Phase III Report

Github Repository: https://github.com/dylanmccoy/ee-461l-idb
Website: https://je66yliu.github.io/songtrackr/

Team Information

Team Members	Email	Github Username
Kevin Chen	kchen3568@gmail.com	kxc3568
Matthew Golla	matthew.golla8@gmail.com	matthewgolla
Jerry Liu	je66y.liu@gmail.com	je66yliu
Dylan McCoy	dylan.mccoy@utexas.edu	dylanmccoy
Diana Shao	dianashao01@gmail.com	dianashao
Ryuichi Yanagi	ryanagi@utexas.edu	ryuichiyanagi

Canvas Group: Morning-6
Project Name: SongTrackr

Motivations and Users

The main objective of the application is to categorize songs and artists which have appeared on the Billboard Hot 100 charts at some week in the past few years, linking them to other instances of interest, such as an artist's popular songs or a song's top chart occurrence or number of weeks spent on the charts. In addition, the application will allow users to look up a specific chart from Billboard from a desired week. The users of our application will range from casual listeners of popular music to hardcore fans of a particular genre or artist. The site should be a useful reference source of generally popular songs to music fans of all knowledge levels and varying interests. We have built the site in a way that is clean and easily-navigated, creating a user-friendly experience that caters to users of all technological backgrounds.

Requirements

Phase III User Stories

Our Team

- As a hipster, I want to be able to find songs that did not reach the top 50, so I can impress my other hipster friends with my knowledge of more obscure music.
  To implement this, for our song model page we have a filter which only displays songs that peaked below a value inputted by the user. Estimated time: 1 hour, Actual time: 30 minutes
- As a top 40 radio DJ, I want to be able to quickly find only songs that reached top 40, so I can play them on my show.
  To implement this, for our song model page we have a filter which only displays songs that peaked above a value inputted by the user. Estimated time: 15 minutes, Actual time: 10 minutes
- As a fan of a particular artist, I want to see how popular they have become out of personal curiosity/support.
  To implement this, we have a popularity metric scraped from the Spotify API associated with each artist and we display it. Estimated time: 30 minutes, Actual time: 30 minutes
- As a record label executive, I want to see how many cumulative week's our artist's songs have spent on the charts.
  To implement this, we display the sum of an artist's song's number of weeks on the chart. Estimated time: 1 hour, Actual time: 30 minutes
- As a dancer, I want to be able to find songs from an artist of a certain genre, so that I can find multiple songs that work together for a routine.
  To implement this, we added a filtering mechanism to the artist page using a dropdown menu, allowing users to filter artists by genre. Estimated time: 2 hours, Actual time: 1.5 hours

Customer Team

- As a cover artist, I want to be able to look up the lyrics of a song so I can sing them correctly.
  To implement this, we scraped the song lyrics from Genius API and display them on a song's page. Estimated time: 2 hours, Actual time: 3 hours

- As an avid Dance Monkey fan, I want to be able to see the weeks that Dance Monkey was a Top 100 hit.
  To implement this, we designed the charts page so that users can skip forward and backward one week, and thus track how a song moves up or down the chart. Additionally, users can also see the top chart hits of any song. Estimated time: 1 hour, Actual time: 1 hour
- As a person who only knows the partial title of a song, I want to be able to type in the part I know and find the song.
  To implement this, we display all songs that contain the substring that the user has typed into a search bar on the songs page. Estimated time: 1 hour, Actual time: 30 minutes
- As a fan, I want to be able to see a picture of my favorite artists, so I can match the music to a face.
  To implement this, we display an image scraped from Spotify API on each artist's page. Estimated time: 45 minutes, Actual time: 1 hour
- As a person with an addictive personality, I want to be able to see an artist's most popular albums, so I can binge-listen to their entire discography.
  To implement this, we scraped the data from Spotify API, storing it in a MongoDB database and displaying the top results on the artist's page. Estimated time: 1 hour, Actual time: 1.5 hours

