Final Report

Group 32 - Drop Guys

Link to YouTube Video:
https://youtu.be/TByy7S5CvWA

Briefly describe what the project accomplished
Our mobile app is a music sharing platform that allows users to share where they are when listening to a certain song. Users can add friends and view where they are when listening to music, along with having the ability to create custom playlists of all songs listened to in a certain area.

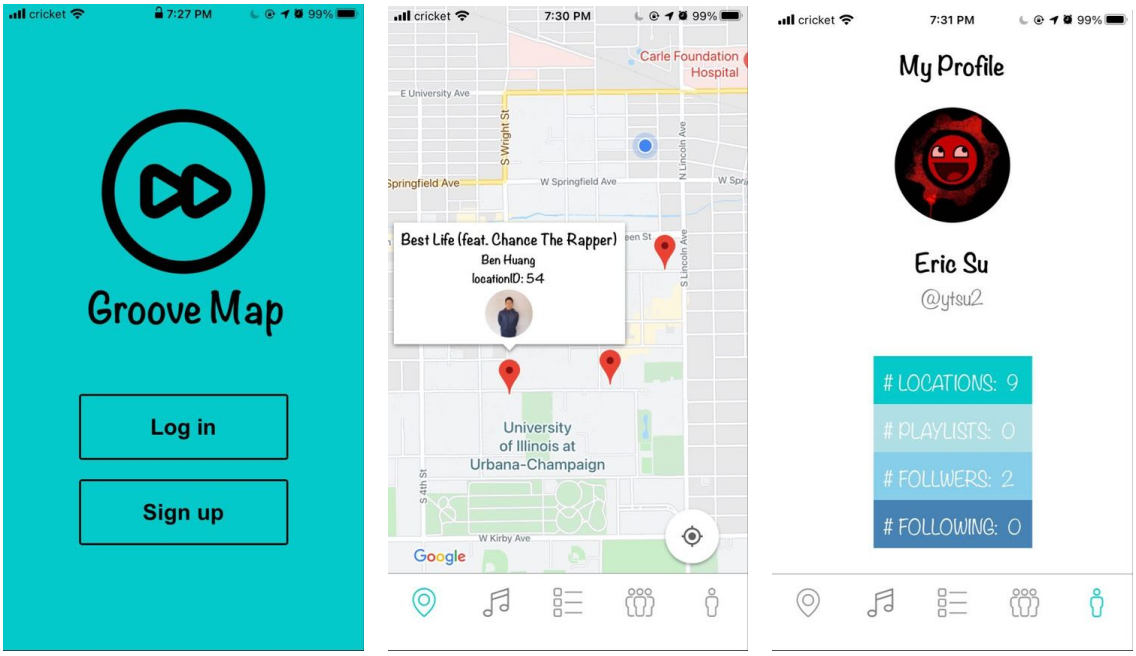

Figure 1: Screenshots of the GrooveMap mobile app

Usefulness of our project; what problem we solved
A large factor in determining what music an individual listens to is based on where they are and what they are doing. For example, someone will listen to different music when they are in the gym working out or when they are in the library studying. There currently doesn't exist a social media platform that combines location sharing and music sharing. Our platform solves this problem by allowing users to share what specific song they listen to at a certain location.

Discuss the data in your database

To make this application possible we need to store a variety of things including track name, location information, etc. In our SQL database, we decided to store all of the information related to track information, album information, location information, user information, and artist information. This data is used for the core features of our application, including adding a song to a location on the mapScreen, following another user, clicking on a specific track and listening to it in Spotify, etc. On the other hand, we store playlist information in the NoSQL database for better performance.

ER Diagram and Schema

Albums (<u>albumID</u>, **artistID**, albumName, albumUri, albumImage)
Artists (<u>artistID</u>, artistName, artistImage, genre)
Tracks (<u>trackID</u>, **albumID**, trackName, trackNumber, trackUri)
Users (<u>userID</u>, displayName, username, password, userImage)
Playlists (<u>playlistID</u>, **userID**, **trackID**, playlistName)
Location(<u>locationID</u>, **trackID**, **userID**, latitude, longitude)
follow(**followerID**, **followedID**)

Assumptions:

- A playlist can contain 0 or more songs.
- A user cannot follow itself.

