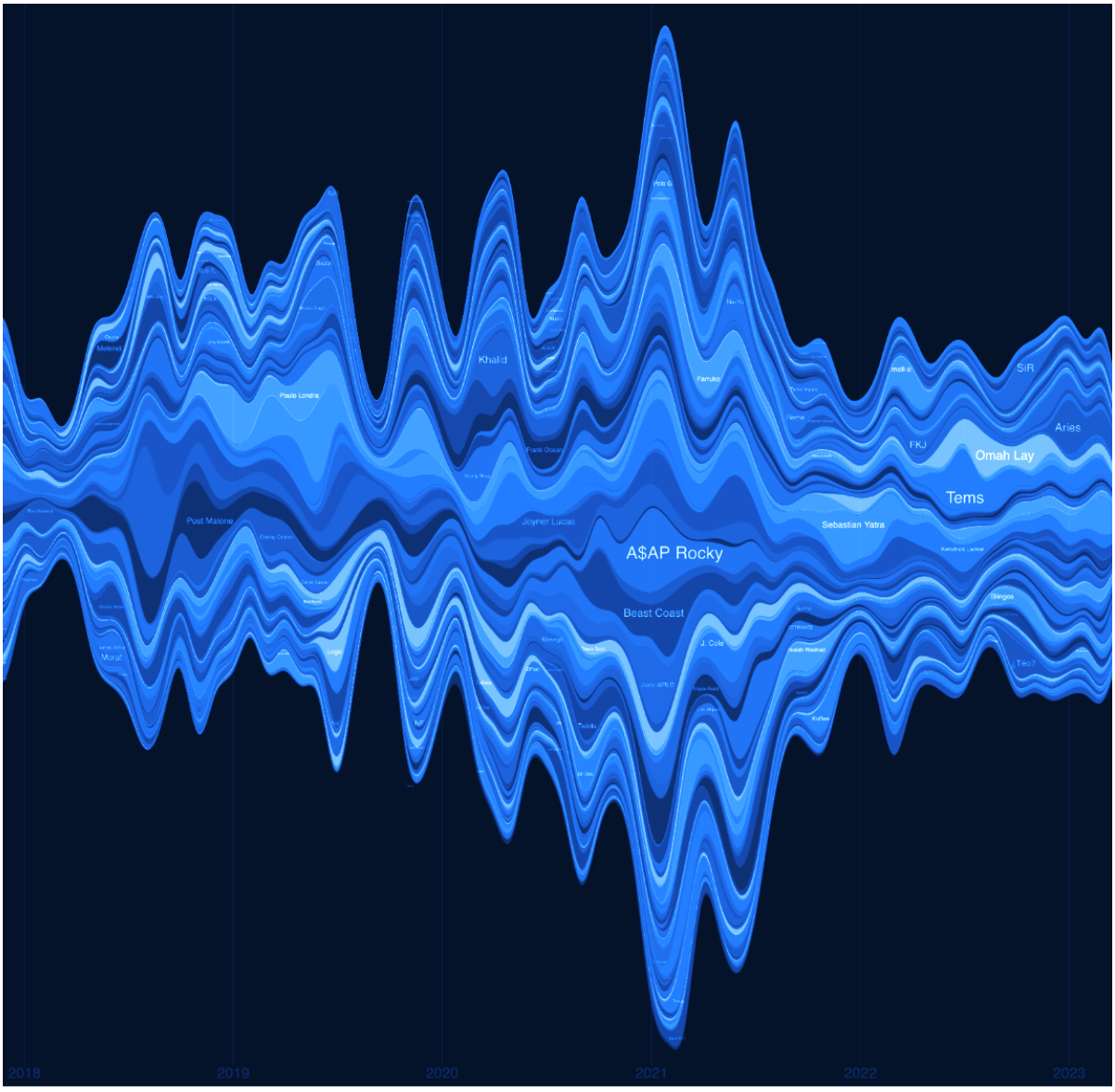
Ripplify: Your Music Taste At A Glance

Erik Kieran Boesen

CPSC 490

Advised by Professor Stephen Slade

December 1, 2023



Abstract

For my CPSC 490 senior project, I have built a sophisticated data visualization platform that enables users to examine their Spotify music listening history through beautiful stream graphs. Spotify allows customers to download records of all the music listening that has ever occurred on their account, but there are few tools available online to visualize this information. I present a highly refined and visually pleasing graphing tool that enables users to see the entire story of their music taste's evolution over time at a glance, and discuss the challenges and best practices I've identified while developing the service.