Phase II User Stories

- As a Spotify user, I want to be able to open the song on Spotify so I can listen to it.
  To implement this, we added a link on each song's page that will open the song in Spotify Web Player when the album art is clicked. Estimated time: 2 hours, Actual time: 1 hour
- As a person who wants to get into music, I want to order songs by ranking to find the most popular ones quickly.
  To implement this, we have the songs in a sortable chart that can be ordered by both peak position as well as their rank in a current chart. Estimated time: 1 hour, Actual time: 1.5 hours
- As a busy person, I want to be able to quickly search a song by title and find it.
  To implement this, we have a text field that will display only the songs containing the string that is entered. The results update upon a new character being entered. Estimated time: 2 hours, Actual time: 3 hours

- As an album art creator, I want to see each song's album art for inspiration when I go to the song's page.
  To implement this, we pull an image from the Spotify API. When the user goes to a song's page, we display the square image. Estimated time: 1 hour, Actual time: 1 hour
- As a record label manager, I want to search songs by artist to analyze how different songs by the same artist compare in ranking.
  To implement this, we have a text field that will display only the artists containing the string that is entered. The results update upon a new character being entered. Estimated time: 2 hours, Actual time: 1 hour

Phase I User Stories

Our Team

- As an on the go person, I want to be able to use the application on my phone, so I can access information whenever I want.
  To achieve this, we tested our website on mobile and designed a mobile-friendly layout. Estimated time: 2 hours, Actual time: 1 hour
- As a music listener, I want to be able to search for a specific song by name, so I can look up information about it.
  To implement this, we have a text field that only displays songs containing the substring entered by the user. Estimated time: 1 hour, Actual time: 1 hour
- As a user with a large monitor, I want to be able to show many entries at once, so I can just scroll rather than hitting next.
  To implement this, we have 9 songs displayed in a grid like fashion on each page. On table view we display 10. Estimated time: 1 hour, Actual time: 2 hours
- As an organized person, I want to sort songs in alphabetical order, so I can look up songs beginning with a certain letter.
  To implement this, we display the songs alphabetically when the user clicks on Name in the table view. Estimated time: 1 hour, Actual time: 1 hour
- As an organized person, I want to sort artists in alphabetical order, so I can look up artists beginning with a certain letter.
  To implement this, we display the artists alphabetically when the user clicks on Artist in the table view. Estimated time: 1 hour, Actual time: 15 minutes

Customer Team

- As a fan of a particular song, I want to see the album it is on, so I can listen to it.

To implement this, we display the name and picture of the album with the name of the song. Estimated time: 2 hours, Actual time: 1 hour

- As a person with interests in app development, I want to be able to see more info about the project, so that I can see how I might do something similar.
  To implement this, we have an about tab which displays information about us. Estimated time: 2 hours, Actual time: 2 hours
- As a new fan of Drake, I want to be able to see all of his song names, so I can go and binge listen to them.
  To implement this, on the artist page we list their most popular songs. Estimated time: 1 hour, Actual time: 30 minutes
- As a restaurant owner, I want to be able to view the most popular songs of the past week, so that I can create a playlist for my restaurant.
  To implement this, we display the charts for a desired week. Estimated time: 1 hour, Actual time: 45 minutes
- As a nostalgic person, I want to see what was on the charts at this time last year.
  To implement this, we use a built in calendar module that allows the user to display the charts from that week. Estimated time: 2 hours, Actual time: 3 hours

Use Case Diagram

Design

The design of the application consists of a landing page with links to all relevant pages that the user would want to access. On every page there is a header consisting of the logo for the application, as well as three different options on the right hand side. The user can click Home from any page to return to the landing page. Next, there is a dropdown menu labelled Models, in which the user can choose to visit Songs, Artists, or Charts. Lastly, the user can click About to

view our team's information. So far our models are displayed in a responsive grid that displays nine songs/artists per page or table that can be configured to display 10, 20, 50, or 100 songs per page. The instances can also be sorted ascending or descending by each column category (i.e. sort by name, artist, peak position, or rank). Sorting is done by taking an javascript array of all the instance objects, creating a comparator function, and calling the build in Sort() on the array.

```
64    handleSortName = () => {
65        const { songList, updateSongList } = this.props;
66        songList.sort(this.byName);
67        updateSongList(cloneDeep(songList));
68    };
```

```
86    byName = (a, b) => {
87        const x = a.name.toLowerCase();
88        const y = b.name.toLowerCase();
89        if (x < y) return -1;
90        if (x > y) return 1;
91        return 0;
92    };
```

