Earbook

This is a group project made by Computer Science & Engineering students at Weber State University.

The goal of this project is to create an environment where people in the same room or virtually are able to sign into and share music. This includes the option to create multiple rooms with users being able to choose which room they want to join. The first person who joins the room will become the host. The host will have additional options that other users who join will not have. While in the room, users are able to share music.

This is a MERN (MongoDB, Express, React, Node) project. We are creating both the front and back end with this project. The frontend will be React/Javascript with CSS styling. The backend will be a server file that posts onto the MongoDB database. API calls are made throughout this project using Axios.

Development:
As the semester progressed, we found division of labor was the best way to go. Juan Santos stood up the initial project by connecting the Front and Back end. Below are the individual contributions each team member made that ended up in the final product:
Juan - Home Page, User Registration Page, Login Page, Random Playlist page, Likes
Hayden - Spotify Authentication, Spotify Dashboard, Spotify Search / multiplier computer connection
Sione - Socket Integration, Rooms, Music Manipulation, Play/Pause, Skip
The group also worked in pairs to develop other work:
Juan and Hayden (In final product)- Spotify Dashboard manipulation, Add songs to the Queue, Deployment
Tyler Gregory and Tyler Godfrey - Room Dashboard Feed, Documentation of sprint goals

Back End:

Juan implemented the front to back end communication. This project has a server and a client-side server. The front end can be found inside of the client folder. The back end can be found inside of the server folder.

The majority of the backend can be found inside of the server.mjs file. The configuration for the backend can be found inside of the db/conn.mjs file. The MongoDB information was stored inside of the .env file.

Tyler Gregory implemented JSON templates that used "…" notation in functions to fill them out and send them to MongoDB. These included structures for songs and posts. The songs object was originally intended to store information such as times streamed/played and a list of users who liked or disliked the song. The main function for this in the backend was /newSongRecord that would take the song ID from Spotify that would:

- Search the DB collection called "songs" for the song with song_id

- If results was greater than 0 (i.e. the song already existed in the DB), it would increase the play counter by 1
- Else, the function would create a new song record using { …songTemplate, song_id: songID } and set the play count to 1

All in all, this probably could have had a better name to suggest that it handled play counts of current songs, or the function could have been divided into two separate functions to better adhere to the Single Responsibility Principle, but the issue was that whenever we played a song, we would call this function, and it would have use precious bandwidth to ask Mongo if this song already existed, then go into 1 of 2 separate functions. Back to the first part, it could have had a better name such as /updateSongRecord.

The template for the post included things such as a timestamp of when it was created, the author, number of likes, and the body of the post. More of this will be discussed later in the Front End section labeled “Room Dashboard Feed Ideas”.

Front End:

Users:

Juan implemented the users feature. Users will be able to create a profile. This includes creating a username, password, and the type of user.

The Registration process:

- The user makes their username, password, and chooses their account type
- The system will make a call from the front end to the back end for a unique salt. The salt is generated in the back end in a function called saltShaker. It generates a random salt using a string of all upper and lower case letters and numbers.
- The password + salt is hashed with the sha-256 hashing algorithm
- The front end sends the username, hashed password, salt, and account type to the back end as a JSON object
- The back end takes this information, unpacks the JSON object, and stores this data inside of the users collection inside of MongoDB.
- The back end will return a complete status and redirect the user to the home page

The Login process:

- The user will enter their username and password
- Every time the username field changes, an API call is made from the from end to the back end looking for a user inside of the user collection that matches. If it does not match, it returns nothing, if a username matches, it returns the salt
- The password + salt is hashed with the sha-256 hashing algorithm
- The username and hashed password is sent to the backend as a JSON object
- The username and password is unpacked and validated against the users collection
- If the username and password matched, the back end will send the front end a logged in status. If it does not match, a pop up will appear saying the password did not match.
- The front end will have a logged in status stored inside of the session and will be redirected to the home page.

This code can be found inside of the Register.jsx and Login.jsx files.

Navigation:
Juan implemented the Navigational features for this project.

- Navigation will be done using Routes. The routes can be found and configured inside of the App.jsx file.
- The Navigation bar is found inside of Navigation.jsx. The navigation will be done with the native React Navbar component.

Spotify:
Juan and Hayden partnered to put together the music portion of this project. The developers decided to use the Spotify API in order to build their music components. We encountered an issue deep in the developmental phase. This issue caused the developers to change the direction of how this project would be completed and ultimately scrapped this portion from the final product.
In order to use the Spotify API, a developer signed up for it using his personal Spotify account. Once the room components were finished and we were ready to start testing playing the music through the Spotify Dashboard, we discovered that only the developer who signed up for the API initially could use the app without issues. The other developers would receive a red pop up saying that the user must have Spotify Premium while having a Spotify Premium account.

