Node.js to PostgreSQL

By Ariel Shultz

What this presentation covers

- What is Node.js
- What is PostgreSQL
- Why we use Node.js & PostgreSQL
- A simple step by step example of connecting Node.js with PostgreSQL
- Asynchronous programming and using async/await
- More in depth look at the ‘pg’ package: pg.Client, pg.Pool, pg.Result
- An example application that utilizes Node.js and PostgreSQL

Understanding Node.js & PostgreSQL

What is Node.js

An open-source JavaScript runtime environment for running web applications outside the client’s browser.

- Instead of running a web applications directly in your browser (the client browser), Node.js allows developers to run parts of the web application on a server.

Why do we use Node.js

Just some of the reasons:

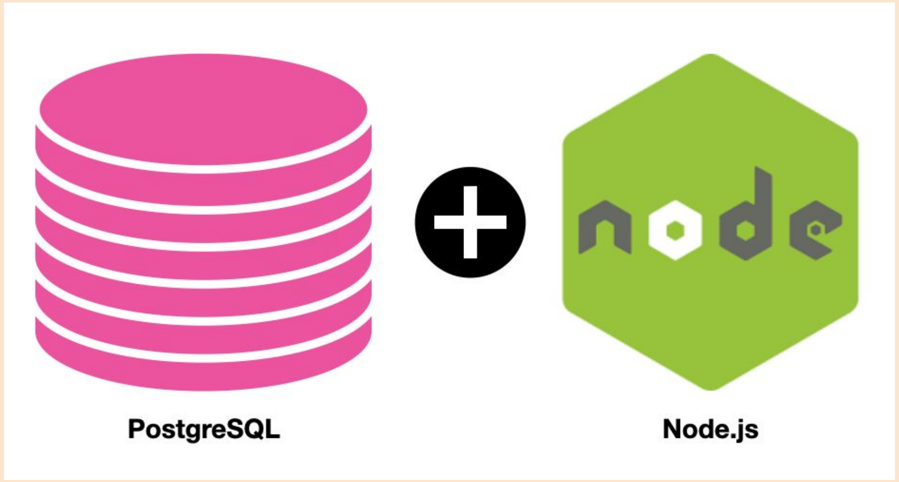
- High performance
- A LOT of libraries and modules
- Cross-platform

What is PostgreSQL

PostgreSQL is an object-relational database management system

- Can store and manage large amounts of data in an organized and efficient way
- Combines features of both relational databases (MySQL) and object-oriented databases.

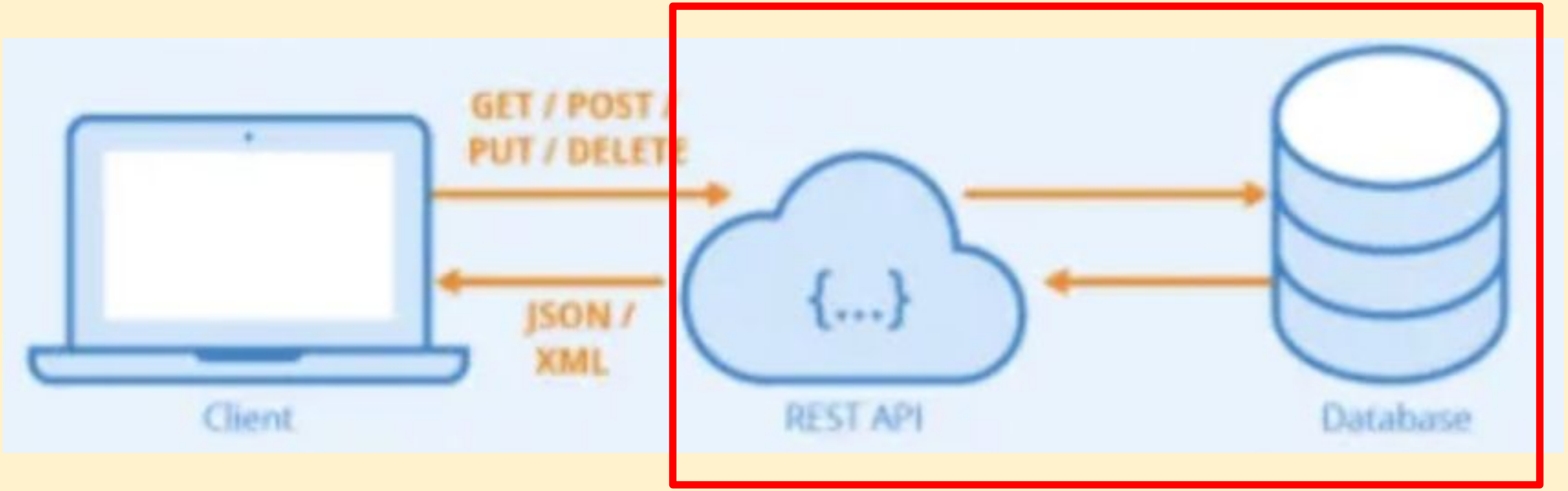
Node.js and PostgreSQL



Node.js and PostgreSQL can work together to build powerful web applications. Together, they create a seamless experience for users interacting with applications.

A Simple Step-By-Step Tutorial of connecting Node.js with PostgreSQL

Connecting Node.js with PostgreSQL

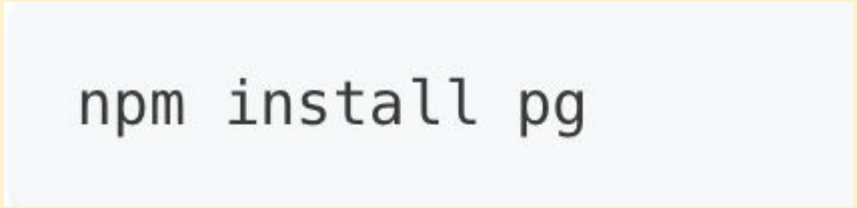


Step 1: Installing the ‘pg’ package

To create a connection to a PostgreSQL database using Node.js, we will use the ‘**pg**’ package.

- **How to do:** npm install pg (run this command in your terminal before running Node.js)

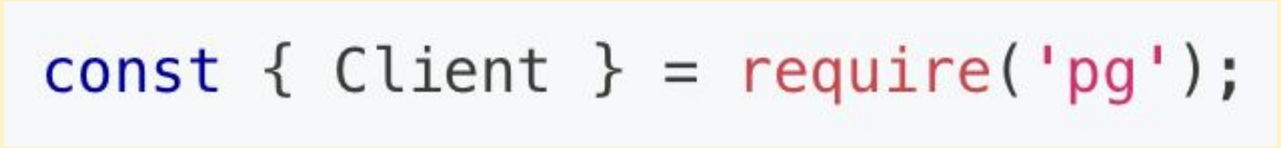
“pg is a popular Node.js library used to work with Postgres. It serves as a PostgreSQL database driver for Node.js applications.”



```
npm install pg
```

Step 2: Initialize the ‘pg’ package

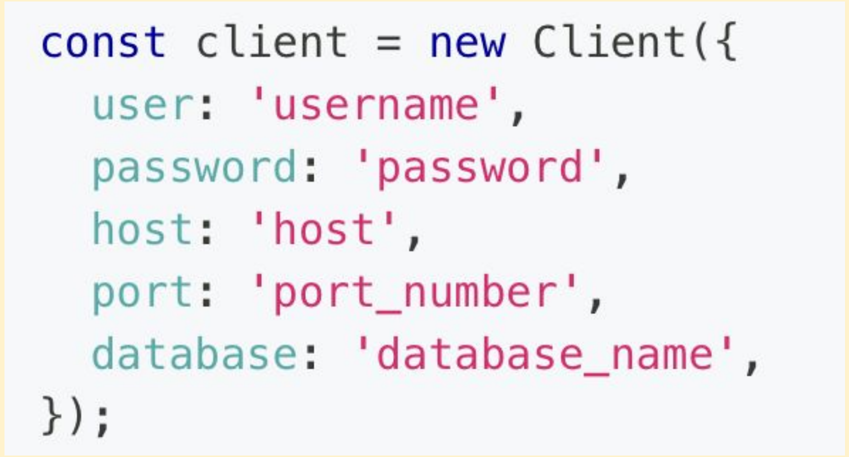
Initialize the pg package in your Node.js script and get the Client from it:



```javascript
const { Client } = require('pg');
```

Step 3: Create a PostgreSQL client object

Create a PostgreSQL client object consisting of essential PostgreSQL database credentials:

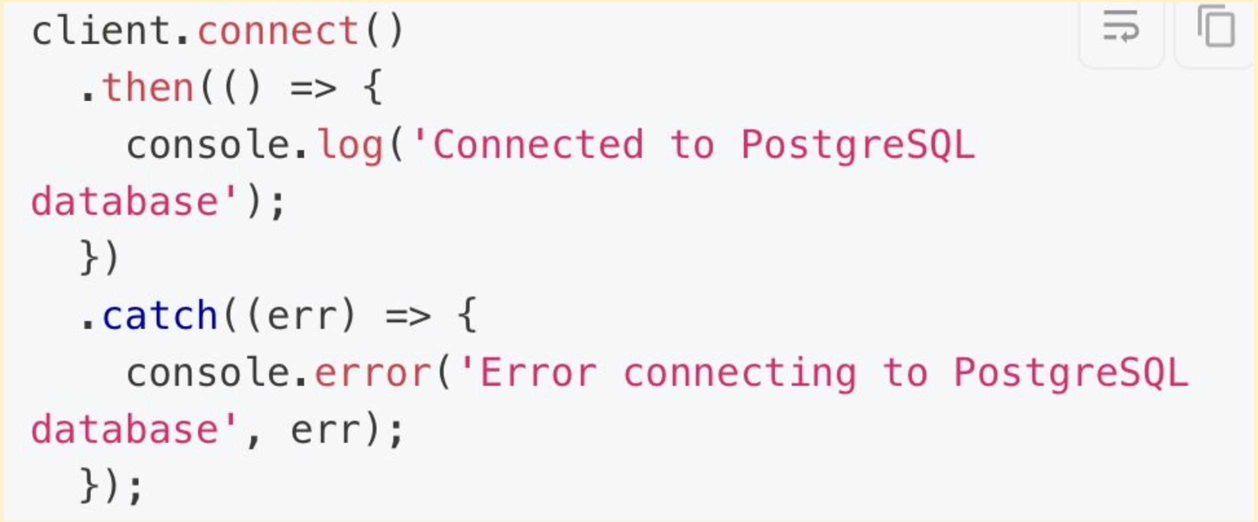


```
const client = new Client({
  user: 'username',
  password: 'password',
  host: 'host',
  port: 'port_number',
  database: 'database_name',
});
```

