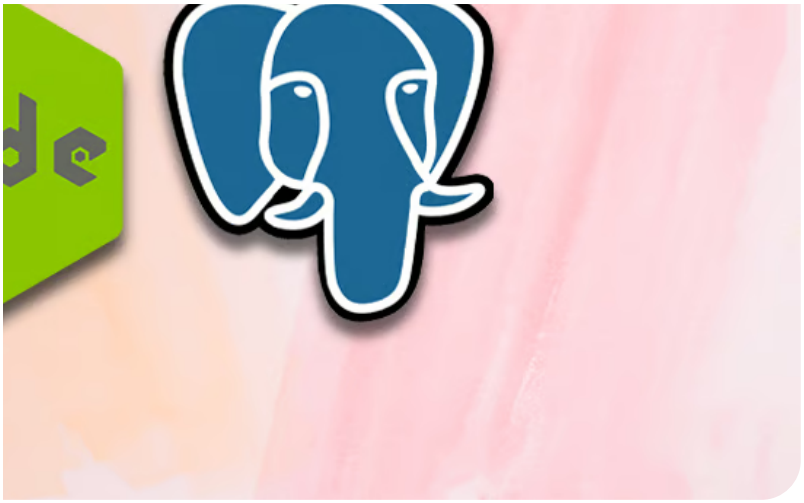
Jan 26, 2024 · 15 min read

CRUD REST API with Node.js, Express, and PostgreSQL

Tania Rascia
Software developer, writer, maker of things. Find me online at **tania.dev**.

Table of contents

Editor's note: *This post was last updated by Emmanuel John on 26 January 2024 to discuss securing the API and provide solutions to two common issues developers may encounter while developing APIs. A section was also added to cover potential next steps for developers. This article was previously updated on 6 June 2022 to reflect updates to the pgAdmin client.*



ommunication between software systems is crucial
ıis tutorial, we'll create a CRUD RESTful API in a
n an Express server and uses a PostgreSQL database.

ng an Express server with PostgreSQL using node-
handle the HTTP request methods that correspond
which the API gets its data. You'll also learn how to install PostgreSQL and work with it through the CLI.

Our goal is to allow CRUD operations — `GET`, `POST`, `PUT`, and `DELETE` — on the API, which will run the corresponding database commands. To do so, we'll set up a route for each endpoint and a function for each query.

To follow along with this tutorial, you'll need:

- Familiarity with the JavaScript syntax and fundamentals
- Basic knowledge of working with the command line
- Node.js and npm installed

The complete code for the tutorial is available in this GitHub repo. Let's get started!

What is a RESTful API?

Representational State Transfer (REST) defines a set of standards for web services.

An API is an interface that software programs use to communicate with each other. Therefore, a RESTful API is an API that conforms to the REST architectural style and constraints.

REST systems are stateless, scalable, cacheable, and have a uniform interface.

What is a CRUD API?

When building an API, you want your model to provide four basic functionalities. It should be able to create, read, update, and delete resources. This set of essential operations is commonly referred to as CRUD.

RESTful APIs most commonly utilize HTTP requests. Four of the most common HTTP methods in a REST environment are `GET`, `POST`, `PUT`, and `DELETE`, which are the methods by which a developer can create a CRUD system:

- `Create`: Use the `HTTP POST` method to create a resource in a REST environment
- `Read`: Use the `GET` method to read a resource, retrieving data without altering it

- `Update` : Use the `PUT` method to update a resource
- `Delete` : Use the `DELETE` method to remove a resource from the system

What is Express?

According to the official Express documentation, Express is a fast, unopinionated, minimalist web framework for Node.js. Express is one of the most popular frameworks for Node.js. In fact, each E in the MERN, MEVN, and MEAN stacks stands for Express.

Although Express is minimalist, it's also very flexible. This supports the development of various Express middlewares that you can use to address almost any task or problem imaginable.

What is PostgreSQL?

PostgreSQL, commonly referred to as Postgres, is a free and open source relational database management system. You might be familiar with a few other similar database systems, like MySQL, Microsoft SQL Server, or MariaDB, which compete with PostgreSQL.

PostgreSQL is a robust relational database that has been around since 1997. It's available on all major operating systems — Linux, Windows, and macOS. Since PostgreSQL is known for stability, extensibility, and standards compliance, it's a popular choice for developers and companies.

It's also possible to create a Node.js RESTful CRUD API using Sequelize. Sequelize is a promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server.

For more on how to use Sequelize in a Node.js REST API, check out the video

tutorial below:

YOUTUBE DAVID TANG

What is node-postgres?

node-postgres, or pg, is a nonblocking PostgreSQL client for Node.js. Essentially, node-postgres is a collection of Node.js modules for interfacing with a PostgreSQL database.

node-postgres supports many features, including callbacks, promises, async/await, connection pooling, prepared statements, cursors, rich type parsing, and C/C++ bindings.

Creating a PostgreSQL database

We'll begin this tutorial by installing PostgreSQL, creating a new user, creating a database, and initializing a table with a schema and some data.

Installation

If you're using Windows, download a Windows installer of PostgreSQL.

If you're using a Mac, this tutorial assumes you have Homebrew installed on your computer as a package manager for installing new programs. If you don't, simply click on the link and follow the instructions.

Open up the terminal and install `postgresql` with `brew`:

```
brew install postgresql
```

You may see instructions on the web reading `brew install postgres` instead of `PostgreSQL`. Both options will install PostgreSQL on your computer.

After the installation is complete, we'll want to get `postgresql` up and running, which we can do with `services start`:

```
brew services start postgresql
==> Successfully started `postgresql` (label: homebrew.mxcl.postgre
```

If at any point you want to stop the `postgresql` service, you can run `brew services stop postgresql`.

With PostgreSQL installed, let's next connect to the `postgres` command line where we can run SQL commands.

PostgreSQL command prompt

`psql` is the PostgreSQL interactive terminal. Running `psql` will connect you to a PostgreSQL host. Running `psql --help` will give you more information about the available options for connecting with `psql`:

- `-h` or `--host=HOSTNAME`: The database server host or socket directory; the default is `local socket`
- `-p` or `--port=PORT`: The database server port; the default is `5432`
- `-U` or `--username=USERNAME`: The database username; the default is `your_username`
- `-w` or `--no-password`: Never prompt for password
- `-W` or `--password`: Force password prompt, which should happen automatically

We'll connect to the default `postgres` database with the default login information and no option flags:

```
psql postgres
```

You'll see that we've entered into a new connection. We're now inside `psql` in the `postgres` database. The prompt ends with a `#` to denote that we're logged in as the superuser, or root:

```
postgres=#
```

Commands within `psql` start with a backslash `\`. To test our first command, we can check what database, user, and port we've connected to using the `\conninfo` command:

```
postgres=# \conninfo
You are connected to database "postgres" as user "your_username" vi
```

The reference table below includes a few common commands that we'll use throughout this tutorial:

- `\q`: Exit `psql` connection
- `\c`: Connect to a new database
- `\dt`: List all tables
- `\du`: List all roles
- `\list`: List databases

Let's create a new database and user so we're not using the default accounts, which have superuser privileges.

