Group 4: Sharify

John-Paul Zebrowski, Jacob Dumford, Luigi Grazioso

Abstract:

Our group has created a web application that enables users to listen to music in a social way. The main idea is to allow people to start ‘streaming sessions’ that can be collaborative, public, or private. When a collaborative stream is established, other people using the application can join the stream and influence which songs will be played next. These songs are stored in a queue that is visible to everybody who has joined the stream. Public streams are similar in function, but do not permit anyone besides the stream creator to edit the queue. Private streams are only visible to the stream creator. Our application uses Spotify’s API to find playlists and songs, and to play music from the browser. Stream data, user profiles, and auxiliary statistics are stored in our database, which is connected to the site by means of a Node.JS module.

Table Of Contents:

1. Functionality

The goal of our project was to create a music application to connect people to the music that their friends are listening to. Our vision of how to do this was to allow people to stream music as they're listening to it. A person starts a stream, and depending on whether it's public or private, anyone who is following that person can see the music they're playing as they listen to it. If someone else decides to tune into the stream, they hear the music that the stream owner chose to play, and they can see which songs are coming up in the queue. We also wanted the option of making this listening experience collaborative, so we envisioned a way for listeners to upvote and downvote potential songs in the queue. Songs in the queue are then sorted by vote with the higher voted songs being played first.

2. Team Breakdown

To organize our group and split up the work in a way that would make the implementation of our application more efficient we broke the project up into three main components. The front-end/user interface, the backend/database, and the Spotify web interface. We decided that each group member would be the owner of one of those three components. Jake started with the front-end, Luigi was in charge of the backend, and JP learned the ins and outs of the Spotify API. Once each of us had some of the basic functionalities working separately, we met as a group to combine the pieces. Database calls are meaningless if the user can't fire them, and a pretty looking music site is worthless without being connected to the music people want to hear.

3. Framework/Infrastructure

As stated in the previous section, there were three main components to our application. They each have separate importance, but the integration of the parts together is where the application gets its functionality and worth. We will describe the technologies and programming strategies of each of these components and then describe the process of linking them together in the next section.

3.1 Database

Part of our back-end was the Oracle Database. Due to how Spotify's API has a lot and most of the information we need regarding users, and songs in our database we store things particular to our app. Some of the things we store in our database include the queue, user votes, and stream information, to name a few. The most important aspect of our database is the stream data. In our application a stream can be consider like a "radio session" which users can host or join. Part of our database manages all these, being the queue from a stream, and the history. Part of what makes our application special is how users really have an effect on what songs get played. This is done by allowing them to vote for the songs they like in the queue, with said queue being affected by these votes. To manage users' votes we use a votes table.

A big issue when working with websites and databases is that the database can be vulnerable to SQL injection. In order to have a more secure database, we decided to use PL/SQL. We created a couple of packages in order to simplify the connection of the database and the front-end. We were able to create several procedures and functions to retrieve relevant information. We also created some to manage data both updating values and inserting values. We had several useful functions and procedures but a really useful and crucial procedure was playedSong. This is really useful because it controls the flow of the queue and keeps track of songs in the history.

3.2 Front-End

For the front-end of our application, we wanted to create a sleek design that would be straightforward to use and make it obvious how to interact with the application. The basic technologies we used to build this were HTML, CSS, and JavaScript. We used jQuery for increased js functionality and Bootstrap to easily style the page. The top bar of the site and the right side bar which contains the queue never move. They are static as soon as you load the page. The main section of the page is updated using jQuery, which allows the user to navigate between pages. Different elements are shown and hidden based on which pages the user is trying to see as well as the user's permissions.

3.3 Spotify API

Spotify provides a fairly comprehensive API that is well-documented with sample calls and use cases. It also provides a web console to try calls using your own account and view the response. The API was robust enough that we didn't have to store any music data on our own database.

