CIS 550: Database and Information Systems

Homework 2: Web DB

Please read this handout from start to finish before proceeding to work on the assignment

Introduction

In this two-part assignment, you will incrementally build an interactive Spotify-themed web application using React and Node.js backed by a MySQL RDS database.

To help you understand the fundamentals of web interface design and implementation, in the first part (Part 1, worth 50 points) you will build an API on Node.js following the specification that we lay out for you (outlined below). This will help you understand how APIs are able to process and facilitate data interchange between various (client) applications and databases.

Once you familiarize yourself with the two 'lowest' tiers of the architecture (the API server and database), you will use your API to develop a frontend application using React.js that uses it. Specifically, in Part 2, worth 50 points, you will develop an interactive multi-page client using UI component libraries MUI (based on Google's Material Design system) and recharts (for data visualization).

Part 2 primarily serves as a primer on front-end web development and by the end of the assignment, you can hope to have a great template that you can refer to, modify, and use for the project. It is developed to be similar to a 'follow-along' exercise on React and its component libraries. In fact, we provide you with most of the code and many examples, and the short tasks ask you to fill in or correct some of this implementation. In doing so, you will have the benefit of learning these (much needed) web development skills while avoiding a steeper learning curve.

Advice to Students

For those of you that come into class with an understanding of web development, we hope this homework will still be beneficial for you to refresh your skills and also potentially learn about new UI libraries and techniques. If you are relatively more inexperienced, you might find this homework more time-consuming; please attend recitation and complete Exercise 3 to prepare for this homework.

This assignment is crucial for you to understand many external applications of databases and will provide you with the opportunity to develop many important qualities and skills that you need as a software developer like learning to read documentation, understanding and implementing specifications, and debugging.

If you are already familiar with version control (Git), we recommend you use it to regularly commit and save your work. Since this assignment also serves as a precursor to the term project (which will require Git) use this as an opportunity to learn some Git basics! There are plenty of

resources available online, such as Atlassian's guide on Git. While you can certainly choose to not use Git right away, please be sure to save your work from time to time.

The SQL queries in Part 1 are intentionally designed to be relatively easy. Instead, we would like you to get hands-on experience with web-development that uses databases at its core. You should also pay careful attention to all the details in the specifications for this part.

Part 2 of the assignment will guide you through the implementation of a React client using UI libraries 'in a bubble'. This is because we want you to take advantage of a highly-guided and gentle introduction before you eventually are able to fully implement such software independently. For example, we already made the application structure, selected and imported the libraries, will point you to the exact page of the documentation that you need to follow, and made examples that you can use to understand the usage of these in a very specific context. However, if this were your project, you would have to do a lot of these tasks on your own (even if you use the code developed for this assignment as a template)!

You should not underestimate the time you might need to spend on choosing the right libraries, finding the correct portion of the documentation, debugging, and going through the many other steps that you need to complete even before you are ready to code your planned implementation.

You will also develop some code collaboration skills from this assignment - when working on a large project, you will inevitably have to read, understand, and possibly modify or add to code written by your peers. This also includes debugging and correcting that code. You'll find this analogous to the short tasks we lay out for you (in Part 2) in that you will need to understand first our implementation and the requirements first, and then edit or extend the code. Of course, you will need to do this much more independently for the project, but working through this part with this in mind will help you build the right mindset.

We are confident that a thoughtful attempt at this assignment will prepare you well for the project and provide you with many necessary applied skills to successfully apply them for your software career beyond the course!

Please start early, be patient, and avoid last minute Ed Discussion and OH traffic! Please also **read all the comments** to better understand the components as a whole, not just the parts you are writing.

Setup

Required

You will need the latest version of Node.js on your machine for this assignment. You should verify that the following commands run and give a reasonable output on your terminal:

```
npm -v
node - v
```

The recommended Node version is 19.3.x, where x can be any number - slightly older/newer versions of Node would probably work as well. If you have problems with older Node versions, you should update Node.

You will also need to use the (built-in) terminal for your operating system and should have a code editor (with the ability to open *.js*, and *.json* files)

