F20DV Lab 4 Report

DEMONSTRATED TO AMIT PAREKH: 01-04-2022

CAMERON DOUGLAS – H00295794

Course code and name:	F20DV Data Visualization and Analytics
Type of assessment:	**Individual**
Coursework Title:	Lab 4 – Custom Visualization
Student Name:	Cameron Douglas
Student ID Number:	H00295794

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):* *Cameron Douglas*

Date: *01/04/2022*

Copy this page and insert it into your coursework file in front of your title page.
For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

Choice of Data Sources

For this coursework, I chose a dataset from Kaggle. This dataset contains the top 100 songs on Spotify from each year dating from 1921 to 2020.

The dataset can be found here: https://www.kaggle.com/datasets/ektanegi/spotifydata-19212020

The data headings are as follows:

acousticness, artists, danceability, duration_ms, energy, explicit, id, instrumentalness, key, liveness, loudness, mode, name, popularity, release_date, speechiness, tempo, valence, year

For the secondary data source, I decided to enrich this dataset by using the Spotify WebAPI, more specifically, their `https://api.spotify.com/v1/tracks/{id}` endpoint. This endpoint accepts a track id (which I obtain from my dataset), and returns a JSON of related data which I am able to put to use in this visualization.

Code

In this project, I have reused code from the previous lab exercises, enhancing upon it where necessary. Any code which was modified from an external source is credited in comments at the point of using.

Visualization

Using the data described previously, I created the following dashboard:

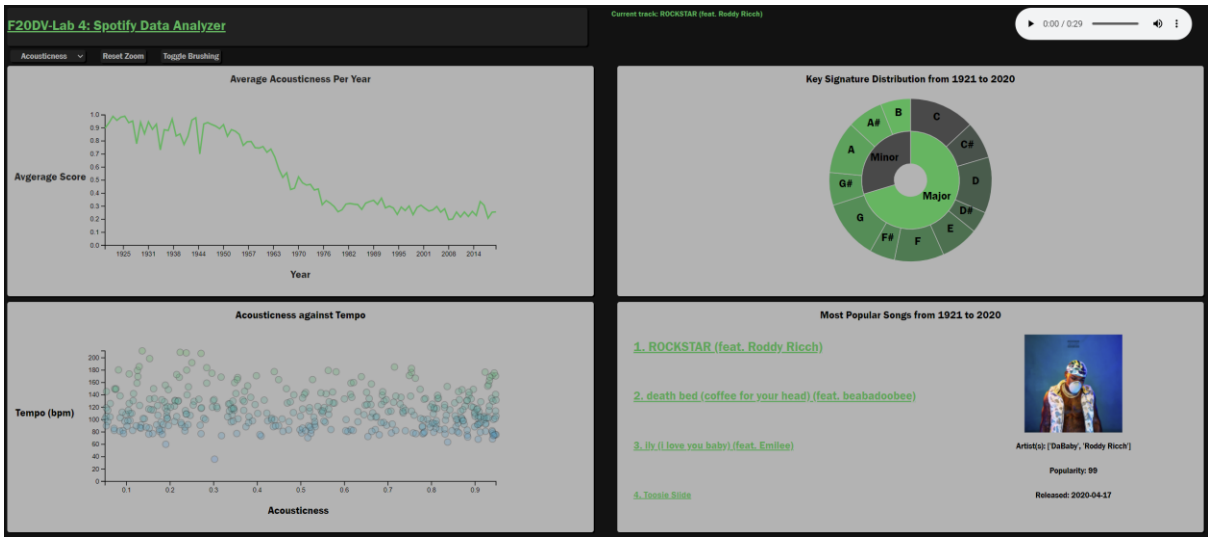

Figure 1 - Lab 4 Dashboard

The dashboard is split into four quadrants, each telling a story about the data. These quadrants will now be discussed in more detail. Each quadrant is appended to a which in turn contains its own responsive SVG element.

```
// https://stackoverflow.com/a/56239268
// Add a responsive SVG
const svg = d3.select("body")
    .append("div")
    .attr("class","chart-container")
    .append("svg")
    .attr("preserveAspectRatio", "xMinYMin meet")
    .attr("viewBox", "0 0 " + xSize + " " + ySize)
    .append("g")
    .attr("transform","translate(" + margin + "," + 75 + ")");
```

By using the *viewbox* and *preserveAspectRatio* attributes, the SVG and its contents will be scaled appropriately with window width to ensure the visualization is viewable on devices of all size and resolution.

