NODE-PG

NODE-PG

So this is all great except we don't write SQL straight on the database. We need a way to interface our application with our database of choice. There are usually a number tools available for this kind of thing depending on the language. For Node, we will be using a node module called `node-pg` to help us interact with our postgres instance.

CRUD WITH NODE-PG

CRUD - READ

What do you notice?

```
pool.query("SELECT * FROM ShoppingCart").then(function(result) {
    console.log(result.rows);
});
```

CRUD - READ

- `pool` is a variable we use to access our Postgres database. (More on that soon.)
- SQL is used. It's in a String.
- query() returns a Promise so we need to use `then()`.
- The results are found on `result.rows`.

```
pool.query("SELECT * FROM Avengers").then(function(result) {
    console.log(result.rows);
});
```

CRUD - READ

`result.rows` will be an array of objects representing the matching rows in our Postgres table.

```
[
  { id: 1, name: "Bruce Banner", hero_name: "Hulk", primary_power: "Strength"},
  { id: 2, name: "Steve Rogers", hero_name: "Captain America", primary_power: "Tacti
  ...
]
```

CRUD - CREATE

What do you notice?

```
var sql = "INSERT INTO Avengers(name, hero_name, primary_power) "+
      "values($1::text, $2::text, $3::text);"
var values = ['Peter Parker', 'Spider-Man', 'Mouthing off'];

pool.query(sql, values).then(function() {
    console.log("Inserted.");
});
```

CRUD - CREATE

- SQL again... but with numbered parameters
- The values for those parameters are passed as an array.
- The numbers count from 1 not 0. (Womp womp.)
- A Promise again.

```
var sql = "INSERT INTO Avengers(name, hero_name, primary_power) "+
      "values($1::text, $2::text, $3::text);"
var values = ['Peter Parker', 'Spider-Man', 'Mouthing off'];

pool.query(sql, values).then(function() {
    console.log("Inserted.");
});
```

CRUD - UPDATE

What do you notice?

```
var sql = "UPDATE Avengers SET primary_power=$2::text WHERE hero_name=$1::text;"
var values = ['Spider-Man', 'Web Slinging'];

pool.query(sql, values).then(function() {
    console.log("Updated.");
});
```

CRUD - DELETE

What do you notice?

```
var sql = "DELETE FROM Avengers WHERE hero_name=$1::text;";
var values = ['Hawkeye'];

pool.query(sql, values).then(function() {
    console.log("Deleted.");
});
```

CRUD - READ WITH WHERE

We can also use parameters in a SELECT.

```
var sql = "SELECT * FROM Avengers WHERE name = $1::text;";
var values = ['Cluck Kent'];

pool.query(sql, values).then(function(result) {
    console.log(result.rows[0]);
});
```

PARAMETERS

- `$`(number)
- Numbered starting at 1.
- Good idea to specify type (`::text`, `::int`, `::real`, `::boolean`)

WITH EXPRESS

REST SERVER USING DATABASE

Client --> Express --> PostgreSQL

Client <-- Express <-- PostgreSQL

GET ALL ROOMS

```
app.get('/rooms', function(req, res) {
    res.send(...);
});
```

plus

```
pool.query("SELECT * FROM Rooms").then(function(result) {
    console.log(result.rows);
});
```

GET ALL ROOMS

What do you notice?

```
app.get('/rooms', function(req, res) {
    pool.query("SELECT * FROM Rooms").then(function(result) {
        res.send(result.rows);
    });
});
```

GET ALL ROOMS

It's also a good idea to handle errors. Otherwise we'll have a hard time figuring out what went wrong.

```
app.get('/rooms', function(req, res) {
    pool.query("SELECT * FROM Rooms").then(function(result) {
        res.send(result.rows);
    }).catch(function(err) {
        console.log(err);
        res.status(500); // 500 Server Error
        res.send("ERROR");
    });
});
```

POST & PUT

For POST and PUT methods, we need to get the JSON body of the request. This requires additional Express configuration. Add the following to the top of your server.js.

```
var bodyParser = require('body-parser');
app.use(bodyParser.json());
```

ADD A ROOM

`req.body` will give you the JSON body parsed to a JavaScript Object.

```
app.post('/rooms', function(req, res) {
    var room = req.body; // <-- Get the parsed JSON body
    var sql = "INSERT INTO Rooms(name, capacity, available) "+
        "VALUES ($1::text, $2::int, $3::boolean)";
    var values = [room.name, room.capacity, room.available];

    pool.query(sql, values).then(function() {
        res.status(201); // 201 Created
        res.send("INSERTED");
    });
});
```

GET A ROOM

In Express, you can put Variables in the URL.

```
app.get('/rooms/:id', function(req, res) {
  var id = req.params.id; // <-- This gets the :id part of the URL

  pool.query("SELECT * FROM Rooms WHERE id = $1::int", [id]).then(function(result) {

      if (result.rowCount === 0) {
        res.status(404); // 404 Not Found
        res.send("NOT FOUND");
      } else {
        // Return the first result. There should only be one.
        res.send(result.rows[0]);
      }

  });
});
```

SETTING UP NODE PG

Require the module

```
var pg = require('pg');
```

SETTING UP NODE PG

Create a *connection pool* and point it to the right database.

```
var pool = new pg.Pool({
    user: "postgres",
    password: "*****",
    host: "localhost",
    port: 5432,
    database: "postgres",
    ssl: false
});
```

CONNECTION POOL

Making a connection to the database is expensive (it takes extra milliseconds and CPU power). We can save time by reusing connections, keeping them in a pool.