Creating a role in Postgres

First, we'll create a role called `me` and give it a password of `password`. A role can function as a user or a group. In this case, we'll use it as a user:

```
postgres=# CREATE ROLE me WITH LOGIN PASSWORD 'password';
```

We want `me` to be able to create a database:

```
postgres=# ALTER ROLE me CREATEDB;
```

You can run `\du` to list all roles and users:

```
me          | Create DB                          | {}
postgres    | Superuser, Create role, Create DB  | {}
```

Now, we want to create a database from the `me` user. Exit from the default session with `\q` for quit:

```
postgres=# \q
```

We're back in our computer's default terminal connection. Now, we'll connect `postgres` with `me`:

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

Instead of `postgres=#`, our prompt now shows `postgres=>`, meaning we're no longer logged in as a superuser.

Creating a database in Postgres

We can create a database with the SQL command as follows:

```
postgres=> CREATE DATABASE api;
```

Use the `\list` command to see the available databases:

```
Name    |    Owner    | Encoding |   Collate   |    Ctype    |
api     | me          | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

Let's connect to the new `api` database with `me` using the `\c` connect command:

```
postgres=> \c api
You are now connected to database "api" as user "me".
api=>
```

Our prompt now shows that we're connected to `api` .

More great articles from LogRocket:

- Don't miss a moment with The Replay, a curated newsletter from LogRocket
- Learn how LogRocket's Galileo AI watches sessions for you and proactively surfaces the highest-impact things you should work on
- Use React's useEffect to optimize your application's performance
- Switch between multiple versions of Node
- Discover how to use the React children prop with TypeScript
- Explore creating a custom mouse cursor with CSS
- Advisory boards aren't just for executives. Join LogRocket's Content Advisory Board. You'll help inform the type of content we create and get access to exclusive meetups, social accreditation, and swag

---

Creating a table in Postgres

Finally, in the `psql` command prompt, we'll create a table called `users` with three fields, two `VARCHAR` types, and an auto-incrementing `PRIMARY KEY` ID:

```
api=>
CREATE TABLE users (
  ID SERIAL PRIMARY KEY,
  name VARCHAR(30),
  email VARCHAR(30)
);
```

Make sure not to use the backtick ` character when creating and working with tables in PostgreSQL. While backticks are allowed in MySQL, they're not valid in PostgreSQL. Also, ensure that you do not have a trailing comma in the CREATE TABLE command.

Let's add some data to work with by adding two entries to `users`:

```
INSERT INTO users (name, email)
  VALUES ('Jerry', 'jerry@example.com'), ('George', 'george@example
```

Let's make sure that the information above was correctly added by getting all entries in `users`:

```
api=> SELECT * FROM users;
id |  name  |       email
----+--------+--------------------
  1 | Jerry  | jerry@example.com
  2 | George | george@example.com
```

Now, we have a user, database, table, and some data. We can begin building our

Node.js RESTful API to connect to this data, stored in a PostgreSQL database.

At this point, we're finished with all of our PostgreSQL tasks, and we can begin setting up our Node.js app and Express server.

Setting up an Express server

To set up a Node.js app and Express server, first create a directory for the project to live in:

```
mkdir node-api-postgres
cd node-api-postgres
```

You can either run `npm init -y` to create a `package.json` file, or copy the code below into a `package.json` file:

```
{
  "name": "node-api-postgres",
  "version": "1.0.0",
  "description": "RESTful API with Node.js, Express, and PostgreSQL
  "main": "index.js",
  "license": "MIT"
}
```

We'll want to install Express for the server and node-postgres to connect to PostgreSQL:

```
npm i express pg
```

Now, we have our dependencies loaded into `node_modules` and `package.json`.

Create an `index.js` file, which we'll use as the entry point for our server. At the top, we'll require the `express` module, the built-in `body-parser` middleware,

and we'll set our `app` and `port` variables:

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const port = 3000

app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
)
```

We'll tell a route to look for a `GET` request on the root `/` URL and return some JSON:

```
app.get('/', (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
})
```

Now, set the app to listen on the port you set:

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

From the command line, we can start the server by hitting `index.js`:

```
node index.js
App running on port 3000.
```

Go to `http://localhost:3000` in the URL bar of your browser, and you'll see the JSON we set earlier:

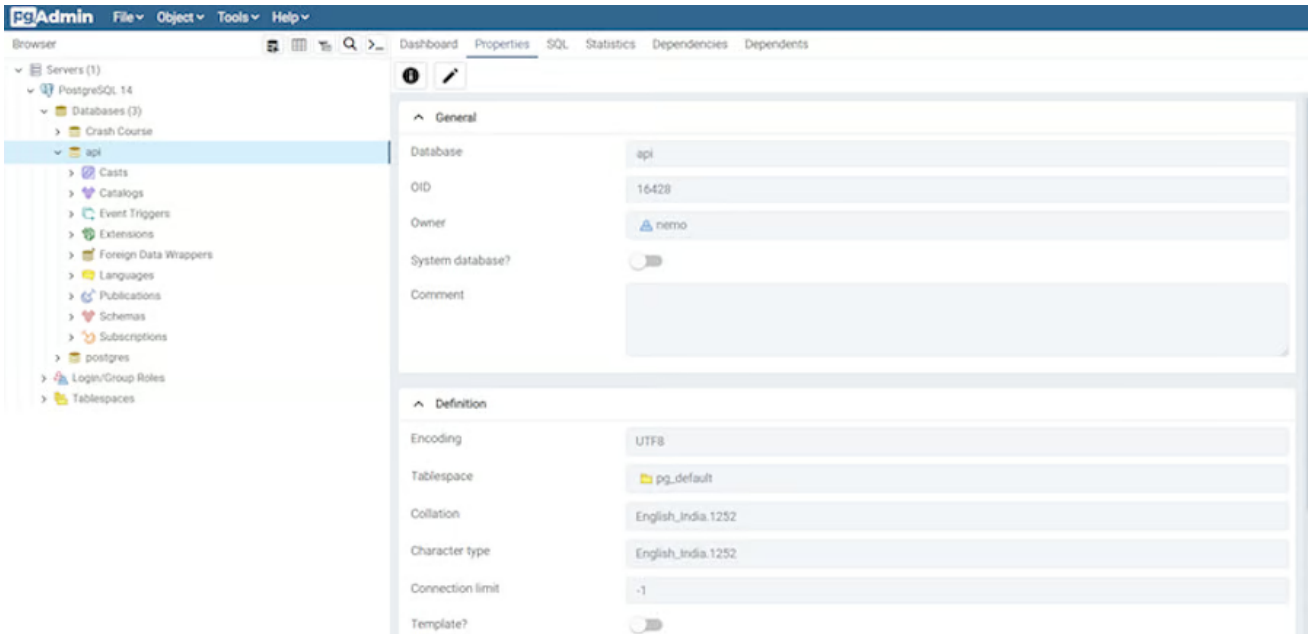
```
{
  info: "Node.js, Express, and Postgres API"
}
```

The Express server is running now, but it's only sending some static JSON data that we created. The next step is to connect to PostgreSQL from Node.js to be able to make dynamic queries.

