Connecting to PostgreSQL

Environment variables

node-postgres uses the same environment variables as libpq to connect to a PostgreSQL server. Both individual clients & pools will use these environment variables. Here's a tiny program connecting node.js to the PostgreSQL server:

```
const { Pool, Client } = require('pg')

// pools will use environment variables
// for connection information
const pool = new Pool()

pool.query('SELECT NOW()', (err, res) => {
  console.log(err, res)
  pool.end()
})

// you can also use async/await
const res = await pool.query('SELECT NOW()')
await pool.end()

// clients will also use environment variables
// for connection information
const client = new Client()
await client.connect()

const res = await client.query('SELECT NOW()')
await client.end()
```

To run the above program and specify which database to connect to we can invoke it like so:

```
$ PGUSER=dbuser \
  PGHOST=database.server.com \
  PGPASSWORD=secretpassword \
  PGDATABASE=mydb \
  PGPORT=3211 \
  node script.js
```

This allows us to write our programs without having to specify connection information in the program and lets us reuse them to connect to different databases without having to modify the code.

The default values for the environment variables used are:

```
PGHOST='localhost'
PGUSER=process.env.USER
PGDATABASE=process.env.USER
PGPASSWORD=null
PGPORT=5432
```

Programmatic

node-postgres also supports configuring a pool or client programmatically with connection information. Here's our same script from above modified to use programatic (hard-coded in this case) values. This can be useful if your application already has a way to manage config values or you don't want to use environment variables.

```
const { Pool, Client } = require('pg')

const pool = new Pool({
  user: 'dbuser',
  host: 'database.server.com',
  database: 'mydb',
  password: 'secretpassword',
  port: 3211,
})

pool.query('SELECT NOW()', (err, res) => {
  console.log(err, res)
  pool.end()
})

const client = new Client({
  user: 'dbuser',
  host: 'database.server.com',
  password: 'mydb',
  password: 'secretpassword',
  port: 3211,
})
client.connect()

client.query('SELECT NOW()', (err, res) => {
  console.log(err, res)
  client.end()
})
```

Connection URI

You can initialize both a pool and a client with a connection string URI as well. This is common in environments like Heroku where the database connection string is supplied to your application dyno through an environment variable. Connection string parsing brought to you by pg-connection-string.

```
const { Pool, Client } = require('pg')
const connectionString =
'postgresql://dbuser:secretpassword@database.server.com:3211/mydb'

const pool = new Pool({
  connectionString: connectionString,
})

pool.query('SELECT NOW()', (err, res) => {
  console.log(err, res)
  pool.end()
})
```

```
const client = new Client({
  connectionString: connectionString,
})
client.connect()

client.query('SELECT NOW()', (err, res) => {
  console.log(err, res)
  client.end()
})
```

Queries

The api for executing queries supports both callbacks and promises. I'll provide an example for both *styles* here. For the sake of brevity I am using the `client.query` method instead of the `pool.query` method - both methods support the same API. In fact, `pool.query` delegates directly to `client.query` internally.

Text only

If your query has no parameters you do not need to include them to the query method:

```
// callback
client.query('SELECT NOW() as now', (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
    console.log(res.rows[0])
  }
})

// promise
client.query('SELECT NOW() as now')
  .then(res => console.log(res.rows[0]))
  .catch(e => console.error(e.stack))
```

Parameterized query

If you are passing parameters to your queries you will want to avoid string concatenating parameters into the query text directly. This can (and often does) lead to sql injection vulnerabilities. node-postgres supports paramterized queries, passing your query text *unaltered* as well as your parameters to the PostgreSQL server where the parameters are safely substituted into the query with battle-tested parameter substitution code within the server itself.

```
const text = 'INSERT INTO users(name, email) VALUES($1, $2)'
const values = ['brianc', 'brian.m.carlson@gmail.com']

// callback
client.query(text, values, (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
```

```
    console.log(res.rows[0])
  }
})

// promise
client.query(text, values)
  .then(res => console.log(res.rows[0]))
  .catch(e => console.error(e.stack))
```

