

Combining Servers and Databases!

The plan for the day:

Set up express server (using Express generator)

Set up a database (using Heroku)

Hook up express app and database (so we can use NODE and JS to start sending our SQL queries)

Set up tables in our DB and get the data in them (populate them)

Hook up our server's logic to talk to the DB instead of a file

Express application generator

Sets up a basic server for us. Uses CommonJS!

`npx express-generator`

Or use Chris Meah's way that converts to ESM for us!

`Npx express-generator-esmodules .`

It definitely is npx, not npm!

Npm allows you install packages in the node-modules folder so you can use them permanently on the plumber. Npx allows it to get set up quickly and then leave straight away without hanging around!

Still need to npm i first!

All code in app.js we used before plus a load of middleware such as a logger. This sets up the listener in bin folder – [www.js](#) – either using the provided port from env.variable if there is one, or 3000 by default.

Heroku

Create new app (not pipeline)

Add-ons – Heroku Postgres

Node-postgres

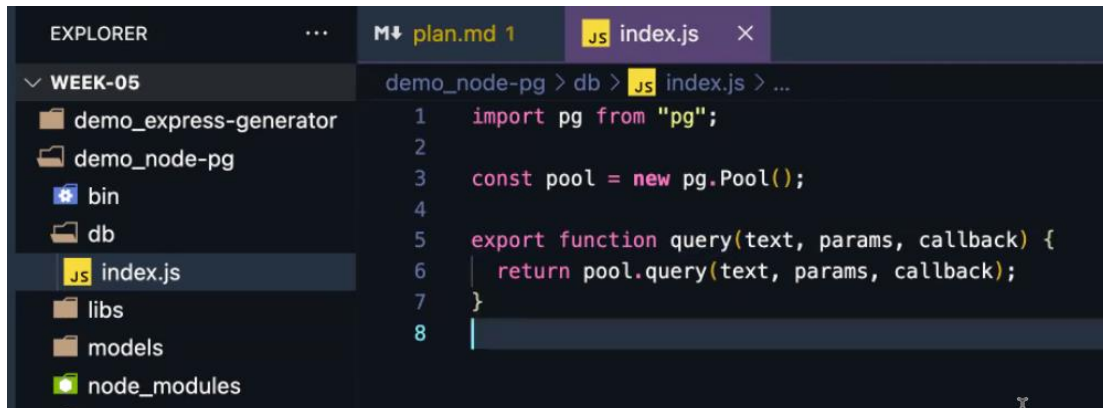
Collection of modules to interface with PostgreSQL DB.

`npm install pg`

Change the CommonJS import syntax to ESM (can still use object destructuring or import pg from "pg" then const pool = new pg.Pool())

Refactor the arrow function

Change the module.exports to export function (query) (CommonJS to ESM)



```
1 import pg from "pg";
2
3 const pool = new pg.Pool();
4
5 export function query(text, params, callback) {
6   return pool.query(text, params, callback);
7 }
8
```

Pooling

Connection pool

Think of a guy with a bunch of hoses that can be connected between server and databases.
The guy is the “pool”!

Query function

This query function converts our Node request in to a SQL query

Always need the text argument but params and callback are optional (but may be needed if doing anything other than simple SELECT * FROM table)

Configuring the Pool

Fill in all fields with details from Database Credentials from Heroku Postgres page

```
const pool = new Pool({
  user: 'dbuser',
  host: 'database.server.com',
  database: 'mydb',
  password: 'secretpassword',
  port: 3211,
  ssl: { rejectUnauthorized: false }, This is specific to Heroku!
})
```

But don't hardcode your credentials! These should be stored in **environment variables**:

Environment variables are a great way to securely and conveniently configure things that don't change often, like URLs, authentication keys and passwords.

