Final Report

Heartbeats

Module: CM3203 One Semester Individual Project

Student Number: 1817285

Author: Alexander Smerdon

Project Supervisor: Asma Ifran

Project Moderator: Yulia Cherdantseva

Table of Contents

Abstract

Strava (Strava, 2021)[1] is an online platform for tracking a user's workouts. Last.fm (Last.fm, 2021)[2] is a platform for tracking a user's music listening habits. The aim of this project is to create an online web app, called Heartbeats, that allows users to connect their accounts on each platform to give an overview and analysis of the music they were listening to during their best workouts.

This report details the background, approach, design decisions, implementation, results and reflection of my web app.

Acknowledgements

I would like to thank my supervisor for helping me stay on track with my project, thank everyone on Stack Overflow (and other support forums) who helped me with coding problems, and thank those on YouTube who provided free tutorials.

[1] https://www.strava.com/features
[2] https://www.last.fm/about

Table of Figures

Introduction

The aim of this project was to create a web application that allows users to see a breakdown of the music they were listening to during their best workouts. To achieve this, I would need to combine two services: Strava and Last.fm. The web app allows users to connect their Strava account and enter their Last.fm username and have the results-breakdown automatically generated for them with the click of a single button.

The intended audience for this project would be those who use both services and would be interested in seeing information about their workout listening habits, but do not want to go through much effort in finding this information out. Strava is used by millions of people around the world, and Last.fm has a sizable dedicated community, so my project will hopefully have an intersection of users from both platforms. With music being such an integral part of many people's workout, I feel having the ability to correlate statistics about the two would serve a useful, real-world purpose when it comes to helping athletes improve their workouts, and be quite popular. My web app is free to use and has been designed to be easily usable for anyone.

The scope for the project I layed out in the initial plan turned out to be too broad within the timeframe I had to complete the project. With unforeseen issues plaguing the development stage, I had to give up on some of the features I had hoped to implement, and focus only on the core aspects of what users may want to see in a breakdown of the songs they were listening to during their workouts. The result however has laid a lot of groundwork for future features to be added with some more time.

My approach to the project was to follow the work plan I made in the initial project plan as closely as I could. This would involve jumping straight into coding and building up the web-app gradually as I went along. The more technical side would be focused on first whilst the design aspect would be a smaller priority until the core functionality was implemented. My specific process and changes I made to the design will be discussed in the implementation section of this report.

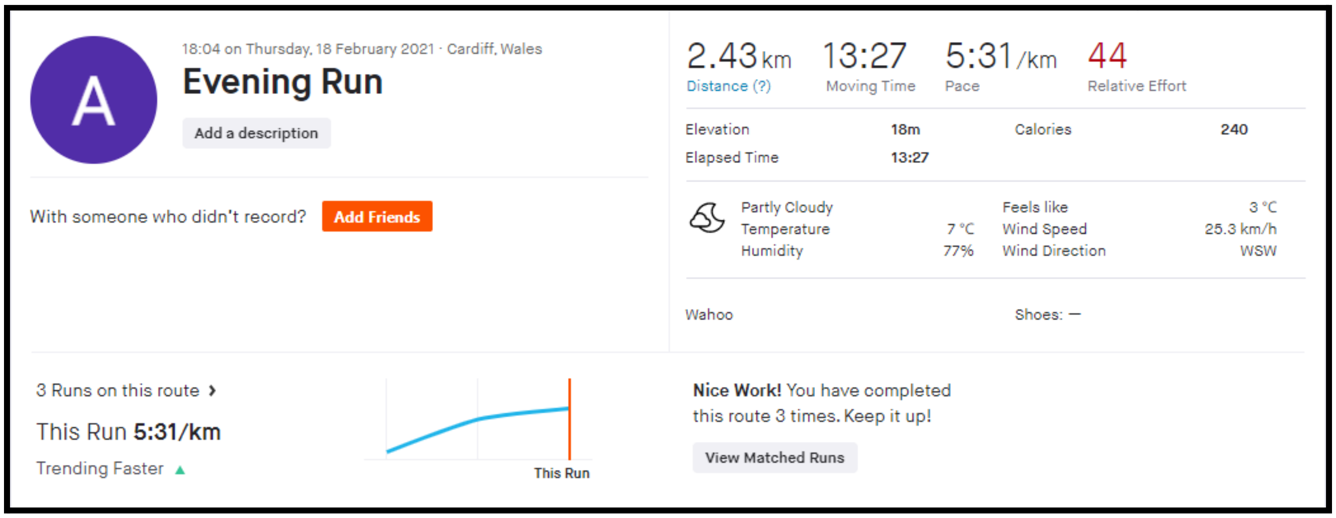
I am assuming that readers of this report have a base-level understanding of programming and programming concepts, however more technical aspects of the solution will be explained in detail. The project is written in standard JavaScript, HTML and CSS with some libraries imported for the design of the web-app.

Throughout university I have learnt various programming languages and concepts, however I have never used JavaScript, worked with APIs or learnt how to securely host a website on the internet. Learning how to do these things are personal aims of mine and this project has been an opportunity to do so.

Background

Use of Strava and Last.fm

The Strava platform allows users to see an analysis of their workouts. A typical workout analysis can be seen in *Figure 1*:


18:04 on Thursday, 18 February 2021 · Cardiff, Wales

Evening Run

Add a description

With someone who didn't record? Add Friends

2.43 km Distance (?) | 13:27 Moving Time | 5:31 /km Pace | 44 Relative Effort

Elevation	18m	Calories	240
Elapsed Time	13:27		

Partly Cloudy		Feels like	3 °C
Temperature	7 °C	Wind Speed	25.3 km/h
Humidity	77%	Wind Direction	WSW

Wahoo

Shoes: —

3 Runs on this route ›

This Run **5:31/km**

Trending Faster ▲

This Run

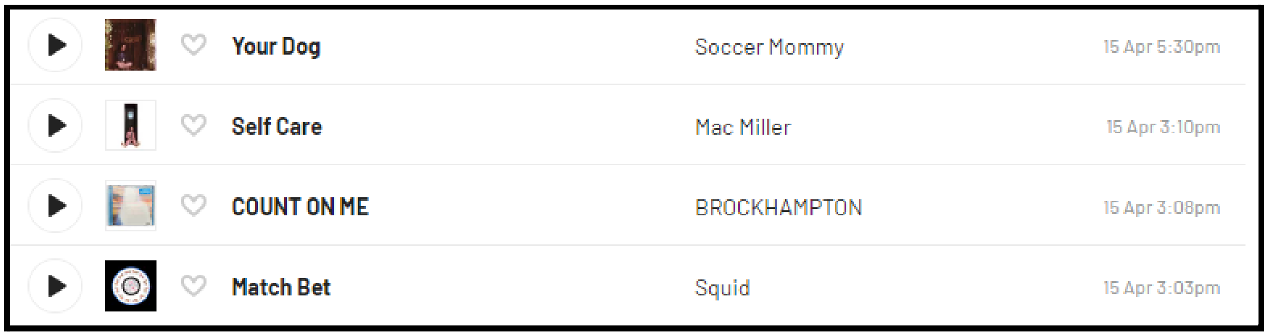
Nice Work! You have completed this route 3 times. Keep it up!

View Matched Runs

Figure 1: Strava Workout

Users can view statistics about an individual workout as well as comparisons to previous workouts that are logged on their account. Strava also functions as a social media platform, where users can follow each other and join groups to compare workouts. Statistics users can view include the speed, duration, elevation, time and distance of workouts (typically runs or cycles rides). For users who have it setup, Strava also tracks heart rate.

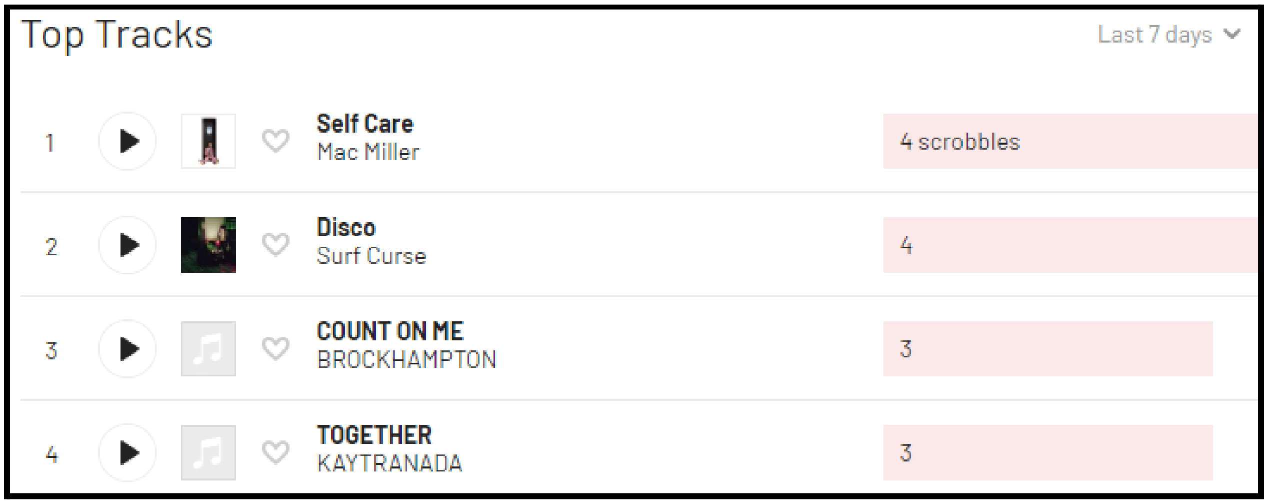
Last.fm allows for users to connect accounts they listen to music on (such as Spotify or Apple Music), and then logs every track a user listens to (as seen in *Figure 2*):



Your Dog	Soccer Mommy	15 Apr 5:30pm
Self Care	Mac Miller	15 Apr 3:10pm
COUNT ON ME	BROCKHAMPTON	15 Apr 3:08pm
Match Bet	Squid	15 Apr 3:03pm

Figure 2: Tracked Last.fm songs

Users can then see general breakdowns of the artists, tracks and albums that they have listened to over a specified period of time, as shown in *Figure 3*:



Top Tracks

Last 7 days

1	Self Care	Mac Miller	4 scrobbles
2	Disco	Surf Curse	4
3	COUNT ON ME	BROCKHAMPTON	3
4	TOGETHER	KAYTRANADA	3

Figure 3: Last.fm Top Tracks

Both of these platforms have APIs (Application Programming Interfaces) that allow for developers to request user data and build their own applications around them. This is what Heartbeats utilises and depends on for functionality.

In the initial project plan, I wrote about how I would require historical records of user data for the application to work, instead of having data collected by my application from when they sign up. I originally thought that Spotify would allow me to do this, as I know that user listening data is collected to create Spotify's yearly "Wrapped" report, however I wrongly assumed this data would be accessible via the API. Instead, historical reports of a user's listening habits can only be accessed through their privacy settings. A request for a CSV file has to be made which can take up to 3 days to get (Spotify, 2021)[3]. Even though Spotify has many more users than Last.fm, which would give my project a much wider audience, having to go through this process is difficult for a user and not what I intended for my project (plus I would need to figure out a way for users to upload large CSV files). This is why I decided to use the Last.fm API instead.

To test my web-app, I used my own account on both platforms. During my workouts I tried to listen to a range of songs and genres to get a better breakdown of results. This will be discussed further in the Results and Evaluation section of this report.

APIs

Both Strava and Last.fm have public APIs that allow for developers to access data. My project depends on these APIs to fetch user data to manipulate. To use both APIs, I was required to set

[3] https://www.spotify.com/ca-en/account/privacy/

up an application to give me access to an API key and a secret key. Should either application key become invalid, Heartbeats will stop functioning and require me to set up a new key. As Strava and Last.fm's APIs are free for users who already have an account (with the Strava platform having a monthly subscription), there is a rate limit for the amount of requests that can be made. For Strava, it is limited to 100 requests every 15 minutes and 1000 daily, and for Last.fm it is not specified but dependent on the number of users served. This should be more than enough to serve users of Heartbeats.

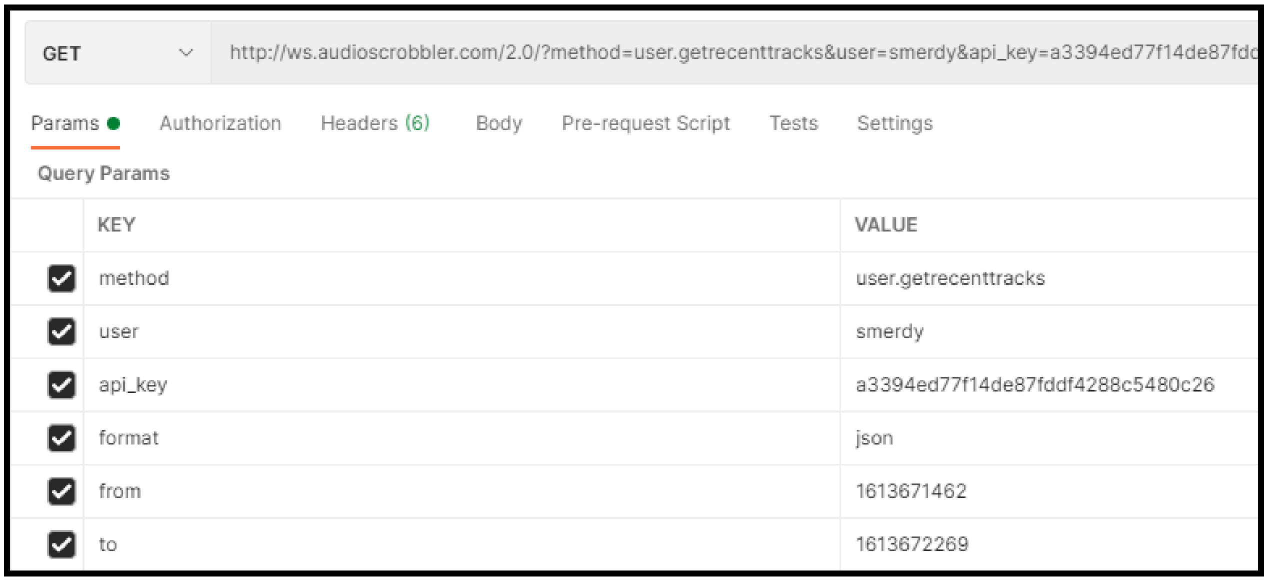
In my initial project plan, one of the key requirements listed was for Heartbeats to be a secure web-app. Initially I thought I would have to handle and store personal user data, however this turned out not to be the case. User data is securely requested through HTTPS and the data generation aspect is handled locally in the browser, meaning none has to be stored and there are no legal guidelines I need to adhere to (barring ones laid out by the API services). When it comes to API keys however, I was unable to find a solution that hides my private keys in the timeframe I had. This means that anyone can see them by opening the browser inspector and viewing sources. Although not ideal, the keys cannot be used for anything more than doing either Strava or Last.fm API requests, and are not tied to any payment methods/cards. Should a bad actor discover and use my keys, the most damage they would be able to do is call so many requests that the keys get revoked for a time-period. Some of the web-apps I discuss in the existing solutions section have their keys exposed, which leads me to believe this isn't too much of an issue. Solutions to this will be discussed in the Future Work section of this report however.

My initial plan to get information about songs (such as genre) was to use a general song fetch request from Last.fm, however this was not possible for reasons that will be discussed in the implementation. The same goes with using Spotify's API for this purpose. My solution ended up being to use TheAudioDB's API for a monthly subscription.

TheAudioDB is a community run database of music metadata (TheAudioDB 2021)[4]. This metadata includes artists, albums, tracks and song lyrics, along with related images. My use of the site's API was to request the genre of song names and artists I had received from users. Sadly, this database is nowhere near as extensive as Last.fm or Spotify's databases, meaning that I could not get the amount of data I wanted about tracks. For example, I was hoping to use the Spotify API to get the BPM of songs, which I could then use to find the average BPM for a given workout. This was not possible with TheAudioDB however, and is a limitation of my solution. This will be discussed further in my implementation and evaluation.

To create fetch requests for the APIs, I used Postman (Postman 2021)[5] to help break down the different parts of the request. *Figure 4* is what a typical API request in Postman looks like. In this example I am requesting the tracks listened between two timestamps from my Last.fm account:

[4] https://www.theaudiodb.com/
[5] https://www.postman.com/


GET http://ws.audioscrobbler.com/2.0/?method=user.getrecenttracks&user=smerdy&api_key=a3394ed77f14de87fdd

	KEY	VALUE
✓	method	user.getrecenttracks
✓	user	smerdy
✓	api_key	a3394ed77f14de87fddf4288c5480c26
✓	format	json
✓	from	1613671462
✓	to	1613672269

Figure 4: Postman Request

The format for these requests is returned in JSON.

JavaScript and JSON

JavaScript is the world's most popular programming language and the basis for millions of websites and web-apps. JSON, or Java-Script Object Notation, is a data-format that allows for the easy transfer of data between a server and a web-page (JSON, 2021)[6]. It is the format data is stored in after API fetch requests are made to servers, and the format in which my program works with data. The easy usability of JSON, as well as the object-oriented approach it allows for, is why I chose to write my program in JavaScript instead of Python. *Figure 5* is an example of a Last.fm API request for a user's track returning this data in JSON (and logged to Chrome console):

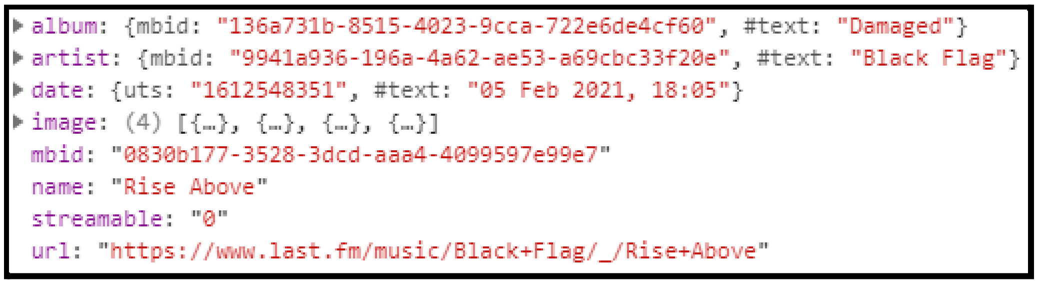

```
▸ album: {mbid: "136a731b-8515-4023-9cca-722e6de4cf60", #text: "Damaged"}
▸ artist: {mbid: "9941a936-196a-4a62-ae53-a69cbc33f20e", #text: "Black Flag"}
▸ date: {uts: "1612548351", #text: "05 Feb 2021, 18:05"}
▸ image: (4) [{…}, {…}, {…}, {…}]
  mbid: "0830b177-3528-3dcd-aaa4-4099597e99e7"
  name: "Rise Above"
  streamable: "0"
  url: "https://www.last.fm/music/Black+Flag/_/Rise+Above"
```

Figure 5: Last.fm Track Object in JSON

This data is of course abstracted from users, however it is what my program manipulates and the basis for understanding how my implementation works.

[6] https://www.json.org/json-en.html

I used GitHub to save and update my code.

Existing Solutions

The closest solution to what my project achieves is Perform (Perform 2021)[7]. Users connect their Strava account and have the music they listen to automatically sync to their workouts. It then gives users data insights into what their top tracks are based on their best performances. It also creates workouts based on what users listen to. Although a great service, the issue with Perform is that it cannot provide a historical view of information about music from workouts done before signing up to the app. This I feel is the crucial aspect of Heartbeats that makes it unique.

To give me inspiration for my project, I looked through already existing web-apps that use the Last.fm API in interesting ways. LastWave (2021)[8] is a web-app that visualises a user's listening habits in a wave format. The simplicity of the web-app helped me with the design aspect of Heartbeats, and I especially liked how the loading bar gave the user feedback that results were in the process of being generated. Scatter.fm (2021)[9] is a web-app for Last.fm that plots every track a user has ever listened to on a historical graph. This helped me think of ways to visualise user data in an interesting manner.

Heartbeats is the first web-app of my knowledge that uses Strava and Last.fm together to show users statistics about the music they were listening to during their workouts.