Connecting to a Postgres database using a Client

A popular client for accessing Postgres databases is the pgAdmin client. The pgAdmin application is available for various platforms. If you want to have a graphical user interface for your Postgres databases, you can go to the download page and download the necessary package.

Creating and querying your database using pgAdmin is simple. You need to click on the **Object** option available on the top menu, select **Create**, and choose **Database** to create a new connection. All the databases are available on the side menu. You can query or run SQL queries efficiently by selecting the proper database:



Connecting to a Postgres database from Node.js

We'll use the node-postgres module to create a pool of connections. Therefore, we don't have to open and close a client each time we make a query.

A popular option for production pooling would be to use `[pgBouncer](https://pgbouncer.github.io/)`, a lightweight connection pooler for PostgreSQL.

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'me',
  host: 'localhost',
  database: 'api',
  password: 'password',
  port: 5432,
})
```

In a production environment, you would want to put your configuration details in a separate file with restrictive permissions so that it is not accessible from version control. But, for the simplicity of this tutorial, we'll keep it in the same file as the queries.

The aim of this tutorial is to allow CRUD operations — `GET`, `POST`, `PUT`, and `DELETE` — on the API, which will run the corresponding database commands. To do so, we'll set up a route for each endpoint and a function corresponding to each query.

Creating routes for CRUD operations

We'll create six functions for six routes, as shown below. First, create all the functions for each route. Then, export the functions so they're accessible:

- `GET : / | displayHome()`
- `GET : /users | getUsers()`
- `GET : /users/:id | getUserById()`
- `POST : /users | createUser()`
- `PUT : /users/:id | updateUser()`
- `DELETE : /users/:id | deleteUser()`

In `index.js`, we made an `app.get()` for the root endpoint with a function in it. Now, in `queries.js`, we'll create endpoints that will display all users, display a single user, create a new user, update an existing user, and delete a user.

`GET` all users

Our first endpoint will be a `GET` request. We can put the raw SQL that will touch the `api` database inside the `pool.query()`. We'll `SELECT` all users and order by ID.

```
const getUsers = (request, response) => {
  pool.query('SELECT * FROM users ORDER BY id ASC', (error, results
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}
```

`GET` a single user by ID

For our `/users/:id` request, we'll get the custom `id` parameter by the URL and

use a `WHERE` clause to display the result.

In the SQL query, we're looking for `id=$1`. In this instance, `$1` is a numbered placeholder that PostgreSQL uses natively instead of the `?` placeholder that you may recognize from other variations of SQL:

```
const getUserById = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('SELECT * FROM users WHERE id = $1', [id], (error, res
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}
```

`POST` a new user

The API will take a `GET` and `POST` request to the `/users` endpoint. In the `POST` request, we'll add a new user. In this function, we're extracting the `name` and `email` properties from the request body and inserting the values with `INSERT`:

```
const createUser = (request, response) => {
  const { name, email } = request.body

  pool.query('INSERT INTO users (name, email) VALUES ($1, $2) RETUR
    if (error) {
      throw error
    }
    response.status(201).send(`User added with ID: ${results.rows[0
  })
}
```

`PUT` updated data in an existing user

The `/users/:id` endpoint will also take two HTTP requests, the `GET` we created for `getUserById` and a `PUT` to modify an existing user. For this query, we'll combine what we learned in `GET` and `POST` to use the `UPDATE` clause.

It's worth noting that `PUT` is idempotent, meaning the exact same call can be made over and over and will produce the same result. `PUT` is different than `POST`, in which the exact same call repeated will continuously make new users with the same data:

```
const updateUser = (request, response) => {
  const id = parseInt(request.params.id)
  const { name, email } = request.body

  pool.query(
    'UPDATE users SET name = $1, email = $2 WHERE id = $3',
    [name, email, id],
    (error, results) => {
      if (error) {
        throw error
      }
      response.status(200).send(`User modified with ID: ${id}`)
    }
  )
}
```

`DELETE` a user

Finally, we'll use the `DELETE` clause on `/users/:id` to delete a specific user by ID. This call is very similar to our `getUserById()` function:

```
const deleteUser = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('DELETE FROM users WHERE id = $1', [id], (error, resul
    if (error) {
      throw error
    }
    response.status(200).send(`User deleted with ID: ${id}`)
  })
}
```

Exporting CRUD functions in a REST API

To access these functions from `index.js`, we'll need to export them. We can do so with `module.exports`, creating an object of functions. Since we're using the ES6 syntax, we can write `getUsers` instead of `getUsers:getUsers` and so on:

```
module.exports = {
  getUsers,
  getUserById,
  createUser,
  updateUser,
  deleteUser,
}
```

Our complete `queries.js` file is below:

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'me',
  host: 'localhost',
  database: 'api',
  password: 'password',
```

```
  port: 5432,
})
const getUsers = (request, response) => {
  pool.query('SELECT * FROM users ORDER BY id ASC', (error, results
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const getUserById = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('SELECT * FROM users WHERE id = $1', [id], (error, res
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const createUser = (request, response) => {
  const { name, email } = request.body

  pool.query('INSERT INTO users (name, email) VALUES ($1, $2)', [na
    if (error) {
      throw error
    }
    response.status(201).send(`User added with ID: ${results.insert
  })
}

const updateUser = (request, response) => {
  const id = parseInt(request.params.id)
  const { name, email } = request.body
```

```
  pool.query(
    'UPDATE users SET name = $1, email = $2 WHERE id = $3',
    [name, email, id],
    (error, results) => {
      if (error) {
        throw error
      }
      response.status(200).send(`User modified with ID: ${id}`)
    }
  )
}

const deleteUser = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('DELETE FROM users WHERE id = $1', [id], (error, resul
    if (error) {
      throw error
    }
    response.status(200).send(`User deleted with ID: ${id}`)
  })
}

module.exports = {
  getUsers,
  getUserById,
  createUser,
  updateUser,
  deleteUser,
}
```

Setting up CRUD functions in a REST API

Now that we have all of our queries, we need to pull them into the `index.js` file and make endpoint routes for all the query functions we created.