Query config object

`pool.query` and `client.query` both support taking a config object as an argument instead of taking a string and optional array of parameters. The same example above could also be performed like so:

```
const query = {
  text: 'INSERT INTO users(name, email) VALUES($1, $2)',
  values: ['brianc', 'brian.m.carlson@gmail.com'],
}

// callback
client.query(query, (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
    console.log(res.rows[0])
  }
})

// promise
client.query(query)
  .then(res => console.log(res.rows[0]))
  .catch(e => console.error(e.stack))
```

The query config object allows for a few more advanced scenarios:

Prepared statements

PostgreSQL has the concept of a prepared statement. node-postgres supports this by supplying a `name` parameter to the query config object. If you supply a `name` parameter the query execution plan will be cached on the PostgreSQL server on a **per connection basis**. This means if you use two different connections each will have to parse & plan the query once. node-postgres handles this transparently for you: a client only requests a query to be parsed the first time that particular client has seen that query name:

```
const query = {
  // give the query a unique name
  name: 'fetch-user',
  text: 'SELECT * FROM user WHERE id = $1',
  values: [1]
}

// callback
```

```
client.query(query, (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
    console.log(res.rows[0])
  }
})

// promise
client.query(query)
  .then(res => console.log(res.rows[0]))
  .catch(e => console.error(e.stack))
```

In the above example the first time the client sees a query with the name `'fetch-user'` it will send a 'parse' request to the PostgreSQL server & execute the query as normal. The second time, it will skip the 'parse' request and send the *name* of the query to the PostgreSQL server.

Be careful not to fall into the trap of premature optimization. Most of your queries will likely not benefit much, if at all, from using prepared statements. This is a somewhat "power user" feature of PostgreSQL that is best used when you know how to use it - namely with very complex queries with lots of joins and advanced operations like union and switch statements. I rarely use this feature in my own apps unless writing complex aggregate queries for reports and I know the reports are going to be executed very frequently.

Row mode

By default node-postgres reads rows and collects them into JavaScript objects with the keys matching the column names and the values matching the corresponding row value for each column. If you do not need or do not want this behavior you can pass `rowMode: 'array'` to a query object. This will inform the result parser to bypass collecting rows into a JavaScript object, and instead will return eacho row as an array of values.

```
const query = {
  text: 'SELECT $1::text as first_name, select $2::text as last_name',
  values: ['Brian', 'Carlson'],
  rowMode: 'array',
};

// callback
client.query(query, (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
    console.log(res.fields.map(f => field.name)) // ['first_name', 'last_name']
    console.log(res.rows[0]) // ['Brian', 'Carlson']
  }
})

// promise
client.query(query)
  .then(res => {
    console.log(res.fields.map(f => field.name)) // ['first_name', 'last_name']
    console.log(res.rows[0]) // ['Brian', 'Carlson']
```

```
  })
  .catch(e => console.error(e.stack))
```

Types

You can pass in a custom set of type parsers to use when parsing the results of a particular query. The `types` property must conform to the Types API. Here is an example in which every value is returned as a string:

```
const query = {
  text: 'SELECT * from some_table',
  types: {
    getTypeParser: () => (val) => val
  }
}
```

Pooling

If you're working on a web application or other software which makes frequent queries you'll want to use a connection pool.

The easiest and by far most common way to use node-postgres is through a connection pool.

Why?

- Connecting a new client to the PostgreSQL server requires a handshake which can take 20-30 milliseconds. During this time passwords are negotiated, SSL may be established, and configuration information is shared with the client & server. Incurring this cost *every time* we want to execute a query would substantially slow down our application.
- The PostgreSQL server can only handle a limited number of clients at a time. Depending on the available memory of your PostgreSQL server you may even crash the server if you connect an unbounded number of clients. *note: I have crashed a large production PostgreSQL server instance in RDS by opening new clients and never disconnecting them in a python application long ago. It was not fun.*
- PostgreSQL can only process one query at a time on a single connected client in a first-in first-out manner. If your multi-tenant web application is using only a single connected client all queries among all simultaneous requests will be pipelined and executed serially, one after the other. No good!

Good news

node-postgres ships with built-in connection pooling via the pg-pool module.

Examples

The client pool allows you to have a reusable pool of clients you can check out, use, and return. You generally want a limited number of these in your application and usually just 1. Creating an unbounded number of pools defeats the purpose of pooling at all.