Line Chart

The first element of the dashboard is the line chart:

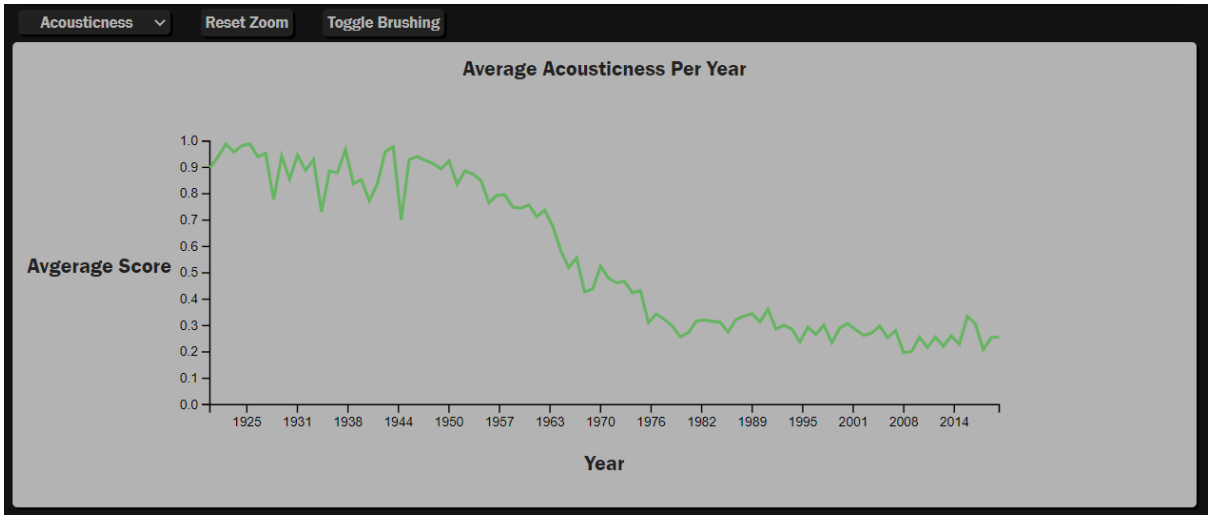

Figure 2 - Line Chart

The line chart plots the average scored variables from the dataset (acousticness, danceability, energy, instrumentalness, liveness and speechiness) against the year of the song's release. This serves to show the user how popular music has evolved over time. In the top left there is a dropdown which can be used to select which data series to view. There is also an option to enable brushing which lets the user zoom in on the line chart, and also filter data to some of the other visualizations.

To build this chart, I first create a map of all data which maps years to an array of songs:

year => [[song],[song],…]

```
for(let i = 0; i<data.length;i++){
        if(data[i].release_date.split("-")[0] == year){
            yearSongs.push(data[i]);
        } else{
            dataMap.set(year,yearSongs);
            yearSongs = [];
            year = data[i].release_date.split("-")[0];
            yearSongs.push(data[i]);
        }
        SongMap.set(data[i].id,data[i]);
    }
```

This gets each individual song (in this instance will be *data[i]*) and appends it to *yearSong*, which, once full, appends that to the map with the year as the key, and the array of songs as the value.

Then, for each {key,value} in the dataMap, I call the function build*InsertField*Data which then update the global arrays of values for each field

```
/**
 * Populates the Acousticness array with pairs {year, average acousticness}
 * @param {*} array of songs from a specific year
 * @param {*} string given year for the data array
 */
function buildAcousticData(data,year){

    let acouSum = 0;
    for(let i = 0; i<data.length; i++){
        acouSum += parseFloat(data[i].acousticness);
    }
    let acouAvg = acouSum/data.length;
    let date = new Date(`${year}-01-01`)
    acousticness.push({x:date,y:acouAvg});
    acousticness.sort(function(a,b){
        return a.x - b.x;
    })
```

This averages the value field (in this instance acousticness), and appends it to the global array. This array is then used to build the scatter plot. On initialisation, the axes are drawn and the line appended, on update, the axes scales are modified and a new line merged using the *merge()* function.

```
  // ----- MERGE PATHS TO TRANSITION TO NEW GRAPH -----
    var l = svg.selectAll(".line")
        .data([data],datum=>datum.x)

    l.enter()
        .append("path")
        .attr("class","line")
        .merge(l)
        .transition()
        .duration(transitionSpeed)
        .attr("d", d3.line()
        .x(function(datum) { return x(datum.x) })
        .y(function(datum) { return y(datum.y) })
        )
        .attr("stroke", "#1db954")
        .attr("fill", "none")
        .attr("stroke-width", 2.25);
```

