COMP 4513 Assignment #1

Due Friday February 13th at midnight
Version 1.0, January 13 2026

Overview

This assignment provides you with an opportunity to create an API in Node. You will also be required to provision a database on supabase.

The data you have been provided with contains data from Spotify about songs, artists, genres, and artist types (I believe it's hit songs from the years 2016-2019).

Grading

The grade for this assignment will be broken down as follows:

Programming Design and Documentation	15%
Hosting + Correct readme	10%
Functionality (follows requirements)	75%

Database

Data has been provided as an SQLite database and as a variety of csv files. The database consists of the following tables:

- **songs**: Information about songs. Each song has a foreign key indicating its artist and genre so you will need to use `INNER JOINS` where necessary.
- **artists**: Information about individual artists
- **genres**: Information about genres
- **types**: Information about different artist types.
- **playlists**: Information about user-created playlists. In assignment 2, you will be creating and modifying playlists; here there are a handful of sample playlists.

Database Choice

You have two choices as to how you approach the database retrieval in this assignment.

1. Use the provided sqlite database. As you have seen in Lab14b exercises 1-4, sqlite is a file-based SQL database that requires using some type of API for running SQL queries. The lab uses the sqlite3 package. There is also a built-in sqlite package called node:sqlite which you are welcome to use instead if that is your preference.
2. Use a cloud-based DBMS. Lab14b exercises 5-9 uses supabase. The disadvantage to supabase is that it has its own learning curve and you can't "see" or easily modify the SQL generated by the supabase API. The advantages are twofold: a) you can put "practical experience with a cloud database" on your resume, which is actually a pretty desirable attribute in a entry-level developer, and b) when you create your assign2 in react, it will considerably simplify your hosting requirements, since you won't need to maintain an external node host to run your web API; instead, you will be able to use the supabase API directly in your React application (as in exercise 9 in Lab14b).

Submitting and Hosting

You will be using Node in this assignment. This will mean your assignment will need to reside on a working host server. Static hosts (e.g., github pages) will not work.

For this assignment, I would recommend using free hosting option on either render.com or vercel.com (over the past 12 months, many students have told me using render was quite straightforward; on the other hand, getting vercel hosting experience is a nice thing to put in your resume). Do note that these free projects go to sleep after a set period of inactivity, so be aware that the first request of a slept hosted node application will take some time to awaken.

It is possible that there are other node hosting options which are superior to these options. Feel free to use whatever hosting environment you like.

When your hosting is working and the assignment is ready to be marked, then send me an email with the following information:

- The URL of the github repo so that I can mark the source code. If your repo is private, then add me as a collaborator.
- In the `readme.md` file for your repo, provide links to the APIs on glitch/render so I can test them (see details below).

NOTE: Free tier supabase projects will go inactive after one week of inactivity (and thus your API won't work). Please login into supabase and run a test query every few days after the due date so this doesn't happen!!

Recommended Workflow

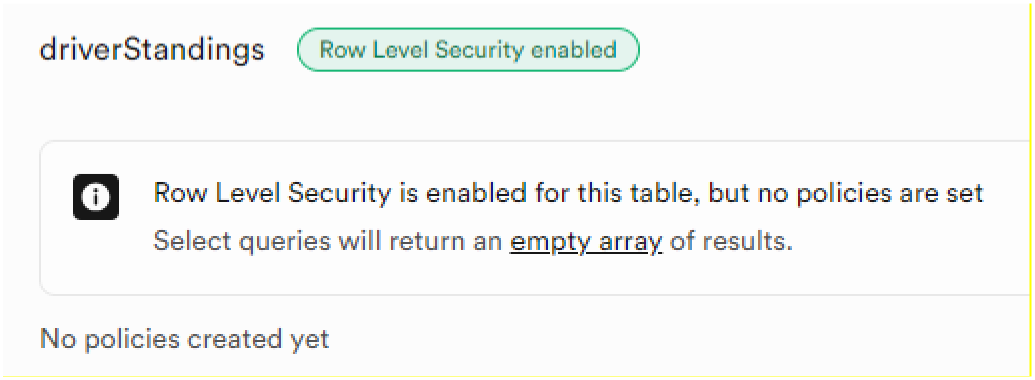
I recommend you approach this assignment in the following order:

1. Complete Lab14b
2. Set up a github repo for your source code.
3. Implement the SQL for each of the API routes using the SQL Editor in supabase or one of the sqlite tools available. Save the SQL query text in a file. While this step is not strictly necessary, it will make step 5 easier.
4. Compare your query results to someone else's query results. Again, this isn't necessary, but it might help you catch errors in your query logic for the more complex queries.
5. Implement the APIs in Express. If you have created the SQL queries first, then you can simply convert them to supabase's query builder syntax or sqlite code.
6. Set up the hosting. The hosting might take longer to set up then you anticipate, so be sure to leave ample time for it. Be willing to post questions on the class discord if you have issues with your hosting; be willing to answer others' questions!
7. Construct a fulsome `readme.md` file in your github repo with the expected example API request links (see page 5).
8. **Test the link APIs in the readme file after hosting!!!**
9. Send me an email with the required info (see page 2).