Checkout, use, and return

```
const { Pool } = require('pg')

const pool = new Pool()

// the pool with emit an error on behalf of any idle clients
// it contains if a backend error or network partition happens
pool.on('error', (err, client) => {
  console.error('Unexpected error on idle client', err)
  process.exit(-1)
})

// callback - checkout a client
pool.connect((err, client, done) => {
  if (err) throw err
  client.query('SELECT * FROM users WHERE id = $1', [1], (err, res) => {
    done()

    if (err) {
      console.log(err.stack)
    } else {
      console.log(res.rows[0])
    }
  })
})

// promise - checkout a client
pool.connect()
  .then(client => {
    return client.query('SELECT * FROM users WHERE id = $1', [1])
      .then(res => {
        client.release()
        console.log(res.rows[0])
      })
      .catch(e => {
        client.release()
        console.log(err.stack)
      })
  })

// async/await - check out a client
(async () => {
  const client = await pool.connect()
  try {
    const res = await pool.query('SELECT * FROM users WHERE id = $1', [1])
    console.log(res.rows[0])
  } finally {
    client.release()
  }
})().catch(e => console.log(e.stack))
```

You must **always** return the client to the pool if you successfully check it out, regardless of whether or not there was an error with the queries you ran on the client. If you don't check in the client your application will leak them and eventually your pool will be empty forever and all future requests to check out a client from the pool will wait forever.

Single query

If you don't need a transaction or you just need to run a single query, the pool has a convenience method to run a query on any available client in the pool. This is the preferred way to query with node-postgres if you can as it removes the risk of leaking a client.

```
const { Pool } = require('pg')

const pool = new Pool()

pool.query('SELECT * FROM users WHERE id = $1', [1], (err, res) => {
  if (err) {
    throw err
  }

  console.log('user:', res.rows[0])
})
```

node-postgres also has built-in support for promises throughout all of its async APIs.

```
const { Pool } = require('pg')

const pool = new Pool()

pool.query('SELECT * FROM users WHERE id = $1', [1])
  .then(res => console.log('user:', res.rows[0]))
  .catch(e => setImmediate(() => { throw e }))
```

Promises allow us to use `async`/`await` in node v8.0 and above (or earlier if you're using babel).

```
const { Pool } = require('pg')
const pool = new Pool()

(async () => {

  const { rows } = await pool.query('SELECT * FROM users WHERE id = $1', [1])
  console.log('user:', rows[0])

})().catch(e => setImmediate(() => { throw e }))
```

Shutdown

To shut down a pool call `pool.end()` on the pool. This will wait for all checked-out clients to be returned and then shut down all the clients and the pool timers.

```
const { Pool } = require('pg')
const pool = new Pool()

(async () => {
```

```
  console.log('starting async query')
  const result = await pool.query('SELECT NOW()')
  console.log('async query finished)

  console.log('starting callback query')
  pool.query('SELECT NOW()', (err, res) => {
    console.log('callback query finished')
  })

  console.log('calling end')
  await pool.end()
  console.log('pool has drained')
})()
```

The output of the above will be:

```
starting async query
async query finished
starting callback query
calling end
callback query finished
pool has drained
```

The pool will return errors when attempting to check out a client after you've called `pool.end()` on the pool.

Transactions

To execute a transaction with node-postgres you simply execute `BEGIN / COMMIT / ROLLBACK` queries yourself through a client. Because node-postgres strives to be low level an un-opinionated it doesn't provide any higher level abstractions specifically around transactions.

You **must** use the *same* client instance for all statements within a transaction. PostgreSQL isolates a transaction to individual clients. This means if you initialize or use transactions with the `pool.query` method you **will** have problems. Do not use transactions with `pool.query`.

