# Set.lystr

Antero Duarte
40211946@live.napier.ac.uk
Edinburgh Napier University - Advanced Web Technologies (SET09183)

## Abstract

Set.lystr is a music catalogue exploration/playlist creation tool geared towards musicians who play cover songs. Set.lystr allows its users to search and compare songs and build the ultimate playlist.

**Keywords** – Advanced, Web, Technologies, Python, Flask, MusicBrainz, AcousticBrainz

## 1 Introduction

**This** web application aims to be more than a music catalogue website. By providing users with acoustic metrics, such as BPM (beats per minute), a song's musical key and others, a user can meticulously plan their their setlist/playlist for the best flow.

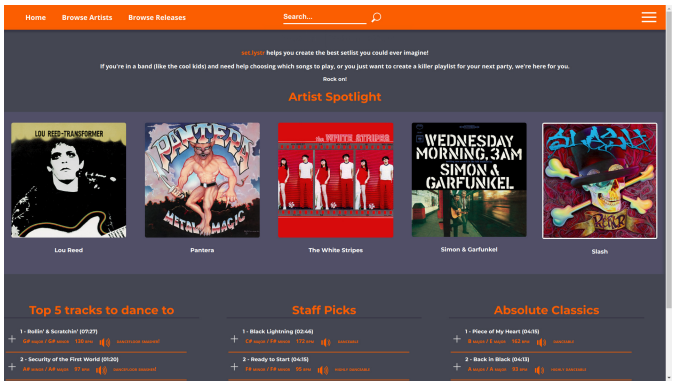


Figure 1: **Set.lystr** - Home page

Apart from said metrics, the website also provides music exploration capabilities, by listing every release created by a selection of well known artists, and every track in each release. If a user is not sure of what they are looking for, they can browse artists and releases. If a user knows exactly what they want, they can search and get a list of artists, releases and tracks that match their search terms.

**Set.lystr** uses MusicBrainz [**?**] as a data source, and links to acoustic data from AcousticBrainz [**?**]. Once a user has found a song they would like to include in their playlist, they can add it to their personal setlist, which builds up while informing the user of the selected songs metrics.

## 2 Design

This web app was designed in line with modern standards and best practices. All the data gathered is available in the public domain, and the only library used was iconocss [**?**], which is a pure css icon library.

### 2.1 Data gathering

**Data** was gathered by collating a list of artist names from a list of the top 100 pop/rock bands on IMDb [**?**] and my personal spotify playlists. The total number of artists after merging the lists, removing duplicates, removing artist names such as "Various artists" and removing guilty pleasures was 192. Using Node.js[**?**], I "scraped" the MusicBrainz API to lookup artists based on their name from the list, as seen on listing **??** in the appendices. After this I downloaded every release (listing **??** in the appendices), every track (listing **??** in the appendices) and matched each track to their acousticbrainz match, if there was one (listing **??**, **??**, **??** in the appendices). This resulted in the following hierarchy for the data:

- Artists
  - Releases
    - Tracks
    - Track Acoustics

### 2.2 Application planning

Once all the data was gathered, I planned what the application was going to be like. I decided which pages I wanted to have:

- Index page
- Artist Browse page
- Release Browse page
- Search results page

#### 2.2.1 Templating

I decided to create a base template, which would be extended by every other page. Having learnt and worked with frontend javascript libraries that use the concept of self contained components that are placed on containers[**?**], I tried to follow the same mindset for this app. Rather than having frontend defined components, I applied the Single File component architecture, as seen on Vue.js [**?**] to the server side. To achieve this, I created *content*, *extra_styles* and *extra_scripts* blocks on my

base.html file, which allowed me to generate page specific javascript and CSS that gets injected at runtime. This makes each template file self contained and means that the code is close to where it is used, rather than having monolithic javascript and css files that turn into spaghetti really quickly [**?**]. For instances when the code can be reused, it is moved to the monolithic files, because DRY(Don't Repeat Yourself [**?**]) can sometimes trump keeping the code close to where it is used, because it introduces a single point of maintenance. This might introduce a runtime penalty, since the template which includes the styles/script has to be compiled, but it is negligible, especially considering that there is nothing complex about rendering what are to the server just plain strings.

### 2.2.2 URL Structure

The URL structure for this kind of application is well documented and has been implemented widely. Moesif[**?**] provides a good guide for some of the common problems with API/URL design. The following figure shows the URL mapping in the web application.