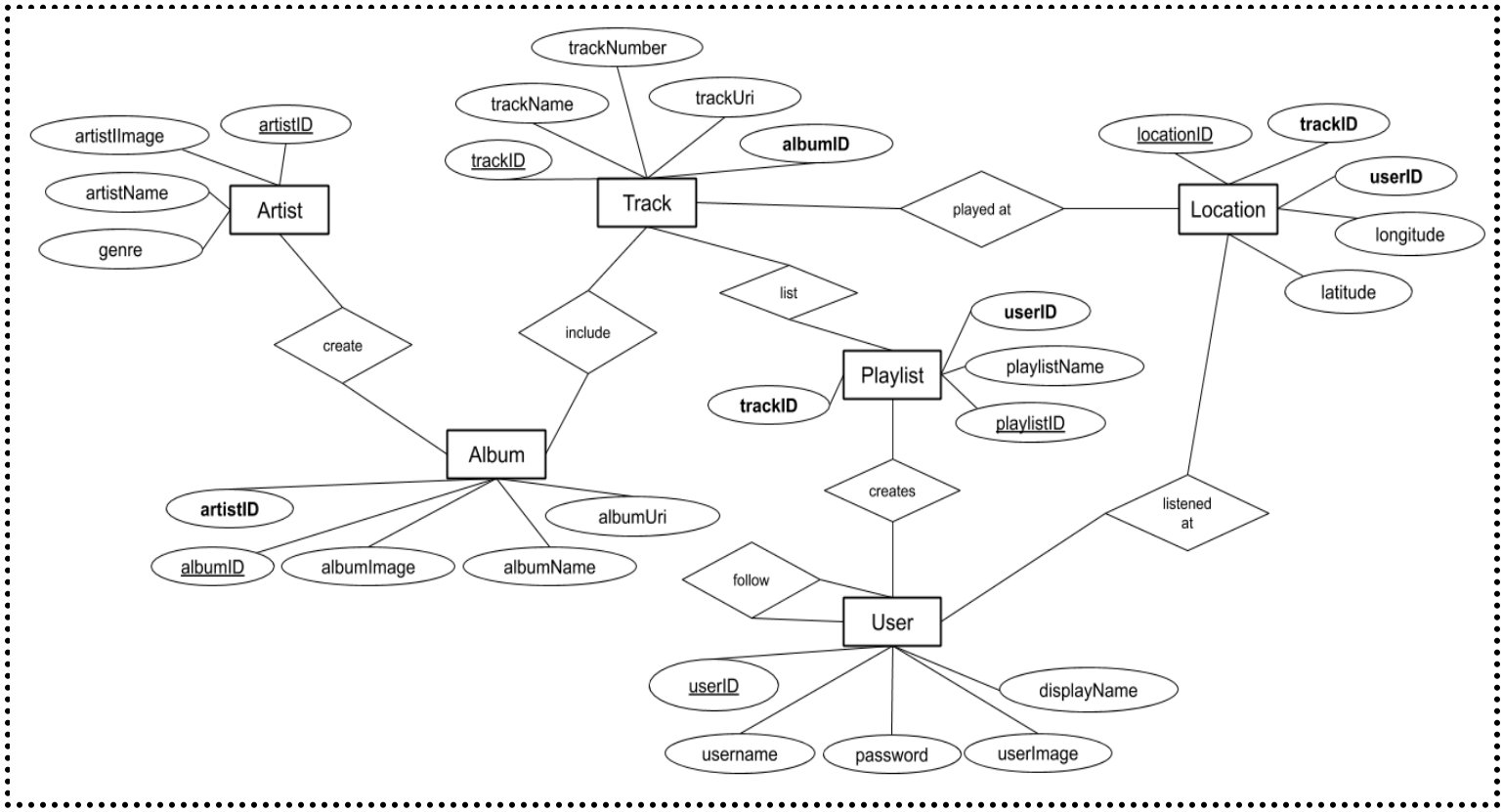

Figure 2: ER Diagram

Schema:

```
CREATE TABLE Artist (
  artistID VARCHAR(45),
  artistName    VARCHAR(45),
  artistImage   VARCHAR(45),
  genre         VARCHAR(45),
  PRIMARY KEY (artistID)
);

CREATE TABLE Albums (
  albumID  VARCHAR(45),
  artistID VARCHAR(45),
  albumName     VARCHAR(45),
  albumImage    VARCHAR(45),
  albumUri VARCHAR(45),
  PRIMARY KEY (albumID),
  FOREIGN KEY (artistID) REFERENCES Artist (artistID) ON DELETE
CASCADE
);

CREATE TABLE Tracks (
  trackID  VARCHAR(45),
  albumID  VARCHAR(45),
  trackName     VARCHAR(45),
  trackNumber   INTEGER,
  trackUri VARCHAR(45),
  PRIMARY KEY (trackID),
  FOREIGN KEY (albumID) REFERENCES Albums (albumID) ON DELETE CASCADE
);

CREATE TABLE Users (
  userID   INTEGER,
  displayName   VARCHAR(45),
  username VARCHAR(45),
  password VARCHAR(45),
  userImage     VARCHAR(45),
  PRIMARY KEY (usertID)
);
```

```
CREATE TABLE Playlists (
  playlistID    INTEGER,
  userID   INTEGER,
  trackID  VARCHAR(45),
  playlistName  VARCHAR(45),
  PRIMARY KEY (playlistID),
  FOREIGN KEY (userID) REFERENCES Users (userID) ON DELETE CASCADE,
  FOREIGN KEY (trackID) REFERENCES Tracks (trackID) ON DELETE CASCADE
);

CREATE TABLE Location (
  locationID    INTEGER,
  userID   INTEGER,
  trackID  VARCHAR(45),
  latitude DECIMAL(9,6),
  longitude     DECIMAL(9,6),
  PRIMARY KEY (locationID),
  FOREIGN KEY (userID) REFERENCES Users (userID) ON DELETE CASCADE,
  FOREIGN KEY (trackID) REFERENCES Tracks (trackID) ON DELETE CASCADE
);

CREATE TABLE follow (
  followerID    INTEGER,
  followedID    INTEGER,
  FOREIGN KEY (followerID) REFERENCES Users (followerID) ON DELETE
CASCADE,
  FOREIGN KEY (followedID) REFERENCES Users (followedID) ON DELETE
CASCADE
);

A template of Playlists Collection:
{
     "playlistID": int,
     "playlistName": "string",
     "userID": int,
     "trackID": [ "string" ]
}
```

Data Collecting
For user data, we created a signup feature in our application and asked some friends to create user accounts. For real-world data, we wrote Python programs to extract track information, album information, and artist information from Spotify using its API and inserted the collected data into our SQL database. Here is a link to the Spotify Scraper:
https://github-dev.cs.illinois.edu/wch4/Spotify-Scraper

NoSQL Database
We use MongoDB as the main NoSQL database that supports our implementation of playlist tables. Because in our design, playlists are generated based on our Advanced Function 1 it is not specifically defined by users, so it needs some flexibility to contain tracks.
Moreover, considering that a user entity can have multiple playlists, of which each of them can contain multiple tracks, we believe that using MongoDB can greatly improve the performance because we can store multiple tracks as an array of values.

Decisions between Relational and Non-relational
We store user data, music data, and location data into a relational database because items that existed in those data are clearly defined and users or administrators know exactly what is the structure of the data table. And since we do not want to have some random unknown data that would become junk information if they do not follow the table format, using a relational database could effectively reject such cases and remain a reasonable and stable structure.
We choose Non-relational for the playlist because as mentioned before, a user can have more than one playlist and the playlist will contain a number of various music as well. And after a music list is generated, the exact tracks contained by it won't be updated or edited while we only expect users to add or delete a whole playlist in our App. In such a case, there is no need to implement a relational database which outputs a worse performance compared to a non-relational database under our circumstance.

List the functionality of your application (feature specs)
Key features of this application include the ability to take a song and add it to a mapScreen which shows all of the locations where users listened to specific songs. Users can click on a song and add it to their mapScreen, along with clicking on songs they've already added and curating a playlist of songs that have been listened to near that location.

Explain a basic function
Once a user has selected a song and pinned where they listened to that song on their map, they have the ability to move or delete that pin.