Pie Chart

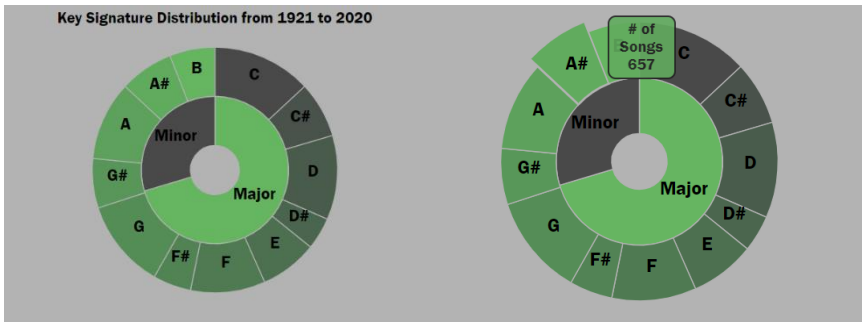

Figure 3 - Left: Pie chart with header, Right: Pie chart with on hover tooltip

The outer pie chart shows the distribution of each key signature during the date range selected from the graph (default is 1921-2020 as shown), the inner shows the distribution of 'key mode' over the same age range. On hover, the segments expand and show a tooltip giving the exact figures for each key and mode. The following function populates the two maps which are used to create the data. Taking in an array of songs from a given year, if the key or mode has not yet appeared in the respective map, it sets that key/mode's value to one, else sets it to the current value + 1.

```
/**
 * Populates the keyMap with counts of each key signature
 * @param {*} array of songs from a specific year
 */
function buildKeyData(data){

    for(let i = 0; i<data.length; i++){
        let base = parseInt(data[i].key);
        if(keyMap.get(base) == null){
            keyMap.set(base, 1);
        } else{
            keyMap.set(base, keyMap.get(base)+1);
        }

        let mode = parseInt(data[i].mode);
        if(MajMinMap.get(mode) == null){
            MajMinMap.set(mode, 1);
        } else{
            MajMinMap.set(mode, MajMinMap.get(mode)+1);
        }
    }
}
```

The pie charts are then drawn using their own draw functions. I have implemented this is this manner as attempting to alter the arc variable (which is necessary to change the sizes of each chart) even after the chart has been drawn, results in both charts using the same variable, and I am unsure as to why D3 behaves in this way.

Each pie chart is drawn as per the pie drawing code provided in a previous lab exercise, and is updated using attrTween interpolation.

```
/**
 * Transition the current data in the key pie chart to the new data
 * @param {*} dataset array containing the count of key signatures
[num,num,...,num]
 * @param {*} yearList list of selected years
 */
function updateKey(dataset, yearList){

    // Tween the path to change the pie chart
    let path = pieSvg.selectAll(".keyPath")
        .data(pie(dataset))
        .transition()
        .duration(1000)
        .attrTween("d", function(datum){

            //https://bl.ocks.org/jonsadka/fa05f8d53d4e8b5f262e

            var interpolar = d3.interpolate(this._current, datum);
            this._current = interpolar(0);
            return function(time) {
                return keyArc(interpolar(time));
            };
        });
...
```