Common Supabase Problems

1. API returns a blank screen. Evidently error messages are for rookies and not for supabase users. Generally this happens because there is a problem in your SQL (i.e., your field list, your filter, or your modifier). Fix it! Check for field names being incorrectly spelled or that have the wrong case. Are you including a field from a table in which relations haven't been set? **Look for trailing commas as they will mess it up.**

2. API returns an empty array. You likely haven't set a row-level security policy that allows read only access for one of the tables. For instance (ignore the table name, this is from last year):



driverStandings Row Level Security enabled

Row Level Security is enabled for this table, but no policies are set
Select queries will return an empty array of results.

No policies created yet

3. You are getting results back from supabase that shouldn't be returned based on the filters. For instance, instead of a full object, you will see a **null** value instead. You will see this when you are trying to use a filter on a joined table, e.g., from last year's assignment (different data)

```
.select(`raceId,year, circuits (circuitRef,name)`)
.eq('circuits.circuitRef','monza')
```

   Strangely (at least to me), supabase seems to default to a FULL JOIN rather than an INNER JOIN in this situation. For instance, let's say from last last year's assignment, you want to see races and circuits data for races at monza. When it does a FULL JOIN, you will get not only races at monza, but all races not at monza. But because the filter is in place, the races not at monza will have a circuits value of null.

   To fix this, you need to tell supabase to make the join an inner join via:
```
.select(`raceId,year, circuits!inner (circuitRef,name)`)
```

4. Your sort isn't working or is resulting in the dreaded blank screen. To sort on a field in a linked table, you have to add in a `referencedTable` property, e.g.,
```
.order('year', { referencedTable: 'races', ascending: false })
```

API Functionality

You must create the following APIs with the specified routes and functionality. The returned data must be JSON format.

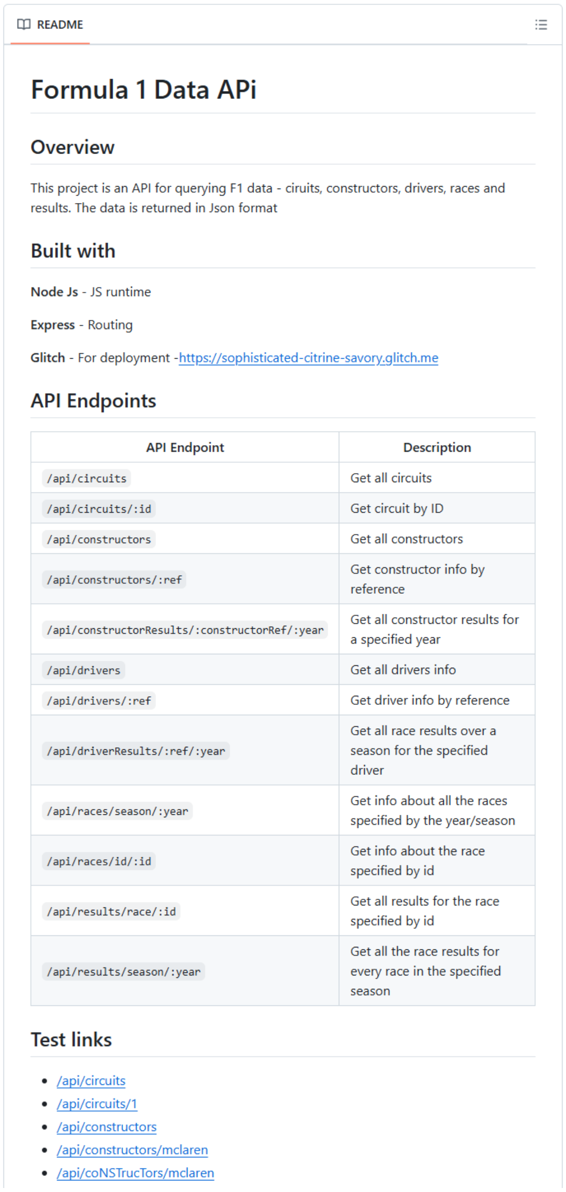
`/api/artists`	Returns all data for all artists sorted by artist_name. For the returned data for all the artist routes, don't provide the foreign keys for the artist_type; instead provide the following fields: types (`type_name`)
`/api/artists/`*ref*	Returns just the specified artist (use the `artist_id` field), e.g., `/api/artists/129`
`/api/artists/averages/`*ref*	Returns the average values for bpm, energy, danceability,loudness,liveness,valence,duration, acousticness, speechineess, popularity for the specified artist
`/api/genres`	Returns all the genres
`/api/songs`	Returns all data for all the songs sorted by song title. For the returned data for all the songs routes, don't provide the foreign keys for the artist and genre; instead provide the following fields: artist (`artist_id`, `artist_name`), genre (`genre_id`, `genre_name`)
`/api/songs/sort/`*order*	Returns all the songs sorted by order field. Possible values are: id, title, artist (name), genre (name), year, duration
`/api/songs/`*ref*	Returns just the specified song (use the `song_id` field), e.g., `/api/songs/1028`
`/api/songs/search/begin/`*substring*	Returns the songs whose title (case insensitive) **begins** with the provided substring, e.g., `/api/songs/search/begin/lov`
`/api/songs/search/any/`*substring*	Returns the songs whose title (case insensitive) **contains** the provided substring, e.g., `/api/songs/search/any/lov`
`/api/songs/search/year/`*substring*	Returns the songs whose year is equal to the provided substring, e.g., `/api/songs/search/year/2017`
`/api/songs/artist/`*ref*	Returns all the songs for the specified artist, . e.g., `/api/songs/artists/129`
`/api/songs/genre/`*ref*	Returns all the songs for the specified genre, . e.g., `/api/songs/genre/115`

