LEARN NODE SQLITE

Introduction

One of the most essential skills as a programmer is being able to identify and utilize the appropriate tool for a specified task. In the context of database management, this will mean using SQL to specify, store, update and retrieve data. In the context of web programming, this will mean writing JavaScript to automate, manipulate, and return relevant values — for presentation in a website or use in a backend script. What happens, then, when we need both? What if we want to retrieve data from a SQL database (using our database administration skills) and then manipulate and expose that data through JavaScript functions (using our web programming skills)?

In this lesson, we will learn how to manage an SQLite database from within JavaScript. We will see how to perform all the fundamental features of database management — `CREATE`ing `INSERT`ing and `SELECT`ing, and then interacting with that data using the full force of JavaScript — writing functions, wielding objects, and performing calculations. It's important to know that many of the results herein could be obtained purely through SQL or purely through JavaScript if need be. But something simple to perform (and read back) with one language might be very hard to write and understand in another.

In the workspace, there's code that opens a connection to an SQLite database. There's a function `getAverageTemperatureForYear()` that will take a `year` as an argument. The function retrieves the temperatures from that year and then calculates the year's average. We've called it with different years, illustrating the power of being able to power our SQL queries with JavaScript.

Instructions

1.

Try the code written in the editor, pass different years to the function and observe the output. The data in the TemperatureData table spans years from the mid 1800s to about 2004, but it is a small data set representing only a few recording stations, so don't take any average temperature data from this data set as representative of the real world average temperature.

2.

Press "Next" when you're ready to introduce SQL to your JavaScript!

app.js

```
const { printQueryResults } = require('./utils');

const sqlite3 = require('sqlite3');

const db = new sqlite3.Database('./db.sqlite');

const getAverageTemperatureForYear = year => {
  if (!year) {
    console.log('You must provide a year!');
    return;
  }
  db.get('SELECT year, AVG(temp_avg) as average_temperature from TemperatureData
WHERE year = $year',
   { $year: year },
   (err, row) => {
    if (err) {
      throw err;
    }
    printQueryResults(row);
  })
}

// Call this function with a few years to view the average temperature that year
// This database has values from 1851 - 2004
getAverageTemperatureForYear(1914)
getAverageTemperatureForYear(1939)
getAverageTemperatureForYear(1977)
```

utils.js

```
const calculateAverages = obj => {
  let result = []
  for (key in obj) {
    const average = obj[key].reduce((accum, curr) => accum + curr) / obj[key].len
gth
    result.push({
      year: key,
      temperature: average,
    });
```

```
  }
  return result;
}

const extractKeys = array => {
  if (array.length === 0) {
    return [];
  }
  let keys = Object.keys(array[0]);
  let isCopy = array.every(element => {
    if (typeof element !== 'object') {
      throw new Error(`Array must be made entirely of objects`);
    }
    return Object.keys(element).every(key => keys.indexOf(key) !== -1);
  })
  if (isCopy) {
    return keys;
  } else {
    throw new Error(`Array's object's keys do not match`);
  }
}

const printQueryResults = array => {
  if (typeof array !== 'object') {
    console.log(array);
    return array;
  }
  if (!Array.isArray(array)) {
    array = [array]
  }
  let keys = extractKeys(array);
  let output = [];
  output.push(`\t${keys.join('\t')}`);
  array.forEach(row => {
    output.push(`\t${Object.keys(row).map(key => row[key]).join('\t')}`);
  });
  output = output.join('\n');
  console.log(output);
  return output;
}
```

```
const addClimateRowToObject = (row, obj) => {
  if (!obj[row.year]) {
    obj[row.year] = [];
  }
  obj[row.year].push(row.temp_avg);
  return obj;
}

const logNodeError = error => {
  if (error) {
    throw error;
  }
}

module.exports = {
  calculateAverages,
  printQueryResults,
  addClimateRowToObject,
  logNodeError
}
```

```
year    average_temperature
1914    20.526666666666667
year    average_temperature
1939    19.13611111111111
year    average_temperature
1977    20.768333333333334
```

Opening A Database

Throughout this lesson, we're going to access an SQLite database with temperature data for countries over the last 150 years. We're going to take this data, collect it per year in a JavaScript object, average it, and save it into a new SQL database!

