BeatBuilders Spotify Playlist Generator Design Decisions and Setup

Arjun Thomas, Jason Burghart, Nick Ospelt, Ross Imbrock, Toby McGuire, Tom Nemeck, Tyler Jachimski, Vishal Kumar

Design Decisions

Utilizing Firebase Authentication

At the beginning of our project, our group decided to use Firebase Authentication for our web application's login and sign up pages. We didn't have prior experience implementing Firebase Auth, but we were aware that it would simplify user authentication. It did simplify authentication. Instead of creating a SQL data store and validating users in a USERS table, we were able to call the Firebase Auth api. We saved a lot of developer time and simplified our codebase. Firebase Auth also automatically secured our users' credentials. We didn't have to implement password hashing.

Utilizing Firestore

Our group decided to use Firestore to persist users' playlist data because we were already using Firestore Authentication for our login and sign up pages. Now, we can monitor all our users and their playlist information in the Firebase console. Firestore is a no-SQL data store. The group didn't have prior experience working with no-SQL data stores, but Firestore was intuitive to use. Data is stored in documents, which are grouped into collections. Information is queried by specifying the document and collection path. It was simple to implement CRUD operations with Firestore. Google had excellent written documentation and videos detailing simple CRUD implementations. In hindsight, Firestore was an excellent design decision for a persistent data store.

Docker Compose Configuration

The team decided to use Docker-Compose to compile the many working parts of this project. The docker-compose file defines the different containers for the different services within the program. The program consists of 5 containers that rely on each other to build the program successfully. The web service launches the Next.JS webpage that the frontend website is hosted on. The elasticsearch service hosts the elasticsearch server that is queried through the flask_server service when the user submits a request, therefore the web service depends on it. The python_elasticsearch service connects the user's preferences with elasticsearch, therefore it depends on the elasticsearch container and the web service depends on it. The flask_server service is used as an api endpoint for the user's queries to be sent to the LLM and elasticsearch and returns the generated songs to the frontend, and the web service depends on it. The proxy_server service allows for the Deezer API to avoid a CORS error for when the Spotify API does not have a song preview available, and the web service depends on it. Dependencies in

the docker-compose mean that the container is launched after the containers it depends on. This allows our program to launch in the way that every service has its required dependency services up and running before it launches. So all of the working parts of the main web service are launched before the web service itself.

Langchain and the Gemini API

Our group wanted to leverage the reasoning capabilities of LLMs to map the abstract sentiment of user input to the database's song parameters. It is no secret that LLM outputs introduce a lot of variability and tend to hallucinate. So the group leveraged Langchain and its prompt engineering tools to instruct the LLM to take user queries and construct suitable elasticsearch queries to run on the song database. The Gemini API integrates easily with langchain and avoids the long inference times that local llms like Llama2 induce. Although the generator is limited to 60 user queries per minute due to the Gemini API, it provides song recommendations within seconds.

Spotify API Integration

The Spotify API was used in several different ways in our project. On a base level without signing into Spotify with our application, the Spotify API generates an access token when the page first loads to be used for song information. Then, when a user submits a request and the song titles are returned from the database, the Spotify API is queried with the song title and artist and returns a large JSON of all songs that it finds. From that, our program takes the best matching result from that and creates a Track object with the song's title, artist, URL to its cover art, URL to its 30 second preview, and the ID from Spotify. Then the Tracks are converted to SongCards which are displayed on the main page. Also, if the Spotify API does not have a song preview URL for the song, the program uses the Deezer API and pulls a preview link for it. Once you have created your desired playlist you are able to login to Spotify and link your account with our application. Then, a new access token with elevated privileges is generated and it is able to add your playlist to your personal Spotify account.

Next.js Integration

We decided on building our application utilizing the web development framework Next.js. A few team members had prior React experience, so the learning curve for Next.js, which is a React Framework, was relatively benign. Next.js adds a variety of features to vanilla React, such as server-side rendering and static site generation, which can greatly reduce page loading times and improve security. It also made the app routing significantly easier, and enabled the team to use NextAuth.js, an open source authentication library for Next.js, to seamlessly integrate the Spotify Authentication and manage the user access tokens. Using Next.js also enables the team to easily deploy the site for free with Vercel. The team was also drawn to the popularity of Next.js and adding a new framework to their toolkit.

Setup Instructions

Some prerequisites are required in order for the Spotify Playlist Generator to run.
1. Install Node.js version 20.11.0
2. Install Docker Desktop (or Install the latest versions of Docker and Docker-Compose)
3. Run on a Unix based system like Linux or MacOS ideally, to have Makefile functionality

Once you have installed the required prerequisites, the rest of the dependencies can be installed by our program's Makefile functions.

1. To begin, launch Docker Desktop, or Launch Docker on your system.
2. Then, navigate to the BeatBuilders directory where you have saved our codebase via `cd BeatBuilders Code`
3. To run the program, run `make start` into your terminal window
4. To tear down the application run `make stop`
5. To clean the application's volumes for a hard reset, ensure the application is not running and run `make clear`

If you are running on a Windows system, you can run our application with these commands instead.

To start the application run `docker-compose up -d --build --remove-orphans`

To stop the application run `docker-compose down --remove-orphans`

To clean the application run `docker system prune -af --volumes`