`/api/playlists/ref`	Returns all the songs for the specified playlist, . e.g., `/api/playlists/3` Return the following fields: playlist, song_id, title, artist name, genre name, year
`/api/mood/dancing/ref`	Returns the top number (determined by ref parameter) of songs sorted by danceability parameter in descending order. For this and the other mood routes, include the same data as the other songs routes. If the number is missing, is less than 1 or greater than 20, then default to 20.
`/api/mood/happy/ref`	Returns the top number (determined by ref parameter) of songs sorted by valence parameter in descending order.
`/api/mood/coffee/ref`	Returns the top number (determined by ref parameter) of songs sorted by liveness divided by acousticness in descending order.
`/api/mood/studying/ref`	Returns the top number (determined by ref parameter) of songs sorted by the product of the energy and speechiness parameters in ascending order.

For each of the requests that take parameters, your API needs to handle a Not Found condition. For instance, if a ref or year doesn't exist, **return a JSON-formatted error message** that indicates the requested request did not return any data. For the routes with a start and end year, provide a different error message if the end year is earlier than the start year.

GitHub Readme

Remember that potential employers will look at your github repos and will want to see best practices. That is, think of your github repo readme files as if they were part of your resume/portfolio. Here are some excellent sample github readmes from other classes that demonstrate the level of detail that you may want to emulate.



README

Formula 1 Data APi

Overview

This project is an API for querying F1 data - ciruits, constructors, drivers, races and results. The data is returned in Json format

Built with

Node Js - JS runtime

Express - Routing

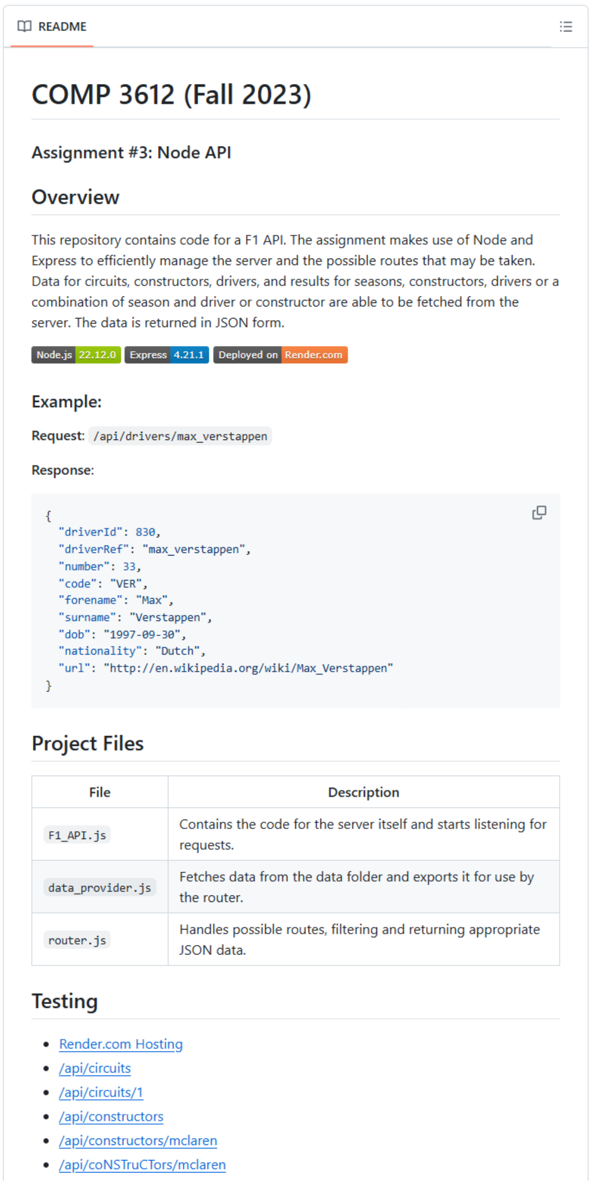
Glitch - For deployment -https://sophisticated-citrine-savory.glitch.me

