Sean McShane

Senior Project Description

Spotify Artist Finder

https://gitlab.com/seanmcshane/spotify-artist-finder

For my senior project, I have decided to create a web platform that will identify new and emerging artists on the popular music-streaming service, Spotify. The platform will be marketed towards record labels as a way to easily target new artists to sign. In order to achieve this result, the project will consist of three main components: obtaining, updating, and storing artists' information, analyzing the obtained data, and presenting the findings through the platform.

Spotify provides a free, public web API to retrieve and control data in its systems. This API will be utilized to achieve the first main component of the platform, namely obtaining artist information. The API is based on RESTful principles and can be used by sending simple HTTP requests to the Spotify server. To obtain information about specific artists that host their music on Spotify, I will make use of the search capabilities of the API. This part of the API is designed to allow a user to search for artists, albums, etc. using regular expressions and filters. It is possible to retrieve a list of every artist currently on Spotify, however the list is so large (over 3 million artists) that it is impossible to reach every artist using this method. This is because the API call returns a list of only 20 artists and then one can put an offset in the query to return the next 20 artists. The maximum offset is 100,000, so reaching 3,000,000 artists is impossible. To circumvent this problem, I will add filters to the API calls. These filters will allow for a much smaller result set. Once the search API call returns, I will use the artists' IDs that are returned with other API calls to extract more artist information.

Before the artist information is obtained, the database in which it will be stored must be designed. The main table will consist of basic artist information: artist name, genre, and ID. This table will be used to extract artist IDs and query other tables. One of these other tables will include the artist's performance data like followers, popularity rating, and the artist's calculated growth potential score. This table will hold current and some historic performance data, while other tables can be used to hold older data. The calculated score will be the main metric used to determine the best artists to present on the platform.

After the initial artist data is collected, it must be analyzed and kept up-to-date to see which artists are growing the fastest. A metric will be created that is a score based on several data points. Currently, these data points include: popularity rating (metric created by Spotify), number of followers, and most recent release. This metric will be a number that gets larger as the analyzing function reports an artist's popularity growing faster and will be stored in one of the database tables.

The presentation of the data will consist of a website that presents the findings stored in the database. When a user first visits the site, they will be shown the top emerging artists. In addition, a user will be able to request a list of emerging artists in different genres. When users are shown lists of this type, they will be able to click on the artist's name in order to receive more information about the artist. These screens will also include aesthetically pleasing charts and graphs that allow the data stored to be easily visualized.

Platform Choice

Before proceeding any further, it is important to discuss the platform used to host this project. Initially, the most obvious choice was to locally host all the services created. This would consist of hosting both the database and web server on a local machine. Some positives of this approach include having full control over how every service is set up and the ease of starting out, as it is my computer and I know how it operates. However, the database must always be up and running for the daily processes to operate correctly and the web server must be running at all times for the website to be available. This meant that I would be responsible for turning my desktop into an always-on server, which was unrealistic. Also, if this platform were to someday be turned over to somebody else, it would be difficult and time-consuming to recreate my system on somebody else's.

After some discussion and research, it seemed the most elegant solution would be to take advantage of some readily available cloud-based services. By hosting the platform remotely, it would simplify the process of ensuring the always-on functionality of the server as the remote machine would be a part of the hosting provider's infrastructure. In addition, in the event that this platform is handed off to somebody else, I can easily grant access to that person to my project on the cloud platform.

The two most popular cloud-based providers include Amazon Web Services and Google Cloud Platform. On the surface, both providers appear to offer the same services. AWS offered several bonuses during a free trial period that made their services look attractive, but prices for their services would quickly increase after the trial was over. On the other hand, Google offered a free-tier platform and their prices past that tier were not exceedingly high. Also, after some digging, it showed that Google's platform provided more freedom when it came to how I chose to host my services, which is why I chose to use Google Cloud Platform.

