Intro to Postgres with Node

Goals

Download Demo Code <../express-pg-intro-demo.zip>

- Use ***pg*** to connect and execute SQL queries
- Explain what SQL injection is and how to prevent it with ***pg***
- Examine CRUD on a single resource with Express and ***pg***

Introduction

The Node SQL Ecosystem

- ORMs
- Query builders
- SQL driver (what we will be using)
- You can read more about <https://www.rithmschool.com/blog/different-approaches-express> it from the one and only Joel Burton!

pg

Scaffolding for Our Demo

demo/simple/app.js

```
/** Express app for pg-intro-demo */

const express = require("express");
const app = express();
const ExpressError = require("./expressError");

// Parse request bodies for JSON
app.use(express.json());

const uRoutes = require("./routes/users");
app.use("/users", uRoutes);

// ... 404, global err handler, etc.
```

pg

- Similar to ***psycopg2*** with python

- Allows us to establish a connection to a database and execute SQL

```
$ npm install pg
```

Using *pg*

It's common to abstract this logic to another file, so let's create a file ***db.js***:

demo/simple/db.js

```
/** Database setup for users. */

const { Client } = require("pg");

let DB_URI;

if (process.env.NODE_ENV === "test") {
  DB_URI = "postgresql:///users_test";
} else {
  DB_URI = "postgresql:///users";
}

let db = new Client({
  connectionString: DB_URI
});

db.connect();

module.exports = db;
```

What did we just do?

- Specified a database to connect to
  - Depending on an environment variable we specify to use the test DB or not
  - We're going to need this conditional logic later when we test!
- Established a connection
- Exported out the connection

Queries

Making our first query

demo/simple/routes/users.js

```
const db = require("../db");
```

(results)

```
[nothing!]
```

demo/simple/routes/users.js

```
/** Get users: [user, user, user] */

router.get("/all", function (req, res, next) {
  const results = db.query(
        `SELECT * FROM users`);

  return res.json(results.rows);
});
```

What's the bug here?

DB queries are asynchronous! We have to wait for the query to finish before!

Fixing with async/await

demo/simple/routes/users.js

```
/** (Fixed) Get users: [user, user, user] */

router.get("/", async function (req, res, next) {
  try {
    const results = await db.query(
          `SELECT id, name, type FROM users`);

    return res.json(results.rows);
  }

  catch (err) {
    return next(err);
  }
});
```

(results)

```
[
  {
    "id": 1,
    "name": "Juanita",
    "type": "admin"
  },
  {
    "id": 2,
    "name": "Jenny",
    "type": "staff"
  },
  {
    "id": 3,
    "name": "Jeff",
    "type": "user"
  }
]
```

API Example Continued: Search

demo/simple/routes/users.js

```
/** Search by user type. */

router.get("/search", async function (req, res, ne
  try {
    const type = req.query.type;

    const results = await db.query(
      `SELECT id, name, type
       FROM users
       WHERE type='${type}'`);

    return res.json(results.rows);
  }

  catch (err) {
    return next(err);
  }
});
```

(results for 'staff' type)

```
[{
  "id": 2,
  "name": "Jenny",
  "type": "staff"
}]
```

But there's a problem...

SQL Injection

What is SQL Injection?

A technique where an attacker tries to execute undesirable SQL statements on your database.

It's a common attack, and it's easy to be vulnerable if you aren't careful!

What's the Problem?

If our search type is `"staff"`, everything works fine.

But what if our search type is `"bwah-hah'; DELETE FROM users; --"` ?

```
SELECT id, type, name
FROM users
WHERE type='bwah-hah'; DELETE FROM users; --'
```

Ut oh.

Solution: Parameterized Queries

- To prevent against SQL injection, we need to sanitize our inputs
- ORMs typically do this for you automatically
- We can sanitize our inputs by using **parameterized queries**

> Note: Prepared Statements

> It's not the most important part to understand, but if you're curious how the ***pg*** module does this, it uses a feature called "prepared statements".
>
> Prepared statements are a database tool used to templatize and optimize queries you plan on running frequently. You've seen prepared statements already when we worked with SQLAlchemy in Flask, though we didn't specifically call them out as such.
>
> You don't need to worry about the details, but because of the way that prepared statements work on the database level, they naturally protect against SQL injection. If you're curious about the details, check out this <https://en.wikipedia.org/wiki/Prepared_statement> article on Wikipedia.