Scatter Chart

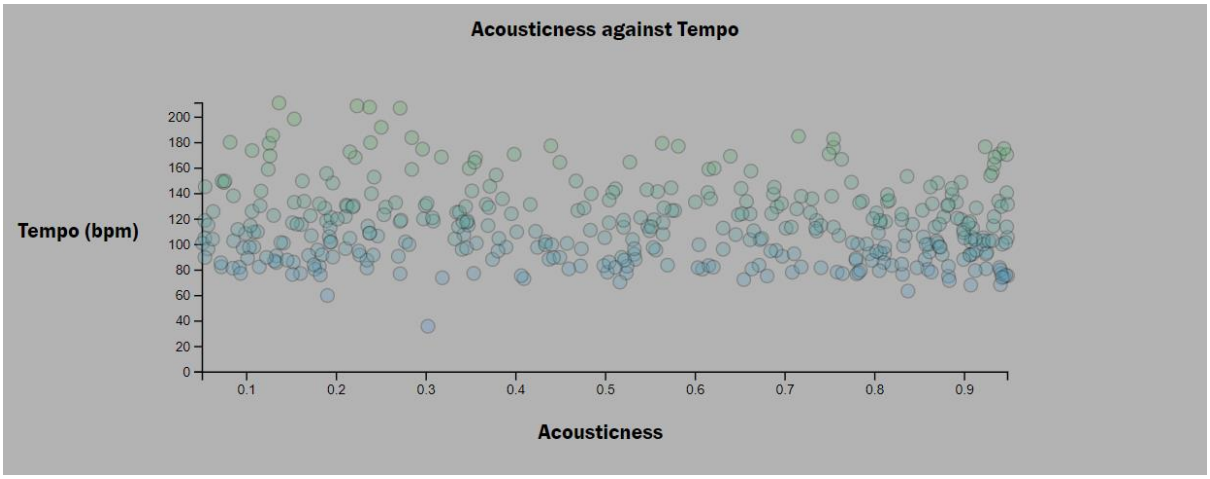

Figure 4 - Scatter Chart

The next visualisation is a scatter chart which plots a given field (determined by the dropdown described in the line chart section) against tempo, to see if any of the fields have a correlation with the speed of the music.

Similarly to the pie chart, the points show a tooltip on hover which shows the specific values and the name of the song associated with that data point.

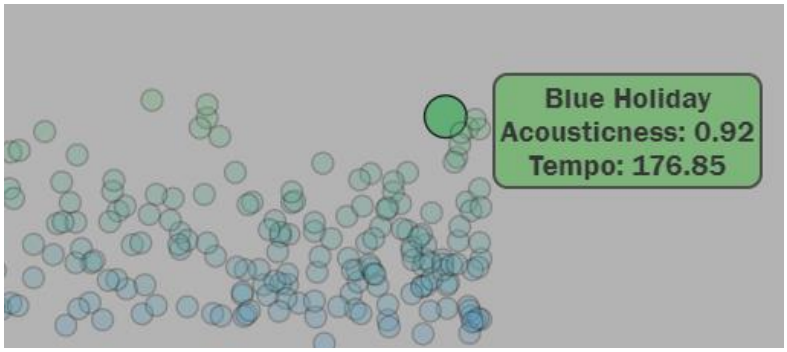

Figure 5 - Scatter chart tooltip

To build the scatter chart, created a new data map which has a key of song ID and value of the song data

```
for(let i = 0; i<data.length;i++){
        // ...Building previous map

        SongMap.set(data[i].id,data[i]);
    }
```

This allowed me to iterate over the map and append the values and the song name to an array which is then used to build the plot:

```
/**
 * Build Data for scatter plot [{float,float,string},...]
 * @param {*} data array of songs from a specific year
 * @param {*} field string describing the comparator field (what variable the
scatter plot will compare with tempo)
 */
function buildScatterData(data,field){
    for(let i=0; i<data.length; i++){
        if(parseFloat(data[i][field]) > 0.05 && parseFloat(data[i][field]) <
0.95 ){
            scatter.push({x:parseFloat(data[i][field]),y:parseFloat(data[i].
            tempo),z:data[i].name});
        }
        i = i*2;
    }
}
```

This then lets me easily build a scatter plot, using the same axis code from the line chart, and appending circles rather than a path.