Now, Google Cloud Platform is essentially a high-level container for the different cloud-services they provide. As I began to investigate the platform, I noticed there were two separate routes I could take, namely, Google's App Engine and Compute Engine. App Engine abstracted many of the elements of hosting a service on the cloud and made it easy for the inexperienced user. One just had to provide their code and Google would take care of the services used to host it and would automatically scale the project's resources when needed. Of course, this came with a cost and was more expensive than the counterpart Compute Engine. Compute Engine allowed a user access to their own private server on Google's infrastructure. This meant that the owner of the server had full command-line access to the system and could choose to do whatever they pleased with it. For me, this freedom was attractive and the price was lower, which is why I ended up choosing Google's Compute Engine on Google Cloud Platform to host my project.

Database Design

When it came to designing the database that would be used for storing all of the artist's information, I knew that I would need to use a relational database management system in order to receive the best performance as I would use each artist's Spotify ID as a primary key and use that key to access other rows in other tables. I began by creating an abstract visual representation of the design by using a relational schema, which can be seen in Figure 1.

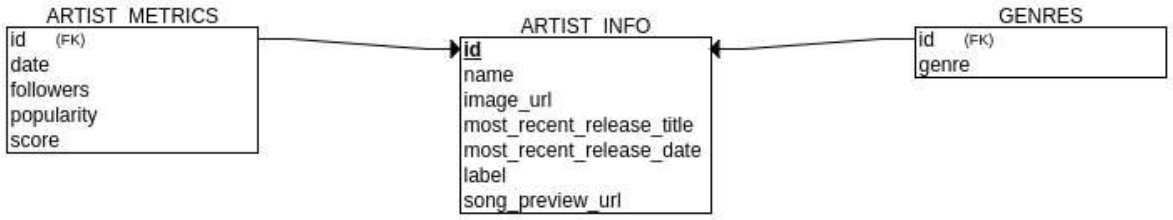

Figure 1. Database Design.

The main table in the database is the ARTIST_INFO table and holds basic and detailed information about each artist. The id column is used as the primary key for the table and as shown, is used as foreign keys in each of the other tables. Some of these data points will be used in the ranking of the artist, while some, like the image_url column, are used simply for presentation on the website. The first design of the database included a genre column in the ARTIST_INFO table. However, after more research regarding the Spotify API, each artist can be a part of several genres. So, it made more sense to normalize the database and create a new table, namely GENRES, that would hold all of the different genres each artist was a part of. Finally, the last table shown in the figure is ARTIST_METRICS. This table serves as the place where each daily metric collection will store its findings. The data in this table will be used to rank each artist. After this design was created, I implemented it using a popular RDMS, MySQL.

Obtaining Artist Information

Now that the database is in place, the artist's information can start to be stored. The first step in starting this process was selecting an appropriate programming language that could efficiently handle sending out thousands of HTTP requests, parsing the JSON that the Spotify API returns, and inserting the information in the MySQL database. All of these goals can be achieved in many different languages, but it seemed most appropriate to choose Node.js. Most importantly, Node.js operates on asynchronous principles. This is important as it allows the program to send out thousands of requests without waiting for each previous one to respond before moving on to the next. Secondly, JSON is native to JavaScript, which Node is based on, so the program can handle the JSON response from Spotify as a normal JavaScript object. Finally, the MySQL Node.js library provides full support for interacting with the database.

The process of obtaining artist information is partitioned into two separate processes, each of which run at different times. The first of these two is retrieving and storing a list of all artists on Spotify who have released music recently. The second process retrieves selected metrics for each artist in the database which will be used to rank each artist's growth potential.

In order to generate a large list of artists for the platform to rank, the first process runs once a week, currently Monday at 2 am, and gathers basic information about all artists who have released music on Spotify within the past two weeks. This was made simple as the Spotify API has the potential to perform this exact query by using the "tag:new" filter at the end of a search request. The program written to do this, gather_artist_info.js, is broken up into three main parts: gathering a list of artist ids who released music within past two weeks, using these ids to retrieve artist meta data (name, image, etc.), and using the ids to retrieve artists' release information (most recent release date, song preview url, etc.). The division of work here is necessary as each of the three processes require different calls to the Spotify API. Each process begins by making

a call to the API, shown in Figure 3, handling any errors, and then scraping the JSON returned by the API, shown in Figure 4. After the JSON is returned, the artist's name must be modified to not include any characters that are not supported by the MySQL character set, which is why the replace function call is needed.

```
460 function searchForArtists(conn){
461     var options = {
462         url: "https://api.spotify.com/v1/search?q=tag:new&type=album&market=US&limit=50",
463         method: "get",
464         headers: {
465             "Authorization": "Bearer " + accessToken
466         },
467         json:true
468     }
469
470     request(options, function(err, resp, body){
471         analyzeSearchResults(conn, err, resp, body);
472     });
473 }
```

