Creating a Web Application for Visualizing Music Discovery

Alexander Tenczar
aat1006@wildcats.unh.edu
Master Project in Information Technology
Department of Applied Engineering and Sciences
University of New Hampshire
Manchester, New Hampshire, USA

ABSTRACT

As music streaming services are becoming the de facto standard, it has never been easier to listen to new music. Although these services already make recommendations to their users, they're often cryptic and offer little reason as to how they made these suggestions. To solve this dilemma, my goal was to build a web application capable of querying two different music service APIs with correlations in artist data and visualizing the artists as a single data source. To achieve this goal, concepts such as web application development, RESTful web services, and data visualization were studied throughout the duration of this project.

How the application works is quite simple. To visualize an artist's similarity, a searched artist generates a unique bubble chart composed of various artists that are considered similar to them. Each bubble's radius is calculated based on how similar they are. The bigger the bubble, the more similar the artist is. To make each bubble unique and identifiable, it is filled with an image of the similar artist. Every bubble contains a link to a visualization of its artist's similar artists. This gives the web application an interactive quality that maintains user interest. To achieve implementing various features, this web application requests and combines data from both Spotify and Last.fm's publicly accessible APIs. The web application itself is built on a Node.js runtime environment using the React.js library to assist with creation of stateful components. It was then deployed to Vercel. The result of this project's work was a web application that renders interactive and engaging data visualizations, assisting users with discovering new artists to listen to.

This paper describes the development, design, deployment, and testing of the web application that made this idea a reality. Specific research topics covered include: web application architecture, API requests, data visualization, user interface design, and deployment to a hosting platform.

CCS CONCEPTS

• **Human-centered computing** → *Information visualization*; *User interface design*; • **Information systems** → *RESTful web services*; **Web applications**.

1 INTRODUCTION

Considering streaming platforms have made music so accessible, more people are looking to take advantage of their new endless music libraries and discover artists they've never heard of. I am one of them. Although I have the ability to listen to anything I like, I occasionally couldn't think of who to listen to next. To fix this, I began looking for websites that could help me find artists similar to ones I enjoyed. I soon stumbled upon Last.fm and Music-Map.com. Both of these websites provide ways to discover new music, but neither are particularly interactive or visually engaging.

Last.fm is essentially a collection of music data its users contribute in the form of their listening activity on streaming platforms or by their local files. Every song its users listen to is referred to as a "scrobble" and is sent to Last.fm to be recorded on their profile. This website is extremely useful, but discovering new artists lacks any visual components. Instead, the user clicks through conventional web pages to find new artists. Music-Map on the other hand, is interactive. The site prompts the user to search for an artist and proceeds to display a visualization of all their similar artists. This can be done infinitely, allowing the user to go down a "rabbit hole" of musician suggestions. The issue with Music-Map is that the site and its visualization are severely lacking. Instead of seeing the artist's picture and learning more about them, you're only provided their name. This requires the user to search for each artist on the internet before truly "discovering" them. On top of this, Music-Map's data is reliant on user suggestions which means the site's artist data doesn't align with streaming platforms and most likely presents a requirement for additional data upkeep. Despite these faults, Music-Map was the site that gave me the most inspiration to create this web app.

What I wanted to do was create something that combined Last.fm's already expansive collection of music data with Music-Map's concept of visualizing musician suggestions. I decided to do this by utilizing both Last.fm and Spotify's APIs. This allows the user to query over 1 million musicians and display a selection of artists similar to them. Both Last.fm and Spotify's APIs provide publicly available artist information that the web app requests and temporarily stores. Once the web app has the data it requires (similar artists, similarity ratings, and artist images), it visualizes the data on the front end of the web app for the user. Over the course of developing this web app, concepts such as RESTful web services, application state management, user interface, and data visualization were investigated. The goal of this project was to build a web application capable of querying two different music service APIs with correlations in artist data and combining them as a single source to visualize their data.

To achieve the goal of this project, I set and completed several objectives. The most apparent one was sending requests to both Last.fm and Spotify's APIs. It was necessary to use two APIs instead of one because Last.fm recently stopped providing artist images through its API. Last.fm provides the list of similar artists and their similarity ratings, while Spotify retrieves each artist's image. This meant I had to find a way to associate Last.fm's and Spotify's API responses so each artist has the correct image. Once I had the necessary API data, I moved onto visualizing it. I then proceeded

to refine the web application and design its user interface. At this point, the web app was ready to be deployed. Following deployment, I optimized its code and tested it across multiple browsers to ensure it worked correctly.

I used a variety of technologies to build the web app. It runs a React project on top of a Node.js runtime environment. Within React, an HTTP request client known as Axios makes API requests to the two APIs. To visualize the data, another JavaScript library known as D3.js is used to create the data visualizations. D3 is imported inside a React functional component and is fed the API data, rendering a visualization. The user interface of the application uses SASS style sheets with inspiration taken from existing popular websites' user interfaces. Finally, the web application has been deployed to Vercel, a cloud hosting platform which offers automatic deployment by interfacing with Git version control.

By completing these objectives, I learned the entire development process from start to finish of successfully deploying a web application that queries two different music service APIs with correlations in artist data and visualizes the artists as a single data source. In this paper I begin by elaborating on the goal and objectives of the project. In the next section, I then explain the approach taken to complete each objective. Following the detailed overview of each objective's approach comes results. This describes my process for each approach and what it produced. Accompanying each result is an evaluation which offers my opinion on what went right and what could have been done better. A conclusion ends the paper.

2 GOAL AND OBJECTIVES

The goal of this project was to build a web application capable of querying two different music service APIs with correlations in artist data and visualizing the artists as a single data source. To complete this goal, I determined the steps that were crucial in progressing the project's work. These steps became objectives of the project. While determining my project's objectives, I thought about what features set this web app apart from others and what would be done in order for it to properly work. To have a clear timeline of my project's work, I ordered my objectives similar to my steps of development.

My first objective was determining the architecture of the web app. To complete this objective, I decided how many web pages I needed, what UI components each had, and where both APIs were queried. URL routing was also another important part of this objective because the search page's URL needed a query parameter to be dynamically routed. Without knowing how each piece fits together, it wouldn't have been possible to implement my vision.

The second objective was arguably the most important aspect of the project, this was the retrieval and storage of the two APIs data. This involved setting up the APIs, authenticating with Spotify, storing/comparing their data, and sending it between components. If the data wasn't properly utilized, the web app's visualization might as well have not existed as it wouldn't have had any coherent data to visualize.

Another integral objective was learning to manipulate my data with the D3.js library. Data visualization with JavaScript was a completely new concept for me and posed the biggest threat towards achieving my goal. This objective's development ultimately decided how my web app was used by its users.

The deployment of my web app followed once I had a properly working visualization. By deploying my app before the project was complete, I could better test the web app on a wide range of devices and browsers. The method of deployment was automatically run every time a new commit was pushed to GitHub. This made the deployment process painless once the repo was initially linked to the deployment platform. My platform of choice inevitably was on a cloud deployment platform known as Vercel.

