Beatdrops

Team F

Authors: Matthew Hung, Adelle Vo,
Griffin Scotti, Naomi Donato

Created On: 1/31/22
Last Updated: 3/27/22

https://github.com/matthewhung09/beatdrops

Product Vision and Scope

---

a. Summary:

We thought it would be interesting to create a web app that makes it easy to discover new music and figure out what music is popular in a certain area. After signing up for an account on Beatdrops and linking it with your Spotify account, Beatdrops will display the song you're currently listening to to other users who are in close proximity. At any time, you will be able to see up to 10 songs that are currently being listened to by Beatdrop users who are closest to you.

b. User Personas

Person 1: Student	**Person 2:** Student
Goal: Trying to create a playlist for a party that incorporates a wide variety of music relevant to the party's demographic. **Challenges:** Hard and inconvenient to individually coordinate with everyone at the party to figure out their music tastes.	**Goal:** Trying to create a study playlist for finals week. **Challenges:** Hard to find good study music, so sitting at Kennedy Library and letting the web app run in the background is a good way to see what kind of music others are listening to when they study.

c. Stories

- As a user, I want to be able to post my playlists and songs so that I can share them with others.
- As a user I want to be able to filter posts by time and likes so that I can display songs within my desired filter.
- As a user I want to be able to connect with my Spotify so that I can post playlists and songs I already have saved on my Spotify.
- As a user, I want to be able to view posts by location so that I can know the popular songs and playlists in that specific area.
- As a product owner, I want users to be able to login to an account so they can make posts.
- As a product owner, I want users to be able to stay logged in until they log out so that they have a more convenient user experience and don't have to log in every time they want to access the web app.

- As a product owner I want to users be able to like other users' posts so that users can see popular playlists/songs
- As a system administrator, I want users to be able to login to an account with a username and password for security.

d. Non-Goals

- As a system administrator I want to be able to moderate posts so that comments and posts stay appropriate.
- As a product owner I want users be able to leave comments on others' posts so that the users can exchange their thoughts on playlists/songs

e. Future Goals

- As a user, I want to be able to login to my account so that I can view saved posts.
- As a user I want to be able to save posted playlists and songs to my Spotify so that I have them to listen to later.
- As a user, I want to be able to save posts that I see so I know which ones I like and can easily access them later.

f. Assumptions

- Our application is reliant on the Spotify API to retrieve song data and user information like playlists and liked songs.
- We assume that the user has a spotify account to use for our application
- Our application relies on the Geolocation API to get location information (latitude and longitude) and on the Geoapify API for reverse geolocating
- We assume that the user authorizes our applications to use there location