[7] https://perform.fm/home
[8] https://savas.ca/lastwave/#/
[9] https://scatterfm.markhansen.co.nz/

Approach

The approach I took with this project was an Agile one, where I continually researched and developed areas of the project whilst trying my best to stick to the original work plan laid out in my initial project plan. I felt this approach would help me best manage my time. If I felt an aspect of the project was taking too long to solve, and not essential to the final outcome, I would abandon it.

Setup

The first part of the approach was to set up everything I would need to make development easier. This mainly involved setting up my version management. I chose to use Git and have my repository hosted on GitHub. This allowed me to easily work on different branches as I experimented with different solutions for code, and meant my files were always backed up online and could be accessed from any machine. For my IDE, I chose Microsoft Visual Studio Code for its lightweight interface, it's easy usability, and because it is free.

Research

Starting off the project with zero experience in working with APIs, I had a lot of research to do in understanding them and getting them to work. This research was done continually throughout the development process. Both Strava and Last.fm had API documentation to help, and there were several tutorials on YouTube that explained how to use APIs.

Development

My approach to development differed from each problem that needed solving, most of my time was spent bug-fixing and trying to understand why my solution was behaving in unexpected ways. This was often due to a lack of knowledge regarding the solutions I was using to get the implementation working (for example, the way a specific API call was returned).

Specification and Design

The processing and sorting of returned API data is the main aspect of my project. The code to do this needs to be designed in such a way that any user who uses the web-app will have their personalised results displayed - no matter how big or small the API returns. Should the user have done hundreds of workouts, getting a return from Heartbeats should happen in a reasonable amount of time. This meant that my code design had to prioritise efficiency.

When it came to collating the Strava and Last.fm user data, I needed to decide how I wanted the data to be fetched and generated. My initial idea was to iterate through the Strava output JSON string, and compare the different values to get the best workout. The Last.fm data would then be pulled using this data. I was reluctant to take this approach however, as I felt it was a rather inelegant solution that required a lot of comparisons and it may not give me the data I want or the right amount of it, although it would mean only one Last.fm API request is made. Another solution would be to create an object of each workout with the songs pulled from Last.fm, and then do the comparisons from there. My main concern with this however was trying to find the best way to compare all these objects to find the best workout, and also making sure there is scalability for users who may have hundreds of workouts. This was the solution I went with however, and proved to be efficient.

Figure 6 is the class of userWorkout. The premise of my design is to only have the relevant attributes of a workout in the class so that these attributes can be compared to decide which workout is the best performing. An example of this would be if I wanted to compare every workout to see which one had the fastest speed. I would compare every object by the max_speed attribute and return the result (a diagram of this sorting can be seen in *Figure 7*).

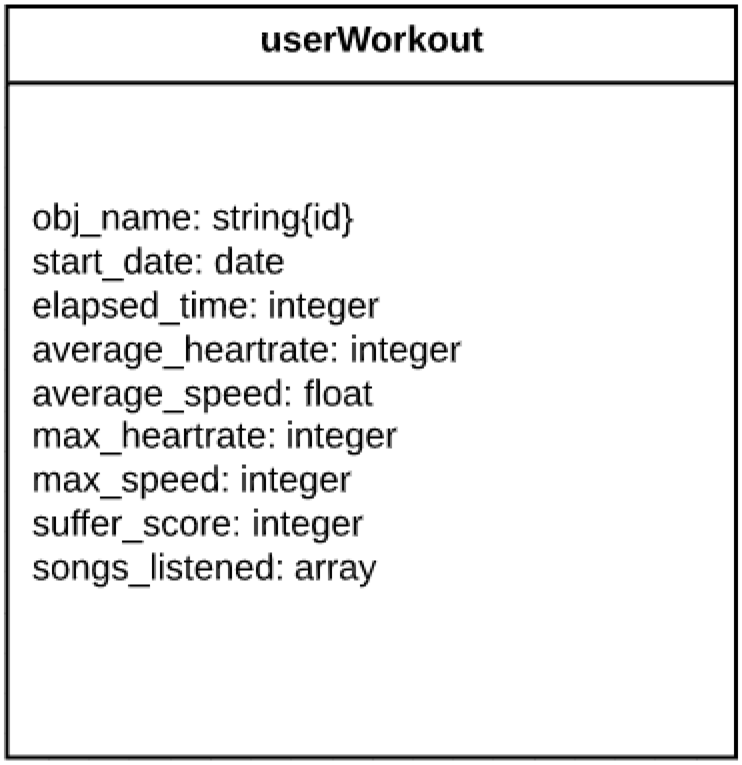


Figure 6: userWorkout Class

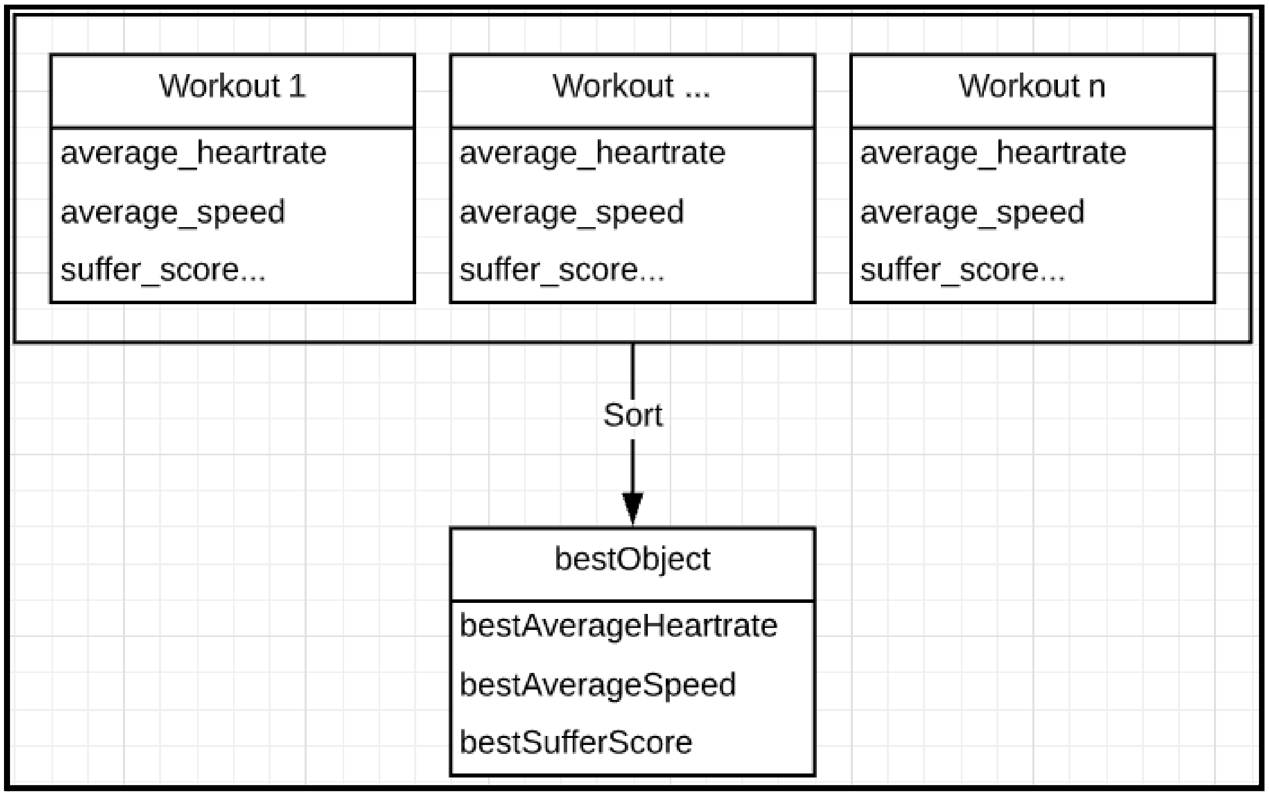


Figure 7: Sorting by Attributes

I identified the best: average heart rate, average speed, max heart rate, max speed and suffer score as ways to judge a user's top performing workouts. Within the class, there is also the object name, the start date, the elapsed time of a workout and the songs listened during the workout. I decided to include heart rate as a measurement because I wanted people who use Strava to measure stationary gym workouts to also be able to use my web-app to see a music breakdown. The full "best" class can be seen in *Figure 8*:

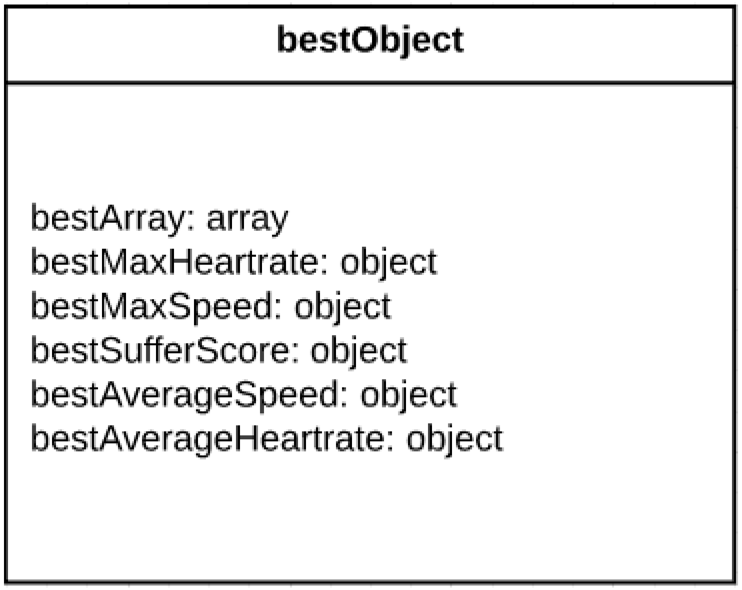


Figure 8: bestObject class

With user simplicity being a key factor, I needed to abstract as much from the user as possible. When the user goes on Heartbeats, they should have to do as little as possible to get the data. The interactions a user has can be seen in *Figure 9*:

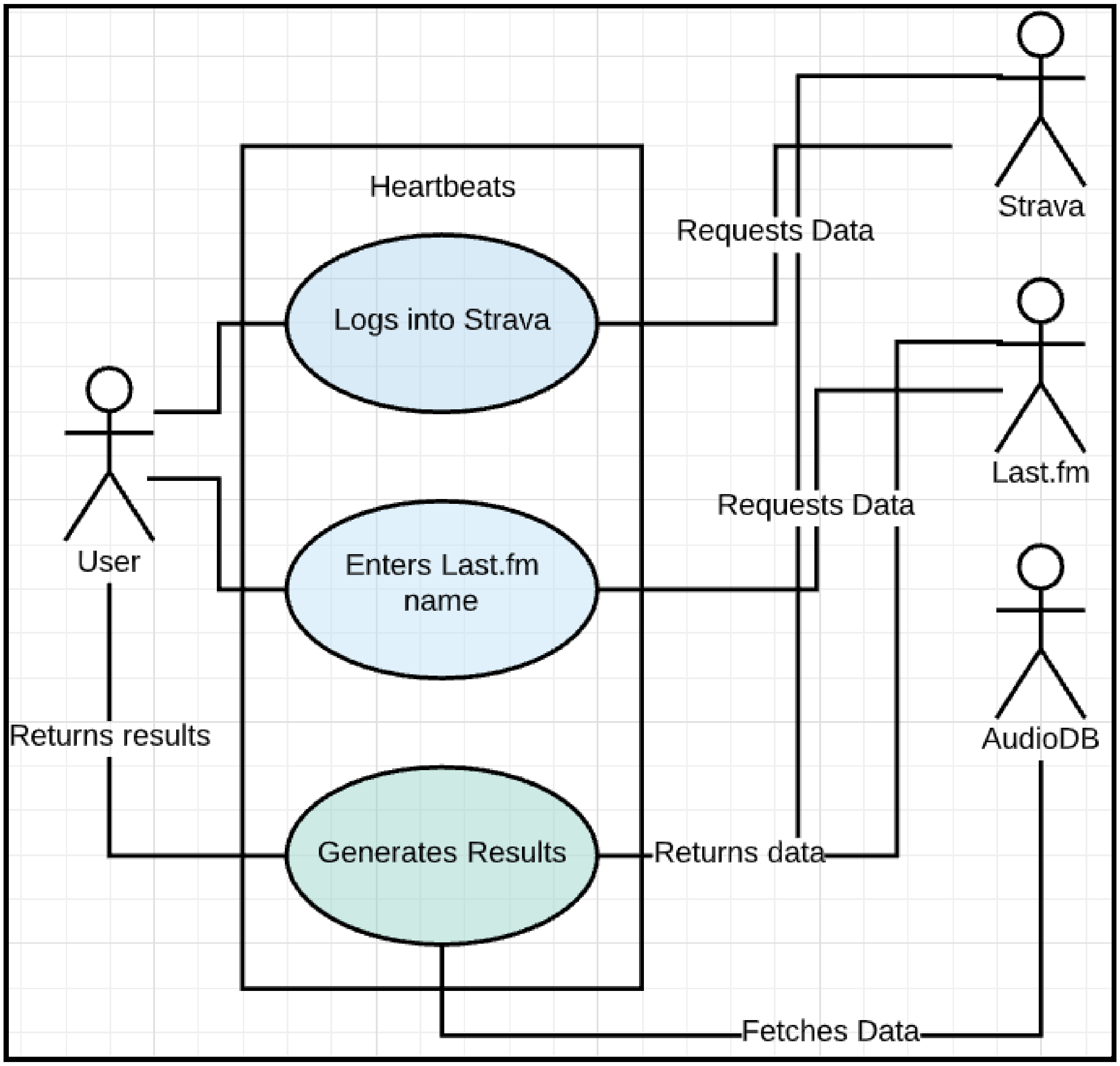


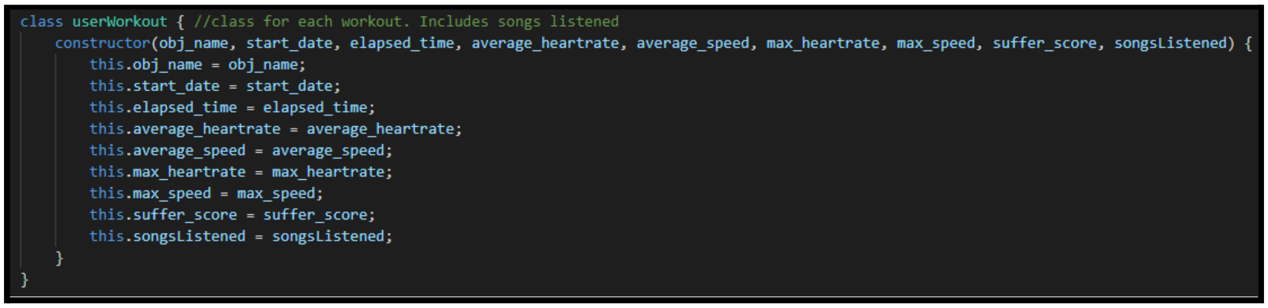
Figure 9: UML Diagram showing user interaction

Implementation

This section aims to give a clear description of how my code functions, focusing on the most important areas regarding data generation. This section follows the order the code runs but does backtrack, as different functions are called within functions. To try and keep it clear, I have broken this section down into different sub-sections. A full version of the code has been included in the submission. Code that requires it has been highlighted and has been properly referenced.

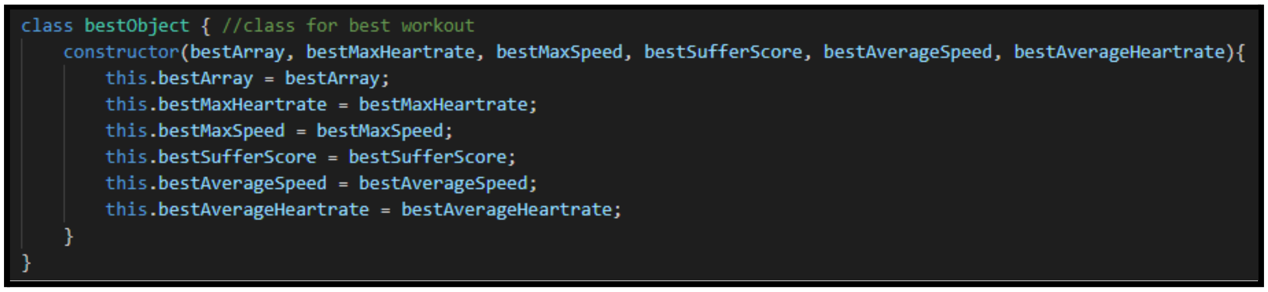
Constructors

The first part of my code are the class constructors. These are the “userWorkout” class (*Figure 10*) and the “bestObject” class (*Figure 11*) presented in the previous section:



```
class userWorkout { //class for each workout. Includes songs listened
    constructor(obj_name, start_date, elapsed_time, average_heartrate, average_speed, max_heartrate, max_speed, suffer_score, songsListened) {
        this.obj_name = obj_name;
        this.start_date = start_date;
        this.elapsed_time = elapsed_time;
        this.average_heartrate = average_heartrate;
        this.average_speed = average_speed;
        this.max_heartrate = max_heartrate;
        this.max_speed = max_speed;
        this.suffer_score = suffer_score;
        this.songsListened = songsListened;
    }
}
```

Figure 10: userWorkout constructor

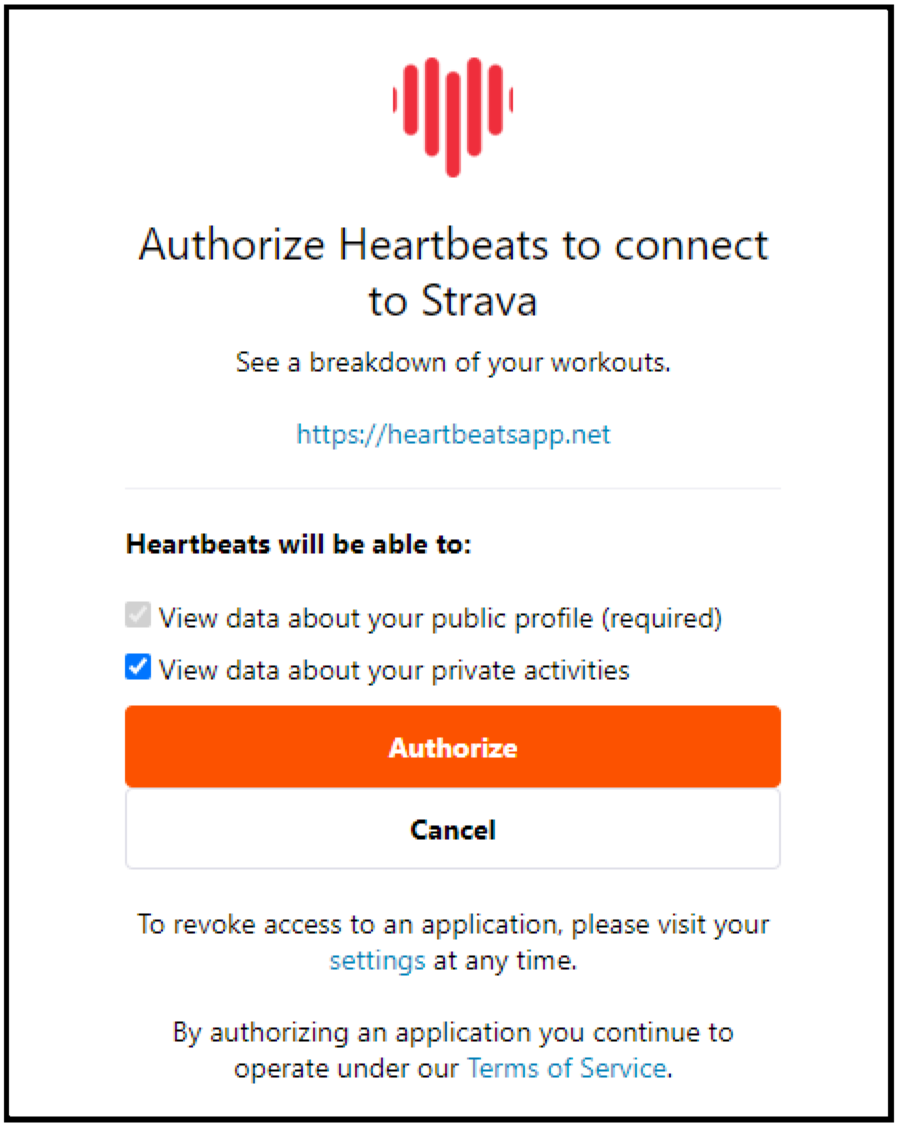


```
class bestObject { //class for best workout
    constructor(bestArray, bestMaxHeartrate, bestMaxSpeed, bestSufferScore, bestAverageSpeed, bestAverageHeartrate){
        this.bestArray = bestArray;
        this.bestMaxHeartrate = bestMaxHeartrate;
        this.bestMaxSpeed = bestMaxSpeed;
        this.bestSufferScore = bestSufferScore;
        this.bestAverageSpeed = bestAverageSpeed;
        this.bestAverageHeartrate = bestAverageHeartrate;
    }
}
```

Figure 11: bestObject constructor

Strava Login

As I am requesting user data, I need to have users authorize their account to work with Heartbeats. This is handled by an OAuth call, which redirects users to Strava’s website to login and select what data Heartbeats is allowed to view (as seen in *Figure 12*). For my app to work, I need view data about a user’s public profile and view data about private activities:



Authorize Heartbeats to connect to Strava

See a breakdown of your workouts.

https://heartbeatsapp.net

Heartbeats will be able to:

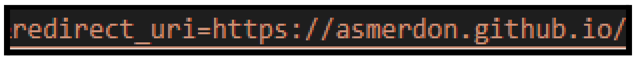
- ✓ View data about your public profile (required)
- ✓ View data about your private activities

To revoke access to an application, please visit your settings at any time.

