LAB 14b

Working With SQL Databases (Node + JS)

What You Will Learn

- How to read data from SQLite in Node
- How to use Supabase to create a cloud-based Postgres database
- Accessing Supabase in Node and browser JS

Note

This chapter's content has been split into three labs: Lab14a, Lab14b, Lab14c.

Approximate Time

The exercises in this lab should take approximately 45 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: January 21, 2024
Revisions: renamed sqlite files, added supabase, moved MongoDB to Lab14c

Preparing Directories

1. The starting `lab14b` folder has been provided for you (within the zip folder downloaded from Gumroad).

Working with SQLite

In this lab, you will be making use of different data sources in Node. To begin, you will use SQLite, a small relational database management system. Unlike more familiar database systems such as MySQL or Oracle, SQLite is a database engine built into and used by mobile apps, desktop applications, and, of course, Node on the server. SQLite is widely used (but hidden from the user) within applications such as cell phones, TVs, browsers, PCs, and Macs. According to the `https://www.sqlite.org` website, there are likely billions of SQLite databases in active use, making it by far the most used database engine on the planet.

A SQLite database is contained within a very small compressed single file. The drawback with the single file approach is that it can't handle numerous multiple simultaneous INSERT/UPDATE queries. But for small Node web apps (or ones that are mostly read-only with a low number of write requests), SQLite is fantastic since no other software is required on the server. With Node, you will simply add the sqlite3 package to your application.

Exercise 14b.1 — Setting Up SQLite

1. You do not need to install anything on your development machine or your production server to use SQLite. Nonetheless, you may find it convenient to install some of the SQLite tools on your development machine (though, again, there are not necessary) if you want to test some queries, modify the database, etc. There is a command-line shell program and an excellent GUI program named SQLite Studio.

2. If you do install any of these tools, you can examine the provided SQLite databases (`art.db`) in the `data` folder.

Exercise 14b.2 — Setting Up SQLite-Based Node App

1 In the terminal, make sure the current folder is where you want to create the node application. If it is, then type the following command:

```
npm init
```

It doesn't really matter how you answer these questions.

2 Enter the following commands:

```
npm install express
npm install sqlite3
```

Exercise 14b.3 — Using SQLite in Node

1 Create a new file named `lite-tester.js`.

2 Add the following code:

```
const path = require("path");
// the verbose is optional but gives better error messages
const sqlite3 = require("sqlite3").verbose();
```

3 Add the following code:

```
// open the database
const DB_PATH = path.join(__dirname, "data/art.db");
const db = new sqlite3.Database(DB_PATH);
```

If you copied the starting code, you should have a folder named data with the sqlite database file in it.

4 Add the following code:

```
let sql = `SELECT GenreID,GenreName,EraID,Description,Link
            FROM Genres;`;
// retrieve all the data into memory
db.all(sql, [], (err, rows) => {
    if (err) {
        throw err;
    }
    rows.forEach( genre => {
        console.log(genre.GenreName);
    });
});
// close the database
db.close();
```

Because of the asynchronous nature of Node, a callback function must be passed to the `all()` method: this callback will be passed all the retrieved data in an array.

5 Save and test the page. This time, do this using the `node` command:

```
node lite-tester.js
```

No browser is needed: this program runs (and then exits) entirely in the command window/terminal.

6 Add in the following code before closing the database:

```
// only put a row at a time into memory
sql = `SELECT ArtistID,FirstName,LastName
            FROM Artists WHERE NATIONALITY=? ;`;
const params = ['France'];
db.each(sql, params, (err, artist) => {
    if (err) {
        throw err;
    }
    console.log(`${artist.FirstName} ${artist.LastName}`);
});
// close the database
db.close();
```

Instead of reading the entire table into memory (which would be very memory intensive if the table was large), the callback gets called for each record.

7 Save the file. Stop the previous Node process and re-run.

8 Add in one more example and test:

```
// now get just a single record
sql = `SELECT PaintingID,Title
              FROM Paintings where PaintingID=?;`;
db.get(sql, [501], (err, painting) => {
    if (err) {
        throw err;
    }
    console.log('**** ' + painting.Title);
});
// close the database
db.close();
```

When you run this, you may find that this third query finishes before the artist query because it is simpler.