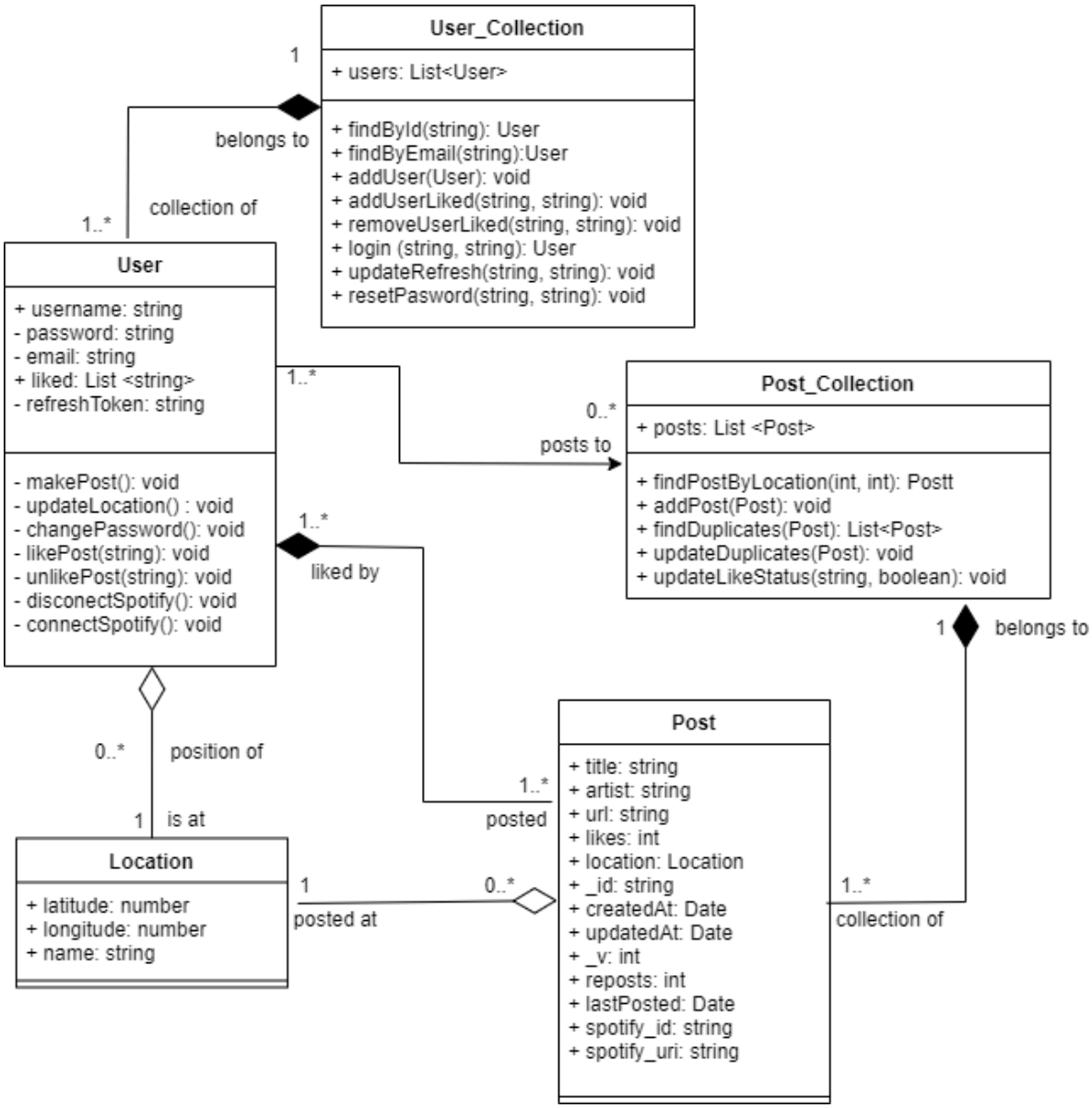
Design

Tech Stack:
Frontend: React
Backend: Express
Database: MongoDB
External APIs: Spotify, Geolocation, Geoapify, React Leaflet