Since testing and optimization were both performed throughout the length of the project, it was best to include them as an objective. If I didn't constantly fix bugs throughout my project's development, the web app would have been riddled with issues and appeared unfinished. The tests of the project involved checking the various API calls, state assignments, and ensuring the app was responsive across all device sizes. The app was optimized by inspecting the existing code and determining whether there was a better, more efficient way of implementing its purpose.

The final list of objectives consisted of:

- Determine the Web App's architecture and how all its features would work together.
- Set up Last.fm and Spotify's APIs and find a way to relate their data to be used as single data source.
- Learn to implement D3.js's visualization library and use it for visualizing my app's data.
- Deploy the project to a cloud platform that utilizes version control for easy, automatic deployment.
- Test and optimize the web app's code to ensure it is in a performant, presentable state.

3 APPROACH

This project's approach placed a large emphasis on the web app's data retrieval and visualization capabilities. This is because both are integral to whether the app functioned as intended. I began by determining how it would operate. From there I initialized a web application to add code to. This could be thought of as creating the skeleton of the project. Following the completion of my project's main objectives, I deployed, tested, and optimized the web app. These steps are essential for any successful web app.

This project's development wouldn't have been possible without research and assistance from online resources. One thing one might notice about the approach I took was how much I used Google. Google is probably the most comprehensive tool for finding answers to your questions. I began determining my approach by searching for what other people had done under similar circumstances. From there, It was relatively easy to find websites that had the answers to my questions. Very often I was lead to various articles written by fellow developers. Many of these articles came from Medium.com, and Dev.to. These websites have become popular in recent years for people to post their own articles about what interests them. While it may seem this could introduce articles detailing poor development practices, this wasn't an issue for me. The majority of the information only helped me progress through my project. If I had a question regarding a best practice, I ended up verifying it on other sites. Usually official documentation sources like Mozilla's

Developer Network. By reading these articles and seeing how someone might achieve a task similar to mine, I successfully applied the fundamental takeaways to what was relevant to my web app.

In addition to using online publication sites, I consulted my tools' documentation and looked up issues on Stackoverflow.com. With Stackoverflow's approach of offering multiple answers to a question, I could frequently attempt multilple methods to solving my issues. Not only did I make progress thanks to Stackoverflow, I gained additional knowledge at the same time.

As mentioned earlier, Mozilla's Developer Network offers amazing up-to-date information regarding HTML, JavaScript, and CSS. If I wanted to find a web element's list of attributes, I often referred to MDN's documentation. In addition to Mozilla, I visited Facebook's Create React App documentation and d3.js' wiki on GitHub [2]. By looking at a library's documentation, I not only saw the correct usage of the library/tool's functions, I learned new functions and attributes I hadn't known existed.

3.1 Web Application Architecture

While I was determining BandViz's architecture, I primarily consulted Create React App's documentation. This was because to get the web application up and running, I felt it necessary to commit to a specific folder structure. If a certain folder structure wasn't followed from the beginning, it would have been a waste of time reorganizing the web app's various components and style sheets as new files continued to be introduced. Within Create React App's documentation, there is a section devoted to showing and explaining a recommended file structure for your React application [7]. I modeled my project's file structure similar to what was documented.

Following the creation of the file structure, I mapped out the site's pages. This part of my approach did not rely on as much documentation. The reason was from the start, I had a good idea of what pages were necessary. The one resource I did use was React Router's documentation. This was necessary since I never created a dynamically routed page [14]. The page that would be dynamically routed was the search page responsible for visualizing each artist's results. React Router's documentation was specifically used because this was the Node package responsible for the routing functionality. React Router is currently one of the most popular routing solutions for React. Once I had determined how to dynamically route the web application, the architecture was in a good enough state to move onto collecting the data.

3.2 Utilizing RESTful API Data

Aside from creating some fetch calls to a Content Management System for my previous internship, I had no little to no experience with using API data. To begin my learning journey with APIs, I went to Google. My initial searches were aimed at finding tutorials that covered making calls to APIs with React. After finding a multitude of results, I narrowed my search to tutorials that explained how to query Spotify's API in React. This lead me to find some helpful articles I could apply to the project. An article that helped was "How to Create a Spotify Music Search App in React" by Yogesh Chaven [1]. Although this article had nothing to do with visualizing Spotify's data, it was an important stepping stone towards achieving this goal. The article provided information regarding authenticating and querying Spotify's API with Axios. Up until this point, I hadn't determined whether I should use Axios (an HTTP request library for React). After reading this article, I was convinced that Axios was a valuable asset. I proceeded to incorporate the methods detailed and began sending requests to Spotify as well as Last.fm.

At this point it was somewhat obvious what API data this project needed. Now the challenge was figuring out what API endpoints were used to retrieve this data. This step was completely done by referring to both Last.fm and Spotify's API documentation. Each have a list of possible endpoints. As my requirements for requests grew, I referred back to these pages and find which available endpoints satisfied them.

3.3 Implementing D3.js

The objective of visualizing the API data required the assistance of tutorials. If I hadn't had access to any example code that used D3, I would have been completely lost. This is mainly because while D3.js is a JavaScript library, it operates in its own way. How the SVG canvas is initialized, the mapping of data, not to mention the animations, were all new territory. Credit has to be given to the developers for creating a comprehensive documentation of all the function's attributes. My biggest issue was how little information was provided by D3 on how to actually start. Luckily, many articles existed that instructed how to create a bubble chart similar to what I wanted to create. Throughout my development of the visualization, I followed a couple tutorials that explained how to create a bubble chart in D3. The most helpful one was Jim Vallandingham's "Creating Bubble Charts with D3v4" [18]. In addition to this tutorial, I referred to D3.js's documentation when further explanation of how a function or attribute worked was required [2].

3.4 Deployment

The steps required to deploy the web app were quite minimal. Other than following the Vercel's documentation, I only had to learn Create-React-App's approach to environment variables.

The way I found my deployment option Vercel, was by searching on Google for other developers' recent opinions of various cloud deployment platforms. I primarily did this by searching through Reddit.com for posts made within the same year that recommended deployment options for React. Something I noticed while looking through these posts was that two platforms: Vercel and Netlify were recommended the most. I eventually settled on using Vercel because it was the first option I tried and it simply worked.

Vercel was easy to set up. All that I needed to do was sign in with my GitHub account and choose the repository I wanted to deploy. From there, I only needed to figure out how to provide my API keys to Vercel without pushing them to the repository. This is done through environment variables. To learn how environments were handled by both Vercel and Create-React-App, I read their documentation [19] [4]. After the environment variables were properly created and referenced, the app was deployed to Vercel and performed the same it had on my local development environment.

3.5 Testing and Optimization

The goal of testing and optimizing the web application was to ensure that it was performant and presentable. To achieve this, I researched JavaScript and CSS best practices in addition to my previous experience with testing web sites. The bulk of the testing done throughout the project's development would be considered functional testing and debugging. Functional testing is simply the testing of the web application's functionality. In addition to testing basic functionality, functionality testing also includes testing usability, accessibility, and the handling of errors [3]. While creating the web app, I would continually visit it to make sure it operated how I intended. If it wasn't, I used that indication to adjust my code.

The majority of this project's optimization work dealt with how the similar artists were queried. Examples of my optimization include: asynchronous vs synchronous functions, which loop type was the fastest, and the ideal number of artists to display. All of these played a key role in the development of the project because it meant my web app could continue to get more efficient and support additional users.