Spotify Authorization:
Hayden worked on the Spotify Authorization. In order for the developer to get access to spotify's API the developer needs to have an account with spotify. Once the developer has an account, they will need to go to the spotify developer page to make an app. The app will contain information that we will need for authenication. The app will also track the data for the various API calls we make throughout the program to spotify. In the settings tab of the spotify developer app we can find a couple of key points for us to authenticate the user to access spotify, such as client id, and the client secret. But first in order for us to access spotify we must first send an authentication URL to the spotify. The url must contain some key components, such as client id, response type, a redirect uri, and the different scopes.To authenticate, we have the client id, and the client secret sent to spotify to get an authentication token. This authentication token will be needed for basically every single interaction with spotify, but first we need to login with spotify to get access to this. These are the steps:

- User will click the spotify button and the button will redirect the user to a spotify authentication website.
- The button will send authentication credentials to spotify, such as client secret and client id.
- Once the authentication is done and the user has a premium account, they will be redirected back to the dashboard.
- When the user gets redirected, spotify will send back an authentication token(which we save) that we can use for the interactions we have with spotify.

Spotify Dashboard and Player:
Hayden developed the Spotify Dashboard and Player. The dashboard is where we can search for music, play what music we want to from spotify. At the top of the dashboard that has a

search feature. The search feature uses use effects, first we need to check to see if we have an access token, if we do, we can proceed. The search uses a spotify API that we imported to grab data like artist names, songs names, and pictures. The search will query for twenty different songs. Once this API call is made from the search function, songs will populate down through the dashboard. The dashboard will have a card-like, clickable song list that once a card is clicked, that song will be sent to the player to play. We later added a “add to queue button” next to each song, so we could add songs into the queue. This was done from an API call that used axios “get”. If the credentials were there, we could send that API request.

The Player is an imported feature that comes from “react-spotify-web-playback” where it has all the tools we need to make a good player. The player will show a back, play, skip, time on song, which device is playing the music, and a volume adjuster. The player will take an access token to verify that it has access to spotify. The player will also take the song's uri, which is the unique code that identifies each song. Once a song is clicked from the dashboard, those credentials will be sent to the spotify player, the player will automatically start playing the song that has been selected. The simplified steps go like this:

- After login, the user will be redirected to the dashboard
- The dashboard will be black and the user can type into the search bar
- The search bar will search after every keystroke, updating as more keystrokes are pressed.
- One a song is selected, the songs URI will be sent to the player and will automatically being playing
- On the dashboard next to each song the user can click add to queue, once clicked it will add to queue and play after the next song.

Random Playlist:
Juan developed the Random Playlist feature. This is the ability to add a random playlist to the queue. Spotify’s API did not have an API call to directly put together a random playlist. The developers were able to create a random playlist by using some of the existing API calls. Spotify’s API had the ability to write a category and return a playlist via a JSON object. The steps work as follow:

- User types into a text bar a category
- The developers make a GET API call to return a playlist object
- The developers go through and retrieve each song’s individual unique identifying ID
- The developers make a POST API call to add each song to the queue, looping through and making a single API call at a time
- The songs are added to the queue and the user can play through them immediately upon completion

This code can be found inside of the RandomPlaylist.jsx file. This feature was done using asynchronous functions.

Likes:
The ability to like songs was a feature Juan completed. Inside of the Audio Room that Sione developed is the handleLike function. This function is called when the Like button is clicked. This process works as follows:

- The user will click the like button
- The song name is pulled from the songList array using the position of the props.songIndex. The song name is sent to the back end from the front end as a JSON object using a POST request
- The back end receives and unpacks the object and looks for the song name existing inside of the likes collection. If the song does not exist, the song name and 1 like are added to the database. If the song does exist, it will increment the likes by 1.
- This is the “/likeSong” call.

Room Dashboard Feed Ideas:
Both Tylers worked together to develop this work. The initial idea was to have a feed—much like other social media platforms—that would show things such as “Room X just skipped Y song” or “Song A is now playing in Room B”. It would also show landmarks such as when a song was first streamed on the platform, and users could like these posts and get an idea of what was going on in other rooms when they first logged into the site. The population of the dashboard was handled in /posts:

- The DB collection “posts” was pulled from Mongo
- It would then use collection.find({}) to get all posts in the DB
- At the end of that line was .sort({ timestamp: -1}).toArray() to sort the posts so that the newest would appear on the top of the dashboard, with older ones being towards the bottom

Inside functions that changed the state of a song to “play” or created rooms, we added calls to /newPost to add a new post to the DB, which would:

- Take the { body } from req.body
- Open the DB collection of posts
- Create a new post as { …postTemplate, “timestamp”: new Date(), “body”: body }
    - The author was set to “” for the time being as the author was technically just the system, and we probably could have removed this field in the end as it did not serve a true purpose—the user didn’t have a way to create their own post, it was simply automated through song playing and room creation.
- After this post was created, it would be inserted into the Mongo database

Posts could be liked and rather than being redundant and defining our own id, we used the ObjectId that MongoDB gave each new item in the collection. This was a minor challenge to figure out, as the ObjectId is not an int, it is its own item of type ObjectId. This was solved with the following lines:

- Import { ObjectId } from “mongodb”
- …
- Let { id } = req.body
- Const objectID = new ObjectId(id)
- …updateOne({ _id: objectID}, {$inc: {likes: 1} })