Figure 3. Making a call to the Spotify API and passing the results to analyzing function.

```
147 function scrapeSearchResults(conn, albumObj){
148     for(artist in albumObj.artists){
149         if(albumObj.artists[artist].name.replace(/[^\x00-\x7F]/g, "") != "")
150             ids.push([albumObj.artists[artist].id, albumObj.id]);
151     }
152 }
```

Figure 4. Scraping JSON result from API.

Once all of the API calls return, the information found will be stored in the database and when the process is complete, an email will be sent regarding the number of artists found and how long the process took. There is a script, run_weekly_process.sh, that is scheduled to run at the designated time and writes the output seen from the program into a new log file for that day in the /logs/weekly directory. An example of one of these log files can be seen in Figure 5.

```
1 Starting data collection at: 1520128802062
2 Connecting to database...
3 Connected
4 Spotify access token retrieved
5 Found 67656 artists with new music in the last 2 weeks.
6 Starting to look for 55248 unique artist details, but sleeping first
7 Sleeping for 120
8 1520129478084: 55248 artists left
9 1520129479586: 55198 artists left
10 1520129481089: 55148 artists left
```

Figure 5. Weekly Process Log File.

When querying the Spotify API, it is important to not overload their systems as they will make a client wait for a specified amount of time before allowing them to continue sending queries. This is why in several points in the program, and can be seen in Figure 5, the program sleeps for several seconds to ensure that it is not sending out too many requests too quickly. After this first process has run at least once, it is possible for the second process of collecting artists' metrics to start running. This is handled similarly to the first process, invoked by a script that dumps output to a log file, but is designated to run once a day, currently at 2 am. The program, daily_metrics_collection.js, is similar in structure to the first process but is much simpler as this process only relies on one API function, rather than the three previously. In order to know which artists to search for, the program begins by querying the ARTIST_INFO table and retrieves every artist id that is stored. It then uses this list along with the API to retrieve the current number of followers and popularity of each artist. This information is then stored in the ARTIST_METRICS table. An example of this process can be seen in Figure 6, where the function loops through each artist stored in the database, queries the API, and then calls a function to analyze the results.

```
function searchForArtistMetrics(conn){
    var lastFlag = false;
    var artistIdList = "";

    console.log(new Date().getTime() + ": " + ids.length + " artists left");

    for(var i = 0; i < 50; i++){
        if(ids.length == 0){
            lastFlag = true;
            clearInterval(searchingForMetrics);
            break;
        }
        artistIdList += ids.pop().id + ",";
    }
    var options = {
        url: "https://api.spotify.com/v1/artists?ids=" + artistIdList.substring(0, artistIdList.length-1),
        method: "get",
        headers: {
            "Authorization": "Bearer " + accessToken
        },
        json:true
    }

    request(options, function(err, resp, body){
        analyzeArtistSearchResults(conn, err, resp, body, lastFlag);
    });

    artistIdList = "";
    if(startTime + ((expiresIn - 500) * 1000) < new Date().getTime()){
        startTime = new Date().getTime();
        console.log("Getting new access token");
        getAccessToken();
    }
}
```

Figure 6. daily_metrics_collection.js Code Example.

Analyzing Stored Artist Information

Next in the process of creating this platform is actually looking at the artist data being collected and determining which artists are performing the best. To do this, a mathematical function was created that scored artists based on their increase in followers and popularity, which can be seen in Figure 7. The variables *F* and *P* represent followers and popularity, respectively, while the subscripts represent the day number with 1 as the current day.

$$\left(\frac{\frac{F_1-F_2}{F_2}+\frac{F_1-F_7}{F_7}}{2}\right)\times 100+\left(\frac{(P_1-P_2)+(P_1-P_7)}{2}\right)$$

Figure 7. Function used to score artists.