To get these two worlds to communicate, we will be importing a package into our JavaScript code. This package will allow us to open the channels of

communication with our SQLite database. Once we do that, we can start writing SQL directly in our JavaScript!

The first order of business is to import the module that will facilitate this connection. Recall that to import a module in JavaScript we can use `require()` like so:

```
const sqlite3 = require('sqlite3');
```

The code above gives us a JavaScript object, called `sqlite3` that we can interact with via *methods*. The first method we're going to use on `sqlite3` is going to be the method that opens up a new database. In SQLite, a database corresponds to a single file, so the only argument required to open this database is the *path* to the file that SQLite will use to save the database.

```
const db = new sqlite3.Database('./db.sqlite');
```

This code will create a new database file, in the current folder, named `db.sqlite`. Then we'll have a database to interact with!

Instructions

1.

Require the `sqlite3` package and save it in a `const` named `sqlite3`

Hint

`const sqlite3 = require('sqlite3');`

2.

Open the database by invoking a `new` `sqlite3.Database` and providing the path to your database file (`db.sqlite`). Assign this to the variable `db`.

Hint

Your database is located in the same directory, so you can open it and assign it with the command `new sqlite3.Database('./db.sqlite');`

app.js

```
const { printQueryResults } = require('./utils');
// require the 'sqlite3' package here
const sqlite3 = require('sqlite3');

// open up the SQLite database in './db.sqlite'
const db = new sqlite3.Database('./db.sqlite');


db.all('SELECT * FROM TemperatureData ORDER BY year', (error, rows) => {
  if (error) {
```

```
    throw error;
  }
  printQueryResults(rows);
});
```

utils.js

```
const calculateAverages = obj => {
  let result = []
  for (key in obj) {
    const average = obj[key].reduce((accum, curr) => accum + curr) / obj[key].len
gth
    result.push({
      year: key,
      temperature: average,
    });
  }
  return result;
}

const extractKeys = array => {
  if (array.length === 0) {
    return [];
  }
  let keys = Object.keys(array[0]);
  let isCopy = array.every(element => {
    if (typeof element !== 'object') {
      throw new Error(`Array must be made entirely of objects`);
    }
    return Object.keys(element).every(key => keys.indexOf(key) !== -1);
  })
  if (isCopy) {
    return keys;
  } else {
    throw new Error(`Array's object's keys do not match`);
  }
}

const printQueryResults = array => {
  if (typeof array !== 'object') {
```

```
    console.log(array);
    return array;
  }
  if (!Array.isArray(array)) {
    array = [array]
  }
  let keys = extractKeys(array);
  let output = [];
  output.push(`\t${keys.join('\t')}`);
  array.forEach(row => {
    output.push(`\t${Object.keys(row).map(key => row[key]).join('\t')}`);
  });
  output = output.join('\n');
  console.log(output);
  return output;
}

const addClimateRowToObject = (row, obj) => {
  if (!obj[row.year]) {
    obj[row.year] = [];
  }
  obj[row.year].push(row.temp_avg);
  return obj;
}

const logNodeError = error => {
  if (error) {
    throw error;
  }
}

module.exports = {
  calculateAverages,
  printQueryResults,
  addClimateRowToObject,
  logNodeError
}
```