For MacOS Users

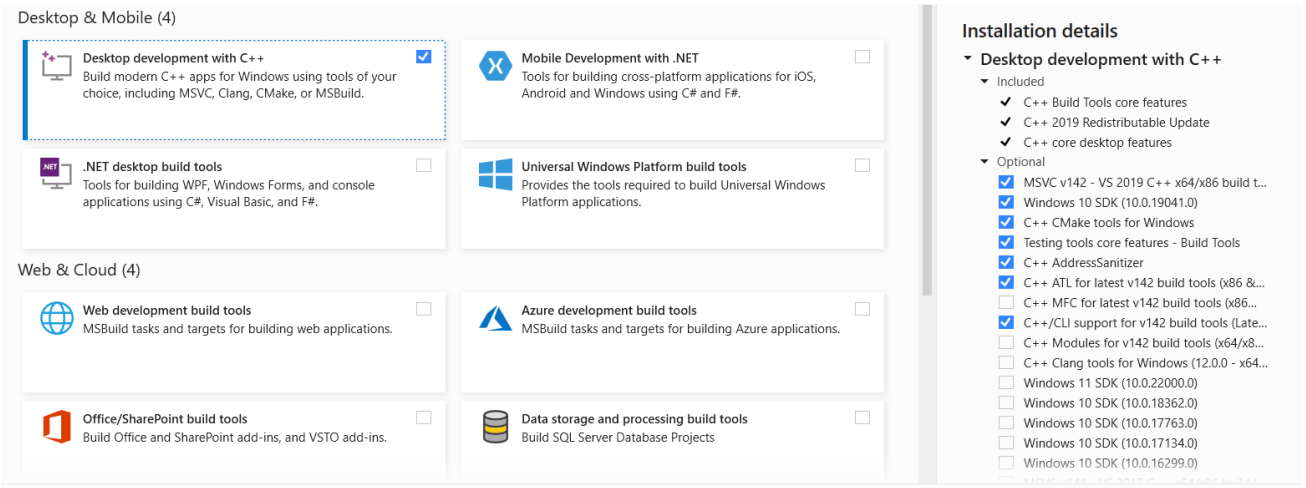
You will also need to install the XCode command-line tools (**if you are using a Mac**). To do this, run:

```
xcode-select --install
```

If you have an (incompatible or outdated) XCode version from a previous installation, you might need to update it (see here).

For Windows Users

On **Windows**, you will need to install Microsoft's Visual Studio Build Tools (specifically, the C++ build tools). The 'Desktop development with C++' should show recommended modules but it is recommended that you also install the CLI support modules as shown below:



You might find this article helpful for troubleshooting if need be.

If you are on **Windows**, you will also need a Python installation.

Recommended

We recommend that you use Visual Studio (VS) Code (as the code editor and its built-in terminal). Another great option is WebStorm, which is included in the same student license as DataGrip.

You will also need a web browser with a developer console (highly recommended for testing your API). We recommend Google Chrome, but most major browsers will support equivalent functionality.

Application Structure

Unzip HW2_student.zip from the assignment page. You should have the following files (please read over the explanations for each file):

/server

This folder holds the server application files, tests, and dependencies (as required by Node.js).

- .gitignore: A gitignore file for the Node application. Read more on .gitignore files here.

- config.json: Holds the RDS connection credentials/information and application configuration settings (like port and host). You will need to edit this file with your database credentials.
- package.json: maintains the project dependency tree; defines project properties, scripts, etc
- package-lock.json: saves the exact version of each package in the application dependency tree for installs and maintenance.
- routes.js: This is where the code for the API routes' handler functions go. We have already defined the necessary routes for you - follow the *'TODO:'* comments and implement/modify them as specified). You will need to edit this file and complete the labeled tasks.
- server.js: The code for the routed HTTP application. You will see that it imports *routes.js* and maps each route function to an API route and type (like GET, POST, etc). For this HW, we will only use GET requests. It also 'listens' to a specific port on a host using the parameters in *config.json*.

/server/__tests__

This folder contains the test files for the API:

- results.json: Stores (some) expected results for the tests in a json encoding.
- tests.js: Runs the test cases using the Jest JavaScript testing library.

/client

- .gitignore: A gitignore file for the client application. Read more on .gitignore files here
- package.json: maintains the project dependency tree; defines project properties, scripts, etc
- package-lock.json: saves the exact version of each package in the application dependency tree for installs and maintenance

/client/public

This folder contains static files like index.html file and assets like robots.txt for specifying web page titles, crawlability, et cetera (more info here).

/client/src

This folder contains the main source code for the React application. Specifically:

- App.js: This holds the root component of the React application and provides the theme of the application (using MUI).
- config.json: Holds server connection information (like port and host). Do not edit unless for some reason you change the default "localhost" and port "8080" in /server/config.json.
- index.js: This the main JavaScript entry point to the application and stores the main DOM render call in React. For this application, page routing via components and imports for stylesheets are also embedded in this file.
- **/helpers**: This folder contains a JavaScript file to format data:
  - formatter.js: Provides two functions to format times and dates
- **/components**: Similar to the /pages folder, but this folder contains files for React components corresponding to smaller, reusable components, especially those used by pages.
  - LazyTable.js: A lazy loading (only fetches the rows it needs) table leveraging MUI's basic table component with syntax very similar to MUI's DataGrid component. You will need to edit this file and complete the labeled tasks.

    - NavBar.js: Top navigation bar to navigate between different pages of the app.
    - SongCard.js: Modal containing information about a selected song. You will need to edit this file and complete the labeled tasks.
- **/pages**: This folder contains files for React components corresponding to the four pages in the application (see the sections below for more details). These are:
    - AlbumsInfoPage.js: A page to show information about a specific album. You will need to edit this file and complete the labeled tasks.
    - AlbumsPage.js: A page that lists all albums and contains links to album specific pages (defined in *AlbumsInfoPage.js*). You will need to edit this file and complete the labeled tasks.
    - HomePage.js: The landing page for the app. Contains among other things a song of the day and tables with the top songs and albums, with links to relevant pages. You will need to edit this file and complete the labeled tasks.
    - SongsPage.js: A page to search and filter results for songs based on various parameters (name, duration, plays, danceability, energy, valance, explicit). You will need to edit this file and complete the labeled tasks.

