Max Stetter
Spring 2023

Skipper

Music Management For The Gym

Problem:

There is nothing worse than going to the gym, forgetting your headphones and having to listen to whatever the gym is playing. Sometimes it's hardcore metal and other times it's Ed Sheeran. What if you could have a say in what was playing at the gym. This is where Skipper comes into play.

Solution:

Skipper is a web based application that allows gym guests to skip music that is being played at the gym, hence the name. My solution is a simple and straightforward way for gym goers to skip songs. Gyms would place QR codes that lead to the website and also post a room ID along with those QR codes. Upon opening the gym the staff would host a session on skipper using the posted room ID. Guests would then be able to scan the QR code, enter the room ID, then vote. This handles the problem of listening to bad gym music while simultaneously letting you continue to focus on your workout by being simple and straightforward. I will of course go into more detail about my solution in my solutions video. (Links sections at the bottom)

Technical Overview:

Skipper is a React web application that was created with Next.js. Over the last summer I created my first React app. During that development process I found it very annoying to have to rebuild my app each time I wanted to see my changes. To tackle this I wanted something that had hot reloading as well as fast build times. Next.js came into the picture here as Next.js provides both of those things. Next.js proved to be a good candidate and allowed me to accelerate the application building process. I used normal css and html to style Skipper.

Skipper interfaces with Spotify's web api to retrieve information about songs and the current user. In order to get playlists, current songs, search songs and manipulate the web player. Spotify's web api uses OAuth 2.0 to allow 3rd parties to access their information. I spent a decent chunk of time integrating with this as OAuth 2.0 was completely new to me. I get the access token using the code that spotify returns when logging in. From there I store it and use it with an npm package called spotify-web-api-node. Spotify-web-api-node is an npm package that makes calling Spotify's api easier. Spotify's access token is only valid for 1 hour; however they offer a refresh token that can be used to reset the timer without making the user login again. I

handle this by having a timer that uses the refresh token automatically for the user 59 minutes after it is initially given.

There are two main roles that you can be in Skipper, hosts and guests. Hosts are determined by whether they select the host role and whether there is an access token present or not. Upon choosing the host role, users are asked to login with their Spotify premium account. Once logged in the access token is now present and the UI will prompt the host to choose a vote skip threshold and a room ID. After starting the session with the information filled out, hosts will be able to select a playlist and it will begin playing. Normal web player controls are all there for the host to use. The web player is implemented using an npm package called react-spotify-web-playback. This package just adds a basic web player that uses spotify api requests to manipulate the playback as well as adds the web player as a playable device. There is also a search bar that provides hosts with the ability to search and queue specific songs. The search bar was made so that once the user is done typing, it sends a request to the Spotify api for results. Once you click on a result, it clears the search bar and sends another request to add the track to the queue. Hosts are able to switch between their playlists as well.

The other role of Skipper is the guest. Guests have a much more simple process and UI. Upon loading Skipper and selecting join, They are shown the default vote threshold which is 5 and a blank room ID. All the guest has to do is enter the room id and hit join. Upon joining they are given the current playing song, the vote threshold and any current votes. From there they are able to press the skip button which sends a vote to skip the song. After pressing the button, there is a 15 second cooldown to prevent users from spam voting. If you really want to skip a song and the threshold is 5, you can spend 75 seconds waiting to skip. That's all there is to being a guest.

The relationship between hosts and guests is where I spent the other main chunk of my time. Skipper does not utilize any sort of database as there really isn't a need for one. I did not want people to have to sign up for yet another website and worry about a password or username. You should be able to open the website and it should just work. So how does Skipper convey information between hosts and guests? The answer would be websockets. I used socket.io to implement the websockets however there were some interesting issues that I ran into when deciding to use websockets with Next.js. Next.js implements a serverless function style of web application. This means that there isn't a dedicated server file. Traditionally you would initialize the websocket server inside of the server file however being as there is no server file, I had to find a way around this. Next.js offers serverless function api routes so if you call a certain api it will do the work inside of that api. I decided to place my websocket server inside of one of these api

routes. Whenever a user connects to Skipper, it checks if there is a websocket server already running, if there is, you're good to go and you are now connected. If not, spin up the websocket server then connect to it. The jackbox-style rooms are implemented using websockets as well. When someone is a host, they set the room ID and join that room. Once the host role is assigned and the host is in a room, they will begin polling the spotify api every 5 seconds. The main purpose of this is to handle when songs are changed while simultaneously updating guests as a fallback. I could not find any real way to stream continuous data from the spotify web api so I resorted to polling. On each poll, the host asks for the current song information. Using state objects from react, if they are empty, on the first poll the song information is saved into its respective state objects and the current and previous songs are set to the first song. After this initial poll, the host's only gain from polling is to see if the currently playing track's ID has changed from the previous id. If the previous id and the current id are not the same, then there is a new track playing and thus the vote count needs to be reset and guests need to be informed. After this information is conveyed to the guests via the websocket, the previous id is set to the current id and the polling cycle continues. The host also is listening for update requests and is broadcasting changes to the entire room. In order to distinguish hosts from guests, I use the access token. If you have an access token, you are a host therefore you can send updates. Guests are not able to send updates, only update requests. When a guest joins a room, they request an update to the host. When the host receives an update request, they take all of their current state objects, package them up, and broadcast it to the whole room. Guests also can send votes to the host. When they send a vote, the host counts it and changes their state object that keeps track of the vote count to be count + 1. This results in a change in one of the tracked state objects and thus triggers the host to broadcast an update message to the whole room that there is a new vote count. Leftover votes are handled in the polling that the host does mentioned above. If the song changes, the host resets the count and sends an update with the new song information and a reset vote count. If the vote threshold is reached, the host will automatically skip the current playing song. After skipping the song, the vote count is reset and the spotify api is hit for the new current song information. This results in changed state objects which triggers another update message to be broadcast.

Deploying with Next.js is supposed to be very easy… IF you aren't using websockets. Initially I was going to deploy using Vercel however they do not support websockets and websockets are a key part of Skipper. I decided to go with Heroku instead and deploying with them was as simple as linking my github repo and moving my directory up one level.

Research Summary:

- Spotify Web Api
  Main source of information about how to manipulate the spotify web player and make certain requests. Also a source for how to use it's OAuth 2.0 system.

- Next.js Documentation
  Used to learn basic information about Next.js and about its serverless functions.

- Several Web Dev Simplified videos on youtube
  I learned a pretty decent amount from him, especially about the special npm packages that could be used to make interacting with the spotify web api easier.

- Several PedroTech videos on youtube
  I watched a couple videos about his web socket integration with React using Socket.io.

- Stack Overflow, W3Schools, GitHub Docs, Google
  Various posts were viewed in order to solve minor issues and to look up coding conventions.

Further Work:

- Ability for guests to request songs.
  This is a simple addition that could be done I just ran out of time. Guests can request songs in a simple message that is sent to the host and it is up to the host to manually queue the song.

- Guests can see upcoming songs and vote on them.
  Get the host's queue list and send it to guests, allow guests to vote on the songs and the one with the most votes becomes the next song to be played. If no one votes then it goes in order.

- Allow users to search for any public playlist.
  Currently users are only shown their playlists. It would be nice for hosts to be able to play any playlist that they like.

Max Stetter
Spring 2023

Links:

- GitHub
- Deployment
- Demo
- Code Overview