



Final Report

Clef

CPSC 2350 Project

Instructor:

Parsa Rajabi

Team (*Please remember that we are a group of 3*):

Darrick He

Chao Xu

Arshpreet Kaur

Github Repository: <https://github.com/DivoHub/Clef.git>

Table of Contents

Project Overview / SDLC	3
High-level features.....	4
API Features.....	5
Testing.....	7
CI/CD infrastructure.....	9
High-level Data Flow Diagram.....	12
Lessons Learnt during project.....	13

Overview of the project

With a growing user-base and a variety of music streaming services, users can often switch companies and memberships without any transitional support. Members who spend the time to curate their playlists can find themselves trapped from moving between different services.

Clef is a web application that helps users automate that process. Clef utilizes the Deezer Public API and the Spotify API to pull data from the user's playlists, and then clones them onto their target service.

Overview of the SDLC model

The SDLC model that we chose for this project was Agile Kanban (Github Projects).

Agile Kanban was originally chosen due to suitability for small teams, its ubiquity within the software industry, as well as its flexibility for updates/changes to the software product. This later proved to be an invaluable aspect of the SDLC due to the number of hurdles and changes faced throughout the implementation process.

What Worked: Agile Kanban provided a medium for us to track progress visually, and allowed us to distribute different tasks effectively. Using Kanban accommodated an asynchronous workflow and remote collaboration without compromising on efficiency.

Clef MVP only supports two services (Deezer, Spotify) out of the several that are available on the market (Tidal, Amazon Music, Apple Music, Youtube). Further implementation is needed in order to support a wider range of music services and user base. Consequently, the UI and Design will also need to be updated accordingly. Choosing an Agile framework was crucial in sustaining a long-term, constantly-changing application.

What Did Not Work: Our issues stemmed from poor execution/inexperience of the Agile Kanban framework rather than the issue being the Agile Kanban framework itself. It was difficult to determine what tasks deserved their own card, as well as how to minimize overlap. Organizing branches, cards, and features had a learning curve of its own.

High-level features

1. Clef fetches playlist data from Spotify after being granted an access token from user authorization redirect. All user's playlists' names are displayed on UI
2. After a Spotify playlist is selected, another GET request is made to fetch song names and artists from the selected playlist.
3. Deezer is iteratively queried with each song name and artist name, which returns a list of top selections for each query. The Song IDs are then pushed to the Playlist ID of the corresponding playlist the user has selected.
4. Playlist data is pulled from User's Deezer account, and then displayed onto the web page UI.
5. After a Deezer playlist is selected, a GET request fetches song names and artists from the selected playlist.
6. Spotify is iteratively queried with each song name and artist name from Deezer. The top result of each query is then pushed to the selected Spotify playlist.

API Features

From Spotify API

- Get the authorization of usage of user's information when a user clicks the Spotify logo in the red rectangle, the browser will turn to the login page of the Spotify server and get the authorization of usage of user's information through the Spotify API.

Easy way to transfer your playlists between different music services


Please choose the source music service:



- Retrieve playlists from user's data when Clef application gets the playlists from user's data on the deezer server, these playlists will display on either source area or target area.

Clef

Easy way to transfer your playlists between different music services

Playlist(s) from source service: 

playlistS1 ☐

playlistS2 ☐

playlistS3 ☐

playlistS4 ☐


Option:

Convert chosen playlist(s) ☐

Convert all playlist(s) ☐

Start Convert

Log out

Playlist(s) from target service: 

playlistT1 ☐

playlistT2 ☐

playlistT3 ☐

playlistT4 ☐

- Fetch songs from playlist in spotify
- Add songs to the spotify playlist

From Deezer API

- Get the authorization of usage of user's information when a user clicks the deezer logo in the red rectangle, the browser will turn to the login page of the deezer server and get the authorization of usage of user's information through the deezer API.

Easy way to transfer your playlists between different music services



Please choose the source music service:



- Retrieve playlists from user's data when Clef applications get the playlists from user's data on the deezer server, these playlists will display on either source area or target area.