Exercise 14b.4 — Creating an API using Sqlite

1 Open the provided file named `lite-api-art.js` and add the following:

```
app.use(express.json());
const provider = require('./scripts/painting-provider.js');

// root endpoint will retrieve all paintings
app.get("/", (req, resp) => {
    provider.retrievePaintings(req, resp);
});

// this endpoint will retrieve single painting
app.get("/:id", (req, resp) => {
    provider.retrieveSinglePainting(req, resp);
});
```

2 Open the provided `scripts/painting-provider.js` and add the following:

```
const path = require("path");
const sqlite3 = require("sqlite3").verbose();

const DB_PATH = path.join(__dirname, "../data/art.db");
const db = new sqlite3.Database(DB_PATH);
```

So you don't have so much tedious typing, the SQL statement and the conversion function to JSON has been provided.

3 Add the following function after the provided comment:

```
// Retrieve all paintings
const retrievePaintings = (req, resp) => {
    db.all(sql, [], (err, rows) => {
        if (err) {
            throw err;
        }
        console.log('getting all paintings...');
        const paintings =
               rows.map(row => convertRecordToJson(row));
        resp.json( paintings );
    });
};
```

4 Add the following function after the provided comment.

```
// retrieve just a single painting based on the id
const retrieveSinglePainting = (req, resp) => {
    let mySQL = sql + " WHERE PaintingID=?";
    db.get(mySQL, [req.params.id], (err, row) => {
        if (err) {
            throw err;
        }
        console.log('getting single painting...');
        resp.json( convertRecordToJson(row) );
    });
};
```

5 Examine the provided function `convertRecordToJson`.

If you are implementing a JSON API using data retrieved from a SQL database, you may need to manually convert the record data into the nested JSO required for the API.

6 Run `lite-api-art.js` and test in the browser via these two requests:

`http://localhost:8080/290`

This will retrieve just a single painting.

`http://localhost:8080/`

This will retrieve all paintings.

Working With Supabase

One recent trend in web development has been using cloud-based database management systems such as firebase, supabase, planetscale, and neon. In this lab, you will use supabase, which has gained a lot of interest and activity amongst the developer community in 2023. It also has a generous free option that is ideal for students.

Supabase allows you to use a web interface to create, manage, view, and edit Postgres databases (which has by-passed MySQL as the most popular relational database system).

Exercise 14b.5 — Setting up a Supabase Database

1 This exercise describes how to setup and populate a Postgres database hosted on supabase.

To do so, you will need to create an account on `supabase.com`. Be sure to select the Free option!

NOTE: the web-based user interface for supabase is very much subject to change. The screens you see may look different than the screen captures provided here in this lab.

2 After creating your account, you will see a screen that allows you to create a new project. Feel free to give your project any name. I named mine `f1`.

3 Click on the Table Editor option and click the **Create a new table** button.

You should see the Create option on a screen similar to Figure 14b.1. You won't have any existing tables yet (the screen capture shows three existing tables; you will create these in the next several steps).