Each API call demanded that an access-token be sent in the header, and all the calls regarding current playback information were specific only to the logged-in user. Spotify offers a web endpoint that we used to log the users in using OAuth2, which is how we got their access tokens. For most requests, we were able to use the current user's token, which was stored in a cookie. However, for security reasons, we didn't want to keep all user's tokens in cookies. To sidestep this issue, we had to store each logged-in users access token in a session variable on the node server. Then, when we needed to show the 'streaming users' activity, we looped through each access-token and made many distinct API calls. The tokens expired every hour, so we wrote logic to refresh them before an hour expired.

4. Integration

From the front-end of our application, we communicated with the node server using ajax calls. On the node server, we created endpoints that would communicate with Spotify's API. When possible, we also communicated with the API straight from the client.

We connected the node server with our database using a node module named ‘oracledb’. We installed oracledb using yum, but being somewhat blind to the installation process had its drawbacks. Oddly enough, after the installation we experienced issues using sudo to make any database calls from our node server, but starting the server without sudo worked. We initially were using port 443 for our application but could not do this without using sudo, so we made https available on 8888 also.

5. Challenges

Just about all the functionality of our site required the ability to navigate from page to page, changing the view, but without stopping the music player. The user can’t have their music starting and stopping every time they click on something. This was actually very challenging to implement. Since the music player is an HTML element, navigating to a new page gets rid of the old player and renders a new one. This means that playback stops. Our initial implementation of the application used the Express framework to render views. This framework allows you to easily organize your code and easily navigate from view to view. The problem with this is that each time you change views, the page reloads. To combat this, we decided to render the whole application as one page and use javascript to show and hide elements. When navigating to a new part of the site, rather than a new page loading, the old page is merely hidden and the new page is made visible. This allows the music to continue to play while navigating around the site. The initial loading of the page may be a tiny bit slower, but this makes navigation lightening fast, which is an additional benefit of this implementation.

Some of Spotify’s API functionality is still in Beta testing, so we occasionally encountered unexpected behavior. For example, when making a call to view a user’s currently playing song, no response at all was sent back if the user had no playback information. We had anticipated an empty JSON response, or some message that no song was available. The lack of any body whatsoever was confusing until we were able to root out the cause.

As of June 2017, Chrome disabled support for Encrypted Media Extensions via unsecured channels. With Chrome being the most popular browser around, we wanted to be able to use it with our application.This meant that we were required to use https for all of our communication with Spotify’s API. We used a self-signed certification to enable our node server to use https. Unfortunately, users are now greeted with a security warning upon visiting our page. We could have resolved this by purchasing a domain and getting a real SSL certification.

Another tricky situation arose from the Web Playback SDK, which we used to play music from the browser. It was entirely client-side and initially didn’t play nicely with the functionality we wrote for Node.JS. Furthermore, most critically for our application, it didn’t provide any obvious way to tell when a song had ended. As a workaround, we wrote logic that would listen for state changes, and on any change, we checked whether the size of an array of previously played songs had grown.

6. Testing

Some of our more advanced actions are not fully functional yet, but we did preliminary testing to ensure that our initial functionality works as intended. Each group member logged into the site concurrently to test viewing others' streams. This alerted us to a bug in which our application is not properly waiting for certain ajax calls which produces unpredictable results. We also had other friends log into our site to make sure there are no ways to crash the server while using it.

7. Conclusion

Music is one of the ways that we all come together. We hope that our application can aid people in connecting with their friends and sharing diverse music preferences. Sharify seamlessly integrates an Oracle database and a user-friendly front-end to provide music listeners with an enjoyable and functional product. This project allowed us to explore modern web development tools and learn the process of full stack programming to implement a fully functional application. We also learned a lot about the importance of working in a team and meeting deadlines rather than leaving everything to the night before.

8. References

Bootstrap Components: *https://getbootstrap.com/docs/3.3/components/*
Express Routing: *https://expressjs.com/en/guide/routing.html*
jQuery API: *https://api.jquery.com/*
Node + OracleDB: *https://oracle.github.io/node-oracledb/INSTALL.html#quickstart*
PL/SQL help: *https://www.tutorialspoint.com/plsql/index.htm*
Spotify Web Console: *https://beta.developer.spotify.com/console/*