

Implementación y gestión de una base de datos (Parte II)

Pooling

Competencias

- Reconocer el concepto de Pooling para permitir conexiones múltiples a PostgreSQL utilizando el entorno Node.
- Reconocer el procedimiento de configuración para conexiones múltiples a una base de datos PostgreSQL utilizando el entorno Node.

Introducción

Cuando desarrollamos aplicaciones que persisten información con bases de datos SQL, tenemos que tomar en cuenta la demanda que se pueda tener por parte de nuestros clientes. Si la demanda es alta y estamos utilizando la clase Client del paquete pg, es probable que un cliente A realice una consulta en el momento que el servidor está en pleno procesamiento de la consulta de un cliente B. Entonces, ¿Qué ocurrirá con el cliente A? Recibirá un error puesto que la conexión está ocupada y será rebotada su solicitud.

Para resolver este escenario, en este capítulo, aprenderás a usar y configurar la clase Pool, para ofrecer la gestión de múltiples consultas de manera simultánea, permitiendo la multiconexión de clientes.

Pooling

La forma en la que implementa la conexión a la base de datos la clase Client, no es apropiado para aplicaciones multitarea, donde la misma puede llegar a utilizar en paralelo múltiples operaciones sobre la base de datos.

Para solucionar las falencias que puede tener este método, en informática existe el concepto de Connection Pooling, que hace referencia al manejo eficiente de un conjunto de conexiones abiertas a una base de datos, permitiendo gestionar múltiples consultas de forma simultánea. Este concepto es implementado por el método Pool de la librería pg donde sus principales características son:

- Manejo eficiente del conjunto de conexiones.
- Implementa políticas para agrupar las conexiones.
- Reutilización de conexiones cerrando conexiones inactivas.
- Manejo eficiente de las conexiones a la base de datos, administrando eficientemente la asignación a los hilos de ejecución del procesador.

Mediante los hilos de ejecución se pueden administrar en forma eficiente las tareas del procesador y de sus distintos núcleos. La imagen que verás a continuación corresponde a un diagrama que representa la forma como se conectan varios clientes a través del método Pool a la base de datos.

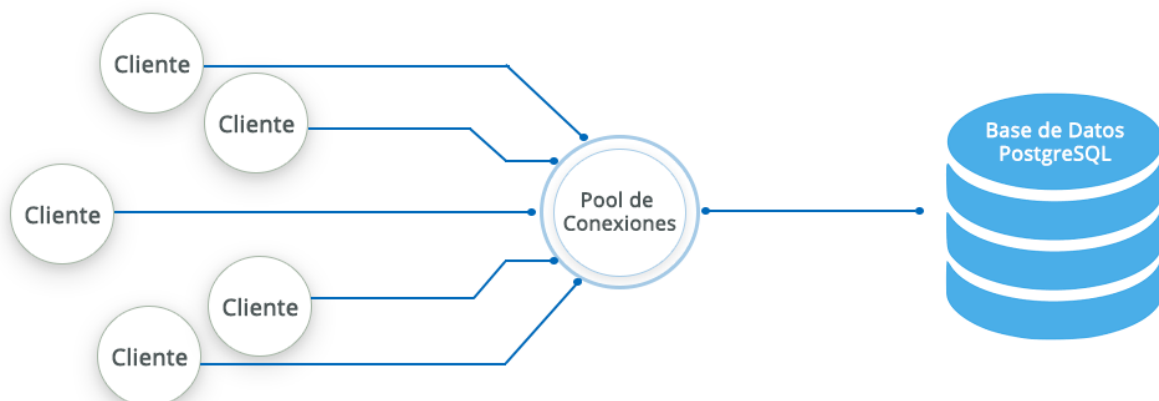


Imagen 1. Diagrama de conexión utilizando el método Pool.

Fuente: Desafío Latam

Este método del módulo pg permite agrupar conexiones de varios clientes a la base de datos.