Examples

A pooled client with callbacks

```
const { Pool } = require('client')
const pool = new Pool()

pool.connect((err, client, done) => {

  const shouldAbort = (err) => {
    if (err) {
      console.error('Error in transaction', err.stack)
      client.query('ROLLBACK', (err) => {
        if (err) {
          console.error('Error rolling back client', err.stack)
```

```
        }
        // release the client back to the pool
        done()
      })
    }
    return !!err
  }

  client.query('BEGIN', (err) => {
    if (shouldAbort(err)) return
    client.query('INSERT INTO users(name) VALUES($1) RETURNING id', ['brianc'],
(err, res) => {
      if (shouldAbort(err)) return

      const insertPhotoText = 'INSERT INTO photos(user_id, photo_url) VALUES ($1,
$2)'
      const insertPhotoValues = [res.rows[0].id, 's3.bucket.foo']
      client.query(insertPhotoText, insertPhotoValues, (err, res) => {
        if (shouldAbort(err)) return

        client.query('COMMIT', (err) => {
          if (err) {
            console.error('Error committing transaction', err.stack)
          }
          done()
        })
      })
    })
  })
})
```

I omitted any additional libraries from the example for clarity, but if you're using callbacks you'd typically be using a flow control library like async.

A pooled client with async/await

Things are considerably more straightforward if you're using async/await:

```
const { Pool } = require('client')
const pool = new Pool()

(async () => {
  // note: we don't try/catch this because if connecting throws an exception
  // we don't need to dispose of the client (it will be undefined)
  const client = await pool.connect()

  try {
    await client.query('BEGIN')
    const { rows } = await client.query('INSERT INTO users(name) VALUES($1)
RETURNING id', ['brianc'])

    const insertPhotoText = 'INSERT INTO photos(user_id, photo_url) VALUES ($1,
$2)'
    const insertPhotoValues = [res.rows[0].id, 's3.bucket.foo']
    await client.query(insertPhotoText, insertPhotoValues)
    await client.query('COMMIT')
  } catch (e) {
    await client.query('ROLLBACK')
```

```
    throw e
  } finally {
    client.release()
  }
})().catch(e => console.error(e.stack))
```

Types

PostgreSQL has a rich system of supported data types. node-postgres does its best to support the most common data types out of the box and supplies an extensible type parser to allow for custom type serialization and parsing.

strings by default

node-postgres will convert a database type to a JavaScript string if it doesn't have a registered type parser for the database type. Furthermore, you can send any type to the PostgreSQL server as a string and node-postgres will pass it through without modifying it in any way. To circumvent the type parsing completely do something like the following.

```
const queryText = 'select int_col::string, date_col::string, json_col::string from my_table'
const result = await client.query(queryText)

console.log(result.rows[0]) // will contain the unparsed string value of each column
```

type parsing examples

uuid + json / jsonb

There is no data type in JavaScript for a uuid/guid so node-postgres converts a uuid to a string. JavaScript has great support for JSON and node-postgres converts json/jsonb objects directly into their JavaScript object via `JSON.parse`. Likewise sending an object to the PostgreSQL server via a query from node-postgres, node-posgres will call `JSON.stringify` on your outbound value, automatically converting it to json for the server.

```
const createTableText = `
CREATE EXTENSION IF NOT EXISTS "pgcrypto";

CREATE TEMP TABLE IF NOT EXISTS users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  data JSONB
);
`

// create our temp table
await client.query(createTableText)

const newUser = { email: 'brian.m.carlson@gmail.com' }
// create a new user
```

```
await client.query('INSERT INTO users(data) VALUES($1)', [newUser])

const { rows } = await client.query('SELECT * FROM users')

console.log(rows)
/*
output:
[{
  id: 'd70195fd-608e-42dc-b0f5-eee975a621e9',
  data: { email: 'brian.m.carlson@gmail.com' }
}]
*/
```

date / timestamp / timestamptz

node-postgres will convert instances of JavaScript date objects into the expected input value for your PostgreSQL server. Likewise, when reading a `date`, `timestamp`, or `timestamptz` column value back into JavaScript, node-postgres will parse the value into an instance of a JavaScript `Date` object.

```
const createTableText = `
CREATE TEMP TABLE dates(
  date_col DATE,
  timestamp_col TIMESTAMP,
  timestamptz_col TIMESTAMPTZ,
);
`
// create our temp table
await client.query(createTableText)

// insert the current time into it
const now = new Date()
const insertText = 'INSERT INTO dates(date_col, timestamp_col, timestamtz_col'
await client.query(insertText, [now, now, now])

// read the row back out
const result = await client.query('SELECT * FROM dates')

console.log(result.rows)
// {
// date_col: 2017-05-29T05:00:00.000Z,
// timestamp_col: 2017-05-29T23:18:13.263Z,
// timestamptz_col: 2017-05-29T23:18:13.263Z
// }
```

