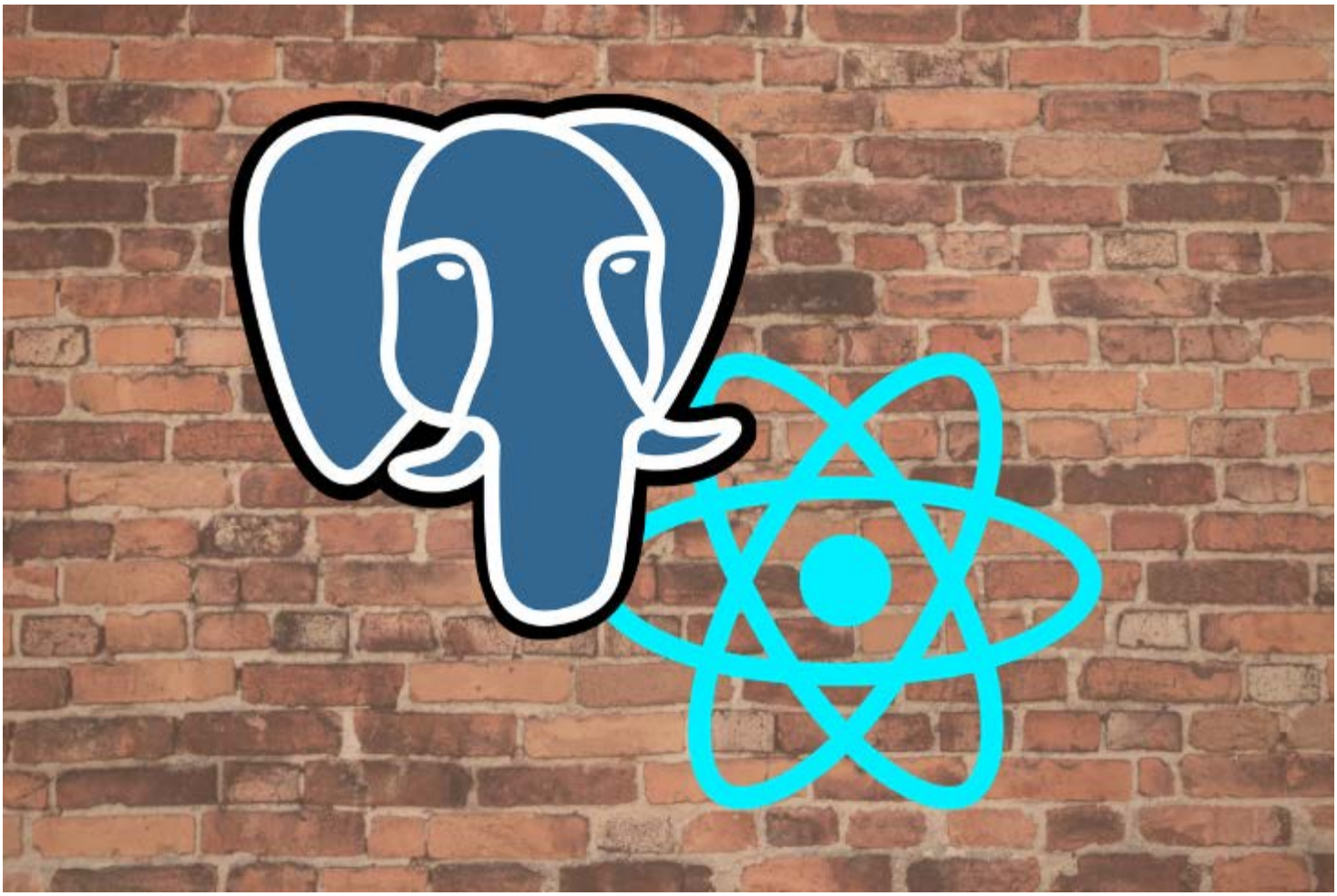
Getting started with Postgres in your React app

March 19, 2020 7 min read



Frontend developers typically don’t need to understand the process of receiving, recording, and removing information. That’s a job for backend developers.