Configuración

Pool es una clase al igual que Client, recibe un objeto donde son válidos los mismos parámetros utilizados con la clase Cliente, pero la diferencia está en otras propiedades para generar un grupo de conexiones con la base de datos, a continuación te muestra a cuáles propiedades hago referencia:

- **max:** Recibe un valor numérico, que establece la cantidad máxima de clientes conectados que puede contener el grupo. Por defecto, se establece en un valor de 10.
- **min:** Recibe un valor numérico, que establece la cantidad mínima de clientes conectados para iniciar sus consultas. Por defecto, se establece en un valor de 0.
- **idleTimeoutMillis:** Recibe un valor numérico que indica la cantidad de milisegundos que un cliente puede permanecer inactivo antes de que sea desconectado. Por defecto, se establece en un valor de 10.000 que equivalen a 10 segundos, si este valor se establece en 0 deshabilitará la desconexión automática para los clientes inactivos.
- **connectionTimeoutMillis:** Recibe un valor numérico que indica la cantidad de milisegundos, que deben transcurrir antes que se agote el tiempo de espera para conectar a un nuevo cliente. Por defecto, se establece en un valor de 0, que significa que no hay tiempo de espera.
- **ssl:** Recibe un valor booleano, es decir, un true o un false como valor, permite definir si la conexión a la base de datos soporta un protocolo de transporte encriptado.

La función constructora de la clase Pool al igual que la clase Cliente se instancia utilizando la palabra new, como se muestra en el siguiente código donde se escribió unos valores de ejemplo para las propiedades nuevas.

```
const pool = new Pool({
  user: 'NOMBRE_USUARIO',
  host: '127.0.0.1',
  database: 'app_ejemplo',
  password: 'CLAVE_USUARIO',
  port: 5432,
  max: 20,
  min: 2,
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000,
});
```

¿Cómo se interpretaría esta configuración? Leyendo propiedad por propiedad, declarando lo siguiente:

- El nombre de usuario para conectarse a PostgreSQL.
- Servidor donde está alojada la base de datos.
- Base de datos a conectarnos.
- Contraseña correspondiente al usuario indicado.
- Puerto de conexión para el servicio.
- 20 clientes máximos para el Pool.
- 2 clientes como mínimo para que el Pool inicie las consultas.
- 30 segundos como tiempo de espera máximo de inactividad, es decir, que el cliente no realizó otra consulta en este tiempo.
- 2 segundos máximo de espera para recibir la consulta de otro cliente.
- El ssl no está declarado puesto que no aplica para esta lectura, por lo que por defecto su valor es false.

Mi primera consulta con Pool

Competencia

- Construir una aplicación que consulte una base de datos PostgreSQL con la clase Pool del paquete pg.

Introducción

En la lectura anterior aprendiste a realizar consultas con la clase Client del paquete pg, la cual consistía en realizar una query a la base de datos de forma individual, es decir, un cliente abría una conexión, realizaba la consulta y seguidamente se cerraba la conexión.

En este capítulo, aprenderás a realizar consultas SQL aplicando el concepto de Pooling con la clase Pool, por medio de su método query (al igual que la clase Client) pero en esta ocasión trabajando la consulta como un callback, en el que se recibirá un parámetro cliente para realizar finalmente las consultas necesarias y posteriormente “liberarlo” con el método “release” para cerrar la conexión.

Desarrollando con la clase Pool estarás preparándote para mantener aplicaciones escalables que permitan una alta demanda de solicitudes a la base de datos.

Consultando con Pool

Ahora que aprendiste cómo configurar la clase Pool, realicemos una simple consulta para ver la estructura de este método.