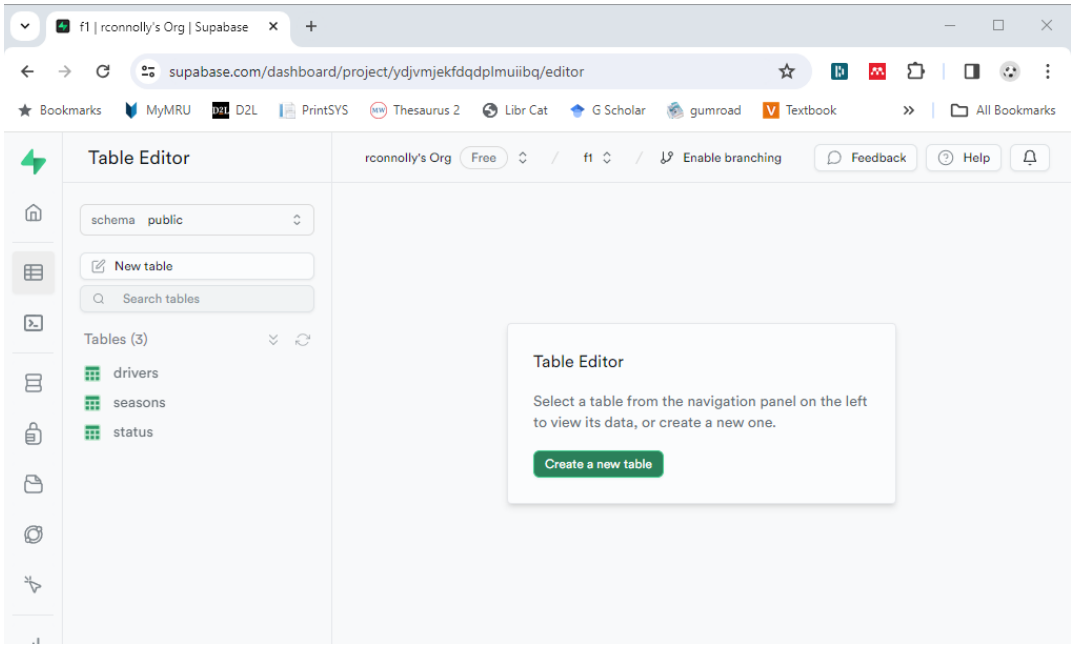


Figure 14b.1 – Supabase Step 1: Table Editor

4 Name your new table **status** and click the **Import data via spreadsheet** button as shown in Figure 14b.2.

When you create a new table, you can define the fields manually. Instead, you will be defining and populating your table from the provided CSV tables.

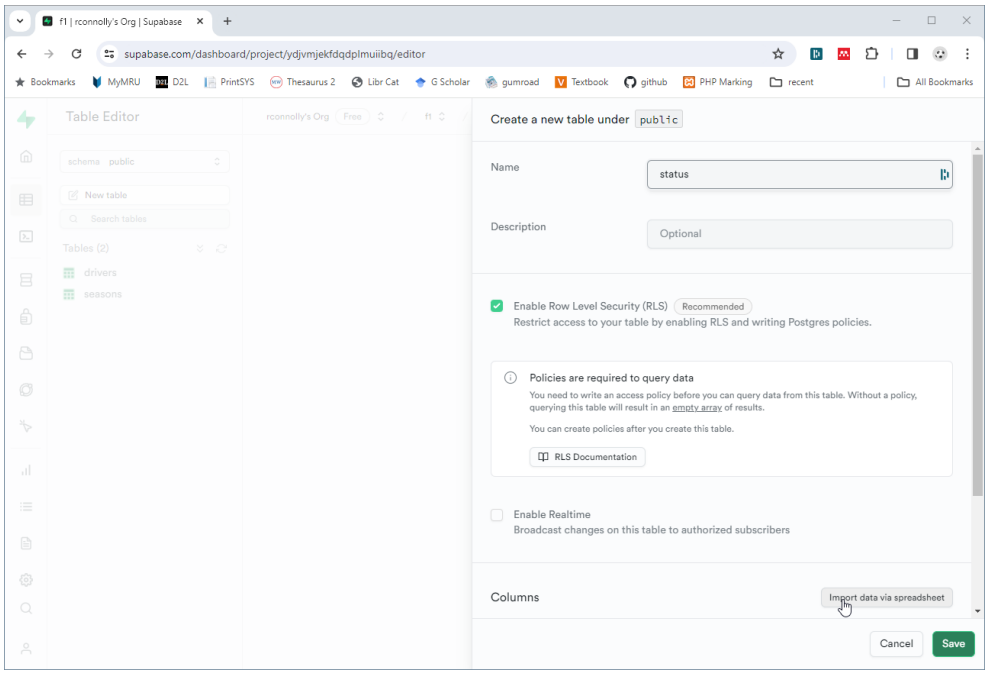


Figure 14b.2 – Supabase Step 2: Create a new table

4 Drag and drop the csv file `status.csv` from the provided `f1` folder onto the drag-and-drop location on the page.

Each of the csv files will be turned into a database table. This database of Formula 1 racing data comes from `https://ergast.com/mrd/`. It has an Attribution-NonCommercial-ShareAlike 3.0 Unported Licence, which means you can use it for personal, non-commercial applications and services including educational and research purposes. The ERD and description of each table have also been provided.