**Replace with your actual Postgres database credentials*

Step 4: Setup a connection with the database

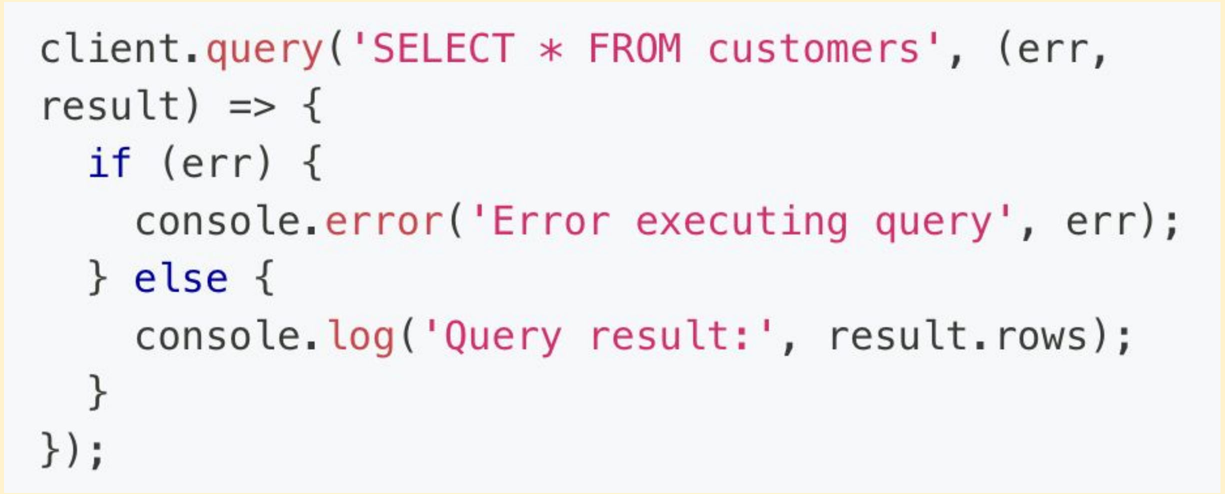
Setup a connection with the database using the connect method with the manually created client object (from before):



```
client.connect()
  .then(() => {
    console.log('Connected to PostgreSQL
database');
  })
  .catch((err) => {
    console.error('Error connecting to PostgreSQL
database', err);
  });
```

Step 5: Execute your desired SQL query

Execute your desired SQL query to get the data. You can use the query method to run the SQL query:



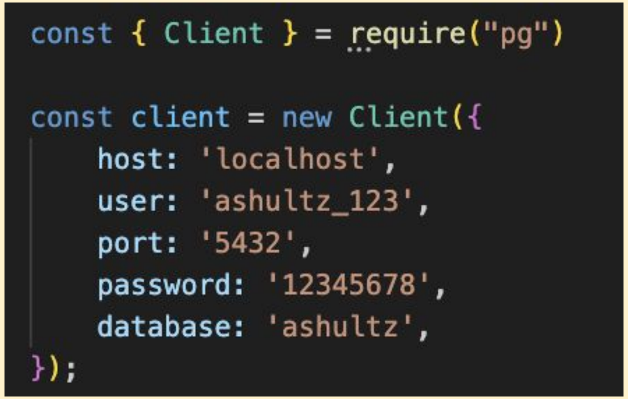
```
client.query('SELECT * FROM customers', (err,
result) => {
  if (err) {
    console.error('Error executing query', err);
  } else {
    console.log('Query result:', result.rows);
  }
});
```

Step 6: Close the connection

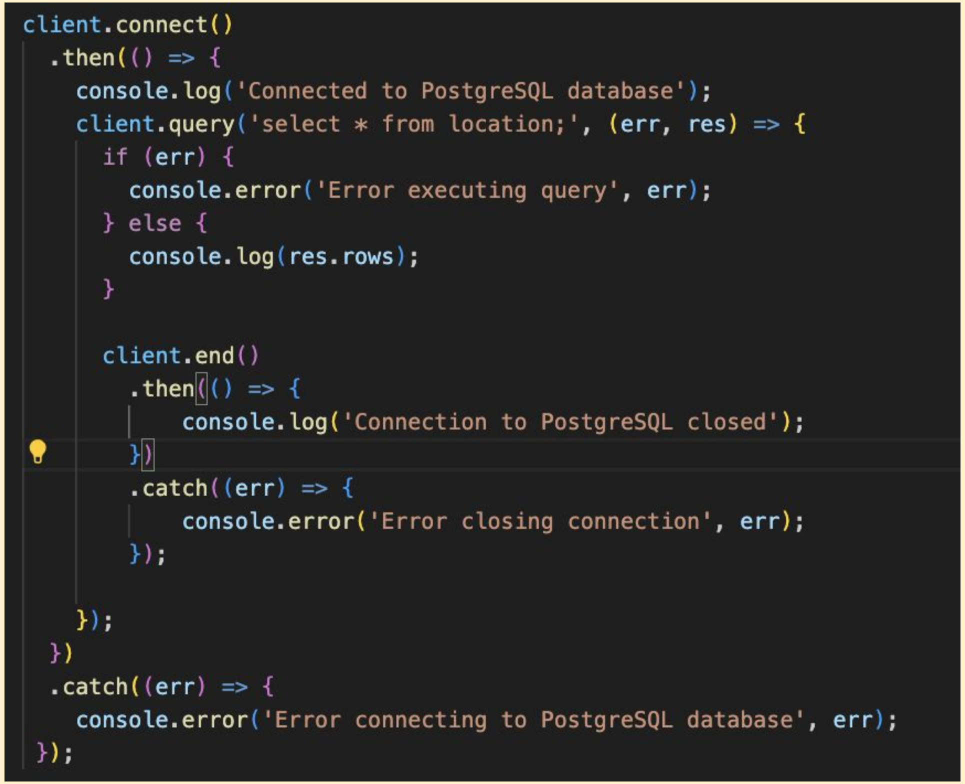
Close the connection after the whole work is done. Use the end method to close the connection:

```
client.end()
  .then(() => {
    console.log('Connection to PostgreSQL closed');
  })
  .catch((err) => {
    console.error('Error closing connection', err);
  });
```

The complete code: Putting all the steps together

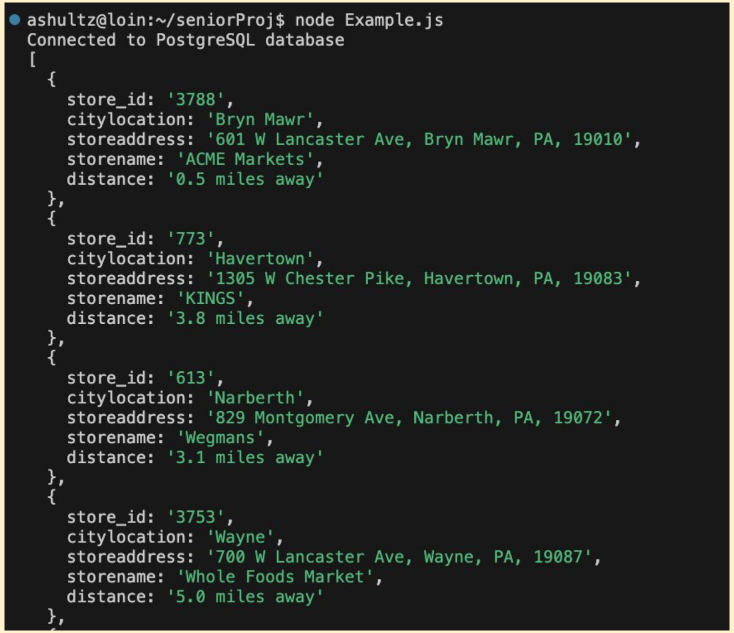


```
const { Client } = require("pg")

const client = new Client({
    host: 'localhost',
    user: 'ashultz_123',
    port: '5432',
    password: '12345678',
    database: 'ashultz',
});
```

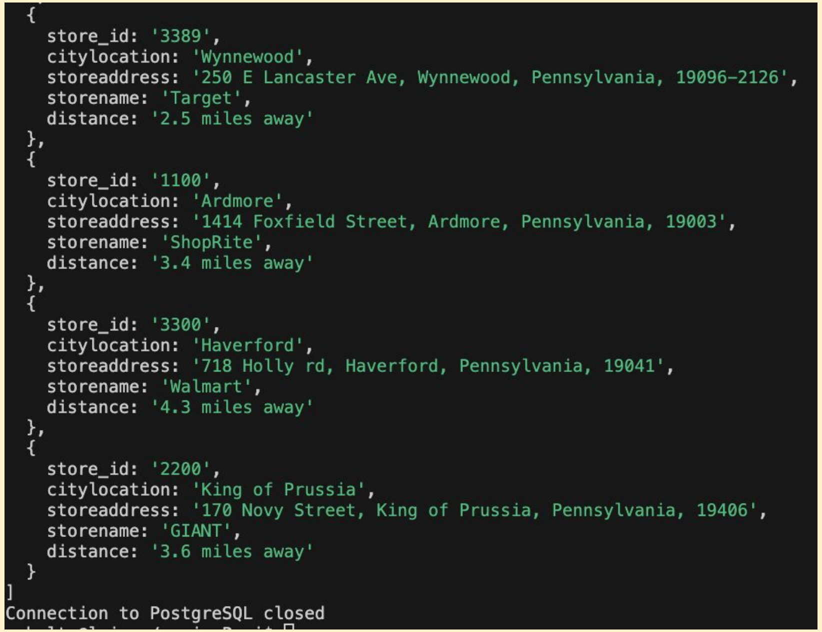


```
client.connect()
  .then(() => {
    console.log('Connected to PostgreSQL database');
    client.query('select * from location;', (err, res) => {
      if (err) {
        console.error('Error executing query', err);
      } else {
        console.log(res.rows);
      }

      client.end()
        .then(() => {
            console.log('Connection to PostgreSQL closed');
        })
        .catch((err) => {
            console.error('Error closing connection', err);
        });

    });
  })
  .catch((err) => {
    console.error('Error connecting to PostgreSQL database', err);
});
```

Output of the code:



```
ashultz@loin:~/seniorProj$ node Example.js
Connected to PostgreSQL database
[
  {
    store_id: '3788',
    citylocation: 'Bryn Mawr',
    storeaddress: '601 W Lancaster Ave, Bryn Mawr, PA, 19010',
    storename: 'ACME Markets',
    distance: '0.5 miles away'
  },
  {
    store_id: '773',
    citylocation: 'Havertown',
    storeaddress: '1305 W Chester Pike, Havertown, PA, 19083',
    storename: 'KINGS',
    distance: '3.8 miles away'
  },
  {
    store_id: '613',
    citylocation: 'Narberth',
    storeaddress: '829 Montgomery Ave, Narberth, PA, 19072',
    storename: 'Wegmans',
    distance: '3.1 miles away'
  },
  {
    store_id: '3753',
    citylocation: 'Wayne',
    storeaddress: '700 W Lancaster Ave, Wayne, PA, 19087',
    storename: 'Whole Foods Market',
    distance: '5.0 miles away'
  },
```



```
  {
    store_id: '3389',
    citylocation: 'Wynnewood',
    storeaddress: '250 E Lancaster Ave, Wynnewood, Pennsylvania, 19096-2126',
    storename: 'Target',
    distance: '2.5 miles away'
  },
  {
    store_id: '1100',
    citylocation: 'Ardmore',
    storeaddress: '1414 Foxfield Street, Ardmore, Pennsylvania, 19003',
    storename: 'ShopRite',
    distance: '3.4 miles away'
  },
  {
    store_id: '3300',
    citylocation: 'Haverford',
    storeaddress: '718 Holly rd, Haverford, Pennsylvania, 19041',
    storename: 'Walmart',
    distance: '4.3 miles away'
  },
  {
    store_id: '2200',
    citylocation: 'King of Prussia',
    storeaddress: '170 Novy Street, King of Prussia, Pennsylvania, 19406',
    storename: 'GIANT',
    distance: '3.6 miles away'
  }
]
Connection to PostgreSQL closed
```

Other operations

In the previous example, I demonstrated how you can read the data, but you can also perform other operations on the PostgreSQL database like Insert and Update.