Background

There are many extant websites that offer Spotify music analysis features, such as Receiptify, Icebergify, Festify, and countless others. Such sites enable users to authenticate with their Spotify accounts, and then generate an image showing the user their favorite musical artists stylized as a receipt, an iceberg, a music festival poster, etc. However, Each of these websites is based on performing a query to Spotify's API, which provides precomputed records of users' top 10 or 50 artists over the course of the prior month, six months, or the entire history of the account. This information is doubtless a delight to explore. However, Spotify doesn't show all its cards through its API. In truth, the company stores a profound amount of data, including a detailed every single song play that a user has ever performed. Each record holds:

- The ID, name, artist, and album of the track (or episode and show name, for podcasts)
- The timestamp the song was played at and the number of milliseconds for which it was played

- The country, IP, operating system, and (if the play was executed through a browser) a user agent string from which the music was played
- The reason the song was played: was it directly clicked on, played at the end of the previous song in a series, played as the result of the user clicking on a playlist, arrived at after skipping another song, arrived at after the user clicked the "back" button, etc.?
- The reason the song stopped playing: did it end, was it paused and never returned to, did the user skip it, did the user quit the Spotify app entirely, etc.?
- Was the song played while in shuffle mode, while offline, or in private listening mode?

This data is likely used behind the scenes by Spotify to compute the far more limited top-artists and top-tracks information that they provide through their API. The full dump of "Extended Listening History" can be obtained through the Spotify privacy settings and submitting a request. Unfortunately, this data can take as long as 30 days for Spotify to send to a user, limiting the virality of any platform based upon it. Nevertheless, the ability to see a complete visual analysis of one's taste evolving over time, with new artists and songs entering one's life and then fading to the background or persisting across many years of life experience, is an alluring notion for many. It has already inspired a number of users to request their data and begin generating content through my website (more on this later).

When investigating potential approaches for visualizing the entire detailed history of a Spotify user's music listening behavior, my mind quickly was drawn to the "stream graph" data visualization method described in Byron & Wattenberg (2008). There fortunately exist a variety of open-source libraries that enable relatively straightforward creation of stream graphs. This includes an official package written by Lee Byron, one of the authors of the 2008 paper, which implements the graph format in Processing, a Java-like language (Byron 2010). Other third party