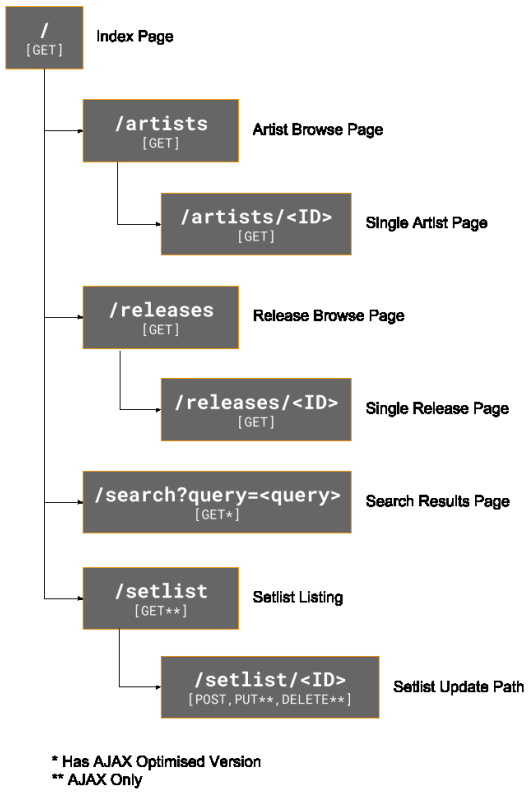


Figure 2: **URL map**

One of the rules of URL building is to give preference to what are called static URLs instead of dynamic URLs. This does not mean that the URL points to a static page, but rather that it does not use url query parameters as page identifiers. Example:
`http://example.org/pages/1`
Instead of
`http://example.org/pages?ID=1`
This was popular with Content Management Systems, because it makes the code simpler, but since the wide adoption of url rewriting, the use of dynamic URLs has become obsolete. An exception is the search page, which uses the *query* URL parameter, for one main reason: URL parameters handle spaces and special characters better than URL fragments.

### 2.2.3 Navigation map

As for navigation, the application presents the map in figure **??**.

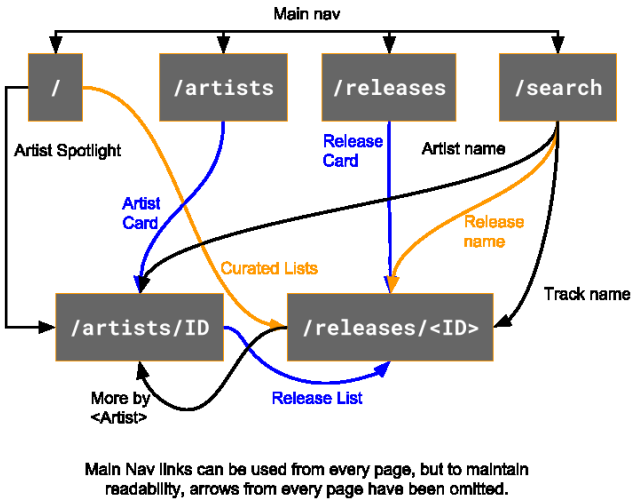


Figure 3: **Navigation map**

Due to the links to the main pages in the main nav, all the main pages are accessible from other pages. Pages dedicated to a single artist/release are accessible from listings that are displayed in several pages. For example, an artist's main page can be accessed from the index if the artist is in the artist spotlight, from the search autocomplete or the search results, from the artists' browse page, of from an artist's release page (through the *see more by artist* button). The only element in the hierarchy that does not present its own page is a single track. This is due to the context of the album being lost, but also because all the meaningful elements of a track can be displayed in the release page.

**One** extra that is not present in the maps is that when a release page is accessed by clicking a track name, said track is highlighted in the release page. This happens because clicking on a track does not make it obvious that the user will navigate to the release page.

# 3 Enhancements

"It's not a bug, it's a feature"

There are several things I would have done if I had more time to develop this project. A lot of the data that I gathered, I ended up not using, or at least in the way I intended to use it.

### 3.1 Track similarity

An example of this can be seen in a comment in the model.py file:

```
1 # Good idea, but won't actually use this, maybe for a ↩
      future project
2 def track_distance(a, b):
3   # ...
4   return n_dim_euclidean(track_a, track_b)
```

Listing 1: Track distance function definition

This function was meant to return the euclidean distance between 2 tracks that are placed on an 8th dimensional plane. It's a very rough attempt at creating a track similarity function. Had I had more time, I would have implemented this function in suggesting tracks to add to the setlist, based on the tracks that a user added previously.