id	location	year	temp_avg
552	San Francisco, USA	1851	13.591666666666667
553	San Francisco, USA	1852	12.985714285714282
554	San Francisco, USA	1853	12.883333333333333
555	San Francisco, USA	1854	13.183333333333332
556	San Francisco, USA	1855	14.2
557	San Francisco, USA	1856	11
2	Capetown, South Africa	1857	16.40833333333333
558	San Francisco, USA	1857	14.116666666666667
3	Capetown, South Africa	1858	16.608333333333334
559	San Francisco, USA	1858	13.466666666666667
4	Capetown, South Africa	1859	16.525000000000002
560	San Francisco, USA	1859	12.966666666666669
5	Capetown, South Africa	1860	16.783333333333335
6	Capetown, South Africa	1861	17.01666666666667
561	San Francisco, USA	1861	13.291666666666666
7	Capetown, South Africa	1862	16.508333333333333
562	San Francisco, USA	1862	12.783333333333333
8	Capetown, South Africa	1863	16.566666666666666
563	San Francisco, USA	1863	12.483333333333334
9	Capetown, South Africa	1864	16.916666666666664

Retrieving All Rows

In the previous exercise we were able to import the 'sqlite3' library and use that to open our SQLite database — so far so good! But we still haven't retrieved any information from it. Since we have access to our database as a JavaScript object, let's try running a query on it. Recall that a *query* is a statement that speaks to a database and requests specific information from it. To execute a query and retrieve all rows returned, we use `db.all()`, like so:

```
db.all("SELECT * FROM Dog WHERE breed='Corgi'", (error, rows) => {
  printQueryResults(rows);
});
```

In the previous example, we used the `db.all()` method to retrieve every dog of breed "Corgi" from a table named Dog and print them.

Instructions

1.

Open a call to `db.all()`. Inside, add a query that will select all the rows from the `TemperatureData` table. For now, you can leave the callback empty.

Hint

The SQL syntax for selecting all rows from a table is

```
SELECT * FROM TableName
```

2.

Create your callback function as the second argument of `db.all()`. It should take two arguments and print the second with the `printQueryResults()` function imported at the top of your file.

3.

Replace your query with a new query that will only `SELECT` the rows in the TemperatureData table with the year 1970.

Hint

The SQL syntax for filtering results by some conditional is:

```
SELECT * FROM TableName WHERE col_name = condition;
```

app.js

```
const { printQueryResults } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

// Your code below:

db.all('SELECT * FROM TemperatureData WHERE year = 1970', (error, rows) => {
  printQueryResults(rows);
});
```

```
        id      location        year    temp_avg
        189     Sao Paolo, Brazil       1970
19.150000000000002
        302     Lagos, Nigeria  1970
26.808333333333334
        322     Shanghai. China 1970
15.416666666666666
        434     Jakarta Observatory, Indonesia  1970
27.125
        475     Christchurch, New Zealand       1970
12.491666666666665
        517     Istanbul, Turkey        1970
14.633333333333335
        668     San Francisco, USA      1970
14.225000000000001
```

Retrieving A Single Row

`db.all()` is a useful tool to fetch all the data we have that meets certain criteria. But what if we only want to get a particular row? We could do something like this:

```
db.all("SELECT * FROM Dog", (error, rows) => {
  printQueryResults(rows.find(row => row.id === 1));
});
```

In this example, we fetch all the rows from a database. Doing this populates a JavaScript variable, `rows`, that contains the results of our `SELECT` statement (all the rows from the database). We use JavaScript's `.find()` method to find the row with an ID of 1. Then print out that row.

With a tiny database, this might be OK, but it will be a considerable and unnecessary load if the database is large in any sense. Luckily, we have a different method that will fetch a single row from a database: `db.get()`. See it in action:

```
db.get("SELECT * FROM Dog WHERE owner_name = 'Charlie'", (error, row)
=> {
  printQueryResults(row);
});
```

Sometimes all we need to know is whether a record matching our query exists (for instance: the code above would answer the question "Does Charlie own a dog?" depending on whether or not `row` is `undefined`). Sometimes we know that

there's only a single row because we are searching for a specific ID. And sometimes we only want an example of a row that would match our description. In the code above we would only print information about one dog. To accomplish this, we use `db.get()` instead of `db.all()`.

It's important to note that even if multiple rows match the query, `db.get()` will only return a single result. In the example above, if "Charlie" owns multiple dogs, the code provided will still only print information about one dog.