4 RESULTS

By the end of this project, I met the expectation that web application would be fully functioning. Both Last.fm and Spotify's APIs are queried their data linked, then visualized on the front end of the web app. Although the web app works as intended, there is room for improvement. The possible improvements primarily relate to the optimization of the web app. Taking these improvements into consideration, there are additional development opportunities I can carry out after the course has concluded. The project was evaluated on how much functionality I could implement according to the original goal.

4.1 Web Application Architecture

The first objective of my project was to determine the architecture of the web app. This included what technologies I would utilize as well as the site's overall structure. These determinations were made after a combination of researching ways developers recently tackled similar problems, and applying my acquired knowledge from my past internship. Although some of my originally planned approaches fell flat, many of them worked as expected and cooperated quite well with the rest of the application.

In order to get my web application initialized and running, I used Facebook's Create-React-App Node package. Create-React-App is a tool used to quickly start up a new React project for you to immediately add code to. After CRA is installed globally into your Node Package Manager (NPM), you enter the command "npx create-react-app name" and a new running React project is added to your current directory. CRA is pre-configured and comes with build scripts to help optimize the app when it is ready to be deployed. Inside the project directory there is a package.json file for specifying project's scripts and node dependency versions, a folder that stores the project's node dependencies, a public folder for storing the files that are served to the web server, and a src folder [8]. The src folder is where all the web app's content resides.

Once the React app was initialized, I began making some minor adjustments to the inner structure of the src folder. Because this

```
my-app/
  README.md
  node_modules/
  package.json
  public/
        index.html
        favicon.ico
  src/
        App.css
        App.js
        App.test.js
        index.css
        index.js
        logo.svg
```

Figure 1: The default folder structure of Create-React-App

web app would be more complex than the starting code, I used additional folders to help organize all the moving parts of the web app. I first added a component folder for my components. To store functions for things such as authentication, I created a config folder. I then created a hooks folder for any custom React hooks that may be reused. To store each page of my app, I created a pages folder. The web app also utilizes local static assets; thus, a static folder was added. Finally, a styles folder was created to organize the various style sheets that were applied to specific components and pages.

```
webapp/
  README.md
  node_modules/
  package.json
  public/
        index.html
        manifest.json
        robots.txt
  src/
        components/
        config/
        hooks/
        pages/
        styles/
        App.js
        index.js
```

Figure 2: The project's folder structure after creation of subfolders

One decision I made for this project was to use SASS for the styling of my web app instead of CSS. This change was made possible with Dart SASS, another Node package. Using Dart Sass, I can import SASS style sheets into my components and pages and have them applied the same way CSS does. SASS provides many advantages over CSS, however two particular ones were the reason why I decided to use SASS. The first advantage of SASS is that it uses a nested syntax. In conventional CSS, each specific element you want to style must be selected separately. This means any calls

to children must be made in a new selection. In SASS, children can be called inside the selection of a parent. This makes SASS' formatting cleaner, allowing for less confusion down the road. The second benefit of SASS that I took advantage of was its use of variables. In SASS, variables can be set to values and reused across all style sheets. All that needs to be done is initially assign the variable in an .scss file, and import the file in any style sheets that reference the variable. I wanted to use variables in my style sheets so I could easily call frequently used values such as colors or device screen widths.

The next part of planning my web app was determining how it would operate. Because I wanted the app to be user friendly, I decided to make it a two page web app. One page is a static landing page. The landing page is static because the content always stays the same. It resides at the base URL or "/". This page simply presents the user with a search box to search for their artist. To do this, the landing page makes a call to both APIs and populates a drop down menu with results that match the user's query. Each result provides a different link to a new page containing a query parameter. This new page is the page containing the visualization. Because the visualization page requires a query parameter, it is dynamically routed. A "dynamically routed page" is one that's routed "as the app is rendering" [3]. For example: a search for the artist "Madonna", yields the URL "/search/Madonna". This page's query parameter determines what artist the app visualizes the similar artists for. This means that while the web app has technically only two pages, the search page can provide millions of different visualizations. For each similar artist that's populated in a visualization, a link to another new visualization query for that artist's similar artists is attached. By doing this, a user could infinitely query artists from the visualizations rendered for them. Making the web app appear as if it has many more than two pages.

4.2 RESTful APIs and State Management

This web app uses two RESTful APIs (Last.fm and Spotify) to retrieve the necessary data. My expected result of this objective was to: search for artists, find their similar artists, and store their data.

APIs, or "Application Programming Interfaces" are interfaces that allow products or services to communicate with other products or services without knowing their implementation [3]. For instance, my web app has no idea how Spotify or Last.fm collect or store the data their APIs provide. Instead, all it knows is to read the JavaScript Object Notation (JSON) response sent back from the APIs. What makes an API RESTful is that it conforms to a set of six characteristics considered RESTful. The first is that the API is based around a Client-server architecture, meaning the API serves HTTP requests from a server to various clients. My web app is one of those clients that will be accessing the APIs' servers. The second characteristic is that the API is stateless. A stateless API's requests are isolated and do not change once the client receives the request. Caching is another characteristic. This just means that the API's requests can be cached either by the client or the server. The next characteristic is that it is a layered system. By adding additional layers between the server and the client, the API can implement features such as load balancing and security. The fifth characteristic of a RESTful API is optional. This is being able to provide the client executable code from the server. The final characteristic of a RESTful API is that it has a uniform interface. By having a uniform interface, one RESTful API can operate in the same manner as another. For instance, you can send a request to Last.fm's API the same way you would to Spotify's API [15]. Within their respective documentations, both Last.fm and Spotify's APIs are identified as following the principles of REST [17][13].

Once I got my web application up and running, I began preparing to query the APIs. First, I retrieved my API keys from Last.fm and Spotify. In order to send requests to both APIs, a unique key needs to be sent to their API to identify the application that's requesting the data [10]. Last.fm's API was simple to set up as it only requires one API key. This can be done by going to the Last.fm's API site and requesting a key. Spotify's API was a little more tricky. To make a call to Spotify's API, I first had to obtain both a client id and a client secret key from their website. The client id is what uniquely identifies the web app while the client secret is a secret key that helps verify the client [16]. Both of these keys are encoded and sent to Spotify so the web app can be verified and receive a token to be used for future calls.

Once I had my API keys, I installed an HTTP client known as Axios. Axios is an NPM package that makes sending HTTP requests such as API calls easier. One of the benefits of using Axios is that it automatically converts the responses to JSON objects [12]. With a conventional fetch request, the response would first need to be manually converted to JSON. Using Axios, I have one less step to worry about when getting readable data from my APIs.

Before diving into explaining my queries to each API. I need to explain why both Last.fm and Spotify are being used. Why not just use one or the other? My initial plan was to only query Last.fm's API for artists. I chose Last.fm because along with an array of similar artists, they also provide a value that measures how similar each artist is to the one that was queried (known as the similarity value). Spotify does not provide a similarity value with its list of similar artists. I was all set to go with this approach until I found out Last.fm stopped serving artist images. This was initially deal breaking as the image for each artist is crucial. I wanted the visualization to use each artist's image to identify them, making the visualization aesthetically pleasing. This is where Spotify's API comes in. Spotify provides artist images along with each response. Knowing this gave me the idea to query Spotify for each similar artist retrieved by Last.fm, and use its image. Although this approach would inevitably make my web app more complex, it was the only way to achieve my goal of visualizing the artists.