API Example Continued: Create V2

Here's the same approach, but safe from SQL injection.

demo/simple/routes/users.js

```
// (Fixed) Search by user type. */

router.get("/good-search",
      async function (req, res, next) {
  try {
    const type = req.query.type;

    const results = await db.query(
      `SELECT id, name, type
       FROM users
       WHERE type=$1`, [type]);

    return res.json(results.rows);
  }

  catch (err) {
    return next(err);
  }
});
```

(results for 'staff' type)

```
[{
  "id": 2,
  "name": "Jenny",
  "type": "staff"
}]
```

Parameterized Queries Overview

- In your SQL statement, represent variables like ***$1***, ***$2***, ***$3***, etc.
  - You can have as many variables as you want
- For the second argument to ***db.query***, pass an array of values
  - ***$1*** will evaluate to the first array element, ***$2*** to the second, etc.
- **Note:** the variable naming is 1-indexed!

More CRUD Actions

API Example Continued: Create

demo/simple/routes/users.js

```
/** Create new user, return user */

router.post("/", async function (req, res, next) {
  try {
    const { name, type } = req.body;

    const result = await db.query(
          `INSERT INTO users (name, type)
           VALUES ($1, $2)
           RETURNING id, name, type`,
        [name, type]
    );

    return res.status(201).json(result.rows[0]);
  }

  catch (err) {
    return next(err);
  }
});
```

> **Note: Status Code 201**
>
> Note that we use HTTP status code 201 ("Created") here, not 200 ("Ok").
>
> Some APIs do return 200 for object-was-created, but the REST standard suggests 201 is the proper code here.

RETURNING clause

In SQL, for INSERT/UPDATE/DELETE, you can have a ***RETURNING*** clause.

This is to *return data* that was inserted, updated, or deleted:

```
INSERT INTO users (name, type) VALUES (...) RETURNING id, name;

INSERT INTO users (name, type) VALUES (...) RETURNING *;
```

> **Note: Security Warning About `*`**
>
> It's typically a bad idea to use `SELECT *` or `RETURNING *` in the SQL used in applications. That returns all columns and, if new sensitive columns were added after the code was written, it would risk returning that sensitive data.
>
> It's far better to explicitly list the columns that should be selected or returned.

API Example Continued: Update

demo/simple/routes/users.js

```
/** Update user, returning user */

router.patch("/:id", async function (req, res, next) {
  try {
    const { name, type } = req.body;

    const result = await db.query(
          `UPDATE users SET name=$1, type=$2
           WHERE id = $3
           RETURNING id, name, type`,
        [name, type, req.params.id]
    );

    return res.json(result.rows[0]);
  }

  catch (err) {
    return next(err);
  }
});
```

API Example Continued: Delete

demo/simple/routes/users.js

```
/** Delete user, returning {message: "Deleted"} */

router.delete("/:id", async function (req, res, next) {
  try {
    const result = await db.query(
        "DELETE FROM users WHERE id = $1",
        [req.params.id]
    );

    return res.json({message: "Deleted"});
  }

  catch (err) {
    return next(err);
  }
});
```

Committing

With SQLAlchemy, you had to commit after all changes — because SQLAlchemy put all work into a db transaction.

