework for html and websites. To pull data from MongoDB a plugin was used for php, and php was used to upload files and data to the database.

2.1.1 The Main Page. This page has sections for uploading tracks, and has an embedded player and the current playlist on the left side.

2.1.2 The Tracks Page. This page shows all the tracks that have been uploaded so far with a button to request them again.

Fig. 1. The main page



Add a song from the web

Song Title: Ride of the Valkyries

Youtube Link:

Add a song from your computer

Song Title: Taking the Hobbbits to Isengard

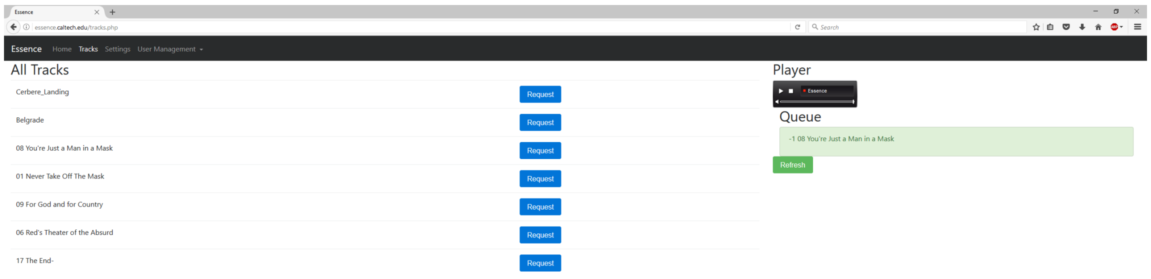
Music File: No file selected.

Can be most formats.

Queue

-1 08 You're Just a Man in a Mask

Fig. 2. The tracks page



All Tracks

Cerbere_Landing	Request
Belgrade	Request
08 You're Just a Man in a Mask	Request
01 Never Take Off The Mask	Request
09 For God and for Country	Request
06 Red's Theater of the Absurd	Request
17 The End-	Request

Queue

-1 08 You're Just a Man in a Mask

2.2 Back-end

The back-end is implemented in Java, compiled into a runnable jar which includes all of the required dependencies. The project is managed using Gradle, so it is simple to create the jar from source, however the jar will also work standalone, meaning that a person can just download the binaries.

The Java application, called the Essence Runtime, processes uploaded tracks, generates updated playlists as more requests are made, and sends the track data to the Icecast server. Each of these is run in its own thread in order to allow for the minimum delay between playing tracks and sending data, allowing there to be virtually no lag from the runtime.These threads all communicate using MongoDB: as each generates data it is uploaded to the database, allowing the other threads to access it as well. This also makes maintaining state between runs of the system easy, as the threads simply pull data like they normally do when started.

When the system is instructed to stop either by command or by a system signal each of these threads is gracefully allowed to stop.

3 PLAYLIST GENERATORS

The runtime calls a customizable program to generate the playlist, given current requests, the last calculated playlist, and in general all the data available in the database. There are some scripts packed with the runtime which are always available, and you can also place programs into the "modules" folder. The system can then run these to get the data. Essence also includes api's to make building these easier. If you are using Java, one can simply include the Essence runtime. Other languages will be eventually.

Some current and planned Playlist Generators:

- In-order: A generator which creates a playlist from the order of the requests by time.
- Random: This generator randomizes the requests and creates the playlist from this.
- Alternate-Users: Go in order, but every person who has requested a track gets to hear one before a person gets a second one.
- Similarity: Play requested tracks which are similar together.

3.1 Network Graph Generator

The most complicated generator essentially combines all of the desired methods of generating playlists. It both ensures that tracks are played which are similar, adds new tracks if there are too few requests, and makes sure everyone gets to hear tracks of their preference. To do this, all the tracks which are in the Essence local database are passed through the Spotify api to get data regarding their content. This gives a large variety of characterizations for each track. From this, we build a fully connected graph, where each node is a track and each link has a weight which is the similarity between two tracks, by comparing their characterizations. Then, the tracks which have been specifically requested are marked, and the oldest and newest requests are marked as the beginning and ending points. Then by finding the shortest path from start to finish which goes through all requested tracks, and limiting the number of tracks which can be between any two requested ones a playlist is generated which moves between different varieties of music, hitting the requests of everyone. This can also be augmented by increasing the weights dramatically of routes which hit a person's requests more than once continuously, to prevent a monopoly on the system through uploading large numbers of preferences.

4 IMPLEMENTATION OBSERVATIONS

The implementation of audio streaming turned out to be the most difficult aspect of this project, due to the lack of good documentation on the use of services like Icecast for uses which weren't explicitly planned for in their design. This ended up in many weeks being spent on getting the streaming aspect to work, but in the end it was possible directly from java, which makes setup much easier. The process was very informative, and we learned how media streaming works, a fairly complicated system.

5 CONCLUSIONS

The construction of this project resulted in a functional basis for the total desired end result. Everything is in place, and is able to be easily built on to implement more of the desired features. While this is clearly non-ideal when compared to fully completing the project, , enough is complete that the project can continue to be worked on without fear of how it works being forgotten. All the existing code supports the new things that need to be added, so no more redesigns are necessary. It has achieved the minimum needed to function, and so can now be released and used to test new features.