/data
This folder contains the data to be loaded into RDS:

- albums.csv: CSV formatted data for each album.
- songs.csv: CSV formatted data for each song.

Getting Started

Make sure that you have the required software installed and have a good understanding of the lecture and recitation materials before proceeding.

Open the unzipped HW2 folder. If you are using VS Code, you should be able to do this by clicking *File -> Open Folder* from the top menu. You could also just use the 'code' command on the terminal or right click on the folder and select 'open with code' if you have added VS code to the terminal or to the options menu for your system respectively.

Open a new terminal (on VS code) and cd into the server folder, then run npm install:

```
cd server
npm install
```

Do the same for the client (you should run `cd ../client` instead of `cd client` if in the **/server** folder):

```
cd client
npm install
```

This will download and save the required dependencies into the *node_modules* folder within the **/client** and **/server** directories.

Note: You might encounter some warnings about deprecated dependencies and vulnerabilities, but you can safely ignore them for this assignment.

CIS 550 Spring 2023

Importing Data

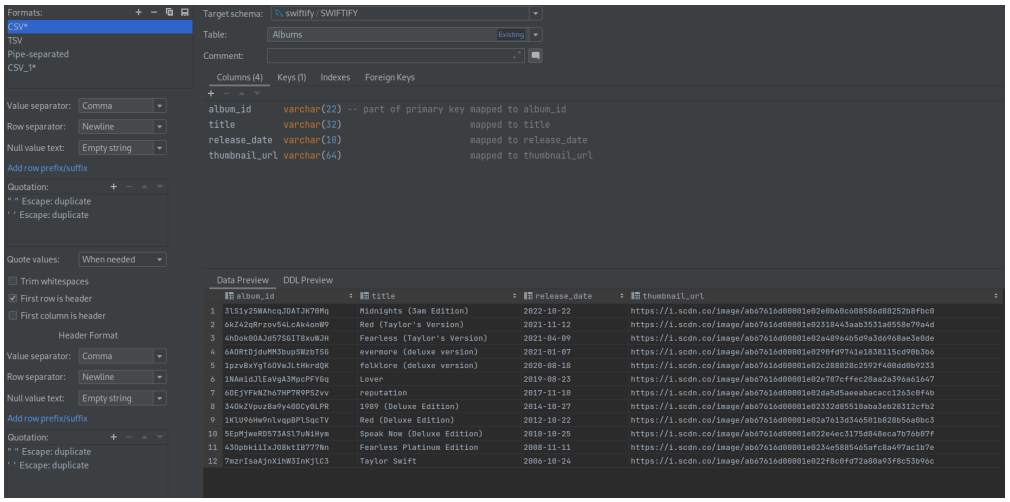
Set up a MySQL instance on AWS RDS (allow in and outbound traffic of 'All Types' from 'Anywhere' for the instance, not just from 'My IP'). Delete any other security group rules. Connect to the database using DataGrip (as outlined in the DataGrip handout from HW1). Open a new query execution console and run the following DDL statements in a database titled SWIFTIFY (that is, first run `CREATE DATABASE SWIFTIFY; USE SWIFTIFY`).

```
CREATE TABLE Albums (
     album_id varchar(22) PRIMARY KEY,
     title varchar(32),
     release_date varchar(10),
     thumbnail_url varchar(64)
);

CREATE TABLE Songs (
     song_id varchar(22) PRIMARY KEY,
     album_id varchar(22),
     title varchar(128),
     number int,
     duration int,
     plays int,
     danceability float,
     energy float,
     valence float,
     tempo int,
     key_mode varchar(8),
     explicit bool,
     FOREIGN KEY (album_id) REFERENCES Albums (album_id)
);
```

Then, use the DataGrip import wizard to import data/Albums.csv and data/Songs.csv into their respective tables (refer to the DataGrip handout and the picture below if you run into issues). When importing the files, ensure that the "First row is header" option is checked, or you will run into errors.

Fill in the db credentials into config.json in the /server folder



Dataset Overview

Most columns in the dataset are fairly self explanatory. A couple notable attributes of Song that may not be obvious are **`Songs.number`** (track number on the album), **`Songs.duration`** (length of song in seconds), **`Songs.danceability`** (float from 0 to 1 representing how danceable the song is as according to Spotify), **`Songs.energy`** (similar, but representing how energetic the song is), and **`Songs.valence`** (similar, but representing how happy the song is). **`Songs.explicit`** is 1 if the song is explicit, 0 otherwise.

JavaScript Syntax Refresher

JavaScript has a very similar syntax to many languages you are familiar with (e.g. Java), with a few peculiarities pointed out in this section. If you are interested in a more comprehensive refresher, the Mozilla docs' JavaScript Language Overview is highly suggested.