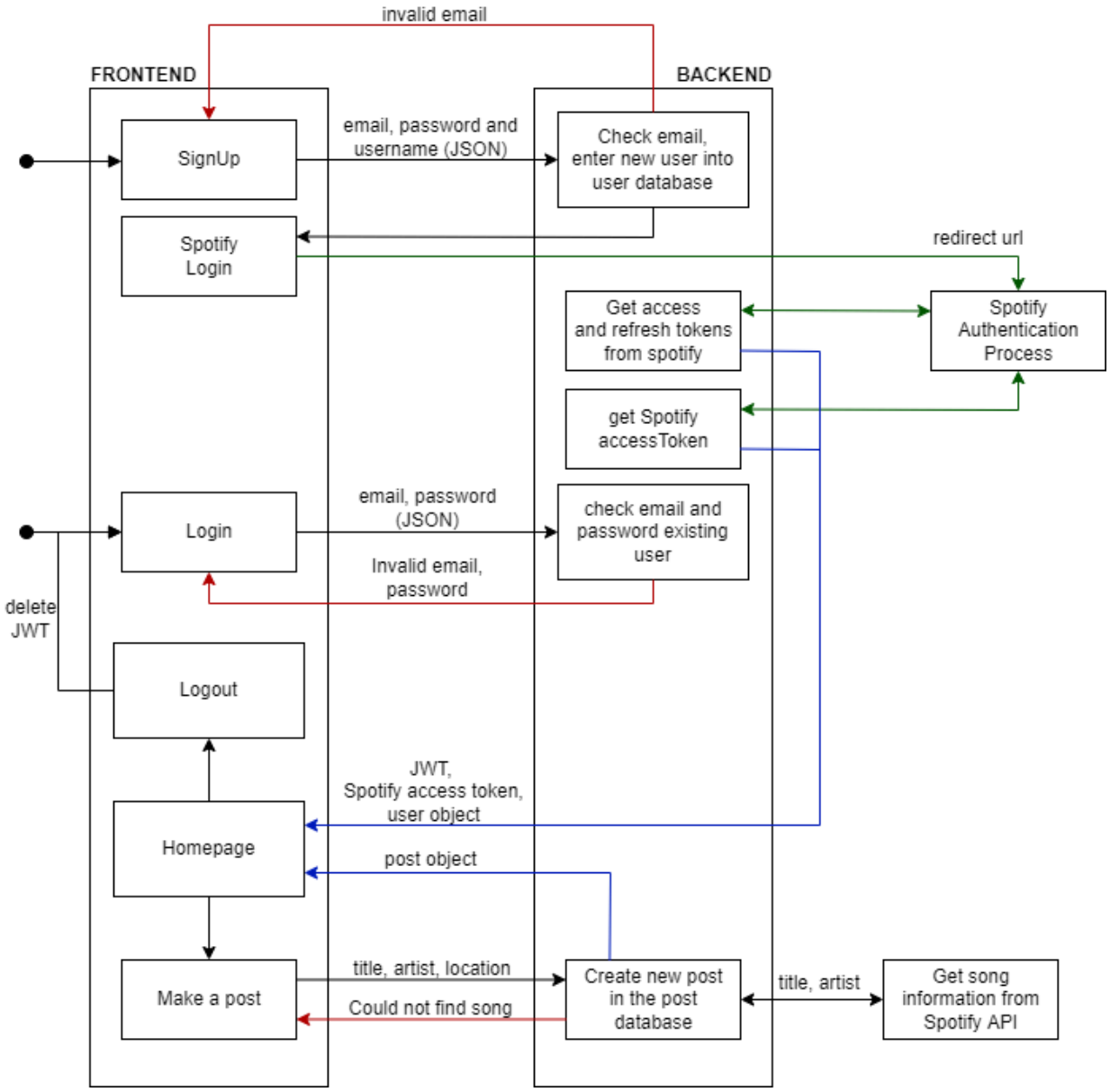
UI Prototype:
https://www.figma.com/file/rHUc0rkb2ESU2wNWrxs2Vy/Team-F---308?node-id=0%3A1