By authorizing an application you continue to operate under our Terms of Service.

Figure 12: Strava User Authorisation

After clicking authorize the user is redirected back to the web-app (*Figure 13*):



```
redirect_uri=https://asmerdon.github.io/
```

Figure 13: redirect URI

Last.fm Username Verification

Last.fm's API works differently when it comes to pulling user data due to the fact all profiles are visible to every user (if they know the account name). This means that a user does not need to verify with Last.fm to allow Heartbeats to read their data. Instead they just need to enter their username and click the button to generate results (*Figure 14*). Having an open textbox in a web-app can be dangerous however, as users can use it to inject their own scripts and potentially do damage, so I needed to validate inputs and make sure the account names being entered actually exist on Last.fm.

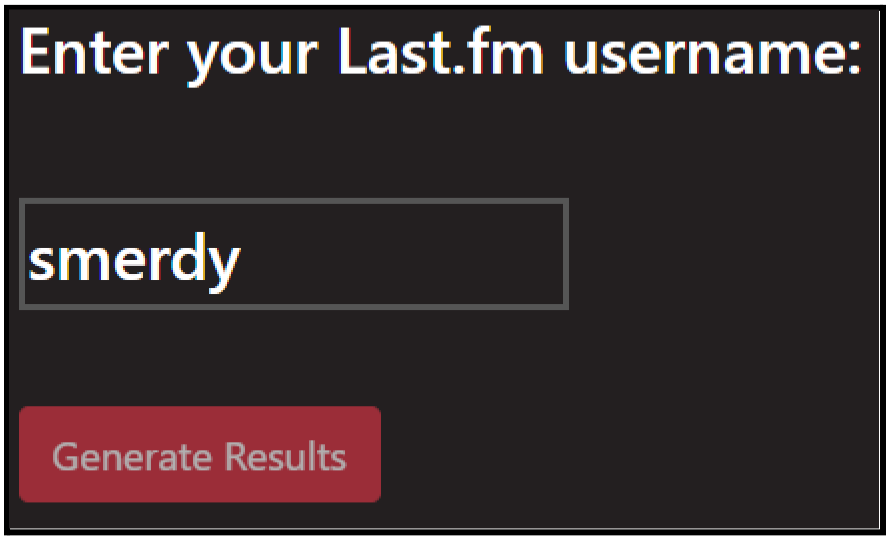
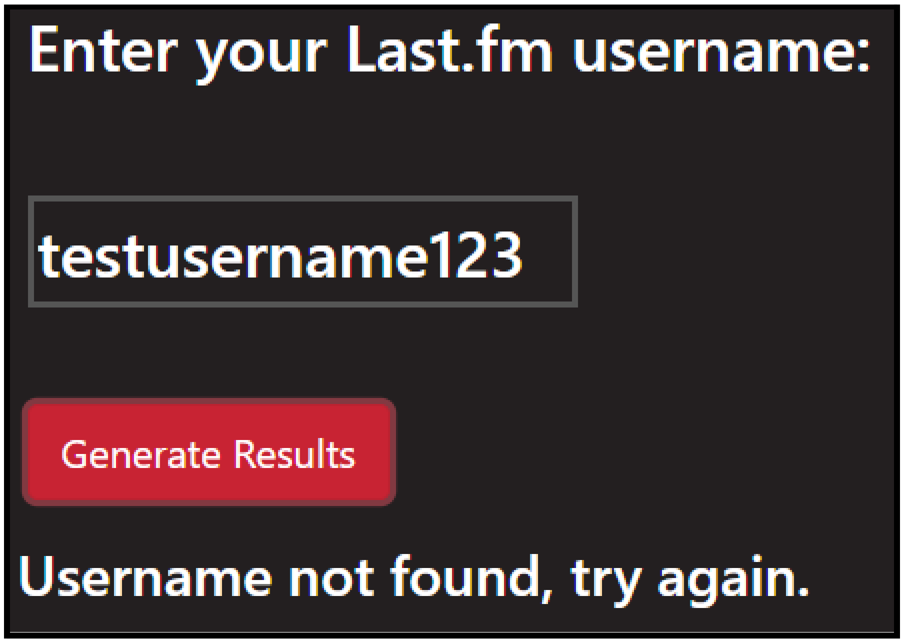


Figure 14: User Enters a Correct Username (my personal Last.fm account)

Once the “Generate Results” button is clicked, it calls a function which takes the value of whatever has been entered in the text box and calls a “getInfo” request. If the returning data has the property called “user”, it means that the username has been found and the rest of the code can execute. If it does not, a text box appears below the “Generate Results” button requesting the user to try again (see *Figure 15*). A correctly entered name will make the button unclickable again unless the user refreshes the page.



Username not found, try again.

Figure 15: User enters an incorrect username

Although this validates Last.fm usernames, it is still up to the user to enter the correct name for data generation to successfully work, as there is no way to verify that the Last.fm username's data will correlate with Strava account data. Incorrect but valid usernames will generate incomplete results. The user only has to refresh the page to try again however.

After the username has been validated, the API calls can begin. To understand how this works however, I will need to explain asynchronous functions, promises and what Async/Await do.

Asynchronous Functions

One of the main hurdles I had to figure out to get the implementation working was learning about how JavaScript handles API calls. JavaScript is a synchronous programming language, meaning that code is run through line by line, with everything executing within the environment - in this case the browser (Oseburg, F 2020)[10]. Web APIs however, although built into the browser, are not actually a part of the JavaScript language. This means that when I use the "fetch" API call to request data from Strava and Last.fm, it is being handled separately to the JavaScript code but within the same browser environment (MDN Web Docs 2021)[11].

For JavaScript to receive the results of the fetch API requests, it would typically have to finish running all of the synchronous code first. This would not work for my web-app however, as the data returned from the fetch requests is what is manipulated in my code and how the final results are generated.

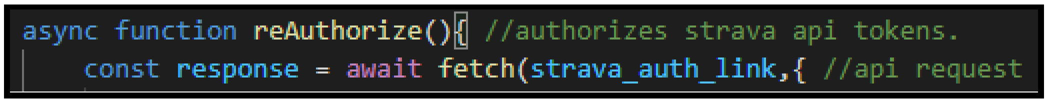
Promises

A promise object represents the eventual completion or failure of a request (Garrison, B 2017)[12]. A promise can be one of three states: pending, fulfilled or rejected. Promises are the data structure returned when a fetch request is made.

Async/Await

The solution I found, after finally understanding the above, was to use Async/Await for each of my API request functions. This is a way of writing asynchronous functions that work in a synchronous way. When one is called, it pauses the JavaScript code until the fetch promise has been fulfilled. *Figure 16* is a code snippet of Async/Await in use:

[10] https://www.freecodecamp.org/news/async-await-javascript-tutorial/
[11] https://developer.mozilla.org/en-US/docs/Web/API
[12]
https://medium.com/@_bengarrison/javascript-es8-introducing-async-await-functions-7a471ec7de8a



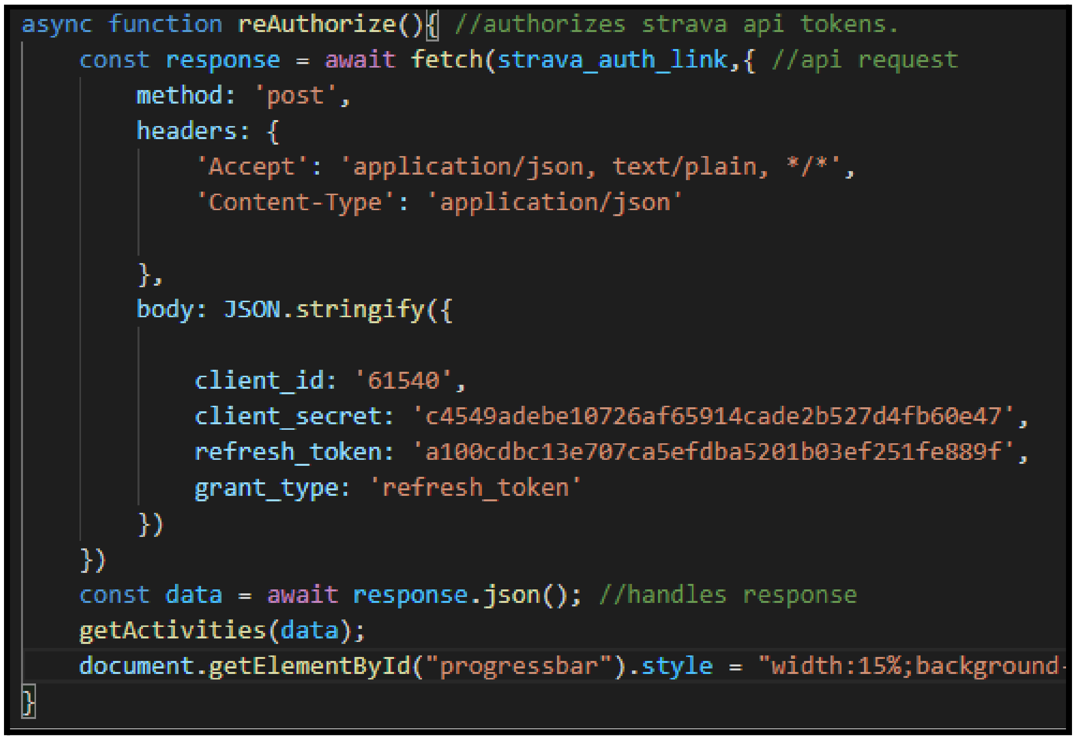
```
async function reAuthorize(){ //authorizes strava api tokens.
    const response = await fetch(strava_auth_link,{ //api request
```

Figure 16: Async/Await

Before figuring this out, every time I made a request for a user's Last.fm music data, it would execute the rest of the JavaScript program before fulfilling the promise and returning the results. This meant that I couldn't properly create the workout objects, as they would get created with the songs_played attribute being empty. This caused several weeks of errors and debugging.

API Calls

The first API call made is to authorize the tokens given by Strava. To get this working, I followed a tutorial YouTube tutorial (Polignano, F 2020)[13] of Strava's "Getting Started" documentation (Strava Developers, 2021)[14]. The client ID, the client secret, the refresh token and the grant type are sent to the Strava API, and if all these credentials are correct an access token is sent back. This access token is then put into JSON format and is used to request user data (*Figure 17*):

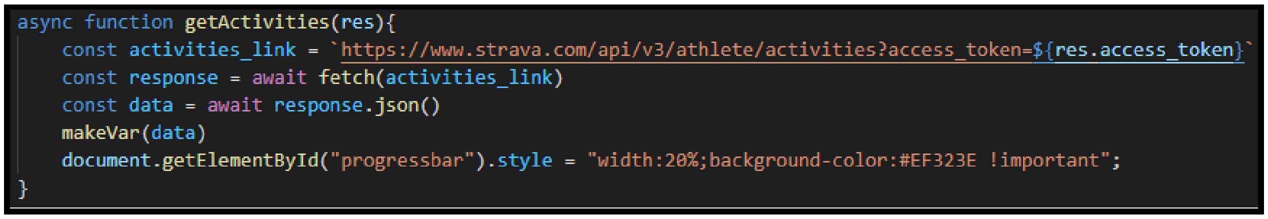


```
async function reAuthorize(){ //authorizes strava api tokens.
    const response = await fetch(strava_auth_link,{ //api request
        method: 'post',
        headers: {
            'Accept': 'application/json, text/plain, */*',
            'Content-Type': 'application/json'

        },
        body: JSON.stringify({

            client_id: '61540',
            client_secret: 'c4549adebe10726af65914cade2b527d4fb60e47',
            refresh_token: 'a100cdbc13e707ca5efdba5201b03ef251fe889f',
            grant_type: 'refresh_token'
        })
    })
    const data = await response.json(); //handles response
    getActivities(data);
    document.getElementById("progressbar").style = "width:15%;background
}
```

Figure 17: Strava API refresh token request

In *Figure 18*, the method is "post", meaning that data is being sent to a server. Responses need to be converted into JSON.

[13] https://github.com/franchyze923/Code_From_Tutorials/tree/master/Strava_Api
[14] https://developers.strava.com/docs/getting-started/

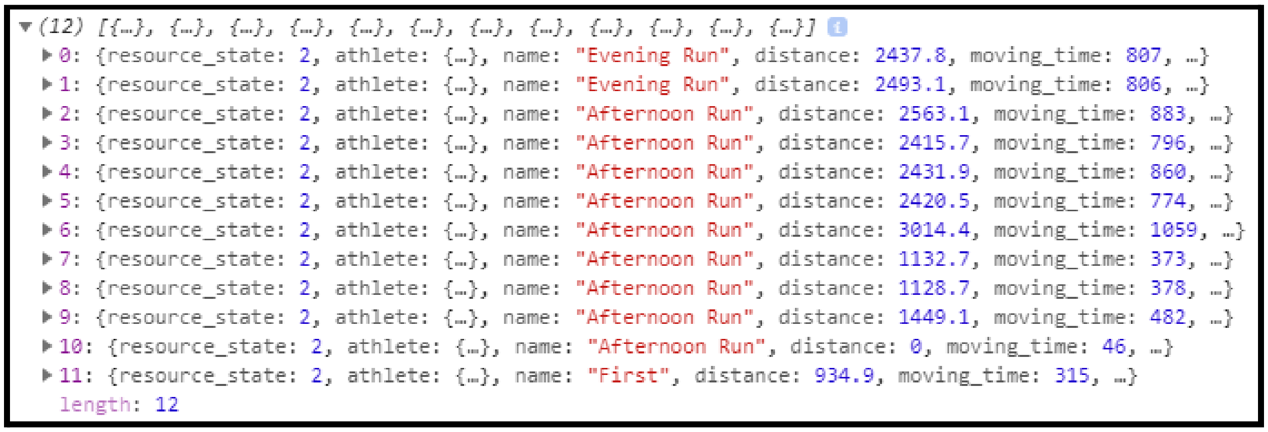


```
async function getActivities(res){
    const activities_link = `https://www.strava.com/api/v3/athlete/activities?access_token=${res.access_token}`
    const response = await fetch(activities_link)
    const data = await response.json()
    makeVar(data)
    document.getElementById("progressbar").style = "width:20%;background-color:#EF323E !important";
}
```

Figure 18: Strava API user activities request

Data Handling

The response of this request is every workout a user has ever done formatted in JSON. *Figure 19* is a response from my personal account:

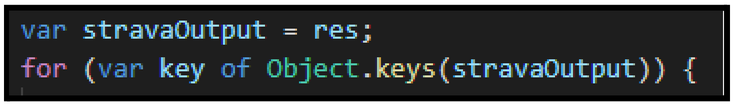


```
▼(12) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶0: {resource_state: 2, athlete: {…}, name: "Evening Run", distance: 2437.8, moving_time: 807, …}
  ▶1: {resource_state: 2, athlete: {…}, name: "Evening Run", distance: 2493.1, moving_time: 806, …}
  ▶2: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 2563.1, moving_time: 883, …}
  ▶3: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 2415.7, moving_time: 796, …}
  ▶4: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 2431.9, moving_time: 860, …}
  ▶5: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 2420.5, moving_time: 774, …}
  ▶6: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 3014.4, moving_time: 1059, …}
  ▶7: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 1132.7, moving_time: 373, …}
  ▶8: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 1128.7, moving_time: 378, …}
  ▶9: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 1449.1, moving_time: 482, …}
  ▶10: {resource_state: 2, athlete: {…}, name: "Afternoon Run", distance: 0, moving_time: 46, …}
  ▶11: {resource_state: 2, athlete: {…}, name: "First", distance: 934.9, moving_time: 315, …}
   length: 12
```

Figure 19: Strava request console logged in JSON format

Within each workout object is a very long list of attributes about the workout. This ranges from the number of athletes to the timezone the workout was done in. For the scope of this project however I am only interested in the attributes listed out in the userWorkout class.