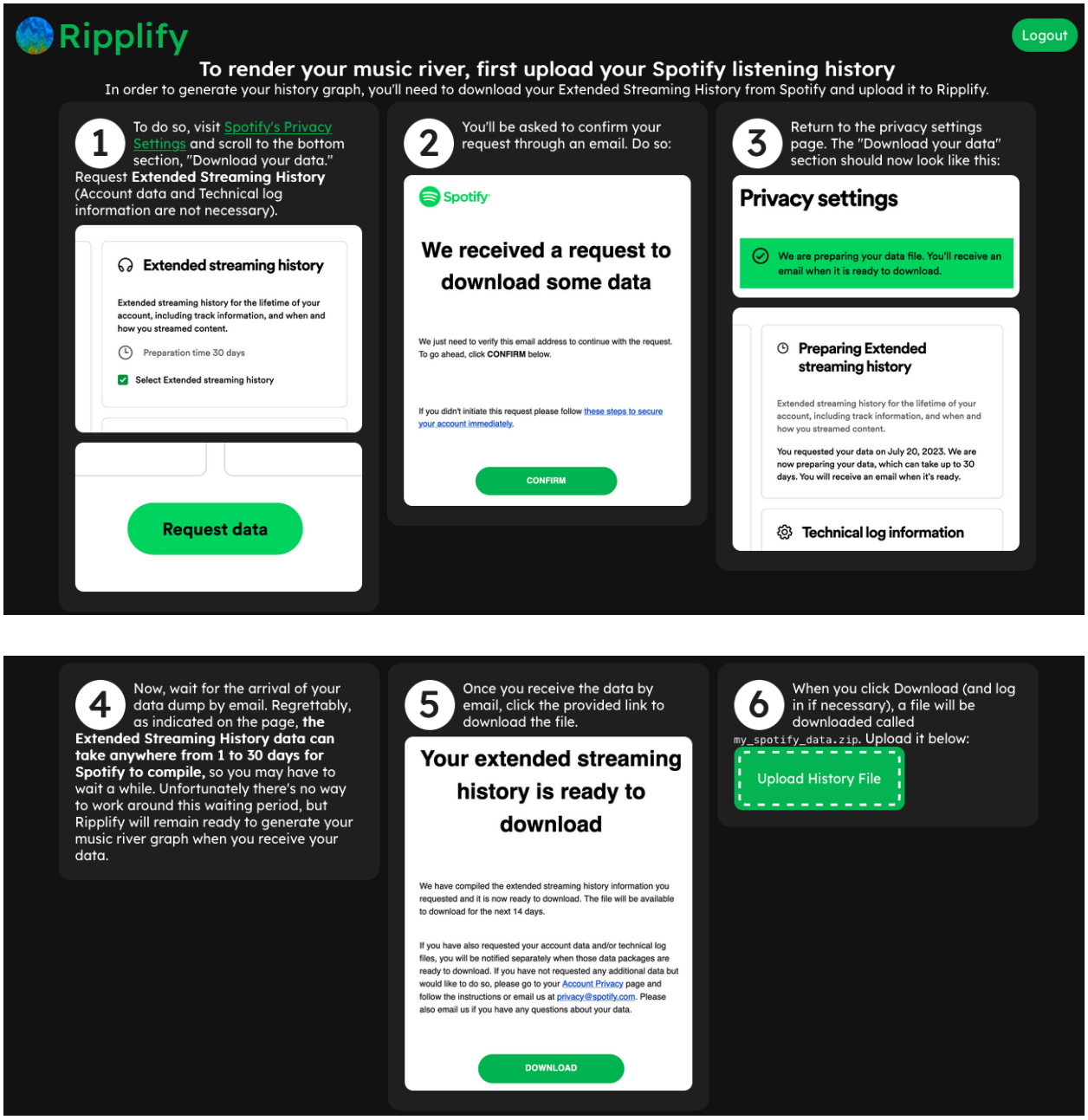
libraries include d3-area-label, which extends D3.js's implementation of the stream graph visualization algorithms (which D3 refers to as "area" and "stack" graphs), and adds labeling functionality. I chose to use D3, because 1) I originally envisioned the application as web-based, and 2) I