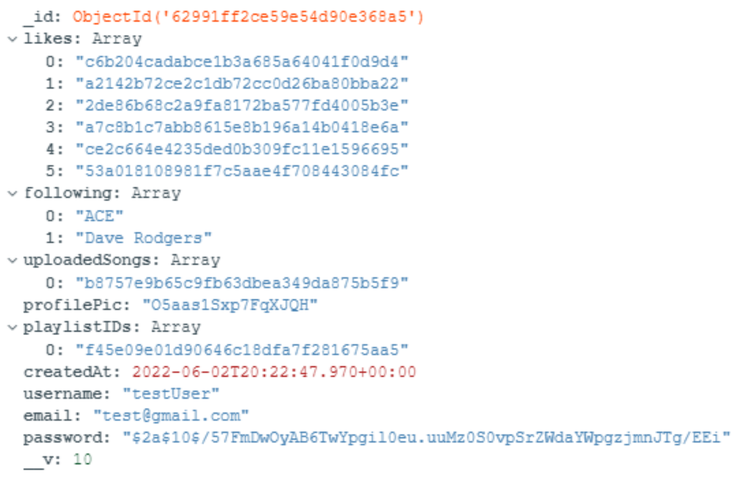
Origin Music: Final Project Report

Project Overview:

The goal of this project was to create a working music sharing platform with many of the common features associated with such applications. Examples include liking songs, adding songs to playlists, following artists, and searching for music. We hoped to introduce several distinct features that many popular music apps don't currently have, for example built in searching by audio and comment sections. This would serve to create an app that could help foster smaller creators and help them grow their passion for music into a career.

Project Design:

Database: The database chosen for this project was mongodb. Its document based system is favorable for the creation of small, metadata heavy objects. For example, a user account object can be found via a userID and then all the data associated with it can be accessed. This makes the storing of arrays of data like songIDs easy to store and load, therefore decreasing the latency of loading something like a liked songs page (Krishnan). However this design is poor for the storage of large files such as songs and images. For this, I used the extension of mongo, GridFS. This acts as a middleware for storing files into mongo by creating a metadata file that points to chunks that compose the large file. While this leads to more overhead, it also has the unique feature of allowing the data for these files to be streamed in chunks. This means that for audio streaming there is close to no buffering as the data is streamed in chunks (not to be confused with the packets that the chunks are sent with) and then can be read in chunks by the backend. Therefore audio can begin being played before it has fully loaded. Another key technology used was docker. Docker is a containerizer that runs services in a linux based vm. This allows the services to be run on any machine that has docker, essentially removing the "works on my machine problem". It also provides an easy way to manage bringing up multiple apps at the same time, for example it can initiate the backend and database in the same container under the same network, ensuring that they can communicate with each other regardless of host machine network policies.

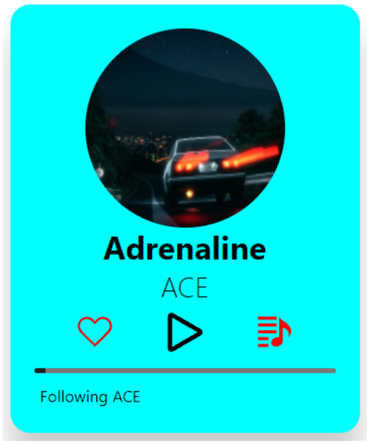


```
  _id: ObjectId('62991ff2ce59e54d90e368a5')
v likes: Array
    0: "c6b204cadabce1b3a685a64041f0d9d4"
    1: "a2142b72ce2c1db72cc0d26ba80bba22"
    2: "2de86b68c2a9fa8172ba577fd4005b3e"
    3: "a7c8b1c7abb8615e8b196a14b0418e6a"
    4: "ce2c664e4235ded0b309fc11e1596695"
    5: "53a018108981f7c5aae4f708443084fc"
v following: Array
    0: "ACE"
    1: "Dave Rodgers"
v uploadedSongs: Array
    0: "b8757e9b65c9fb63dbea349da875b5f9"
  profilePic: "O5aas1Sxp7FqXJQH"
v playlistIDs: Array
    0: "f45e09e01d90646c18dfa7f281675aa5"
  createdAt: 2022-06-02T20:22:47.970+00:00
  username: "testUser"
  email: "test@gmail.com"
  password: "$2a$10$/57FmDwOyAB6TwYpgil0eu.uuMz0S0vpSrZWdaYWpgzjmnJTg/EEi"
  __v: 10
```