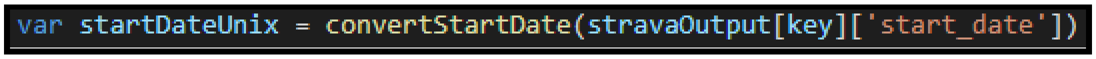
To create an object of each workout with the attributes, I passed the request results into a function called "makeObj". To iterate through a JSON object, you go through each key and then find the attribute corresponding to that key. In the Strava response, each workout is a key (*Figure 20)*:



```
var stravaOutput = res;
for (var key of Object.keys(stravaOutput)) {
```

Figure 20: Strava output iteration

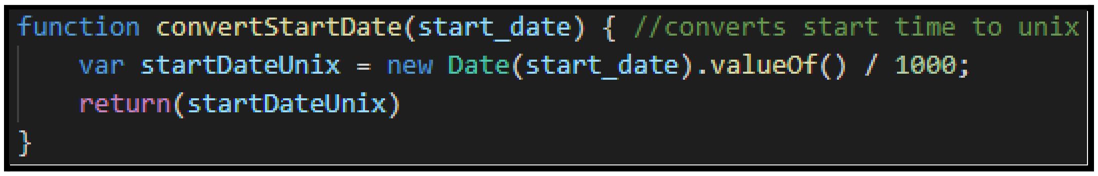
For every key, the first functions called are to get the start date and the elapsed time, and to convert them into unix time (*Figure 21*):



```
var startDateUnix = convertStartDate(stravaOutput[key]['start_date'])
```

Figure 21: Calling of function to convert the start date

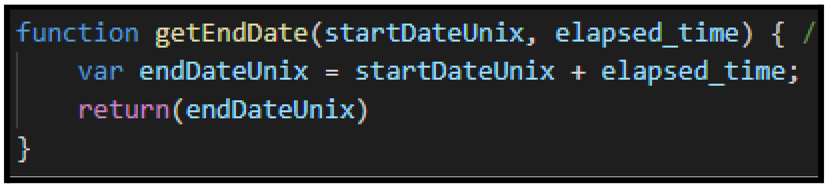
The dates need to be converted into unix time for the Last.fm API calls (*Figure 22*):



```
function convertStartDate(start_date) { //converts start time to unix
    var startDateUnix = new Date(start_date).valueOf() / 1000;
    return(startDateUnix)
}
```

Figure 22: Function to convert date to Unix

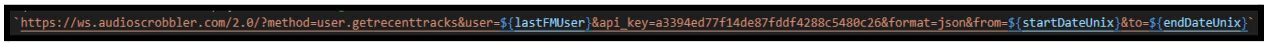
To calculate the end date, you just add the converted start date and elapsed time together (*Figure 23*):



```
function getEndDate(startDateUnix, elapsed_time) { /
    var endDateUnix = startDateUnix + elapsed_time;
    return(endDateUnix)
}
```

Figure 23: Function to get the end date in Unix

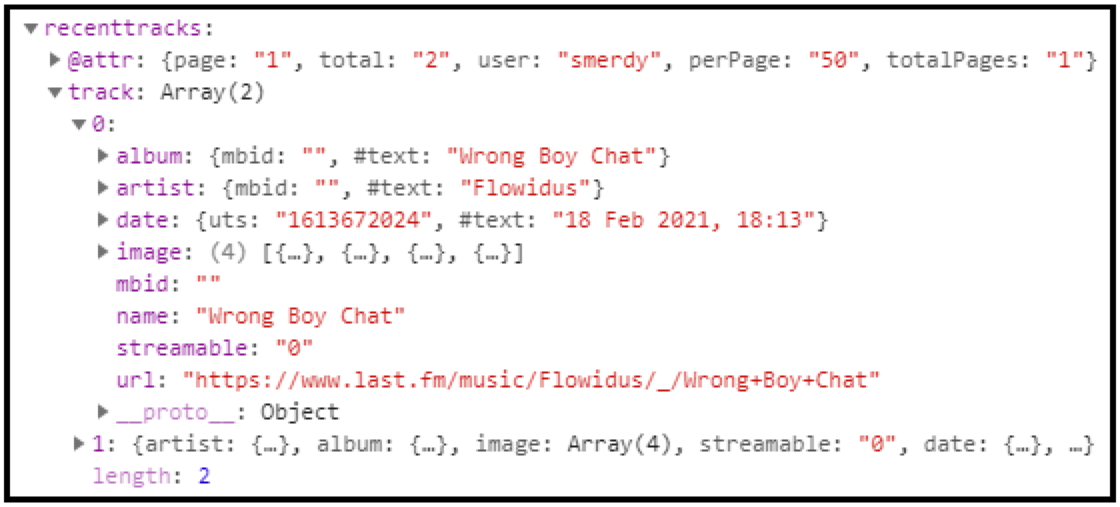
The next function called within the for loop is to get the tracks played within the workout. This is done by sending a fetch request to the Last.fm API with the username and the start and end dates (in unix time). This can be seen in *Figure 24*:



```
`https://ws.audioscrobbler.com/2.0/?method=user.getrecenttracks&user=${lastFMUser}&api_key=a3394ed77f14de87fddf4288c5480c26&format=json&from=${startDateUnix}&to=${endDateUnix}`
```

Figure 24: Last.fm API call for user's tracks

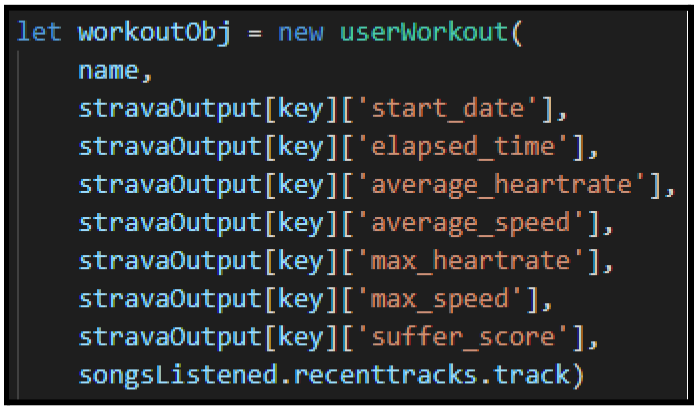
Figure 25 shows an example of a Last.fm fetch result:



```
▼ recenttracks:
  ▶ @attr: {page: "1", total: "2", user: "smerdy", perPage: "50", totalPages: "1"}
  ▼ track: Array(2)
    ▼ 0:
      ▶ album: {mbid: "", #text: "Wrong Boy Chat"}
      ▶ artist: {mbid: "", #text: "Flowidus"}
      ▶ date: {uts: "1613672024", #text: "18 Feb 2021, 18:13"}
      ▶ image: (4) [{…}, {…}, {…}, {…}]
        mbid: ""
        name: "Wrong Boy Chat"
        streamable: "0"
        url: "https://www.last.fm/music/Flowidus/_/Wrong+Boy+Chat"
      ▶ __proto__: Object
    ▶ 1: {artist: {…}, album: {…}, image: Array(4), streamable: "0", date: {…}, …}
      length: 2
```

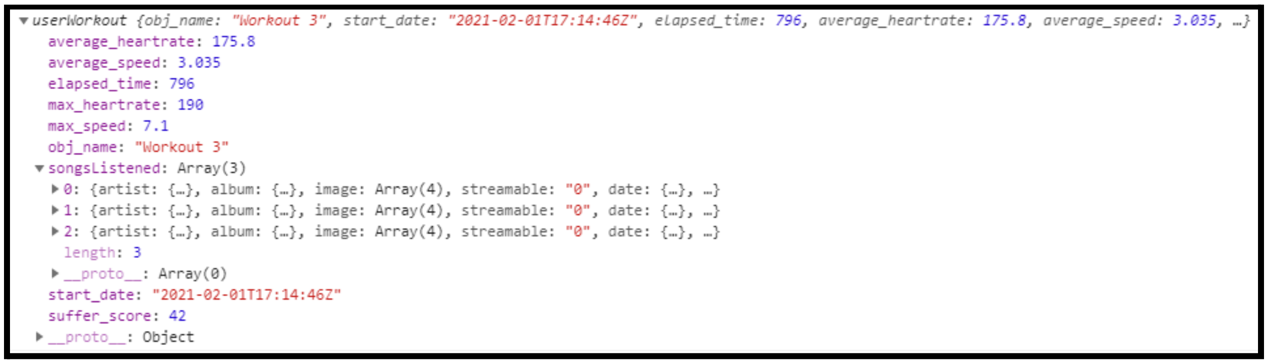
Figure 25: Last.fm user songs request console logged in JSON format

To name each object, I combined the string “Workout” with the key. The next step was to create an instance of the workout with the “userWorkout” class constructor. All the variables are within the scope of the loop, and so are simply passed in. This can be seen in *Figure 26*, and an example object in JSON can be seen in *Figure 27*:



```
let workoutObj = new userWorkout(
    name,
    stravaOutput[key]['start_date'],
    stravaOutput[key]['elapsed_time'],
    stravaOutput[key]['average_heartrate'],
    stravaOutput[key]['average_speed'],
    stravaOutput[key]['max_heartrate'],
    stravaOutput[key]['max_speed'],
    stravaOutput[key]['suffer_score'],
    songsListened.recenttracks.track)
```

Figure 26: Creation of workout object



```
▼ userWorkout {obj_name: "Workout 3", start_date: "2021-02-01T17:14:46Z", elapsed_time: 796, average_heartrate: 175.8, average_speed: 3.035, …}
    average_heartrate: 175.8
    average_speed: 3.035
    elapsed_time: 796
    max_heartrate: 190
    max_speed: 7.1
    obj_name: "Workout 3"
  ▼ songsListened: Array(3)
    ▶ 0: {artist: {…}, album: {…}, image: Array(4), streamable: "0", date: {…}, …}
    ▶ 1: {artist: {…}, album: {…}, image: Array(4), streamable: "0", date: {…}, …}
    ▶ 2: {artist: {…}, album: {…}, image: Array(4), streamable: "0", date: {…}, …}
      length: 3
    ▶ __proto__: Array(0)
    start_date: "2021-02-01T17:14:46Z"
    suffer_score: 42
  ▶ __proto__: Object
```

Figure 27: Example of a userWorkout Object in JSON

Each object is then added to an array. After the For loop has iterated through every key in the Strava response, all workout objects have been created. Each workout object includes the tracks being listened to during the workout. The array of all workouts is then passed into a function called “getBest”.

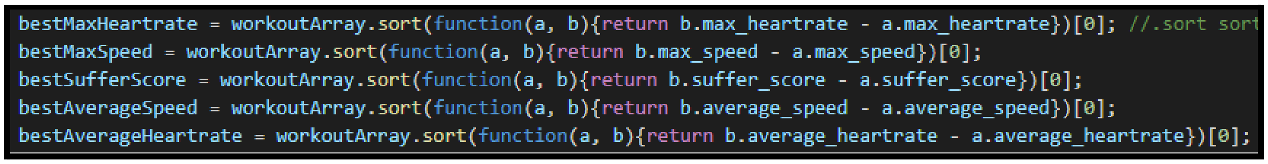
Finding the Best Workouts

As mentioned, these are the ways I am defining the “best” workouts:

- Best average heart rate
- Best average speed
- Best maximum heart rate
- Best maximum speed
- Best suffer score

A suffer score is Strava’s own way of calculating how intense a workout has been judging by your heart rate variation across a whole workout.

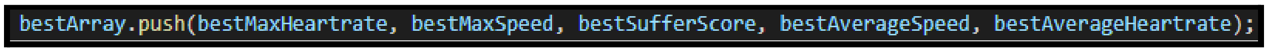
To find the best of an attribute, JavaScript has a built-in sort function which compares every attribute in an array and puts it in ascending order. To get the best I select the first in the array (*Figure 28*):



```
bestMaxHeartrate = workoutArray.sort(function(a, b){return b.max_heartrate - a.max_heartrate})[0]; //.sort sort
bestMaxSpeed = workoutArray.sort(function(a, b){return b.max_speed - a.max_speed})[0];
bestSufferScore = workoutArray.sort(function(a, b){return b.suffer_score - a.suffer_score})[0];
bestAverageSpeed = workoutArray.sort(function(a, b){return b.average_speed - a.average_speed})[0];
bestAverageHeartrate = workoutArray.sort(function(a, b){return b.average_heartrate - a.average_heartrate})[0];
```

Figure 28: JavaScript sort functions

Each of these best objects are then pushed to an array (*Figure 29*):



```
bestArray.push(bestMaxHeartrate, bestMaxSpeed, bestSufferScore, bestAverageSpeed, bestAverageHeartrate);
```

Figure 29: Objects being pushed to bestArray

Genre Breakdown

The next part of the implementation focuses on generating data about the songs themselves. The web-app works by having an initial overview of all the top workouts generated, and then giving options to the user to choose the specific top workout they want to see. I will first explain how song information gets generated then show how this works with user selection.

To get an overview of all the genres from all the top workouts, the array in *Figure 30* is passed into the asynchronous function “getTrackInformation”:

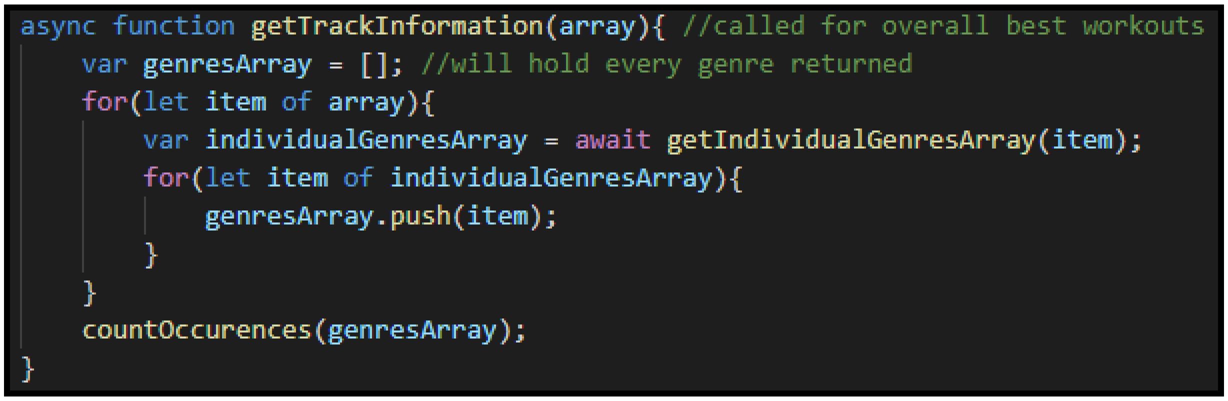

```
async function getTrackInformation(array){ //called for overall best workouts
    var genresArray = []; //will hold every genre returned
    for(let item of array){
        var individualGenresArray = await getIndividualGenresArray(item);
        for(let item of individualGenresArray){
            genresArray.push(item);
        }
    }
    countOccurences(genresArray);
}
```

Figure 30: getTrackInformation function

Within this function, each individual workout is passed into a function called "getIndividualGenresArray" (*Figure 31*):

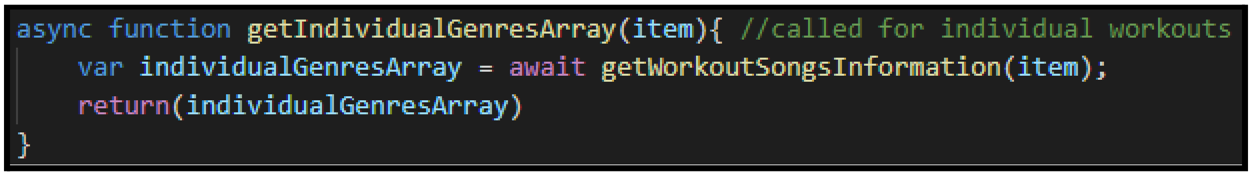

```
async function getIndividualGenresArray(item){ //called for individual workouts
    var individualGenresArray = await getWorkoutSongsInformation(item);
    return(individualGenresArray)
}
```

Figure 31: getIndividualGenresArrray

Within this function, each workout is passed into a function called "getWorkoutSongsInformation". This is the function that returns the genres within a workout. It works by iterating through every track within a workout (*Figure 32*):

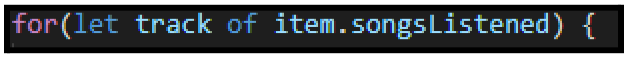
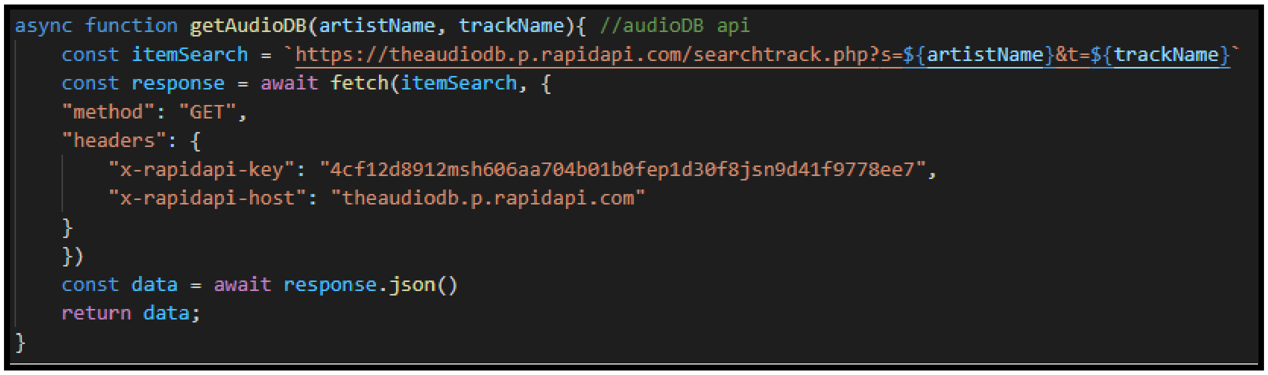

```
for(let track of item.songsListened) {
```

Figure 32: Iteration of songs

The first check is to see if the first track has a property called "@attr". When Last.fm calls are made, if a user is currently listening to a track, it will return this within the call even if it is not within the time frame. Currently playing tracks have this property, which is why this check is needed. Next, the artist name and the song name are turned into strings and passed into a function called "getAudioDB". This can be seen in *Figure 33.*

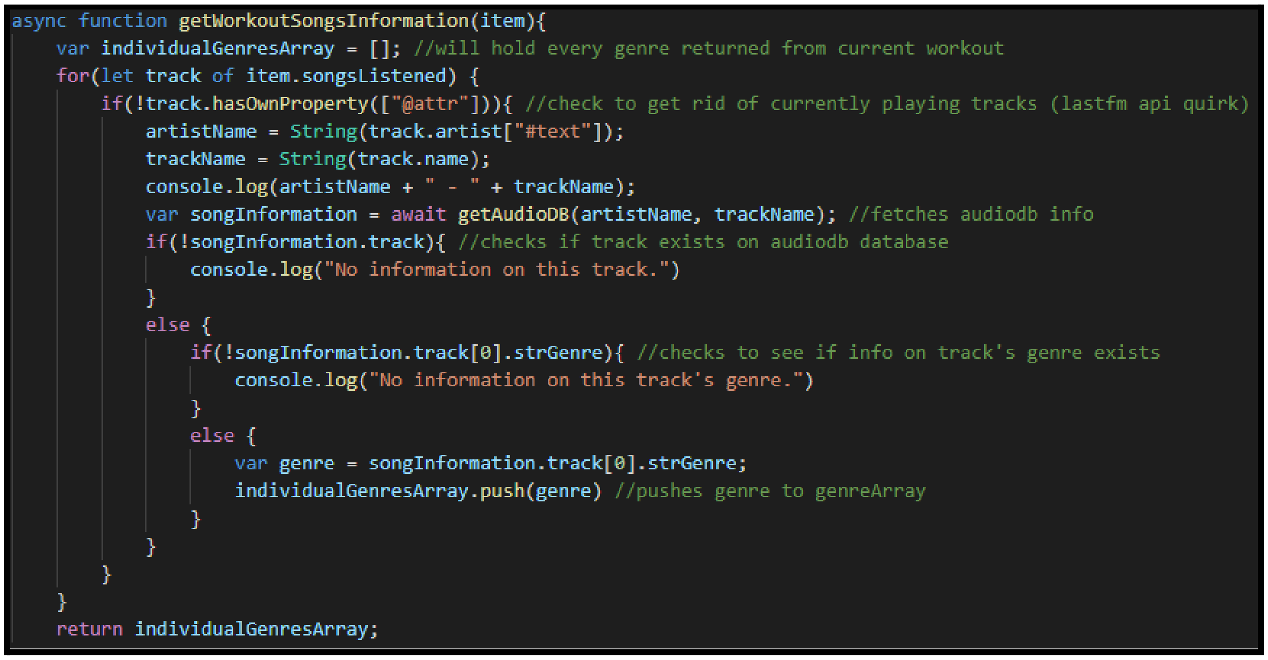
The "getAudioDB" function is how the implementation actually finds a track's genre using the AudioDB's song database:



```
async function getAudioDB(artistName, trackName){ //audioDB api
    const itemSearch = `https://theaudiodb.p.rapidapi.com/searchtrack.php?s=${artistName}&t=${trackName}`
    const response = await fetch(itemSearch, {
    "method": "GET",
    "headers": {
        "x-rapidapi-key": "4cf12d8912msh606aa704b01b0fep1d30f8jsn9d41f9778ee7",
        "x-rapidapi-host": "theaudiodb.p.rapidapi.com"
    }
    })
    const data = await response.json()
    return data;
}
```

Figure 33: getAudioDB function

The artist's name and track name are put into the request, and the response is returned to the "getWorkoutSongsInformation" function (*Figure 34*). A check is then made to see if the track exists on the database. If it does exist, a check is then made to see if there is information about the track's genre on the database. If there is, the genre is added to an array of all the genres in the workout. This is all done using Async/Await.