That said, there are plenty of good reasons for a frontend developer to learn about backend programming and database interaction. For instance:

- You’ll stand out from other frontend developers because you’ll know how your application work as a whole
- You’ll be able to work on both the front and back side of your app
- You can be promoted to a full-stack developer and take on a bigger role with a higher salary
- Knowledge of both frontend and backend programming — as well as designing scalable systems and building solid application architecture — is a requirement to be a tech lead

In this tutorial, we’ll demonstrate how to create a small application using Express and Node.js that can record and remove information from a PostgreSQL database according to the HTTP requests it receives. We’ll then create a simple React app to test and see how the entire application flows from front to back.

I'll assume that you understand how a React application works and are familiar with frontend JavaScript HTTP requests. We won't cover how to validate data before interacting with the database. Instead, we'll focus on showing how requests from the interface are recorded into a database.

I published a GitHub repo for this tutorial so you can compare your code if you get stuck. Now let's get our database running.

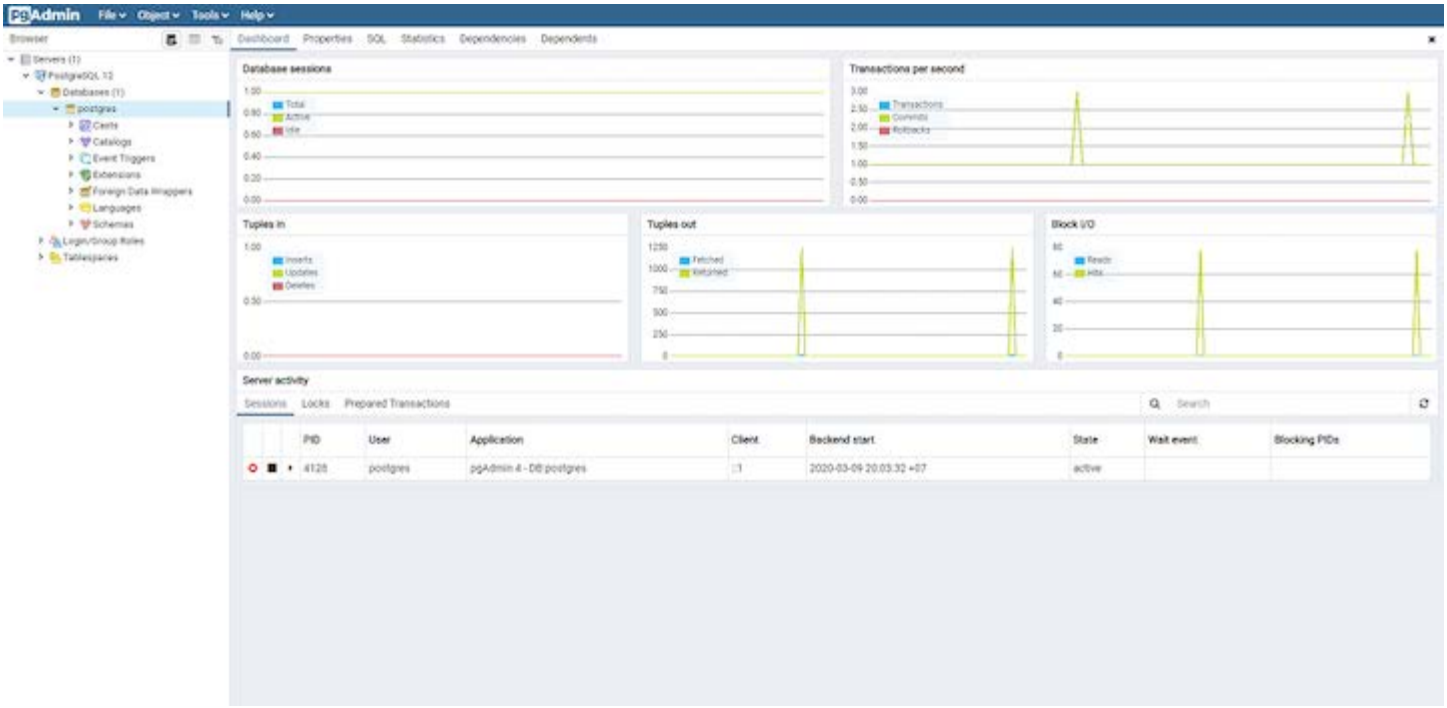
Setting up PostgreSQL

PostgreSQL, or Postgres, is a relational database management system that claims to be the world's most advanced open-source relational database. It has been maintained since 1996 and has a reputation for being reliable and robust.