Show SQL and NoSQL Code Snippets

Since we are using more than a handful of SQL/NoSQL functions, we will only display an example code snippet for each subsection. The rest can be found here.

Example function using MySQL:

```
// Get album image of a track
app.get('/getTrackImage/:id', (req, res) =>{
        pool.getConnection((err, connection)=> {
                let trackID = req.params.id;
                var sql = "SELECT AL.albumImage FROM Tracks T NATURAL JOIN Albums AL WHERE T.trackID = ?;";
                connection.query(sql, [trackID], (qerr, rows, fields) =>{
                        if(!qerr)
                                res.send(rows);
                        else
                                console.log(qerr);
                })
                connection.release();
    })
});
```

Example function using NoSQL (MongoDB):

```
// Get playlist by userID
app.get('/playlist/:id', async function(req, res){
        try {
                //const connect = await client.connect();
                //const collection = client.db("GrooveMap").collection("userPlaylist");
                // perform actions on the collection object
                let currUser = req.params.id;
                currUser = parseInt(currUser, 10);
                var pipeline = [
                        { $match: {
                                "userID": currUser
                        }}
                ];
                const Find = await db.collection("userPlaylist").aggregate(pipeline).toArray().then(results =>{
                        res.status(200),
                        res.send(results)
                });
        }
        catch(err) {
                console.log(err);
        }
});
```

Example function using both MySQL and NoSQL:

```
// delete a playlist by playlistID
app.post('/delPlaylist', async function(req, res){
        let ID = req.body.playlistID;
        pool.getConnection((err, connection)=> {
                var sql = "DELETE FROM Playlist WHERE playlistID = ?;";
                connection.query(sql, [ID], (qerr, rows, fields) =>{
                        if(qerr)
                                console.log(qerr);
                })
                connection.release();
    });
        // delete entry in mongodb
        try {
                var pipeline = {
                        playlistID: ID
                };
                const Find = await db.collection("userPlaylist").remove(pipeline);
                res.status(200);
        }
        catch(err) {
                console.log(err);
        }
});
```

List and briefly explain the dataflow

The dataflow of our application starts when the user logins to his/her account. The user would enter the username, and the userID corresponding to the username is stored in the application. In the Map Screen tab, the application fetches and displays a list of locations belonging to the user, as well as the locations belonging to the friends of the user. To update a location pin, a user drags the pin to a different location and the new coordinates of the pin will be stored in the database. To delete a location pin, the user simply clicks on the callout of the pin and select ‘delete location’ and the database will be updated. To create a pin, the user first selects a track on the Music Library tab and the selected trackID will be passed to the MapScreen. Then the user clicks on a location to set the pin and a new location will be created in the database. The Music Library tab returns a list of tracks corresponding to the search text entered by the user. The Playlist tab stores a list of playlists, which consists of tracks generated by the locations from the Map Screen tab. The Follow tab fetches all the user profiles in the database and specifies which friends are being followed by the user. Finally, the Profile Tab fetches and displays the user information.