Instructions

1.

Open a `db.get()` query. Inside, use a `SELECT` statement to get all columns from the first row in `TemperatureData` with data in the year you were born.

Hint

The SQL syntax for restricting results uses `WHERE`:

```
SELECT * FROM TableName WHERE col_name = condition;
```

2.

Create a callback function that takes two arguments: `error` and `row`. Log the row to the console using the provided `printQueryResults` function.

Hint

To use `printQueryResults`, just pass in the array to log:

```
printQueryResults(myArray);
```

app.js

```
const { printQueryResults } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

// Your code below:
db.get("SELECT * FROM TemperatureData WHERE year = 1977", (erro, row) => {
  printQueryResults(row);
})
```

```
        id      location        year    temp_avg
        196     Sao Paolo, Brazil       1977
19.908333333333335
```

Using Placeholders

Now we know how to retrieve data from a database when we know exactly what we're looking for. But we may not always know what values we will need to search for when writing our program. When we write a JavaScript function, we give the function *parameters* that will have many different values when the function gets called. *Placeholders* solve a similar problem in the world of SQL queries. Sometimes we'll want to search our database based on a user's submission. Or we might find ourselves wanting to perform a series of queries looping over some external data.

In those cases, we will have to use a placeholder. A placeholder is a part of our SQL query that we want to be *interpolated* with a variable's contents. We want the value of the JavaScript variable to be placed within the SQL query. To do this properly, we'll need to pass a particular argument to our `db.run()` command that will tell it how to interpolate the query.

```
const furLength1 = "short";
const furLength2 = "long";
const furColor1 = "brown";
const furColor2 = "grey";

const findDogByFur = (length, color) => {
  db.all(
    "SELECT * FROM Dog WHERE fur_length = $furLength AND fur_color
= $furColor",
    {
      $furLength: length,
      $furColor: color
    },
    (error, rows) => {
      printQueryResults(rows);
    }
  );
});

findDogByFur(furLength1, furColor1); // prints all dogs with short
brown fur.
findDogByFur(furLength2, furColor1); // prints all dogs with long
brown fur.
findDogByFur(furLength1, furColor2); // prints all dogs with short
grey fur.
```

```
findDogByFur(furLength2, furColor2); // prints all dogs with long grey fur
```

As we can see in the example above, the power of placeholders is that we don't need to know precisely the data we're searching for at the time of writing our query. We can use these placeholders and then later, when we have values we want to find, we can plug them into the query. This is a highly effective tool that will allow us to harness our programming skills within our database queries.

Instructions

1.

Create a loop that iterates over the given `ids` array. For now, just log every id number.

2.

Within your loop, call `db.get()`. Add a query to `SELECT` a row from table `TemperatureData` with the matching id. You will need to use placeholders to match the id as you iterate. Inside the callback function, use `printQueryResults` to print each row.

Hint

The syntax for using placeholders is

```
db.get('SELECT * from TableName WHERE col_name = $placeholder',
  {
    $placeholder: 'some value'
  },
  (err, row) => {
    // do something with results
  }
)
```

where `col_name` is the name of a column to match, and `$placeholder` is filled in with the value associated with the `$placeholder` in the object provided as the second argument.

There are other signatures for using placeholders in `sqlite3`. You can use any of them to pass this checkpoint.

app.js

```
const { printQueryResults } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const ids = [1, 25, 45, 100, 360, 382];
// your code below:

ids.map(id => {
  db.get("SELECT * FROM TemperatureData WHERE id = $id",  {
        $id: id
        },
    (error, rows) => {
      printQueryResults(rows);
    }
)
})
```

```
        id      location        year    temp_avg
        382     Jakarta Observatory, Indonesia  1916    26.216666666666672
        id      location        year    temp_avg
        25      Capetown, South Africa  1880    16.736363636363638
        id      location        year    temp_avg
        1       Vostok, Antarctica      1957    -48.987500000000004
        id      location        year    temp_avg
        360     Jakarta Observatory, Indonesia  1894    25.899999999999995
        id      location        year    temp_avg
        45      Capetown, South Africa  1900    16.999999999999996
        id      location        year    temp_avg
        100     Capetown, South Africa  1955    16.683333333333334
```