The first part of my web app that makes API calls is the search bar component. This component is the search bar rendered on the homepage. What I wanted the search bar to do was upon user input on the search field, send a request to both Last.fm and Spotify. The queries would contain the same search terms and determine whether that exact artist exists on both Spotify and Last.fm. Such a verification process is done by first searching Last.fm and then Spotify. Once both API requests are returned, the web app compares the results from Last.fm with those from Spotify using a for-loop. If an identical match is found for the artist's name, the search bar's drop down is populated with the name and link to their visualization. Later in my development, I queried Last.fm's API for the top

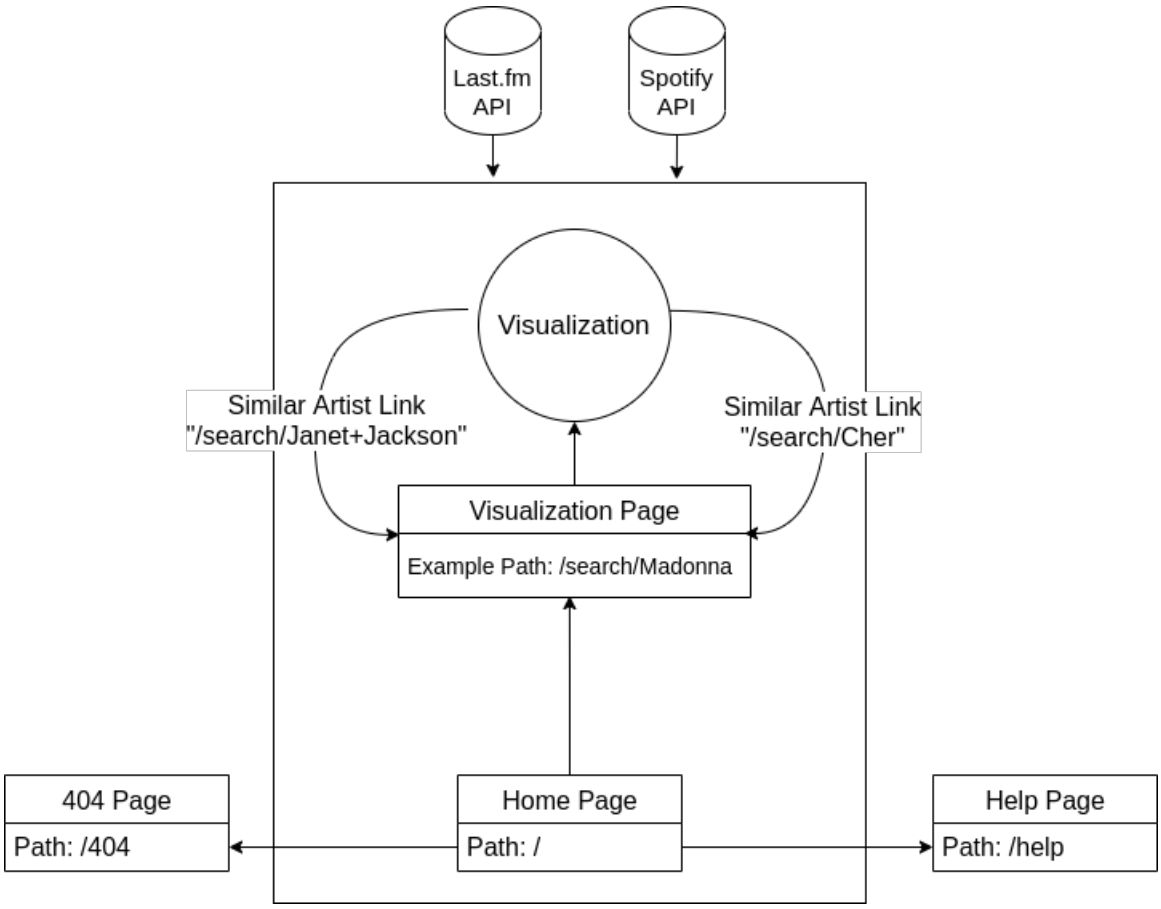

Figure 3: The Structure of the Web Application

50 trending artists. This data allowed me to create an animated placeholder with suggestions that can change each day.

The main portion of the web app that makes API calls is the visualization page. The component on this page responsible for making these calls is known as SimilarArtists.js. My decision for how I wanted to visualize the artists was pretty simple. If the search page is authenticated by Spotify, it will render a SimilarArtists component. This component then contains all the logic and functions for retrieving the similar artists, their images, and the queried artist's info/top tracks. Once enough data is retrieved to populate a visualization, either a Last.fm or Spotify visualization component is rendered. There are two different visualization types because I later found out that for a small number of artists, it doesn't provide an array of similar artists. Spotify is the fallback visualization because it doesn't provide a similarity value for each artist.

Retrieving the similar artists from Last.fm became a little complicated once I discovered they stopped providing artist images through their API endpoints. This meant that to create a visualization that still contained artist images, I would need to utilize another API. Luckily, the Spotify API continues to provide images for its artists. Initially I considered dropping Last.fm's API entirely in favor of Spotify's, however Spotify does not provide similarity values for similar artists. This posed a difficult scenario as it meant the web app needed to combine the data from both APIs to provide my visualization component sufficient data. After brainstorming, I settled on a relatively simple solution. It was to first retrieve the list of similar artists as well as their values from Last.fm. This list is stored in an array of objects. Then for each artist in the array, send a request to Spotify's API which helps determine whether it also had that artist. If the artist did exist on Spotify, the artist was pushed to another array of objects designated for Spotify artists. After this is done, both arrays are run through a comparison function that finds any artists that don't match. This filters out any artists that were initially found by Spotify but aren't actually the same artist. Once the process is done, both arrays are assigned to two React state arrays for them to be accessed by the visualization components. At this point one of the two visualization components will render depending on whether the Last.fm state array contains children. If it does, the ideal visualization of similar artists and their similarity ratings will render. If not, the web app will depend on Spotify's list of similar artists that lacks any similarity values. After testing the web app throughout the semester, I estimate that only one out of every twenty times may you encounter an artist that doesn't have similar artists served by Last.fm. These occurrences are somewhat

random but usually depend on how popular or new the artist is. The more popular they are, the more of a chance they will be correctly visualized.

A big issue I ran into with BandViz was the sheer number of API requests it had to send to Spotify on every artist search. This is because when Last.fm returns an array of similar artists, the web app needs to send a request to retrieve each artist's image. The issue is that Spotify does not provide unlimited requests to its API users. If it notices a high number of requests coming from an API key over a short period of time, it will timeout future requests as a means of throttling the requester. Through testing of the web app, I've discovered the user will need to wait a couple seconds before they can refresh the page and request a new visualization.