To get all the exported functions from `queries.js`, we'll `require` the file and assign it to a variable:

```
const db = require('./queries')
```

Now, for each endpoint, we'll set the HTTP request method, the endpoint URL path, and the relevant function:

```
app.get('/users', db.getUsers)
app.get('/users/:id', db.getUserById)
app.post('/users', db.createUser)
app.put('/users/:id', db.updateUser)
app.delete('/users/:id', db.deleteUser)
```

Below is our complete `index.js` file, the entry point of the API server:

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const db = require('./queries')
const port = 3000

app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
)

app.get('/', (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
})

app.get('/users', db.getUsers)
```

```
app.get('/users/:id', db.getUserById)
app.post('/users', db.createUser)
app.put('/users/:id', db.updateUser)
app.delete('/users/:id', db.deleteUser)

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

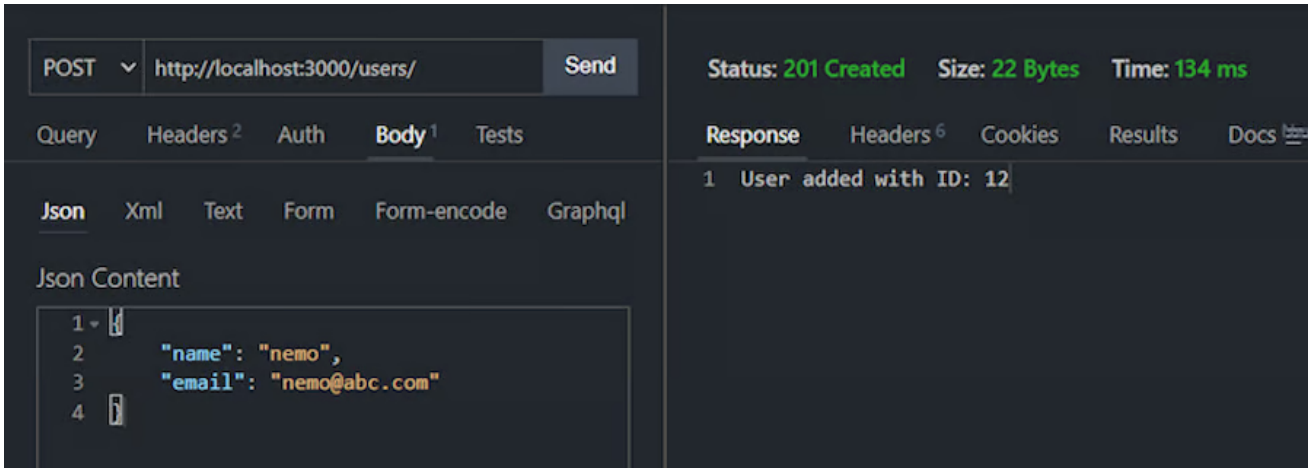
With just these two files, we have a server, database, and our API all set up. You can start up the server by hitting `index.js` again:

```
node index.js
App running on port 3000.
```

Now, if you go to `http://localhost:3000/users` or `http://localhost:3000/users/1`, you'll see the JSON response of the two `GET` requests.

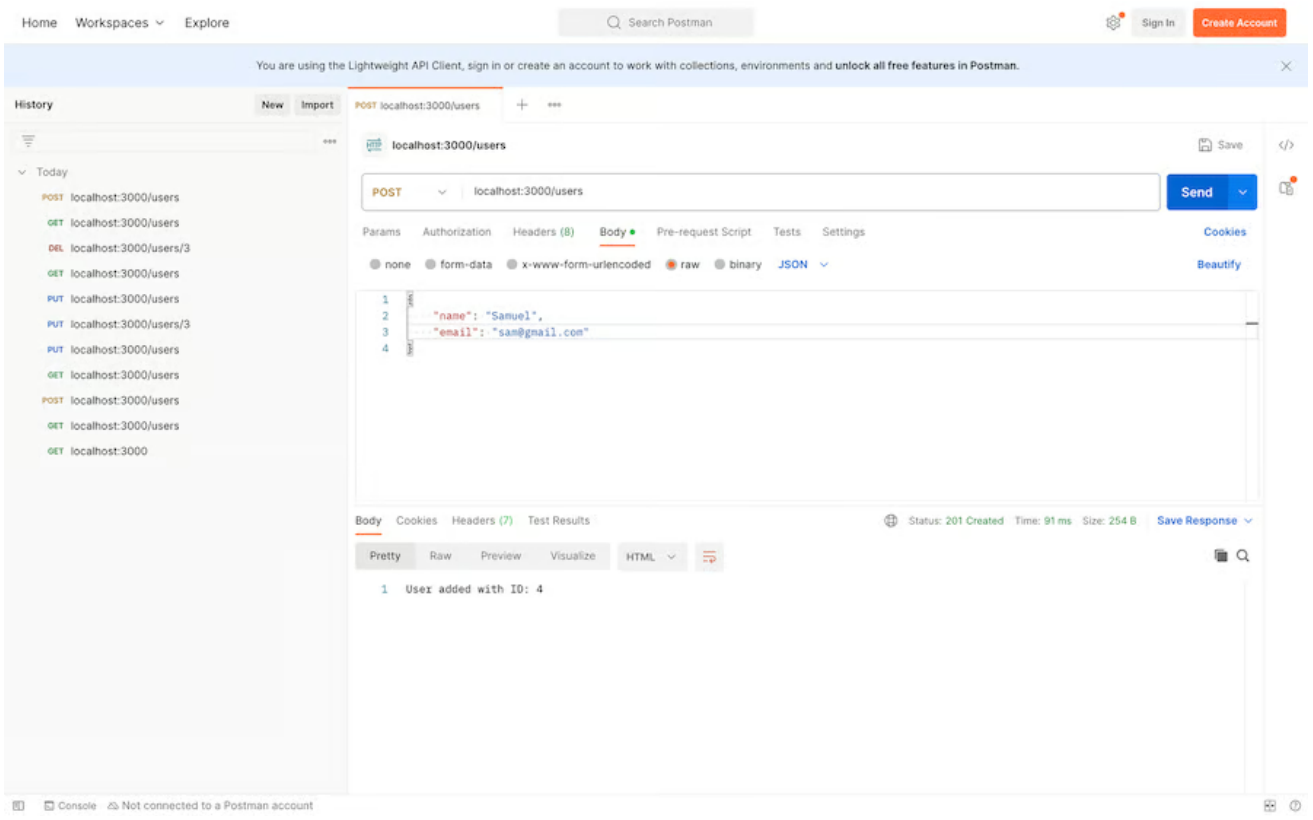
To test our `POST`, `PUT`, and `DELETE` requests, we can use a tool like Postman or a VS Code extension like Thunder Client to send the HTTP requests. You can also use curl, a command-line tool that is already available on your terminal.