Design Diagrams:

Below is our class diagram (fig.1) and sequence diagram (fig.2). The class diagram is mostly representative of our database organization. User/User_Collection and Post/Post_Collection mostly represent our database schemas and functions for users and posts respectfully. For more information about the data schemas for users and posts, see the **Data Schema** inside the **Design** section. The sequence diagram mostly shows the login/signup related actions. The "posting" sequence provides a good overview of other actions as they all have similar sequences. For the most part, external API's, like Spotify where Post gets most of its data from, are left out of both diagrams and are explained more fully in the **External Software Interfaces** section.



(fig.1: Beatdrops Class Diagram)

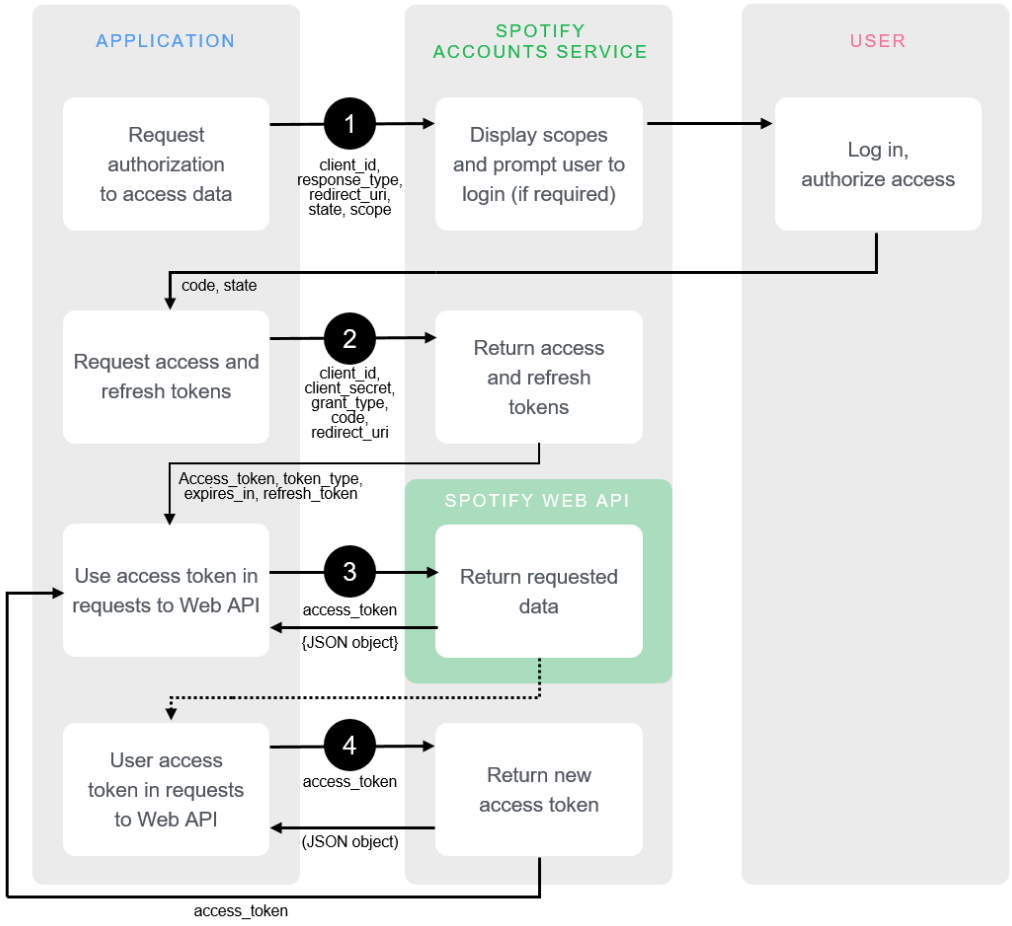


(fig. 2: BeatDrops Sequence Diagram)

External Software Interfaces:

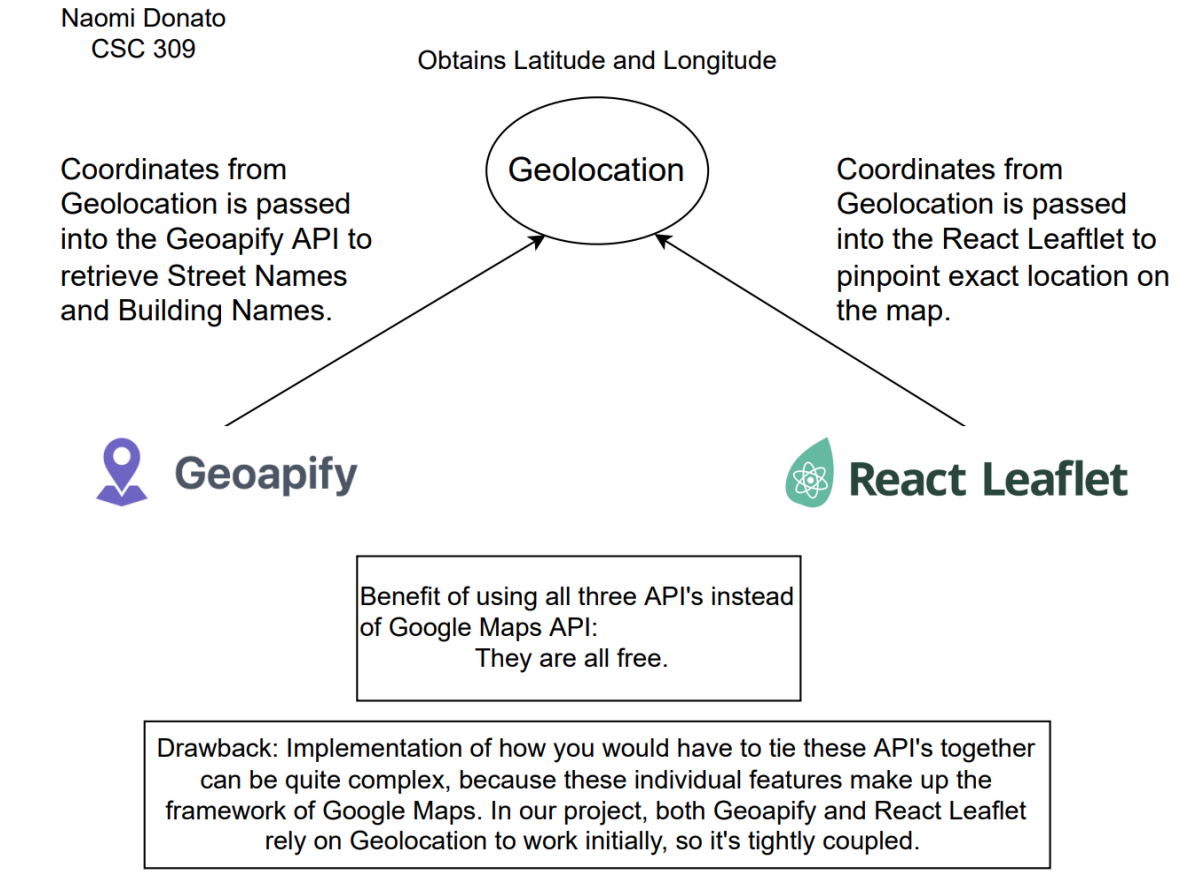
- **Spotify API**: We are using the Spotify API for 2 major parts of our application - to allow users to login to their personal account, and to query the Spotify database for song information. For the first one, authentication is necessary to allow us to see the user's currently playing song and allow them to post it easier. This is done through Spotify's authentication system - when the user clicks the login to Spotify link, we send a request to Spotify's API that then redirects the user to authorize our application's use of their