Let's look at examples for Inserting and Updating

Insert statements →

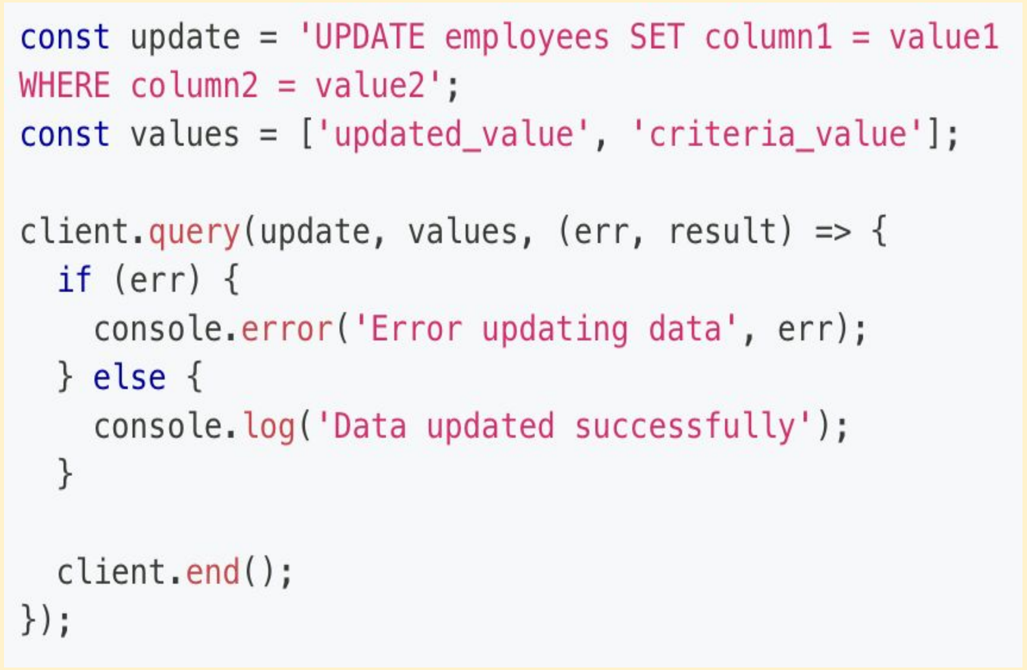
Select statement VS Insert statement:

```
client.query('SELECT * FROM customers', (err,
result) => {
```

```
client.query(insert, values, (err, result) => {
```

```
client.connect()
  .then(() => {

    const insert = 'INSERT INTO employees(column1,
column2) VALUES (value1, value2)';
    const values = ['value1', 'value2'];

    client.query(insert, values, (err, result) => {
      if (err) {
        console.error('Error inserting data', err);
      } else {
        console.log('Data inserted successfully');
      }

      client.end();
    });
  })
  .catch((err) => {
    console.error('Error connecting to PostgreSQL
database', err);
  });
```

Update Statement →



```
const update = 'UPDATE employees SET column1 = value1
WHERE column2 = value2';
const values = ['updated_value', 'criteria_value'];

client.query(update, values, (err, result) => {
  if (err) {
    console.error('Error updating data', err);
  } else {
    console.log('Data updated successfully');
  }

  client.end();
});
```

The tutorial I followed:

https://tembo.io/docs/postgres_guides/connecting-to-postgres-with-nodejs

Asynchronous Operations in Node.js and PostgreSQL: Using async & await

Asynchronous

In the context of Node.js and PostgreSQL, this refers to the way in which operations are executed without blocking the execution of other code.

Why it’s important:

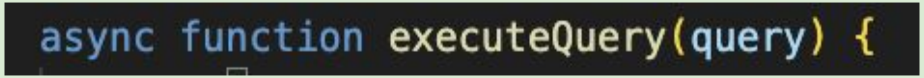
- JavaScript is single-threaded, meaning it can only do one thing at a time
- So, asynchronous operations allow JavaScript to perform multiple tasks simultaneously without waiting for each task to finish before moving onto the next one.
- Async operations keep the program responsive and efficient by allowing it to multitask effectively

Understanding async/await in Node.js

Async/await is a feature in Node.js that makes its easier to manage tasks that take time, like waiting for a response from an API.

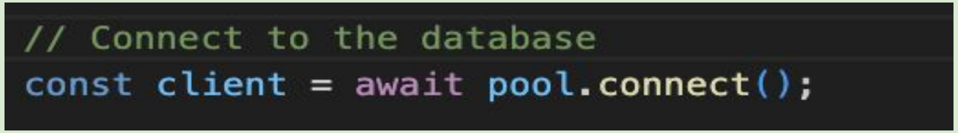
Async/await is a more readable and straightforward syntax for writing asynchronous code in JavaScript that you can incorporate into your Node.js code.

- async is used in a function's signature →

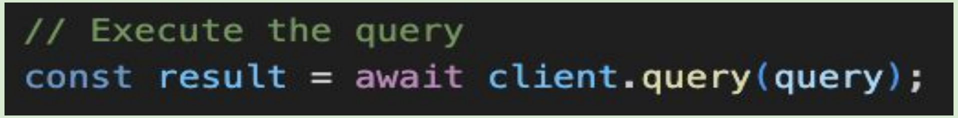


```
async function executeQuery(query) {
```

- await is used within an async function and can be used for connecting to a PostgreSQL database and executing a query →



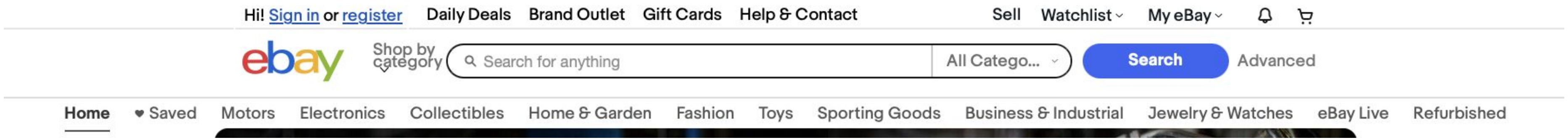
```
// Connect to the database
const client = await pool.connect();
```



```
// Execute the query
const result = await client.query(query);
```

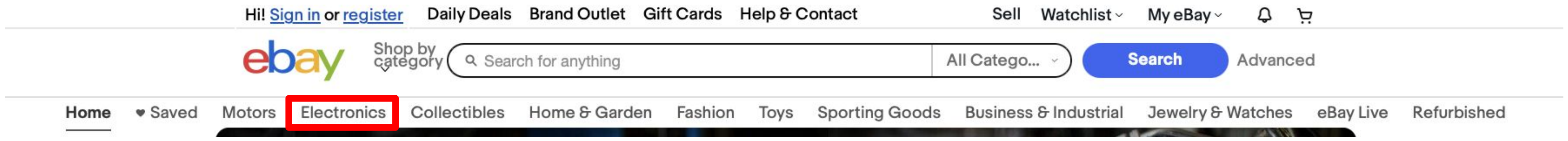
Example of asynchronous calls from Node.js to PostgreSQL

Consider an online marketplace platform like eBay:



Simplified breakdown of applications that utilize asynchronous operations

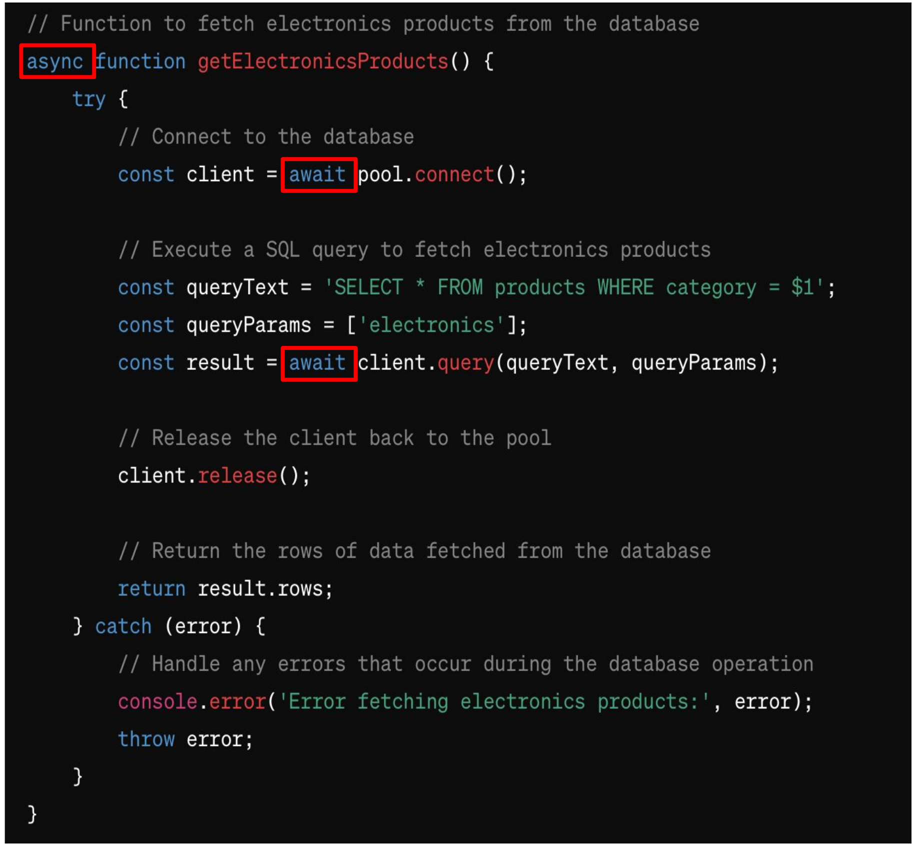
User interaction: a user visits the eBay webpage and navigates to ‘Electronics’



At this point: The Node.js application detects the users request to navigate to the ‘Electronics’ page and will try to retrieve and display the electronic products on that webpage.