A solution to Spotify's API bandwidth limitation would be to send a single request to them that combines all the artists. Unfortunately, this simply isn't possible at the moment. Spotify does not provide an endpoint for bulk searching multiple artists. My only other possible solution would be if Last.fm provided the Spotify ID of each artist through its endpoint. While Last.fm does provide Spotify integration through their service, it's quite unlikely that they will go out of their way to add Spotify IDs to each of its artists on the API. Instead, the web app needs to search for each artist through Spotify's API by their name. Provided these API limitations, my solution was to lower the number of similar artists visualized by the web app. Last.fm's response returns 100 similar artists; that is simply too much information for both the user and Spotify. I lowered it to around 30 artists being visualized. That reduced the load on Spotify's API for each visualization by about one third.

In addition to creating API requests for artist search and visualizations, the web app uses a couple other endpoints to add additional functionality. Two features I added to the web app that required additional API endpoints were track previews and an animated search placeholder. Track previews are retrieved for each searched artist and help the user get a taste of what the artist sounds like. I used Spotify's "top tracks" endpoint to retrieve this data. On the homepage, BandViz's search bar has an animated placeholder. This placeholder cycles through recommendations of artists to search for. These recommendations are pulled from Last.fm's "Top 50" artists. The API request responsible for this returns a list of artists that my animated placeholder function is populated with. In comparison to the number of Spotify artist image requests, these two play little effect on my API request quota. The reward of having this extra functionality is well worth the two extra requests.

4.3 Visualization

The purpose of this web app is to visualize similar artist data. To achieve this, visualizing the API data was made an objective. I expected to produce a result where I would effectively display an artist's similar artists for the end user to see and comprehend effectively.

Initially, I wanted to take inspiration from Music-Map's method of visualization and have a grouping of similar artists surrounding the originally queried artist in the middle. This visualization heavily relied on calculating the distance between the searched artist and its similar artists. The distance would have been relative to how similar each artist is. The closer the similar artist is, the more similar they are. The issue with this was that while Music-Map only displays names, my visualization identifies each artist with an image of them. If I tried to fit in 20 or more artist images, each with a calculated distance, the images would have to have been scaled too small. This required me to change my original idea and transition to another method of visualization.

The new visualization technique is still quite similar as it utilizes the "similarity" value provided by Last.fm's API. This time the visual cue is the size of the artist's image. For the web app's visualization, each similar artist is displayed as a bubble. Each bubble possess three attributes. The name of the artist, their similarity value, and the link to their image. Every artist's bubble has their image placed over it with the similarity value determining the size of the bubble's radius. Luckily, the similarity value is something I did not need to worry about as it is calculated by Last.fm and sent through the API. When hovering over a bubble, a hovering tooltip is displayed next to the mouse cursor showing the artist's name, and similarity value as a percent. On mobile devices, the user will tap each bubble to show the tooltip and tap one more time to visit their visualization. This was my method of implementing an equivalent to mousing over the bubble. The placement of the bubbles is completely randomized. By controlling the size and image of each bubble, the visualization ends up producing various sized bubbles all filled with different images. This makes for an interesting visualization that captures the user's attention. An "onClick" event is assigned to each bubble to provide a link to their own visualizations. Each bubble's "onClick" event contains a route change to the search page with a query parameter containing their name. For example: if Madonna's bubble is present on a visualization, her onClick event would be "/search/madonna".

D3.js or "Data Driven Documents" was the tool that made visualizing the data possible. I began my work with D3 by looking at various tutorials. Initially, I followed a simple tutorial to create a bar chart accepted values from a CSV. This required me to find ways to feed an array into D3 instead of a CSV. I soon found an example that detailed how this could be done. Following the tutorials and producing a test was very quick and resulted in the creation of a bar chart that populated artists across the X axis with their similarity values on the Y axis. After completing this test run, I moved onto creating a bubble chart of my data. The task of creating a bubble chart was a lot more time consuming and required reference to various tutorials. My plan was to combine relevant features from each to create my own unique, functional bubble chart. The first tutorial I followed was Jim Vallandingham's tutorial on how to create bubble charts with D3V4 [18]. This tutorial was very helpful and got me started by providing me with a boilerplate bubble chart that had some cool animations. The tutorial also implemented features such as splitting the bubble chart into smaller charts, but this was unnecessary for my work. I then further adapted my new visualization's code to use values such as my Last.fm similarity values to dictate the radius size of my bubbles. by now, I had a visualization populated with my similar artists. There was just one issue. The bubbles lacked images. To fix this, the next tutorials I found taught me how to add patterns with background images over SVG circle elements. One of the few helpful tutorials I found detailing this was in the form of a YouTube video playlist. The tutorial in question is made by a journalism lecturer for Columbia University by the name of Jonathan Soma [11]. In these tutorials, Soma explains how

```
for(var i = 0; i < data.artists.items.length; i++) {
    const spotifySearchResult = data.artists.items[i]
    if (typeof spotifySearchResult !== "undefined" && typeof data.artists !== "undefined"
    && this.state.artist !== undefined && similarQuery.name.toUpperCase() === spotifySearchResult.name.toUpperCase()) {
        if (spotifySearchResult.images.length == 0) {  //Setting default image if image of artist doesn't exist
            spotifySearchResult.images.push({ url: '/images/default-avatar.png' });
            spotifySearchResult.images.push({ url: '/images/default-avatar.png' });
            spotifySearchResult.images.push({ url: '/images/default-avatar.png' });
        }
        similarQuery.spotify = spotifySearchResult;
        filteredLastArtistArray.push(similarQuery); //Pushing Last.fm artist match to filtered array
        filteredSpotArtistArray.push(spotifySearchResult);  //Pushing Spotify artist match to filtered array
        z++; //Keep count of artist matches found
        break;
    }
}
```

Figure 4: For-loop responsible for the comparison between both arrays.

API Method	Endpoint URL	Reason for Usage
Last.fm Top Artists Chart	/2.0/?method=chart.gettopartists&api_key=YOUR_API_KEY&formatjson	Animated Search Placeholder
Last.fm Artist Search	2.0/?method=artist.search&artist=cher&api_key=YOUR_API_KEY&formatjson	Auto Complete and Similar Artist Search
Last.fm Similar Artists	2.0?method=artist.getsimilar&artistcher&api_key=YOUR_API_KEY&formatjson	Get Similar Artists and Similarity Values
Spotify Artist Search	v1search?qartist&type=artist	Auto Complete and Similar Artist Search
Spotify Artist by ID	v1artists{id}	Get Artist Details and Genres
Spotify Top Tracks	v1artists{id}top-tracks	Get Artist Top Track Previews

Figure 5: Last.fm and Spotify API endpoints used by BandViz.

to overlay images he had stored locally in his development environment. The difference between his example and my project was that I was pulling in image URLs from Spotify. Luckily this was only a minor difference and easily adjusted my code to link my bubbles' background images with the URLs. After this was done, my visualization was pretty close to being usable. The final thing I did was add an "onClick" event to each bubble that redirects to the selected artist's visualization. This was done with a simple JavaScript .push method call on my React App's browser history. The end result was a visualization that allowed for seemingly infinite redirects to new visualizations. This captured my vision of BandViz having a "rabbit hole-like" quality.