Spotify data, and then redirects them to the home page after that access is granted. A token and refresh token is given, which allow us to get information about the user's account, like their currently playing song. The refresh token is used to refresh their access token by making another request to the API and is necessary to allow them to stay logged in (See below image for Spotify authorization code flow). We also use the Spotify API to get song details when users make a post. When the user inputs a song title and artist to create a post, it sends this information to our backend, which in turn queries the Spotify API to get information like the album cover or song URL. Then we use this data to display the song and the in-browser player. For this, we use our Spotify developer credentials stored in the backend. Once the request is made, the song data is stored in our MongoDB database and sent back to the front end to update the post list.



(fig.2: Spotify Authorization Code Flow)

Location Implementation:



- **Geolocation API:** Implemented in the frontend. **The Geolocation API** allows web applications to retrieve the user's location data. The data we are using for our program is a user's current coordinates of latitude and longitude. The API is accessed by calling the read-only property, **navigator.geolocation**, which returns a geolocation object that provides Web content access to the location of the user's device. A browser pop-up will ask a user for permission to access their location data. If the user clicks 'Allow', the browser then retrieves the device's current location by the method: **Geolocation.getCurrentPosition()**. On success, the coordinates (latitude, longitude) can be accessed. In our program, when the user makes a song post, in that instance, their current location coordinates are obtained. The user can move to a different area and their coordinates will update when they make another post in the new area.
  See fig.3 below for how the api is implemented in our program.

```
// get coordinates from navigator API
useEffect(() => {
    if (navigator.geolocation) {
        navigator.permissions.query({ name: "geolocation" }).then(function (result) {
            if (result.state !== "denied") {
                // console.log(result.state);
                navigator.geolocation.getCurrentPosition((position) => {
                    setLat(position.coords.latitude);
                    setLong(position.coords.longitude);
                });
            }
        });
    }
}, [lat, long]);
```

(fig.3: Use of the Geolocation API inside our web application)

- **Geoapify API:** Implemented in the frontend. The coordinates obtained from the Geolocation API are crucial for the **reverse geolocation API** implemented, called **Geoapify**. After many unsuccessful trials of finding a Maps/Places API that does not require credit card information, the location platform provides a free service, good enough for the current state of our program (because we are using its free service option, we are limited to about 5 request/second, with isochrones up to 15 min, and isodistances up to 10 km). This platform offers the Map API that we need in order to obtain the actual building name of a location, instead of just a street name. In order to have access to the url and API functionalities, we are given an API Key once signed up. The user's latitude and longitude coordinates are passed into the Geoapify's API URL, allowing access to descriptive properties about the location of the user. It is able to display street name, city, address line 1 (which is the building name), country, etc. For our project, we wanted address line 1, and were successfully able to display the name of the building in Cal Poly Slo where the song post was made. The building names displayed are quite specific to the point that it displays building numbers as well (Ex: Frank E. Pilling Building (14)), therefore building numbers are stripped away when location is displayed in a song post. For location limitations, only building or area names within Cal Poly Slo will display, so when a song post is made outside of campus, the street name will be presented instead of specific building/area name. This allows our application to be exclusive to our university. See fig.4 for how the Geoapify API is implemented in our program.

```
// use reverse geocoding API to get location based on coordinates
async function getPostPosition() {
    const url = `https://api.geoapify.com/v1/geocode/reverse?lat=${lat}&lon=${long}&apiKey=211461662e434824aa8bd651b237c69a`;

    try {
        // const response = await axios(url);
        const response = await http.get(url);
        let geoData = response.data.features[0].properties;

        // random off campus place
        if (geoData.name === undefined) return geoData.street;
        // not at a specific campus building
        else if (geoData.name === "California Polytechnic State University")
            return geoData.street;
        // strip number from on campus buildings
        else if (geoData.name.includes("("))
            return geoData.name.substring(0, geoData.name.indexOf("("));
        // by default, return name of place
        return geoData.properties.name;
    } catch (error) {
        console.log(error);
    }
}
```