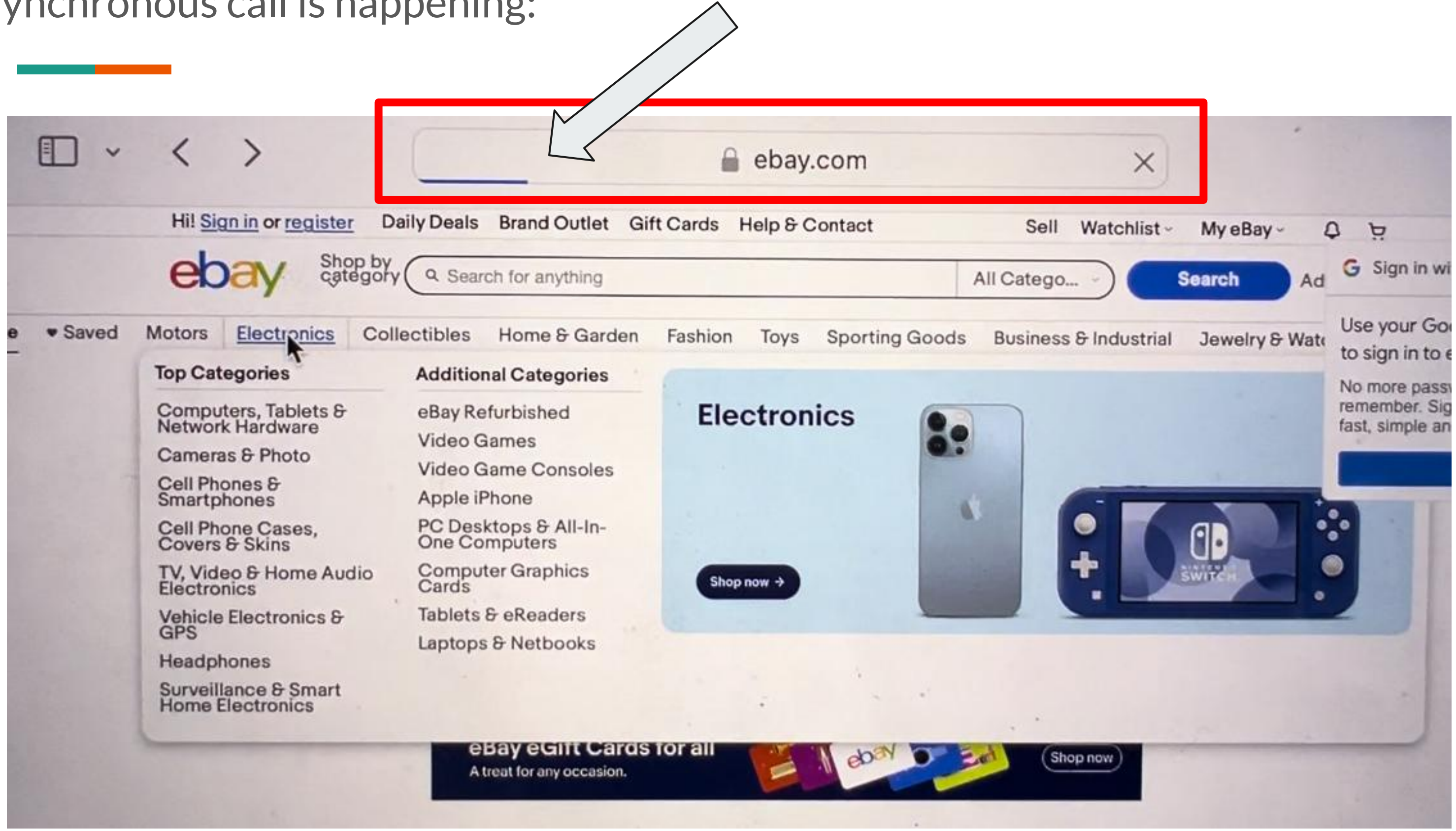
Asynchronous Database Query: An asynchronous database query is made to fetch the product listings from the PostgreSQL database.

Simplified example code

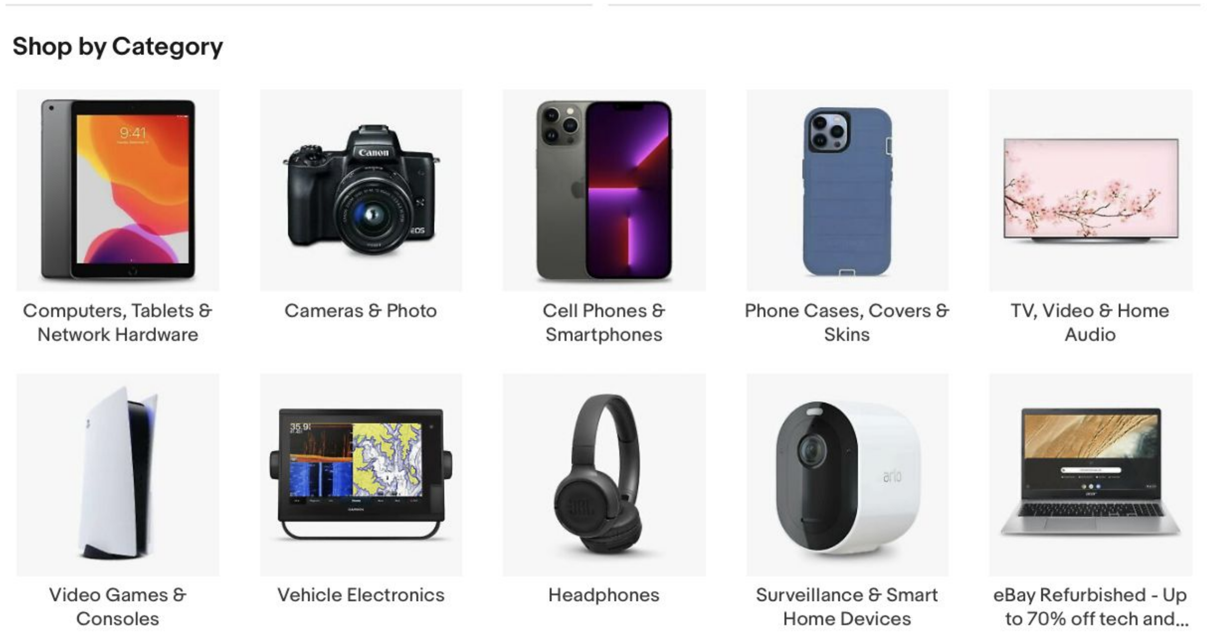


```
// Function to fetch electronics products from the database
async function getElectronicsProducts() {
    try {
        // Connect to the database
        const client = await pool.connect();

        // Execute a SQL query to fetch electronics products
        const queryText = 'SELECT * FROM products WHERE category = $1';
        const queryParams = ['electronics'];
        const result = await client.query(queryText, queryParams);

        // Release the client back to the pool
        client.release();

        // Return the rows of data fetched from the database
        return result.rows;
    } catch (error) {
        // Handle any errors that occur during the database operation
        console.error('Error fetching electronics products:', error);
        throw error;
    }
}
```

Asynchronous call is happening:



Asynchronous call completed and the products were displayed on the Electronics page:


Shop by Category

Computers, Tablets & Network Hardware

Cameras & Photo

Cell Phones & Smartphones

Phone Cases, Covers & Skins

TV, Video & Home Audio

Video Games & Consoles

Vehicle Electronics

Headphones

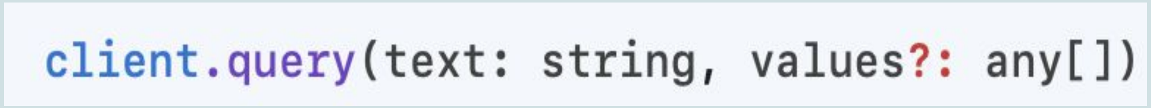
Surveillance & Smart Home Devices

eBay Refurbished - Up to 70% off tech and...

Back to connecting Node.js with PostgreSQL: More in depth look at the ‘pg’ package (pg.Client, pg.Pool, pg.Result)

pg.Client

When using ‘client.query’ you can pass various objects based on the following signature:



```
client.query(text: string, values?: any[])
```

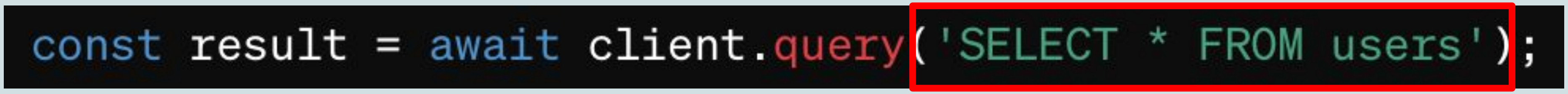
text: string → the raw query text (ex. ‘select * from locations’) **MUST BE INCLUDED**

values?: Array<any> → an array of query parameters (ex. [‘Bryn Mawr’, ‘Haverford’]) **Optional**

Plain text query:

The most straightforward way to send a query to the PostgreSQL database.

Refers to a SQL query expressed as a simple string of text.