### 3.2 Setlist flow

An idea I had since the start was to measure a setlist's flow.

**By flow** here, I mean how one song progresses into the next one. Key changes, beats per minute, loudness and other acoustic metrics that I have access to in the data could be used to calculate how easily one can move from one song to the other. Other data could also be gathered to try to make the flow calculations better.

**For example,** songs in different tunings sometimes mean that artists have to switch instruments or tune them differently, which breaks the flow in a live concert scenario. Songs with different instruments than the previous one might mean that a band member has to switch instruments or come on/off stage. Songs with a low dynamic complexity (low variation between calm/agitated periods) and high bpm, or high danceability and bpm, might make the band tired after playing a lot of them in a row, which means they would need to rest between songs.

**These and other** conditions could be made into a program that tries to leverage the elements of different songs to keep a crowd's engagement.

### 3.3 Data collection

Finally, a feature that could be extremely useful in growing an app like this one would be to collect data about how people are using the website. This could become a big advantage when developing new functionality, and could also enable machine learning based features, if said data was used to train models.

### 3.4 Dedicated setlist page

When creating this application, the initial idea involved a page dedicated to managing the songs in the current setlist. This idea was dropped when due to time constraints I realised that I would not be able to create the necessary mechanisms to have enough functionality that makes it worth having such a page.

**The setlist page** would include the previously mentioned track suggestion feature, a track reordering (using drag and drop) and overall setlist metrics, such as total running time, and possibly even a dashboard type interface that would show the characteristics of the current playlist.

### 3.5 Dedicated track page

Similarly to the setlist page, the amount of effort needed to create a track page, when compared to the functionality that would be in that page, just did not justify its creation.

### 3.6 Direct connection to data APIs

All of the data collection process could have been avoided if a direct connection was made to the APIs offered by both data sources. This would have opened up a lot of possibilities, considering that the artist catalogue is quite limited, but on the other hand, it would have made the range of different outcomes too unwieldy, which would have slowed development because the code would have to accommodate for gaps in the data that were filled during the data gathering stage.

## 4 Crtitical evaluation

While I'm happy with the current state of Set.lystr, several things about it make me want to spend more time on it, even if just for fun/portfolio building.

### 4.1 Architecture

One of the things I enjoy the most about developing projects such as this one is to look at the holistic system architecture. While being a relatively simple project that is composed of a single codebase, it's architecture is also rather simplistic. While there are things about this project that might not pass the scalability test, for example not splitting different routes into different files, I personally consider this a good architecture when considering the broader context. This application and its development have a scope, and one of the best skills when thinking about the architecture of an application is to take into account the balance of time vs features.

**I could** have made this application more scalable. I could have decoupled the frontend from the backend, which would mean less work on the server side. I could have deployed n instances of the application and load balance them. But calling that overkill would be an understatement.

**That being said,** I am particularly happy about features such as the streaming search results function, which is backed by a generator function that yields results as they become available. This improves the perceived loading times, and when matched with client-side functionality that sorts the results as they come, creates a good User Experience.

**Another example** that I am particularly happy with is the balance between AJAX functionality for improved user experience and the fall-backs that are in place for the case where a user is not running javascript. While it could be argued that users whose browsers don't support javascript would not benefit from the modern CSS in the page, having these fall-back capabilities means that even with a broken layout and a few ugly pages (I'm looking at you, default Flask redirect-after-form-submit page), a user is fully capable of using the system. This is also true for screen readers, which makes the web application more accessible.

### 4.2 Responsiveness

While responsive to a point, this web application still has some pages that are not optimised for mobile view. One of the worse elements being the setlist, which should be made to take the whole screen when on mobile views.

**This** not only makes the website less appealing, but also harder to interact with when on mobile devices. This becomes a major problem, when considering that upwards of 60% of the web's traffic according to statistics from April 2018, are from mobile devices[**?**].

#### 4.2.1 Sub par search results

When searching for a set of keywords, a user can clearly see that the search results are just not up to the standard that a user has come to expect when using a website. And while this is partly due to users being "spoiled" by great user experiences, such as the ones that big companies/products provide, the truth is that the technology for achieving such functionality is more accessible than ever. Besides dedicated search engines that index all pages in a website, there are libraries that could be used to improve string comparison, which in itself would have made search results better.