psql output:

```
bmc=# select * from dates;
  date_col  |      timestamp_col      |      timestamptz_col
------------+-------------------------+---------------------------
 2017-05-29 | 2017-05-29 18:18:13.263 | 2017-05-29 18:18:13.263-05
(1 row)
```

node-postgres converts `DATE` and `TIMESTAMP` columns into the **local** time of the node process set at `process.env.TZ`.

note: I generally use `TIMESTAMPTZ` when storing dates; otherwise, inserting a time from a process in

one timezone and reading it out in a process in another timezone can cause unexpected differences in the time.

Although PostgreSQL supports microseconds in dates, JavaScript only supports dates to the millisecond precision. Keep this in mind when you send dates to and from PostgreSQL from node: your milliseconds will be truncated when converting to a JavaScript date object even if they exist in the database. If you need to preserve them, I recommend using a custom type parser.

TLS/SSL

node-postgres supports TLL/SSL connections to your PostgreSQL server as long as the server is configured to support it. When instantiating a pool or a query you can provide an `ssl` property on the config object and it will be passed to the constructor for the node TLSSocket.

Self-signed cert

Here's an example of a configuration you can use to connect a client or a pool to a PostgreSQL server.

```
const config = {
  database : 'database-name',
  host     : "host-or-ip",
  // this object will be passed to the TLSSocket constructor
  ssl : {
    rejectUnauthorized : false,
    ca   : fs.readFileSync("/path/to/server-
certificates/maybe/root.crt").toString(),
    key  : fs.readFileSync("/path/to/client-key/maybe/postgresql.key").toString(),
    cert : fs.readFileSync("/path/to/client-
certificates/maybe/postgresql.crt").toString(),
  }
}

import { Client, Pool } from 'pg'

const client = new Client(config)
client.connect((err) => {
  if (err) {
    console.error('error connecting', err.stack)
  } else {
    console.log('connected')
    client.end()
  }
})


const pool = new Pool(config)
pool.connect()
  .then(client => {
    console.log('connected')
    client.release()
  })
  .catch(err => console.error('error connecting', err.stack))
  .then(() => pool.end())
```

Suggested Project Structure

Whenever I am writing a project & using node-postgres I like to create a file within it and make all interactions with the database go through this file. This serves a few purposes:

- Allows my project to adjust to any changes to the node-postgres API without having to trace down all the places I directly use node-postgres in my application.
- Allows me to have a single place to put logging and diagnostics around my database.
- Allows me to make custom extensions to my database access code & share it throughout the project.
- Allows a single place to bootstrap & configure the database.

example

note: I am using callbacks in this example to introduce as few concepts as possible at a time, but the same is doable with promises or async/await

The location doesn't really matter - I've found it usually ends up being somewhat app specific and in line with whatever folder structure conventions you're using. For this example I'll use an express app structured like so:

```
- app.js
- index.js
- routes/
  - index.js
  - photos.js
  - user.js
- db/
  - index.js <--- this is where I put data access code
```

Typically I'll start out my `db/index.js` file like so:

```
const { Pool } = require('pg')

const pool = new Pool()

module.exports = {
  query: (text, params, callback) => {
    return pool.query(text, params, callback)
  }
}
```

That's it. But now everywhere else in my application instead of requiring `pg` directly, I'll require this file. Here's an example of a route within `routes/user.js`:

```
// notice here I'm requiring my database adapter file
// and not requiring node-postgres directly
const db = require('../db')

app.get('/:id', (req, res, next) => {
  db.query('SELECT * FROM users WHERE id = $1', [id], (err, res) => {
```

```
    if (err) {
      return next(err)
    }
    res.send(res.rows[0])
  })
})

// ... many other routes in this file
```

Imagine we have lots of routes scattered throughout many files under our `routes/` directory. We now want to go back and log every single query that's executed, how long it took, and the number of rows it returned. If we had required node-postgres directly in every route file we'd have to go edit every single route - that would take forever & be really error prone! But thankfully we put our data access into `db/index.js`. Let's go add some logging:

```
const { Pool } = require('pg')

const pool = new Pool()

module.exports = {
  query: (text, params, callback) => {
    const start = Date.now()
    return pool.query(text, params, (err, res) => {
      const duration = Date.now() - start
      console.log('executed query', { text, duration, rows: res.rowCount })
      callback(err, res)
    })
  }
}
```

That was pretty quick! And now all of our queries everywhere in our application are being logged.