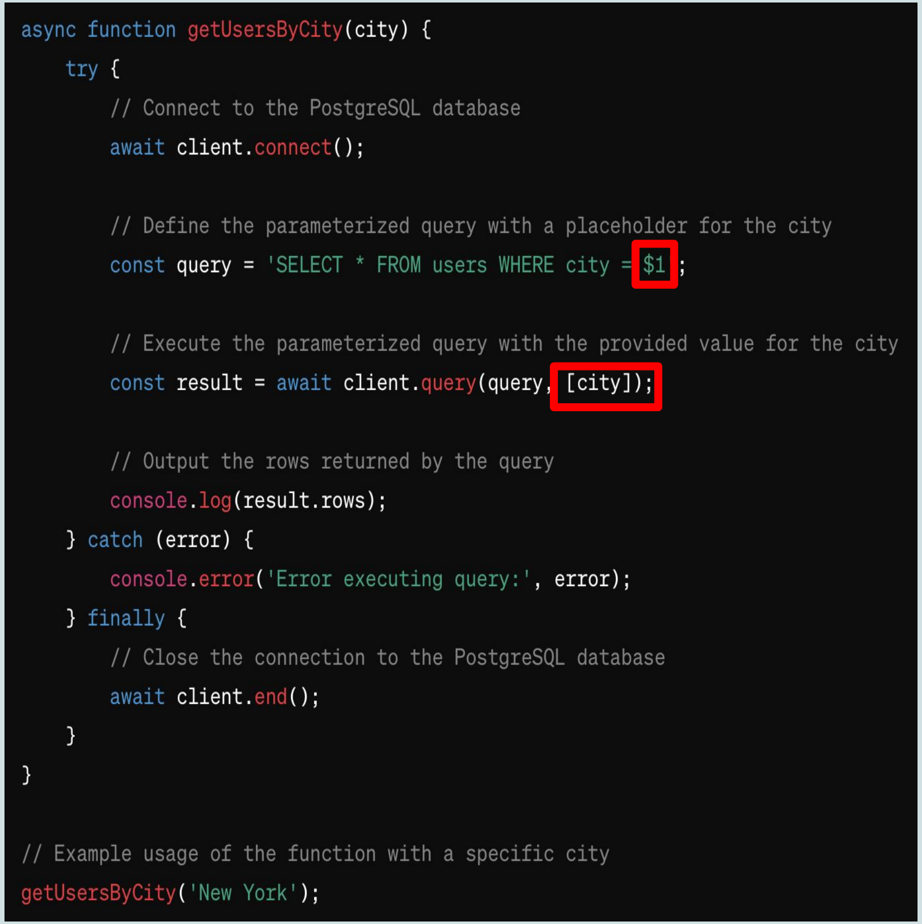


```
const result = await client.query('SELECT * FROM users');
```

Parameterized query

An SQL query where placeholders are used to represent the values that will be supplied at execution time. These placeholders are then filled with the actual values when the query is executed.

- This allows for safe and efficient handling of user input
- Prevents SQL injection attacks



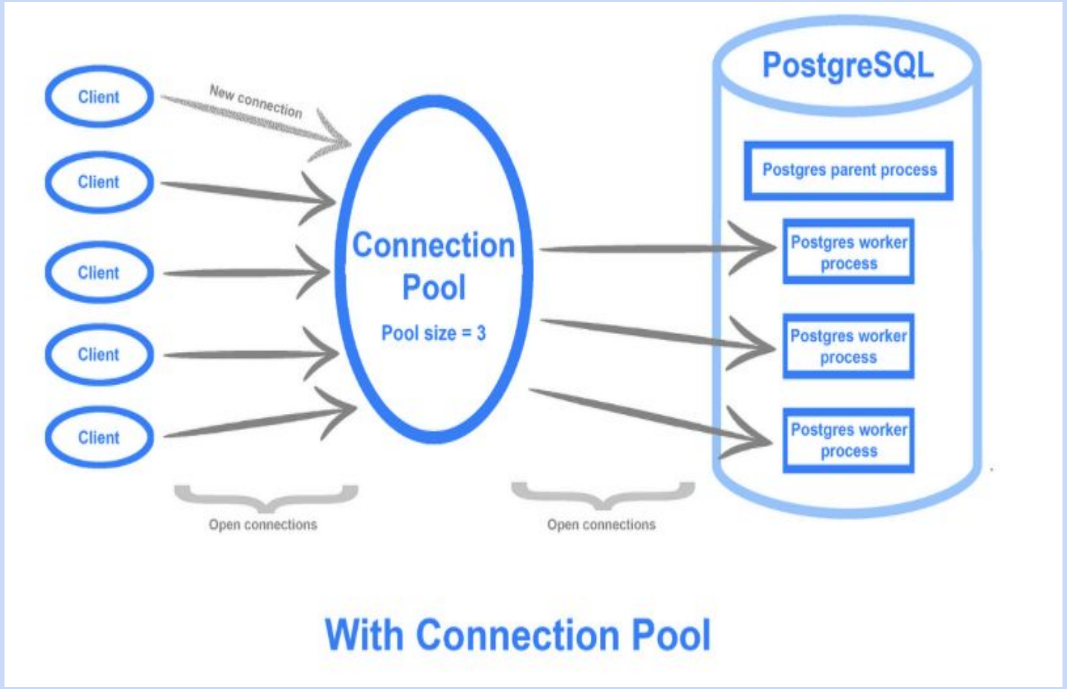
```
async function getUsersByCity(city) {
    try {
        // Connect to the PostgreSQL database
        await client.connect();

        // Define the parameterized query with a placeholder for the city
        const query = 'SELECT * FROM users WHERE city = $1';

        // Execute the parameterized query with the provided value for the city
        const result = await client.query(query, [city]);

        // Output the rows returned by the query
        console.log(result.rows);
    } catch (error) {
        console.error('Error executing query:', error);
    } finally {
        // Close the connection to the PostgreSQL database
        await client.end();
    }
}

// Example usage of the function with a specific city
getUsersByCity('New York');
```

There are other ways to pass an object to client.query. More information on this can be found here: https://node-postgres.com/apis/client

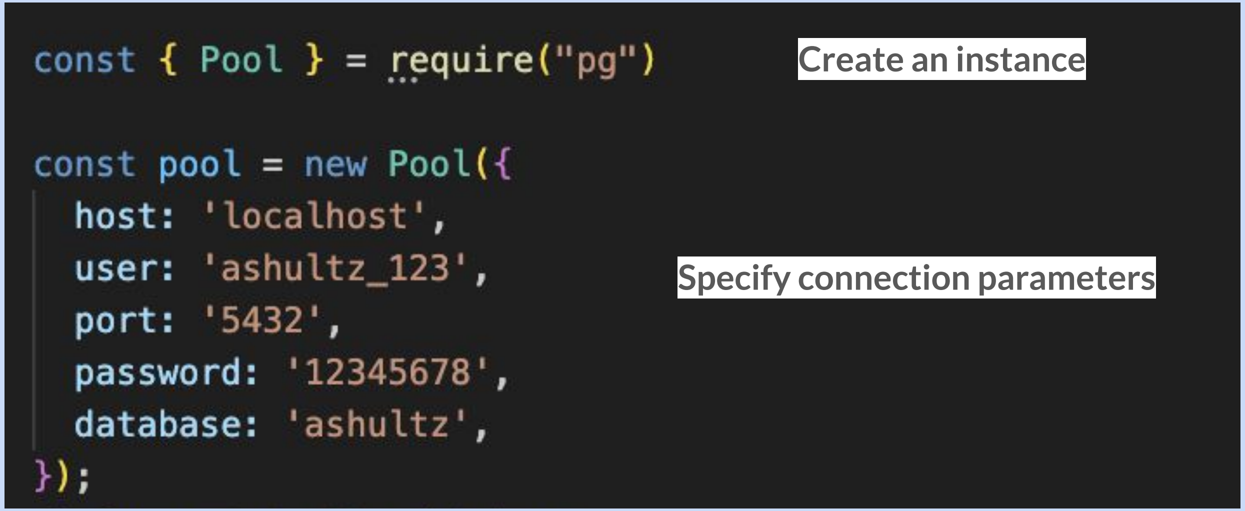
pg.Pool

A class that represents a pool of client connections to a PostgreSQL database.

- Instead of creating a new database connection for each query, a connection pool manages a set of reusable connections that can be shared among multiple queries



Create a Connection to the PostgreSQL Database

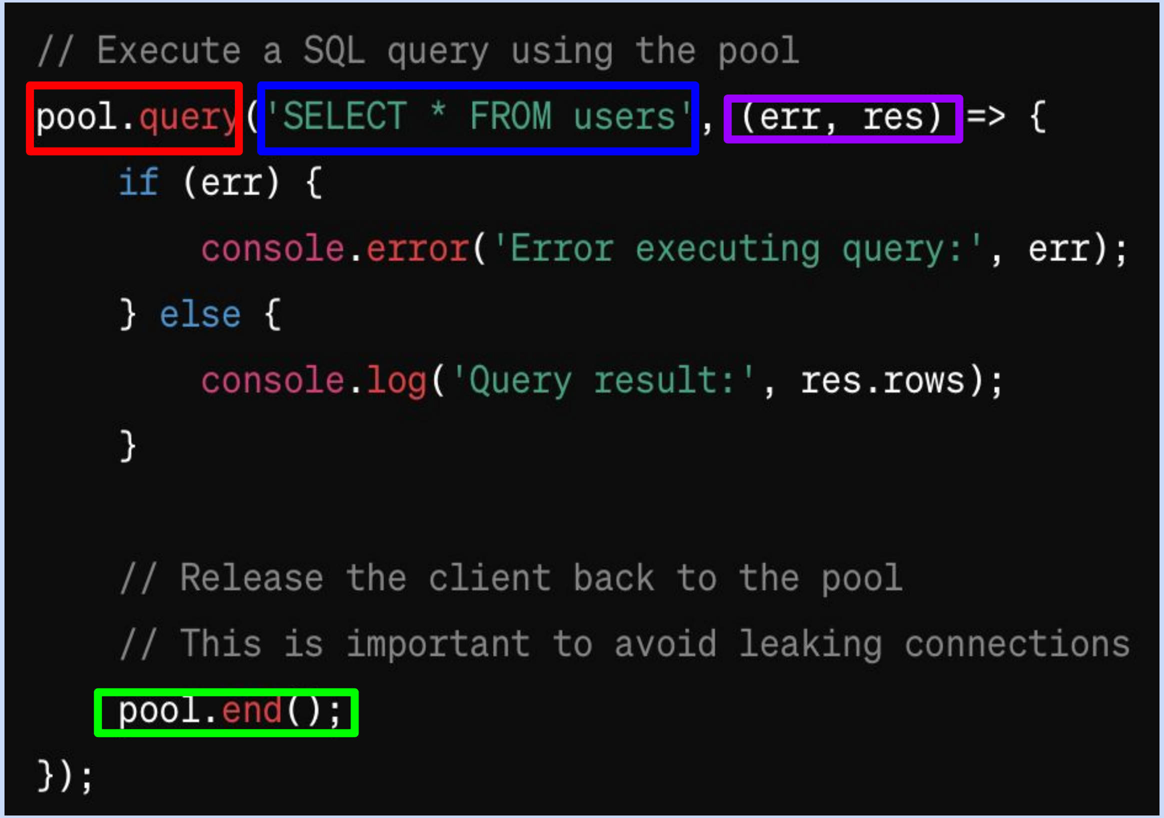


```
const { Pool } = require("pg")

const pool = new Pool({
  host: 'localhost',
  user: 'ashultz_123',
  port: '5432',
  password: '12345678',
  database: 'ashultz',
});
```

pool.query

The pool.query() method allows you to execute SQL queries against a PostgreSQL database using a connection from the connection pool.

1. Acquiring Connection
2. Executing Query
3. Handle the Result
4. Release Connection



```
// Execute a SQL query using the pool
pool.query('SELECT * FROM users', (err, res) => {
    if (err) {
        console.error('Error executing query:', err);
    } else {
        console.log('Query result:', res.rows);
    }

    // Release the client back to the pool
    // This is important to avoid leaking connections
    pool.end();
});
```

pool.connect

The pool.connect() method allows you to acquire a client connection from the connection pool.

