**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 — CREATEing INSERTing and SELECTing, 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.

Checkpoint 2 Passed

**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].length

    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 = {

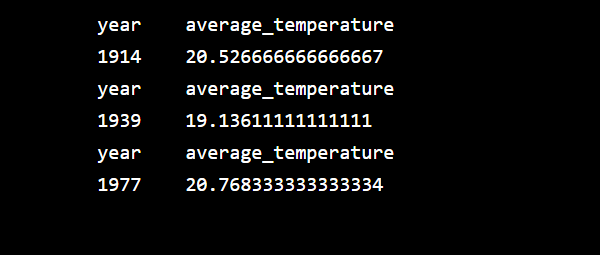
  calculateAverages,

  printQueryResults,

  addClimateRowToObject,

  logNodeError

}

****

**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

Checkpoint 2 Passed

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.

Checkpoint 3 Passed

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].length

    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 = {

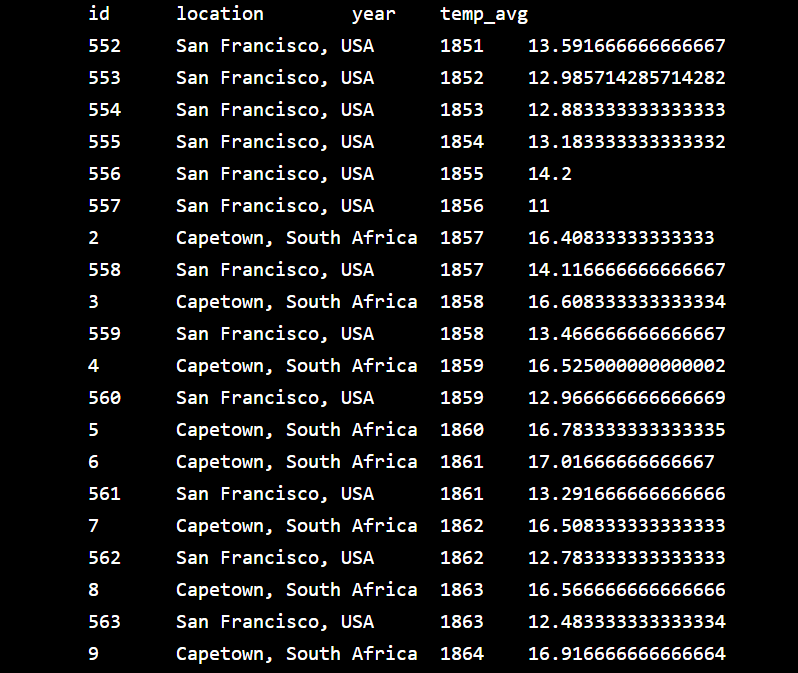
  calculateAverages,

  printQueryResults,

  addClimateRowToObject,

  logNodeError

}

****

**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.

Checkpoint 2 Passed

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.

Checkpoint 3 Passed

**3.**

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

Checkpoint 4 Passed

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);

});

****

**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.

Checkpoint 2 Passed

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.

Checkpoint 3 Passed

Hint

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

printQueryResults(myArray);

**app.js**

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

const sqlite = require('sqlite3');

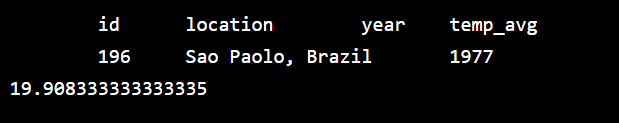
const db = new sqlite.Database('./db.sqlite');

// Your code below:

db.get("SELECT \* FROM TemperatureData WHERE year = 1977", (erro, row) => {

  printQueryResults(row);

})

****

**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.

Checkpoint 2 Passed

**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.

Checkpoint 3 Passed

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

        },

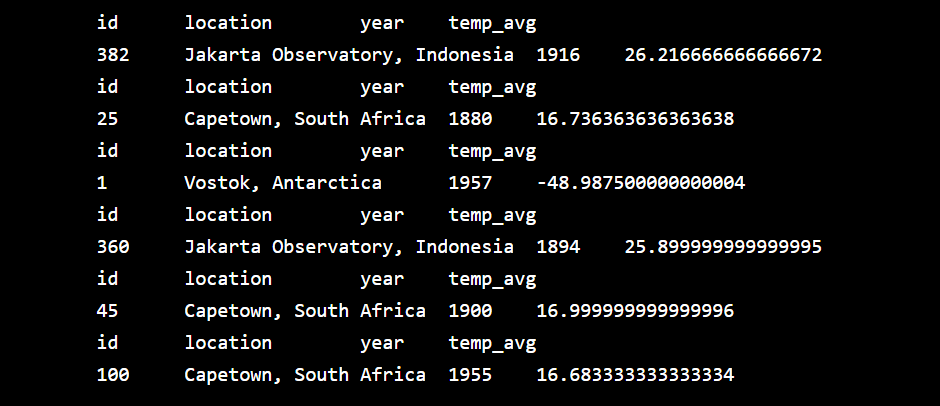
    (error, rows) => {

      printQueryResults(rows);

    }

)

})

****

**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 INSERTing a row, a callback to db.run() will be able to access this.lastID, the ID of the last INSERTed 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.

Checkpoint 2 Passed

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.

Checkpoint 3 Passed

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)', {

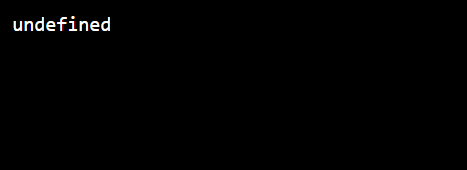
  $location: newRow.location,

  $year: newRow.year

}, function(error) {

    console.log(this.lastID)

});

****