Al igual que la clase Client, la clase Pool requiere de una conexión previa a la emisión de las consultas SQL. Para realizar esta conexión debemos ocupar el método "connect()", el cual contiene una función callback con los siguientes parámetros:

- Primer parámetro: El posible error de conexión.
- Segundo parámetro: La instancia "Client", que contiene el método "query()" con el que realizaremos las consultas SQL.
- Tercer parámetro: La función "release", que usaremos para liberar al cliente una vez que terminemos de procesar sus consultas

A continuación se presenta la sintaxis, la cual se puede manejar como callbacks, promesas o async/await.

```
pool.connect((error_conexion, client, release) => {  
  client.query(<Consulta SQL>, (error_query, res) => {  
  
  });  
  pool.end();  
});
```

Hay varias cosas que notar acá, te dejo un listado con el flujo del código para entenderlo mejor:

- Nos conectamos a la base de datos con el método connect.
- Recibimos como parámetro una función callback.
- La función callback tiene 3 parámetros que representan el error de conexión, el cliente y la función para liberar al cliente.
- Se ocupa el parámetro client y se ejecuta su método query para realizar una consulta, la cual funcionará como un callback que recibe el error de la consulta y la respuesta de esta.

- Al terminar de procesar la consulta y ejecutar cualquier lógica se cierra la conexión con el método "end()".

Continuando con la temática de la tienda de ropa H&B, en un plano real estos negocios podrían crecer de manera industrial y disponer incluso de una tienda online a sus clientes, por lo que la base de datos recibiría varias consultas por día, y es probable que varias de estas se generen al mismo tiempo. ¿Entonces qué hacemos? Usamos la clase Pool para gestionar las peticiones y de esta manera lograr que todos los usuarios puedan recibir respuesta a sus solicitudes al momento de interactuar con la base de datos.

Ejercicio guiado: Gestionando una alta demanda

Desarrollar una aplicación que utilice la clase Pool y que al ser ejecutada, devuelva por consola todas las prendas de ropa guardadas en la tabla **ropa** que creamos en la lectura anterior:

- **Paso 1:** Importar el paquete pg y extraer directamente la clase Pool.
- **Paso 2:** Crear una constante de configuración.
- **Paso 3:** Instanciar la clase Pool pasándole a su constructor el objeto de configuración y almacenando su instancia en una constante.
- **Paso 4:** Conectarse a PostgreSQL con el método "connect()".
- **Paso 5:** Realizar una consulta con el parámetro "client" e ingresar con el método "query()" una consulta para insertar unos zapatos de color negro y talla 44 en la tabla ropa. Ocupa el RETURNING * para obtener el registro que se está haciendo.
- **Paso 6:** Liberar al cliente de la consulta con el parámetro release.
- **Paso 7:** Imprimir por consola la última inserción realizada.
- **Paso 8:** Desconectar Pool con el método "end()".

```
// Paso 1
const { Pool } = require("pg");
// Paso 2
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

// Paso 3
const pool = new Pool(config);

// Paso 4
pool.connect((error_conexion, client, release) => {

  // Paso 5
  client.query("insert into ropa (nombre, color, talla) values
('zapatos', 'negro', '44') RETURNING *;", (error_query, resul) => {

    // Paso 6
    release();

    // Paso 7
    console.log("Ultimo registro agregado: ", resul.rows[0]);
  });

  // Paso 8
  pool.end();
});
```

Si ejecutas esta aplicación deberás recibir lo que te muestro en la siguiente imagen:

```
$ node index.js
Ultimo registro agregado: { id: 1, nombre: 'zapatos', talla: '44', color: 'negro' }
```

Imagen 2. Inserción de datos con la clase Pool.
Fuente: Desafío Latam

¡Felicidades! Has logrado tu primera inserción con la clase Pool.

Ejercicio propuesto (1)

Utilizar la tabla **usuarios** creada en la lectura anterior e ingresar por medio de la clase Pool un nuevo registro, imprimiendo por consola la inserción de la consulta.

Consultas con texto parametrizado

Competencias

- Reconocer la importancia del uso de parámetros en consultas para evitar un ataque de tipo SQL Injection.
- Construir una aplicación Node para realizar una consulta a PostgreSQL con texto plano parametrizado.