5 You will see a preview of the file content, as shown in Figure 14b.3. Click the Save button.

6 Scroll down until you see the Columns. You should see a message that no primary keys are selected. To fix, turn on the primary check box and then click on the gear icon to the right of the first field (year) and then turn on Is Unique, as shown in Figure 14b.4. Click Save.

The table should import without any errors. You now have created and populated your first Postgres table.

7 After creating a table, you will have to click on Table Editor button to see the Create a new table button. Import the following tables: `seasons`, `drivers`, `constructors`, and `circuits`.

8 Repeat for the tables: `races`, `qualifying`, `results`, `driverStandings`, `constructorStandings`, and `ConstructorResults`.

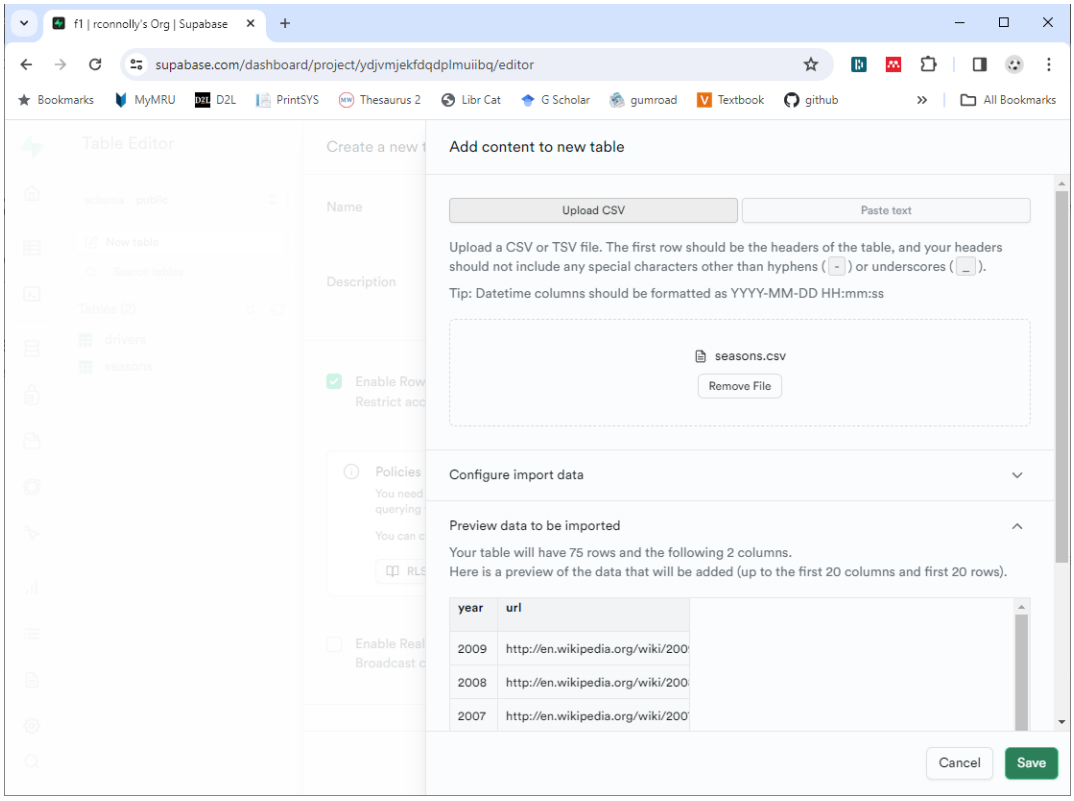


Figure 14b.3 – Supabase Step 3: Saving the table

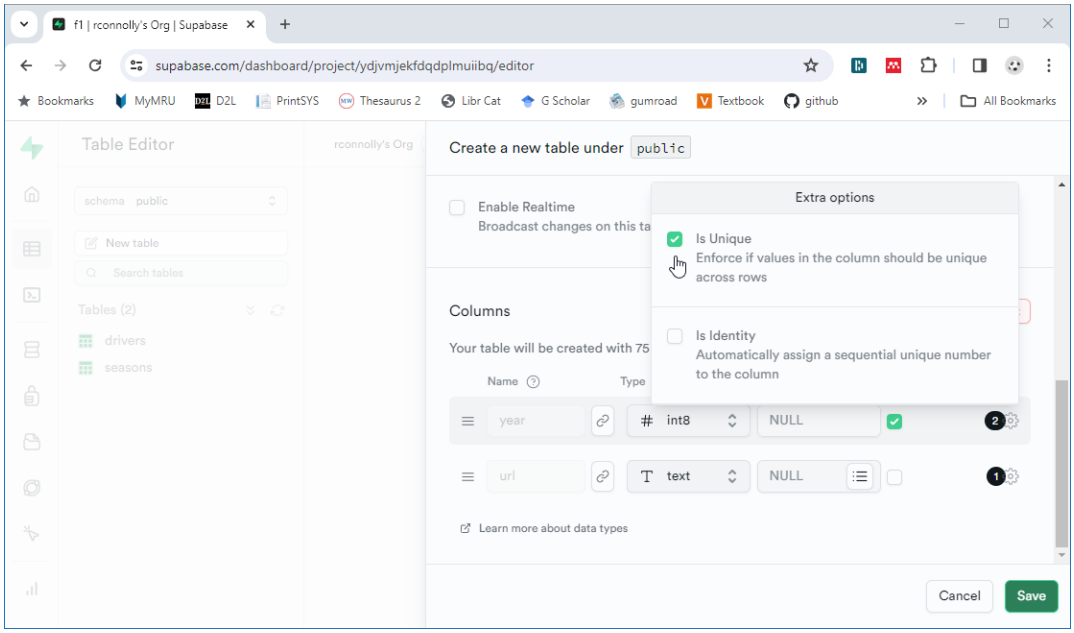


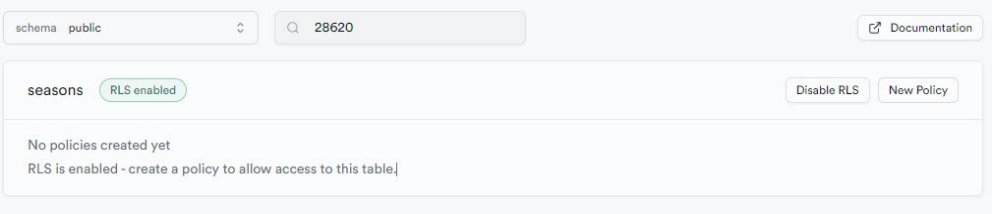
Figure 14b.4 – Supabase Step 4: Editing the columns

Exercise 14b.6 — Setting RLS Policies

1 Take another look at Figure 14b.5. Notice that it says policies are needed to query data (otherwise you will get an empty array from every query.

You have two options: One is to disable Row Level Security (RLS) for each table. While this will work, a better option is to instead define a RLS policy for each table in which you specify who can view or modify a record.

2 In the Table Editor, click on the **status** table. This should display its data. Just above the column names, you should see a button or link that says **No active RSL policies**. Click on it to set up the RSL policies.



3 Click on the **New Policy** button. Choose the **Get started quickly** option.

4 Choose the **Enable read access to everyone** option and then click the **Use this template** button, as shown in Figure 14b.5. The Adding new policy dialog will show your selection. Click Review button and then the Save policy button.

5 Great news: you get to do this for all the other tables in your database! To save some time, you could do this just for the `seasons`, `drivers`, `constructors`, `circuits`, `results` and `races` tables.

