



Richard Haines

Posted on 9 févr. 2021



How to seed a postgres database with node

#postgres #database #node #seed

This guide was originally posted on my blog richardhaines.dev

This guide will walk through seeding a Postgres database. It will cover creating a Node script to seed the data and touch on some of the pros and cons as to the chosen approach.

The source code for this guide can be found here: github.com/molebox/seed-postgres-database-tutorial

Prerequisites

- You must have Postgres [installed on your machine](#)
- You must have Node [installed on your machine](#)
- It is assumed that you have Postgres setup and know-how to access its databases, though the guide will cover some basic commands towards the end.
- This guide uses the default user `postgres` for accessing the database. If you have another user account you would prefer to use then swap in that.

The guide uses version 13.1 of Postgres and version 14.15.1 of Node. It was developed on a **Windows** machine.

What do we mean by seed?

The process of seeding (in the context of databases) is to insert or populate the initial data into the database. This can be either a manual or automated step in the setup of an application. Seeding can be used when testing different branches, for example, if you have a dev branch where you want to test some new sorting query against the database, seeding would be a good way to test against data that won't affect a production build. Of course, there are many reason one might choose to seed a database. In some instances, an applications database requires some form of data present before it will work properly, such as an admin account. But more often than

not seeding would take place pre-install and thus allow the user to begin using the app without any issues.

The seed script

The seed script will aim to accomplish the following:

- Create a database with a table.
- Create a csv file and populate it with fake data using the [faker](#) library. It will default to 10 rows but allow the user to specify an amount if they like.
- Parse that data and insert it into the table - seed the database.

Begin by creating a `schema.sql` file at the root of your project. This file will enable you to lay the groundwork for how your database and its table will look.

schema.sql

```
-- Seeing as we will be testing out this script alot we can destroy the db be
DROP DATABASE IF EXISTS translationsdb;

-- Create the db
CREATE DATABASE translationsdb;

-- Move into the db
\c translationsdb

-- Create our table if it doesn't already exist
CREATE TABLE IF NOT EXISTS Translations
(
    key character varying(100),
    lang character varying(5),
    content text
);

-- Changes the owner of the table to postgres which is the default when insta
ALTER TABLE Translations
    OWNER to postgres;
```

db.js

To interact with the Postgres database, you can install the [node-postgres](#) package, a collection of modules made for interacting with Postgres. You'll use it to establish an initial connection to the database and insert some fake data. Create a new file `src/db.js` and add the following:

```
const { Pool } = require('pg');
const { host, user, database, password, port } = require('./config');

// Create a pool instance and pass in our config, which we set in our env var
const pool = new Pool({
  host,
  user,
  database,
  password,
  port,
});

module.exports = {
  query: (text, params, callback) => {
    return pool.query(text, params, callback);
  },
  connect: (err, client, done) => {
    return pool.connect(err, client, done);
  },
};
```

The Pool class takes some optional config and the values passed in enable a connection with the database. They are set as environment variables (env vars) and imported from a separate config file. This file exports two functions. The query, which will be used to run an `INSERT` statement, and a connect function which will be used to connect to the database.

config.js

Storing all the env vars in one place and exporting them means you have one source of truth and can easily swap them out from one place instead of multiple files. Create a new file and name it `config.js`.

```
const dotenv = require('dotenv');
dotenv.config();
// Single source to handle all the env vars
module.exports = {
  host: process.env.PGHOST,
  user: process.env.PGUSER,
  database: process.env.PGDATABASE,
  password: process.env.PGPASSWORD,
  port: process.env.PGPORT,
};
```

An example of how your env vars might look:

```
PGUSER=postgres
PGHOST=localhost
PGPASSWORD=test1234
PGDATABASE=translationsdb
PGPORT=5432
```

main.js

In a real-world scenario, you would perhaps have some data stored in a csv file. This example will make use of the [faker library](#), and some other packages. Install the following:

```
yarn add dotenv faker fast-csv minimist pg validator
```

Use the Faker library by creating a function that will mimic the shape of the table set in `schema.sql`. It will return a template literal string to be added to a csv file later on.

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

// Create some fake data using the faker lib. Returns a template string to be
function createTranslation() {
  const key = faker.address.country();
  const lang = faker.address.countryCode();
  const content = faker.random.word();

  return `${key},${lang},${content}\n`;
}
```

Next, you will need to import fs and create a stream. This will write to an as yet non-existent csv file.

```
// other imports..
const fs = require('fs');

// The path to write the csv file to
const output = './src/output.csv';

// other functions..

// Create a stream to write to the csv file
const stream = fs.createWriteStream(output);
```


Enabling the user to choose how many rows they would like to seed the database with is an extra and worthwhile step. The `minimist` package helps with parsing

argument options. In the case of the script, it allows the user the option to pass in an amount, if the user chooses not to pass any additional arguments then you can set a default value. Create a new function that will write the fake data to the csv file.