(fig.4: Implementation of the Geoapify API in our web application)

React-Leaflet

React-leaflet is a component friendly implementation of Leaflet. It is a leading open-source Javascript library for interactive maps. We used the map to display song posts in specified areas on Cal Poly Slo's campus. Pinned (music notes) locations on map are dependent on latitude and longitude of each song post. Therefore, React-Leaflet is dependent on the Geolocation API. Song title and user from each song post are displayed on the popup of their specific note. Therefore, if a note pin in a certain building/area was clicked on, the popup displays all the song titles and artists posted within that location. To avoid overlapping of note pins, if two or more notes are within a proximity of 0.000005 degrees, they will merge into one note that contains each of the pin's song posts as a list on the popup.

Data Schema:

Token: A Token contains the userId of the user that made the request to reset their password and then a token attribute that represents the slug of their unique password reset link url.

Post: A Post contains information about a post and is used to add a new post to the database when the user clicks on "Post a song". Title, artist, likes, and location are obtained from the queries to the Spotify API and are all displayed on the frontend for the user and the likes field updates when users like/unlike posts. The url is used to display the embedded song player so users can play songs in the browser. The repost keeps track of how many times a post is reposted, and we were planning to display this to the users. The lastPosted attribute keeps track of data that allows us to expire the posts after 72 hours. See fig.5 below for the code of our post shema as represented in our MongoDB posts database.

```
const PostSchema = new mongoose.Schema(
  {
    title: {
      type: String,
      required: true,
      trim: true,
    },
    artist: {
      type: String,
      required: true,
      trim: true,
    },
    likes: { type: Number },
    lastPosted: {
      type: Date,
      default: new Date(),
      required: true,
      expires: 259200,
    },
    location: {
      name: String,
      lat: Number,
      long: Number,
      onCampus: Boolean,
    },
    url: {
      type: String,
      required: true,
      trim: true,
    },
    spotify_id: {
      type: String,
      trim: true,
    },
    spotify_uri: {
      type: String,
    },
  },
  { collection: "Posts", timestamps: true }
);
```

(fig.5: Schema for a Post inside our post database)

Post Relationships:

- Posts can be made by a User
    - (many-to-one)
- A Post can be liked by Users
    - (one-to-many)

User: A User contains information about a user, such as username and liked songs, and is used to add a new user to the database when the user signs up. The password is encrypted before being stored in the database, and the email field is marked unique so that one email can't be used to sign up multiple times. When a user logs in, we add a cookie to the browser containing their user information, and then requests to the backend contain that cookie to update the correct user accordingly. The refresh token is from keeping users logged into Spotify for our app. Spotify provides a refresh token to us when users authenticate throughout our app. We use this token to get access tokens for the SPotify API (see *Spotify API* in *External Software Interfaces*). See fig.6 below for the code of our user schema as represented in our MongoDB users database.

User Relationships

- A User can make Posts
    - (one-to-many)
- A User can like Posts
    - (one-to-many)

```
const UserSchema = new mongoose.Schema(
  {
    username: {
      required: true,
      type: String,
      trim: true,
    },
    password: {
      required: true,
      type: String,
      trim: true,
    },
    email: {
      required: true,
      type: String,
      unique: true,
      lowercase: true,
    },
    liked: [mongoose.ObjectId],
    refresh_token: { type: String },
  },
  { collection: "Users", timestamps: true }
);
```

(fig.6: Schema for a User inside our user database)

Deliberation

- Frequency of when to update post list for users, refresh page automatically in interval
- How to modify schema in order to load x most recent posts
- Getting BeatDrops approved by Spotify for full deployment
- How to handle location information when posts are made outside of campus