Introducción

Hasta ahora hemos realizado consultas a las bases de datos con sentencias SQL, en donde se declaran todos los datos que se están pidiendo o ingresando a PostgreSQL, esto puede representar un problema de seguridad, porque fuera de los ejercicios académicos y en un plano laboral, los usuarios de las aplicaciones serán quienes escriben esos datos involucrados en las sentencias SQL y esto pudiese permitir ataques de tipo SQL Injection a tus bases de datos.

En este capítulo, aprenderás a realizar consultas con texto parametrizado ocupando el mismo método `query()` que se utilizó en el capítulo anterior, pero ahora tendrás más control de los datos que se están enviando a PostgreSQL, puesto que manejarás los valores de las consultas a través de parámetros, para evitar un posible ataque SQL Injection.

Query utilizando texto plano con parámetros

El método query además de permitir realizar consultas mediante un texto plano, también nos permite realizarlas en forma parametrizada. De esta forma, el servidor PostgreSQL recibe la consulta sin modificaciones con los parámetros en forma separada, estos son sustituidos en forma segura en la base de datos, a través de un código de sustitución de parámetros, que se denotan utilizando el signo "\$" seguido del número del parámetro. Por ejemplo, si necesitamos sustituir dos parámetros en la consulta SQL, estos deben indicarse de la forma \$1, \$2.

¿Por qué utilizaría texto plano parametrizado? En SQL se popularizó un ataque a los sistemas y bases de datos llamado SQL Injection, este consiste en lograr ingresar una instrucción que vulnere, copie, elimine o afecte de alguna manera la base de datos. Si quieres saber más sobre SQL Injection revisa el **Material Apoyo Lectura - SQL Injection**, ubicado en "Material Complementario".

Ahora que entiendes la importancia de fragmentar la consulta SQL con parámetros, veamos cómo construir una consulta parametrizada. A continuación te muestro su sintaxis:

```
Pool.query(text[String], values[Array])
```

Como puedes ver se usó de igual manera el método query pero ahora estamos agregando un parámetro. ¿Qué es esto? sucede que ahora el primer parámetro sería una sentencia SQL, que incluye caracteres con el signo de peso(\$) acompañados de números enteros y el segundo parámetro serán valores ordenados en un arreglo, representando las posiciones indicadas en la instrucción. Para manejar de una forma más limpia el código, trataremos esta consulta como una función asíncrona.

Ejercicio guiado: Consultando con parámetros

Desarrollar otra inserción para la tabla **ropa**. En esta ocasión agregaremos unos calcetines verdes talla M, pero a diferencia del ingreso que hicimos en el capítulo anterior, esta inserción la haremos a través de una consulta parametrizada.

- **Paso 1:** Declarar que la función callback de la conexión será asíncrona.
- **Paso 2:** Realizar una consulta con parámetros para agregar unos calcetines verdes de talla M.
- **Paso 3:** Liberar al cliente de la consulta.

- **Paso 4:** Imprimir por consola la inserción.
- **Paso 5:** Cerrar la conexión.

```
// Paso 1
pool.connect(async (error_conexion, client, release) => {
  // Paso 2
  const res = await client.query(
    "insert into ropa (nombre, color, talla) values ($1, $2, $3)
RETURNING *;",
    ["calcetines", "verde", "M"]
  );

  // Paso 3
  release();

  // Paso 4
  console.log(res.rows[0]);

  // Paso 5
  pool.end();
});
```

Ahora ejecuta tu aplicación y deberás ver algo como lo que te muestro en la siguiente imagen:

```
$ node index.js
{ id: 2, nombre: 'calcetines', talla: 'M', color: 'verde' }
```

Imagen 3. Inserción de datos con texto parametrizado.

Fuente: Desafío Latam

Ahí lo tienes, tu primera consulta con texto parametrizado. Cabe destacar que esto ayuda al proceso para evitar una inyección SQL, no obstante, podrías notar que aún es posible realizar un ataque a la base de datos puesto que cada valor de parámetro pudiese contener una sentencia SQL. Para evitar esto, utilizamos los objetos JSON como argumento de una consulta y lo verás en el siguiente capítulo.

Ejercicio propuesto (2)

Realizar una nueva inserción a la tabla **usuarios** con los siguientes datos:

- Id: 5
- Nombre: Jonathan
- Teléfono: 989786545

Realizar la consulta usando texto plano parametrizado y `async/await`.

JSON como argumento de una consulta

Competencias

- Construir una aplicación en Node que realice una consulta a PostgreSQL pasando un JSON como argumento para evitar un SQL Injection.
- Implementar el uso del Row Mode en el JSON como argumento de una consulta para recibir los datos de PostgreSQL en formato de arreglo.

Introducción

Ahora que aprendiste sobre consultas parametrizadas y SQL Injection, has entendido la importancia de proteger tu base de datos, desde el puente de comunicación entre las aplicaciones y la fuente de datos.

En este capítulo, aprenderás a segmentar aún más la consulta, descartando cualquier ataque SQL por medio de un JSON, como argumento del método `query()`. Además, aprenderás cómo recibir los registros que necesites en formato de arreglo en caso que lo consideres necesario, a diferencia del formato de objetos que hasta ahora has manipulado.

Realizar consultas con un JSON como argumento sellará la seguridad implementada en la lógica de tus aplicaciones en Node con el paquete `pg`.

Objeto JSON como argumento de consultas

Los métodos Client y Pool de la librería pg, permiten el manejo de un objeto con la configuración de la consulta a la base de datos. En términos de seguridad, el uso de este objeto es el óptimo para utilizar una cadena de texto y concatenar los parámetros que reciba en forma dinámica la consulta. El manejo de una cadena de texto genera vulnerabilidades en nuestra aplicación, debido a que quedamos expuestos a recibir un ataque de Sql Injection.

Al manejar un objeto JSON o variables independientes en la ejecución de la consulta, la base de datos evaluará en forma separada los parámetros del resto de la consulta y descartará cualquier instrucción SQL

¿Y cuál es la estructura de este JSON? Este objeto solo debe tener una propiedad "text" que contendrá la sentencia SQL a ejecutar y "values" donde serán definidos los parámetros que recibirá la consulta. A continuación te muestro como sería la sintaxis de esto:

```
{
  text: '<consultas con parámetros $1,$2, ...',
  values: ['valor1', 'valor2', ...],
};
```

Este objeto sería un argumento del método query() y será interpretado por el paquete pg sin problema.

Ejercicio guiado: Consultas SQL con JSON

Realizar otra inserción pero en este caso se tratará de un cinturón color gris de talla 98. Para esto sigue los siguientes pasos:

- **Paso 1:** Crear un objeto con las propiedades text y values en donde deberás definir la consulta parametrizada y los valores en forma de arreglo respectivamente.
- **Paso 2:** Enviar el objeto JSON creado en el paso 1 como argumento del método query

```
pool.connect(async (error_conexion, client, release) => {  
  // Paso 1  
  const SQLQuery = {  
    text:  
      "insert into ropa (nombre, color, talla) values ($1, $2, $3)  
RETURNING *;",  
    values: ["cinturon", "gris", "98"],  
  };  
  
  // Paso 2  
  const res = await client.query(SQLQuery);  
  
  release();  
  console.log(res.rows[0]);  
  pool.end();  
});
```

Ejecuta tu aplicación y deberás ver algo como lo que te muestro en la siguiente imagen:

```
$ node index.js  
{ id: 3, nombre: 'cinturon', talla: '98', color: 'gris' }
```

Imagen 4. Inserción de datos pasando como argumento un JSON.
Fuente: Desafío Latam

Row Mode

En el objeto JSON construido es posible definir otra propiedad llamada Row Mode con el valor "array", cuyo objetivo es devolvernos los registros que estemos agregando o consultando en forma de arreglo. ¿Cuál es la diferencia? Como has podido notar y en la imagen anterior se muestra, los registros por defecto son retornados en forma de objeto, pero en caso de querer manejar esta data en forma de arreglos independientes podemos hacerlo gracias al Row Mode.