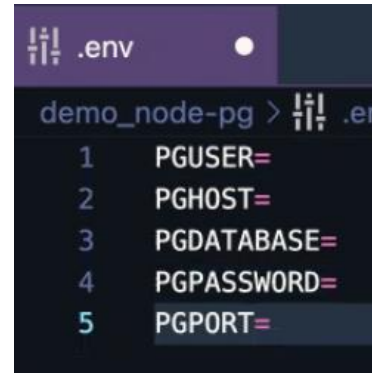
They are supported out of the box with Node and are accessible via the env object (which is a property of the process global object). But we are going to use DotEnv for now (<https://www.freecodecamp.org/news/how-to-use-node-environment-variables-with-a-dotenv-file-for-node-js-and-npm/>)

Convention is to write environment variables in all caps
No semicolon at end of the line

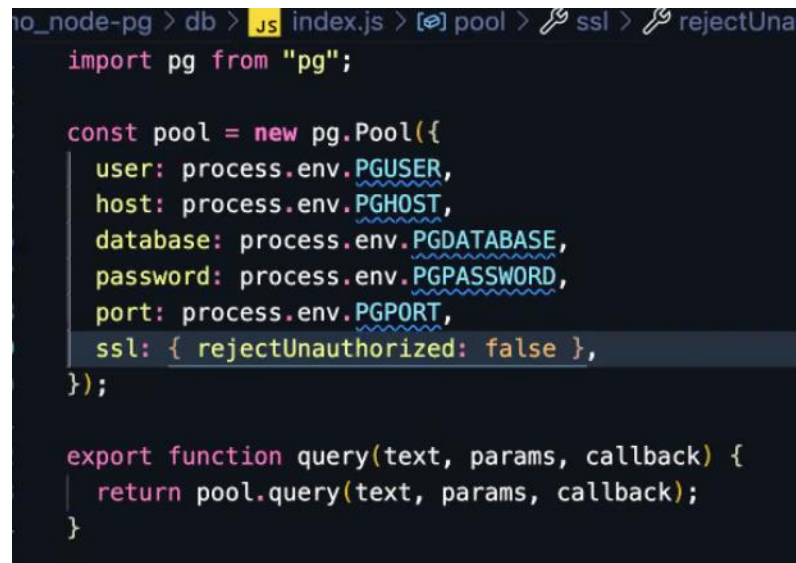
Npm I --save-dev dotenv
Can always use -D instead of --save-dev

Create .env file and put all the PG variables in:

Then swap these variables in to the index.js file using process.env.PGUSER for example:



```
.env
demo_node-pg > .env
1 PGUSER=
2 PGHOST=
3 PGDATABASE=
4 PGPASSWORD=
5 PGPORT=
```



```
demo_node-pg > db > JS index.js > pool > ssl > rejectUnauthorized
import pg from "pg";

const pool = new pg.Pool({
  user: process.env.PGUSER,
  host: process.env.PGHOST,
  database: process.env.PGDATABASE,
  password: process.env.PGPASSWORD,
  port: process.env.PGPORT,
  ssl: { rejectUnauthorized: false },
});

export function query(text, params, callback) {
  return pool.query(text, params, callback);
}
```

Then use the Preload section of the docs to see how to automatically load dotenv when it runs, using the script in the package-json, in the dev key:



```
{
  "name": "workshop-rest-express",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "start": "node ./bin/www.js",
    "dev": "nodemon -r dotenv/config ./bin/www.js"
  }
}
```

Then ensure to add .env to .gitignore (it may already be there as Express Generator sets it for you) so your passwords don't get uploaded to GitHub

Set up tables in our DB and get the data in them (populate them)

Create new folder inside db called scripts, then new file called createBooksTable.JS

The plan

- Import query function from db/index.js
- Write SQL string that creates our table
 - Columns: book_id INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY, author_id INT, title TEXT, publishedDate TEXT (for now as we are beginners but can use DATE!)
- Use our query function to communicate with the database (passing in SQL string as the text argument)

Import { query } from “../index.js”;

```
const sqlString = `CREATE TABLE IF NOT EXISTS books (book_id INT PRIMARY KEY  
GENERATED ALWAYS AS IDENTITY, author_id INT, title TEXT, publishedDate TEXT);`;
```

IF NOT EXISTS simply checks that the table doesn't already exist so you don't create duplicate tables or overwrite. Semicolon inside backticks is for SQL, the one outside is for JS.

```
async function createBooksTable() {  
  const res = await query(sqlString);  
  console.log('created books table');  could use something that you get back from SQL  
}
```

createBooksTable()

Note from Arshi: could get away with awaiting outside of async function as we are using ESM.