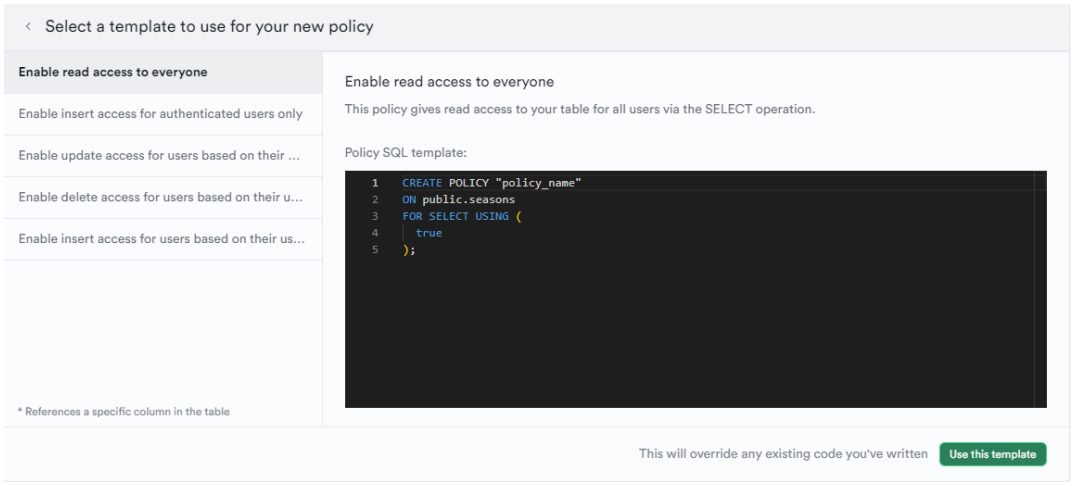


Figure 14b.6 – Setting the RLS policy

Exercise 14b.7 — Querying Supabase in Node

1 Enter the following commands:

```
npm install @supabase/supabase-js
```

2 Create a new file named `f1-server.js` and add the following content.

```
const express = require('express');
const supa = require('@supabase/supabase-js');
const app = express();

const supaUrl = '<YOUR_PROJECT_URL>';
const supaAnonKey = '<YOUR_ANON_API_KEY>';

const supabase = supa.createClient(supaUrl, supaAnonKey);
```

You will need to supply your own project URL and Anon Api key. You can find yours in the Project Home, as shown in Figure 14b.

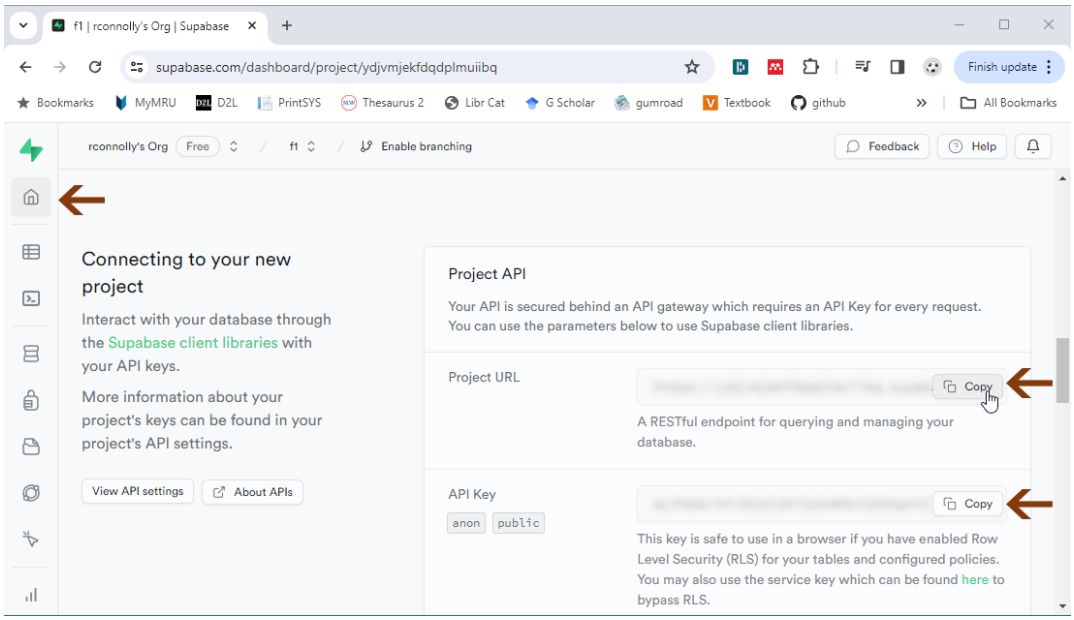


Figure 14b.7 – Finding the Project URL and API key.

3 Add the following route.

```
app.get('/f1/status', async (req, res) => {
   const {data, error} = await supabase
       .from('status')
       .select();
   res.send(data);
});
```

Rather than constructing SQL statements and "sending" them to the DBMS, you instead make use of a query builder API.

4 Add the following.

```
app.listen(8080, () => {
   console.log('listening on port 8080');
   console.log('http://localhost:8080/f1/status');
});
```

5 Test by running `node f1-server` and then requesting `http://localhost:8080/f1/status`

This should display the content from the status table in JSON format.

6 Add the following route.

```
app.get('/f1/seasons', async (req, res) => {
   const {data, error} = await supabase
       .from('seasons')
       .select();
   res.send(data);
});
```

So what is this code doing? It is using supabase's API to run the following query: `SELECT * FROM seasons`.

7 Modify the following and test the new route.

8 Add the following route and test.

```
app.get('/f1/races', async (req, res) => {
   const {data, error} = await supabase
       .from('races')
       .select(`
         raceId, year, round, circuitId, name
       `);
   res.send(data);
});
```

Here we are specifying only these five fields.

9 Modify this route by adding a **filter** and a **modifier** and then test.

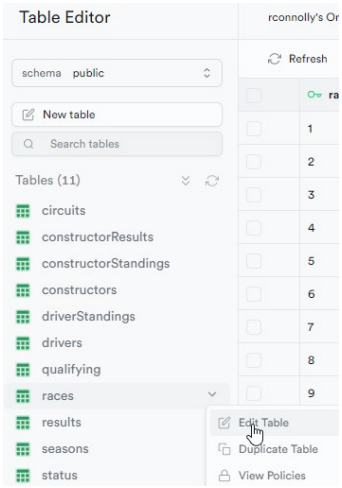
```
   const {data, error} = await supabase
         .from('races')
         .select(`
         raceId, year, round, circuitId, name
         `)
         .eq('year',2020)
         .order('round', { ascending: false });
```

This is equivalent to "WHERE year=2020 ORDER BY round DESC" in SQL. For more information on filters and modifier, see `https://supabase.com/docs/reference/javascript/using-filters` and `https://supabase.com/docs/reference/javascript/using-modifiers`.

Exercise 14b.8 — Defining and Using Relations

1 Examine the provided ERD diagram. You will notice that many of the tables contain foreign keys. For instance, the circuits and races tables exist in a one-to-many relationship. If we define those foreign key relationships, then we can perform multi-table queries: supabase will set up the SQL joins for us.

2 Click on Table Editor. Click on the `races` table, and from the drop-down, click on Edit Table from the context menu.



3 Scroll down until you see the columns. Click on the **Edit foreign key relations** button besides the field `circuitID` as shown in Figure 14b.8.

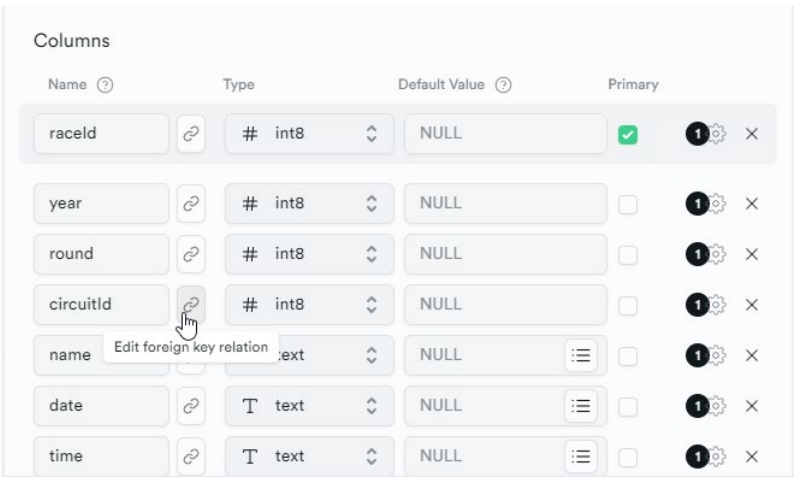


Figure 14b.8 – Defining foreign key relationships

4 Choose the table `circuits` as the reference table and `circuitID` as the reference column. Set the actions for updating and deleting rows to **Cascade**. Save these changes.

5 Now modify the route from the previous exercise as follows and test.

```
.select(`
   raceId, year, round, name, circuits (name,location,country)
`)
```

This adds three fields from the circuits table. Notice that the data from the joined table appears nested.

