Matt Brown

CS5610

Final Packet Documentation

Executive Summary:

Music Box is a dynamic social media platform that aims to build a community of avid music fans through the reviewing of new, old, and classic albums. Discussion can be made about each user's review sparking debate and new albums can be discovered through looking at reviews of albums you have not heard before.

The actual web app itself allows anonymous users to search freely through the spotify API and look at albums by three criteria: by artist, by specific album name, or by users. They can click on the album which redirects them to a page which will hold all the recent reviews of the album.

Users with an account can follow each other, like other posts, and even comment. This is the way MusicBox is meant to be used!

Project Background:

The first thing I thought of when the final project was described was an idea for a social media reviewing platform for music. I am a huge fan of this specific app called Letterboxd which is basically a social media app for movie reviewing, so I took that same concept and just applied it to music. I think the main thing that this webapp addresses is that there is no real social media website or app for reviewing music.

I choose a music based webapp because of my own close association with music. I am a drummer and a musician at heart and love to talk music with other people. I also love hearing people’s takes on new music, and this seemed like the best way to streamline that.

A lot of my framework and ideas for functionality literally just came from Letterboxd and the basic functionality it provides to users and non-users. What I essentially hoped to achieve was an app that

would dynamically display albums average user score as well as the ability for people to like and reply to each other's posts. I was also hoping for a search portion that would allow other users to search for albums via different parametrics. I also have a search portion for actual users so they can look people up and follow/unfollow them.

Project Development:

Overall structure:

For the overall structure of this project I used a MERN stack implementation. The front end was built mostly using the CSS framework of bootstrap react and some MUI features such as the rating system they have.

Login:

My login feature was implemented using a REST api to post request a user form to my mongodb database. The request is made from the loginCall api call feature where the email and password are sent in as an object and then sent through an axios request. The axios request then dispatches to the context API which then delivers a user or a null user with an error attached if the login fails. Something that was really messing me up was initially that every time I would log in but refresh the page I would lose my user. I did not know this, but apparently you have to store the user credentials in the local storage part of your application because your App resets itself every time the page refreshes. This gave me a ton of headaches when I first implemented this, so it felt good to finally get this down.

Overall user state:

To handle the overall user state I implemented a technique often used with the react useContext API. I had three different components to handle user state: the actions, the reducer, and the provider. The actions component just lists out all the different actions that could happen based on different cases (login,

logout, etc). The reducer actually delegates out the different case based actions based on a successful login or logout. The provider then provides the user context by wrapping all the children elements. I then wrapped the <App/> in the index.js file to give the context to every element in <App/>. From there I made ternary operators to load different page components based on user state. For example, if the user state is null then if someone tries to go to the /userhome portion of the project they will be redirected to the home "/".

Nonusers:

Non-users have the profile part of their navbar removed and can only access the home page and the search portion as well as user profiles. However, buttons that usually allow you to like a post will redirect the user to the login page if they try to like or comment on a post or follow a user. Non-users also see different data for their home screen. They will see the last 8 reviews made by the community and then see the last 4 users to sign up for the website.

Search API:

The search API was created using the Spotify open API. What I basically had to do was first get the access token, then use that access token as part of the GET request to the server and then specify the user input in the query string part of the search api. There are three different parameters that the user or non-user can search: by specific album name, by artists (displays all the albums the artist has released newest to latest), and by user (this uses my own restful API not the Spotify API). The results are then mapped to the albums state as an array and are then mapped out on the page accordingly. If the user enters something that can't be found an appropriate error message appears.

Profile:

The profile page calls from the rest API to retrieve the User from the context, and the reviews associated with that user from the Reviews collection in mongodb. Each review document has an album

Object associated within it, an array of likes, comments, name of the album, and object id. The profile page loads the user image, name, and then how many reviews they have made, how many followers they have, and how many people are following them (this is from the User collection in mongodb). Note that I did not map out a separate response for /profile and /profile/{id} based on login. /profile/{id} is the only way to get someone's profile even if it is your own. This implementation made a bit more sense to me.

Overall completeness:

This was a very tough final project for me. I had tons of trouble getting my user authentication to work and trying to figure out Azure Labs. If I had spent less time on Azure Labs and more time troubleshooting I think I could have gotten more completed, but unfortunately that was not the case. Overall it is a working website, but some features are static. Commenting on posts does work, but unfortunately I did not have time to implement it fully, so that is something that will be static in my website. Additionally, I was not able to fully delete or update a user's account. This was due to the fact that I had implemented a lot of my get requests and put requests by the username instead of the userid which means that if the user updates their username I would have to go into every single like and comment and manually modify this. Now I know what to do in the future!

The other thing I wasn't able to fix was the speed of the API. The overall speed of loading users information and reviews is VERY slow. Sometimes it takes up to 15 seconds to load all the reviews on the screen. Unfortunately I did not have time to implement a loading screen, but in the future I will definitely do this. So whoever is grading this…give it 15 seconds at least!

Project reconsiderations:

After having completed the project, there are many things I would do differently if I could. The biggest thing I would have done is instead of creating my own login system I would have just used an OAuth system. More specifically I would have used the spotify OAuth system. This would have allowed me to get the access token initially and assign it to the specified user in the beginning so that I would not

have to continue fetching for the key each time I wanted to use the API. This would have made things a lot easier for me. Additionally, each spotify account logged in could generate user specific information about the user such as their own playlists and top artists that would have been cool to include in their profiles.

In the future, I would love to fully implement a feature that would calculate all the different albums reviewed, get all of their averaged scores, and then sort out the top 4 rated albums and display it on the home page. I was very close to getting this to work, but unfortunately due to the time constraint I was not able to get it 100 percent working which is why I ended up not including it in the final package.

Additionally, if I was to do it again, I would try to spend more time planning out all the individual components and how they work instead of just coding away. I think it's human nature to want to avoid planning and just get to work on it, but looking back I could have saved myself a whole lot of time if I had planned out my project more.