JavaScript is a weakly typed language. Variables are defined with **`let`** or **`const`** (**`var`** is discouraged in modern usage). They can be numbers, booleans, strings (can use single or double quotes), arrays (defined with **`[]`**), objects (defined with **`{}`**), **`undefined`**, or **`null`**. **`undefined`** and **`null`** are falsy, meaning when evaluating boolean statements they are cast to a boolean false when used in evaluating conditional statements. Similarly, so are 0 and empty strings. Most other values are truthy.

Two operators used often in JavaScript (and the homework) but less so in other languages is the ternary statement **`a ? b : c`** which returns b if a is true, otherwise returning c, and the nullish statement **`a ?? b`** which returns b if a is null or undefined and a otherwise.

Finally, lambda expressions are heavily used in JavaScript (and the homework). They can be assigned to variables, and can be defined in various ways. The following are equivalent.

`const f = function(a) { return a + 1 };`
`const f = (a) => a + 1;` (one line functions only)
`const f = (a) => { return a + 1 };` (for multi-line functions)

CIS 550 Spring 2023

Part 1: Node API

(50 points)

Understanding API Routes

Recitation 3 and the lecture materials provide some great guidance on this, but here is a summary of how routes work on a server. First, you should start the server application by running the command **`npm start`** in a terminal window and follow along as needed. The following output confirms that the server is running on localhost:

```
$ npm start

> server@1.0.0 start
> nodemon server.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server running at http://localhost:8080/
```

The server application accepts incoming HTTP requests by 'listening' on a specified port on a host machine. For example, this application (server.js) runs on the host 'localhost' and port 8080 as specified using the configuration file (*config.json*).

Upon receiving a request from a client, the server application parses the URL string to map it to a registered route handler, extracting important information including the route and query parameters. For example, a request to 'http://localhost:8080/author/name' from your browser will use the GET route registered on the server application, map it to the route handler function **`author(req, res)`** in *routes.js*, (this is Route 1 in the code) and return the string `"Created By John Doe"` using the res.send function. Here, **`req`** maps to the request object and **`res`** the response object.

You should look closely at these parts of server.js and routes.js. to confirm and consolidate your understanding, and note that the behavior of the route is different when route parameter **`type`** (which in our example equals "name") is changed to "pennkey." Once fully implemented, this would return `"Created By jdoe"` instead. This route parameter can be accessed via the request object as **`req.params.type`**.

Now take a quick look at the second route, **`random(req, res)`**. Notice that it does not have a route parameter, it uses an optional query parameter instead which can be accessed with **`req.query.explicit`**. If the value is "true" (http://localhost:8080/random?explicit=true), the route will return a song that may or may not be explicit, and if the value is anything else or undefined (http://localhost:8080/random) the route will return a song that is definitely not explicit.

With this understanding, let's move on to implementing the routes in routes.js.

CIS 550 Spring 2023

Testing

When writing these routes, you will of course need to test them and see the results they return. Here are two ways to do so.

Method 1 (Inspecting responses through a browser)

After starting the server application, open Google Chrome (or any web browser). We can see this in route 1.

There are three cases to test here in terms of behavior (if `req.params.type` equals “name”, “pennkey”, or neither). Head over to the following three links on your web browser and inspect the output against the spec:

- http://localhost:8080/author/name
- http://localhost:8080/author/pennkey
- http://localhost:8080/author/error

Observe that the case where `req.params.type` equals “pennkey” returns an error instead. You will need to fix this in task 2.

Method 2 (Unit testing)

Ensure that the server is not currently running (to close the server process if it’s running on a terminal, use Command + C on a Mac or Ctrl + C on Windows). You can run the provided tests in the __test__ directory within the server folder by running:

```
npm test
```

You will see 2 failing tests for this route. You will also see other failing tests, but those are not for this route, so don't worry about them for now!

- `GET /author/name`
- `GET /author/pennkey`

Let’s look at the first test case (`GET /author/name`), which is contained within the following function in tests.js. The second test case (`GET /author/pennkey`) is very similar.

```
test( name: 'GET /author/name', fn: async () => {
  await supertest(app).get('/author/name')
    .expect(200)
    .then((res) => {
      expect(res.text).toMatch( expected: /(?!.* John Doe$)^Created by .*$/);
    });
});
```

Briefly, this test case uses regex to check if the returned string contains the string “Created by” and does not contain the string John Doe. This test case can easily be fixed to pass by completing task 1 (changing the name and pennkey variables), while the second test case requires you to complete task 2 (fix the case where type equals “pennkey”.

Our test cases are in no way exhaustive, so it is best practice to write more test cases of your own, however these will be the same test cases used for grading so it is sufficient to just pass the provided test cases.

CIS 550 Spring 2023

Routes/API Specification

(50 Points)
Here, all the backend routes, their API specifications, and related tasks are listed. These are tasks 1-12 and consist of half the points in the assignment.