6 Set up the relationships for the `results` table (i.e., to `races`, `drivers`, and `constructors`).

7 Add the following new route:

```
app.get('/f1/results/:race', async (req, res) => {
   const {data, error} = await supabase
         .from('results')
         .select(`
            resultId, positionOrder, races (year, name),
            drivers (forename,surname), constructors (name)
         `)
         .eq('raceId',req.params.race)
         .order('positionOrder', { ascending: true });
   res.send(data);
});
```

Notice that we are specifying the race results using a parameter.

8 Test using the following URL: `http://localhost:8080/f1/results/1034`.

Exercise 14b.9 — Consuming Supabase in the Browser

1 You can also use cloud-based databases such as supabase in the browser as well as on the server. Examine `vanilla-tester.html`.

Your code will eventually populate the lists from your supabase database. Notice the external supabase JavaScript library.

2 Add the following code to `vanilla-tester.js`.

```
const { createClient } = supabase;

const supaUrl = '<YOUR_PROJECT_URL>';
const supaAnonKey = '<YOUR_ANON_API_KEY>';

const db = createClient(supaUrl, supaAnonKey);
fetchRaceData(2020);
```

3 Continue by adding the following:

```
async function fetchRaceData(year) {
    // uses the same API as the Node examples
    const { data, error } = await db.from('races')
              .select('*')
              .eq('year',year)
              .order('round', { ascending: true });;

    if (error) {
      console.error('Error fetching data:', error);
      return;
    }
    // populate first unordered list
    const first = document.querySelector("#first");
    for (let d of data) {
      const li = document.createElement("li");
      li.textContent = d.name;
      li.value = d.raceId;
      first.appendChild(li);
    }
}
```

4 Test by viewing `vanilla-tester.html` in the browser.

You should see a list of 2020 races.

5 Add a click event listener for the race items in the first column.

```
for (let d of data) {
  const li = document.createElement("li");
  li.textContent = d.name;
  li.value = d.raceId;
  first.appendChild(li);

  li.addEventListener('click', e => {
    document.querySelector("#title").textContent =
       "Results for " + e.target.textContent;
    fetchResultsData(e.target.value);
  })
}
```

6 Continue by adding the following and then test:

```
async function fetchResultsData(raceid) {
  const { data, error } = await db.from('results')
        .select(`
          resultId, positionOrder, drivers (forename,surname),
          constructors (name)
         `)
        .eq('raceId',raceid)
        .order('positionOrder', { ascending: true });

  if (error) {
    console.error('Error fetching data:', error);
    return;
  }

  const second = document.querySelector("#second");
  second.innerHTML = "";
  for (let d of data) {
      const li = document.createElement("li");
      li.textContent = `${d.positionOrder}:
        ${d.drivers.forename} ${d.drivers.surname}
        [${d.constructors.name}]`;
      second.appendChild(li);
  }
}
```

This will populate the second column with the results for the selected race.

Test Your Knowledge #1

1 Add the following routes to `f1-tester.js`. You will need to also set up foreign key relationships for the `qualifying` table.

2 Route 1: qualifying from a specific `raceId` (e.g., http://localhost:8080/f1/qualifying/1034). You have been provided with a sample data record:


```
{
    "qualifyId": 8438,
    "position": 1,
    "q1": "1:25.900",
    "q2": "1:25.347",
    "q3": "1:24.303",
    "races": {
        "name": "British Grand Prix",
        "year": 2020
    },
    "drivers": {
        "surname": "Hamilton",
        "forename": "Lewis"
    },
    "constructors": {
        "name": "Mercedes"
    }
},
```


The data should be sorted by `position` in ascending order.

3 Route 2: races (i.e., all fields in the `races` table) between two years inclusive (e.g., http://localhost:8080/f1/races/2020/2022). This example would include all races in 2020, 2021, and 2022. The data should be sorted by `year` in ascending order.

4 Route 3: drivers who surname begins with the provided value (e.g., http://localhost:8080/f1/drivers/name/sch/limit/12 would return 12 drivers whose surname begins with sch). The data should be sorted by `surname` in ascending order and be limited to the provided number.

5 Add some sample requests of these three routes via `console.log`.