Start by downloading and installing PosgreSQL. It supports all major operating systems, so choose the right one for your computer and follow the instructions to set up the database. The setup wizard will prompt you to enter a superuser password. Make sure you remember this password; you'll need it to login later.

Once the installation is complete, you can access your database by using pgAdmin, a graphical interface tool database management that is installed automatically with PostgreSQL.

Once opened, pgAdmin will ask for your password to log in. Below is the overview of a newly installed PostgreSQL database.



Creating a Postgres database

To better understand SQL language, we need to create a database and table from the terminal.

To access PostgreSQL from the terminal, use the command `psql` with the option `-d` to select the database you want to access and `-U` to select the user. If the terminal replies that the `psql` command is not found, you'll most likely need to add the Postgres `bin/` and `lib/` directories into your system path.

```
psql -d postgres -U postgres
```

You will be asked to input your password. Use the password you created earlier. Once you're logged in, create a new user by adding a login permission with the password "root."

```
CREATE ROLE my_user WITH LOGIN PASSWORD 'root';
```

A user is simply a role that has login permission. Now that you have one, give it permission to create databases by issuing the `ALTER ROLE [role name] CREATEDB` syntax.

```
ALTER ROLE my_user CREATEDB;
```

Log out from your `postgres` superuser and log in as `my_user` using the command `\q`.

```
\q
psql -d postgres -U my_user
```

Now that you're back inside, create a new database named `my_database`.

```
CREATE DATABASE my_database;
```

You might be wondering, why can't we just use the default `postgres` user to create the database? That's because the default user is a superuser, which means it has access to everything within the database. According to the Postgres documentation, "superuser status is dangerous and should be used only when really needed."

An SQL-based database stores data inside a table. Now that you have a database, let's create a simple table where you can record your data.

```
CREATE TABLE merchants( id SERIAL PRIMARY KEY, name VARCHAR(30), email VARCHAR(30) );
```

One database can have multiple tables, but we'll be fine with one table for this tutorial. If you'd like to check the

created database and table, you can use the command `\list` and `\dt`, respectively. You might see more rows or less, but as long as you have the database and the table you created previously, your table should look like this:

```
my_database=> \list
          List of databases
 Name        |  Owner   | Encoding
 my_database | my_user  | UTF8
 postgres    | postgres | UTF8
 template0   | postgres | UTF8
 template1   | postgres | UTF8


my_database=> \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | merchants | table | my_user
```

Now have a table into which you can insert data. Let's do that next.

Basic SQL queries

Postgres is an SQL-based system, which means you need to use SQL language to store and manipulate its data. Let's explore four basic example of SQL queries you can use.

1. Select query

To retrieve data from a table, use the `SELECT` key, followed by the name of the columns you want to retrieve and the name of the table.

```
SELECT id, name, email from merchants;
```

To retrieve all columns in the table, you can simply use `SELECT *`.

```
SELECT * from merchants;
```

2. Insert query

To insert new data into a table, use the `INSERT` keyword followed by the table name, column name(s), and values.

```
INSERT INTO merchants (name, email) VALUES ('john', 'john@mail.com');
```

3. Delete query

You can delete a row from a table by using the `DELETE` keyword.

```
DELETE from merchants WHERE id = 1;
```

When you use the delete query, don't forget to specify which row you want to delete with the `WHERE` keyword. Otherwise, you'll delete all the rows in that table.

4. Update query

To update a certain row, you can use the `UPDATE` keyword.

```
UPDATE merchants SET name = 'jake', email = 'jake@mail.com' WHERE id = 1;
```

Now that you know how to manipulate data inside your table, let's examine how to connect your database to React.

Creating an API server with Node.js and Express

To connect your React app with a PostgreSQL database, you must first create an API server that can process HTTP requests. Let's set up a simple one using NodeJS and Express.

Create a new directory and set a new npm package from your terminal with the following commands.

```
mkdir node-postgres && cd node-postgres
npm init
```

You can fill in your package information as you like, but here is an example of my `package.json`:

```
{
  "name": "node-postgres",
  "version": "1.0.0",
  "description": "Learn how NodeJS and Express can interact with PostgreSQL",
  "main": "index.js",
  "license": "ISC"
```

```
}
```

Next, install the required packages.

```
npm i express pg
```

Express is a minimalist web framework you can use to write web applications on top of Node.js technology, while `node-postgres(pg)` is a client library that enables Node.js apps to communicate with PostgreSQL.