Now what if we need to check out a client from the pool to run sever queries in a row in a transaction? We can add another method to our `db/index.js` file when we need to do this:

```
const { Pool } = require('pg')

const pool = new Pool()

module.exports = {
  query: (text, params, callback) {
    const start = Date.now()
    return pool.query(text, params, (err, res) => {
      const duration = Date.now() - start
      console.log('executed query', { text, duration, rows: res.rowCount })
      callback(err, res)
    })
  },
  getClient: (callback) {
    pool.connect((err, client, done) => {
      callback(err, client, done)
    })
  }
}
```

Okay. Great - the simplesting thing that could possibly work. It seems like one of our routes that checks out a client to run a transaction is forgetting to call `done` in some situation! Oh no! We are leaking a client & have hundreds of these routes to go audit. Good thing we have all our client access going through this single file. Lets add some deeper diagnostic information here to help us track down where the client leak is happening.

```
const { Pool } = require('pg')

const pool = new Pool()

module.exports = {
  query: (text, params, callback) {
    const start = Date.now()
    return pool.query(text, params, (err, res) => {
      const duration = Date.now() - start
      console.log('executed query', { text, duration, rows: res.rowCount })
      callback(err, res)
    })
  },
  getClient: (callback) {
    pool.connect((err, client, done) => {
      const query = client.query.bind(client)

      // monkey patch the query method to keep track of the last query executed
      client.query = () => {
        client.lastQuery = arguments
        client.query.apply(client, arguments)
      }

      // set a timeout of 5 seconds, after which we will log this client's last
query
      const timeout = setTimeout(() => {
        console.error('A client has been checked out for more than 5 seconds!')
        console.error(`The last executed query on this client was: $
{client.lastQuery}`)
      }, 5000)

      const release = (err) => {
        // call the actual 'done' method, returning this client to the pool
        done(err)

        // clear our timeout
        clearTimeout(timeout)

        // set the query method back to its old un-monkey-patched version
        client.query = query
      }

      callback(err, client, done)
    })
  }
}
```

That should hopefully give us enough diagnostic information to track down any leaks.

Express with async/await

My preferred way to use node-postgres (and all async code in node.js) is with `async/await`. I find it makes reasoning about control-flow easier and allows me to write more concice and maintainable code.

This is how I typically structure express web-applications with node-postgres to use `async/await`:

```
- app.js
- index.js
- routes/
  - index.js
  - photos.js
  - user.js
- db/
  - index.js <--- this is where I put data access code
```

That's the same structure I used in the project structure example.

My `db/index.js` file usually starts out like this:

```
const { Pool } = require('pg')

module.exports = {
  query: (text, params) => pool.query(text, params)
}
```

Then I will install express-promise-router and use it to define my routes. Here is my `routes/user.js` file:

```
const Router = require('express-promise-router')

const db = require('../db')

// create a new express-promise-router
// this has the same API as the normal express router except
// it allows you to use async functions as route handlers
const router = new Router()

// export our router to be mounted by the parent application
module.exports = router

router.get('/:id', async (req, res) => {
  const { id } = req.params
  const { rows } = await db.query('SELECT * FROM users WHERE id = $1', [id])
  res.send(rows[0])
})
```

Then in my `routes/index.js` file I'll have something like this which mounts each individual router into the main application:

```
// ./routes/index.js
const users = require('./user')
const photos = require('./photos')

module.exports = (app) => {
```

```
  app.use('/users', users)
  app.use('/photos', photos)
  // etc..
}
```

And finally in my `app.js` file where I bootstrap express I will have my `routes/index.js` file mount all my routes. The routes know they're using async functions but because of express-promise-router the main express app doesn't know and doesn't care!

```
// ./app.js
const express = require('express')
const mountRoutes = require('./routes')

const app = express()
mountRoutes(app)

// ... more express setup stuff can follow
```

Now you've got `async/await`, node-postgres, and express all working together!