Backend: The backend for the project was written in NodeJS using express. Starting with user authentication, I used a json web token stored in a http only cookie to keep track of user sign ins and authenticate actions such as uploading. Each user's password used bcrypt along with a salt to create salted hashes that were stored in the database (Arias). This protects user's passwords against data breaches. For uploading songs, new GridFS models are created. Then data is appended from multipart for requests, taking in the song file and other metadata. Multer was used to handle parsing files from the requests. The multer-gridfs-storage module allowed for the two to be connected and have the file uploaded to the database. I made the decision to only handle one file per request, and create a different collection in the database for images and songs. This was partially to reduce complexity of single requests, but also to support storing cover art with profile pictures. I took the convention of naming cover images the same ID as their songs, which also made requesting them easy. I used crypto to create a random string of bytes to ID each object by, this way songs and files with similar names could be stored without conflicting. For example, two songs with the same name can exist in the database with no issue. For searching, I implemented an edge n-grams algorithm by Johannes Fahrenkrug. This was put into a search field for each object, and then indexed in the database for text searching. This type of searching allows for results that partially match to come up, along with relevant keywords to result in matches (Elastic.co). This was how I used elastic search to do searching through the server data. The search would then return a weighted match result that would be used to return results in terms of relevancy. To implement the audio identification, I used Audd.io, which is a massive song hash library. To match the songs I used audio fingerprint hashing, which utilizes fourier transform's ability to split waveforms into frequencies, in order to hash individual frequencies and compare them between songs (Mapelli). This is then sent to the database and compared against the library of songs, which also contain the copyright/publisher info for the song. This allows for copyrightIDing of the songs within the database. The audio matching can then use the song name to do a normal text search through our database and find songs. I also implemented some simple api calls for things like liking songs and getting song data, which are used by the front end to create interactiveness. These work by finding the corresponding songs by ID and then updating their GridFS model and saving it. Two more models were created for comments and playlists. For comments it was a simple model to hold the comment section for each song, with an array of comments that had date, author, and comment text. For playlists, they stored an array of songIDs, but more importantly, were also added to a user's profile. I implemented all these features, along with an api for them to interact in the expected way. For example, adding songs/removing songs from a playlist, creating a playlist. These api calls served as the backbone for our project and did most of the heavy lifting, since this project was more based on its capabilities rather than its showiness.

```
"status": "success",
"result": {
    "artist": "Reol",
    "title": "Lost Paradise",
    "album": "Bunmei EP",
    "release_date": "2019-03-20",
    "label": "UMG - Universal Music Taiwan",
    "timecode": "04:12",
```

Frontend: The frontend of this project was done in React by both John and me. I implemented the web requests made to the backend from the front end shell. For example, the post request for uploading a song, cover image, and title. The user token was stored in browser http cookies and sent along with every request. This was how authentication for actions was handled. To handle prompting for re-login, a timer was held in localstorage that would expire when the cookie did, and thus prompt the user to log in again. The aesthetic of the app was influenced by a combination of spotify and sound cloud. Sound cloud's overall browsing experience felt better and more conducive to a small community. However, Spotify's clean and simple music player appeals to more people. Most of my work was put into the creation of the music player, since this was the main interface to many requests. I also fleshed out the functionality of the search and other library components with the api requests. For example, the likes library page needed to request the user's profile to get their likes, then request the information for each song, and then append that information to html elements stored in an array that were finally rendered and rendered. I filled these components into the shells created by the frontend team.



Future Improvements:

Security: Implementing more robust security for the application would be a must before any sort of public release. For example, my implementation of jwt cookies doesn't protect against cross site scripting attacks as for this demo it is not necessary. Also getting SSL certificates and implementing secure connections for the service to ensure that users are not at risk of man in the middle attacks. These are only a few of the security issues that come with a full scale release of such a product. Currently the data to be compromised is not very sensitive, however future expansion to monetization systems could cause much more sensitive data to be stored on the database.

UI: Due to the unfortunate group situation, the UI took a major hit in quality. This is one area that future improvements are necessary in. For one, creating a more uniform look throughout the application with buttons and text. Also bringing in more color and interactivity to the elements would make them more appealing.

Pre-loading Songs: For the site to fulfill its purpose, there needs to be music on it for people to listen too. However, no one will upload to the site if no one is one it listening. To break this circular dependency, creating an automated uploader and uploading songs from popular artists/moving songs over from others sites would give a great boost to appeal. Of course considerations to the legality of this approach would need to be considered.

Citations:

Krishnan, Hema & Elayidom, M.Sudheep & Santhanakrishnan, T.. (2016). MongoDB – a comparison with NoSQL databases. International Journal of Scientific and Engineering Research. 7. 1035-1037.

Mapelli, Francesco & Pezzano, R & Lancini, Rosa. (2004). Robust audio fingerprinting for song identification.

Mapelli, Francesco and Rosa Lancini. "Audio hashing technique for automatic song identification." International Conference on Information Technology: Research and Education, 2003. Proceedings. ITRE2003. (2003): 84-88.

Elastic.co. 2022. *Edge n-gram tokenizer* | *Elasticsearch Guide [8.2]* | *Elastic*. [online] Available at: <https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-edgengram-tokenizer.html> [Accessed 3 June 2022].

Arias, D., 2021. *Hashing in Action: Understanding bcrypt*. [online] Auth0 - Blog. Available at: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/> [Accessed 3 June 2022].