It's like having a pool of 10 bikes that employees can use. You don't have to buy a new bike every time you want to go out for lunch.

CODE ALONG

REST ROOMS

CODE ALONG

Make a new folder for your project. Open it up in Atom and get there on your command prompt.

CODE ALONG

Run `npm init`, run through the prompts, and then...

```
npm install express --save
npm install pg --save
npm install body-parser --save
```

GET EXPRESS UP AND RUNNING

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

app.listen(5000, function () {
  console.log('JSON Server is running on ' + port);
});
```

ADD THE FIRST ROUTE

```
app.get('/rooms', function(req, res) {
    res.send('SUCCESS');
});
```

Test it with Postman.

SET UP THE DB CONNECTION

```
var pg = require('pg');

var pool = new pg.Pool({
    user: "postgres",
    password: "*****",
    host: "localhost",
    port: 5432,
    database: "postgres",
    ssl: false
});
```

SET UP DATABASE

Let's pause for a second to set up our database table in PG Admin.

```
CREATE TABLE Rooms (
    id SERIAL UNIQUE PRIMARY KEY,
    name VARCHAR(40),
    capacity INT,
    available BOOLEAN
);

INSERT INTO Rooms (name, capacity, available)
VALUES ('Sun Room', 30, FALSE);
INSERT INTO Rooms (name, capacity, available)
VALUES ('Green Room', 20, FALSE);
INSERT INTO Rooms (name, capacity, available)
VALUES ('Penthouse', 100, TRUE);
```

FETCH ROOMS FROM THE DB

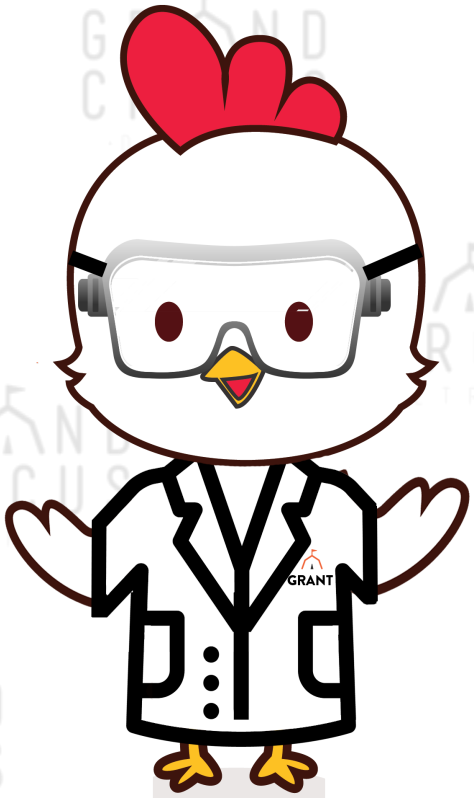
```
pool.query("SELECT * FROM Rooms").then(function(result) {
    res.send(result.rows);
});
```

QUICK NOTE

The module we're using, `node-pg`, is not an Object-Relation Map (ORM). An ORM is a way of mapping a table, and all of its columns, to a data strcture (usually an object). There are ORMs for Node but we don't have time to look at them in detail. We just want to make sure you know what they are.

LAB 24

FULL-STACK SHOPPING CART



FULL-STACK SHOPPING CART

Clone this Repo.

We are giving you an Angular application with all the controllers and views set up. Your job is to create the back-end and connect the end-points to your front-end in a service.

As a user, I should be able to add and remove items on a grocery list. If you complete the first step, figure out how to give the user the ability to edit the items on the list.

INSTRUCTIONS

1. In pgAdmin, create a table called ShoppingCart with the following columns:
   - *id*: auto-generated ID number
   - *product*: holds a 40 character string
   - *price*: holds a decimal number
2. In server.js, add routes to
   - *GET /api/items*: Get all items in the cart as an array.
   - *POST /api/items*: Add an item to the cart.
   - *DELETE /api/items/_ID_*: Remove an item from the cart.
3. Test these routes with *Postman*.

INSTRUCTIONS

4. In cartService.js, fill in the TODOs with calls to your Node endpoints using Angular's built-in $http service.
5. Now you can test your app in the browser.

BONUS

- Add quantity to the shopping cart items.
- Add a total to the shopping cart. (You can use JavaScript or SQL to calculate the total.)
- Add functionality to update an item in the cart. This could be changing just one attribute (eg. quantity) or all the attributes. For this, create a new route in server.js (PUT /api/items/_ID_).

HINT

For the Angular side, here's a good implementation of cartService.getAllItems(). It uses promise chaining.

```
this.getAllItems = function() {
    // GET /api/items
    return $http.get("/api/items").then(function(response) {
        return response.data;
    });
};
```

HINTS

- Create the database in pgAdmin
- Create the table in pgAdmin.
- Set up your basic express app with just console logs to start. Test your endpoints.
- Once you have your routes working, start implementing your database logic.
- It might be easier to add some data to your tables manually and then try to implement a GET route.
- Once that works, try to programmatically add data using a POST