IMPORTANT NOTE: Parameters, unless specified, are required. Optional parameters are marked with an asterisk (*). Default values are indicated when necessary. You will find these links helpful in understanding route and query parameters.

Route 1

Route: `/author/:type`
Description: returns the author of the app in the format "Created by {name/pennkey}".
Route Parameter(s): **`type`** (string)
Query Parameter(s): *None*
Route Handler: **`author(req, res)`**
Return Type: String
Expected (Output) Behavior:

- Case 1: If the route parameter (**`type`**)='name'
  - Return "Created by [your name here]"
- Case 2: If the route parameter(**`type`**)= 'pennkey
  - Return "Created by [your pennkey here]"
- Case 3: If the route parameter is defined but does not match cases 1 or 2:
  - Return "'[**`type`**]' is not a valid author type. Valid types are 'name' and 'pennkey'.

Tasks:

- Task 1 (2 pts): replace the values of name and pennKey with your own
- Task 2 (2 pts): edit the else if condition to check if the request parameter is 'pennkey' and if so, send back response 'Created by [pennkey]'

Route 2

Route: `/random`
Description: Returns a random song
Route Parameter(s): *None*
Query Parameter(s): **`explicit`** (string)*
Route Handler: **`random(req, res)`**
Return Type: JSON Object
Return Parameters: { **`song_id`** (string), **`title`** (string) }
Expected (Output) Behavior:

- Case 1: If the query parameter **`explicit`** is defined and equals "true"
  - Return random song that may or may not be explicit
- Case 2: If the query parameter **`explicit`** is not defined or does not equal "true"
  - Return random song that definitely is not explicit

Tasks:

- Task 3 (2 pts): also return the song title in the response

Route 3

Route: `/song/:song_id`
Description: Returns all information about a song
Route Parameter(s): `song_id` (string)
Query Parameter(s): *None*
Route Handler: `song(req, res)`
Return Type: JSON Object
Return Parameters: { `song_id` (string), `album_id` (string), `title` (string), `number` (int), `duration` (int), `plays` (int), `danceability` (float), `energy` (float), `valence` (float), `tempo` (int), `key_mode` (string), `explicit` (int) }
Expected (Output) Behavior:

- If a valid `song_id` is provided, return the relevant information from the database as specified in return parameters, otherwise the behavior can be undefined (although it is best to just return an empty object)

Tasks:

- Task 4 (4 pts): implement a route that given a song_id, returns all information about the song

---

Route 4

Route: `/album/:album_id`
Description: Returns all information about an album
Route Parameter(s): `album_id` (string)
Query Parameter(s): *None*
Route Handler: `album(req, res)`
Return Type: JSON Object
Return Parameters: { `album_id` (string), `title` (string), `release_date` (string), `thumbnail_url` (string) }
Expected (Output) Behavior:

- If a valid `album_id` is provided, return the relevant information from the database as specified in return parameters, otherwise the behavior can be undefined (although it is best to just return an empty object)

Tasks:

- Task 5 (4 pts): implement a route that given a album_id, returns all information about the album

---

Route 5

Route: `/albums`
Description: Returns all albums ordered by release date
Route Parameter(s): *None*
Query Parameter(s): *None*
Route Handler: `albums(req, res)`
Return Type: JSON Array
Return Parameters: [{ `album_id` (string), `title` (string), `release_date` (string), `thumbnail_url` (string) }, …]

Expected (Output) Behavior:

- Return an array of objects sorted by release date descending

Tasks:

- Task 6 (4 pts): implement a route that returns all albums ordered by release date (descending)

Route 6

Route: `/album_songs/:album_id`
Description: Returns all songs on a given album (with some but not all information about each song) ordered by track number ascending
Route Parameter(s): **`album_id`** (string)
Query Parameter(s): *None*
Route Handler: **`album_songs(req, res)`**
Return Type: JSON Array
Return Parameters: [{ **`song_id`** (string), **`title`** (string), **`number`** (int), **`duration`** (int), **`plays`** (int) }, …]
Expected (Output) Behavior:

- Return an array of objects sorted by track number ascending

Tasks:

- Task 7 (4 pts): implement a route that given an album_id, returns all songs on that album ordered by track number (ascending)

Route 7

Route: `/top_songs`
Description: Returns songs with some relevant information in order of number of plays, optionally paginated
Route Parameter(s): *None*
Query Parameter(s): **`page`** (int)*, **`page_size`** (int)* (default: `10`)
Route Handler: **`top_songs(req, res)`**
Return Type: JSON Array
Return Parameters: [{ **`song_id`** (string), **`title`** (string), **`album_id`** (string), **`album`** (string), **`plays`** (int) }, …]
Expected (Output) Behavior:

- Case 1: If the page parameter (**`page`**) is defined
  - Return songs with all the above return parameters for that page number by considering the **`page`** and **`page_size`** parameters. For example, page 1 and page 7 for a page size 10 should have entries 1 through 10 and 61 through 70 respectively.
- Case 2: If the page parameter (**`page`**) is not defined
  - Return all songs with all the above return parameters

Tasks:

- Task 8 (2 pts): use the ternary (or nullish) operator to set the pageSize based on the query or default to 10
- Task 9 (4 pts): query the database and return all songs ordered by number of plays (descending)

- Make sure you only return the attributes specified in the return parameters and use AS to make sure the schema matches.