Once both are installed, create an `index.js` file with the following content.

```
const express = require('express')
const app = express()
const port = 3001

app.get('/', (req, res) => {
  res.status(200).send('Hello World!');
})

app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})
```

Open your terminal in the same directory and run `node index.js`. Your Node application will run on port 3001, so open your browser and navigate to http://localhost:3001. You'll see "Hello World!" text displayed in your browser.

You now have everything you need to write your API.

Making NodeJS talk with Postgres

The `pg` library allows your Node application to talk with Postgres, so you'll want to import it first. Create a new file named `merchant_model.js` and input the following code.

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'my_user',
  host: 'localhost',
```

```
  database: 'my_database',
  password: 'root',
  port: 5432,
});
```

Please note that putting credentials such as user, host, database, password, and port like in the example above is not recommended in a production environment. We'll keep it in this file to simplify the tutorial.

The pool object you created above will allow you to query into the database that it's connected to. Let's create three queries to make use of this pool. These queries will be placed inside a function, which you can call from your `index.js`.

```
const getMerchants = () => {
  return new Promise(function(resolve, reject) {
    pool.query('SELECT * FROM merchants ORDER BY id ASC', (error, results) => {
      if (error) {
        reject(error)
      }
      resolve(results.rows);
    })
  })
}
const createMerchant = (body) => {
  return new Promise(function(resolve, reject) {
    const { name, email } = body
    pool.query('INSERT INTO merchants (name, email) VALUES ($1, $2) RETURNING *',
[name, email], (error, results) => {
      if (error) {
        reject(error)
      }
```

The code above will process and export the `getMerchants`, `createMerchant`, and `deleteMerchant` functions. Now it's time to update your `index.js` file and make use of these functions.

```
const express = require('express')
const app = express()
```

```
const port = 3001

const merchant_model = require('./merchant_model')

app.use(express.json())
app.use(function (req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', 'http://localhost:3000');
  res.setHeader('Access-Control-Allow-Methods', 'GET,POST,PUT,DELETE,OPTIONS');
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Access-Control-
Allow-Headers');
  next();
});

app.get('/', (req, res) => {
  merchant_model.getMerchants()
  .then(response => {
```

Now your app has three HTTP routes that can accept requests. The code from line 7 is written so that Express can accept incoming requests with JSON payloads. To allow requests to this app from React, I also added headers for `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, and `Access-Control-Allow-Headers`.

Creating your React application

Your API is ready to serve and process requests. Now it's time to create a React application to send requests into it.

Let's bootstrap your React app with the `create-react-app` command.

```
npx create-react-app react-postgres
```

In your React app directory, you can remove everything inside the `src/` directory.

Now let's write a simple React app from scratch.

First, create an `App.js` file with the following content.