Using db.run()

Not all SQL commands return rows, and in fact, some essential SQL commands do not. If we `INSERT` a row or `CREATE` a `TABLE` we will not receive a row in response. To perform SQL commands that do not return rows, we use `db.run()` to `run` the command. `db.run()` does not return a value, but, depending on the SQL command, it may attach properties to the `this` keyword within the scope of the callback. In some cases, like creating a table, `db.run()` will not modify `this`. In other cases, like when `INSERT`ing a row, a

callback to `db.run()` will be able to access `this.lastID`, the ID of the last `INSERT`ed row.

Instructions

1.
Write a `db.run()` command that will `INSERT` the given data into our `TemperatureData` table. Be sure to use `sqlite3` placeholders and not hard-code the values from `newRow`.

Add a function callback with a single argument and leave it empty for now. Make sure that this function is not an arrow.

See the hint for a reminder about the SQL `INSERT` syntax.

Hint
The SQL syntax for inserting is

```
INSERT INTO TableName (col_1_name, col_2_name, ...) VALUES (val_1, val2, ...);
```

2.
In a callback of `db.run()`, log `this.lastID` to see the id of the inserted row.

Hint
Reminder: `db.run()` sytax is as follows:

```
db.run('<SQL query>', function(error) {
  // this callback will be called after the query has run.
}).
```

3.
Notice that the logged value is `undefined`. What went wrong? Move on to the next exercise to find out.

app.js

```
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const newRow = {
  location: 'Istanbul, Turkey',
  year: 1976,
}
// Your code below!
```

```
db.run('INSERT INTO TemperatureData (location, year) VALUES ($location, $year)',
{
  $location: newRow.location,
  $year: newRow.year
}, function(error) {
    console.log(this.lastID)
});
```

Handling Errors Gracefully

No one's perfect. Code, like people, can make mistakes. This is OK! What's important is that we learn how to handle our difficulties while keeping our composure. Handling errors is an important part of the process when dealing with Node & SQL in particular. When our code throws an error, we should be able to handle it before it reaches our end users and incites panic and concern. We still want to know what happened, so that we can perform a suitable action based on the error that occurred — so we *catch* the error.

For `db.run()`, `db.each()`, `db.get()`, and `db.all()`, the first argument to the callback will always be an `Error` object if an error occurred. If there is no error, this argument will be `null`. We can check if this error exists, and if it does exist, we can handle it.

Instructions

1.

Add an `if` check to see if an error exists in the callback of the INSERT statement in `db.run()`. The error will be `null` if no error exists. Log an error if one is passed into the callback and `return` to break out of the callback;

Hint

The syntax to check and log the error is as follows:

```
if (error) {
  return console.log(error);
}
```

2.
You can see the SQLite error in the console: we are missing a value for the `temp_avg` column, whoops! It turns out that the average temperature in Istanbul in 1976 was 13.35. Fix your `INSERT` statement to add it to the `TemperatureData` table. Make sure to leave your log for `this.lastID` in the `db.run()` callback.

3.
Now that the ID is being logged, open a `db.get()` query that `SELECT`s the inserted row by id. You should use a placeholder to interpolate `this.lastID` into your `SELECT` statement.

In a callback to that `db.get()` query, log the row again to show that it's been entered into our database using `printQueryResults`

Hint
Refer to the documentation on placeholder syntax if you need a reminder about how to select the correct row by ID.

app.js

```
const { printQueryResults } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const newRow = {
  location: 'Istanbul, Turkey',
  year: 1976,
  tempAvg: 13.35
}

db.run('INSERT INTO TemperatureData (location, year, temp_avg) VALUES ($location,
 $year, $tempAvg)', {
  $location: newRow.location,
  $year: newRow.year,
  $tempAvg: newRow.tempAvg
}, function(error) {
  // handle errors here!
  if(error){
    return console.log(error);
  }
```

```
  console.log(this.lastID);

  db.get('SELECT * FROM TemperatureData WHERE id = $id', {
      $id: this.lastID
  },
  function(error, row){
    printQueryResults(row);
  });
});
```

```
725
        id      location        year    temp_avg
        725     Istanbul, Turkey        1976    13.35
```

Using db.each()

While learning JavaScript, you likely learned about the powerful method `.forEach()` that allows us to process every element in an array separately. Now we will use a similar method that will enable us to process every row returned from a database query.

```
db.each("SELECT * FROM Dog WHERE breed = 'Labrador'",
  (error, row) => {
    // This gets called for every row our query returns
    console.log(`${row.name} is a good dog`);
  },
  (error, numberOfRows) => {
    // This gets called after each of our rows have been processed
    console.log(`There were ${numberOfRows} good dogs`);
});
```

In the code above we `SELECT` all the Labrador dogs from our `Dog` database. We offer affirmation to each of the animals individually and then announce how many received this praise in sum.

`db.each()` takes a query and a callback function that it performs on each row returned from the query. This is a useful technique for transforming or updating rows. This is also useful for memory management — we only have to look at one row at a time instead of trying to process every returned row at

the same time. `db.each()` additionally takes an optional second callback function, which will be called when all of the queries are completed and processed.

Instructions

1.
Create an empty `temperaturesByYear` object before your query; we'll use this to store temperature data for each year.

2.
Inside a `db.each()` call, `SELECT` all the rows from `TemperatureData`.

3.
In a callback from your `db.each()` call, add the temperature's value to the `temperaturesByYear`. To do this, use the provided helper function `addClimateRowToObject(row, object)`. The first argument is the row to add, the second argument is the object to add it to (`temperaturesByYear` in your case).

Hint
Your callback function should look something like:

```
(error, row) => {
  addClimateRowToObject(row, temperaturesByYear);
}
```

4.
In the second callback to your `db.each()` call, create a final `averageTemperatureByYear` variable and set it equal to calling function `calculateAverages` with your `temperaturesByYear` object.

5.
Log these averages using the given `printQueryResults()` function.

app.js

```
const { printQueryResults, calculateAverages, addClimateRowToObject } = require('
./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const temperaturesByYear = {};

db.run('DROP TABLE IF EXISTS Average', error => {
  if (error) {
    throw error;
```

```
  }
  db.each('SELECT * FROM TemperatureData',
    (error, row) => {
      if (error) {
        throw error;
      }
      addClimateRowToObject(row, temperaturesByYear);
    },
    error => {
      if (error) {
        throw error;
      }
      const averageTemperatureByYear = calculateAverages(temperaturesByYear);
      printQueryResults(averageTemperatureByYear);
    }
  );
})
```

```
year    temperature
1851    13.591666666666667
1852    12.985714285714282
1853    12.883333333333333
1854    13.183333333333332
1855    14.2
1856    11
1857    15.2625
1858    15.037500000000001
1859    14.745833333333335
1860    16.783333333333335
1861    15.154166666666669
1862    14.645833333333332
1863    14.525
1864    15.037499999999998
1865    14.808333333333334
```

Creating A New Table
So far we've managed to:

1. Query a database for weather records by location.
2. Reformat that data into a JavaScript object.
3. Manipulate that JavaScript object to find new, meaningful information.

That's pretty significant! Now it's time to take that useful information and add it to a table of our own. Since creating a table is another operation that does not return a row, we can use the same `db.run()` we used to `INSERT` rows. Let's see what happens when we `CREATE` a `TABLE` and `INSERT` our data into it.

Notice there's a statement declaring "DROP TABLE IF EXISTS" — this is because we want to make sure when we create our new table that we won't run into an error telling us the table already exists (from previous times running our code).

Instructions

1.
Use `db.run()` to `CREATE` a new table `Average` with `id`, `year`, and `temperature` columns.

```
id: INTEGER PRIMARY KEY
year: INTEGER NOT NULL
temperature: REAL NOT NULL
```

Hint
SQL CREATE TABLE syntax:

```
CREATE TABLE TableName (
  --pattern:
  row_name TYPE constraints,
  --example:
  id INTEGER PRIMARY KEY,
);
```

2.
After creating your table with `db.run()`, iterate through your `averageTemperatureByYear` array and `INSERT` each into your table using a `db.run()`.

Hint
Each element in your `averageTemperatureByYear` has a `year` and `temperature` property. You can inject these into an `INSERT` statement using placeholders.

3.

It looks like an error was thrown. Add a callback after your `INSERT` query in `db.run()` and log the error to the console if it exists.

4.

We've triggered a lot of errors! It looks like some `INSERT`s are happening before the table has been created. Click 'Next' to find out why this is happening and how to fix it.

app.js

```
const { calculateAverages, addClimateRowToObject, logNodeError, printQueryResults
 } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const temperaturesByYear = {};

// start by wrapping all the code below in a serialize method

db.run('DROP TABLE IF EXISTS Average', error => {
  if (error) {
    throw error;
  }
  db.each('SELECT * FROM TemperatureData',
    (error, row) => {
      if (error) {
        throw error;
      }
      addClimateRowToObject(row, temperaturesByYear);
    },
    error => {
      if (error) {
        throw error;
      }
      const averageTemperatureByYear = calculateAverages(temperaturesByYear);
      db.run('CREATE TABLE Average (id INTEGER PRIMARY KEY, year INTEGER NOT NULL
, temperature REAL NOT NULL)', logNodeError);
      averageTemperatureByYear.forEach(row => {
        db.run('INSERT INTO Average (year, temperature) VALUES ($year, $temp)', {
          $year: row.year,
```

```
        $temp: row.temperature
      }, err => {
        if (err) {
          console.log(err);
        }
      });
    });
  }
);
});
```

```
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
{ Error: SQLITE_ERROR: no such table: Average errno:
1, code: 'SQLITE_ERROR' }
```

Serial Queries

By default, the commands we issue to our database run in *parallel*. Every request we make gets sent to the database — which processes them all as quickly as it can, regardless of the order in which they got sent. This is usually a good thing because it means that we can get results faster, but in our case, we don't want to try to `INSERT` data into a table that hasn't been created yet.

One way to avoid this issue is to write all of our code in nested callbacks, let's take a look at how that might look:

```
db.run("DROP TABLE Dog", error => {
  db.run("CREATE TABLE Dog", error => {
    db.run("INSERT INTO Dog (breed, name, owner, fur_color,
fur_length) VALUES ('Dachschund', 'Spike', 'Elizabeth', 'Brown',
'Short')", error => {
    }
  }
}
```

As you can see, with this technique every command gets increasingly indented, which becomes a bit of an eyesore if we want to guarantee multiple things run chronologically. Another way of accomplishing this task is by using the `db.serialize()` method like so:

```
db.serialize(() => {
  db.run("DROP TABLE Dog");
  db.run("CREATE TABLE Dog");
  db.run("INSERT INTO Dog (breed, name, owner, fur_color, fur_length)
VALUES  ('Dachshund', 'Spike', 'Elizabeth', 'Brown', 'Short')");
});
```

In the previous example, we explicitly tell the database to:

- First, remove the table `Dog` if it exists.
- Second, create an empty table named `Dog`.
- Third, insert a new row into the table. In exactly that order without running any command until the previous one completes.

Instructions

1.

Let's un-nest your code to take advantage of `db.serialize()`. We'll go step by step. First, open a call to `db.serialize()`. Put all of your nested `db` code inside of `db.serialize()`'s callback function.

Hint

Your code should a structure resembling this:

```
db.serialize(() => {
  // All your current db.<method> code
});
```

2.