- Task 10 (4 pts): reimplement TASK 9 with pagination
  - LIMIT/OFFSET: https://www.w3schools.com/php/php_mysql_select_limit.asp

---

Route 8

Route: `/top_albums`
Description: Returns albums with some relevant information in order of aggregate number of plays (across all songs in the album), optionally paginated
Route Parameter(s): *None*
Query Parameter(s): **`page`** (int)*, **`page_size`** (int)* (default: `10`)
Route Handler: **`top_albums(req, res)`**
Return Type: JSON Array
Return Parameters: [{ **`album_id`** (string), **`title`** (string), **`plays`** (int) }, …]
Expected (Output) Behavior:

- Case 1: If the page parameter (**`page`**) is defined
  - Return albums with all the above return parameters for that page number by considering the **`page`** and **`page_size`** parameters. For example, page 1 and page 3 for a page size 3 should have entries 1 through 3 and 7 through 9 respectively.
- Case 2: If the page parameter (**`page`**) is not defined
  - Return all albums with all the above return parameters

Tasks:

- Task 11 (8 pts): return the top albums ordered by aggregate number of plays of all songs on the album (descending), with optional pagination (as in route 7)
  - You will need to use SQL aggregation with a GROUP BY

Route 9

Route: `/search_songs`
Description: Returns an array of song with all their properties matching the search query ordered by title (ascending)
Route Parameter(s): *None*
Query Parameter(s): **`title`** (string)*, **`duration_low`** (int)* (default: `60`), **`duration_high`** (string)* (default: `660`), **`plays_low`** (int)* (default: `0`), **`plays_high`** (int)* (default: `1100000000`), **`danceability_low`** (int)* (default: `0`), **`danceability_high`** (int)* (default: `1`), **`energy_low`** (int)* (default: `0`), **`energy_high`** (int)* (default: `1`), **`valence_low`** (int)* (default: `0`), **`valence_high`** (int)* (default: `1`), explicit* (string)
Route Handler: **`search_players(req, res)`**
Return Type: JSON Array
Return Parameters: [{ **`song_id`** (string), **`album_id`** (string), **`title`** (string), **`number`** (int), **`duration`** (int), **`plays`** (int), **`danceability`** (float), **`energy`** (float), **`valence`** (float), **`tempo`** (int), **`key_mode`** (string), **`explicit`** (int) }, …]
Expected (Output) Behavior:

- Return an array with all songs that match the constraints ordered by song title (ascending). If no song satisfies the constraints, return an empty array without causing an error
- If **`title`** is specified, match all songs with titles that contain the **`title`** query parameter as a substring. If **`title`** is undefined, no filter should be applied (return all songs matching the other conditions).
    - Hint: use LIKE to perform substring matching
    - Hint: although no default value of **`title`** is given, consider what default value would cause an undefined **`title`** to not apply any filter to the search.
- If **`explicit`** is not defined or does not equal “true” then filter out any songs that are labeled as explicit
    - Hint: refer back to our implementation of route 2
- x_high and x_low are the upper and lower bound filters for an attribute x. Entries that match the ends of the bounds should be included in the match. For example, if **`energy_low`** were 0.2 and **`energy_high`** were 0.8, then all players whose **`energy`** attribute was >=0.2 and <=0.8 would be included.

Tasks:

- Task 12 (10 pts): return all songs that match the given search query with parameters defaulted to those specified in API spec ordered by title (ascending)

CIS 550 Spring 2023

Part 2: React Client

(50 points)

Understanding React

Exercise 3, the lecture materials, Recitation, and the official React docs provide some great guidance on this, but here are a few essentials we think you might find helpful. First, you should start the server application that you developed for Part 1 and copied into the modified starter code as mentioned above.

After starting the server application, which should run on port 8080, you should start the React application by running the command npm start within the /client directory in a terminal window and follow along as needed. If there is such a communication issue, you will most likely get an error like '*Unhandled Rejection (TypeError): Failed to fetch*'.