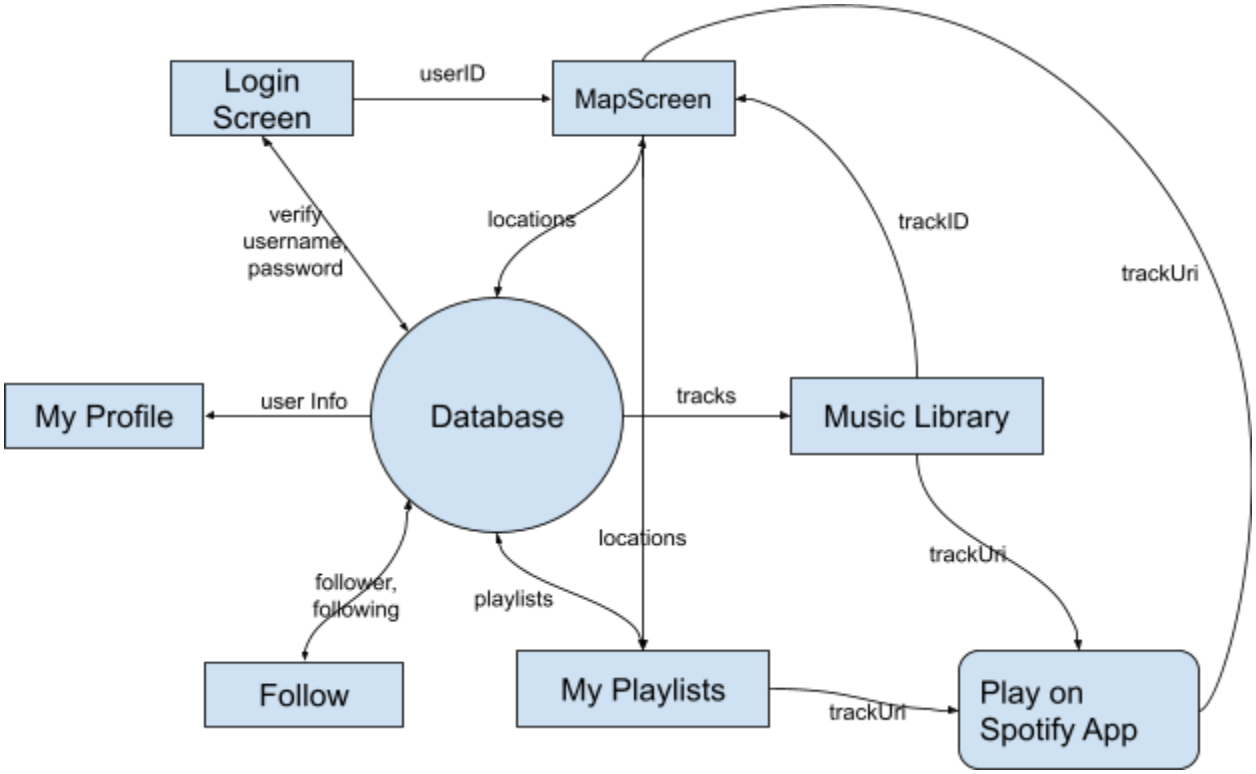

Figure 3: Data Flow Chart

Explain your advanced function 1 (AF1)

Initially, our app allowed any user to share where they listened to a song with any other user on the platform. However, we wanted to make sharing more personal, and to achieve that we added the ability for users to add other users as friends, select which friends' pins to view on their map, and to create a playlist of music their friends listened to within a one-mile radius of any selected pin.

Describe one technical challenge that the team encountered

Our most difficult technical challenges were related to learning the appropriate tools (React, Node.JS, etc.) to actually build our application. Once we got past the learning curves, everything went much more smoothly. For example, when trying to create an image as a button in React Native, and after spending a long time trying to manually implement an onPress() function for an image, we realized that it was almost impossible to get it working how we wanted to. We ended up downloading a separate library called "Icon" from react-native-vector-icons and were able to add the functionality that we were looking for. Another, similar, problem that we ran into was that the built-in buttons in React Native don't have as much customizability as we desired for our app. When we tried to manually implement the functionality that we were looking for, we found that it was taking too long, and we eventually made use of a "Button" from the react-native-elements library to get the functionality we were looking for.

State if everything went according to the initial development plan and proposed specifications

Initially, we had the intention of allowing users to share pictures or videos of what they were doing while listening to certain songs. When trying to implement this, we ran into too many difficulties and eventually decided to scrap the feature of adding photos and videos to posts. We eventually settled on leaving the functionality at allowing users to pin where they listened to specific songs on a map.

Describe the final division of labor and how did you manage teamwork

We managed teamwork by setting guidelines for ourselves in regards to who will do which part of our application. The guidelines we set were pretty general. Eric handled the backend of the app, everything to do with the server and the database. Manik was in charge of developing a user interface (UI), which included designing the layout of the UI and implementing its basic features, and Manik helped develop part of the algorithm for the advanced function. Ben also worked on the implementation of UI features, linked the UI features with the backend of our mobile application, and worked on a portion of our advanced function. Richard led the work on the advanced function and linking all of its parts to the backend, which involved allowing users to follow other users and then create playlists of songs that their friends listened to in a certain area. Finally, to make sure each person did their part, we maintained communication through Slack to ensure progress was being made.