**While** string comparison using the naive python builtin *SequenceMatcher* stems from a limitation imposed as part of the coursework, of not being able to install additional libraries, if I had put some more effort into the search functionality, better results would have been possible. For example, for the cases where an artist has a self-titled release (which is very common, e.g: Pearl Jam by Pearl Jam), due to the text index being just a dictionary of *string* to *string*, whatever comes first is overridden by what comes after, which means searching for Pearl Jam would never return the artist, when more often than not that is the desired outcome.

**This could be fixed** quite easily by just making the text index a dictionary of *string* to *list of string*, but I decided that this would come on a following iteration of the development, due to time constraints.

## 5 Conclusion

## 6 Appendices

### 6.1 Data gathering code

```
const mb = require('musicbrainz')
async function matchToMusicBrainz() {
    let done = 0
    // bands => [String] => all the band names gathered
    const promises = bands.map(async (b) => {
        try {
            const [first] = await searchArtists(b)
            done += 1
            console.error(`${done}/${bands.length}`)
            return {
                id: first.id,
                name: first.name,
                country: first.country,
                lifespan: [first.lifeSpan.begin, first.lifeSpan.end]
            }
        } catch (e) {
            console.error(e)
        }

    })
    const results = await Promise.all(promises)
    results.forEach(r =>{
        console.log(`"${r.id}","${r.name}","${r.country}","${r.lifespan[0]} ${r.lifespan[1]}"`)
    })
}
// Immediately Invoked Function Expression, because async/await can't be used at top level
(async ()=>{
 await matchToMusicBrainz()
})()

function searchArtists(query, filter, force) {
    return new Promise((resolve, reject) => {
        mb.searchArtists(query, filter, force, (err, data) => {
            if (err) return reject(err)
            return resolve(data)
        })
    })
}
```

Listing 2: Artist name to MusicBrainzID lookup

```
const mb = require('musicbrainz')
const fcsv = require('fast-csv')
const axios = require('axios')

async function getMusicBrainzAlbums() {
  let done = 0;
  fcsv.fromPath('./csv/musicbrainzID.csv', {
    headers: ['id', 'name', 'country', 'lifespan']
  }).on('data', async ({
    id
  }) => {
    const artist = await lookupArtist(id, ['release-groups'])
    const albums = artist.releaseGroups.filter(r => r.type == 'Album')
    const promises = albums.map(async (a) => {
      let mainrelease,front,back,stage;
      try {
        stage = 'relgroup'
        const {
          data
        } = await axios.get(`https://coverartarchive.org/release-group/${a.id}/`);
        stage = 'coverart-start';
        mainrelease = (data.release || '').split('/').pop();
        stage = 'coverart-rel';
        front = data.images.find(i => i.front).image
        stage = 'coverart-front';
        back = data.images.find(i => i.back).image
        stage = 'coverart-end';
      } catch (e) {
        console.error(`failed: ${artist.name} - ${a.title} (${stage}) => ${a.id}`)
      } finally {
        switch(stage){
          case 'coverart-end':
```

```
            // artist id, relgroup id, relgroup name, first release date, main release, front cover, back cover
            console.log(`"${id}","${a.id}","${a.title}","${a.firstReleaseDate || ''}","${mainrelease}","${front || ''}","${back || ''}"`)
            break
          case 'coverart-front':
            // artist id, relgroup id, relgroup name, first release date, main release, front cover
            console.log(`"${id}","${a.id}","${a.title}","${a.firstReleaseDate || ''}","${mainrelease}","${front || ''}",""`)
            break
          case 'coverart-rel':
            // artist id, relgroup id, relgroup name, first release date, main release
            console.log(`"${id}","${a.id}","${a.title}","${a.firstReleaseDate || ''}","${mainrelease}","",""`)
            break
          default:
            // artist id, relgroup id, relgroup name, first release date
            console.log(`"${id}","${a.id}","${a.title}","${a.firstReleaseDate || ''}","","",""`)
        }
        done += 1
        console.error(`${done}/192`)
        return Promise.resolve()
      }
    })
    return Promise.all(promises)
  })
}

function lookupArtist(id, links) {
  return new Promise((resolve, reject) => {
    mb.lookupArtist(id, links, (err, data) => {
      if (err) return reject(err)
      return resolve(data)
    })
  })
}
```

Listing 3: Release fetching from Musicbrainz. Includes cover art url fetching

This step gathered errors in lookups and classified them at different stages:

1. coverart-end - has all data
2. coverart-front - has main release and front cover
3. coverart-rel - has main release
4. relgroup - has relgroup only

For the relgroup stage, a lot of the releases are just bootlegs, or live albums, so they can safely be ignored For the coverart-rel stage, a lot of them do have cover images, they're just not in the expected format. I picked 2/3 of the more important ones and manually collected that data. For the coverart-front stage, we're only missing a back cover, which is not that important For the coverart-end stage, we have all data, which is ideal.

