Queries

The api for executing queries supports both callbacks and promises. I'll provide an example for both *styles* her 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) RETURNING *'
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])
    // { name: 'brianc', email: 'brian.m.carlson@gmail.com' }
  }
})
```

```
// promise
client.query(text, values)
  .then(res => {
    console.log(res.rows[0])
    // { name: 'brianc', email: 'brian.m.carlson@gmail.com' }
  })
  .catch(e => console.error(e.stack))

// async/await
try {
  const res = await pool.query(text, values)
  console.log(res.rows[0])
  // { name: 'brianc', email: 'brian.m.carlson@gmail.com' }
} catch(err) {
  console.log(err.stack)
}
```

Query config object

`pool.query` and `client.query` both support taking a config object as an argument instead of taking a strin 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 cache on the PostgreSQL server on a **per connection basis**. This means if you use two different connections each wi have to parse & plan the query once. node-postgres handles this transparently for you: a client only requests 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 parse 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
  }
}
```