Although the visualization itself was functional, I continued to refine it throughout the semester. The next thing I did was add a tooltip that displays when hovering over a bubble. This provides more accurate information on how similar the selected artist is. I later spent a significant portion of time determining the best animation speed for which the visualization would render on the front end of the web app. This was necessary because I found webkit based browsers had significantly worse performance when it came to rendering a large quantity of bubbles that had high resolution images placed over them. I improved D3's performance by lowering the number of frames D3 animated and choosing to use lower resolution artist images Spotify's API provided. The final major addition I made to the visualization was user zoom events. By looking at D3's documentation, I found function calls on mouse events that allow the user to zoom in and out of the visualization as well as move it around. This makes BandViz a truly interactive experience. After countless hours of tinkering with responsiveness, transitions, and interactivity, I had a visualization I was truly proud to display. The end result is a colorful, interactive, and relatively performant representation of my similar artist data.

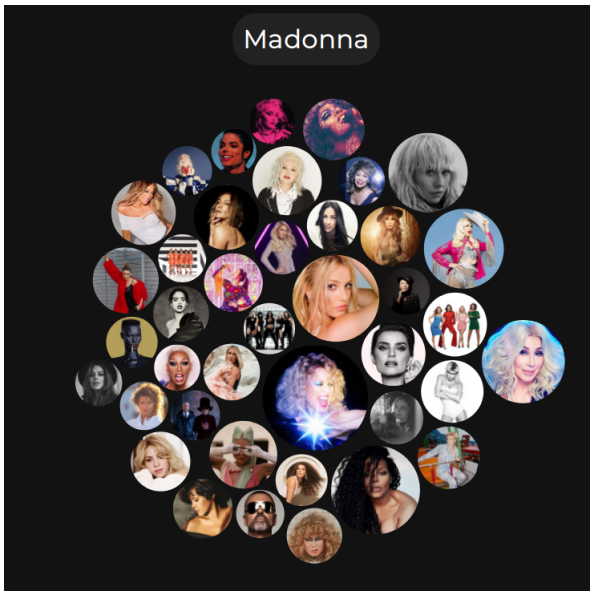

Figure 6: The Final Design of BandViz's Artist Visualization.

4.4 User Interface Design

The design of the web app's user interface is as equally important as the visualization itself. Without a visually appealing, easy to

use interface, the web app wouldn't keep a visitor's interest. By completing this objective, I expected to have a web app that caught my user's attention and maintained it.

To style the web app, I exclusively used SASS style sheets and JavaScript attributes. SASS handles the basic styling of the web app such as colors, layout, and typography. SASS is essentially CSS (Cascading Style sheets) with a more robust feature set. The main reason I chose to use SASS over CSS was that it provides a nested syntax when calling elements, and I can assign variables to values I will repeatedly use. To apply SASS styling to a page or component, I import it in the specific JavaScript file.

```
html {
    background-color: $background-dark;
    -webkit-font-smoothing: antialiased;
}
body {
    font-family: "Montserrat-Regular";
    margin: 0;
    color: $text-dark;
    background-color: $background-dark;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    ::selection {
        color: $background-dark;
        background: $accent-dark;
        }
    ::-moz-selection {
    color: $background-dark;
    background: $accent-dark;
    }
}
```

Figure 7: SASS styling with the help of color variables

The user interface elements created with JavaScript are the React components and D3 visualization. React components are written in JSX. JSX stands for JavaScript XML [21]. JSX is React's way of writing HTML elements while using JavaScript. This allows elements or React components to be assigned to variables and later be rendered conditionally. For example, during my project I chose to display all the genres associated with an artist I queried on Spotify's API. The issue is the data would always change and I didn't know the length of the genres array for the given artist. I solved this by iterating through the API response and pushing the value as a list item element to an array I previously declared. Doing the same thing with regular HTML would have been more difficult. This is because JSX allows us to write values within curly braces while HTML wouldn't.

Because Spotify's API plays a large role in the web app's ability to function, I decided to borrow design queues from Spotify's user interface. This includes rounded buttons and a dark theme. It's fairly easy to explore another website's styling because it's completely available to you. I simply went to Spotify's website and inspected the elements of the website. By inspecting them with my web browser, I can see CSS attributes that are applied to each HTML element rendered on the page. The color scheme of my web app

```
const genres = [];
...
sao.genres.map((i)=> {
    genres.push(<li>{i}</li>)
})
```

Figure 8: Pushing JSX elements to an empty array

follows guidelines specified by Google's Material Design language for dark websites. This means BandViz uses Material's ideal dark background color and accent color saturation level. By doing so, BandViz has a recommended contrast ratio between text and the web app's background which ensures accessibility. Raised elements use a lighter tone of the background to create an illusion that the elements are at a higher elevation [9].

The end result of my web app's design is something I'm pleased with. It's simple and refined. My design causes the user's attention to naturally be drawn to the visualization once it is rendered.

Figure 9: The Final Design of BandViz's Homepage.

4.5 Deployment

Once my the web app could successfully visualize the APIs' data and had an acceptable user interface, it was ready to be deploying to a cloud platform. By the end of my project's work, I expected to have a web app that was fully accessible to anyone on the internet.

After looking through Create React App's documentation on automatic deployment, I noticed a promising option called Vercel. In the documentation, Vercel is described as "...[deploying] instantly, [scaling] automatically... and [requiring] zero configuration" [6]. The icing on cake is that Vercel is free for individual use. This means that the only thing I paid for was my web app since I didn't want it on Vercel's domain.

One of the major benefits of using a service like Vercel is that it syncs with your Git repository. All I needed to do was make an account on Vercel's website and link my GitHub repository. Version control services: GitHub, GitLab, and BitBucket are currently supported by Vercel. To link my repo, I securely logged into my GitHub account through Vercel using single sign on. I then selected the repo I wanted to deploy. Once this was done, Vercel automatically detected my React app and ran NPM build inside the root directory. [20]. By running NPM build on a React App, a deployment optimized version of my site is used [5]. When this was finished, each branch of my repo (main and develop) were deployed to Vercel with a unique URL for each. The domain Vercel provides you with contains your repo's name, branch, and is followed by .vercel.app. If you wish to link an external domain down the road, Vercel allows you to do so. This was something I definitely wanted to do.

Although Vercel's deployment process is described as requiring "zero configuration", I still had to make additional changes to my web app for it to properly run. The biggest change was how the API keys were served to the web app. Originally, I decided to store the API keys locally in a JavaScript file located in a config folder. This proved to be the incorrect method of doing so. What actually should have been done was create environment variables. Vercel steered me in the right direction after I noticed the section in project's settings dedicated to setting the environment variables. Environment variables are keys or values that are meant to be changed based off the system the application is running on. In my case, environment variables were used to deploy a functional web app without pushing the essential API keys to the project's repository. Environment variables are added to the project with the help of a .env.local file. This file only contains environment variables that follows Create-React-App's syntax. To create an environment variable with CRA's syntax, you need to prefix each variable's name with "REACT_APP_". If this syntax is not followed, the variable will not work. For example, my Spotify API key environment variable's name is "REACT_APP_SPOTIFY_SECRET". Now, to ensure the environment variables are not pushed to the repository, .env.local is added to the .gitignore file. After all this is done, the web app's components individually import the variables by accessing React's process.env object. This object is generated from any .env files in the directory of the application.

The final step to get environment variables working with Vercel was adding them through the project settings. This was done by clicking on the settings tab of my project on vercel.com followed by adding the same environment variables I had previously set in the .env.local file. The variables were set for both my develop and production branch deployments. As soon as the environment variables were added to Vercel, BandViz worked as I expected. This time however, anyone with the link to the deployment could access it over the internet.

The last thing I did during my deployment stage was move the site over to my own domain. Initially, the URL for the production deployment was bandviz.vercel.app. I thought it would be more rewarding to have it be simply "bandviz.com". By removing it from Vercel's domain, visitors might consider the site hosted by myself instead of a just being a project spun up on Vercel. The ICANN registrar bandviz.com was purchased through was porkbun.com. This was mainly for its competitive price and low renewal fee. Following the purchase of the domain, Vercel's DNS was added to the domain's list of DNS records. This allows Vercel to assign the custom domain to the project. To do this, I added Vercel.com's ip address as an ANAME and Vercel's nameserver as a CNAME under Porkbun's admin panel for Bandviz.com. The remaining step was to go back to Vercel's project settings and assign the domain name. I ended up assigning BandViz.com to the production branch of my project and adding a subdomain (dev.bandviz.com) for it's development branch. This gives BandViz two unique URLs to use for both development and production environments.

Making my web app accessible from anywhere to everyone was a necessity for me. I felt this way because my schooling was now fully remote and I wanted to show anyone my site how I intended it to be. For me, creating a web application that was only accessible after setting it up locally defeated the purpose of it being a "web" app. By choosing Vercel, BandViz was deployed easily and continuously. It made a world of a difference having a site that deployed with each commit I made to both my development and production branches. Without deployment, I wouldn't be nearly as proud of my project as I am now.

4.6 Testing and Optimization

Without testing or optimization, there was no way my web app would be functioning properly. For this reason, it was essential to test and optimize my web app throughout the entirety of the semester.

Since this React application has countless moving pieces such as API calls and Data visualization, I was constantly bombarded with errors to fix. This usually involved inspecting error and finding the file and line that threw the error. I then thought through what would have gone wrong and attempt to fix it. If I was unable to solve the error on my own, searching for it online was extremely helpful. Oftentimes, fellow developers encountered similar issues and posted to websites such as Stackoverflow.com. Using these resources enabled me to progress throughout the development of my web application at a much quicker pace.

The majority of testing I performed on this web app was functional testing. As mentioned in the goals and objectives section, functional testing is testing all the features of the web app. The biggest priority of my functional testing was making sure my data was correctly visualized. "Is D3 displaying all the similar artists?", "Are these similar artists for the artist I searched for?" these were questions I would often ask myself while functionally testing my web app. After the project's visualization was tested, I moved onto making sure all the pages displayed correctly. This involved navigating to each page of the app and verifying it looked how I intended. To be absolutely sure of this, I used a free cross-browser testing platform known as Lambdatest to test my website across all devices, operating systems, and web browsers. Very often, a web site might not look the same on a different computer. I used Lambdatest to minimize this possibility and maintain a consistent appearance for all users of the web app. Functionally testing BandViz was done over the course of the semester. This was integral to the advancement of my project and the completion of its objectives.

Throughout the development process, I looked for ways to optimize the way my web app operated. This involved cleaning up old

code, refactoring functions, as well as utilizing features of React I previously have had no knowledge of. One of the biggest refactoring tasks was moving my API calls to separate functions. In the early stages of development, I made calls to Last.fm's API and proceeded to request data from Spotify's API immediately after receiving the data. This was fine, but the code quickly got out of hand. I solved this by creating a separate function that queries Spotify's API. I then called that function once Last.fm received a response. Refactoring of code allows for it to evolve and improve even after it has been implemented. By doing this, I reduced the number of errors my code produced and kept the code readable for both me and anyone else.

The last optimization task I worked on was reducing the number of API requests sent for each artist query. Not only do more API requests slow the load time of the application, Spotify's API has a maximum number of requests that can be sent over a period of time. While testing the web app's visualizations, I sometimes found myself hitting this maximum number of requests. This was because every visualization sent individual requests for the artists' images. In the early stages of the project, each visualization was displaying one hundred similar artists. The result of this was that in addition to sending requests to Spotify for the searched artist's genres and preview tracks, BandViz sent an additional one hundred requests each time. If several visualizations were rendered in quick succession, there could be over a thousand requests sent to Spotify. When Spotify receives too many requests from a single source, it returns a 429 status code. Whenever I received a 429 status code, the visualization would fail to render until I refreshed the page. This posed a scalability issue with BandViz if multiple concurrent users were searching for artists. Although solving this would be impossible without changing how the artist images were retrieved, I could lower the total number of artists that were visualized. This would in turn reduce the number of requests sent to Spotify, allowing for more searches to be made before Spotify throttled my web app's requests. After changing the number of queried artists several times, I settled on visualizing a maximum of forty artists, a reduction of over half the possible similar artists. While this should be seen as a short coming in terms of how many requests can be sent, it was probably best that the number of artists per visualization was reduced. One hundred bubbles displayed was often overwhelming to the user. Through reducing how many requests were sent to Spotify artist search, I improved potential scalability issues.

My testing process evolved from troubleshooting JavaScript errors into checking every function of the web app. This ensured that the user experienced BandViz the way I intended. While a period of time was spent refactoring functions, the bulk of my work optimizing was spent reducing the number of queries to prevent scalability issues. This was another extremely important step in my development process because it allows more people to visualize artists at the same time. Through the testing and optimization of BandViz's code, I delivered something that was much closer to a finished product.

5 EVALUATION

My work on BandViz over the course of this semester ultimately resulted in the accomplishment of my project's goal. Although there are aspects of the project that can be improved, I developed a web app that uses Last.fm and Spotify's API data to visualize and discover similar artists. This accomplishment was extremely gratifying. What made this possible was me continuously setting new tasks for my project's work each week of the semester. By setting new tasks, the rate at which I worked never stagnated and I always made progress.

5.1 APIs

The first objective responsible for the completion of my project's goal was successfully querying my APIs. I began this objective by implementing an autocomplete search box to find artists. This required me to set up Last.fm and Spotify's APIs on my web app and determine a way to compare the artists that both data sources had in common. This was done by creating a search query on my user's input to Last.fm and proceeding to send an API call to Spotify for the first six results. If a result was found by Spotify that exactly matched Last.fm's result, the autocomplete's dropdown was populated with that result. The only thing I might change is how often the APIs are queried when the user types in the search box. Currently, API calls are sent on every keypress. While this means the autocomplete is instantaneous, a user could send too many calls to the APIs, potentially exhausting the site's API quota provided by either Last.fm or Spotify.

The task of retrieving the similar artist data was possibly the most important aspect of this project. I did this by sending a "similar artist request" to Last.fm for the artist that was originally searched for. The response is an array of 100 similar artists and their similarity ratings. This query is extremely powerful as it only requires one API call to return all the similar artists. The issue was that Last.fm stopped providing the images of each artist. This forced me to send a request to Spotify's API for each artist in this list. For this reason, each time I got the images for my similar artists, 100 API calls to Spotify were made. This approach is far from ideal. I believe that this method of sending multiple requests to Spotify for one Last.fm response is necessary to retrieve the web app's data, but it could easily exhaust the API quota for my Spotify API key. A potential solution to this would be for Spotify to create an API endpoint for multiple artists in one request. Unfortunately, this doesn't exist and probably won't ever. There only exists an endpoint to retrieve multiple Spotify artists with their individual artist IDs. By the nature of my web app, we don't know any of the similar artists' IDs because it is Last.fm that provides these artists. This issue comes down to my web app requiring more data than Spotify can return in a single request.

5.2 Visualization with D3

The execution of my web app's visualization exceeded my expectations. Seeing that I had no prior experience with D3.js, I took a gamble when I chose to use it in combination with my React app. During the initial stages of implementation, I spent the majority of my time absorbing D3 tutorials and looking through its documentation. After providing D3 with access to my web app's DOM, I was relatively quick to render visualizations of my data. I ended up with a visualization of my similar artists that was close to what I originally wanted it to look like. One of the only issues I

encountered was the visualization's performance when rendered on a webkit-based web browser (Safari). When rendering 50+ similar artist bubbles, the rendering animation of my visualization became jittery and less visually appealing. My solution for this was reducing the number of frames D3 rendered for each visualization. This reduced strain on Webkit-based browsers. Sadly, the performance on Webkit/Safari is still less than ideal. One thing I wish this visualization did better was optimize the performance for every web browser available.

5.3 Deployment

Out of all my objectives, the web app's deployment probably went the smoothest. By using Vercel, I could link my project's GitHub repo then automatically build and deploy the web app with each commit I pushed. This resulted in an extremely painless process of completing the objective. The only hurdle I faced during this objective was setting the environment variables. I originally wasn't aware of how Vercel and Create-React-App handled API keys. For this reason, I initially assigned them inside a JavaScript file. If I kept it this way, Vercel wouldn't have had access to my keys. During Vercel's set up, I learned to change my API keys over to environment variables. This took a little bit of time to change all the references to my API keys and I wish I knew to use environment variables from the very beginning.

5.4 Optimization

While I improved my code over the course of this project several times, I believe it is not nowhere near perfect. I am still learning new things in React every day and I know there are aspects of my project I could have done better. Whether it's better utilizing the React lifecycle methods, or simply following best practices, there is a lot of room for me to improve this web app's code. I hope to further improve the performance of this web app by transitioning it to the Next.js framework which uses server side rendering.

6 CONCLUSION AND NEXT STEPS

With online music streaming services still continuing to mature, few options currently exist to discover the millions of musicians that reside on them. With this project, I set out to deliver a web application that combined artist data from Last.fm and Spotify's APIs and visualized it to an end user. Through the completion of my various objectives, I did just that. The final product is an easy to use web application with a clean interface and an engaging, interactive user experience. By providing the user with a seemingly infinite number of similar artist visualizations, this web app maintains user attention and throws them down a "rabbit hole" of music discovery. Because I'm yet to find a web app with the same feature-set of mine, I could see my web app gaining a user base in the near future if I continue to improve it. My experience working on this project was truly eye-opening both in terms of the power of APIs, and how data visualizations can be made interactive.

I plan to continue to improve my project for some time with the addition of a new framework and variety of optimizations.

Firstly, I want to improve the efficiency of my API requests. Because I needed to query two APIs and not one, I am sending 50+ requests at a time when searching for similar artists. There may be an easier, more scalable way of doing this. One possible solution would be implementing caching of frequently searched artists on BandViz's server. This would reduce the number of requests sent to the APIs.

The next thing I'll be working on is improving D3's performance on Webkit/Safari browsers. This might not be entirely feasible, but I feel a need to look into more performant ways of rendering my visualization. It is possible other developers have found ways to reduce the load of D3 visualizations on these browsers.

To utilize server side rendering, I will be moving BandViz from React.js to the Next.js framework. This will improve the web app's security, search engine optimization, and performance.

Lastly, I want to improve this web app's overall code. I'm always learning new methods and best practices with React and I know there are aspects of my web app that can be improved.

These improvements to BandViz presented themselves throughout the work of this project and I simply didn't have nearly enough time to implement them. I can't wait carry on with improving my work and see BandViz's potential grow. I never thought I'd see my web app become something I continue to enjoy using and sharing with other people. I'm excited to see how far I can bring this.

REFERENCES

[1] Chaven, Y. 2020. How to Create a Spotify Music Search App in React. https://levelup.gitconnected.com/how-to-create-a-spotify-music-search-app-in-react-1d71c8007e45
[2] Data-Driven Documents. 2020. Data-Driven Documents - Wiki. https://github.com/d3/d3/wiki
[3] Data-Driven Documents. 2021. What is Functional Testing? https://www.guru99.com/functional-testing.html
[4] FACEBOOK, Inc. 2021. Create-React-APP - Adding Custom Environment Variable. https://create-react-app.dev/docs/adding-custom-environment-variables/
[5] FACEBOOK, Inc. 2021. Create-React-APP - Create a New React App. https://reactjs.org/docs/create-a-new-react-app.html
[6] FACEBOOK, Inc. 2021. Create-React-APP - Deployment. https://create-react-app.dev/docs/deployment/
[7] FACEBOOK, Inc. 2021. Create-React-APP - File Structure. https://reactjs.org/docs/faq-structure.html#is-there-a-recommended-way-to-structure-react-projects
[8] FACEBOOK, Inc. 2021. Create-React-App - Folder Structure. https://create-react-app.dev/docs/folder-structure/
[9] Google. 2021. Material Design - Dark Theme. https://material.io/design/color/dark-theme.html
[10] INTERNATIONAL BUSINESS MACHINES CORPORATION. 2021. Understanding API keys. https://cloud.ibm.com/docs/account?topic=account-manapikey
[11] Jonathan Soma. 2016. Using images in D3 bubble charts. https://www.youtube.com/watch?v=FUJjNG4zkWY
[12] Kelhini, F. 2019. Axios or fetch(): Which should you use? https://blog.logrocket.com/axios-or-fetch-api/
[13] LAST.FM, Ltd. 2021. Last.fm API - Introduction. https://www.last.fm/api/intro
[14] REACT Training. 2021. React Router - Philosophy. https://reactrouter.com/web/guides/philosophy
[15] REDHAT, Inc. 2021. What is an API? https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces
[16] SPOTIFY AB. 2021. Spotify - App Settings. https://developer.spotify.com/documentation/general/guides/app-settings/
[17] SPOTIFY AB. 2021. Spotify - Web API. https://developer.spotify.com/documentation/web-api/
[18] Vallandingham, J. 2016. Creating Bubble Charts with D3v4. https://vallandingham.me/bubble_charts_with_d3v4.html
[19] Vercel Inc. 2021. Create-React-APP - Adding Custom Environment Variable. https://vercel.com/docs/environment-variables
[20] Vercel, Inc. 2021. Vercel - Build Step. https://vercel.com/docs/build-step
[21] W3Schools. 2021. React JSX. https://www.w3schools.com/react/react_jsx.asp