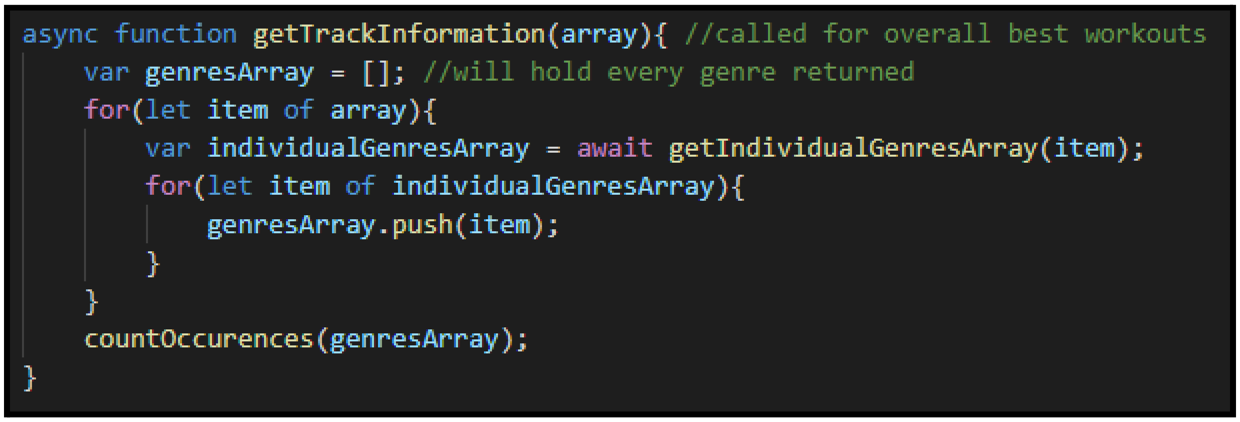


```
async function getWorkoutSongsInformation(item){
    var individualGenresArray = []; //will hold every genre returned from current workout
    for(let track of item.songsListened) {
        if(!track.hasOwnProperty(["@attr"])){ //check to get rid of currently playing tracks (lastfm api quirk)
            artistName = String(track.artist["#text"]);
            trackName = String(track.name);
            console.log(artistName + " - " + trackName);
            var songInformation = await getAudioDB(artistName, trackName); //fetches audiodb info
            if(!songInformation.track){ //checks if track exists on audiodb database
                console.log("No information on this track.")
            }
            else {
                if(!songInformation.track[0].strGenre){ //checks to see if info on track's genre exists
                    console.log("No information on this track's genre.")
                }
                else {
                    var genre = songInformation.track[0].strGenre;
                    individualGenresArray.push(genre) //pushes genre to genreArray
                }
            }
        }
    }
    return individualGenresArray;
```

Figure 34: getWorkoutSongsInformation function

In figure x, a console log is done when a track or its genre are not found, however this is not presented to the user unless they open up the console.

After all tracks have been iterated through and their genres added to an array, the array is returned to the "getIndividualGenresArray", which returns this to the "getTrackInformation" function (*Figure 35*). Each item of individualGenresArray is then added to an array called “genresArray”. This process then repeats until every song’s genre from the top workouts has been added “genresArray”:



```
async function getTrackInformation(array){ //called for overall best workouts
    var genresArray = []; //will hold every genre returned
    for(let item of array){
        var individualGenresArray = await getIndividualGenresArray(item);
        for(let item of individualGenresArray){
            genresArray.push(item);
        }
    }
    countOccurences(genresArray);
}
```

Figure 35: Showing the full “getTrackInformation” function

“genresArray” is then passed into a function called “countOccurences”.

Last.fm and Spotify Track Search

As previously mentioned during the background, the solution using the Audio DB’s API is not what I initially intended when designing my implementation. The initial idea was to use Last.fm’s API for getting track data. I assumed this would work, as the API is working for user data, however it turns out that, although in the documentation, Last.fm has seemingly abandoned support for this function. After getting many errors, the only discussion on this I could find was on the official Last.fm API forums saying that this functionality had been broken for a few years and not to expect it to work any time soon, or even expect Last.fm to address that it is broken. The implementation would be almost identical to what it is currently, except that the itemSearch URL in figure x would be to Last.fm’s servers.

Using this, I believe, would have allowed for some more interesting analysis as, unlike theAudioDB, Last.fm’s metadata on tracks includes similar tracks. I was hoping to use this to try and recommend similar tracks to the users of Heartbeats.

As an alternative to Last.fm, I then went back to trying to use Spotify’s API. This involved setting up a refresh token so that I always had a key to access the API. Spotify holds a significant amount more metadata on songs than both Last.fm and theAudioDB, and I assumed it would be great for the implementation, however I ran into a lot of issues surrounding the track search.

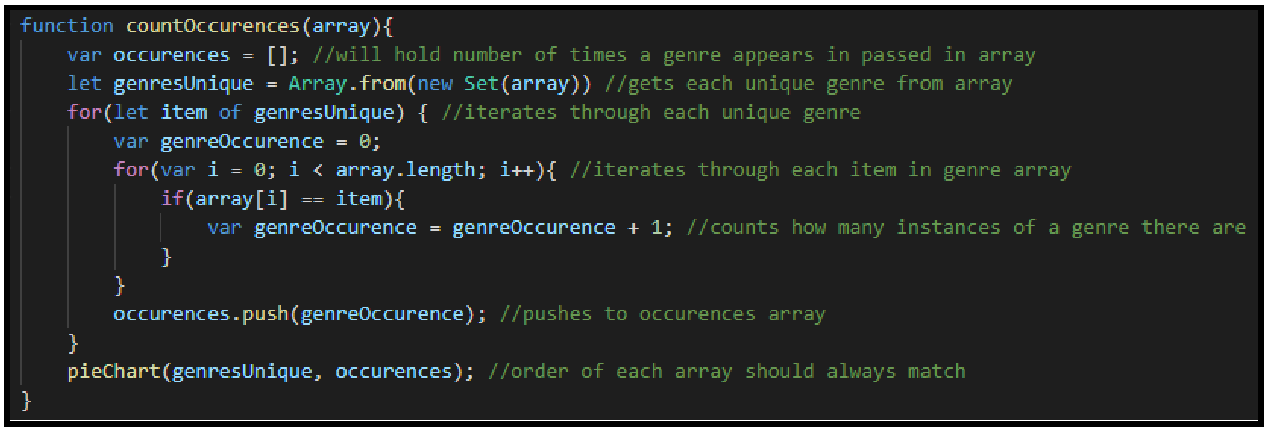
Although functional when using the Spotify app, and despite the fact Spotify has one of the most documented and extensive APIs going, I was unable to find a way to just search for an artist and a song and get a result. As the implementation currently does with the AudioDB, I was hoping to just pass this data in automatically through a function, but Spotify’s API search is designed for broad searches of artists or albums. Every combination of track and artist request sent to Spotify's API would return wildly different results (ranging from prioritising cover bands and cover songs to tracks to songs with different names), and in most cases return nothing at all. A good week and a half of the development time was spent setting up the API and trying to get this search function working, however in the end I had to cut my losses as I had fallen behind the work plan and use theAudioDB instead.

I feel this is the most significant limitation with the solution, as Spotify's song metadata includes attributes like the BPM, as well as genres and similar tracks. I was hoping to calculate the average BPM of the top tracks, as I felt this would be a very interesting piece of data to see. Heartbeats might also then suggest tracks or Spotify playlists that a user might be interested in. It is also an API I am already paying for with my general Spotify subscription, whereas theAudioDB is an extra monthly subscription I need to pay for.

Count Occurences and Chart.js

To show a breakdown of the genres, I decided the best way was to use a pie chart, with each section being the number of times a genre appears. This would make seeing the top genres listened to easy. The best solution to getting a pie chart in JavaScript was Chart.js (Chart.js 2021)[15].

The Chart.js pie chart abstracts away most of the code involved with getting a working animated pie chart in JavaScript, and only requires the labels and corresponding dataset from users of the plugin (Chart.js 2021)[16]. This data is generated in the "countOccurences" function (*Figure 36*):



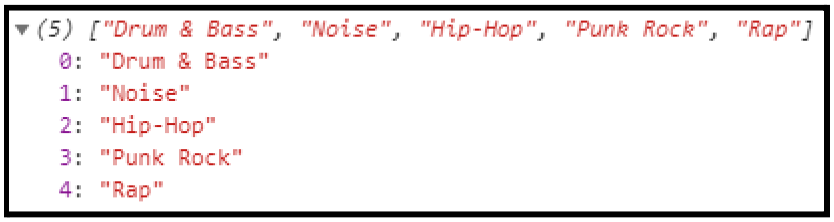
```
function countOccurences(array){
    var occurences = []; //will hold number of times a genre appears in passed in array
    let genresUnique = Array.from(new Set(array)) //gets each unique genre from array
    for(let item of genresUnique) { //iterates through each unique genre
        var genreOccurence = 0;
        for(var i = 0; i < array.length; i++){ //iterates through each item in genre array
            if(array[i] == item){
                var genreOccurence = genreOccurence + 1; //counts how many instances of a genre there are
            }
        }
        occurences.push(genreOccurence); //pushes to occurences array
    }
    pieChart(genresUnique, occurences); //order of each array should always match
}
```

Figure 36: countOccurences function

The first thing that happens in the function is a new array called "genresUnique" (*Figure 37*) is created from the array passed in that only contains one instance of every genre. For example, if

[15] https://www.chartjs.org/docs/latest/
[16] https://www.chartjs.org/docs/3.0.2/charts/doughnut.html

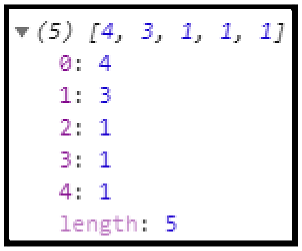
the array were “D&B, D&B, D&B, Electronic, Electronic, House”, then the new array would just contain “D&B, Electronic, House”. This is what forms the labels for the pie chart.



```
▼(5) ["Drum & Bass", "Noise", "Hip-Hop", "Punk Rock", "Rap"]
   0: "Drum & Bass"
   1: "Noise"
   2: "Hip-Hop"
   3: "Punk Rock"
   4: "Rap"
```

Figure 37: Example of “genresUnique” array console logged

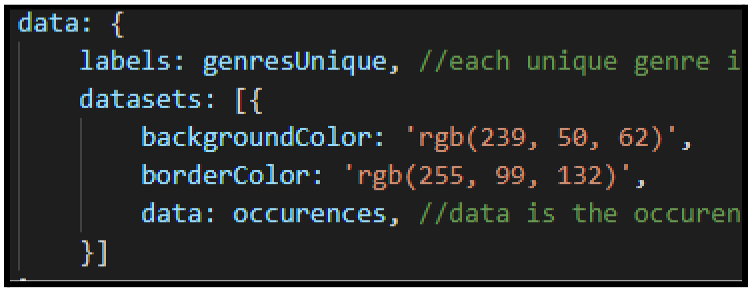
Next, this new array is iterated through item by item, and on each loop every genre in the initial array is compared to the item to count the number of times they appear. The result of each loop (“genreOccurence”) is then added to an array called occurrences (*Figure 38*). Both arrays are then passed into the “pieChart” function, which creates the unique pie chart. The order of the “occurrences” array will correlate with the “genresUnique” array.



```
▼(5) [4, 3, 1, 1, 1]
   0: 4
   1: 3
   2: 1
   3: 1
   4: 1
   length: 5
```

Figure 38: Example of “occurrences” array

The Pie Chart is drawn onto an HTML canvas. *Figure 39* shows the arrays in the “pieChart” function, and *Figure 40* shows the complete pie chart:



```
data: {
    labels: genresUnique, //each unique genre i
    datasets: [{
        backgroundColor: 'rgb(239, 50, 62)',
        borderColor: 'rgb(255, 99, 132)',
        data: occurences, //data is the occuren
    }]
```

Figure 39: Datasets in Chart.js pie chart

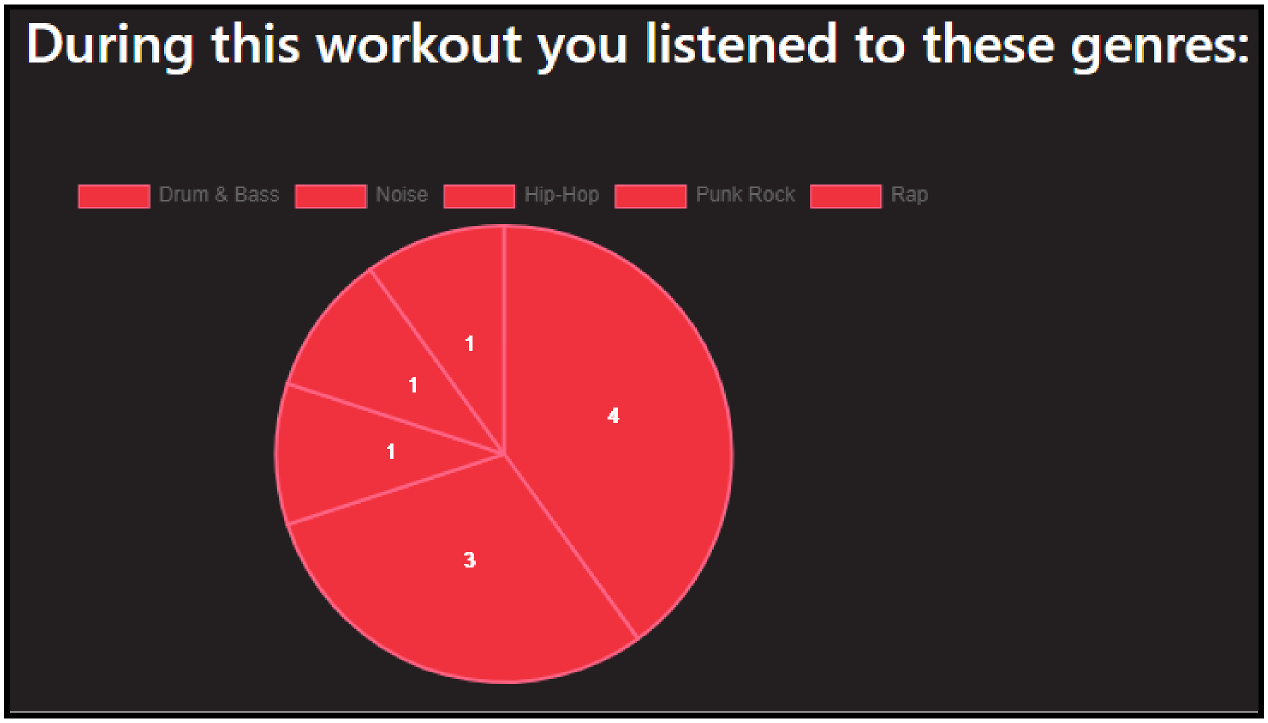


Figure 40: Pie chart in web-app

Results Selection and Displaying Workout

The final part of the implementation is around how the user selects which workout breakdown they would like to see and how the tracks listened to during this workout are displayed. The steps gone through in the previous section are for a breakdown of all the top workouts, however for individual best workouts all of these functions are reused.

I decided the best approach would be to use a dropdown box of all the best workouts, and overall (*Figure 41* and *Figure 42*):

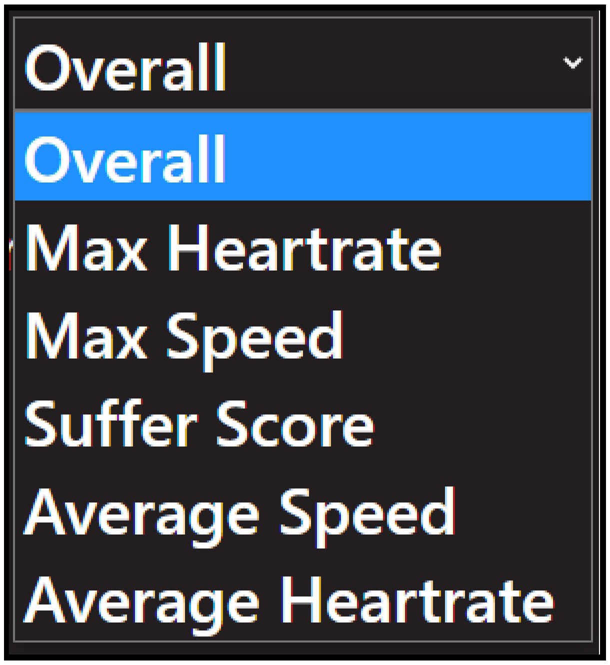
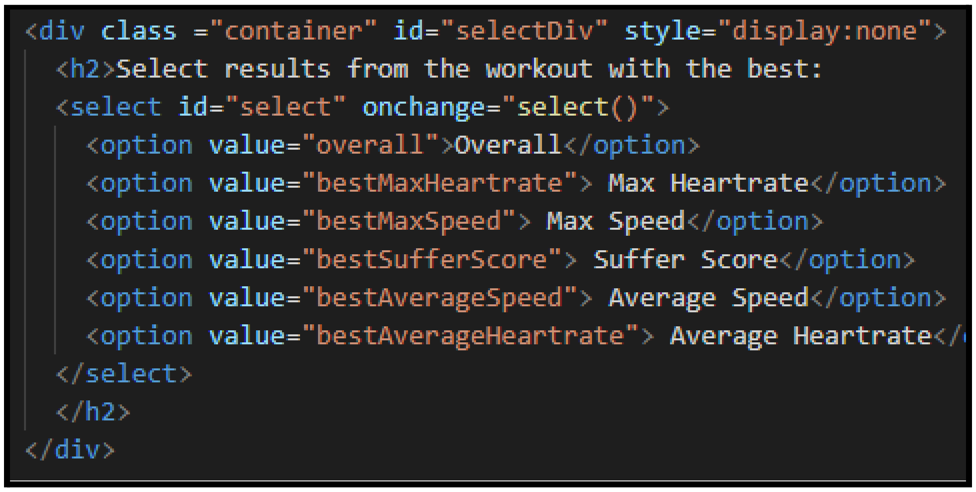


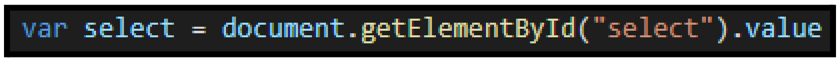
Figure 41: Dropdown box of best workouts



```
<div class ="container" id="selectDiv" style="display:none">
  <h2>Select results from the workout with the best:
  <select id="select" onchange="select()">
    <option value="overall">Overall</option>
    <option value="bestMaxHeartrate"> Max Heartrate</option>
    <option value="bestMaxSpeed"> Max Speed</option>
    <option value="bestSufferScore"> Suffer Score</option>
    <option value="bestAverageSpeed"> Average Speed</option>
    <option value="bestAverageHeartrate"> Average Heartrate</
  </select>
  </h2>
</div>
```

Figure 42: HTML of dropdown box

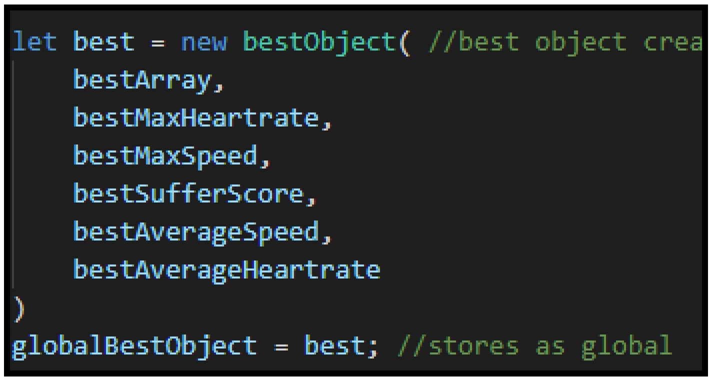
A function called “select” is called every time this selection box changes (*Figure 43*):



```
var select = document.getElementById("select").value
```

Figure 43: Select variable

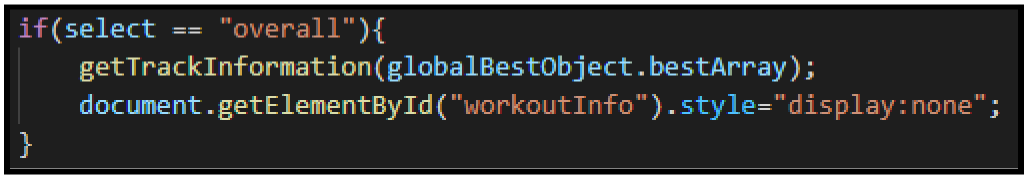
In the “getBest” function, an object called “best” was created using the “bestObject" class constructor. This object was then stored as a global variable called “globalBestObject”. This can be seen in *Figure 44*:



```
let best = new bestObject( //best object crea
    bestArray,
    bestMaxHeartrate,
    bestMaxSpeed,
    bestSufferScore,
    bestAverageSpeed,
    bestAverageHeartrate
)
globalBestObject = best; //stores as global
```

Figure 44: Instance of a bestObject class

Whilst the local object was used to create the initial overall results, all subsequent results are generated through the global variable. For example, if the user selects Overall again from the dropdown box, the following code (*Figure 45*) is run to the generate the results:

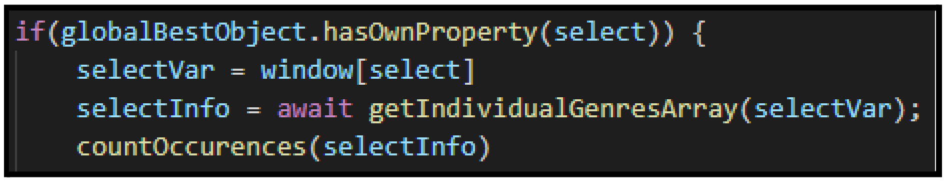


```
if(select == "overall"){
    getTrackInformation(globalBestObject.bestArray);
    document.getElementById("workoutInfo").style="display:none";
}
```

Figure 45: Code that runs when the user selects “Overall”

The array stored in the global object (“globalBestObject.bestArray”) is passed into the “getTrackInformation” function. This then calls all the previously mentioned functions that generate the genre data.