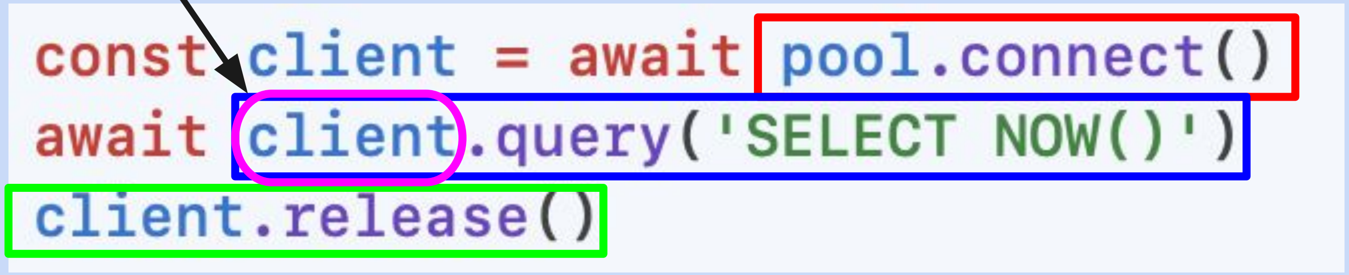
This method is used when you need to execute multiple queries within the same database connection rather than using the pool.query() method for each individual query.

1. Acquiring Connection

2. Executing Query

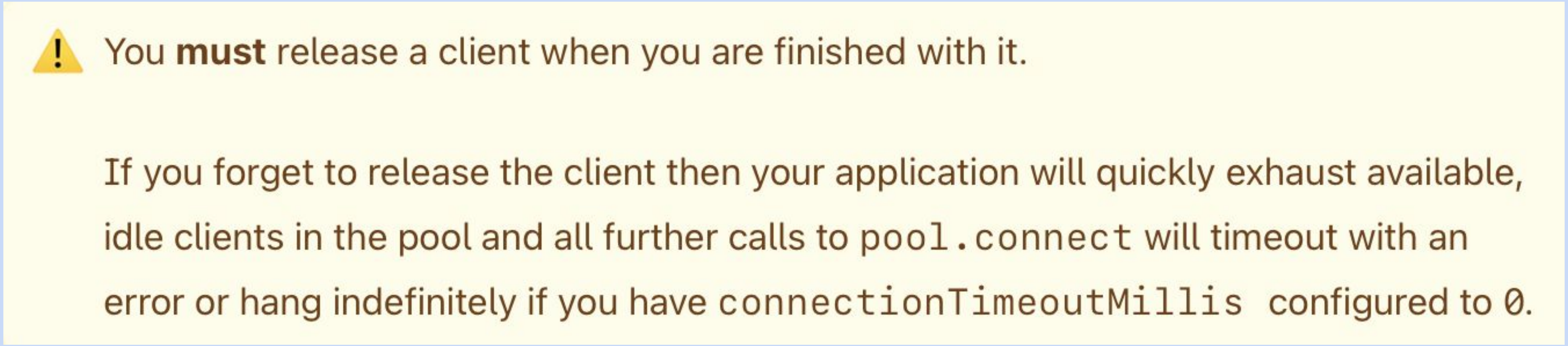
3. Release Connection

Acquired Connection



```
const client = await pool.connect()
await client.query('SELECT NOW()')
client.release()
```

Disclaimer

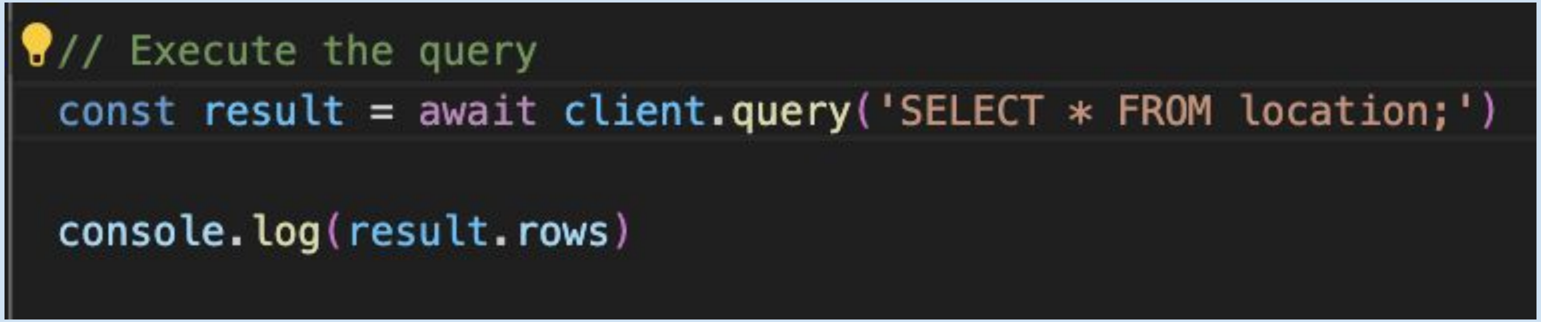

⚠️ You **must** release a client when you are finished with it.

If you forget to release the client then your application will quickly exhaust available, idle clients in the pool and all further calls to `pool.connect` will timeout with an error or hang indefinitely if you have `connectionTimeoutMillis` configured to `0`.

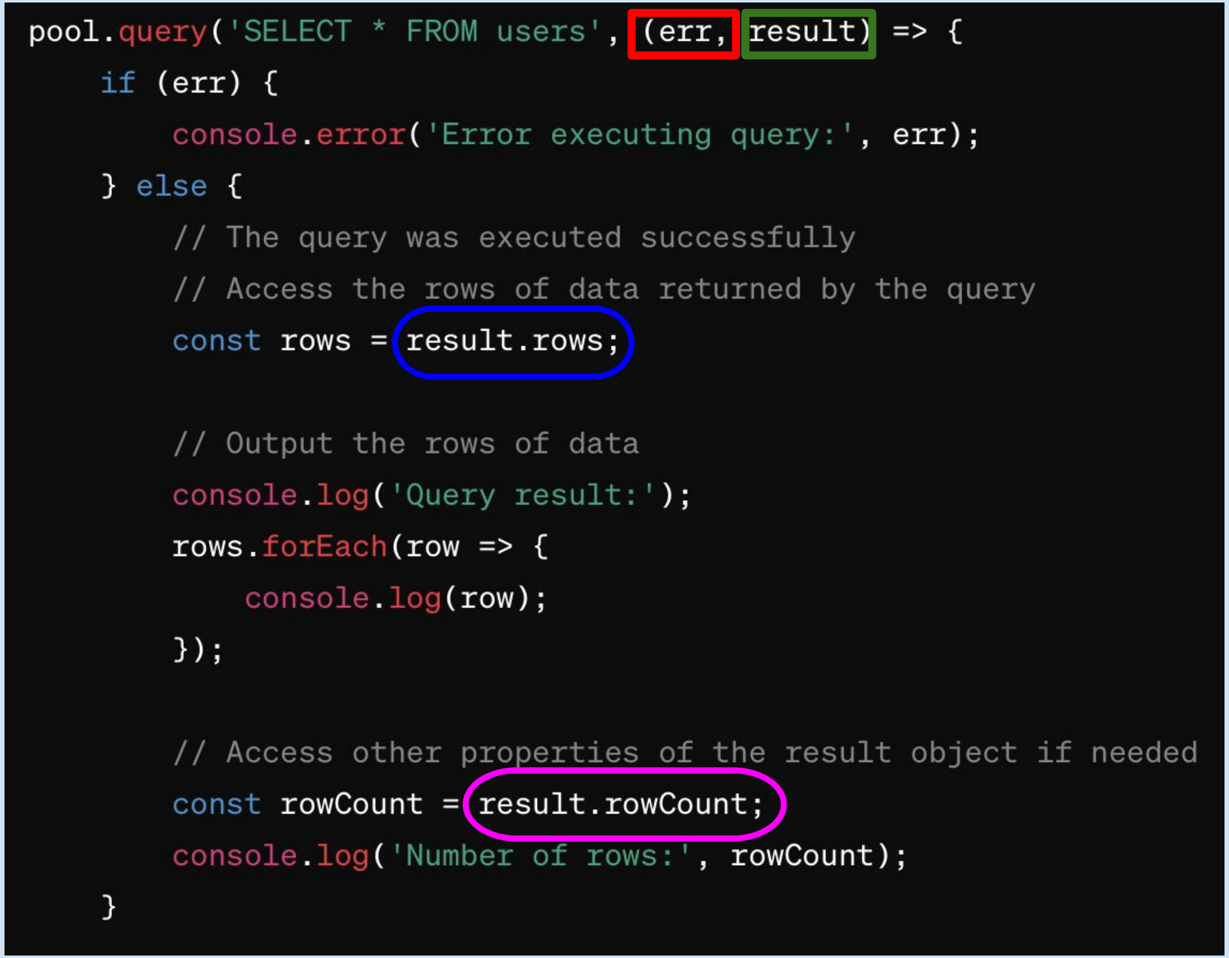
pg.Result

Represents the result of a successfully executed query from a PostgreSQL database.

