SongSift: A Playlist Generator for the Seldom Played

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Paul Wesley Stepler

Spring, 2024

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Upsorn Praphamontripong, Department of Computer Science

Introduction

For my capstone project this semester, I created a web application called SongSift. It is an app that connects to the user's Spotify account, and its primary purpose is to use it to generate new and interesting playlists based on user preferences.

A quick Google search will reveal there are already plenty of playlist generators out there, so why use this one? SongSift stands out in that it is designed to introduce the user to artists they have never heard of before. Behind the scenes, Spotify assigns each track a popularity score, on a scale of 0 to 100, based on how many streams that song is getting. When a user generates a playlist using SongSift, there are several different parameters they can use to fine-tune the results. One of those parameters is the popularity score—whatever number the user inputs will be the maximum popularity score a song Spotify suggests can have. This way, users can generate playlists containing songs they have likely never heard before. Trying to find new music on Spotify is often an overwhelming task—and not made easier by the fact that much of the content on Spotify-generated playlists for you are songs the user already knows—SongSift makes this process a little bit easier.

Additionally, SongSift has a few key accessibility features. Within the HTML code, each image item contains an alt tag, which provides a description of the image for those who are using a screen reader software will be able to know what the image is. Each webpage is also designed to be easy to read, with a white background and black text, to make it easier for those with vision problems to use SongSift effectively.

Overview

To give a basic overview of how SongSift works, the user will start on the home page, which will prompt them to log in with their Spotify account. After the user logs in with their

Spotify account, they will be redirected to the SongSift homepage, which will display the first fifty playlists in their library. From the homepage, the user can navigate to any of their current playlists by clicking on its image, navigate to their Spotify profile by clicking on their Spotify profile photo in the top right corner, or go to one of four pages: "Generate Playlist", "How Does it Work?", "About", or "Feedback". The "Generate Playlist" page is where the user can create their new playlist, which we will return to shortly. The "How Does it Work?" page explains how to use the generate playlist page and includes a screen recorded tutorial, the "About" page contains some information about me and why I created SongSift, and the "Feedback" page contains an email the user can use to send any feedback they have about the site.

When the user navigates to the "Generate Playlist" page, they will see ten input fields: Artists, Genres, Tracks, Total Songs, Popularity, Acousticness, Danceability, Energy, Liveness, and Tempo. Most of these are optional parameters the user can use if they have a more specific type of playlist in mind—the only ones that are required are Total Tracks, and at least one Artist, Genre, or Track. Once the user has entered their desired parameters, they will click the "Generate Playlist" button, and the suggested tracks will populate the right side of the screen. The user can then generate another playlist based on those parameters, or if they like what they have, they can create it. Once they have provided a name and option description and click create, they will be redirected to their homepage, where they will see their new playlist, which will also be available to listen to directly from Spotify.

Design

For the overall design of SongSift, I used a modified version of model-view-controller (MVC)—a very well-known architectural design pattern for web development. MVC applications are split into three parts: the models, which are often classes representing data from

a database; the views, which are the actual webpages the user sees; and the controllers, which control what data the users see. For this project, because I retrieved all my data directly from Spotify, using models was not necessary, but using views and controllers was. SongSift is broken into three controllers and their corresponding views: the welcome controller, the home controller, and the playlists controller.

The welcome controller handles everything related to the login process. The only view associated with this controller the user sees is the landing page when they first navigate to the site, but there are a few other views the user cannot access which contain necessary code for authentication. Behind the scenes, the welcome controller sends a login request to Spotify, then parses through Spotify's response and configures everything so the authenticated user can access SongSift.

The home controller handles all other views not related to playlist generation. Encapsulated within the home controller is the home view the user gets redirected to after a successful login, the "About" view, the "How Does it Work?" view, and the "Feedback" view. The home page displays the playlists in the current user's library, so the controller compiles all the information about the playlists into a few lists and passes them to the view. The other views do not require anything to be passed to them, so the controller simply renders the views to make them visible to the user.

Finally, the playlist controller handles the views related to playlist generation—namely, the generate and create playlist view, and one additional view that displays the valid genres a user can input into a playlist generation query. For playlist generation, the controller takes all the user inputs and compiles them into a query string, which is then sent in an API request to Spotify, and then displays the returned recommendations on the page. For playlist creation, the controller

takes all the recommended tracks, as well as the user inputted name and description for the playlist and sends a request to the API first to create a new, empty playlist with the inputted name and description, and then a second request to populate that playlist with the recommended tracks. Lastly, the genres view simply displays the results of an API request for available genres to use in a playlist generation query.

Implementation

I implemented SongSift using the C# programming language and the ASP.NET 8.0 framework. There are many other languages and frameworks I could have used—most notably Python with either the Django or Flask framework, but I decided to use C# and ASP.NET for a few reasons. Firstly, they are a powerful and efficient language-framework pair for web development projects, and it is an industry-standard development environment. Secondly, while I have extensive experience Python, the Django and Flask frameworks, and other languages such as C++, I had never done any legitimate programming in C# before, so using it for SongSift was a great opportunity to gain experience with a new language. Additionally, in my upcoming professional career, I will be developing in C#. So in addition to learning a new language through this capstone project, I am also better prepared for a successful transition into the industry.