```
// other imports..
const args = require('minimist')(process.argv.slice(2));

// other functions...

async function writeToCsvFile() {
  // The user can specify how many rows they want to create (yarn seed --ro
  let rows = args['rows'] || 10;
  // Iterate x number of times and write a new line to the csv file using t
  for (let index = 0; index < rows; index++) {
    stream.write(createTranslation(), 'utf-8');
  }
  stream.end();
}
```



Now that the csv file is created and populated with fake data, you can begin the process of actually seeding that data into the Postgres database. `fast-csv` is a library for parsing and formatting csv files. You'll use it in combination with the `validator` library and `node-postgres`.

```
// other imports...
const fastcsv = require('fast-csv');
const db = require('./db');
const contains = require('validator/lib/contains');

// other functions...

function insertFromCsv() {
  let csvData = [];
  return (
    fastcsv
      .parse()
      // validate that the column key doesn't contain any commas, as so
      .validate((data) => !contains(data[0], ','))
      // triggered when a new record is parsed, we then add it to the d
      .on('data', (data) => {
        csvData.push(data);
      })
      .on('data-invalid', (row, rowNumber) =>
        console.log(
          `Invalid [rowNumber=${rowNumber}] [row=${JSON.stringify(r`
```

```

    )
  )
  // once parsing is finished and all the data is added to the array
  .on('end', () => {
    // The insert statement
    const query =
      'INSERT INTO translations (key, lang, content) VALUES ($1'
    // Connect to the db instance
    db.connect((err, client, done) => {
      if (err) throw err;
      try {
        // loop over the lines stored in the csv file
        csvData.forEach((row) => {
          // For each line we run the insert query with the
          client.query(query, row, (err, res) => {
            if (err) {
              // We can just console.log any errors
              console.log(err.stack);
            } else {
              console.log('inserted ' + res.rowCount +
            }
          });
        });
      } finally {
        done();
      }
    });
  });
});

```

The function first validates the contents of the row using the `contains` function from the `validator` library. This is necessary because some countries can have an extra comma in their name. An extra comma in a csv file equates to an extra column and the table created and defined in the `schema.sql` file dictates that only 3 columns will exist. If this check fails `fast-csv` will not accept the row and throw an event, which is used to print a message to the console to inform the user.

If the row is accepted then it is added to an array. Once the parsing is finished and all the row data is added to the array the connection is established with the Postgres database. The data array is then iterated over, for each row in the array a client instance is acquired from the pool and an `INSERT` query is used as an argument, along with the row data. If the row is successfully inserted into the table then its corresponding data is printed to the console, if any errors occur they are also printed

to the console. Finally, the done function is called to release the clients back to the pool.

The final function called seed is where the data is written to the csv file, a stream is created to read the data from the output file and then the `INSERT` function is piped to the stream.

```
// all the other code from main.js

async function seed() {
  await writeToCsvFile();
  let stream = fs.createReadStream(output);
  stream.pipe(insertFromCsv());
}

seed();
```

Add two scripts to the `package.json` file. The first `create-db` will ask the user to login and connect to their Postgres database and then run the commands in the `schema.sql` file. The second script will run the first before running the seed function.

The user may run the script with additional arguments to set the number of rows created in the table.

- Extra rows: `yarn seed --rows=200`
- Default 10 rows: `yarn seed`

```
"scripts": {
  "create-db": "psql -U postgres < schema.sql",
  "seed": "yarn create-db && Node src/main.js"
},
```

Check the database

To check the database table you can run the following commands from your terminal:

```
// login and connect to the database
psql -U postgres -d translationsdb

// get all the rows in the table
select * from "translations";
```

Final thoughts

There are many ways this could have been accomplished, in fact, there are many libraries that support using Node with Postgres. This method was chosen for its relative simplicity. It's not a generic solution that would fit all scenarios but it could be built upon to incorporate extra features.

Positive take-aways

- The fact that no external API was used for the data removed the overhead of having to make any requests.
- Using Nodes inbuilt `process.argv` combined with a small parser library meant the user could add some level of configuration when seeding.

Possible improvements

- The function that creates the fake data could be added via config to match the schema definition of the table.
- It would also work quite nicely with a CLI, this would enable the end user much more configuration.

Top comments (2)



Andrew Njoo • 2 mars 22 • Edited



Thank you for this tutorial! I was looking for a seed tutorial for node / postgres.

I would recommend installing [faker@5.5.3](#) since [faker@6.6.6](#) doesn't work (due to author deleting his repo)



Romaric P. • 14 mai 22



You can use [Replibyte](#) (open-source) to seed your Postgres database with your production data while keeping safe your sensitive data.

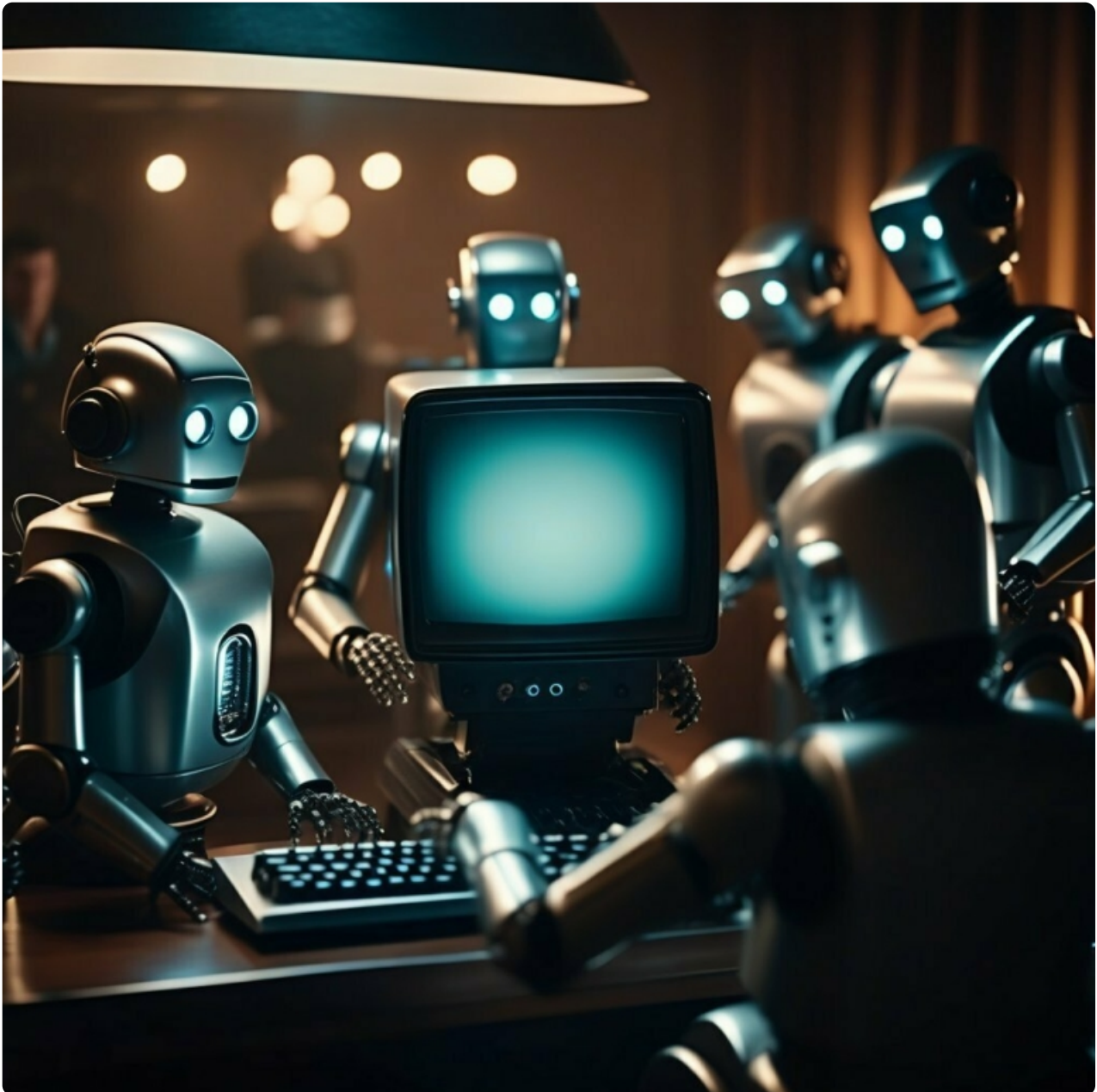
[Code of Conduct](#) • [Report abuse](#)



AWS Community Builders

PROMOTED





[Amazon Bedrock For JavaScript and TypeScript Developers](#)

The blog post provides codes and layout instructions for developers working with Amazon Bedrock SDK V3 in a variety of languages including JavaScript and TypeScript. It directs them on implementing models with detailed code breakdown.

[Read full post](#)



Richard Haines

Writing docs @prisma I love to create content and fun experiences while exploring new technologies

LOCATION

Örnsköldsvik

WORK

Tech Writer at Prisma

JOINED

3 déc. 2018

Trending on DEV Community 🔥



Why Angular Will not Survive 2024

#javascript #angular #node #frontend



Meme Monday (Holiday themed?)

#discuss #watercooler #jokes



What you learning about this weekend? 🧠

#codenewbie #discuss #learning #beginners



DEV Ads Partner PROMOTED

