INTRO

Pre-reading

Reading and writing to a PostgreSQL database occurs through a number of layers. Let's take a look at those:

The DBMS: `postgres`

The Database Management System is the brain of our persistent storage solution. It is responsible for translating declarative incoming *queries* into actual *file system operations*. In the case of PostgreSQL, the DBMS is the `postgres` process running behind the scenes. In fact, `postgres` is a TCP/IP **server** listening on a port! …Not an HTTP-protocol server, but rather a Postgres-protocol server.

DBMS Clients

The `postgres` DBMS/server sits in the background ready to receive queries. So how do we actually transmit a query to `postgres`? A **client** must connect to it and send some SQL according to the protocol that `postgres` expects.

The `psql` client: a human interface (shell)

The `psql` shell is a client designed to accept SQL queries input via keyboard by a human being, transmit those queries to `postgres`, and print the results. It is

conceptually similar to the Node.js REPL, or a Bash/Zsh prompt. Internally, `psql` knows how to form a connection to & communicate with `postgres`.

The `pg` client: an application interface (driver)

Our Wizard News application isn't a person and has no need for a keyboard entry prompt. If we studied the protocol for connecting to `postgres` and transmitting queries, we might be able to implement that complex logic within our Node app, and the app would be a database client. That would be a lot of work, though, and very error-prone. Imagine every developer having to write their own TCP/IP code; it probably wouldn't go very well!

Instead, we can rely on a **database driver**. A driver is a client written to bridge the gap between applications and the DBMS. Like `psql` or any other client, it correctly implements the protocols to speak with `postgres`. But where `psql` exposes a keyboard input method, drivers expose an **API** — Application Programming Interface. Apps written in their native language can then use that API to send queries.

The client we'll use in our Node server is node-postgres, available on npm as `pg`. This driver allows us to use JavaScript to send queries to our database. The final path for a SQL query is as follows:

SENDING THE DATABASE REQUEST

1. Node app: our code calls a function provided by `pg`, passing a string SQL query

2. `pg` driver: connects to `postgres` as a client and sends the SQL query using correct protocol
3. `postgres` server: translates query into executing a series of file system operations

RECEIVING THE DATABASE RESPONSE

1. `postgres` server: sends result of file system ops back to connected `pg` driver
2. `pg` driver: parses the raw response and builds a JS array of row data
3. Node app: receives a Promise that will eventually contain the array.

Summary

There's a lot of jargon stuffed into this page, some of it overlapping. We've put a list of terms in the Epilogue of this workshop for your reference and study.

SETTING UP POSTGRES

Intro

Our Wizard News app is great, but when the Node process ends we lose all our data! Today, we're going to fix that by adding a SQL (specifically postgres) database. We're going to completely replace the `postBank.js` file. You may delete it if you wish.

Use the Wizard News Part 2 Starting Point (will share the zip file) as your starting point, or your own solution if you wish. If you use your own solution, be aware that the forthcoming review will be relative to the shared starting point, and will be more difficult to follow if you are working from your own implementation.

Creating a new database

Now that you're familiar with postgres and the `psql` command line, let's go ahead and create our first database. Go ahead and use the CREATE DATABASE command to create a database called wnews.

Do you remember how to create a database?

In the command line:

```
CREATE DATABASE dbname;
```

```
                                List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
 wnews     | fsa      | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

Define the tables

Now we're going to store Users who have "names", and Posts with title, body and references to their author IDs.

Notice that there is not an upvotes column for posts - Let's ignore it for now (We will talk about it again in the Bonus section).

First, let's connect to our database using the `\c` command

```
bs=# \c wnews
```

Next, go ahead and use some SQL to create these tables:

```
CREATE TABLE users (
id SERIAL PRIMARY KEY,
name TEXT DEFAULT NULL
);

CREATE TABLE posts (
id SERIAL PRIMARY KEY,
userId INTEGER REFERENCES users(id) NOT NULL,
title varchar(255) DEFAULT NULL,
content TEXT DEFAULT NULL,
date timestamp DEFAULT now()
);
```

If you use the `\d` command, you should now see Users and Posts!

If you use the `\d+` command, you can see detailed schema information. This command will be useful when you are frequently changing your schema later.

By the way, wondering what the `SERIAL` business is all about? This is the equivalent of `AUTOINCREMENT` in other database management systems - whenever a new post or user is created, it will automatically increment the primary key! This prevents you from needing to explicitly include a primary key when adding rows to your table.

Seed the database

A common need during development is to be able to quickly and easily *seed* a database — fill it with realistic (albeit often fake) data so we can test how our application works. To that end, a *seed script* comes in handy. We've gone ahead and made a SQL script for you (either in the repo or at this gist) which you can use. First, download the file or copy its contents into a new file. Then, do one of the following:

OPTION A: FROM THE TERMINAL PROMPT

```
psql -d your_database_name -a -f path/to/your/seed.sql
```

OPTION B: FROM THE PSQL PROMPT, AFTER CONNECTING TO THE DATABASE YOU WANT TO SEED:

```
\i path/to/your/seed.sql
```

Confirm successful database seeding with some basic `SELECT` statements for both tables.

Installing the client

OK, so we know what a driver is now — let's use it! Begin by installing the module.