Using a tool like Postman or Thunder Client makes it simple to query endpoints with different HTTP methods. Simply enter your URL, choose the specific HTTP method, insert the JSON value if the endpoint is a PUT or POST route, and hit **Send**:

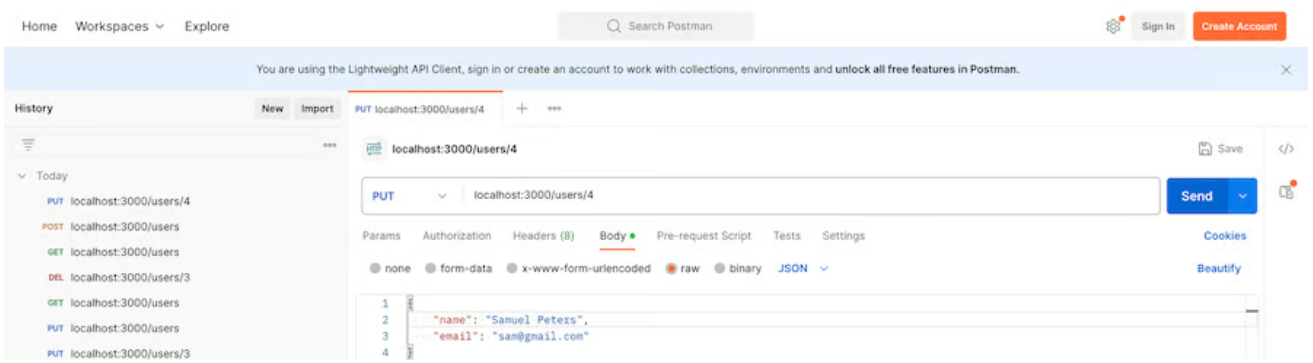


The example above shows sending a `POST` request to the specified route. The `POST` option suggests that it is a `POST` request. The URL beside the method is the API endpoint, and the JSON content is the data to be sent to the endpoint. You can hit the different routes similarly.

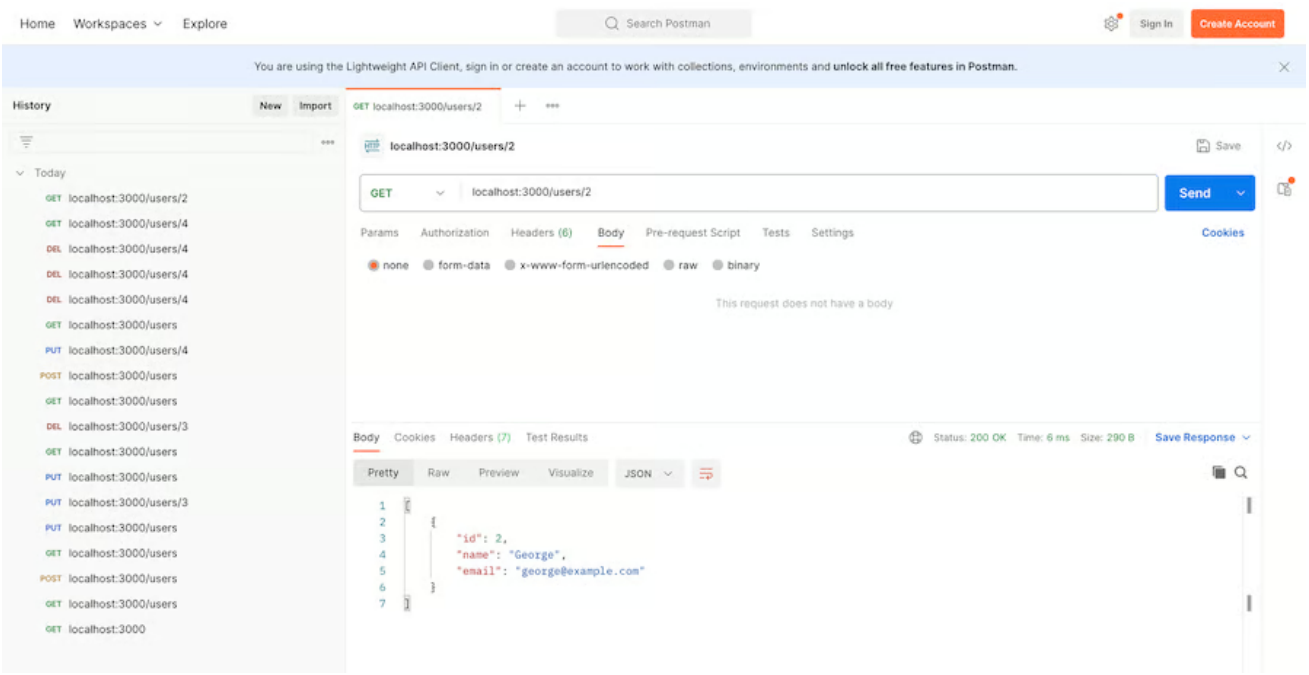
Here's an example of sending a `POST` request to the specified route to create a new user using Postman:



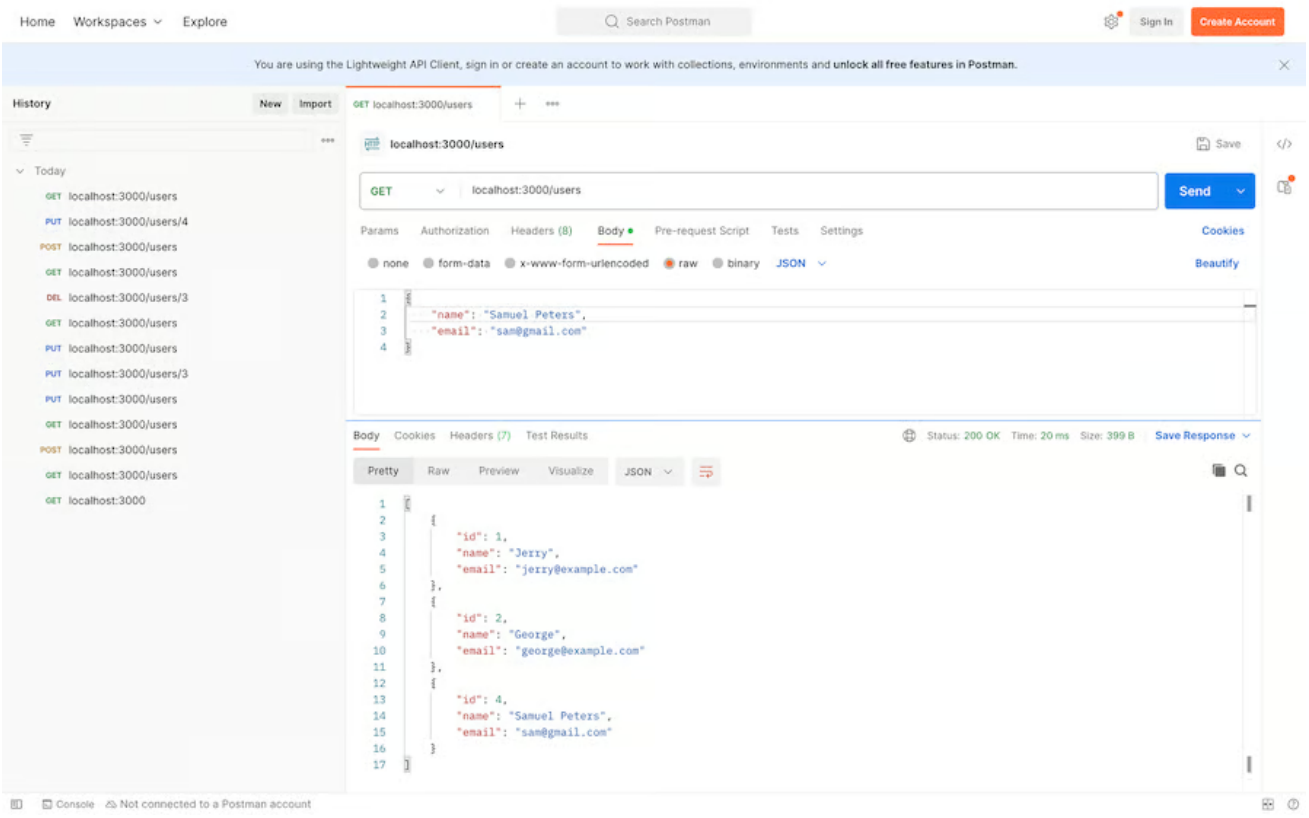
Here's an example of sending a `PUT` request to the specified route to modify a user by its ID:



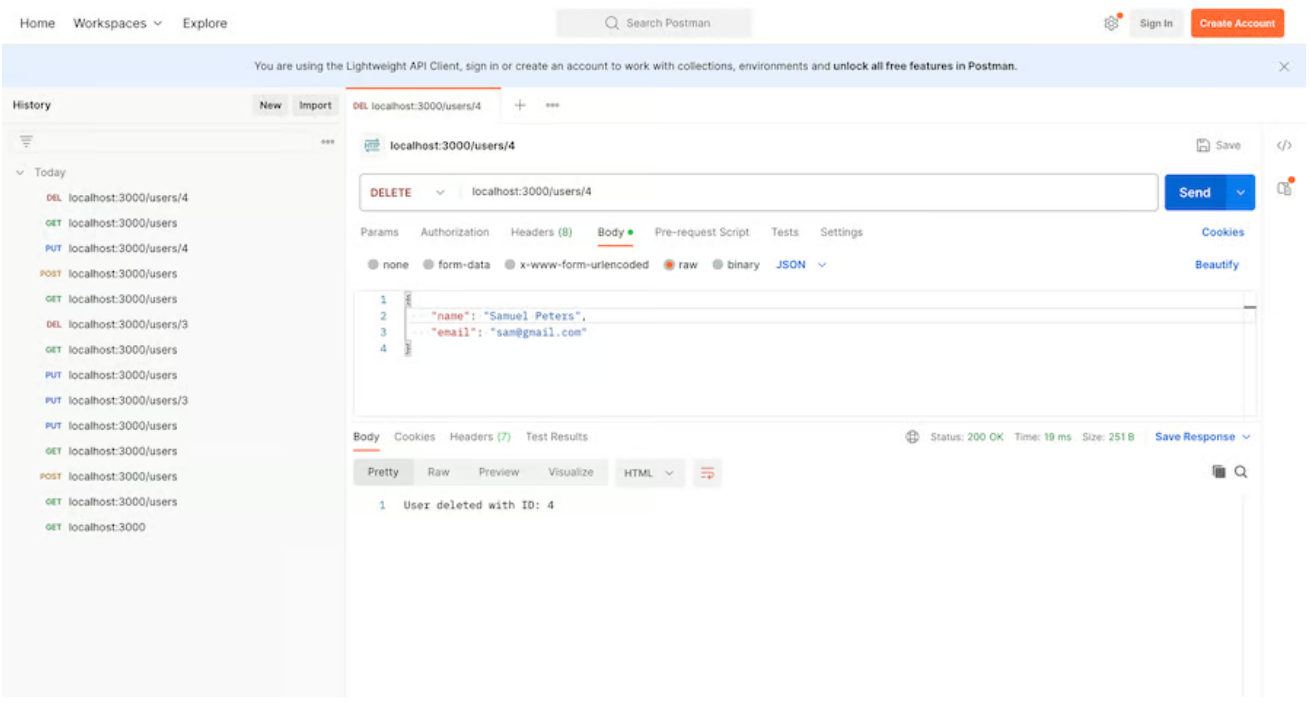
Here's an example of sending a `GET` request to the specified route to retrieve a user by its ID:



Here's an example of sending a `GET` request to the specified route to retrieve all users:



Finally, here's an example of sending a `DELETE` request to the specified route to delete a user by its ID:



Solutions to common issues encountered while developing APIs

Developing APIs can come with various challenges. Let's go over the solutions to two common issues encountered during API development: CORS issues and unhandled errors due to middleware order.

Handling CORS issues

Browser security policies can block requests from different origins. To address this issue, use the `cors` middleware in Express to handle cross-origin resource sharing (CORS).

Run the following command to install `cors`:

```
npm install cors
```

To use it, do the following:

```
var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())
```

This will enable CORS for all origins.

Middleware order and error handling

Middleware order can affect error handling, leading to unhandled errors. To address this issue, place error-handling middleware at the end of your middleware stack and use `next(err)` to pass errors to the error-handling middleware:

```
app.use((req, res, next) => {
    const error = new Error('Something went wrong');
    next(error);
});
// Error-handling Middleware
app.use((err, req, res, next) => {
    console.error('Error:', err.message);
    res.status(500).send('Internal Server Error');
});
```

Securing the API

When it comes to securing APIs, we need to implement various mechanisms to ensure the confidentiality, and integrity of the application and its data. Let's go over a few of these mechanisms now.

Authentication

You can implement strong authentication mechanisms, such as JSON Web Tokens (JWT) or OAuth, to verify the identity of clients. Ensure that only authenticated and authorized users can access certain routes — in our case, the `POST`, `PUT`, and `DELETE` methods.