Clef

Easy way to transfer your playlists between different music services

Playlist(s) from source service: 	Option:	Playlist(s) from target service: 
<div><div>playlistS1 <input type="checkbox"/></div><div>playlistS2 <input type="checkbox"/></div><div>playlistS3 <input type="checkbox"/></div><div>playlistS4 <input type="checkbox"/></div></div>	<div><div>Convert chosen playlist(s) <input type="radio"/></div><div>Convert all playlist(s) <input type="radio"/></div></div>	<div><div>playlistT1 <input type="checkbox"/></div><div>playlistT2 <input type="checkbox"/></div><div>playlistT3 <input type="checkbox"/></div><div>playlistT4 <input type="checkbox"/></div></div>
	<div><div>Start Convert</div><div>Log out</div></div>	

- Fetch songs from playlist in deezer
- Add songs to the deezer playlist

Testing

Unit Tests

1. Test the connection between Spotify logo and login page on Spotify Server
This test will test whether the image hyperlink will lead the page to the Spotify login page for third parties usage correctly.
2. Test the connection between Deezer logo and login page on Deezer Server
This test will test whether the image hyperlink will lead the page to the Deezer login page for third parties usage correctly.
3. Test web component change between source chosen and target chosen
This test will test when a user logs in the source service, whether the page component changes to the target choose component correctly.
4. Test web component change between target chosen and playlists component
This test will test when a user logs in both source and target service, whether the page component changes to the playlists component correctly.
5. Test the display of playlists area with checkbox
This test will test whether the playlists area can display playlists with checkboxes.
6. Since many songs and artists' names contain accented letters which might sabotage the search query, a sanitizeName() function is tested to validate the output after an input with special characters is passed through.
7. The song data is additionally passed through regex parser in a serializeSpotifySearchQuery() function to remove any parentheses, and then formats a string containing the song name and artist name before it is passed through to the search query.
8. Ensure each URL subdirectory for each service is an active endpoint by pinging and looking for the response ("pong")

Integration Tests

```
describe('/api/deezer/me', () => {
  it('has route handler listening for get requests on /api/deezer/me', async () => {
    const response = await request(app).get('/api/deezer/me').send();
    expect(response.status).not.toEqual(404);
  });
});

describe('/api/spotify/me', () => {
  it('has route handler listening for get requests on /api/spotify/me', async () => {
    const response = await request(app).get('/api/deezer/me').send();
    expect(response.status).not.toEqual(404);
  });
});
```

1. Setup route handler to listen for requests at
“http:localhost:5000/api/deezer/me”
In the local environment, or
“https://clefproject.herokuapp.com/api/deezer/me”
Once deployed.

Sends request to mentioned URL to ensure that the returning code is not a 404
(Page not found) and is up and running

2. Setup route handler to listen for requests at
“http:localhost:5000/api/spotify/me”
In the local environment, or
“https://clefproject.herokuapp.com/api/spotify/me”
Once deployed.

Sends request to mentioned URL to ensure that the returning code is not a 404
(Page not found) and is up and running