Software Tests

Unit/Integration Tests:

```
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |   96.72 |      100 |      90 |   96.68 |
 backend              |     100 |      100 |     100 |     100 |
  backend-services.js |     100 |      100 |     100 |     100 |
 backend/models       |   95.34 |      100 |    87.5 |   95.34 |
  post-services.js    |   93.33 |      100 |   85.71 |   93.33 | 7-8
  post.js             |     100 |      100 |     100 |     100 |
  token-services.js   |   92.59 |      100 |   83.33 |   92.59 | 8-9
  token.js            |     100 |      100 |     100 |     100 |
  user-services.js    |   96.42 |      100 |      90 |   96.42 | 8-9
  user.js             |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|-------------------

Test Suites: 2 passed, 2 total
Tests:       43 passed, 43 total
Snapshots:   0 total
Time:        7.664 s
Ran all test suites.
```

Acceptance Tests:

Feature #1: Post song to homepage

Scenario: Successful song post.

GIVEN I have chosen to post a song to the homepage.

WHEN I spell the song and title correctly, and the song exists on Spotify.

THEN I am able to see my song post on the homepage.

Scenario: Unsuccessful song post.

GIVEN I have chosen to post a song.

WHEN I spell a song name incorrectly.

THEN I get an error message telling me that I spelled the song incorrectly.

Feature #2: Get posts

Scenario: Successfully get posts

GIVEN I am on the homepage

WHEN I look at the list of posts

THEN I should see the posts on the homepage.

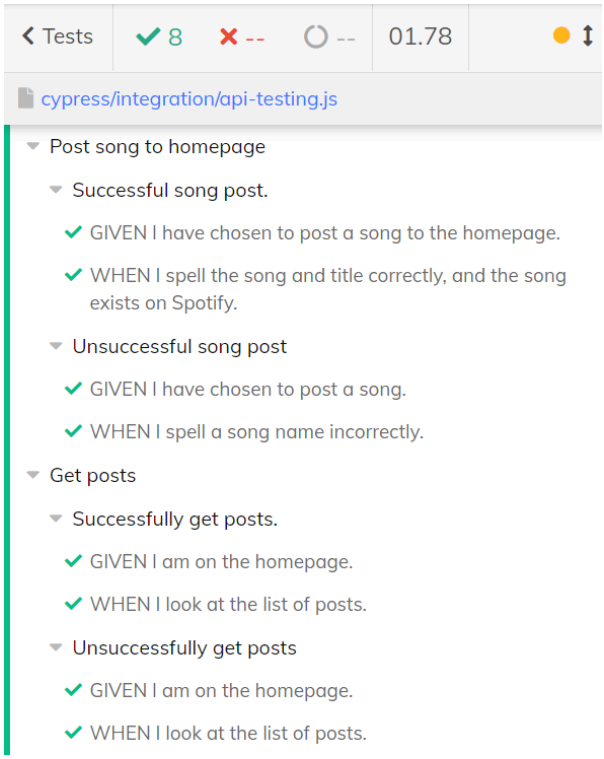
Scenario: Unsuccessfully get posts

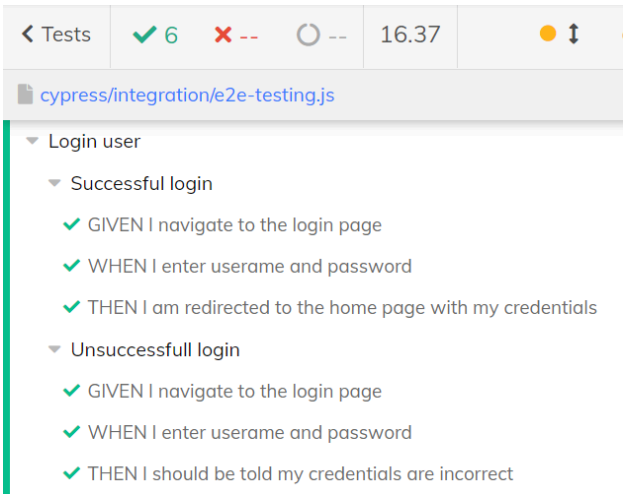
GIVEN I am on the homepage

WHEN I look at the list of posts

THEN I should not see any posts and be redirected to login again

Feature #3: Login
Scenario: Successful login
GIVEN I have chosen to login
WHEN I enter my email and password
THEN I should be redirected to the homepage
Scenario: Incorrect login credentials
GIVEN I have chosen to login
WHEN I enter an incorrect email and password
THEN I should be told my credentials are wrong





Updates from Previous Version

- New class diagram
- React Leaflet
- Token data schema
- Updated existing schemas (post)
- Acceptance tests