```
[user@localhost]$ npm install pg
```

Check out the docs and see if you can get your server talking to your database. The next section will go into detail on our approach, but try it without any guidance first.

Connecting to the database

Our routes will need access to the database client. As the application grows, it might need to reference the database client from multiple places. Let's abstract this client object to a module so we can easily `require` it.

Create a new folder, `db`, and place an `index.js` in it. Inside that new file, `require` the node-postgres package `pg` and create a connection to our database. You can

try to set this up yourself by following the documentation here. Note two exceptions:

> With a normal Postgres installation (using `trust` authentication), you can omit the username and password from the connection string.
> You don't have to `await` for `connect`; `connect` can behave synchronously and will `throw` an error if the connection fails.

Export the client from this module. If you think you've got it, check the hint tag below for our version.

```
// setting up the node-postgres driver

const pg = require('pg');

const postgresUrl = 'postgres://localhost/___YOUR_DB_NAME_HERE___';

const client = new pg.Client(postgresUrl);

// connecting to the `postgres` server

client.connect();

// make the client available as a Node module

module.exports = client;
```

The Big Step

OK, it's time to make some magic (bad joke...).

Import the `client` module in your `app`. Next, modify your route handlers to talk to the **database** instead of to `postBank`.

For example:

```
const data = await client.query('SELECT * FROM posts');
const posts = data.rows;
res.send(postList(posts));
```

The `query` method returns a promise. The returned value is an object that contains some metadata attached to it, but if we just want the rows returned, we can access them via `.rows`.

Note:

Remember that in order to use the `await` keyword you need to have an `async` function. As it turns out, you can simply turn the route handler function into `async`:

.

```
app.get("/", async (req, res) => {
  /* Route Handler */
})
```

Some asynchronous code are naturally prone to fail. Making database query, for example, may fail for a variety of reasons (the server is offline, the sent query is invalid or incorrect, etc).

JavaScript provides a try...catch mechanism to run code that can throw errors in a controlled fashion:

> The try statement allows you to define a block of code to be tested for errors while it is being executed.
> The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

In your route handlers make sure to add try...catch statements. But, what should you do if an error occurs? Two things:

1. `console.log` the error so you can see it in the terminal.
2. Send an http response with the appropriate status

In practice, it would look like this:

```
app.get("/", async (req, res) => {
  try {
    const data = await client.query(/*Some query*/);
    res.send(postList(data.rows));
  } catch (error) {
    console.error(error);
    res.status(500).send(`Something went wrong: ${error}`);
  }
});
```

But wait, there a better alternative!

Express provide a built-in error handler. If you simply pass an error to next() and do not handle it yourself, it will be handled by the built-in error handler - the error will be written to the client with the stack trace (in development environment) and a response with the appropriate status will be sent:

```
app.get("/", async (req, res, next) => {
  try {
    const data = await client.query(/*Some query*/);
    res.send(postList(data.rows));
  } catch (error) { next(error) }
});
```

We can quickly get into a sticky situation once we need to include data from our client into our queries. Fortunately, node-postgres provides a syntax for string interpolation, or parameterized queries.

To use it, you can include numbers appended with the `$` symbol in your querystring, and include an array of *arguments* to pass to the query (note: index `0` of the array corresponds to `$1` in the query):

This serves two purposes. First, it makes it much easier to pass in data to our queries. Second, and more importantly, behind the scenes this protects against