CI/CD infrastructure

Test

Unit and integration tests are done using the Javascript library 'Jest' before being committed and pushed onto respective branches.

```
$ npm test

> server@1.0.0 test
> jest --watchAll=false --no-cache --verbose

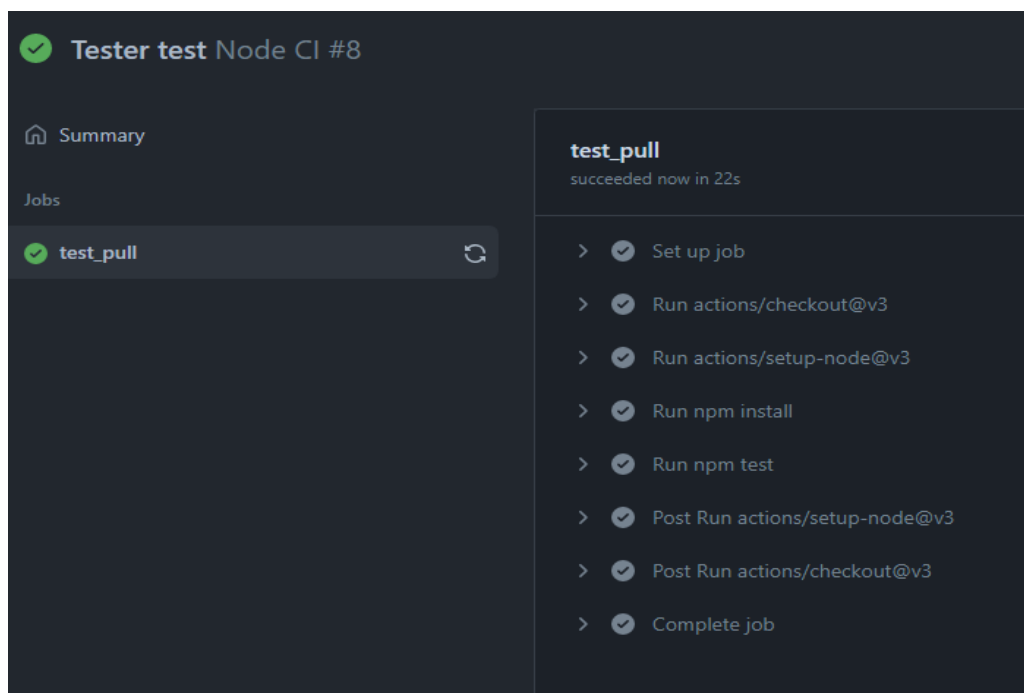
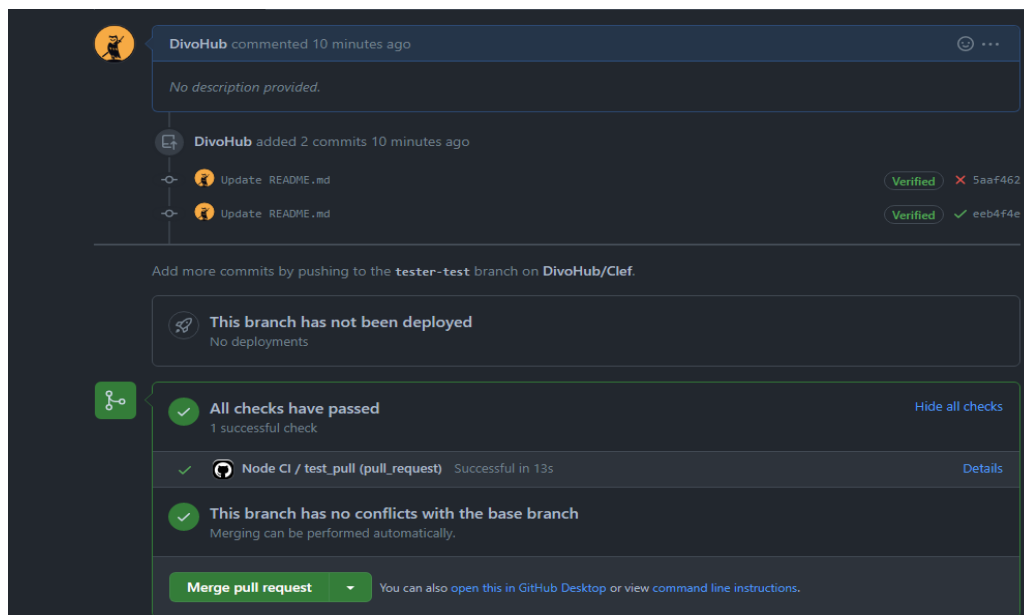
PASS src/test/unit.test.js
  sanitizeName
    ✓ should remove special characters out of the string (2 ms)
    ✓ should remove any parentheses or brackets out of the string
  serializeSpotifySearchQuery
    ✓ should parse the track and artist name in correctly formatted spotify query (1 ms)
    ✓ should remove any parentheses or brackets out of the artist and/or track name

PASS src/test/integration.test.js
  /api/ping
    ✓ should be an active endpoint (16 ms)
  /api/deezer/me
    ✓ has route handler listening for get requests on /api/deezer/me (363 ms)
  /api/spotify/me
    ✓ has route handler listening for get requests on /api/spotify/me (309 ms)

Test Suites: 2 passed, 2 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        2.212 s
Ran all test suites.
```