Figure: Functions used to sort song instances by track name

The user can also search for a specific song or artist, and all chart entries matching the search terms will appear. This is done by cloning the list of all instances into a displayed (or filtered) list, which is then processed by the built in array filter() member function with the parameter taken from the input box and updates on change with the observer design pattern. Categories are automatically populated as the instances are loaded onto the site - these categories then are stored and each can be set as a filter.

```
82      handleSearch = event => {
83          this.setState({ searchField: event.target.value });
84      };
```

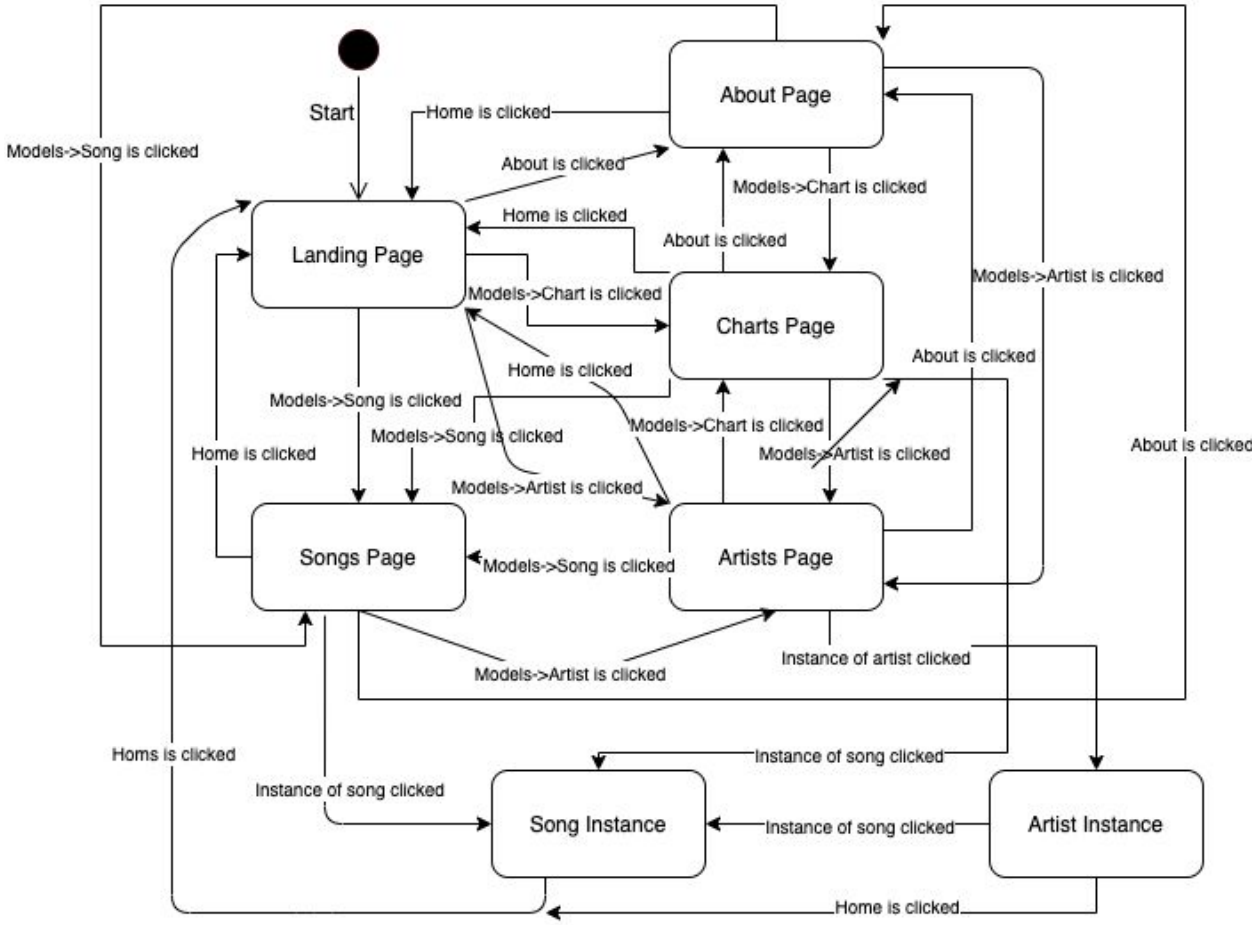
```
110     handleFilter = event =>{
111         let f = this.state.keyFilter;
112         let key = event.target.innerHTML;
113         key = key.trim();
114         if(f.includes(key))
115         {
116             let index = f.indexOf(key);
117             if (index !== -1) f.splice(index, 1);
118         }
119         else{
120             f.push(key);
121         }
122         this.setState({ keyFilter: f});
123         console.log(this.state.keyFilter);
124     }
```