If the user selects an individual best workout (*Figure 46*), just the workout is passed into the getIndividualGenresArray function, and the returned results from that are passed into the countOccurences function:



```
if(globalBestObject.hasOwnProperty(select)) {
    selectVar = window[select]
    selectInfo = await getIndividualGenresArray(selectVar);
    countOccurences(selectInfo)
```

Figure 46: Code that runs when user selects an individual best workout

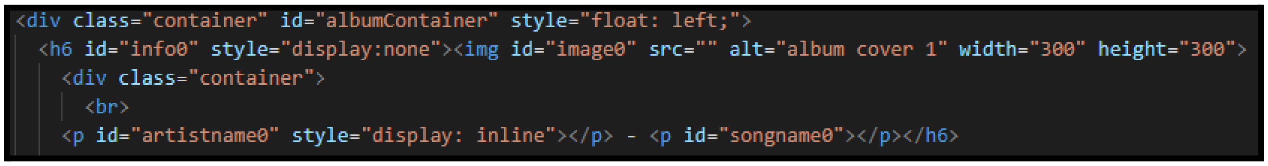
The date of the workout is converted into a readable format using a piece of code available on Stack Overflow(Lee, W 2016)[17]. This code turns the Unix time (which are stored as seconds) back into minutes and hours.

17 https://stackoverflow.com/questions/37096367/how-to-convert-seconds-to-minutes-and-hours-in-javascript/49905383

Displaying Songs

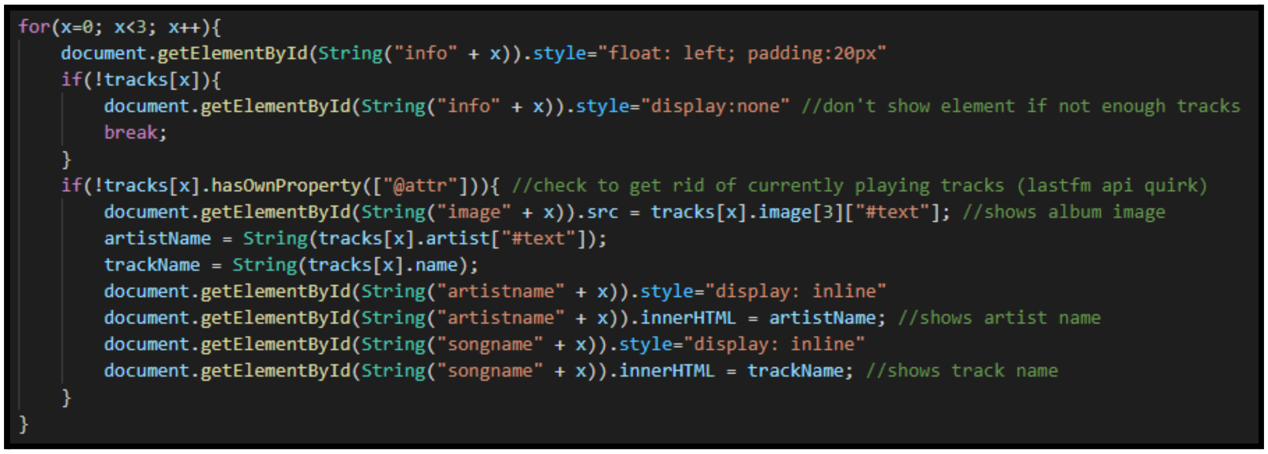
I decided that the most user-friendly way of displaying the tracks the user had listened to during a workout would be through an image of the album art, along with the artist and track underneath. As the amount of tracks a person can listen to during a workout can vary, I decided to only show the first three tracks (or fewer for very short workouts). The album art images are already fetched from Last.fm and in the workout objects.

In the HTML, there are three song containers (*Figure 47*) which are hidden until the required data is passed into them (*Figure 48*). *Figure 49* shows this in action.



```html
<div class="container" id="albumContainer" style="float: left;">
  <h6 id="info0" style="display:none"><img id="image0" src="" alt="album cover 1" width="300" height="300">
    <div class="container">
      <br>
    <p id="artistname0" style="display: inline"></p> - <p id="songname0"></p></h6>
```

Figure 47: Example of an HTML album container



```javascript
for(x=0; x<3; x++){
    document.getElementById(String("info" + x)).style="float: left; padding:20px"
    if(!tracks[x]){
        document.getElementById(String("info" + x)).style="display:none" //don't show element if not enough tracks
        break;
    }
    if(!tracks[x].hasOwnProperty(["@attr"])){ //check to get rid of currently playing tracks (lastfm api quirk)
        document.getElementById(String("image" + x)).src = tracks[x].image[3]["#text"]; //shows album image
        artistName = String(tracks[x].artist["#text"]);
        trackName = String(tracks[x].name);
        document.getElementById(String("artistname" + x)).style="display: inline"
        document.getElementById(String("artistname" + x)).innerHTML = artistName; //shows artist name
        document.getElementById(String("songname" + x)).style="display: inline"
        document.getElementById(String("songname" + x)).innerHTML = trackName; //shows track name
    }
}
```

Figure 48: Code to display songs

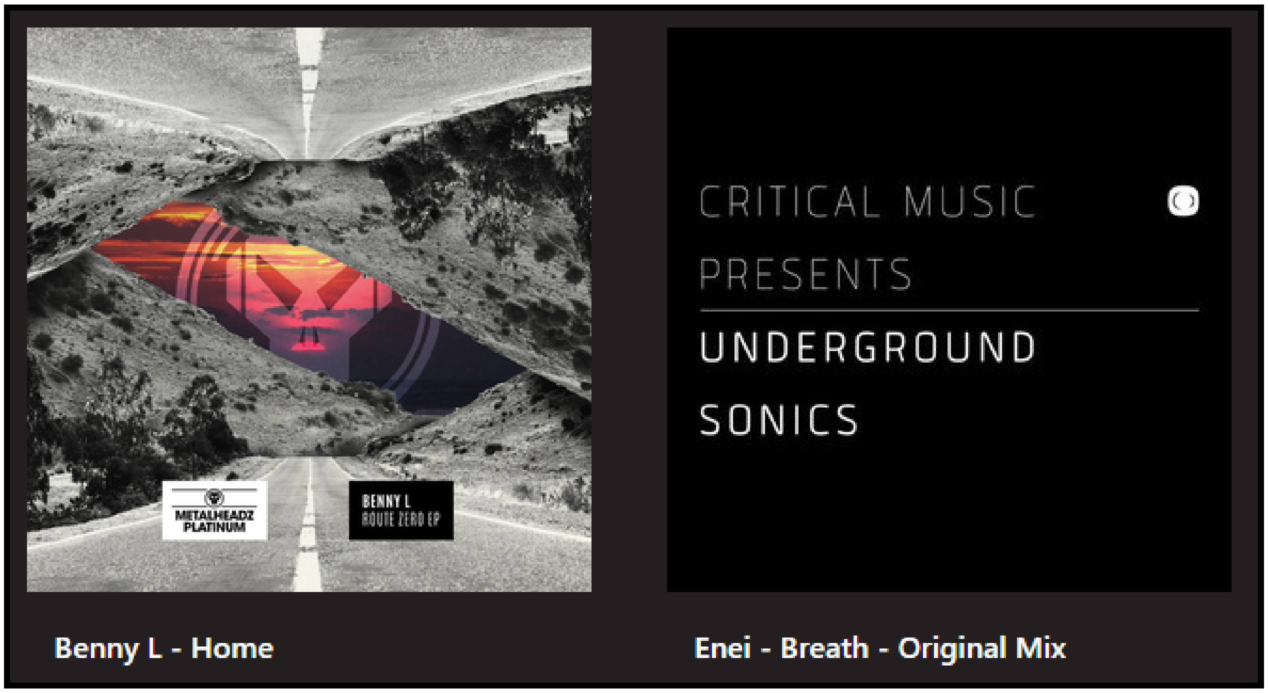


Figure 49: Tracks played during workout displayed in web-app

HTML and CSS

In my project, I am using a number of libraries to make my site look better. The main one is Bootstrap (reference), which is a popular toolkit for designing and building good-looking sites and web-apps (Bootstrap 2021)[18]. These libraries are imported in the HTML head.

In terms of design, my web-app is going for a minimalist look with a consistent colour scheme of black and red to match my logo, with white text. The progress bar (shown in *Figure 50*) works by updating the class style's width as each JavaScript function is called. The progress bar lets the user know that data generation is working.

Figure 50: Part of the Loading Bar

Each time the user selects a different workout, the previous Pie Chart canvas is deleted but then gets redrawn (Farber, M 2017)[19].

[18] https://getbootstrap.com/docs/4.4/getting-started/introduction/
[19] https://stackoverflow.com/questions/40056555/destroy-chart-js-bar-graph-to-redraw-other-graph-in-same-canvas

Web Hosting

As one of the requirements in the initial project overview was to have the web-app available for anyone to use, and also because it was a personal goal of mine to learn, I needed to find a way to get my website hosted online. My initial solution was to use Amazon Web Services (AWS 2021)[20] because through my limited knowledge and research I found they were considered the best place to go to for web-hosting. After spending a few days trying to get a site to load with just the IP to no avail however, I ran out of time on the plan and instead was forced to just use GitHub pages to host my project. This wasn't ideal, as I had already spent the money on the domain "heatbeatsapp.net", however it does come with some advantages, mainly being it's completely free and any changes made to my code will automatically be uploaded to the site with just a simple git push command. It is also accessible to anyone, which fulfills one of my initial criteria for the project.

Security

One of the main requirements for this project was for the web-app to be secure. I have already discussed API keys in the background section, however there are other ways in which security is achieved.

Firstly, I have made sure that here is no mixed content. Mixed content is when an initial website's HTML is loaded securely using HTTPS, but resources are then loaded using an insecure HTTP connection (Bergen, J. and Andrew, R 2019)[21]. This means that a page would be susceptible to man-in-the-middle attacks, where an attacker can intercept and modify packets between two entities (Daucez, D. et al. 2016)[22]. By making sure that all API requests are established using HTTPS this security risk can be avoided.

Another way security is achieved is with what personal information I request users give permission to use. Even though data is being requested securely, if there ever were some sort of breach the type of data exposed would not put the user at any risk. Only information regarding workouts and songs being listened to could ever be revealed, and in most cases this information is already public.

Finally, my web-app is secured because it is hosted on a site that uses a DDOS protection service, meaning any attack of this nature would be detected and mitigated.

The full implementation can be found at https://asmerdon.github.io/. It is currently set up so only my personal account runs for moderator demonstration purposes. Use the username "smerdy" to see results generation in action. To allow all users this will be changed (more on this in the Future Work section).

[20] https://aws.amazon.com/
[21] https://web.dev/what-is-mixed-content/
[22] https://www.ietf.org/rfc/rfc7835.html#section-2.1.1

Results and Evaluation

To test my implementation, I used my own personal accounts and did workouts whilst listening to a variety of music so that there would be a range of data to pull. As Strava accounts require a monthly subscription to use, I only had one account I could test with, however the implementation is designed to work for any account and has no hard-coded parts that would only work with the account I tested on. Variations in the songs I was listening to include tracks that cannot be found on theAudioDB database, tracks that do not have an album cover photo, and listening to few enough tracks that there would only be one genre once results for a workout were generated.

Test Cases

In these test cases, I used a fresh computer to run the tests as even with incognito mode, permissions are still saved, which would impact the results generation.

Strava Login Testing:

Test No.	**Action**	**Expected Result**	**Actual Result**	**Test Result**	**Comments**	**Screen shot**
1	Entering login details and allowing permissions.	User gets redirected back to the web-app.	User is redirected back to the web-app.	Pass	Works correctly, but user can still click “Login with Strava” button on web-app.	Appendix *Figure 56*
2	Entering incorrect login details.	User should be prompted to try again.	User is prompted to try again.	Pass	Handled by Strava.	N/A
3	Entering login details and rejecting permissions.	User is redirected back to web-app but cannot generate results.	User is sent back to web app.	Pass	“Access denied” appears in URL.	Appendix *Figure 57*

Last.fm Username Input Testing:

Test No.	Action	Expected Result	Actual Result	Test Result	Comments	Screenshot
4	Entering valid username and clicking generate results button.	Button is greyed out, loading bar will appear and results generation will begin.	Loading bar appears and results are generated.	Pass	Username used was “smerdy”.	Appendix *Figure 58*
5	Entering invalid username and clicking generate results button.	Prompt should appear beneath the button prompting user to try entering username again.	“Try again” prompt appears.	Pass	N/A	Appendix *Figure 59*
6	Entering nothing and clicking generate results button.	User should see a prompt beneath button asking them to try again.	“Try again” prompt appears.	Pass	N/A	Appendix *Figure 60*
7	Entering non-alphabet characters and clicking generate results.	User should see a prompt beneath button asking them to try again.	“Try again” prompt appears.	Pass	N/A	Same as Test Case 5

Results Generation Testing:

Test No.	Action	Expected Result	Actual Result	Test Result	Comments	Screenshot
8	Result	Results	Results	Fail	As mentioned,	N/A

	generation is attempted without a connected Strava account.	generation fails.	are generated with my personal account.		it is currently configured so only my personal account is linked (more on this is Future Work).	
9	Results generation is attempted with a valid username but not related with the Strava account.	Results generation fails.	No genre breakdown or track breakdown appears, however the loading bar still completes.	Pass	This may be confusing to users so there should be a check to see if no tracks are being returned.	N/A
10	Results generation with correct user details.	Results are successfully generated.	Results are successfully generated.	Pass	N/A	Appendix *Figure 61*
11	Results generation done with music being listened to.	Results are successfully generated and three tracks listed.	Genres breakdown is successful, but first track album art does not appear.	Fail	Last.fm returns what the user is currently listening to no matter the time period specified in request. My attempt to get around this did not work.	Appendix *Figure 62*
12	Breakdown with more than three tracks listened to in a workout.	First three tracks of workout should be displayed.	First three tracks of workout are displayed.	Pass	N/A	Appendix *Figure 63*

13	Breakdown with fewer than three tracks listened to during a workout.	All tracks listened to should be displayed.	All tracks are displayed.	Pass	N/A	Same as Test Case 10.
14	Select “Overall” option	Overall genre breakdown and tracks listed.	Expected result listed.	Pass	No date listed as it is broken down across multiple workouts.	Same as Test Case 10
15	Select “Max Heart Rate” option	Date of workout, genre breakdown and tracks listed.	Expected result listed.	Pass	N/A	N/A - Same as previou s figures.
16	Select “Max Speed” option	Date of workout, genre breakdown and tracks listed.	Expected result listed.	Pass	N/A	N/A - Same as previou s figures.
17	Select “Suffer Score” option	Date of workout, genre breakdown and tracks listed.	Expected result listed.	Pass	N/A	N/A - Same as previou s figures.
18	Select “Average Speed” option	Date of workout, genre breakdown and tracks listed.	Expected result listed.	Pass	N/A	N/A - Same as previou s figures.
19	Select “Average Heart Rate” option	Date of workout, genre breakdown and tracks listed.	Expected result listed.	Pass	N/A	N/A - Same as previou s figures.

To test the success rate of the data generation, I ran it 100 times (*Figure 51*) and recorded whether it passed or failed. I found that out of the 100 tests, 99 passed and 1 failed (*Figure 52*). The one failure was due to a failure to fetch the Strava results:

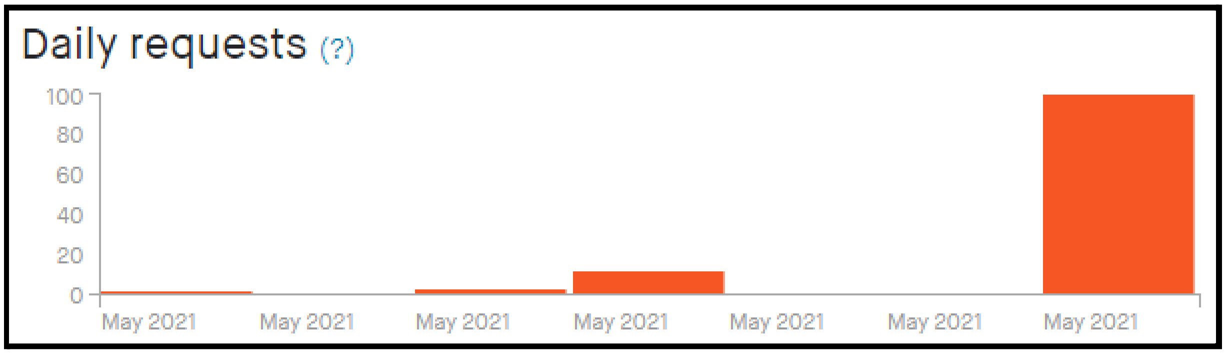


Figure 51: Strava requests in API menu

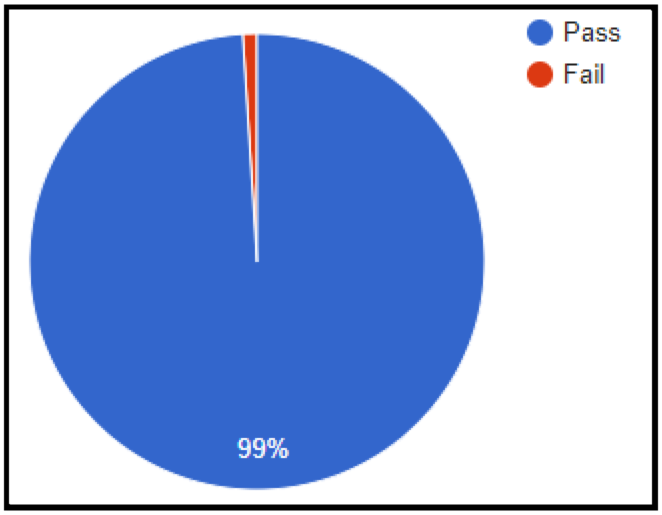


Figure 52: Data Generation Success Rate

The full table of results can be seen in the appendix *Figure 55*.

Initial Objectives

Here is a chart of the objectives from the initial plan, and whether or not I completed them:

Objective Description	Complete/Partially Complete/Incomplete	Comments
Users are able to link their Strava and Spotify accounts to the website.	Partially Complete	*Changed Spotify login to Last.fm

Once a user links their account, the API should automatically fetch the user's data for the website.	Complete	User's data fetched on the click of a button.
Users must have the ability to revoke access to data at any point.	Complete	Users can revoke access through Strava settings.
API use falls within guidelines set by the company.	Complete	API data requested is limited to scope of solution, and requests limited automatically.
Once their data has been fetched using the APIs, users should get results automatically generated for them.	Complete	Results generated at the click of a button.
These results will show the impact of their listening habits on their workouts and will be based off of their best and worst workout performances on Strava.	Partially Complete	*Only the best performances are shown to the user.
Results will include top songs and their genres.	Complete	Three of the top songs of each workout are shown, and genres broken down in a pie chart.
Results will be presented in a web-app.	Complete	Results load on a single-page web-app.
Design of the web-app will make results easy to read and understand.	Complete	Although this is entirely subjective to the user, I have attempted to make the
Website is live and hosted on a domain.	Partially Complete	Domain is on my GitHub pages and domain name is unrelated to the web-app.
Website is accessible to anyone.	Complete	Would only go down if GitHub pages goes down.