I will recommend the Passport middleware for Node.js, which makes it easy to implement authentication and authorization. Here's an example of how to use Passport:

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

```
const LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
    function(username, password, done) {
        // Verify username and password
        // Call done(null, user) if authentication is successful
    }
));
```

Authorization

It's important to enforce proper access controls to restrict access to specific routes or resources based on the user's role or permissions. For example, you can check if the user making a request has `admin` privileges before allowing or denying them permission to proceed with the request:

```
function isAdmin(req, res, next) {
    if (req.user && req.user.role === 'admin') {
        return next();
    } else {
        return res.status(403).json({ message: 'Permission denied'
    }
}
```

You can apply the `isAdmin` middleware defined above to any protected routes, thus restricting access to those routes.

Input validation

Validate and sanitize user inputs to prevent SQL injection, XSS, and other security vulnerabilities. For example:

```
const { body, validationResult } = require('express-validator');
```

```
app.post('/users', [
    // add validation rules
], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(422).json({ errors: errors.array() });
    }
    // Process the request
});
```

The code above allows you to specify validation rules for POST requests to the `/users` endpoint. If the validation fails, it sends a response with the validation errors. If the incoming data is correct and safe, it proceeds with processing the request.

Helmet middleware

You can use the Helmet middleware to set various HTTP headers for enhanced security:

```
const helmet = require('helmet');
app.use(helmet());
```

Configuring HTTP headers with Helmet helps protect your app from security issues like XSS attacks, CSP vulnerabilities, and more.

Additional notes and suggestions

You can build on this tutorial by implementing the following suggestions:

- **Integration with frontend frameworks**: Choose a frontend framework or library (e.g., React, Angular, Vue.js) to build a user interface for your application. Then, implement API calls from the frontend to interact with the

backend CRUD operations. You can consider state management solutions (e.g., Redux, Vuex) for managing the state of your frontend application

- **Containerizing the API**: Write a Dockerfile to define the environment and dependencies needed to run your Node.js app. Create a `docker-compose.yml` file for managing multiple containers, such as the Node.js app and PostgreSQL database. This will make your Node.js application easier to deploy and set up on other machines
- **Implementing unit/integration tests:** Write unit tests for individual functions and components of your application to ensure that they work as expected. Use testing frameworks like Mocha, Jest, or Jasmine for writing and running tests. Implement integration tests to verify the interactions between different components in your front-end application and the overall functionality of your API
- **Continuous integration/deployment (CI/CD)**: Set up CI/CD pipelines to automate the testing and deployment processes. Use tools like Jenkins, Travis CI, or GitHub Actions to streamline the development/deployment workflow

While actually implementing these next steps is beyond the scope of this tutorial, you can use these ideas to apply what we've discussed to a real use case.

Conclusion

You should now have a functioning API server that runs on Node.js and is hooked up to an active PostgreSQL database.

In this tutorial, we learned how to install and set up PostgreSQL in the command line, create users, databases, and tables, and run SQL commands. We also learned how to create an Express server that can handle multiple HTTP methods and use the `pg` module to connect to PostgreSQL from Node.js.

With this knowledge, you should be able to build on this API and utilize it for your own personal or professional development projects.

#node #postgresql

How to create fancy corners using CSS

`corner-shape`

Learn about CSS's `corner-shape` property and how to use it, as well as the more advanced side of `border-radius` and why it's crucial to using `corner-shape` effectively.

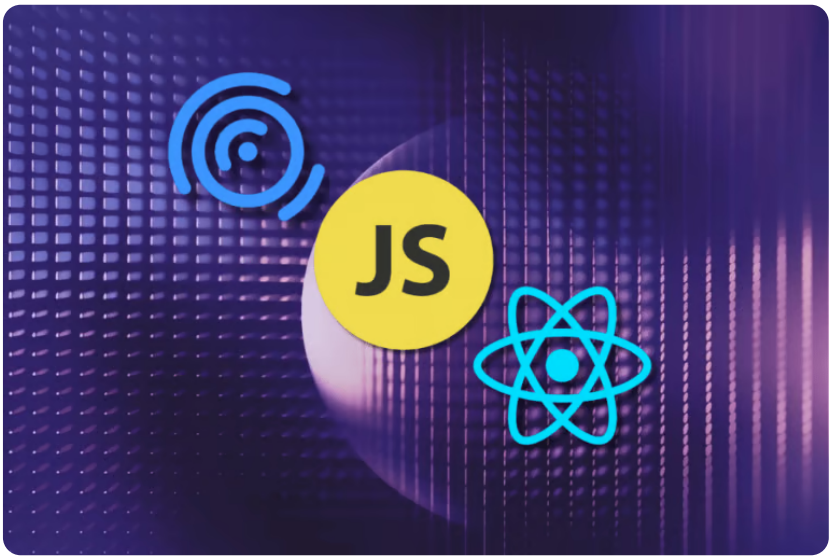
Daniel Schwarz

Nov 26, 2025 · 7 min read

The Replay (11/26/25): An AI reality check, Prisma v7, and more

An AI reality check, Prisma v7, and “caveman compression”: discover what’s new in The Replay, LogRocket’s newsletter for dev and engineering leaders, in the November 26th issue.

Matt MacCormack
Nov 26, 2025 · 35 sec read

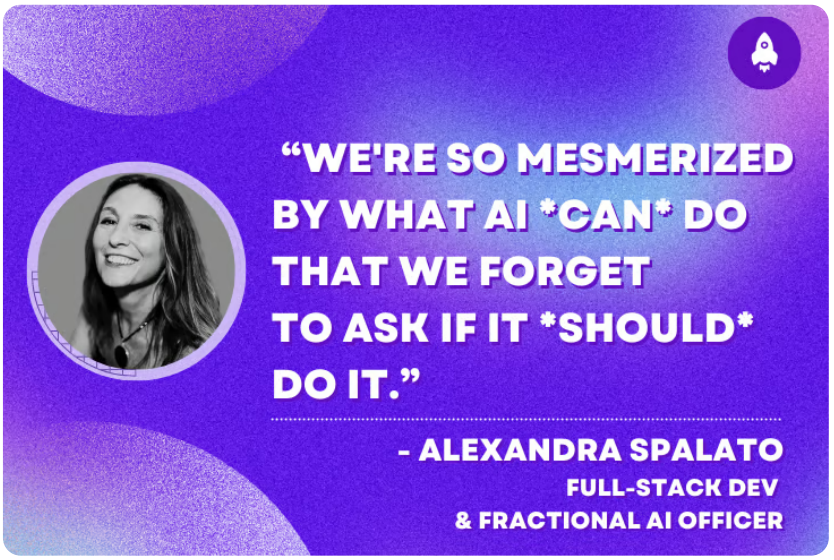


Ripple over React? Evaluating the newest JS framework

RippleJS takes a fresh approach to UI development with no re-renders and TypeScript built in. Here's why it's gaining attention.

