Node & Postgres

Where We Are Now

- So far, we've made simple web apps with no persistent data
- And we've made simple databases with no application that reads or writes data to it
- Using what we know about Node and Postgres, let's combine these two skills to make a database-backed web app

Introducing node-postgres (pg)

```
npm install --save pg
```

```
const pg = require("pg");
```

- The pg module gives us an interface between Node and Postgres
- It provides a "pool" that we can execute the same SQL queries that we learned about, only this time in Node
- Database operations are asynchronous, much like file I/O
- The pg module uses either Promises, or node-style callbacks to handle its asynchronous nature

The "Pool" Pattern

- The "pool" pattern is a common principle seen throughout programming, not just for Postgres
- The idea is that rather than creating one new object each time and managing it individually, there's a shared pool of objects that we pull from to perform actions for us
- This allows us to more efficiently do things, since we're not making new objects for each request, and removing them when we're done
- Don't worry too much about the specifics, just know that pg won't make queries on its own, it'll make a pool that we run queries through

Using pg to make a pool

```
const pg = require("pg");
const pool = new pg.Pool({
    user: "postgres",
    database: "testdb",
    password: "supersecret123",
    host: "localhost",
    port: 5432,
});
```

- To run queries, we need to make a pg.Pool first
- We pass the pg.Pool constructor all of information to connect to our database
- We then use the query function on the pool to access the database

The `query` Function

```
const pg = require("pg");
const pool = new pg.Pool({ /* ... */ });

pool.query("SELECT * FROM users")
    .then(function(res) {
        console.log("All users", res.rows);
    })
    .catch(function(err) {
        console.error("Unable to get users from db", err);
    });
```

- The `query` function does as you'd expect, runs a query and returns a promise
- If it succeeds, you get an object back that includes some data about the query, but most importantly a `rows` key with an array of the results
- Even queries meant to grab one thing have a length 1 array

The `query` Function - Parameters

- Most of our queries will use dynamic parameters
- For instance, our url might be `/blog/8417`, and we'll want to get a blog post of ID "8417"
- We *never* use string concatenation for this, as people could inject harmful code in to our queries that way
- Instead, we use the second argument of `query()`, an array of values to insert in to the query
- In the query string we use `$1`, `$2`, `$3` to indicate replacing with item 1, 2, or 3 in the array

The `query` Function - Parameters (code)

```
// Return a promise that will resolve with a post object,
// given the post's unique ID
function getBlogPostById(id) {
    return pool.query("SELECT * FROM posts WHERE id=$1", [id])
        .then(function(result) {
            return result.rows[0];
        });
}


app.get("/blog/:postId", function(req, res) {
    getBlogPostById(req.params.postId)
        .then(function(post) {
            res.render("blogpost", { post: post });
        })
        .catch(function(err) {
            res.status(404).render("404");
        });
});
```

The `query` Function - Parameters (more code)

```
// Return a promise that will provide a list of search results,
// given the table, which field to search, and the search text
function searchTable(table, field, text) {
    const query = "SELECT * FROM $1 WHERE $2 LIKE '%$3%'";
    const args = [table, field, text];

    return pool.query(query, args).then(function(result) {
        return result.rows;
    });
}

app.post("/user/search", function(req, res) {
    searchTable("users", "username", req.query.username)
        .then(function(users) {
            res.render("search", { users: users })
        })
        .catch(function(err) {
            // Just show no users and log error
            console.error("Search encountered error", err);
            res.render("search", { users: [] });
        });
});
```

The `query` Function - Node Style Callbacks

- The `query` function also takes in an alternative 2nd / 3rd parameter, a node-style callback
- Promises are preferred, but you may see this used elsewhere
- The callback function is passed `(err, res)`
  - `err` being the same as `.catch(err)`
  - `res` being the same as `.then(res)`

```
pool.query("SELECT * FROM users", function(err, res) {
    if (err) {
        return console.error("Error!", err);
    }

    console.log("Niceee", res.rows);
})
```

Building a Query Module

- As the code examples showed, there are a lot of use cases for running queries in our Express apps
- However, it takes a lot of code to make one of those `pg.Pool`s
- So in most cases, we'd want to make one module that creates that `Pool` and exports a function for running queries
- It's quite a bit of code though, so let's go over it together:

https://gist.github.com/wbobeirne/6716b4a710c12beb17fb5f772e71391a

Using Your `query` Module

- Now that we've made it a module, it's as simple as requiring the file and calling it
- Make sure you keep your eyes on the console for any errors that come from mis-configuring the `pg.Pool`

```
const query = require("./path/to/query");

query("SELECT * FROM users").then(function(res) {
    console.log(res.rows);
});
```

Exercise: Bulletin Board

We're going to get a jumpstart on our homework by creating a Bulletin Board application. This is a place where people can post any notices they want, and we'll save them to a database and display them in order of when they were submitted

- The bulletin board will live in a Postgres database called `bulletinboard`
- It will have one table called `messages` with the following schema:
  - **id** - SERIAL (primary key)
  - **title** - VARCHAR (max length of 100)
  - **body** - TEXT (this is an unlimited length varchar)
  - **created** - TIMESTAMP (when the post was created)
- In addition to a `query` function module, you should create a `BulletinBoard` singleton module that has a `getMessages()` function and a `saveMessage(title, body)` function
- Save a few messages yourself manually, and have your homepage render a view that displays them all

Get more detail on the homework assignment here

Additional Reading

- Node-postgres (pg) module docs
- David Walsh's recap on Promises