We start with a clean slate every time the code runs with a `DROP TABLE IF EXISTS` statement. All your queries are currently inside the callback for this query. Close the callback function after the error checking and un-nest the `db.each()` method You can leave the contents of `db.each()` as they are for

now. The `db.each()` query should be on the same level as your `DROP TABLE` query and will run serially after it.

Hint

The DROP TABLE query should be completely closed before the `db.each()`:

```
// Inside db.serialize():
db.run('DROP TABLE ....',
  err => {
    // Handle errors
  }
);
db.each('<query>', (err, row) => {
  // Your other queries
});
```

3.

Move your command to `CREATE` the table `Average` into your `db.serialize()` method call right after dropping the table and before `db.each()`. It should be at the same level of nesting as `db.each()`.

Leave your command to `INSERT` the rows into `Average` inside the second callback of `db.each()`, guaranteeing that the averages are calculated after your table is created.

Hint

Your code inside `db.serialize()` should now have this structure:

```
db.run('DROP TABLE ....', (err) => {
  // Contents for error handling
});
db.run('CREATE TABLE ... ', (err) => {
  // Contents for error handling
});
db.each('<query>', (err, row) => {
  // All the rest of your logic
});
```

4.

No more errors! After all your rows have been inserted with the `averageTemperatureByYear.forEach()` loop inside `db.each()`, create a new `db.all()` query to `SELECT` all rows from the `Average` table and `printQueryResults()` with the transformed data!

Hint

Use `db.all()` to select many rows.

5.

We were able to add this information to the new table, congrats! Review the results logged to the console, do they make sense?

app.js

```
const { calculateAverages, addClimateRowToObject, logNodeError, printQueryResults
 } = require('./utils');
const sqlite = require('sqlite3');

const db = new sqlite.Database('./db.sqlite');

const temperaturesByYear = {};

// start by wrapping all the code below in a serialize method
db.serialize(() => {
  db.run('DROP TABLE IF EXISTS Average', error => {
    if (error) {
      throw error;
    }
  })
  db.run('CREATE TABLE Average (id INTEGER PRIMARY KEY, year INTEGER NOT NULL, te
mperature REAL NOT NULL)', logNodeError);
  db.each('SELECT * FROM TemperatureData',
    (error, row) => {
      if (error) {
        throw error;
      }
      addClimateRowToObject(row, temperaturesByYear);
    },
    error => {
      if (error) {
        throw error;
      }
      const averageTemperatureByYear = calculateAverages(temperaturesByYear);
      averageTemperatureByYear.forEach(row => {
        db.run('INSERT INTO Average (year, temperature) VALUES ($year, $temp)', {
          $year: row.year,
          $temp: row.temperature
        }, err => {
          if (err) {
            console.log(err);
          }
        });
      });
```

```
  db.all('SELECT * FROM Average',
  (error, row) => {
    printQueryResults(row)
  })
  });
});
```

id	year	temperature
1	1852	12.985714285714282
2	1853	12.883333333333333
3	1854	13.183333333333332
4	1855	14.2
5	1856	11
6	1857	15.2625
7	1861	15.154166666666669
8	1859	14.745833333333335
9	1860	16.783333333333335
10	1864	15.037499999999998
11	1862	14.645833333333332
12	1858	15.037500000000001
13	1867	18.305555555555554
14	1863	14.525
15	1868	18.32777777777778
16	1869	21.570833333333333
17	1870	21.166666666666664
18	1866	18.455555555555556
19	1865	14.808333333333334

Wrap Up

Wow, we were able to take data from a source, manipulate it for our needs, and save it separately. We've managed to set up a database, create tables, insert data, and modify it — all within JavaScript! We learned about how to ensure that JavaScript runs a set of commands chronologically so that we can avoid race conditions. We leveraged JavaScript methods fluently alongside our database queries in SQL. Now our knowledge of SQL and our understanding of JavaScript can work together to accomplish more than either could alone.

Good job!

Instructions

Feel free to add more queries and experiment with the results and move on when you’re ready to learn more!