API Endpoints

API Endpoint	Description
`/api/circuits`	Get all circuits
`/api/circuits/:id`	Get circuit by ID
`/api/constructors`	Get all constructors
`/api/constructors/:ref`	Get constructor info by reference
`/api/constructorResults/:constructorRef/:year`	Get all constructor results for a specified year
`/api/drivers`	Get all drivers info
`/api/drivers/:ref`	Get driver info by reference
`/api/driverResults/:ref/:year`	Get all race results over a season for the specified driver
`/api/races/season/:year`	Get info about all the races specified by the year/season
`/api/races/id/:id`	Get info about the race specified by id
`/api/results/race/:id`	Get all results for the race specified by id
`/api/results/season/:year`	Get all the race results for every race in the specified season

Test links

- /api/circuits
- /api/circuits/1
- /api/constructors
- /api/constructors/mclaren
- /api/coNSTrucTors/mclaren



README

COMP 3612 (Fall 2023)

Assignment #3: Node API

Overview

This repository contains code for a F1 API. The assignment makes use of Node and Express to efficiently manage the server and the possible routes that may be taken. Data for circuits, constructors, drivers, and results for seasons, constructors, drivers or a combination of season and driver or constructor are able to be fetched from the server. The data is returned in JSON form.

Node.js 22.12.0 | Express 4.21.1 | Deployed on Render.com

Example:

Request: `/api/drivers/max_verstappen`

Response:

```
{
  "driverId": 830,
  "driverRef": "max_verstappen",
  "number": 33,
  "code": "VER",
  "forename": "Max",
  "surname": "Verstappen",
  "dob": "1997-09-30",
  "nationality": "Dutch",
  "url": "http://en.wikipedia.org/wiki/Max_Verstappen"
}
```

Project Files

File	Description
`F1_API.js`	Contains the code for the server itself and starts listening for requests.
`data_provider.js`	Fetches data from the data folder and exports it for use by the router.
`router.js`	Handles possible routes, filtering and returning appropriate JSON data.

Testing

- Render.com Hosting
- /api/circuits
- /api/circuits/1
- /api/constructors
- /api/constructors/mclaren
- /api/coNSTruCTors/mclaren

Example API Requests

In the `readme.md` file for your assignment repo, you must supply this list of links to allow me to test everyone's APIs fairly and consistently. Please add the following test links (**they must be clickable links**) in this file:

`/api/artists`	`/api/playlists`
`/api/artists/129`	`/api/playlists/3`
`/api/artists/sdfjkhsdf`	`/api/playlists/35362`
`/api/artists/averages/129`	`/api/mood/dancing/5`
`/api/genres`	`/api/mood/dancing/500`
`/api/songs`	`/api/mood/dancing/ksdjf`
`/api/songs/sort/artist`	`/api/mood/happy/8`
`/api/songs/sort/year`	`/api/mood/happy`
`/api/songs/sort/duration`	`/api/mood/coffee/10`
`/api/songs/1010`	`/api/mood/studying/15`
`/api/songs/sjdkfhsdkjf`	
`/api/songs/search/begin/love`	
`/api/songs/search/begin/sdjfhs`	
`/api/songs/search/any/love`	
`/api/songs/search/year/2017`	
`/api/songs/search/year/2027`	
`/api/songs/artist/149`	
`/api/songs/artist/7834562`	
`/api/songs/genre/115`	

Note: you will need to preface the above URLs with the URL of your host. For instance, if your Glitch URL is `https:/smashing-squirrels.glitch.me`, then the URL for the second test link would be `https:/smashing-squirrels.glitch.me/api/songs/1010`