Build

Continuous Integration is automated using Github Actions. A `tester.yml` is activated once a pull request to the main branch has been detected. Github Actions then creates a dockerized container on the latest version of Ubuntu before installing Node.js dependencies for the client (React) directory and the server (Express) directory. “`$npm test`” is run to check unit/integration tests before finally building the application. Once all checks have been passed, the pull request can then be merged onto the main branch.



Deploy

Once pull requests have been tested for errors using Github Actions, they are then merged onto the main branch. Heroku checks the main branch for changes, and then redeploys the new version of the application onto a staging pipeline. The staging environment can then be promoted to production manually by the superuser.

App connected to GitHub

Code diffs, manual and auto deploys are available for this app.

Connected to DivoHub/Clef by DivoHub

Releases in the [activity feed](#) link to GitHub to view commit diffs

Automatically deploys from main

Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

You can now change your main deploy branch from "master" to "main" for both manual and auto follow the instructions [here](#).

☒ Automatic deploys from main are enabled

Every push to main will deploy a new version of this app. **Deploys happen automatically:** be sure that it's always in a deployable state and any tests have passed before you push. [Learn more](#).

☒ Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

[Disable Automatic Deploys](#)

Activity Feed

dhe06@mylangara.ca: Build in progress

Just now · [View build progress](#)

dhe06@mylangara.ca: Deployed 4d2f9281

Today at 4:31 AM · v5 · [Compare diff](#)

dhe06@mylangara.ca: Build succeeded

Today at 4:30 AM · [View build log](#)

dhe06@mylangara.ca: Deployed 2fc28607

Today at 4:21 AM · v4 · [Roll back to here](#) · [Compare diff](#)

dhe06@mylangara.ca: Build succeeded

Today at 4:20 AM · [View build log](#)

dhe06@mylangara.ca: Deployed 8f8af509

Today at 4:16 AM · v3 · [Roll back to here](#)

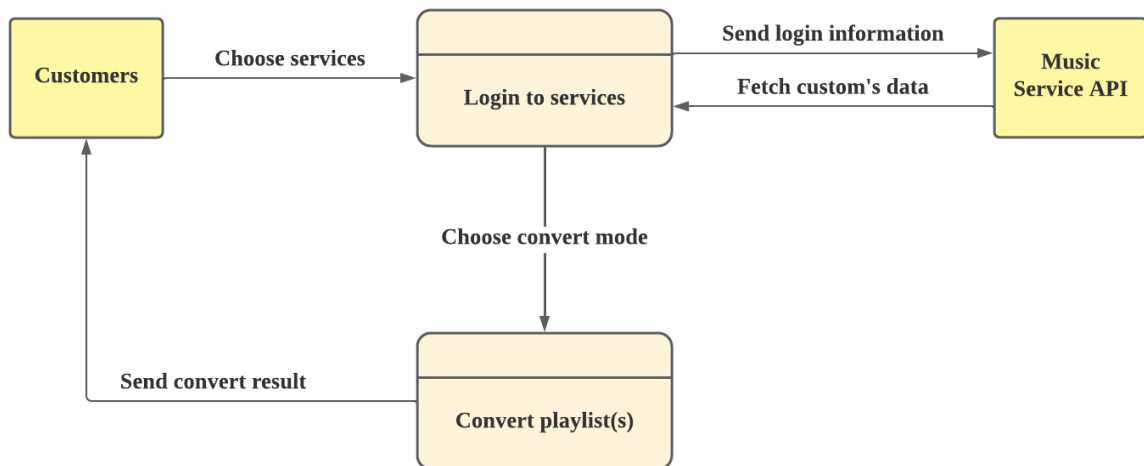
dhe06@mylangara.ca: Build succeeded

Today at 4:14 AM · [View build log](#)

dhe06@mylangara.ca: Build failed

Today at 3:40 AM · [View build log](#)