The function starts by calculating the percentage increase of followers over the past two and seven days. This numbers are then added, averaged, and then normalized to turn the percentages into real numbers. The reason that the increase over the past two days is included is because it allows a heavier emphasis on how the artist is performing more recently, allowing the most recent emerging artists to be scored higher. This half of the score is then added to the popularity portion of the function. This part is almost identical to the first part, however instead of percentage increase, the real number increase is determined because an artist's popularity rarely increases and has a maximum value of 100. The final score is then found and artists that are scored higher are determined to be performing better.

Website Structure

To handle the creation and serving of the actual website that a client will use to view the platform's findings, Node.js was utilized once again. Node was chosen because it allowed easy creation of dynamic web-pages, could quickly read from the MySQL database, and made it simple to create a server and serve requests.

The website consists primarily of two pages that are dynamically filled with different content depending on the request sent to the server. First of these two pages is the main rankings page. This page displays, in a table format, a list of the artists ranked from highest score to lowest. Each page displays up to 100 results at a time, and results can be filtered based on user input. When a request is received for this type of page, an SQL query is constructed based on the filters given by the user. The results are then plugged into the rankings page HTML skeleton and then the modified HTML is sent back to the client.

Figure 8. Rankings Page.

Stemming from the rankings page is the second primary type of page namely the artist detail page. When browsing the rankings page, a user can click on an artist's name in the table and they will be redirected to a new page that presents information specific to that artist. This information includes: a picture of the artist, a data table of metrics, an interactive widget that allows a user to listen to the artist's top songs, and three visual graphs that show the artist's performance over time.

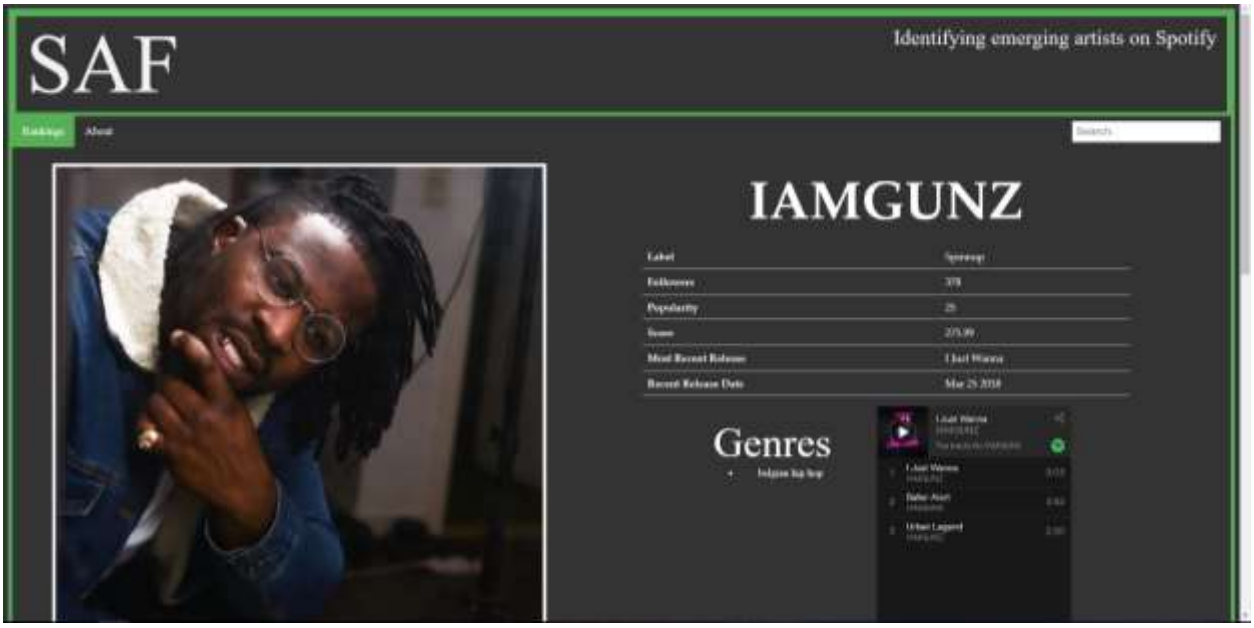

Figure 9. Artist Detail Page.