Then can use this as another script inside the package.json file:

“db:createBooksTable”: “node -r dotenv/config ./db/scripts/createBooksTable.js”

Then add it to scripts in package.json file and use npm run db:createBooksTable to run the script! Can check it's worked by visiting Heroku Postgres page and selecting overview tab, it should say 1 table!

```
1 {
2   "name": "workshop-rest-express",
3   "version": "0.0.0",
4   "private": true,
5   "type": "module",
6   "scripts": {
7     "start": "node ./bin/www.js",
8     "dev": "nodemon -r dotenv/config ./bin/www.js",
9     "db:createBooksTable": "node -r dotenv/config ./db/scripts/createBooksTable.js"
10  },
11  "dependencies": {
12    "cookie-parser": "~1.4.4",
13    "debug": "~2.6.9",
14    "express": "~4.16.1",

```

PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE bash - demo_node-pg + v [] [] ^ x

```
at Client._handleAuthSASLContinue (/Users/lizkaufman/Projects/SoC/BC12/03_demos/week-05/demo_node-pg/node_modules/pg/lib/client.js:257:10)
at Connection.emit (node:events:527:28)
at /Users/lizkaufman/Projects/SoC/BC12/03_demos/week-05/demo_node-pg/node_modules/pg/lib/connection.js:114:12
at Parser.parse (/Users/lizkaufman/Projects/SoC/BC12/03_demos/week-05/demo_node-pg/node_modules/pg-protocol/dist/parser.js:40:17)
at Socket.<anonymous> (/Users/lizkaufman/Projects/SoC/BC12/03_demos/week-05/demo_node-pg/node_modules/pg-protocol/dist/index.js:11:42)
at Socket.emit (node:events:527:28)
at addChunk (node:internal/streams/readable:315:12)
at readableAddChunk (node:internal/streams/readable:289:9)
at Socket.Readable.push (node:internal/streams/readable:228:10)
lizkaufman demo_node-pg npm run db:createBooksTable
```

Populate books table

Create populateBooksTable.js inside db/scripts folder

Plan

- Import my query from db/index
- INSERT INTO statement inserting values from our data into our table
 - Import data array
 - Loop through data array to visit each object
 - For each object, insert the data into our database
- Confirm data is successfully going in

```
import { query } from "../index.js";
```

```
import { books } from "../libs/data";
```

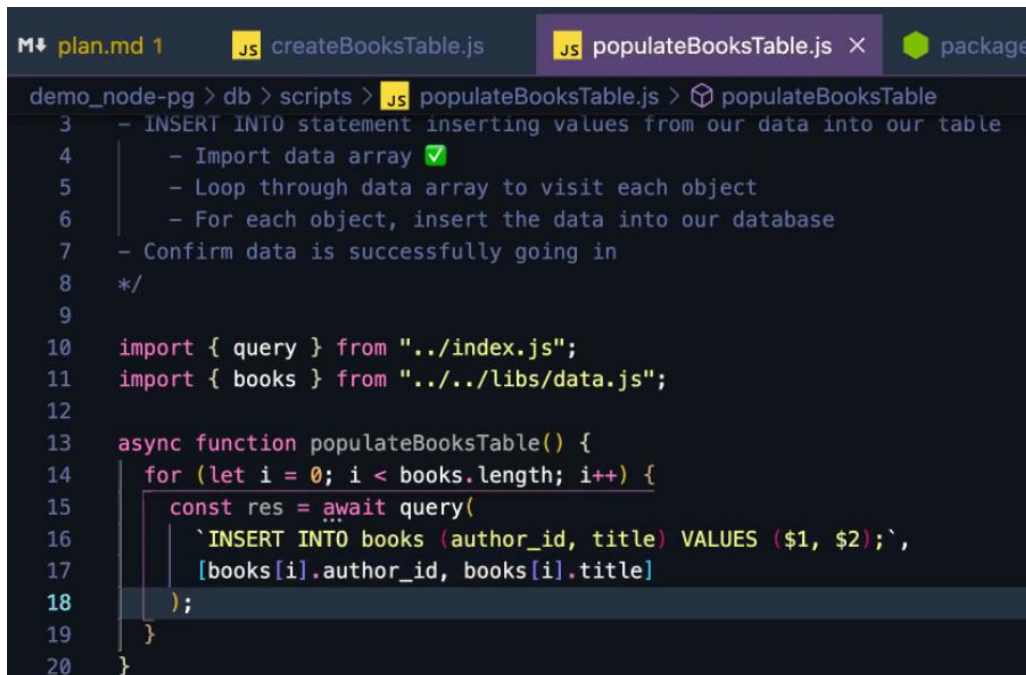
```
async function populateBooksTable(){
  for (let i = 0; i < books.length; i++) {
    Const res = await query(`INSERT INTO books (author_id, title, publishedDate)
VALUES (${books[i].author_id}, ${books[i].title}, ${books[i].publishedDate});`
    );
  }
}
```