Popularity List

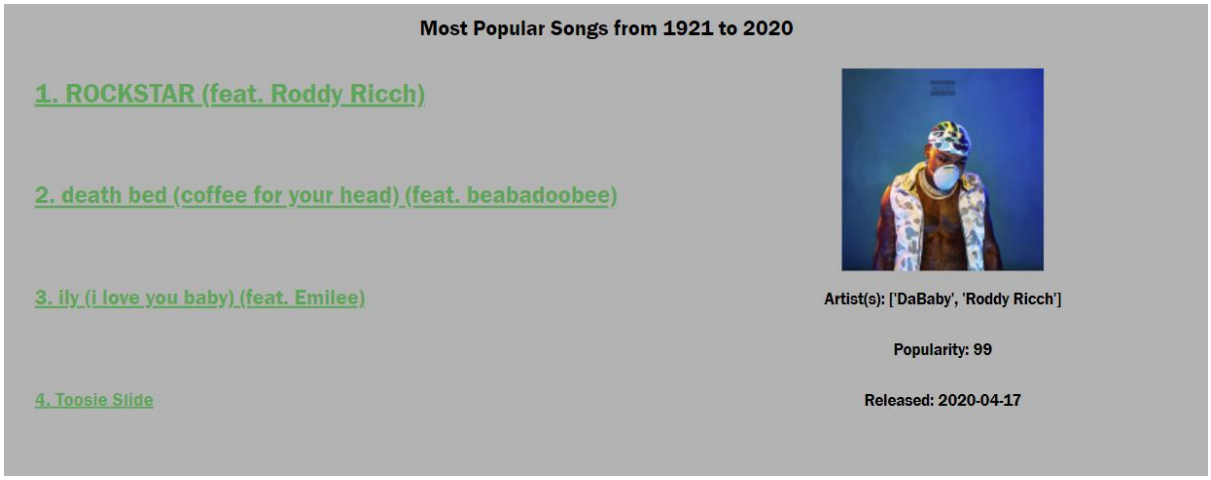

Figure 6 - Popularity List

The final visualization is a list of the most popular songs within the specified date range. Incorporating the Spotify API allows me to obtain the album cover of the track and display it using SVG image. I also use it to obtain the Spotify preview which is a 30 second clip of the song. I append this preview to the source of an HTML audio element which allows my visualiser to play the song sample. Finally from the query, I obtain the full spotify audio link which opens a new window with the spotify page on clicking the song title.

Figure 7 - Spotify Preview Player

On hovering over one of the song titles, the player and the album cover, along with the preview data are updated.