There exists one more type of page that is largely similar to the rankings page namely the search page. On the website, a user can search for any artist they please. The server takes this search and uses it as a regular expression to see if any artists with that name exist in the database. This query is cleaned from any malicious SQL injection characters before being used against the database. The results are then shown to the user in a table format similar to the rankings page.

The creation of these different types of pages is handled by the server with some help from pre-constructed HTML files. An example of these pre-constructed HTML files and the process to fill these pages can be seen in figures 10 and 11, respectively. When the server receives a request, it will read in the correct HTML skeleton page, modify it, and return the resulting HTML to the client. For example, say a client connects to the website. The server will receive this request and will read in “index.html”, as it is the default page shown to a new client. Before being modified, this page consists of the titlebar and an empty table. After the file is read

in, it is converted to a DOM structure using JSDOM. A query is then sent to the MySQL database and JQuery is used to insert HTML code for each row in the query result. The page is then returned and sent. A very similar process is followed for each respective page. For styling, external CSS files are used.

```html
        <div class="bodyContainer">
            <table id="searchTable">
                <tr>
                </tr>
            </table>
            <div class="buttonContainer">
                <form id="prevPage" method="get">
                    <input type="text" placeholder="Search.." name="search" class="search" style="display:none;">
                    <input type="text" name="start" class="defaultVals" id="lowVal">
                    <button class="pagePicker" type="submit">Prev</button>
                </form>
                <form id="nextPage" method="get">
                    <input type="text" placeholder="Search.." name="search" class="search" style="display:none;">
                    <input type="text" name="start" class="defaultVals" id="highVal">
                    <button class="pagePicker" type="submit">Next</button>
                </form>
            </div>
            <div id="pageInfo"></div>
            <script src="../js/main.js"></script>
        </div>
```

Figure 10. Search page HTML skeleton.

```javascript
function fillSearchTable(html, search, startIdx, resultSize, callback){
    conn.query([illegible] + escape(search) + [illegible] + resultSize + [illegible] + (startIdx-1) + [illegible], function(err, result, fields) {
        if(err)
            console.log(err);
        else{
            var tableString =   [illegible] +
                                    [illegible] +
                                    [illegible] +
                                [illegible];

            for(var i = 0; i < result.length; i++){
                tableString +=  [illegible] +
                                    [illegible] + (i + (+startIdx)) + [illegible] +
                                    [illegible] + result[i].name + [illegible] + result[i].name + [illegible] +
                                [illegible];
            }

            var dom = new JSDOM(html);
            var $ = require([illegible])(dom.window);
            $([illegible]).each(function () {
                $(this).html(tableString);
            });
```

Figure 11. Filling HTML table with query results.

Future Improvements and Conclusion

Overall, I was very happy with the ending version of the platform and the features that it possessed. However, there are two main improvements that could be implemented if more time was permitted. Firstly, scoring artists based on of a static mathematical function is good, but machine learning techniques could be implemented to more accurately predict emerging artists.

This technique could look at old artists who successfully emerged onto the music scene and train itself in order to identify which new artists are most likely to be successful. In addition to this, there is no doubt that the front-end of the website could be improved. Graphic design techniques could be used in order to improve both the user interface and the user experience.

Spotify Artist Finder is a platform that fits the needs of record labels looking to sign new and emerging artists. By analyzing metrics over time of artists who have recently been releasing music on Spotify, it is possible to find which artists are growing in popularity the fastest. By presenting these emerging artists on a web platform, a user can easily see how the artist has been performing and can listen to the artist's music. This should tone down the difficulty that record labels have of finding new artists to sign and increase their confidence that the artist will do well under the label. The platform allows a user to have a single place to find emerging artists on the most popular music-streaming service and has the potential to save hours of time searching for artists when compared to other solutions.