appreciated D3's use of SVG vector graphics as opposed to bitmap-by-default.

Technical Description

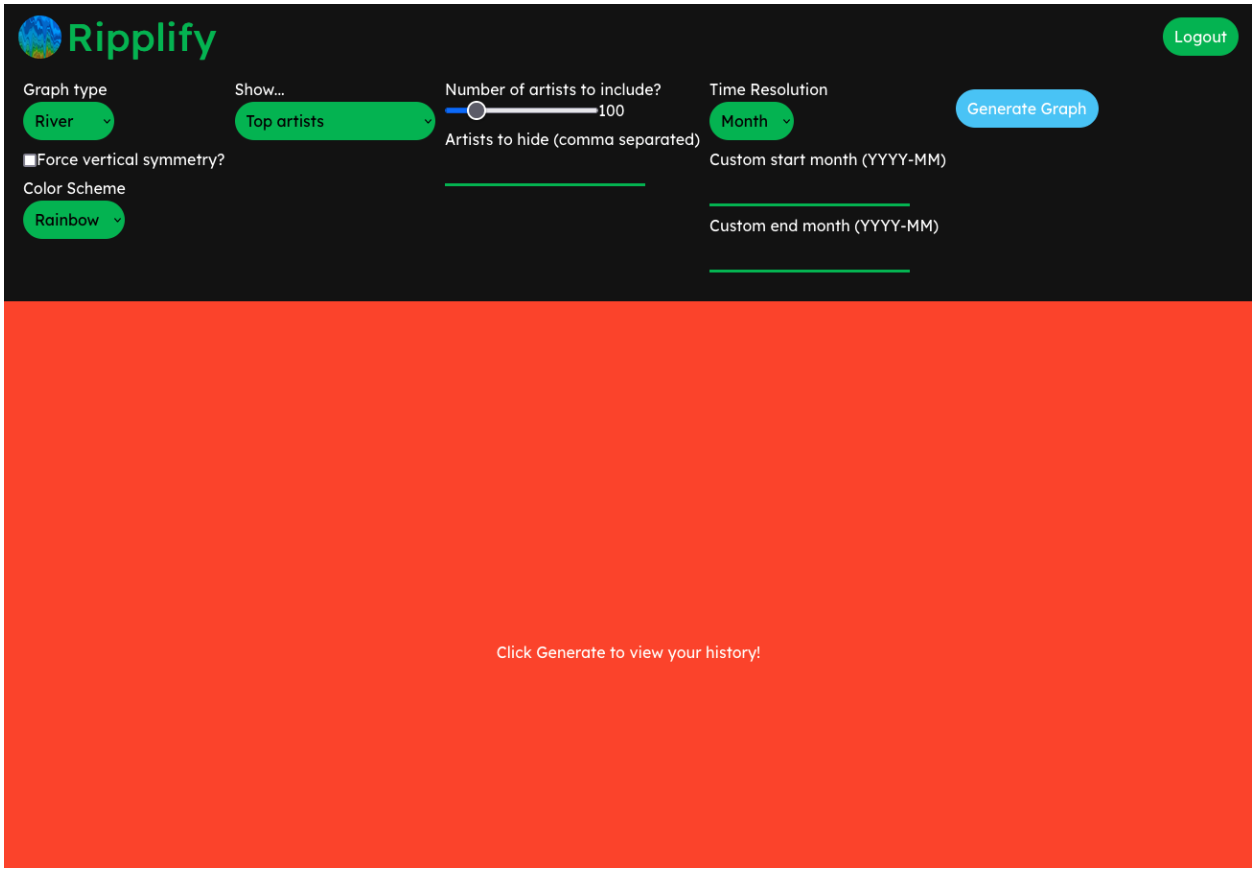
Ripplify is a complex application with three primary components, the front-end, backend API, and rendering server. These components interact with each other through JSON/REST API requests.