```
import React, {useState, useEffect} from 'react';
```

```
function App() {
  const [merchants, setMerchants] = useState(false);
  useEffect(() => {
    getMerchant();
  }, []);
  function getMerchant() {
    fetch('http://localhost:3001')
      .then(response => {
        return response.text();
      })
      .then(data => {
        setMerchants(data);
      });
  }
  function createMerchant() {
    let name = prompt('Enter merchant name');
    let email = prompt('Enter merchant email');
```

This React app will send requests to the Express server you created. It has two buttons to add and delete a merchant. The function `getMerchant` will fetch merchant data from the server and set the result to the `merchant` state.

`createMerchant` and `deleteMerchant` will start the process to add and remove merchants, respectively, when you click on the buttons.

Finally, create an `index.js` file and render the `App` component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

Now run your React app with `npm start`. You can test and see how the data collected from your React application is recorded into PostgreSQL. I'll leave you the implementation of the `UPDATE` query as an exercise.

Conclusion

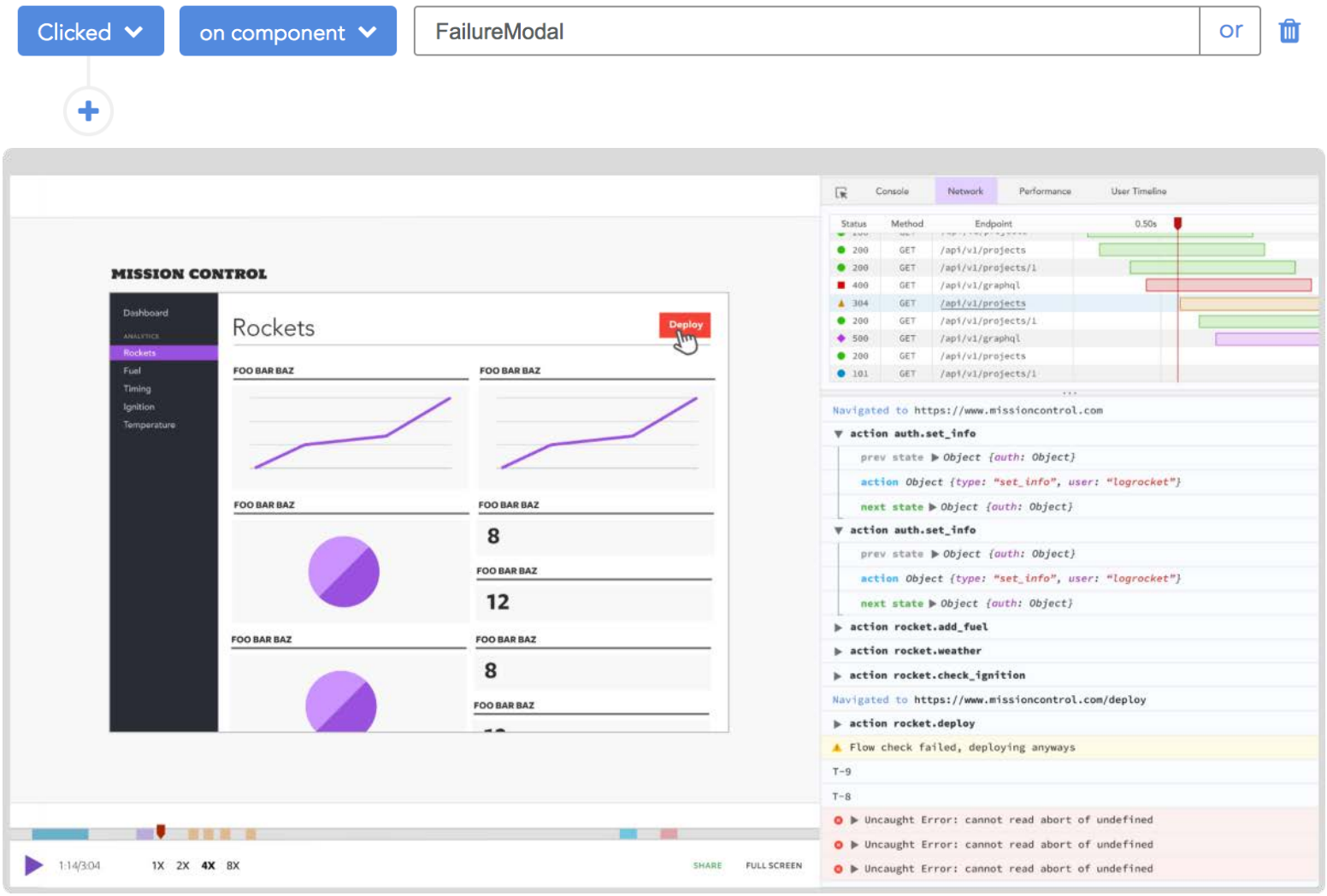
Now you know how to install PostgreSQL database, create a database and table, and build a minimal API to function as a bridge between your React app and your database. We created an end-to-end example of how to use Postgres with React and demonstrated exactly what happens when you send those HTTP requests from your React app.

This tutorial is far from a complete guide to backend programming, but it's enough to help you get started in understanding how the backend side works.

Should you find any error or can't insert data from the application, you can clone the complete source code from this repo and compare the code from the repo with your example.

Full visibility into production React apps

Debugging React applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.



LogRocket is like a DVR for web and mobile apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when

an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Modernize how you debug your React apps — start monitoring for free.

Share this: