TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Obtención de información desde una base de datos.
- Consultas asíncronas.
- Consultas parametrizadas
- ¿Qué es SQL Injection?
- Objeto JSON como argumento de consultas.
- Declaraciones preparadas.
- Modo fila (row mode)
- Tipos (types).
- Tipos de datos.

OBTENCIÓN DE INFORMACIÓN DESDE UNA BASE DE DATOS

La API para ejecutar consultas admite devoluciones de llamadas (callbacks) y promesas (promises). Tanto el método de client.query, como el del pool.query, admiten la misma API. De hecho, pool.query delega directamente a client.query internamente.

El método `query()` como parámetro, recibe una cadena o un string de la consulta que se quiere realizar. Éste devuelve una promesa que, de ser exitosa, muestra los valores de los datos de la consulta en los rows de la respuesta como propiedades.

Client.query()

Pasar texto de consulta, parámetros de consulta opcionales, y una devolución de llamada a client.query, da como resultado una firma de tipo:

```
client.query(
  text: string,
  values?: Array<mixed>,
  callback: (err: Error, result: Result) => void
) => void
```

Si la consulta no tiene parámetros, no es necesario incluirlos en el método de consulta, o consulta en texto plano:

```
// 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))
```

Una consulta sencilla, previamente conectados a la base de datos, sería:

```
client.query('SELECT * FROM Personas')
    .then(res => {
        console.log(res.rows)
        client.end() // Cerrando la conexión
    })
```

Pool.query()

Generalmente, se necesita ejecutar una sola consulta en la base de datos. Por conveniencia, en el grupo de pool se cuenta con un método para ejecutar una consulta en el primer cliente inactivo disponible, y devolver su resultado.

```
pool.query(callback: (err?: Error, result: pg.Result)) => void
```

Por ejemplo:

```
const { Pool } = require('pg')
const pool = new Pool()
pool.query('SELECT $1::text as name', ['Pedro'], (err, result) => {
  if (err) {
    return console.error('Error en la ejecución de la consulta',
err.stack)
```

```
  }
  console.log(result.rows[0].name) // Pedro
})
```

Compatibilidad con promesas:

```
const { Pool } = require('pg')
const pool = new Pool()
pool
  .query('SELECT $1::text as name', ['Pedro'])
  .then(res => console.log(res.rows[0].name)) // Pedro
  .catch(err => console.error('Error en la ejecución de la consulta',
err.stack))
```

CONSULTAS ASÍNCRONAS

Consultas asíncronas con ASYNC/AWAIT

async y await pueden construir sobre promesas y generadores para expresar acciones en línea asíncronas. Esto hace que el código asíncrono, o de devolución de llamada, sea mucho más fácil de mantener. Las funciones con la palabra clave async devuelven una Promise, y son llamadas con esa sintaxis.

Dentro de una función async, la palabra clave await puede aplicarse a cualquier Promise, y hará que todo el cuerpo de dicha función después de la await se ejecute después de que se resuelva la promesa.

Ejemplo:

```
(async () => {
    try {
        const client = await pool.connect();
        const res = await client.query(query);

        for (let row of res.rows) {
            console.log(row);
        }
    } catch (err) {
        console.error(err);
    }
})();
```

CONSULTAS ASÍNCRONAS CON CALLBACKS

Las funciones callback son las mismas de siempre en JavaScript. Estas no disponen de una sintaxis especial, ya que simplemente son pasadas como argumento a otra función.

La función que recibe el callback como argumento, tiene como nombre función de mayor orden. Cualquier función puede ser usada como callback, ya que basta con pasársela a otra como parámetro.

Éstas no son asíncronas por naturaleza, pero suelen ser usadas con dicho propósito. Por ejemplo: cuando son pasadas como argumento a los diferentes eventos que aceptan las APIs de un navegador, permitiendo que JavaScript pueda interactuar con el DOM de una página, o con el sistema.

```
pool.connect()
    .then((client) => {
        client.query(query)
            .then(res => {
                for (let row of res.rows) {
                    console.log(row);
                }
            })
            .catch(err => {
                console.error(err);
            });
    })
    .catch(err => {
        console.error(err);
    });
```

CONSULTAS PARAMETRIZADAS

En las consultas se pasan parámetros, pues generalmente se desea evitar la concatenación de cadenas de parámetros en el texto de la consulta directamente. Esto puede dar lugar a vulnerabilidades de inyección de sql. node-postgres admite las consultas parametrizadas, pasando su texto sin modificar, así como sus parámetros al servidor PostgreSQL, donde éstos se sustituyen de forma segura en la consulta con código de sustitución de parámetros, que es probado en batalla dentro del propio servidor.

Queries con texto parametrizado:

```
const text = 'INSERT INTO users(name, email) VALUES($1, $2) RETURNING
*'
const values = ['Pedro', 'pedro.perez@gmail.com']
// callback
client.query(text, values, (err, res) => {
  if (err) {
    console.log(err.stack)
  } else {
    console.log(res.rows[0])
    // { name: 'Pedro', email: 'pedro.perez@gmail.com' }
  }
})
// promise
client
  .query(text, values)
  .then(res => {
    console.log(res.rows[0])
    // { name: 'Pedro', email: 'pedro.perez@gmail.com' }
  })
  .catch(e => console.error(e.stack))

// async/await
try {
  const res = await client.query(text, values)
  console.log(res.rows[0])
  // { name: 'Pedro', email: 'pedro.perez@gmail.com' }
} catch (err) {
  console.log(err.stack)
```

En el ejemplo se usa una consulta parametrizada, en una instrucción SELECT simple:

```
const text = 'INSERT INTO users(name, email) VALUES($1, $2) RETURNING
*'
```

Esta es la consulta SELECT. El $1 y $2 es un marcador de posición, que luego se reemplaza con un valor de forma segura:

```
const values = ['Pedro', 'pedro.perez@gmail.com']
```

Estos son los valores que se insertarán en la consulta parametrizada:

```
client.query(text, values ….
```

Los valores se pasan al método de consulta como segundo parámetro.

VALIDACIÓN DE PARÁMETROS DE CONSULTA

Aunque generalmente el análisis de consultas es manejado por un servidor, para algunas aplicaciones esto puede ser un problema al validar consultas en el lado del cliente.

Para hacer esto, y validar de manera efectiva los parámetros de consulta, podemos usar un middleware como "express-universal-query-validator", que se instala a través de npm. Esto valida cada parámetro de consulta a través de `decodeURIComponent`, y proporciona una devolución de llamada para tomar medidas en el servidor cuando se detectan parámetros no válidos. El comportamiento predeterminado cuando no se proporciona devolución de llamada es iniciar sesión, y luego, redirigir a la misma ruta con los parámetros no analizables eliminados.

¿QUÉ ES SQL INJECTION?

O Inyección SQL, es una técnica de vulnerabilidad que permite al atacante enviar instrucciones SQL de forma maliciosa y malintencionada dentro del código SQL, permitiendo la manipulación de bases de datos. De esta manera todos los datos almacenados estarían en peligro. Este ataque tiene como fin modificar el comportamiento de nuestras consultas, a través de parámetros no deseados, pudiendo así: falsificar identidades, obtener y divulgar información de la base de datos (contraseñas, correos, información relevante, entre otros), borrar la base de datos, cambiar el nombre a las tablas, anular transacciones, el atacante o pirata informático puede convertirse en administrador de la base de datos y del sistema.

Esto generalmente ocurre por la mala filtración de las variables en un programa que tiene o crea SQL, como en el caso donde solicitas a un usuario entradas de cualquier tipo, y éstas no se encuentran validadas (ejemplo: su nombre y contraseña), pero a cambio de esta información, el atacante envía una sentencia SQL invasora que se ejecutará en la base de datos.

Ejemplo de ataque SQL Injection

Existen muchas formas de ataque. Uno de los más frecuentes es donde se valida una consulta como verdadera. Por ejemplo:

```
1   SELECT * FROM usuarios WHERE username = 'atacante' AND password =
2 'mi_clave' OR 1=1;
```

Allí se observa que esta consulta está formada por el condicional OR, el cual devolverá verdadero al cumplirse, al menos, una de las dos expresiones, por lo que siempre será verdadero ya que 1 = 1. Cuando esto se ejecuta, la base de datos arroja el total de registros en la tabla, aunque el nombre de usuario y contraseña sean incorrectos, pues la condición OR 1=1 siempre se cumple.

OBJETO JSON COMO ARGUMENTO DE CONSULTAS

Tanto pool.query y client.query admiten tomar un objeto de configuración como argumento, en lugar de una cadena y una matriz opcional de parámetros. Por ejemplo:

```
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))
```

DECLARACIONES PREPARADAS

PostgreSQL tiene el concepto de una declaración preparada. node-postgres admite esto, proporcionando un parámetro de nombre al objeto de configuración en la consulta. Si éste se entrega, el plan de ejecución de la consulta se almacenará en el caché del servidor de PostgreSQL por conexión. Esto significa que, si usa dos conexiones diferentes, cada una tendrá que analizar y planificar la consulta una vez. node-postgres maneja esto de manera transparente. Un cliente solo solicita que se analice una consulta la primera vez en particular que la ha visto. Por ejemplo:

```
const query = {
  //  dar a la consulta un nombre único
```

```
  name: 'search-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))
```

En el anterior código de ejemplo, la primera vez que el cliente ve una consulta con el nombre 'search-user', enviará una solicitud de 'análisis' al servidor PostgreSQL, y ejecutará la consulta normalmente. Mientras que la segunda vez, omitirá la solicitud de 'análisis', y enviará el nombre de la consulta al servidor PostgreSQL.

MODO FILA (ROW MODE)

De forma predeterminada, node-postgres lee las filas, y las recopila en objetos de JavaScript con las claves que coinciden con los nombres de las columnas, y los valores que coinciden con el valor de la fila correspondiente para cada columna. Si no necesita, o no desea este comportamiento, puede pasar rowMode: 'Array' a un objeto de consulta. Esto informará al analizador de resultados que omita la recopilación de filas en un objeto de JavaScript y que, en su lugar, devuelva cada fila como una matriz de valores.

```
const query = {
  text: 'SELECT $1::text as first_name, $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(field => field.name)) //
['first_name', 'last_name']
    console.log(res.rows[0]) // ['Brian', 'Carlson']
  }
})
// promise
client
  .query(query)
  .then(res => {
    console.log(res.fields.map(field => field.name)) //
['first_name', 'last_name']
    console.log(res.rows[0]) // ['Brian', 'Carlson']
  })
  .catch(e => console.error(e.stack))
```

TIPOS (TYPES)

Puede pasar un conjunto personalizado de analizadores de tipo, para ser usados al analizar los resultados de una consulta en particular. La propiedad de tipos debe cumplir con la API de tipos. Aquí hay un ejemplo, en el que cada valor se devuelve como una cadena:

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

TIPOS DE DATOS

PostgreSQL tiene un amplio sistema de tipos de datos admitidos. node-postgres hace todo lo posible para admitir los tipos de datos más comunes de forma inmediata, y proporciona un analizador de tipos extensible para permitir la serialización, y el análisis de tipos personalizados.

Cadenas por defecto

node-postgres convertirá un tipo de base de datos en una cadena de JavaScript, si éste no tiene un analizador de tipo registrado para el tipo de base de datos. Además, puede enviar cualquier tipo al

servidor PostgreSQL como una cadena, y lo pasará sin modificarlo de ninguna manera. Para eludir el tipo de análisis por completo, se puede hacer algo como:

```
const queryText = 'SELECT int_col::text, date_col::text,
json_col::text FROM my_table'
const result = await client.query(queryText)
console.log(result.rows[0]) // will contain the unparsed string value
of each column
```

uuid + json / jsonb

No hay ningún tipo de datos en JavaScript para un uuid/guid, por lo que node-postgres convierte un uuid en una cadena. JavaScript tiene un gran soporte para JSON, y node-postgres convierte los objetos json/jsonb directamente en su objeto JavaScript, a través de JSON.parse. De la misma forma, al enviar un objeto al servidor PostgreSQL a través de una consulta de node-postgres, éste último llamará a JSON.stringify en su valor de salida, convirtiéndolo automáticamente a json para el servidor.

```
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
);
`
// Crea una tabla temporal
await client.query(createTableText)
const newUser = { email: 'brian.m.carlson@gmail.com' }
// Crea un nuevo usuario
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' }
}]
*/
```