Part 1: Front-End

Users of Ripplify interact with the service through a front-end website written in ReactJS. This website is hosted on GitHub Pages, which conveniently is free to use as a developer. The website is accessible at https://ripplify.io. It interacts with the Spotify API to perform authentication, obtaining an OAuth access token from that API and then forwarding it to the backend server to obtain a long-living Ripplify JWT token that is not dependent on maintaining continual authentication to Spotify. Only a single Spotify API query is made per user, to obtain the user's name, email, and profile photo (more on this in the Ripplify API section). When a user is authenticated, their token and identifying information is stored in the front end's localStorage. If a user is logged in but has not uploaded their data, a guide explaining how to do so is displayed:



When the user uploads their history file (which is delivered to them as a zip file by mail), the input animates while the file uploads to the API. Following this, the homepage content is replaced with an interface where users are able to choose customizable settings for their graph images.



Users can tweak rendering options such as graph type, which can be either “River,” “Hills,” or “Stretch”:

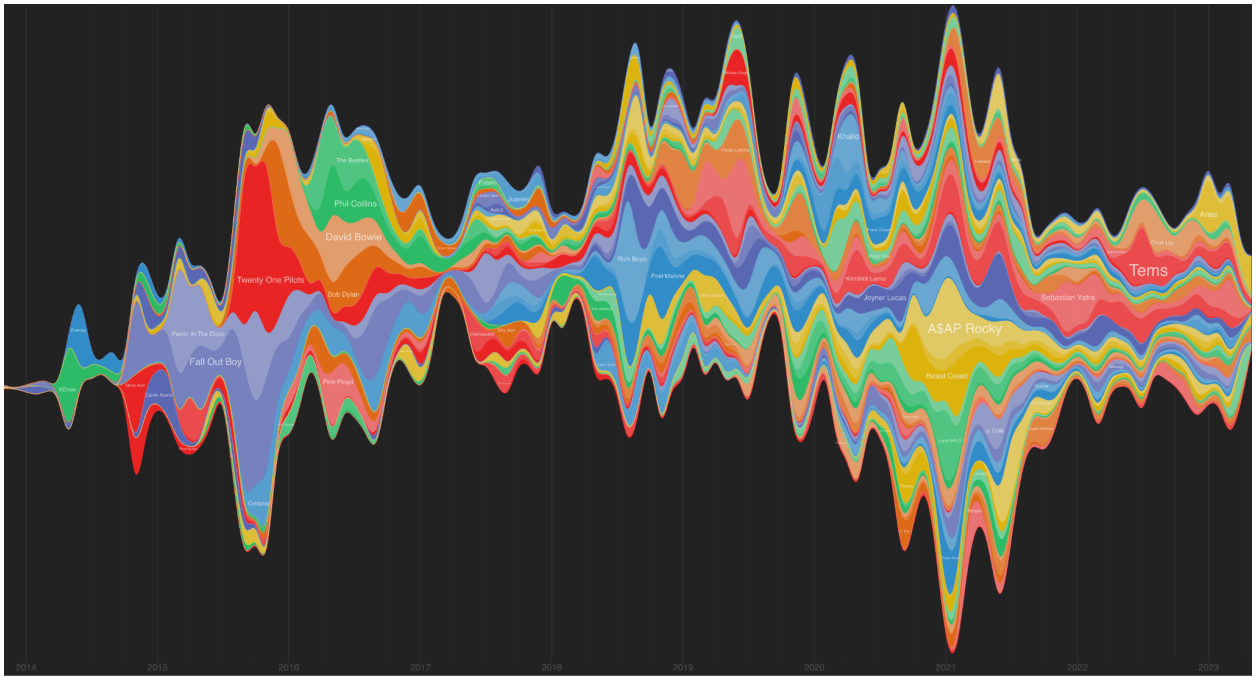


Fig. 2: The "River" graph style

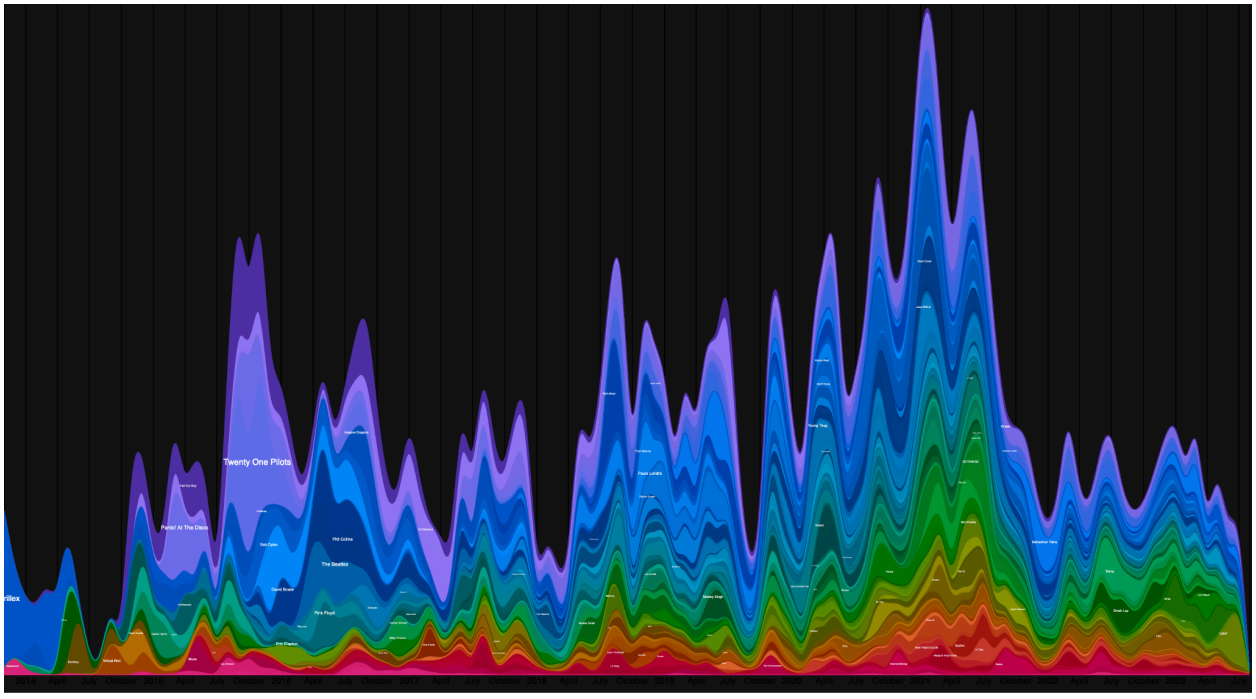


Fig. 3: The "Hills" graph style

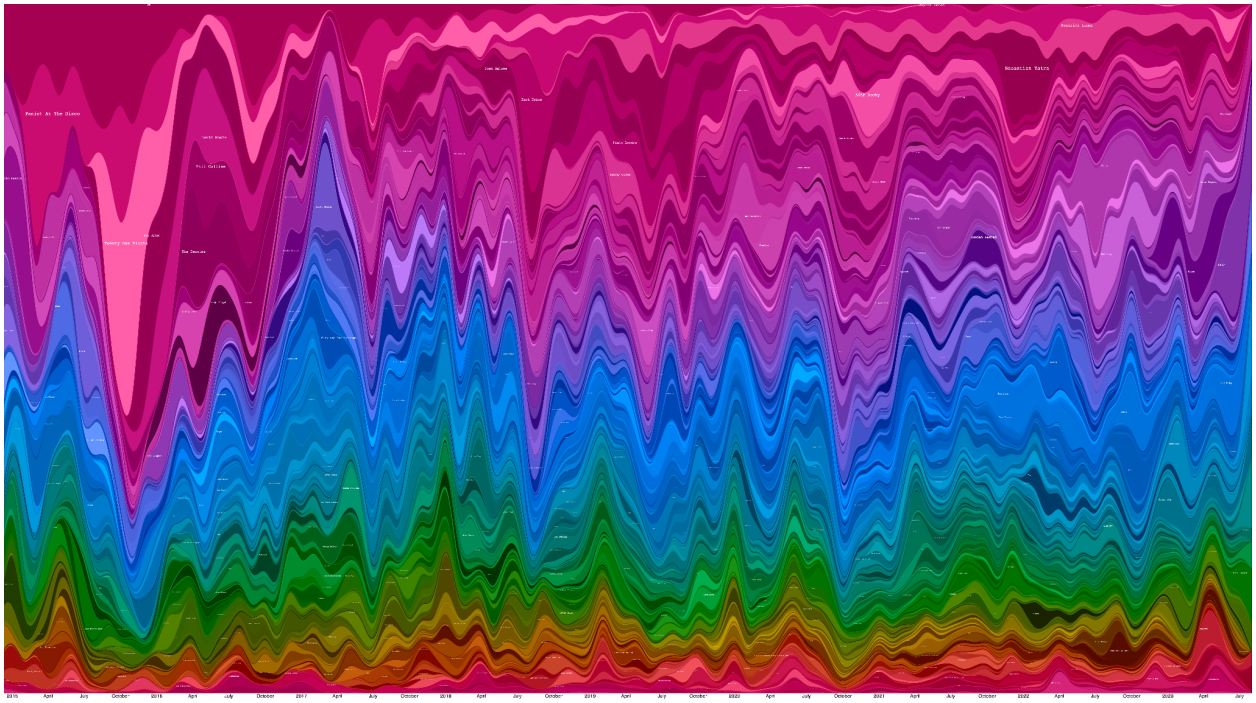


Fig. 4: The "Stretch" graph style

For the river mode (first), users can force the graph to center on the middle of the Y axis, or allow it to "wiggle" arbitrarily to add visual interest. Users can also configure:

- "Classic" or "Rainbow" color scheme (with the river graph above being in classic theme and the others in rainbow)
- Time resolution (daily or monthly)
- Start and end dates (defaulting to the entire history of the user's account)
- Whether to view the top artists the user has listened to or their favorite songs from a specific artist
- How many artists or songs to include in the graph, as too few may leave out artists and be too simple to seem like a sufficient taste descriptor, but too many can make the graph overcomplicated or unpleasant to read

- A list of specific “guilty pleasure” artists to exclude from the graph

Upon the user clicking “Generate Graph,” the values of all these option inputs are sent in a POST request to the backend API, and the rendering process begins.

Part 2: Ripplify API

The backend API of Ripplify is built in Python, using the Flask webserver platform. The API runs on AWS Lambda, a conveniently serverless platform, meaning that it does not keep a consistently running server instance. Instead, Lambda functions activate upon an HTTP query being made, process the query, and return to dormancy. Lambda only charges for the time during which a function is running, which is ideal for a highly spiky/sporadic traffic pattern such as the one Ripplify will likely experience.

When the user logs in on the front end, the Ripplify API accepts a POST request to the endpoint /authorize holding the Spotify API token. It uses that token to make a request to the Spotify API and obtain an email, name, and profile picture for the user in question, which is then stored in a PlanetScale SQL-based database.