The website is secure from different forms of attacks. This includes DDOS attacks and attacks which target user’s data.	Complete	Website does not store user data, and so cannot be stolen from any server. GitHub pages offers DDOS protection.
Encryption methods implemented to make sure user data is secure.	Complete	API requests for user’s data use secure HTTPS requests.

Evaluation

Overall, although not the original vision I had for Heartbeats, I think the solution fulfils most of the criteria that I laid out in the initial report. I believe that the current state of the web-app achieves most of the goals that were laid out in the initial project plan, and is a very solid foundation for the addition of more features. I feel that a lot of the work that went into the implementation is not actually seen in the end result. It is a shame that some of the original planned features could not happen, however I think that given the circumstances of the issues (being out of my control) I feel the solution is the best I could do. If I had more time I would perhaps look for alternate solutions to get these features working (perhaps such as by finding a better API than The Audio DB’s). The approach I took to development was I believe the most appropriate, however I do have some regrets with choosing to use just default JavaScript, instead of an alternative like Node.Js. This will be discussed further in the next section

Future Work

Whilst I am proud of the state the project is currently in, I believe that there is a lot of room for improvement. Sadly due to the time frame, I was unable to implement all the features I was hoping to, and was also unable to fix a few bugs, however from the experience gained from developing in JavaScript and working with APIs would make a future project in these areas a lot quicker and easier to implement.

In its current state, I feel that users do not get much information about their workout, and that perhaps more explanation needs to be given so that they can understand the data presented to them (for example, the Suffer Score is not explained to users). Improving error handling by giving feedback to the user why data generation has failed, along with a way to retry without refreshing the page, would also improve functionality.

One feature that I envisioned implementing was a map of a user's run with the location they started listening to a track placed on top. This is likely feasible, as I already request a Polyline (Google Maps Platform 2021)[23] during the results generation, so all I would need to do is correlate the song times to this. Sadly I did not have enough development time to attempt to implement this feature.

As mentioned during the implementation section, another feature I was hoping to add was the average BPM of the user's tracks. This data is in tracks in Spotify's database, so if the track search for Spotify's API ever works this feature would not be too difficult to add. The same goes with track recommendations, as they are in both Spotify and Last.fm's databases.

Getting the web-app hosted on a proper domain instead of GitHub pages would be a small task I would like to get done because, as previously mentioned, I have already spent money on the "heatbeatsapp.net" domain name. I would probably use an alternative to AWS as I found the platform to be way too confusing and complicated for the small task of hosting a website.

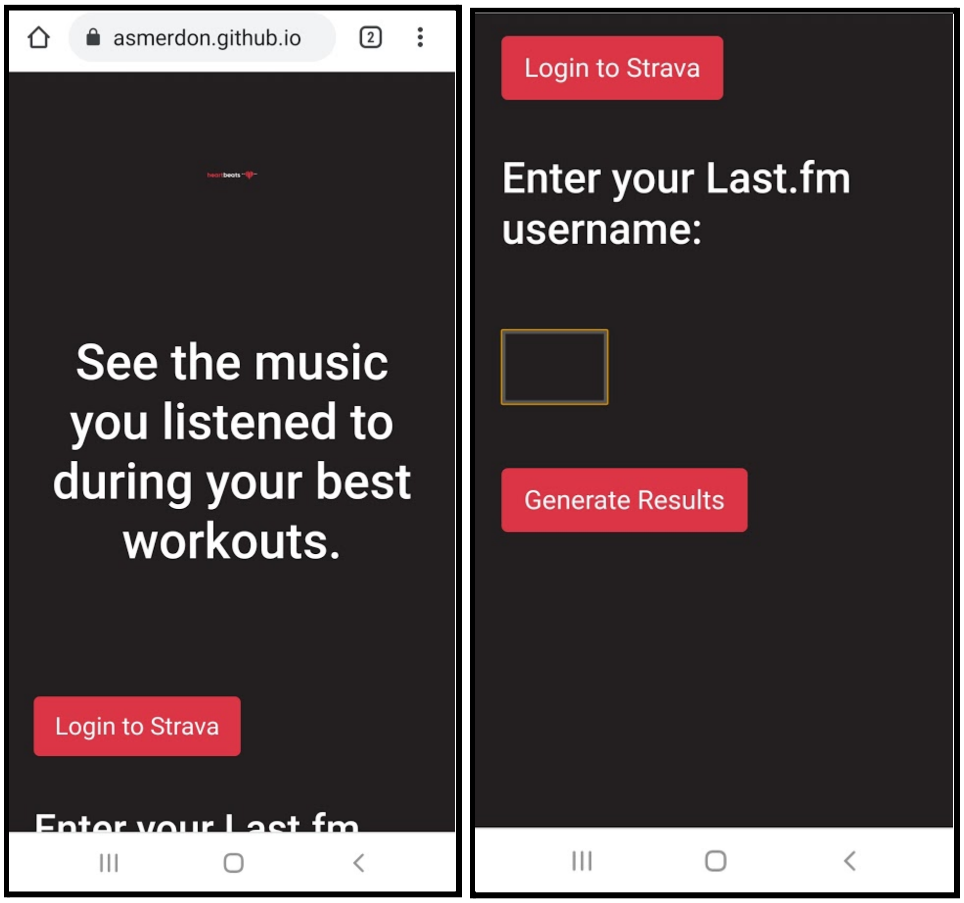
Finally, re-doing how the results are generated would cut down on the amount of requests and speed up results generation. Currently, every time a user selects to view a different workout type (or view overall), the functions that fetch the requests that find the track genres are called. This could be changed so that in the initial overall breakdown requests, the track genres are stored and fetched locally rather than through an API request each time. This would require a reworking of several functions, as well as how data is presented within the web-app, however again would cut down on the generation time and the amount of API requests done.

[23] https://developers.google.com/maps/documentation/utilities/polylineutility

Known Bugs

There are a few small bugs that I know about but haven't had the time to fix. Firstly, if the user is listening to music when they generate a result, the track (including album art) of a workout will not display (this can be seen in the appendix Figure x).

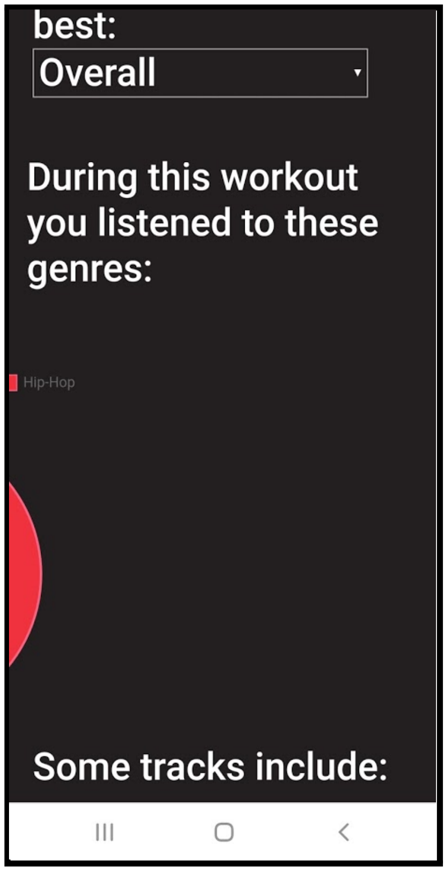
Next, if the user does not have a heart rate monitor setup, it is unknown whether or not the implementation will run. Finally, there are small problems with my HTML implementation. If the browser is resized, the logo will become a lot smaller and the pie chart will misalign. This can best be seen when the web-app is used on a mobile browser (see *Figure 53* and *Figure 54*). Within a browser as well the text box is too small for most inputs.


asmerdon.github.io

See the music you listened to during your best workouts.

Enter your Last.fm username:

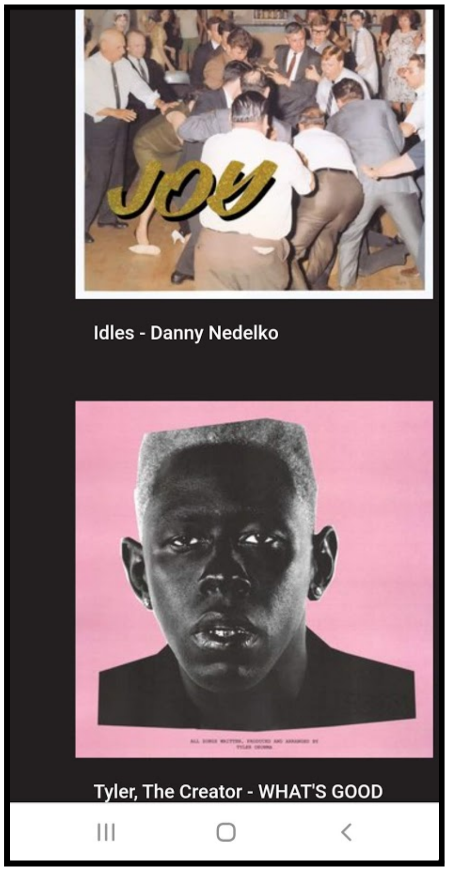
Figure 53: Heartbeats on a mobile browser


best:

Overall

During this workout you listened to these genres:

Some tracks include:


Idles - Danny Nedelko

Tyler, The Creator - WHAT'S GOOD

Figure 54: Results generation on a mobile browser

Finally, as mentioned in the Implementation section and Test Case 8, the solution is set up currently so that only my personal account will generate results. This is for demonstration purposes. This can be changed to allow for any user by using the authorisation response query string code (the code in the URL once a user connects their account) to generate a refresh token.

With some additional time, these errors could be quickly ironed out, however there are further more fundamental changes I would make to the implementation that would take longer to do:

Node.js

If I were to start again from scratch, I believe that redoing the implementation in Node.js instead of standard JavaScript would make development a lot easier. Node.js is an asynchronous event-driven JavaScript runtime (Node.js 2020)[24] that is meant to build network applications. When I was attempting to get Heartbeats hosted on AWS, there was an option to just upload Node.js programs and have all the hosting done automatically. Being able to do this would have saved me a lot of hassle and would likely be a better solution. The fact Node.js is written to be asynchronous would have also made figuring out JavaScript asynchronous functions something

[24] https://nodejs.org/en/about/

I could have avoided, which would have saved me a lot of time during development. Using Node.js, I could have also hidden my API keys, making my web-app more secure.

Conclusions

The main aim of Heartbeats was to create a tool for athletes to use to improve their workouts with music analysis. This implementation contains the core features necessary to get analysis on what users listen to in their workouts, however there were several features I was hoping to include that I was unable to implement - be it from the timescale of the project to dependencies not working.

When compared to my initial aims, I feel that I have hit most of the points I was hoping to, however some areas are better implemented than others. I think the best work is with the API request and workout creation / comparison section of the implementation, as there were many iterations and the final one ended up being the fewest lines of code.

To test my project, I broke down each action a user could take and tested whether or not the expected outcome happened. I also tested the success rate of data generation, and compared my implementation to my initial project goals. These tests proved that there are still issues with the solution, and that there are bugs that need fixing as well as general improvements that could be made to the project.

Overall, this project has been mostly successful in its aims despite the challenges faced.

Reflection on Learning and Performance

Overall, I found areas of the project to be a lot more challenging than I initially anticipated, which caused me to rethink what my implementation would work. I feel if I had not spent time trying to figure out solutions to problems that actually couldn’t be solved (Last.fm and Spotify API for song data), I could have added a lot more features to the project and learnt about different areas (such as getting maps of workouts to work).

On the other hand, over the course of the project I have learnt a lot about areas of web-development that I had no prior experience with, and I feel I have developed areas I had some experience with a lot further. I am also proud of a lot of aspects of data generation, particularly with how workouts are created and sorted, as it was a problem I had to tackle completely alone and I feel is quite an elegant solution.

When it comes to time management, I feel as though I spent a lot longer on the solution than what is reflected in the results. I feel that to have avoided this, I should have done more research into how I was going to carry out my implementation, instead of diving straight in and figuring out the solution from there. I have already mentioned how doing the project in Node.js may have saved me a lot of trouble, but I also would have liked to learn how to work with

Node.Js because it is a lot more employable. Throughout the course of this project however, I feel I have become a lot more effective at managing my work with the time available.

When it comes to my programming skills, I think I have certainly developed my understanding of script-based languages, as well as my ability to independently come up with solutions. As this project was proposed by myself to a supervisor who wasn't familiar with what I wanted to complete, it fell completely on me to steer the direction of the project and come up with solutions, and so I am proud of what I have accomplished, and believe the skills I have picked up will help me with future programming projects - whether they're personal or professional.

Appendices

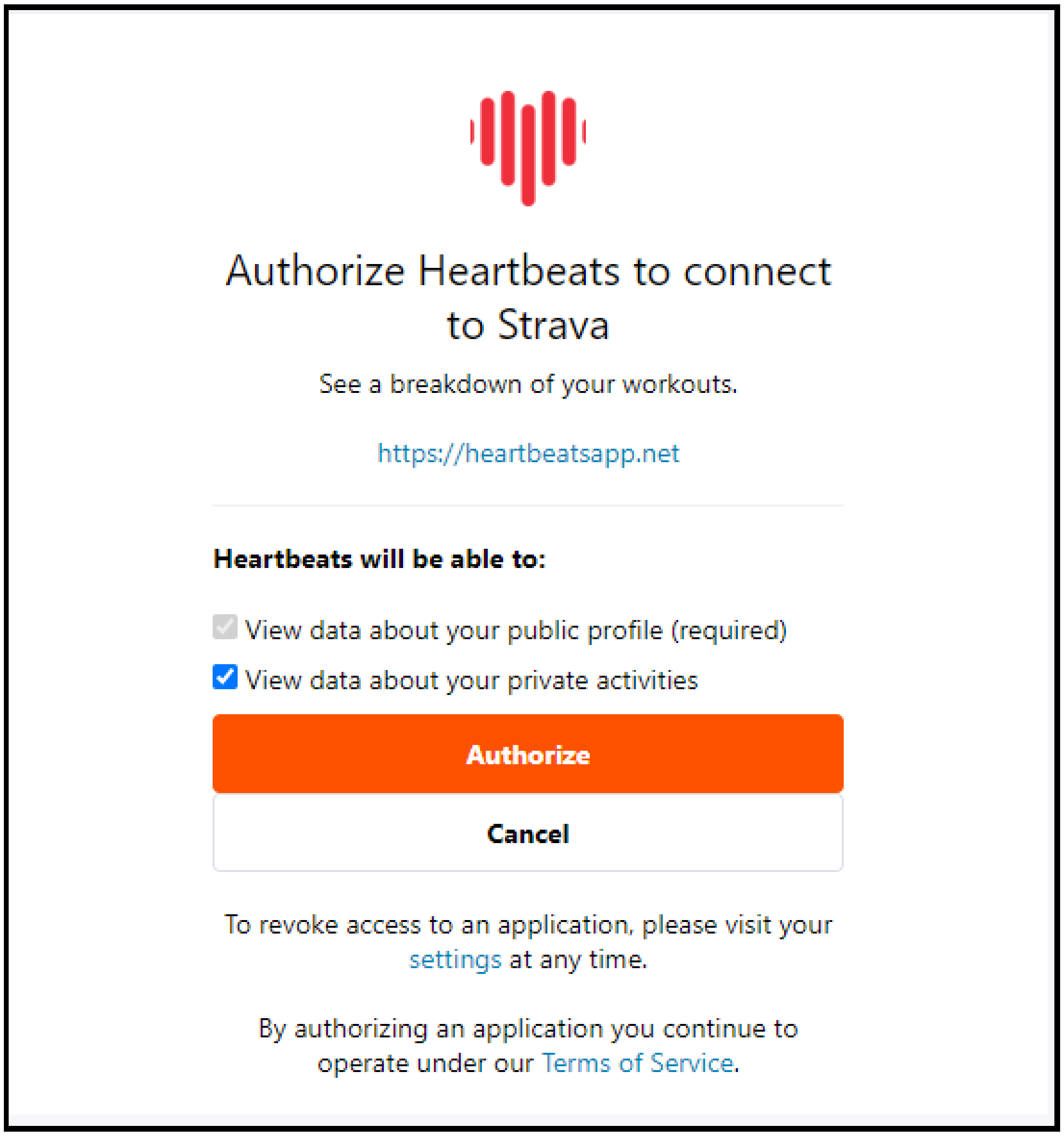
Solution: https://asmerdon.github.io/

Test Number	Pass/Fail	Comment
1	Pass	N/A
2	Pass	N/A
3	Pass	N/A
4	Pass	N/A
5	Pass	N/A
6	Pass	N/A
7	Pass	N/A
8	Pass	N/A
9	Pass	N/A
10	Pass	N/A
11	Pass	N/A
12	Pass	N/A
13	Pass	N/A
14	Pass	N/A
15	Pass	N/A
16	Pass	N/A
17	Pass	N/A
18	Pass	N/A
19	Pass	N/A
20	Pass	Took longer than average but still completed.
21	Pass	N/A
22	Pass	N/A
23	Pass	N/A
24	Pass	N/A
25	Pass	N/A
26	Pass	N/A
27	Pass	N/A
28	Pass	N/A
29	Pass	N/A
30	Pass	N/A
31	Pass	N/A

32	Pass	N/A
33	Pass	N/A
34	Pass	N/A
35	Pass	N/A
36	Pass	N/A
37	Pass	N/A
38	Pass	N/A
39	Pass	N/A
40	Pass	N/A
41	Pass	N/A
42	Pass	N/A
43	Pass	N/A
44	Pass	N/A
45	Pass	N/A
46	Pass	N/A
47	Pass	N/A
48	Pass	N/A
49	Pass	N/A
50	Pass	N/A
51	Pass	N/A
52	Pass	N/A
53	Pass	N/A
54	Pass	N/A
55	Pass	N/A
56	Pass	N/A
57	Pass	N/A
58	Pass	N/A
59	Pass	N/A
60	Pass	N/A
61	Pass	N/A
62	Pass	N/A
63	Pass	N/A
64	Pass	N/A
65	Pass	N/A
66	Pass	N/A
67	Pass	N/A
68	Pass	N/A
69	Pass	N/A
70	Pass	N/A
71	Pass	N/A
72	Pass	N/A
73	Pass	N/A
74	Pass	N/A
75	Pass	N/A
76	Pass	N/A
77	Pass	N/A
78	Pass	Took longer than average but still completed.
79	Pass	N/A
80	Pass	N/A
81	Pass	N/A

82	Fail	Error: Uncaught (in promise): TypeError: Failed to fetch
83	Pass	N/A
84	Pass	N/A
85	Pass	Took longer than average but still completed.
86	Pass	N/A
87	Pass	N/A
88	Pass	N/A
89	Pass	N/A
90	Pass	N/A
91	Pass	N/A
92	Pass	N/A
93	Pass	N/A
94	Pass	N/A
95	Pass	N/A
96	Pass	N/A
97	Pass	N/A
98	Pass	N/A
99	Pass	N/A
100	Pass	N/A

Figure 55: Full Generation Test Data



Authorize Heartbeats to connect to Strava

See a breakdown of your workouts.

https://heartbeatsapp.net

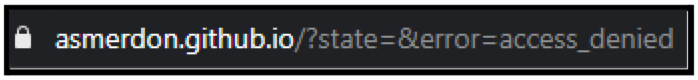
Heartbeats will be able to:

- [✓] View data about your public profile (required)
- [✓] View data about your private activities

To revoke access to an application, please visit your settings at any time.

