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 availabl 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 client.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.