It contains information about the outcome of the query, such as the rows of data returned by the query.



```
// Execute the query
const result = await client.query('SELECT * FROM location;')

console.log(result.rows)
```



```javascript
pool.query('SELECT * FROM users', (err, result) => {
    if (err) {
        console.error('Error executing query:', err);
    } else {
        // The query was executed successfully
        // Access the rows of data returned by the query
        const rows = result.rows;

        // Output the rows of data
        console.log('Query result:');
        rows.forEach(row => {
            console.log(row);
        });

        // Access other properties of the result object if needed
        const rowCount = result.rowCount;
        console.log('Number of rows:', rowCount);
    }
```

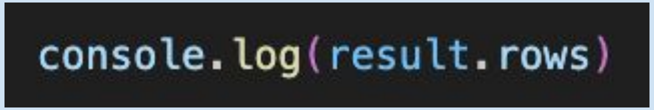
Query either returns:

1. An error
2. The result

Can use the .rows property to access the results

.rowCount is one of many other properties you can use. This returns the number of rows

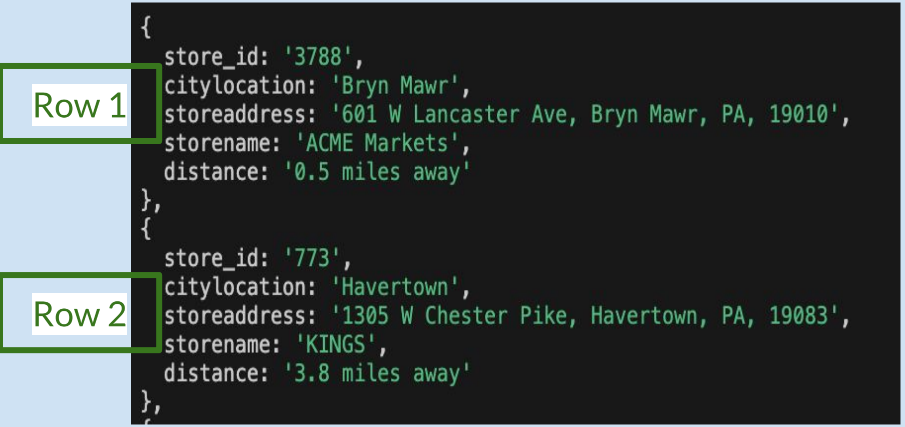
result.rows



```
console.log(result.rows)
```

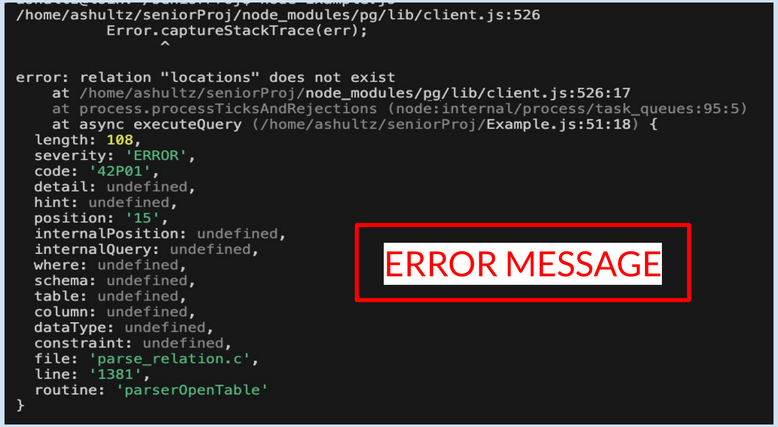
Every result will have a rows array.

- If no rows are returned → the array will be empty.
- Else the array will contain one item for each row returned from the query.



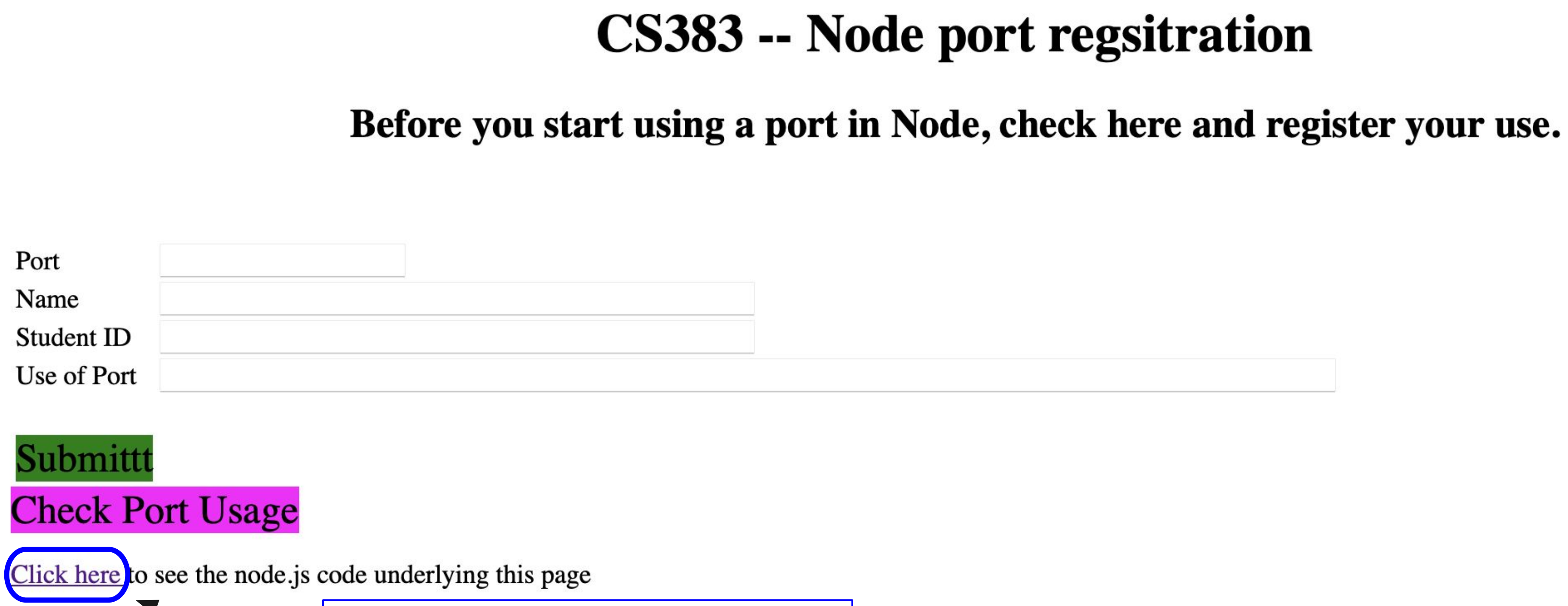
```
{
  store_id: '3788',
  citylocation: 'Bryn Mawr',
  storeaddress: '601 W Lancaster Ave, Bryn Mawr, PA, 19010',
  storename: 'ACME Markets',
  distance: '0.5 miles away'
},
{
  store_id: '773',
  citylocation: 'Havertown',
  storeaddress: '1305 W Chester Pike, Havertown, PA, 19083',
  storename: 'KINGS',
  distance: '3.8 miles away'
},
```

OR



```
/home/ashultz/seniorProj/node_modules/pg/lib/client.js:526
          Error.captureStackTrace(err);
                ^

error: relation "locations" does not exist
    at /home/ashultz/seniorProj/node_modules/pg/lib/client.js:526:17
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async executeQuery (/home/ashultz/seniorProj/Example.js:51:18) {
  length: 108,
  severity: 'ERROR',
  code: '42P01',
  detail: undefined,
  hint: undefined,
  position: '15',
  internalPosition: undefined,
  internalQuery: undefined,
  where: undefined,
  schema: undefined,
  table: undefined,
  column: undefined,
  dataType: undefined,
  constraint: undefined,
  file: 'parse_relation.c',
  line: '1381',
  routine: 'parserOpenTable'
}
```

An example application that uses Node.js and PostgreSQL: Geoff’s website



CS383 -- Node port regsitration

Before you start using a port in Node, check here and register your use.

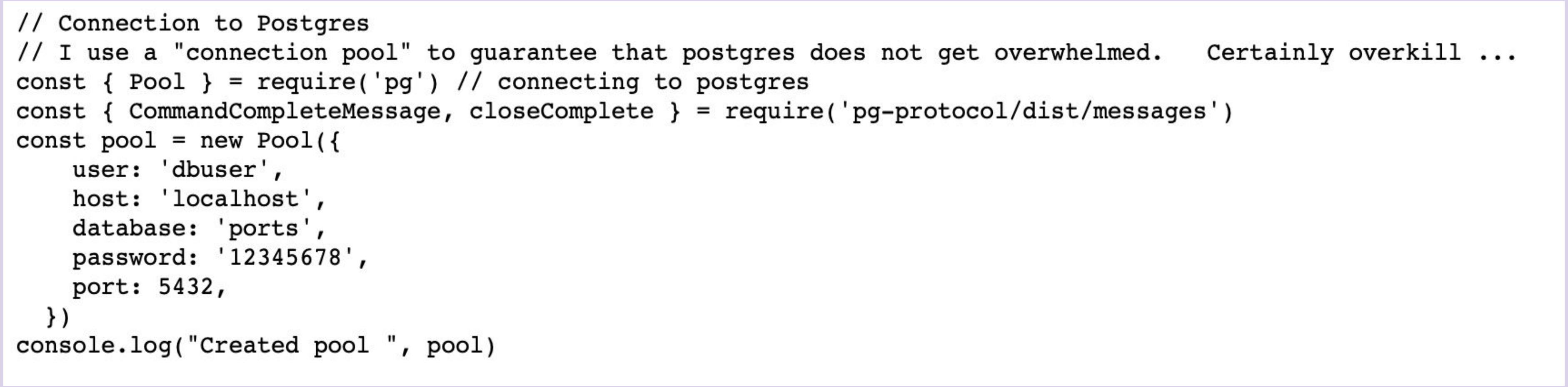
Port	
Name	
Student ID	
Use of Port	

Click here to see the node.js code underlying this page

Let's take a look at some snippets from the Node.js code that makes this webpage work

http://165.106.10.133:30006/index6.html

Code snippet 1:

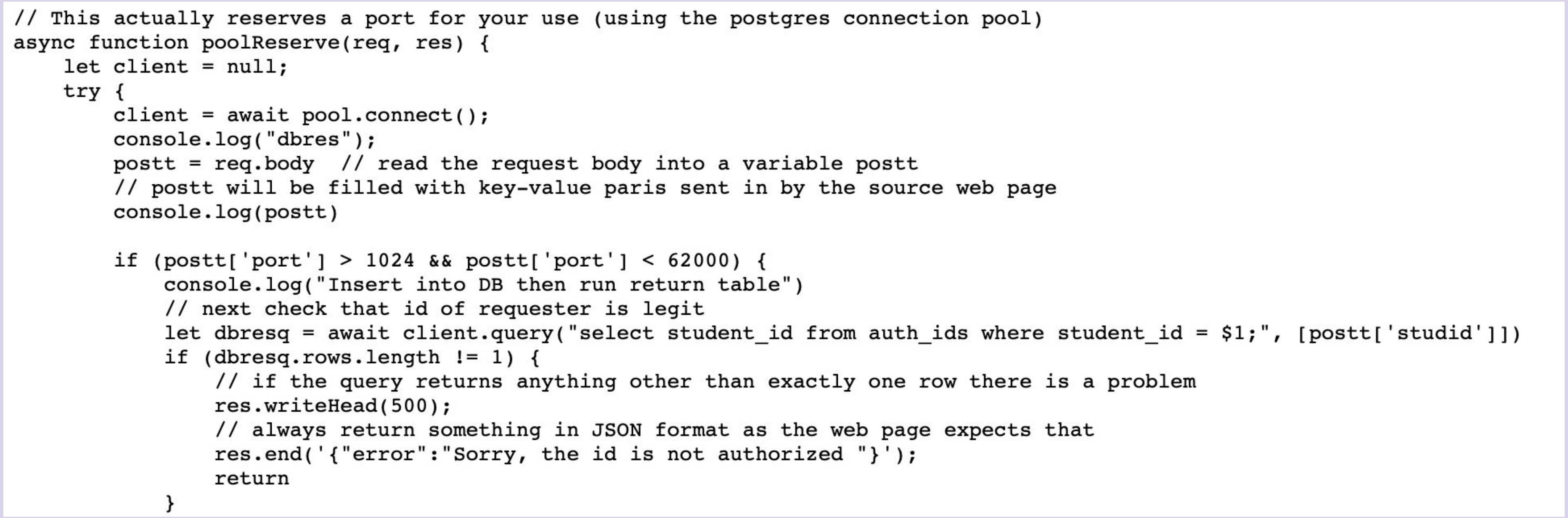


```
// Connection to Postgres
// I use a "connection pool" to guarantee that postgres does not get overwhelmed.   Certainly overkill ...
const { Pool } = require('pg') // connecting to postgres
const { CommandCompleteMessage, closeComplete } = require('pg-protocol/dist/messages')
const pool = new Pool({
    user: 'dbuser',
    host: 'localhost',
    database: 'ports',
    password: '12345678',
    port: 5432,
  })
console.log("Created pool ", pool)
```