Chizaram Ken

Nov 26, 2025 · 15 min read



You don’t need AI for everything: A reality check for developers

As a developer, it’s easy to feel like you need to integrate AI into every feature and deploy agents for every task. But what if the smartest move isn’t to use AI, but to know when not to?

Alexandra Spalato
Nov 26, 2025 · 6 min read

View all posts

82 Replies to "CRUD REST API with Node.js, Express, and PostgreSQL"

phaiza says: Reply

August 19, 2020 at 10:08 pm

thank you!

Thankful Reader says: Reply

September 24, 2020 at 12:28 pm

This is a wonderful article! Thank you for this!

Roger says: Reply

October 2, 2020 at 6:49 am

this is finally what i've been looking for. great start that can be easily expanded by the readers needs.

jodoinscott says: Reply

October 6, 2020 at 4:19 pm

I needed to use

psql -d postgres -h localhost -U me

to log in, otherwise I got

psql: error: could not connect to server: FATAL: Peer authentication failed for user "me"

jodoinscott says: Reply

October 6, 2020 at 4:41 pm

This is great! Exactly what I was looking for. Note there is an error in the POST code snippet. 'results' should be 'result'.

jodoinscott says: Reply

October 6, 2020 at 4:50 pm

Good point. Still true.

jodoinscott says: Reply

October 6, 2020 at 4:51 pm

Great response. Thanks for the help!

yared solomon says: Reply

November 5, 2020 at 11:02 pm

Thank you so much. this is all what i want

wetbadger says: Reply

January 29, 2021 at 11:51 am

I'm confused. Everything works except why would there be a json response on localhost:3000/users ? I don't see anything there. Am I supposed have a users.html?

Luis Montalvo says: Reply

February 3, 2021 at 10:39 pm

Hi Tania,

Thanks very much for your very clear and concise example. I followed it closely.

There was only 1 problem I ran into, which is when creating a user, I could not get back the inserted ID.

I looked this up, and for PosrgreSQL, there is a different way in which the id is returned after an insert, as follows:

const createUser = (request, response) => {

const { username, email } = request.body;

pool.query(

"INSERT INTO users (username, email) VALUES ($1,$2) RETURNING id",

[username, email],

(error, results) => {

if (error) {

```
throw error;
}
response.status(201).send(`User added with ID:${results.rows[0].id} `);
}
);
};
```

The key is in getting the SQL statement correctly, with the “RETURNING id” bit at the end.

Hopefully this will help others if they find the same problem.

Best,

Luis

sentinel1909 says: Reply

July 18, 2021 at 11:27 pm

OMG! Thank you so much, was on the verge of tearing my hair out here… 🙂

dchangusc says: Reply

February 8, 2021 at 4:54 pm

yep! it took me awhile to figure it out. thanks

Jord says: Reply

February 20, 2021 at 10:59 am

Nice at first I thought it was the apostrophe formatting, I actually had the RETURNING id part but couldnt get the results.rows piece – thank you!

Matt says: Reply

March 5, 2021 at 5:42 pm

Sweet tutorial! thanks for the work, it is very easy to follow and works like a charm 🙂

Alex says: Reply

March 10, 2021 at 4:58 pm

I seem to be getting this error when I run node index.js at the end: error: password authentication failed for user “testuser”

Is there a fix for this?

Adnan Lubis says: Reply

March 29, 2021 at 5:58 am

Thanks

Abhishekh Amrut Kamble says: Reply

March 31, 2021 at 10:29 pm

thank you so much

John says: Reply

June 12, 2021 at 11:25 pm

I’ve copied the 2 files completely and when I try and localhost:3000/users it just spins forever. Any way to trace the error?

Simmons says: Reply

December 19, 2021 at 4:20 pm

I’ve failed to INSERT entries via $request.body, I had to use $request.query and I’m stunned nobody else had this issue

The results.rows[0].id thing isn’t working for me either 🙄

Pepper says: Reply

June 16, 2022 at 12:11 am

Thank you! I had the issue with $request.body as well and changing it to $request.query fixed it.

Not sure about results.rows[0] thing though, that one works on my machine.

Vitaliy says: Reply

December 30, 2021 at 8:55 am

Hi! I find the text between the headlines “Installation” and “PostgreSQL command prompt” to be misleading. I’m using Windows and it is unlear to me which of those passages are releveant to my OS and whether I have to run any brew-based or commands or similar ones in npm after downloading the Windows installer (and running it, obviously, though not mentioned) and passin on to the PostgreSQL command prompt section. Clarifications would be appreciated.

Simo says: Reply

March 17, 2022 at 6:14 am

Very nice, Thank you so much 🙂

Ian says: Reply

June 10, 2022 at 8:03 pm

How do I “hit index.js”? How do I run this?

Parmeshwar Rathod says: Reply

June 29, 2022 at 5:40 am

thanks alot man

such great content

Robert Werner says: Reply

June 30, 2022 at 12:06 pm

Tania, I am a very accomplished React & C# & Python software engineer but am working on my own project now. I was all set to have a Python back-end, like is at my work but

then learned about Express.js. Thank you SO MUCH for your excellent, most comprehensive article. It served as the template for building the back-end of my ever evolving application.
By the way, I've built both front-end and back-end with TypeScript so upgraded your Javascript code to TypeScript. Feel free to contact me and I will gladly share my code with you in case you want to update this article or follow-up with another.

uttam says: Reply

August 27, 2022 at 9:47 am

Can you please share your TypeScripit prject.

Darko says: Reply

July 8, 2022 at 4:26 am

For those who have a problem with createUser, change code into:
response.status(201).send(`User added with ID: ${results.rows[0]["id"]}`)

Uttam says: Reply

August 27, 2022 at 9:35 am

I can able to create record in users table but both name and email has null values. request.body tells undefined. Any help appreciated.

alexsvt2 says: Reply

January 19, 2023 at 2:23 pm

Maybe is late but this could work for others.

make use of bodyParser should solve this problem

app.use(bodyParser.json())

app.use(

bodyParser.urlencoded({

extended: true,

})

)

Wesley says: Reply

March 23, 2023 at 4:18 pm

On MacOS – Homebrew required a version for postgresql.

“

brew install postgresql

==> Downloading https://formulae.brew.sh/api/formula.jws.json

100.0%

==> Downloading https://formulae.brew.sh/api/cask.jws.json

100.0%

Warning: No available formula with the name “postgresql”. Did you mean postgresql@13, postgresql@12, postgresql@11, postgresql@15, postgresql@10, postgresql@14, postgresql@9.5, postgresql@9.4, postgrest or qt-postgresql?

postgresql breaks existing databases on upgrade without human intervention.

See a more specific version to install with:

brew formulae | grep postgresql@

“

This worked for me:

“

brew install postgresql@15

“

Comments pagination

« Previous 1 2 Next »

Leave a Reply