In addition to a programming language and a framework, I used the Web API provided by Spotify to get and process data. Spotify's API is extensive and allows developers access to all kinds of data through HTTP GET requests, including individual tracks, artists, albums, and playlists. It also allows for the creation of data through HTTP POST and PUT requests, such as creating and modifying playlists.

Regarding the implementation of the application itself, first and foremost I wrote the code

to authenticate the user with Spotify. This included sending a login request to Spotify using the client ID for SongSift provided by Spotify, then retrieving the access token Spotify returned in response. After this was completed, I roughly followed a pattern of writing the backend code to retrieve and parse the necessary data from Spotify using the API, then writing the frontend code to display this data to the user in the views. I created a class called APICalls, and each API call I used was encapsulated in its own method within that class. When I needed to make an API call, I called those methods in the appropriate controller, which passed the data to the appropriate view.

Testing and Validation

Throughout the entire implementation process, I thoroughly tested SongSift to ensure that it worked properly and ran smoothly. To give a few examples of how this was done, most of SongSift's views require a user to be logged in before accessing them. To enforce this, I added checks to make sure the user trying to access these views was logged in, replaced the original view with an error message if they were not, then checked that a user who was not logged in could not access those pages and the error message loaded correctly.

Additionally, since SongSift pulls all its data from the Spotify API, it was necessary to validate the data that would be displayed on the pages. So, any time I made a change to anything involving data retrieval on the backend, I would ensure that all the data being displayed was still correct. I also included hyperlinks to the actual Spotify page for every song and playlist that displays on SongSift, so I thoroughly tested those links as well to ensure they directed the user to the correct destination.

Finally, the playlist generation feature of SongSift requires parsing through user-inputted data, so I added input validation to each of the input fields to ensure users could not send anything malicious to the server. For each field that requires only a number for input, numbers

are the only type of input allowed, and for the fields that ask for a track, genre, or artist to base the recommendations on, numbers and letters are the only type of input allowed, to prevent a user from launching some sort of injection attack using special characters. If a user attempted to input any unpermitted characters, or entered the name of a track, artist, or genre that does not exist, the API request to Spotify will simply not return anything, and the user can try again.

Challenges

Throughout the development process, I came across some significant challenges. The first and probably biggest challenge I faced was figuring out how to access user data through their Spotify account. This was a crucial step that would allow the site to display user playlists and allow for the creation of playlists. After much trial and error, I eventually determined I was authenticating users and trying to access API data using two different methods. To authenticate the user, I was using a package installed using Visual Studio's package manager, and to access the API I was attempting to use SpotifyAPI-NET, an open-source library meant to make working with Spotify's API in C# easier. To access Spotify's API, an access token is required—a string of random characters Spotify sends in response to a successful authentication request. The method I used to simply allow users to login using their Spotify account did not make this token available on the backend. So, while I was able to successfully authenticate a user using their Spotify account, that authentication did not allow access to API data. To solve this, rather than trying to combine these two methods of authentication, I requested access to Spotify's API using more basic HTTP requests. Once I had retrieved the access token and other necessary credentials using this method, I could successfully access data from the API.

A second problem I ran into is that the SpotifyAPI-NET tool is not very well documented and does not allow for much adjustment on calls to the API. This meant to figure out which

methods would get me the data I needed, I had to look through all the available methods within Visual Studio and piece together which one was for that specific call. Additionally, many of Spotify's API requests involve various parameters that can be adjusted to fine-tune the results it returns and includes default values for these parameters if a value is not specified. SpotifyAPI-NET provided no way of adjusting these default values, making its use very restrictive for certain calls. Because ofthis, while a few of my methods still use SpotifyAPI-NET, I ultimately wrote most of the requests myself, once again using HTTP GET and POST requests, as this allowed for much more versatility for the parameter values in the requests being sent to Spotify.

Finally, towards the end of the development stage, I shared SongSift with some of my friends for beta testing. This ended up being quite useful, as it not only revealed various smaller bugs, but also a very serious one: two users could not properly use SongSift at the same time. When multiple users were logged into the site on their respective machines at the same time, the data for whoever logged in last was showing up on everybody's screens. This meant that if a user logged on, then refreshed their home page after a second user logged on, they would see the second user's playlists, not theirs. After looking through my code I realized I was storing critical, user-specific information as global variables. This resulted in these variables being overwritten each time a new user logged in, even if another user was still actively using the site. I eventually solved this by taking these variables and shifting them from global scope to HTTP context scope. These variables are specific to any given web session, meaning several users can access the site at the same time without data overlapping or being overwritten.

Takeaways

Now at the end of this project, there are a few key takeaways. Before creating SongSift, I had no experience developing using C# or ASP.NET. Initially, I wanted to do this project in

Python, a language I am very familiar with, but my advisor recommended using a language and framework I had not used before, as part of the capstone experience is learning something new in your field. I gained valuable experience coding in C# this semester, and now feel much more prepared to step into industry work after graduation. I also had very minimal experience working with APIs before this project and did not fully understand how they worked. Since the bulk of this project consisted of working with the Spotify API, I now have a much better understanding of both how they work and how to work with them. Finally, there were a couple instances where I had to deal with an error or problem that had little to no documentation online, meaning I was on my own to figure it out. While this can be quite frustrating at times, it ultimately gave me a better understanding of both the environment I was developing in, and how to effectively research answers to problems no one has a full answer to.