```
await client.query('SELECT name FROM users WHERE name=$1', ['Nimit']);

await client.query('INSERT INTO posts (userId, content) VALUES ($1, $2)', [10, 'I love SQL!']);
```

SQL injection. We won't want any cleverly named users wreaking havoc on our tables now, do we?

EXTRA CREDIT

Upvotes: Database Design, table & seed

Congratulations, your application data is now safely persisted in a database, and you are interacting with it from your JavaScript code.

There's one thing missing from the original implementation, though: upvotes. We specifically avoided adding a "votes" column to the `posts` table - Can you thing of any reasons behind this design decision?

Database Design

As you learned in the "Intro to Schema Design" lecture, there are lot's of different possible approaches on how to store different pieces of information and it's relationships in a set of tables. Ideally, one should strive for eliminating data redundancy while ensuring data integrity and accuracy, but ultimately the

decision on where to store and on how serialized or deserialized your data will be lies in correctly identifying the purpose of the pieces of information you need.

In the Wizard News App case, we want no only to know how many upvotes a given post have, but also be able to identity which user upvoted in each post, for a variety of reasons:

> We want to allow an user to vote only one time on a given article,
>
> We want to allow an user to undo his upvote.

For this reason, we opted for an association table called upvotes.

`upvotes` association table.

In the Wizard News app, an user can upvote as many posts as they want. Conversely, a post can be upvoted by multiple uses - this is called a many-to-many association between users and posts - and the classic resolution for this type of association in relational data it to create an extra table to record the relationship. We will call this table `upvotes`, and it will consist of two foreign keys (userId and postId) as well as a date - go ahead and use SQL to create this table:

```
CREATE TABLE upvotes (
userId INTEGER REFERENCES users(id) NOT NULL,
postId INTEGER REFERENCES posts(id) NOT NULL,
date timestamp DEFAULT now()
);
```

Next, you're going to seed this table with some data. We prepared a new seed file for upvotes - download or copy its contents into a new file and then seed it in the wnews database.

Counting upvotes

After seeding, your upvotes table should have over 60 rows (since every vote is registered as a new row). Your next job is to count the number of votes each post has. If you're starting to get concerned about the volume of rows in the table and the amount of counting the database will have to do, don't be: Relational Database Management Systems are designed to handle millions of rows, and count()-operations are very fast.

COUNT & GROUP BY

The SQL `GROUP BY` Clause is used to combine rows under some column value. It is typically used in conjunction with aggregate functions such as `COUNT` to summarize values. Here is an example on how to count all entries in `votes` for each given `postId`:

```
SELECT postId, COUNT(*) as upvotes FROM upvotes GROUP BY postId;
```

Your turn

Now it's your turn: Change your SQL clauses in `app.js` to include the number of upvotes for each post.

You will most likely need a sub query in this case. Remember that a Subquery (or Inner query) is a nested query is a query within another SQL query and embedded within the WHERE clause. Subqueries can be named and joined with other tables - Here's an example of a query that selects all fields from `posts` joined with the count of upvotes:

```
SELECT posts.*, counting.upvotes
FROM posts
INNER JOIN (SELECT postId, COUNT(*) as upvotes FROM upvotes GROUP BY postId) AS counting
ON posts.id = counting.postId;</
```

Epilogue

This workshop is a relatively small one and meant to serve as a conceptual bridge between working with the database directly (SQL via a shell client) and programming an application with an object-relational-mapper (ORM, e.g. Sequelize). We saw that using the `pg` *database driver*, we could make SQL queries from inside our Node application, and use the results.

Main Takeaways

The `postgres` DBMS process acts as a server (via TCP/IP) for the postgres protocol; it converts incoming requests to file system operations and sends back responses

The `pg` module is a postgres driver written for Node; it implements the postgres protocol, enabling us to use JavaScript to send queries to the `postgres` process.

A persistent database lets us store data on the disk, so that even between application server instances (e.g. restarting the server) our information does not disappear.

Writing raw SQL queries as long strings in our JS app is not exactly elegant, but it gets the job done.

Lexicon

It is not critical to know all the jargon in-depth. It is more important to gain a practical command of the available tools, followed by a conceptual grasp of the components and how they fit together. For your reference, however, we attempt to disambiguate some of the terminology below. Unfortunately, the word "postgres" is sometimes loosely used to mean *any* of the following:

Generic Term	Explanation	PostgreSQL example / notes
DBMS	A standardized solution / tool for storing data. A format, and program that can safely & quickly manage stored data in that format.	the entire PostgreSQL system in a holistic sense
database server	a process which listens to a TCP port for incoming database requests, and converts those requests into safe and fast filesystem actions	`postgres` process
database protocol	the rules of how to connect and exchange messages with a database server, implemented in drivers / clients	postgres protocol; most devs never look at it
query language	the syntax for messages which a given database server can understand and act on,	postgres; like SQL, but with some unique syntax, e.g.

	transmitted via the database protocol	`returning`; devs sometimes write it by hand
database driver	a software library which bridges the a certain db server and application code written in a particular language; lets that app become a database client	Node-Postgres, aka `pg`; lets your Node app become a postgres client
database client	a process which can connect to a certain db server, sending queries and handling resulting data	`psql` and Postico; used by humans, not by apps