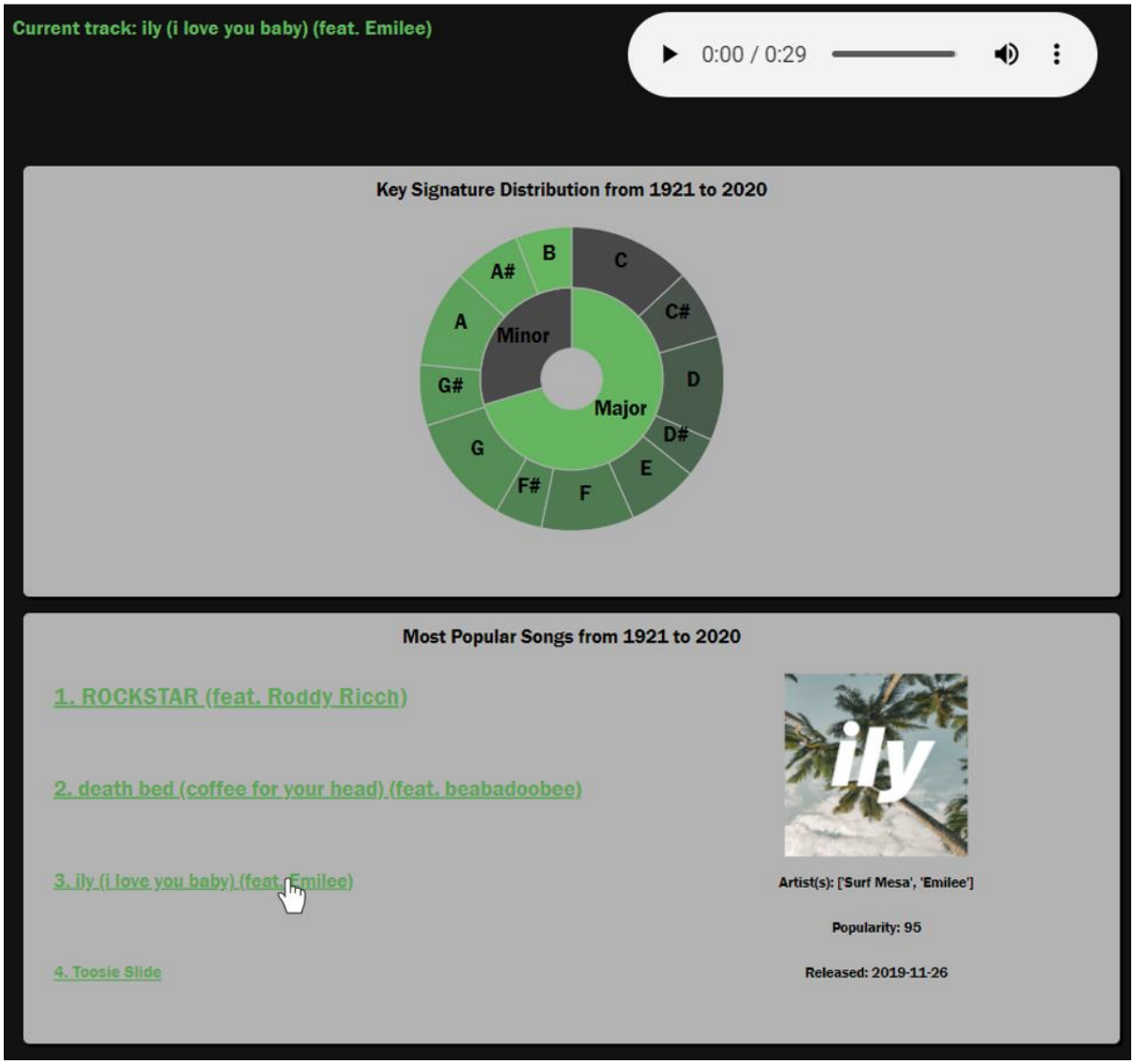

Figure 8 - Popularity list updating on hover

To achieve this, I first build a popularity dataset:

```
/**
 * Set the first->fourth most popular songs in a given year range
 * @param {*} data Individual song enrty
 * @param {*} id The ID of the given song
 */
function buildPopularityData(data, id){

    if(data.popularity>first.pop){
        first.id = id;
        first.pop = data.popularity;
    } else if(data.popularity>second.pop){
        second.id = id;
        second.pop = data.popularity;
    } else if(data.popularity>third.pop){
        third.id = id;
        third.pop = data.popularity;
    } else if(data.popularity>fourth.pop){
        fourth.id = id;
        fourth.pop = data.popularity;
    }
}
```

The variables *first, second,…* are tuples which contain {id,popularity}. Looping over the map created for the scatter plot, I check each song's popularity and set the values of *first, second…* appropriately based on the popularity of the current song. Once this is complete, I append them all to an array to give me a final array of [*first, second, third, fourth*]. I then append the text for each song in the list

```
for(let i = 0; i<data.length; i++){
        // Append each item to the list
        listSvg.append("text")
            .attr("x", -125)
            .attr("y",(75 * i))
            .attr("class","listitem")
            .style("font-size",()=>(20-2*i)+"px")
            .text((i+1) + ". " + data[i].name)
            .on("mouseover",function(){
                initPreview(data[i]);
                d3.select(this).style("cursor", "pointer");
            })
            .on("click",function(){
                getLink(data[i]);
            });
    }
```

It is here where I call the methods to query the Spotify API:

```
    // Define the endpoint
    let url = "https://api.spotify.com/v1/tracks/" + data.id;

    // Using XHR https://developer.mozilla.org/en-
US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
    // Open a new GET request
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url);

    // Append requisite headers
    xhr.setRequestHeader("Accept", "application/json");
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.setRequestHeader("Authorization", `Bearer ${token}`);

    // Check for statechange
    xhr.onreadystatechange = () => {
        if (xhr.readyState === 4) {

            // Parse JSON response
            let response = xhr.responseText;
            let jsonResponse = JSON.parse(response)
            let image = jsonResponse["album"]["images"][1]["url"];
             ...
```

I first declare the *url* which is the Spotify endpoint + the song ID which I will be querying. Next I set up an XMLHttpRequest() and open a GET request. Using the Spotify API docs, I check the headers which are required for this request and append them. This includes the authorization token, this is generated on the Spotify console using my personal Spotify login credentials.

NOTE: The key generated is TEMPORARY and will require updating before using again!

Then I parse the JSON response using JavaScript's JSON.parse method which then allows me to obtain the link for the image.

Using the JSON response, I am then able to update the album cover, and the HTML audio tag for the preview player

```
...
        listSvg
            .append("svg:image")
            .attr("xlink:href", image)
            .attr("width", 150)
            .attr("height", 150)
            .attr("x", xMax - 125)
            .attr("y", -25)
            .attr("id","image");

        // ----- remove old artefacts -----
        d3.selectAll("audio")
            .remove();

        d3.select("#audio-container")
            .selectAll("p")
            .remove();

        // https://stackoverflow.com/a/46761870

        // Add current title label
        d3.select("#audio-container")
            .append("p")
            .text("Current track: " + data.name)
            .style("color","#1db954")

        // Add HTML audio element
        d3.select("#audio-container")
            .append("audio")
            .attr("controls","controls")
                .append("source")
                .attr("src",jsonResponse["preview_url"])
                .attr("type","audio/mpeg")

...
```