You can safely ignore any warnings (especially about unused components or deprecations). Once the build is running, the terminal output (for the starter code) should look like this:

```
$ npm start

> client@0.1.0 start
> react-scripts start

(node:131290) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning: 'onAfterSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
(Use `node --trace-deprecation ...` to show where the warning was created)
(node:131290) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWarning: 'onBeforeSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
Starting the development server...
Compiled with warnings.

[eslint]
src/components/LazyTable.js
  Line 16:20:  'setPageSize' is assigned a value but never used  no-unused-vars
  Line 38:11:  'newPageSize' is assigned a value but never used  no-unused-vars

src/components/SongCard.js
  Line 3:60:   'RadarChart' is defined but never used            no-unused-vars
  Line 3:72:   'Radar' is defined but never used                 no-unused-vars
  Line 3:79:   'PolarGrid' is defined but never used             no-unused-vars
  Line 3:90:   'PolarAngleAxis' is defined but never used        no-unused-vars
  Line 3:106:  'PolarRadiusAxis' is defined but never used       no-unused-vars
  Line 6:7:    'config' is assigned a value but never used       no-unused-vars
  Line 13:20:  'setSongData' is assigned a value but never used  no-unused-vars
  Line 14:21:  'setAlbumData' is assigned a value but never used  no-unused-vars

src/pages/HomePage.js
  Line 47:9:  'albumColumns' is assigned a value but never used  no-unused-vars
src/pages/SongsPage.js
  Line 17:24:  'setDanceability' is assigned a value but never used  no-unused-vars
  Line 18:18:  'setEnergy' is assigned a value but never used        no-unused-vars
  Line 19:19:  'setValence' is assigned a value but never used       no-unused-vars

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.

WARNING in [eslint]
src/components/LazyTable.js
  Line 16:20:  'setPageSize' is assigned a value but never used  no-unused-vars
  Line 38:11:  'newPageSize' is assigned a value but never used  no-unused-vars

src/components/SongCard.js
  Line 3:60:   'RadarChart' is defined but never used            no-unused-vars
  Line 3:72:   'Radar' is defined but never used                 no-unused-vars
  Line 3:79:   'PolarGrid' is defined but never used             no-unused-vars
  Line 3:90:   'PolarAngleAxis' is defined but never used        no-unused-vars
  Line 3:106:  'PolarRadiusAxis' is defined but never used       no-unused-vars
  Line 6:7:    'config' is assigned a value but never used       no-unused-vars
  Line 13:20:  'setSongData' is assigned a value but never used  no-unused-vars
  Line 14:21:  'setAlbumData' is assigned a value but never used  no-unused-vars

src/pages/HomePage.js
  Line 47:9:  'albumColumns' is assigned a value but never used  no-unused-vars

src/pages/SongsPage.js
  Line 17:24:  'setDanceability' is assigned a value but never used  no-unused-vars
  Line 18:18:  'setEnergy' is assigned a value but never used        no-unused-vars
  Line 19:19:  'setValence' is assigned a value but never used       no-unused-vars

webpack compiled with 1 warning
```

This application, by default, runs on localhost - port 3000. Once you run the above command, your default browser should open up a window to localhost:3000 (you might need to wait a few seconds for it to load).

The client application uses a router to accept a request to the port on a host. For example, this application runs on the host 'localhost' and port 3000. Upon receiving a request from a client (your browser in this case), the application uses a routing library (react-router) that parses the URL string to map it to a registered route (see *index.js*), which, in turn, renders the React page component corresponding to that route. For example, the path http://localhost:3000/albums will render the page component **`AlbumsPage`**.

React expresses elements to be rendered using JSX, JavaScript code that looks like html. Within JSX, you can use curly braces to escape and execute normal JavaScript code. For example, you can define **`const element = <h1>Hi {name}</h1>`** to express a heading with the text "hi [name]" where [name] represents the value of some local variable **`name`** in JavaScript.

We have decided to write this homework using functional components and hooks, which we understand may look unfamiliar to students that have primarily worked with object-oriented languages such as Java. We chose to do this since functional components are now the preferred and primary form of React, and as such most documentation students find online will leverage functional components and hooks. If you have trouble understanding functional components, please do not hesitate to reach out on Ed and/or attend the recitation dedicated to web programming. Here is a short refresher on some essentials:

- Functional components take a properties (props) object as its parameters, defining any necessary (or optional) information required to render the component (e.g. the id of the song to render). Note in the code provided, we destructure (variable unpack) the props parameter for ease of use (i.e. instead of defining a functional component as **`const SongCard = (props) => { … }`**, instead we write **`const SongCard = ({ songId, handleClose }) => { … }`**). Note also the use of lambda functions as we explained earlier in this document.
- Components return a JSX object to be rendered.
- Components maintain information across renders using state variables via the **`useState`** hook. The syntax is **`const [state, setState] = useState("default value")`** where the variable **`state`** holds the persistent data and is modified by calling the function **`setState`**.
- Components can perform "side-effecting" functions using the **`useEffect`** hook. In particular, this hook is used primarily to retrieve data from our backend API. The syntax is to call **`useEffect(callback, deps)`** where **`callback`** is a function performing the side-effect and **`deps`** is an optional array parameter that signals to the hook to perform the side-effect only when certain variables change. More detailed examples are provided in the sample code.
- Components have a **`render`** method that describes how the view is rendered in a browser window. This is also the only required method in a component

One reason why React is widely preferred by many developers is the vast collection of UI component libraries that make it easy for developers to quickly and easily create beautiful interfaces. In this assignment, you will be working with two such libraries–MUI and recharts. A careful look at the starter code (especially the files for pages and components) will give you a

good idea of how these components are imported and used, but we strongly recommend looking into the documentation for a better understanding. You will find the task-specific documentation links we provide in the following section very helpful as well.