Before removing the relgroups, had 2065, after removing 206 relgroups, ended up with 1859

The release lookup listed all songs in a record. At the end of the lookup, 18 did not match, so a second pass was made to download the rest. Code:

```
const fcsv = require('fast-csv')
const mb = require('musicbrainz')

async function getMusicBrainzSongs() {
  return new Promise((resolve, reject) => {
    let done = 0;
    fcsv.fromPath('./csv/albums-full.csv', {
      headers: ["artist", "relgroup", "relgroup-name", "reldate", "mainrel", "frontcover", "backcover"]
    }).on('data', async ({
      mainrel
    }) => {
      const release = await lookupRelease(mainrel, ['recordings'])
      release.mediums.forEach(m => {
        const format = m.format && m.format['#'] ? m.format['#'] : m.position;
        m.tracks.forEach(t => {
          try {
            // release id, medium, track id, track name, position, length
            console.log(`"${mainrel}","${format.replace(/"/g,"'")}","${t.recording.id}","${t.recording.title}","${t.position}","${t.length}"`)
          } catch (e) {
            console.error(`failed: ${t.recording.id} => ${e.message}`)
          }
        });
      });
      done += 1;
      console.error(`${done}/1858`)
    }).on('end', () => {
      resolve(true)
    })
  })
}


(async () => {
  await getMusicBrainzSongs()
})()

function lookupRelease(id, links) {
  return new Promise((resolve, reject) => {
    mb.lookupRelease(id, links, (err, data) => {
      if (err) return reject(err)
      return resolve(data)
    })
  })
}
```

Listing 4: Track fetching from musicbrainz

After this, the songs were matched to records in acousticbrainz, a website which holds acoustic data about these songs. The following fields were recorded: [beats per minute, average loudness, chord change rate, chord key, chord scale, song key, song scale, key strength] Using the following code:

```
const fcsv = require('fast-csv')
const axios = require('axios')

async function getAcousticBrainz() {
  let done = 0;
  const promises = [];
  let row = 0;
  fcsv.fromPath('./csv/tracks.csv', {
    headers: ["release", "medium", "track", "name", "position", "length"]
  }).on('data', async ({
    track
  }) => {
    const elem = (async (currRow) => {
      return new Promise(async (resolve, reject) => {
        try {
          await new Promise((r) => {
            setTimeout(r(), (row % 10) * 100)
          })
          const {
            data
          } = await axios.get(`https://acousticbrainz.org/api/v1/${track}/low-level`)
          const bpm = (data && data.rhythm && data.rhythm.bpm) || "0"
          const loud = (data && data.lowlevel && data.lowlevel.average_loudness) || "0"
          const chordchange = (data && data.tonal && data.tonal.chords_changes_rate) || "0"
          const chordkey = (data && data.tonal && data.tonal.chords_key) || ""
          const chordscale = (data && data.tonal && data.tonal.chords_scale) || ""
          const keykey = (data && data.tonal && data.tonal.key_key) || ""
          const keyscale = (data && data.tonal && data.tonal.key_scale) || ""
          const keystr = (data && data.tonal && data.tonal.key_strength) || "0"
          console.log(`"${track}","${bpm}","${loud}","${chordkey} ${chordscale}","${chordchange}","${keykey} ${keyscale}","${keystr}"`)
          done += 1
```

```
          console.error(`${done}/23519`)
          resolve(true)
        } catch (e) {
          console.error(`failed (${row}): ${track} => ${e.message}`)
        }
      })
    })(row)
    promises.push(elem)
    row++
  }).on('end', () => {
    return Promise.all(promises)
  })
}


(async () => {
  await getAcousticBrainz()
})()
```

Listing 5: Matching individual tracks in musicbrainz with entries in acousticbrainz

Out of the original 23519 songs gathered from the previous step, 3587 were not found in acousticbrainz, because acousticbrainz does not have an entry for 100% of the songs in musicbrainz. After a while, because the volume of calls was so big, acousticbrainz started responding with 50x errors, but by recording these errors and doing a 2nd and a 3rd pass, I managed to download the rest of the songs, with the total count being 19932.