There is also an endpoint at /history_files which accepts POST requests containing the .zip files that Spotify provides with all of a user’s listening history. The file is accepted to the endpoint and forwarded for long term storage in AWS S3. At the same time, the endpoint kicks off a task that processes the zip file and extracts summary data, iterating through each listening record and tallying up total play counts for each month. This data is stored as a JSON file in S3 as well, and is referenced during the rendering process to significantly raise the computational

efficiency of that algorithm. The /history_files endpoint also accepts GET requests and provides a list of all the Spotify listening history files that have been uploaded and processed.

Finally, the backend also provides an endpoint at /render that begins the rendering process. This endpoint does not involve much logic within the API server. A record is created in the PlanetScale database that lists the history file that the render is using, the time it was requested, the ID of the user who generated it, etc. Then, a request is made to the rendering server, which handles the actual rendering process.

Part 3: Renderer

The Ripplify graph renderer program runs as a separate AWS Lambda microservice that is distinct from the main API. This service is written in JavaScript and, as previously mentioned, uses D3 to perform rendering. This process begins when the endpoint receives a request from the API, which contains a JSON payload listing the rendering options chosen by the user on the front end. The service runs a function that calls D3 to render the graph, with a variety of conditionals tweaking specific appearance elements to match the options specified in the JSON payload.

This rendering process was deceptively complex to optimize. Initially, I attempted to perform this rendering in the front-end of the website, which is typically where D3 is used. However, the large scale of the data needed and the logical and memory intensiveness of the produced SVG vector image caused this front-end rendering system to invariably hang the entire browser in which it was run. Hence, I decided to move the rendering logic to a back-end service (which I immediately realized would have to be distinct from the main API due to language differences). This proved to be a complex task, as D3 is strongly reliant on being run in a browser with a fully functional Document Object Model (DOM). Hence, running D3 within a server-side NodeJS script takes a great deal of special effort.

Firstly, I employed the node package JSDOM, which emulates a DOM in a Node runtime. Before calling D3 to render the graph, I use JSDOM to create a dummy HTML page with <html>, <head>, and <body> tags, then insert an <svg> tag for D3 to anchor on. I then invoke D3 to do the rendering. One additional challenge to be circumvented was that the d3-area-label package, which handles placement of the names of artists or songs on the graph, is highly reliant on running in a real browser. This is because during the process of calculating where the text would fit, it attempts to place the text element onto the page and use the method `getBBox()` to determine the exact dimensions that the text would take when rendered. Hence, I needed to modify the d3-area-label dependency code to manually approximate the size of rendered text based on string length without querying for real dimensions. Improving the performance of this approximation is an ongoing challenge, but my implementation generally finds a satisfactory position and scaling for labels. At the end of the rendering process, I use JSDOM to fetch the internal HTML of the SVG element and return it as text.

After the microservice obtains the textual SVG content, it reaches out to AWS S3, and creates a new file with `application/svg+xml` content type to hold the image content. It then saves the URL of this file in the PlanetScale database. When the front end sends a rendering request, it begins making repeated API requests every 5 seconds to check whether the graph's rendering has completed. When it has, the URL outputted by the renderer is included in the returned payload, and the image will subsequently be displayed in the front end.

Conclusion

Ripplify takes advantage of beautiful data visualization algorithms paired with deeply fascinating archival data to create a delightful experience for users. Several of those who have

beta tested the app have described spending long periods of time looking closely at their results and constantly noticing moments in their music listening history that they associate with particular life phases, friendships, romantic relationships, and more. Although the platform's use is limited to those individuals who are motivated enough to wait 30 days for their data to arrive, there seem to be enough of such people to motivate ongoing interest. Upon publicly announcing the project through social media, 25 users very quickly registered, at which point I realized that Spotify requires third party users of their API to apply for official approval, and until such a time as approval is granted, the app caps registrations at that 25 user mark. I am awaiting their approval and in the meantime encouraging users to begin the process of requesting their data. After gaining traction online, I hope to begin selling printed copies of users' listening history, to be used as wall decor, or other merchandise. I also plan to work on a wide variety of improvements to the technical side of the project, such as supporting other graph formats besides stream graphs, allowing color-coding of artists and songs by genre, and allowing more manual customization of color schemes.

References

Byron, Lee. *GitHub: Leebyron/Streamgraph*. 2010, https://github.com/leebyron/streamgraph.

Byron, Lee & Wattenberg, Martin. *Stacked Graphs – Geometry & Aesthetics*. 2008, https://leebyron.com/streamgraph/stackedgraphs_byron_wattenberg.pdf.