This query is vulnerable to SQL injection as people could populate the books[i].author_id with SQL command like DROP TABLE “companies” which would delete the entire table!

So this is how we sanitise them:

```
Const res = await query(`INSERT INTO books (author_id, title, publishedDate) VALUES ($1, $2, $3);`, [books[i].author_id, books[i].title]);
```

This parameterises the query which under the hood will be sanitised to remove any ; and stuff that can cause problems.



```
demo_node-pg > db > scripts > JS populateBooksTable.js > populateBooksTable
3  - INSERT INTO statement inserting values from our data into our table
4  |   - Import data array ✓
5  |   - Loop through data array to visit each object
6  |   - For each object, insert the data into our database
7  |   - Confirm data is successfully going in
8  |   */
9
10 import { query } from "../index.js";
11 import { books } from "../../libs/data.js";
12
13 async function populateBooksTable() {
14   for (let i = 0; i < books.length; i++) {
15     const res = await query(
16       `INSERT INTO books (author_id, title) VALUES ($1, $2);`,
17       [books[i].author_id, books[i].title]
18     );
19   }
20 }
```

Add a console.log to show the loop is working each time then call the function:

```

mo_node-pg > db > scripts > js populateBooksTable.js > ...
10 import { query } from "../index.js";
11 import { books } from "../../libs/data.js";
12
13 async function populateBooksTable() {
14   for (let i = 0; i < books.length; i++) {
15     const res = await query(
16       `INSERT INTO books (author_id, title) VALUES ($1, $2);`,
17       [books[i].author_id, books[i].title]
18     );
19     console.log(`populated with ${books[i].title}`);
20   }
21 }
22
23 populateBooksTable();
24

```

Then add to scripts in package.json:

“db:populateBooksTable”: “node -r dotenv/config ./db/scripts/populateBooksTable.js”

```

> Debug
{
  "scripts": {
    "start": "node ./bin/www.js",
    "dev": "nodemon -r dotenv/config ./bin/www.js",
    "db:createBooksTable": "node -r dotenv/config ./db/scripts/createBooksTable.js",
    "db:populateBooksTable": "node -r dotenv/config ./db/scripts/populateBooksTable.js"
  },
  "dependencies": {

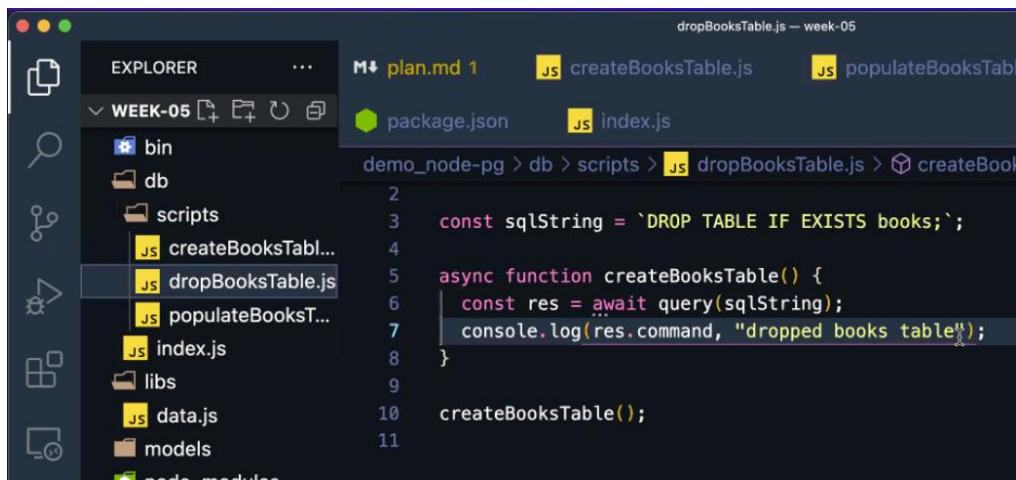
```

Then run using npm run db:populateBooksTable

During lunch, Liz added drop table script to reset table

Missing line 1 – which imports query from index.js

Res is an object that we get back from the DB. The command property tells you what you asked it...The data you get back is in res.rows.



And added 2 scripts to to package.json, first to just drop the table and then a chain of the 3 keys to reset table with && to drop, create and populate table from scratch.



Can also run Dataclip in Heroku Postgres directly to check the table is working

Day 4

Today was mostly recapping yesterday very slowly and in more detail as a lot of people spoke up yesterday to say they were struggling. My main takeaways were the expansion on the Environment Variable:

Environment Variables

Environment variables are a way of setting up what you need specific to your machine.

Plan

- Create a folder
- Initialise a node/npm project (npm init)
- Install dotenv as a dev dependency
- Create some environment variables
 - Put them in a .env file
- Read them into the file we are working with
 - Either import it at top of JS or add it to Scripts in package.json
- Check if they are there
 - If statement to check if undefined and if so, console.log this
- Use them

Created a folder on the desktop called env

Opened it in VS code and created index.js file in root directory

Made a plan with the above steps

Then in the terminal wrote npm init -y

This creates a package.json file in root directory

Created a .env in the root directory too – *this is a glorified text file (so JS syntax doesn't apply), a collection of key:value pairs such as PORT=3000, SECRET="lozhasasecret". The only convention you need is put the variable in all caps and separate each pair with a line (no semi-colon)*

[Dotenv](#) loads environment variables from a .env file in process.env. [Process is a global object](#) which has a property called env that is a global object.

Install it as Dev dependency as we only need it during development:

npm i dotenv -D (shorthand for npm install dotenv --save-dev)

Comment from Arshi – Dev dependencies are only used during “development”. In other words, they’re just tools you’ll use whilst developing. Once you’re done developing, you or your app won’t be using them.

Import it at top of JS file:

```
import 'dotenv/config';
```

If using ES6 syntax as above – it needs to be added as “type”:”module” to package.json (below “main” key)

You can also preload it in your scripts so you don’t need to import:

You can use the --require (-r) command line option to preload dotenv. By doing this, you do not need to require and load dotenv in your application code. Add this to your scripts object in package.json:

```
“scriptname”:"node -r dotenv/config your_script.js”
```

Test it’s working using console logs:

```
console.log(process.env.PORT);  
console.log(process.env.SECRET);
```

Run the file:

```
node index.js This is ok for single use but better to add it to package.json as script
```

“npm run” allows you to select commands from the “scripts” object in the package.json. e.g.:
npm run scriptnamekey

If you have a “start” script, you can just type npm start instead of npm run start.

Loz then added if checks for if process.env.PORT === defined then console.log “Warning port is undefined”.

Loz then went on to demonstrate adding it to scripts object in package.json instead of at top of file – have added comments on how in the individual steps above. This is to demonstrate doing things in more than one way – looks very different but has the same outcome.