Figure: Functions for handling searching and filtering songs

Our website is designed to be as lightweight and as fast as possible, so we opted for a simple bootstrap theme that looks aesthetically pleasing and is also fluid and responsive. Each component is designed around the observer design that notifies on change and updates the necessary related components.

The database is a free MongoDB Atlas cluster hosted with GCP in Iowa, with 0.5GB of storage. It has been separated into two sections during phase one: one for billboard chart information and one for song features from Spotify and Genius Lyrics. These two sections will be merged into one for easier data access in the future. Building the database was done with a Python script to query the Billboard, Spotify, and Genius Lyrics APIs and database insertion was done with pymongo. Retrieving data from the database was done using the express.js web server (hosted on heroku for the time being).

State Diagram



Testing

We did 3 main areas of testing - frontend, backend, and API. For frontend testing, we used Mocha to test the JavaScript code and Selenium for GUI testing. For the JavaScript testing with Mocha, we tested the frontend code we made using the React framework by mocking the DOM and adding different React components into it. We tested the content of the different components to ensure that they displayed what we intended to and had the correct formats. We tested the GUI by using a Selenium test suite for Chrome and checked for different navigation, getting the correct URLs on different screen sizes. We also checked for correctness of table

sorting using Selenium. Both of those used XPaths to each element to check or interact with and using By.xpath() in the webdriver's findElement(). For backend testing, we used Python's unittest library to ensure we were properly handling lyrics returned from Genius API calls. This involved testing songs with different properties, such as multiple artists, foreign language symbols, and recent releases, as well as testing how the code would handle an invalid song request. For API testing, we used Postman to ensure that our calls to our backend API worked as intended. We had two internal API methods to get a list of songs and to search for specific tracks. We tested the generic get with a Postman request and a series of different requests for searching tracks that covered different types of cases such as song names with non-alphabetical characters, numerical characters, etc.

Models

The three models incorporated in the application are Track, Artist, and Chart. First, every week we use a python script that scrapes the data from billboard.com/charts/nameOfChart/week.

```
client = MongoClient("mongodb+srv://user:pass@morning6lp-xifid.gcp.mongodb.net/test?retryWrites=true
db = client.billboard
# delete_all_billboard_songs(db)
statusResult = db.command("serverStatus")
print(statusResult)
chart_name = 'hot-100'
chart = billboard.ChartData(chart_name)
print(chart.previousDate)
for i in range(100):
    result = add_chart_data(chart)
    if result.upserted_id != None:      # a document was inserted
        print("chart", i, "not in db, add songs")
        songs_added = add_song_data(chart, result.upserted_id)
        add_song_ids_to_chart(result.upserted_id, songs_added)
    else:
        result = db.charts.find_one({'date':chart.date})
        # print(result['_id'])
        update_song_data(chart, result['_id'])
        print("chart", i, "in db, updating with new chart id")
    # add_song_data(chart)
    chart = billboard.ChartData(chart_name, date = chart.previousDate, fetch = True, timeout = 25)
```