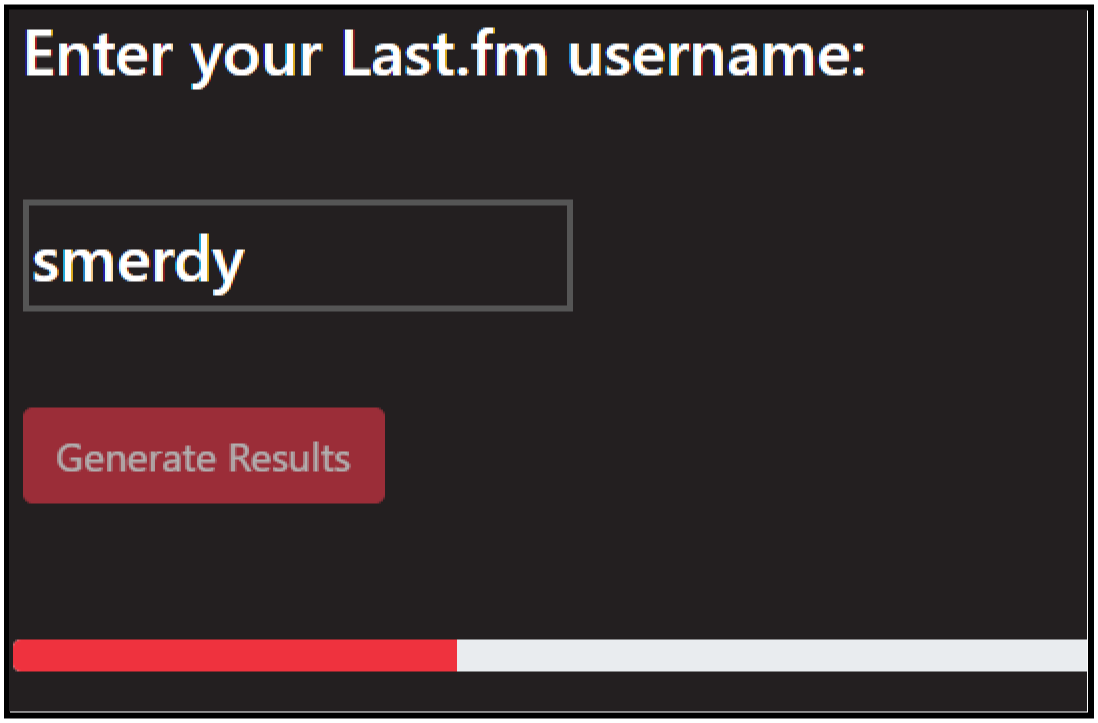
By authorizing an application you continue to operate under our Terms of Service.

Figure 56: Test Case 1



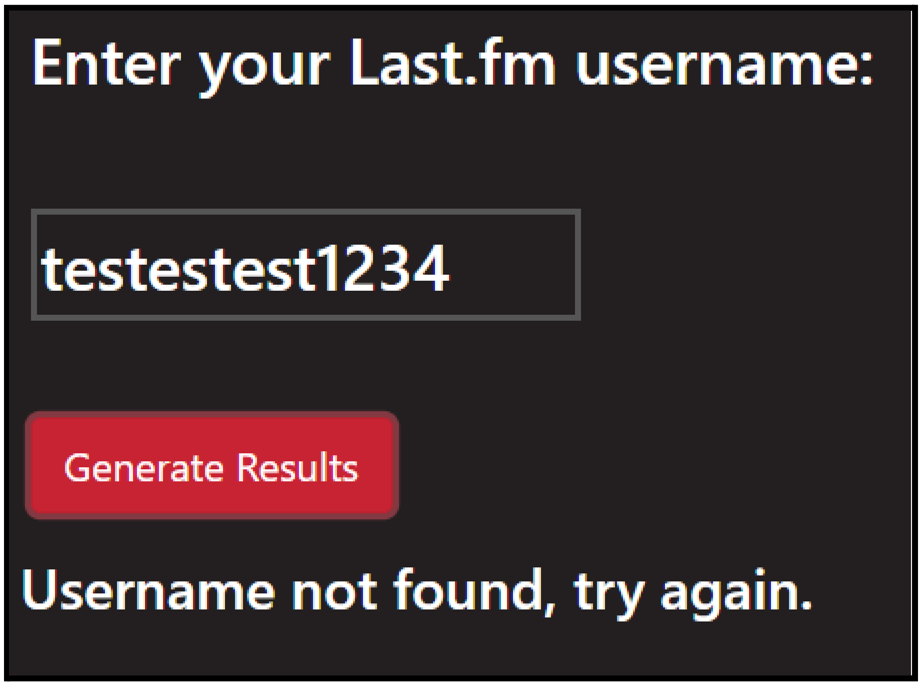
asmerdon.github.io/?state=&error=access_denied

Figure 57: Test Case 3


Enter your Last.fm username:

smerdy

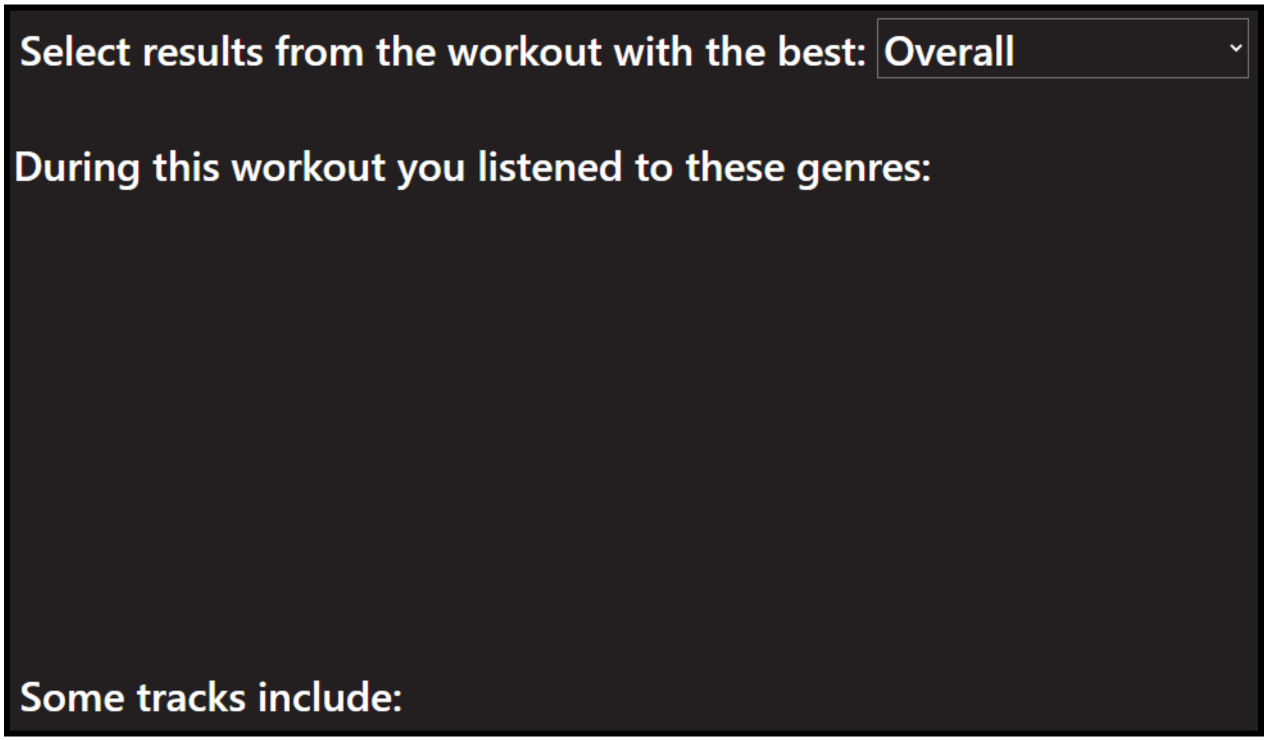
Figure 58: Test Case 4 - Valid input


Enter your Last.fm username:

testestest1234

Username not found, try again.

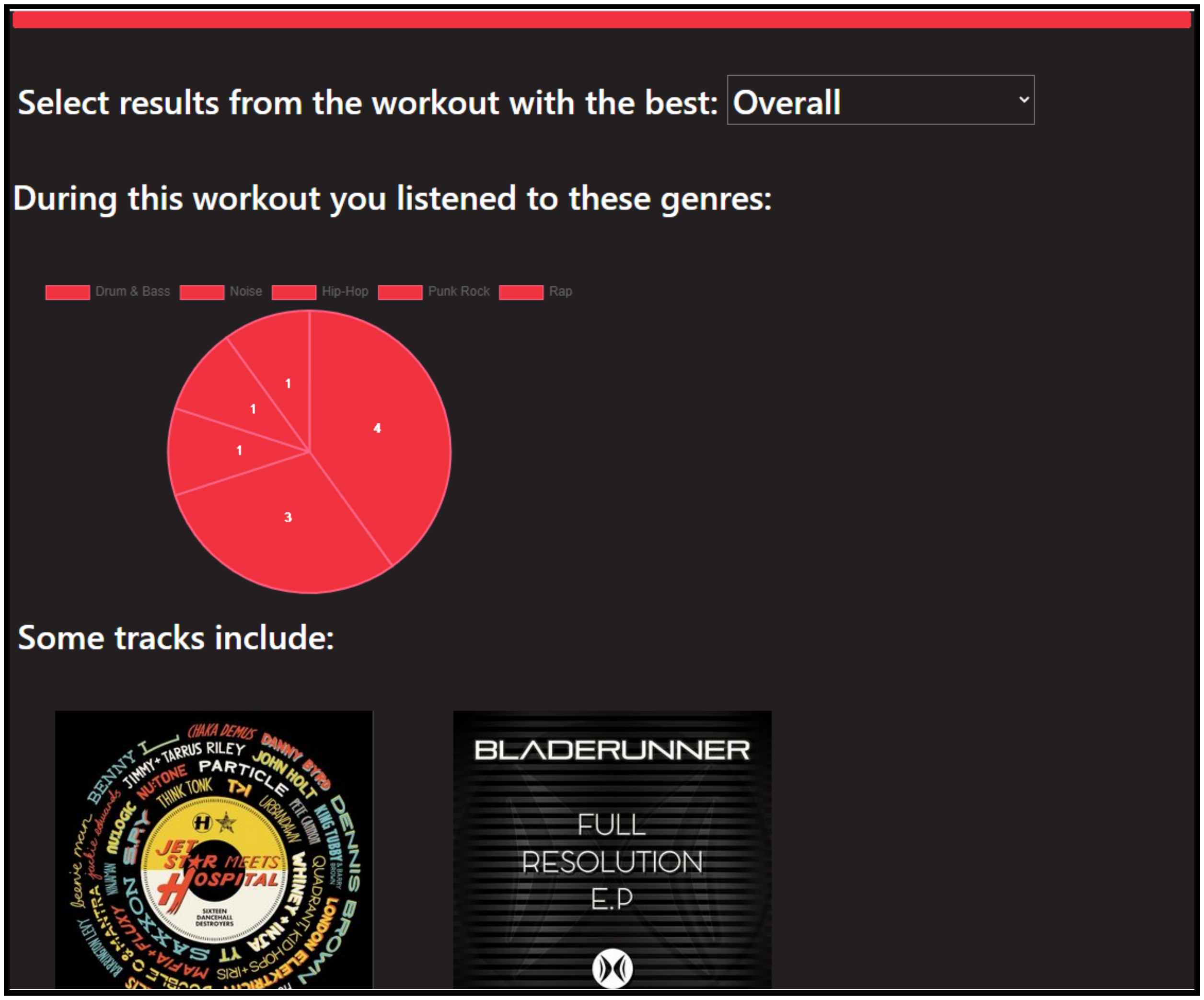
Figure 59: Test Case 5 - incorrect username is entered


Select results from the workout with the best: Overall

During this workout you listened to these genres:

Some tracks include:

Figure 60: Test Case 6 - Generations results after valid but incorrect Last.fm username used



Select results from the workout with the best: Overall

During this workout you listened to these genres:

Some tracks include:

Figure 61: Test Case 10

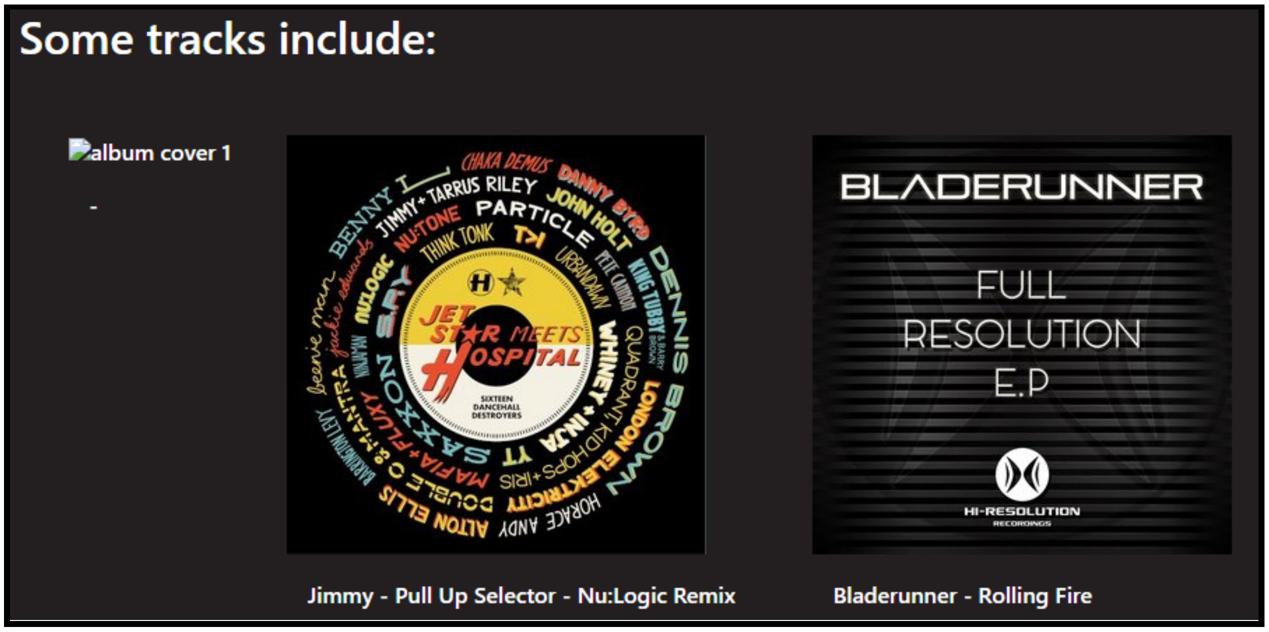

Some tracks include:
album cover 1
-
Jimmy - Pull Up Selector - Nu:Logic Remix
Bladerunner - Rolling Fire

Figure 62: Test Case 11 - song results when listening to music

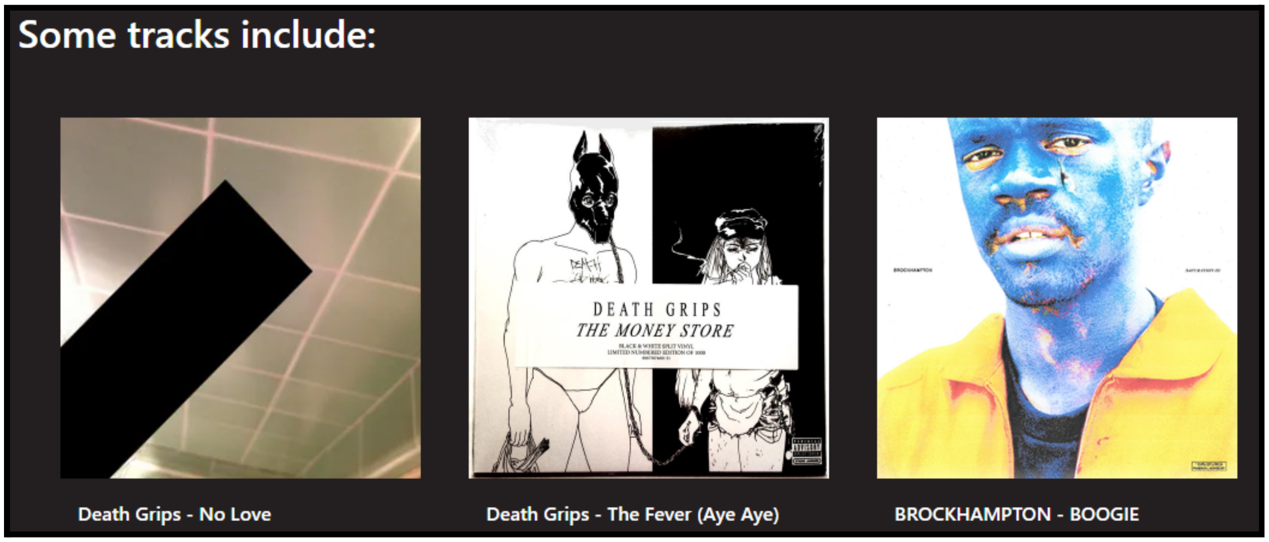


Figure 63: Test Case 12

References

Amazon Web Services. 2021 *Home* [online] Available at: https://aws.amazon.com/ [Accessed April 2021]

Bergen, J. and Andrew, R. 2019 *What is Mixed Content* [online] Available at: https://web.dev/what-is-mixed-content/ [Accessed 3 May 2021]

Bootstrap. 2021 *Getting Started - Introduction* [online] Available at: https://getbootstrap.com/docs/4.4/getting-started/introduction/ [Accessed 12 April 2021]

Chart.js. 2021 *Doughnut and Pie Charts* [online] Available at: https://www.chartjs.org/docs/3.0.2/charts/doughnut.html [Accessed 5 March 2021]

Chart.js. 2021 *Getting Started* [online] Available at: https://www.chartjs.org/docs/latest/ [Accessed 5 March 2021]

Daucez, D. et al. 2016 *Section 2.1.1: On-Path vs. Off-Path Attackers* [online] Available at: https://www.ietf.org/rfc/rfc7835.html#section-2.1.1 [Accessed 3 May 2021]

Farber, M. 2017 *Destroy Chart.js Bar Graph to Redraw Other Graph in Same Canvas* [online] Available at: https://stackoverflow.com/questions/40056555/destroy-chart-js-bar-graph-to-redraw-other-graph-in-same-canvas [Accessed 12 April 2021]

Garrison, B. 2017 *JavaScript ES8 - Introducing ‘async/await’ Functions* [online] Available at: https://medium.com/@_bengarrison/javascript-es8-introducing-async-await-functions-7a471ec7de8a [Accessed 21 April 2021]

Google Maps Platform. 2021 *Interactive Polyline Encoder Utility* [online] Available at: https://developers.google.com/maps/documentation/utilities/polylineutility [Accessed 3 May 2021]

JSON. 2021 *Introducing JSON* [online] Available at: https://www.json.org/json-en.html [Accessed 20 April 2021]

Last.fm. 2021 *About* [online] Available at: https://www.last.fm/about [Accessed 16 April 2021]

LastWave. 2021 *Home* [online] Available at: https://savas.ca/lastwave/#/ [Accessed 20 April 2021]

Lee, W. 2016 *How to convert Seconds to Minutes and Hours in JavaScript* [online] Available at: https://stackoverflow.com/questions/37096367/how-to-convert-seconds-to-minutes-and-hours-in-javascript/49905383 [Accessed 16 April 2021]

MDN Web Docs. 2021 *Web APIs* [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API [Accessed 21 April 2021]

Node.js. 2021 *About Node.js* [online] Available at: https://nodejs.org/en/about/ [Accessed 3 May 2021]

Oseberg, F. 2020 *Async Await JavaScript Tutorial - How to Wait for a Function to Finish in JS* [online] Available at: https://www.freecodecamp.org/news/async-await-javascript-tutorial/ [Accessed 3 May 2021]

Perform. 2021 *Home* [online] Available at: https://perform.fm/home [Accessed 20 April 2021]

Polignano, F. 2020 *Code From Tutorial - Strava API* [online] Available at: https://github.com/franchyze923/Code_From_Tutorials/tree/master/Strava_Api [Accessed 14 Feb 2021]

Postman. 2021 *Home* [online] Available at: https://www.postman.com/ [Accessed 20 April 2021]

Scatter.fm. 2021 *Home* [online] Available at: https://scatterfm.markhansen.co.nz/ [Accessed 21 April 2021]

Spotify. 2021 *Privacy Settings* [online] Available at: https://www.spotify.com/ca-en/account/privacy/ [Accessed 20 April. 2021]

Strava. 2021 *Features* [online] Available at: https://www.strava.com/features [Accessed 16 April. 2021]

Strava Developers. 2020 *Getting Started with the Strava API* [online] Available at: https://developers.strava.com/docs/getting-started/ [Accessed 7 Feb 2021]

TheAudioDB. 2021 *Home* [online] Available at: https://www.theaudiodb.com/ [Accessed 20 April 2021]