Upgrading to 7.0

node-postgres at 7.0 introduces somewhat significant breaking changes to the public API.

node version support

Starting with `pg@7.0` the earliest version of node supported will be `node@4.x LTS`. Support for `node@0.12.x` and `node@.10.x` is dropped, and the module wont work as it relies on new es6 features not available in older versions of node.

pg singleton

In the past there was a singleton pool manager attached to the root `pg` object in the package. This singleton could be used to provision connection pools automatically by calling `pg.connect`. This API caused a lot of confusion for users. It also introduced a opaque module-managed singleton which was difficult to reason about, debug, error-prone, and inflexible. Starting in pg@6.0 the methods' documentation was removed, and starting in pg@6.3 the methods were deprecated with a warning message.

If your application still relies on these they will be *gone* in `pg@7.0`. In order to migrate you can do the following:

```
// old way, deprecated in 6.3.0:

// connection using global singleton
pg.connect(function(err, client, done) {
  client.query(/* etc, etc */)
  done()
})
```

```
// singleton pool shutdown
pg.end()

// ------------------

// new way, available since 6.0.0:

// create a pool
var pool = new pg.Pool()

// connection using created pool
pool.connect(function(err, client, done) {
  client.query(/* etc, etc */)
  done()
})

// pool shutdown
pool.end()
```

node-postgres ships with a built-in pool object provided by pg-pool which is already used internally by the `pg.connect` and `pg.end` methods. Migrating to a user-managed pool (or set of pools) allows you to more directly control their set up their life-cycle.

client.query(...).on

Before `pg@7.0` the `client.query` method would *always* return an instance of a query. The query instance was an event emitter, accepted a callback, and was also a promise. A few problems...

- too many flow control options on a single object was confusing
- event emitter `.on('error')` does not mix well with promise `.catch`
- the `row` event was a common source of errors: it looks like a stream but has no support for back-pressure, misleading users into trying to pipe results or handling them in the event emitter for a desired performance gain.
- error handling with a `.done` and `.error` emitter pair for every query is cumbersome and returning the emitter from `client.query` indicated this sort of pattern may be encouraged: it is not.

Starting with `pg@7.0` the return value `client.query` will be dependent on what you pass to the method: I think this aligns more with how most node libraries handle the callback/promise combo, and I hope it will make the "just works" :tm: feeling better while reducing surface area and surprises around event emitter / callback combos.

client.query with a callback

```
const query = client.query('SELECT NOW()', (err, res) => {
  /* etc, etc */
})
assert(query === undefined) // true
```

If you pass a callback to the method `client.query` will return `undefined`. This limits flow control to the callback which is in-line with almost all of node's core APIs.

client.query without a callback

```
const query = client.query('SELECT NOW()')
assert(query instanceof Promise) // true
assert(query.on === undefined) // true
query.then((res) => /* etc, etc */)
```

If you do **not** pass a callback `client.query` will return an instance of a `Promise`. This will **not** be a query instance and will not be an event emitter. This is in line with how most promise-based APIs work in node.

client.query(Submittable)

`client.query` has always accepted any object that has a `.submit` method on it. In this scenario the client calls `.submit` on the object, delegating execution responsibility to it. In this situation the client also **returns the instance it was passed**. This is how pg-cursor and pg-query-stream work. So, if you need the event emitter functionality on your queries for some reason, it is still possible because `Query` is an instance of `Submittable`:

```
const { Client, Query } = require('pg')
const query = client.query(new Query('SELECT NOW()'))
query.on('row', (row) => {

})
query.on('end', (res) => {

})
query.on('error', (res) => {

})
```

`Query` is considered a public, documented part of the API of node-postgres and this form will be supported indefinitely.

*note: I have been building apps with node-postgres for almost 7 years. In that time I have never used the event emitter API as the primary way to execute queries. I used to use callbacks and now I use async/await. If you need to stream results I highly recommend you use pg-cursor or pg-query-stream and **not** the query object as an event emitter.*