An Overview of Tasks

(50 Points)

In this section, we categorically summarize the tasks that you will complete. The specific instructions and hints for these tasks are included in the code, as inline comments, in the form "***TASK x:*** **…",** for example: `// TODO (TASK 13): add a state variable to store the app author (default to '')`. Each of the twelve (12) tasks is relatively short (anywhere from about 1-10 lines on average).

You should look at the following files in **src/pages** to complete the tasks in numerical order, but feel free to deviate from this slightly (especially if you would like to implement tasks by categories - remember that tasks may build on top of each other, though!):

- HomePage.js (Tasks 13 through 17)
  - Task 13 (2 pts): add a state variable to store the app author (default to '')
  - Task 14 (3 pts): add a fetch call to get the app author (name not pennkey) and store it in the state variable
  - Task 15 (3 pts): define the columns for the top albums (schema is Album Title, Plays), where Album Title is a link to the album page
  - Task 16 (3 pts): add a h2 heading, LazyTable, and divider for top albums. Set the LazyTable's props for defaultPageSize to 5 and rowsPerPageOptions to [5, 10]
  - Task 17 (1 pt): add a paragraph (text) that displays the value of your author state variable from TASK 13
- LazyTable.js (Tasks 18 through 19)
  - Task 18 (2 pts): set pageSize state variable and reset the current page to 1
  - Task 19 (4 pts): the next 3 lines of code render only the first column. Modify this with a map statement to render all columns.
- SongCard.js (Tasks 20 through 21)
  - Task 20 (8 pts): fetch the song specified in songId and based on the fetched album_id also fetch the album data
  - Task 21 (6 pts): display the same data as the bar chart using a radar chart
- AlbumsPage.js (Task 22)
  - Task 22 (2 pts): replace the empty object {} in the Container's style property with flexFormat. Observe the change to the Albums page. Then uncomment the code to display the cover image and once again observe the change, i.e. what happens to the layout now that each album card has a fixed width?
- AlbumInfoPage (Task 23)
  - Task 23 (10 pts): render the table content by mapping the songData array to <TableRow> elements
- SongsPage (Task 24)
  - Task 24 (6 pts): add sliders for danceability, energy, and valence (they should be all in the same row of the Grid)

CIS 550 Spring 2023

Category 1: Fetching / Updating Component State

Description: These tasks involve updating the component state to display data, potentially querying and parsing data from the backend API.
Relevant Tasks: 13, 14, 18, 20
Suggested documentation:

- Using fetch (Mozilla docs)
- Async (Mozilla docs)
- Promise (Mozilla docs)
- useState hook (freeCodeCamp)
- useEffect hook (freeCodeCamp)

Category 2: Basic Use of JSX

Description: These tasks involve using basic JSX to display information to the user.
Relevant Tasks: 16, 17
Suggested documentation:

- JSX (React docs)

Category 3: Tables, Rows, Columns (MUI)

Description: These tasks involve using MUI tables to display data.
Relevant Tasks: 19, 23
Suggested documentation:

- Table (MUI docs)
- DataGrid (MUI docs)
- Lazy loading (Mozilla docs)

Category 4: Using Map to Dynamically Display Data

Description: The tasks involve using the map function to dynamically load data which may not have a fixed schema/size.
Relevant Tasks: 19, 23
Suggested documentation:

- Map (Mozilla docs)
- Lists and keys (React docs)

Category 5: Data Visualization (recharts)

Description: These tasks involve using graphs from recharts to visualize data.
Relevant Tasks: 21
Suggested documentation:

- Bar chart (recharts docs)
- Radar chart (recharts docs)

Category 6: Inline Styling and Flexboxes

Description: These tasks involve using inline styling and flexboxes to alter the look and feel and layout of pages.
Relevant Tasks: 22
Suggested documentation:

- Style attribute (react docs)

- Flexboxes (Mozilla docs)

Category 7: Input Components (MUI)

Description: These tasks involve using the MUI input components to gather user input.
Relevant Tasks: 24
Suggested files/documentation:

- Slider (MUI docs)
- Checkbox (MUI docs)

Category 8: Using Grid to Format Pages (MUI)

Description: These tasks involve using the MUI Grid component to format and lay out pages.
Relevant Tasks: 24
Suggested files/documentation:

- Grid (MUI docs)

Submission

Once you are convinced that you have implemented the application (both parts) as per the specifications, submit only the following files to the Homework 2 Gradescope submission:

- routes.js (from Part 1)
- HomePage.js, LazyTable.js, SongCard.js, AlbumsPage.js, AlbumInfoPage.js, SongsPage.js (from Part 2)

Ensure that the files are exactly named as above. Upon submission, the autograder will:

1. Check if all files were correctly submitted
2. Run tests on Part 1 (all test cases will be visible and align with those provided to you in the sample code)

Once the submission deadline has passed, we will grade the frontend component (part 2) manually for 50 additional points