That isn't the case with ***pg*** — so you don't need to explicitly commit (each INSERT/UPDATE/DELETE commits automatically)

Testing our Database

Adding a test database

We're going to need a different database for testing, so let's configure that!

demo/cats-api/db.js

```
/** Database setup for cats. */
const { Client } = require("pg");

const DB_URI = (process.env.NODE_ENV === "test")
  ? "postgresql:///cats_test"
  : "postgresql:///cats";

let db = new Client({
  connectionString: DB_URI
});

db.connect();

module.exports = db;
```

Running Tests

Make sure you create test database first, otherwise they will hang.

```
$ createdb cats_test
$ psql cats_test < data.sql
```

Once you have database, run your tests as usual with ***jest***

Setting Up and Tearing Down the test suite

Make sure we're using test DB for our tests:

demo/cats-api/routes/cats.test.js

```
// connect to right DB --- set before loading db.js
process.env.NODE_ENV = "test";

// npm packages
const request = require("supertest");

// app imports
```

```
const app = require("../app");
const db = require("../db");
```

Setup at beginning:

demo/cats-api/routes/cats.test.js

```
let testCat;

beforeEach(async function() {
  let result = await db.query(`
    INSERT INTO
      cats (name) VALUES ('TestCat')
      RETURNING id, name`);
  testCat = result.rows[0];
});
```

Teardown at end:

demo/cats-api/routes/cats.test.js

```
afterEach(async function() {
  // delete any data created by test
  await db.query("DELETE FROM cats");
});

afterAll(async function() {
  // close db connection
  await db.end();
});
```

Testing CRUD Actions

Our Restful JSON API

What routes do we need for a RESTful JSON API with full CRUD on cats? (ZOMG so many acryonyms.)

HTTP Verb	Route	Response
GET	/cats	Display all cats
GET	/cats/:id	Display a cat
POST	/cats	Create a cat
PUT / PATCH	/cats/:id	Update a cat
DELETE	/cats/:id	Delete a cat

Testing Read

demo/cats-api/routes/cats.test.js

```
/** GET /cats - returns `{cats: [cat, ...]}` */

describe("GET /cats", function() {
  test("Gets a list of 1 cat", async function() {
    const response = await request(app).get(`/cats`);
    expect(response.statusCode).toEqual(200);
    expect(response.body).toEqual({
      cats: [testCat]
    });
  });
});
```

demo/cats-api/routes/cats.test.js

```
/** GET /cats/[id] - return data about one cat: `{cat: cat}` */

describe("GET /cats/:id", function() {
  test("Gets a single cat", async function() {
    const response = await request(app).get(`/cats/${testCat.id}`);
    expect(response.statusCode).toEqual(200);
    expect(response.body).toEqual({cat: testCat});
  });

  test("Responds with 404 if can't find cat", async function() {
    const response = await request(app).get(`/cats/0`);
    expect(response.statusCode).toEqual(404);
  });
});
```

Testing Create

demo/cats-api/routes/cats.test.js

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function() {
  test("Creates a new cat", async function() {
    const response = await request(app)
      .post(`/cats`)
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toEqual(201);
    expect(response.body).toEqual({
      cat: {id: expect.any(Number), name: "Ezra"}
    });
  });
});
```

Testing Update

demo/cats-api/routes/cats.test.js

```
/** PATCH /cats/[id] - update cat; return `{cat: cat}` */

describe("PATCH /cats/:id", function() {
  test("Updates a single cat", async function() {
    const response = await request(app)
      .patch(`/cats/${testCat.id}`)
      .send({
        name: "Troll"
      });
    expect(response.statusCode).toEqual(200);
    expect(response.body).toEqual({
      cat: {id: testCat.id, name: "Troll"}
    });
  });

  test("Responds with 404 if can't find cat", async function() {
    const response = await request(app).patch(`/cats/0`);
    expect(response.statusCode).toEqual(404);
  });
});
```

Testing Delete

demo/cats-api/routes/cats.test.js

```
/** DELETE /cats/[id] - delete cat,
 *  return `{message: "Cat deleted"}` */

describe("DELETE /cats/:id", function() {
  test("Deletes a single a cat", async function() {
    const response = await request(app)
      .delete(`/cats/${testCat.id}`);
    expect(response.statusCode).toEqual(200);
    expect(response.body).toEqual({ message: "Cat deleted" });
  });
});
```

Looking Ahead

Coming Up

- Associations with pg
- Building our own lightweight ORM!