Figure: Display of script that scrapes billboard charts

The charts contain the title of the chart, the date of the chart, as well as the top 100 ranking songs and artists for that week, which are linked to the respective artist and song instances. For songs, we are using Spotify Web API to store any track which appears in a Billboard Hot 100 chart in our database. We query the Spotify API to get attributes like album name, song image, and key.

```
def get_audio_features(track_response):
    f = {}
    f['album_name'] = track_response['album']['name']
    f['album_uri'] = track_response['album']['uri']
    f['image'] = track_response['album']['images'][0]
    f['ext_url'] = track_response['external_urls'].get("spotify", None)
    f['popularity'] = track_response['popularity']
    f['uri'] = track_response['uri']
    audio_f = sp.audio_features(track_response['uri'])[0]
    for feature in audio_f:
        f[feature] = audio_f[feature]
    f['key'] = key_class[audio_f['key']] if audio_f['key'] != -1 else None
    f['mode'] = mode[audio_f['mode']]
    return f
```

Figure: Code showing feature gathering from Spotify API

For each song instance, the lyrics are fetched by Genius API and will be displayed on the song page, and every song will have a link to the artist of the song, as well as all the charts it is featured in.

```
def request_song_info(search_param):
    base_url = 'https://api.genius.com'
    headers = {'Authorization': 'Bearer ' + '19qlI77HvC4NV0_0f-V5O-TlFh-XKrNCI7ykYH4vCMFv-263Zbh26TOMNAqOTCzX'}
    search_url = base_url + '/search'
    params = {'q': search_param}
    response = requests.get(search_url, params=params, headers=headers)

    #print('response:', response)
    return response
```

Figure: Function used in requesting and parsing Genius for lyrics

For artists, we will again use Spotify Web API to get attributes like artist genre, popularity, albums and album images, as well as images of the artist. We also display information from the

billboard charts like time spent on charts. Every artist instance will also link to all of the songs by that artist as well as the charts that the artist appears on. Each instance of our models will have its own page.

```
client = MongoClient("mongodb+srv://user:pass@morning6lp-xifid.gcp.mongodb.net/test?retryWrites=t
db = client.billboard

charts_docs = db.charts.find({})

for chart in charts_docs:
    print('\n\n', chart['date'])
    for song in chart['songs']:
        print(song['name'])
        song_doc = db.songs.find_one({'_id': song['song_id']})
        # print(song_doc)
        artist_doc = db.artists.find_one({'_id': song_doc['artist_id']})
        # print(artist_doc)
        if 'charts_in' in artist_doc:
            charts_in = artist_doc['charts_in']
        else:
            charts_in = []
        if chart['_id'] not in charts_in:
            charts_in.append(chart['_id'])
        song['artist_id'] = artist_doc['_id']
        db.artists.update_one({'_id' : artist_doc['_id']}, {"$set" : {'charts_in' : charts_in}})
    db.charts.update_one({'_id' : chart['_id']}, {"$set" : {'songs' : chart['songs']}})
```

Figure: Example of linking between models

Framework

The application is being built with React on the frontend, which handles displaying all of the pages and routing between them. The server that queries the database is built in node.js with express, and handles all of the data that flows between the frontend and database through various http endpoints. The database is a 0.5 GB MongoDB cluster supported with GCP in Iowa. The testing frameworks we used are Mocha and Selenium for testing the frontend, Python unittest for testing the backend, and Postman for testing API calls.

Reflection

With our work in Phase III we have all of the desired functionality completed. Despite the current situation, we managed to stay in communication pretty well and work together. Obviously, there have been some difficulties, but through this process we have learned a lot about working remotely via trial by fire. In truth, we could have been more efficient with our time, started earlier, done more work incrementally, but given the circumstances we think this project turned out well. Throughout this phase, we learned more about how to effectively work as a team remotely, refining tests in Mocha, Selenium, Postman, and Python unittest, and implementing different searching, sorting, and filtering algorithms. We also learned how to better communicate with each other and coordinate responsibilities, without face to face meetings.