What it does:

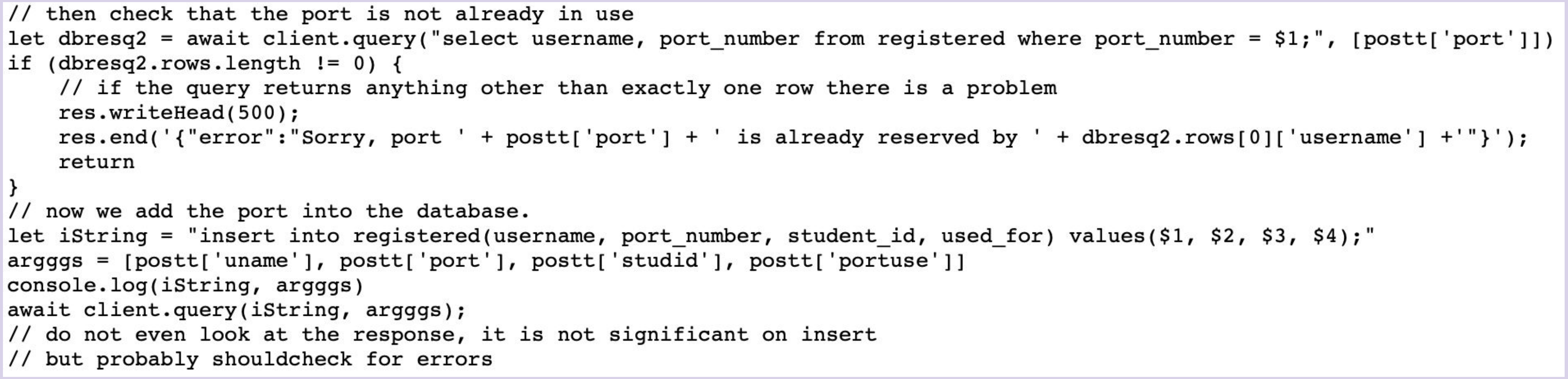
- Establishes a connection to a PostgreSQL database using a connection pool.
- Configures the pool with details such as username, host, database name, password, and port number.
- Logs a message confirming the creation of the pool in the console.

Code snippet 2:



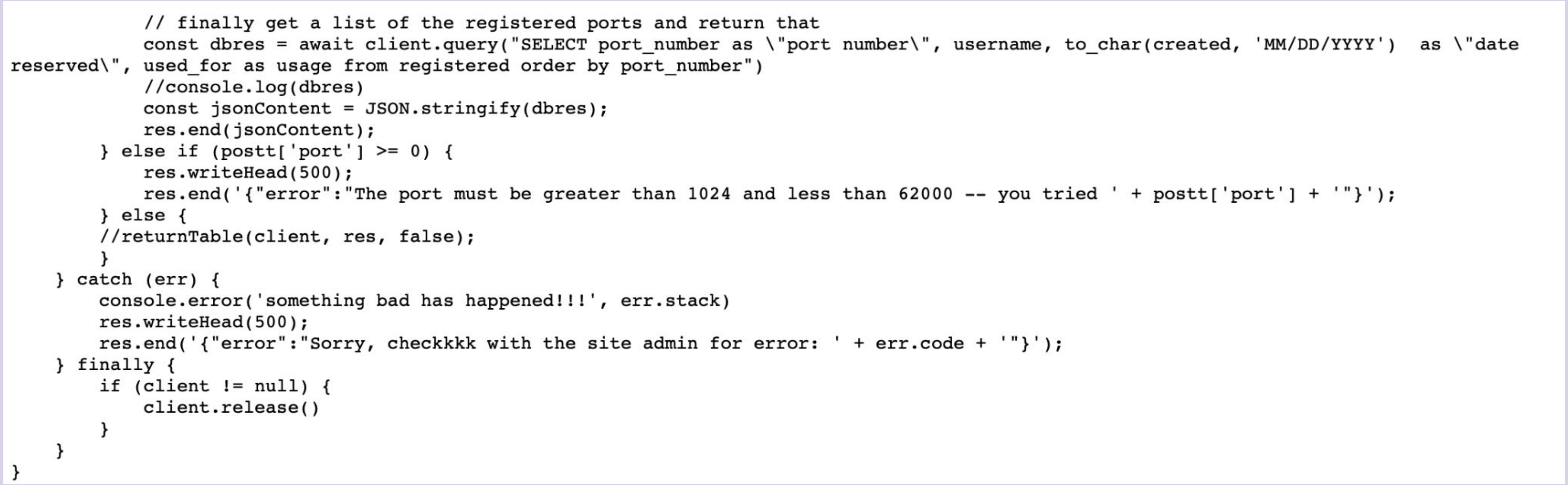
```
// This actually reserves a port for your use (using the postgres connection pool)
async function poolReserve(req, res) {
    let client = null;
    try {
        client = await pool.connect();
        console.log("dbres");
        postt = req.body  // read the request body into a variable postt
        // postt will be filled with key-value paris sent in by the source web page
        console.log(postt)

        if (postt['port'] > 1024 && postt['port'] < 62000) {
            console.log("Insert into DB then run return table")
            // next check that id of requester is legit
            let dbresq = await client.query("select student_id from auth_ids where student_id = $1;", [postt['studid']])
            if (dbresq.rows.length != 1) {
                // if the query returns anything other than exactly one row there is a problem
                res.writeHead(500);
                // always return something in JSON format as the web page expects that
                res.end('{"error":"Sorry, the id is not authorized "}');
                return
            }
```

Code snippet 3:



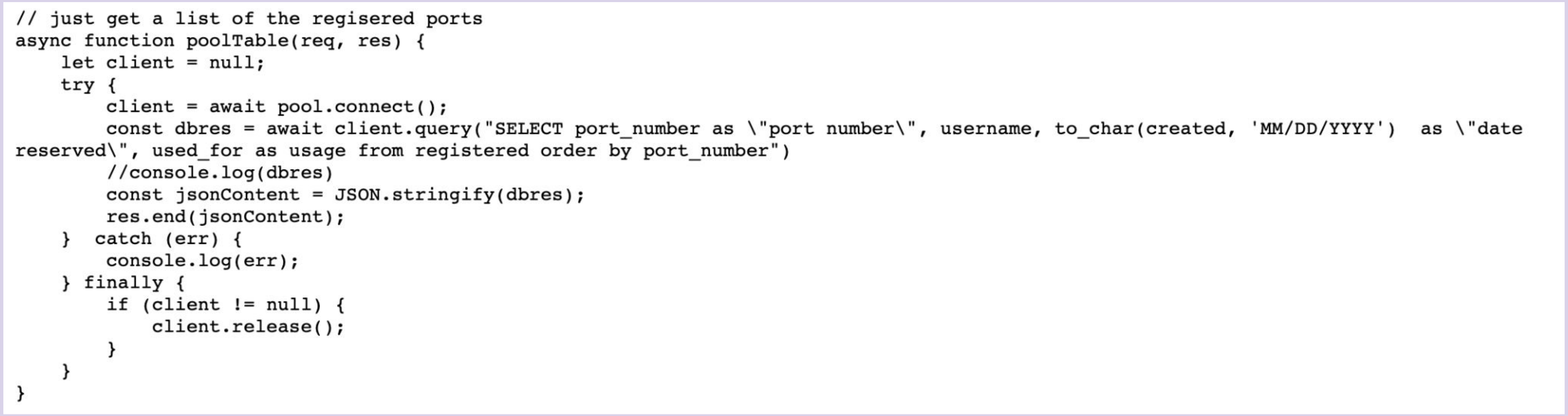
```
// then check that the port is not already in use
let dbresq2 = await client.query("select username, port_number from registered where port_number = $1;", [postt['port']])
if (dbresq2.rows.length != 0) {
    // if the query returns anything other than exactly one row there is a problem
    res.writeHead(500);
    res.end('{"error":"Sorry, port ' + postt['port'] + ' is already reserved by ' + dbresq2.rows[0]['username'] +'"}');
    return
}
// now we add the port into the database.
let iString = "insert into registered(username, port_number, student_id, used_for) values($1, $2, $3, $4);"
argggs = [postt['uname'], postt['port'], postt['studid'], postt['portuse']]
console.log(iString, argggs)
await client.query(iString, argggs);
// do not even look at the response, it is not significant on insert
// but probably shouldcheck for errors
```

Code snippet 4:



```
            // finally get a list of the registered ports and return that
            const dbres = await client.query("SELECT port_number as \"port number\", username, to_char(created, 'MM/DD/YYYY')  as \"date
reserved\", used_for as usage from registered order by port_number")
            //console.log(dbres)
            const jsonContent = JSON.stringify(dbres);
            res.end(jsonContent);
        } else if (postt['port'] >= 0) {
            res.writeHead(500);
            res.end('{"error":"The port must be greater than 1024 and less than 62000 -- you tried ' + postt['port'] + '"}');
        } else {
        //returnTable(client, res, false);
        }
    } catch (err) {
        console.error('something bad has happened!!!', err.stack)
        res.writeHead(500);
        res.end('{"error":"Sorry, checkkkk with the site admin for error: ' + err.code + '"}');
    } finally {
        if (client != null) {
            client.release()
        }
    }
}
```

Code snippet 5:



```
// just get a list of the regisered ports
async function poolTable(req, res) {
    let client = null;
    try {
        client = await pool.connect();
        const dbres = await client.query("SELECT port_number as \"port number\", username, to_char(created, 'MM/DD/YYYY')  as \"date
reserved\", used_for as usage from registered order by port_number")
        //console.log(dbres)
        const jsonContent = JSON.stringify(dbres);
        res.end(jsonContent);
    }  catch (err) {
        console.log(err);
    } finally {
        if (client != null) {
            client.release();
        }
    }
}
```