High-level data flow diagram



Lessons Learned

- Choose tech stack wisely: Unless you are willing to allot and dedicate time to team members to learn a new framework/language, do not implement something they do not already know. Learning something on the fly can also lead to poorly developed code.
- Automate whatever you can: Taking the time to correctly automate a repetitive process will pay dividends in the long run.
- Consider the layout of a web page more comprehensively: Spending more time thinking about how to design web layouts to be user-friendly. Thinking about information the users want to see, and what information does not need to be displayed on the web page for the user.
- Read the documentation of a module/library/API THOROUGHLY before implementing/declaring any features that use it: We knew Apple Music API existed - we knew there was good documentation for it - we knew it was used in many popular applications. We did, however, NOT know that it would cost \$110 a year to use. Thus, we had to scrap all of the resources we dedicated to the API, and we had to start from scratch.
- Do not trust your own/anyone else's memory: Comment/Document everything. Do not assume that you can go to sleep halfway through writing a function, and then come back thinking that you'll be in the same mindset as when you left it.
- Take one small step at a time: In other words, commit often, pull often, test often. Solving errors within 10 lines of committed code is a lot easier than a big commit of 100 lines of code. Solve one problem at a time. The scope of an issue is less daunting once you break it down into smaller pieces.
- Teamwork is mandatory/Do not try to be a hero: Collaboration is the cornerstone of a software project. Frequent back-and-forths between group members is necessary for everybody to understand the mechanics and inner-workings of the software. Undertaking one large task all by yourself is not only taxing for you, but also leaves many of your team members in the dark with how the implementation works.

- Communication is key: Make sure everyone is routinely updated on feature changes or problems. Make sure no one is working on the same thing, and make sure there are no conflicts between your code.
- Time is both the programmers worst enemy, and most valuable resource. Good time management is absolutely paramount for any software project. Everything a developer does takes time. If you want an API to be implemented, it takes time to read through and understand the documentation. If the API is poorly coded, then it takes time to find another one, or to code your own. If a new addition to the tech stack is introduced, it takes time for developers to learn how to implement it.

Project Challenges

- The project initiated with the notion to implement Apple Music API. After reading through the documentation and studying its usage, we found out that an Application Key is not provided unless we have an Apple Developer account, which costs \$110 per year to enroll in. Thus, we had to scrap everything we had done involving Apple Music in search of another music service.
- The Deezer API was very poorly documented, which stressed the importance of good documentation. Get requests and authorizations were relatively straightforward, but there were zero resources on how Post requests were handled - which were needed to implement search queries and to append playlists. There was virtually no support from Deezer, nor did we find any public help. The few videos on Youtube were not in English, and the only public API wrapper with post requests was implemented using Python. Thus, we spent several hours researching, and ended up coding our own.
- Severely underestimated the painstaking complexity of authorization workflows and access tokens from Spotify. 20% of all of the time spent coding the backend was dedicated to figuring out how to get the authorization to work with our application.
- Learning new technologies on the fly. One of us did not have previous React or Express experience. Upon learning, they found out that there are additional preliminary skills to learn to even begin getting started with React. Additional ES6 syntax and functions had to be learned on top of React and Express.

- Judging time constraints: Many features/implementations ended up taking a lot longer than originally thought to have. For instance, 2 hours was estimated for researching and implementing Authorization workflows, but the entire process ended up taking 8 hours.
- Difficulties combining Front-End with Back-End due to lack of experience: We thought that the Front-End and Back-End could be developed in parallel without much consideration of how the elements interact with each other. The intermediary interactions proved to be one of the most difficult parts of the development process. Many features had to be scrapped in the interest of time.

Work divided

Darrick: Back-end Express/Node.js

Chao: Front-end HTML/React

Arshpreet: Documentation/Interaction between front-end and back-end