Way into the server building, I realised I forgot to download a field that I wanted to have, danceability. So I partially re-wrote the code above to download the whole json file for the song's acoustics, so I could just go through the files if I realised I forgot something else.

```
const fcsv = require('fast-csv')
const axios = require('axios')

async function downloadAcousticBrainz() {
  let done = 0;
  const promises = [];
  let row = 0;
  fcsv.fromPath('./csv/fullbrainz.csv', {
    headers: ["release", "medium", "track", "name", "position", "length"]
  }).on('data', async ({ track }) => {
    const elem = (async (currRow) => {
      return new Promise(async (resolve, reject) => {
        try {
          const exists = await fileExists(`./acoustics/${track}.json`)

          if (!exists) {
            await new Promise((r) => {
              setTimeout(r(), (currRow % 10) * 100)
            })
            const {
              data
            } = await axios.get(`https://acousticbrainz.org/api/v1/${track}/low-level`)
            fs.writeFile(`./acoustics/${track}.json`, JSON.stringify(data),(err)=>{
              if(!err){
                done+= 1
                console.log(`${done}/19933`)
                resolve(true);
              }else{
                throw err
                resolve(true)
              }
            });
          }else{
            done+=1
            console.log(`skipping ${track}, done: ${done}/19933`)
          }
        } catch (e) {
          console.error(`failed (${currRow}): ${track} => ${e.message}`)
        }
      })
    })(row)
    promises.push(elem)
    row++
  }).on('end', async () => {
    promises.push(new Promise((r) => {
      setTimeout(() =>{
        console.log("waiting 60 secs for any unresolved promise to resolve")
        r();
      },  60000)
    }))
    return Promise.all(promises)
  })
}


(async () => {
  await downloadAcousticBrainz()
})()
```

Listing 6: Downloading the whole json file for each acousticbrainz entry

This code also allows for multiple passes to be executed without messing with logs, which means I could run this in an infinite loop, where I waited about 10 minutes to let the acousticbrainz servers come back after getting 50x errors.

Also while doing this, I realised that even after a few passes, I was only getting maximum 19048 files, which lead me to investigate. I realised that there are duplicate entries in the tracks.csv file, which made it seem like there are more songs than the 19048

After that I created a function that would go through all downloaded files and actually gather the metrics I was looking for:

```
const fcsv = require('fast-csv');
const axios = require('axios');
const { promisify } = require('util');
const fs = require('fs');
const fileExists = promisify(fs.exists);
const readdir = promisify(fs.readdir);

const readFile = promisify(fs.readFile)
async function parseAcousticBrainz() {
  const files = await readdir("./acoustics");
  // print the headers
  console.log('"id","bpm","loudness","chord_key","chord_change_rate","song_key","key_strength","danceability","dynamic_complexity"')
  files.forEach(async name => {
    try {
      const file = await readFile('./acoustics/' + name)
      // Remove the .json bit from the file name to get the ID back
      const id = name.substring(0, name.length - 5)
      const data = JSON.parse(file);
      const bpm = (data && data.rhythm && data.rhythm.bpm) || "0"
      const danceability = (data && data.rhythm && data.rhythm.danceability) || "0"
      const loud = (data && data.lowlevel && data.lowlevel.average_loudness) || "0"
      const dc = (data && data.lowlevel && data.lowlevel.dynamic_complexity) || "0"
      const chordchange = (data && data.tonal && data.tonal.chords_changes_rate) || "0"
      const chordkey = (data && data.tonal && data.tonal.chords_key) || ""
      const chordscale = (data && data.tonal && data.tonal.chords_scale) || ""
      const keykey = (data && data.tonal && data.tonal.key_key) || ""
      const keyscale = (data && data.tonal && data.tonal.key_scale) || ""
      const keystr = (data && data.tonal && data.tonal.key_strength) || "0"
      console.log(`"${id}","${bpm}","${loud}","${chordkey} ${chordscale}","${chordchange}","${keykey} ${keyscale}","${keystr}","${danceability}","${dc}"`)
    } catch (e) {
      console.error(`Failed: ${name} => ${e.message}`)
    }
  })
}


(async () => {
  await parseAcousticBrainz()
})()
```

Listing 7: Parsing the acousticbrainz files to turn the relevant metrics into csv format