Ejercicio guiado: Obteniendo arreglos como respuesta

Obtener todos los registros de la tabla **ropa** en forma de arreglos, prosigue con los siguientes pasos:

- **Paso 1:** Declarar la propiedad "rowMode" con el valor "array", en el JSON como argumento al método query.
- **Paso 2:** Cambiar el valor de la propiedad "text" para realizar una consulta que pida todos los registros de la tabla ropa.
- **Paso 3:** Imprimir por consola la propiedad "rows" del objeto recibido como respuesta de la consulta para visualizar la data que se está consultando en formato de arreglo.

```
pool.connect(async (error_conexion, client, release) => {  
  
  const SQLQuery = {  
    // Paso 1  
    rowMode: 'array',  
    // Paso 1  
    text: "select * from ropa",  
  };  
  
  const res = await client.query(SQLQuery);  
  
  release();  
  
  // Paso 3  
  console.log(res.rows);  
  pool.end();  
});
```

Ahora ejecuta tu aplicación y deberás obtener lo que te muestro en la siguiente imagen:

```
$ node index.js  
[  
  [ 1, 'zapatos', '44', 'negro' ],  
  [ 2, 'calcetines', 'M', 'verde' ],  
  [ 3, 'cinturon', '98', 'gris' ]  
]
```

Imagen 5. Recepción de los registros de la tabla ropa en forma de arreglos.

Fuente: Desafío Latam

Como puedes notar ahora los registros los estás recibiendo en forma de arreglo.

Ejercicio propuesto (3)

Realizar una consulta a la tabla **usuarios** para obtener todos los registros, pero en esta ocasión pasa un JSON como argumento a la consulta y define el Row Mode con el valor "array". Imprimir por consola los registros obtenidos.

Prepared Statement

Competencia

- Reconocer el uso del Prepared Statement en PostgreSQL a través del paquete pg para optimizar consultas.

Introducción

Una consulta a las bases de datos puede ser tan pequeña y simple, como grande y compleja, esta última amerita un cómputo mucho mayor, requiere de varios procesos de búsqueda y ordenamiento para realizarse, generando un tiempo de espera para el usuario que consulta.

El prepared statement es una técnica aplicada en varias tecnologías y su objetivo es almacenar un identificador en formato de texto que represente un proceso determinado, en el caso de PostgreSQL, una consulta SQL. Utilizar esta técnica tiene más sentido cuando las aplicaciones manejan cantidades industriales de información y las consultas se vuelven complejas. Aprendiendo a utilizar el prepared statement podrás ofrecer un menor tiempo de espera al usuario consultante.

Prepared Statements

La base de datos PostgreSQL implementa el concepto de declaraciones preparadas, que hace referencia al manejo de una memoria caché que existirá en la base de datos y estará asociada a una conexión. El manejo realizado por este caché será en forma transparente para la aplicación y se encuentra enfocado en optimizar los tiempos de respuestas al repetir la misma consulta a la base de datos.

Para indicar a la base de datos PostgreSQL que activaremos el caché en la consulta debemos:

- Crear un objeto que permita definir parámetros de configuración a la consulta generada, en este caso crearemos el objeto llamado `queryObj`, que tendrá un atributo `name` que corresponde al nombre que recibirá la base de datos, para "cachear" la consulta y el atributo `text`, que indicará la sentencia SQL que ejecutará el método `query`, al realizar la consulta a la base de datos.
- Al incluir este parámetro la primera vez que se ejecuta la consulta, se almacenará el plan de ejecución, todos los procesos asociados al análisis y la planificación que requiera la ejecución de la consulta.
- La segunda vez que se ejecute con el mismo nombre, la base de datos descarta los procesos asociados y ejecutará directamente la consulta.

Para implementar esta técnica solo debemos incluir una propiedad al JSON como argumento de la consulta. A continuación, te muestro un código de ejemplo en donde se incluye la propiedad `name` al JSON como argumento:

```
const queryObj = {  
  const queryObj = {  
    name: 'fetch-user', // prepared statement  
    text: `SELECT  
      att_id,  
      app_clients.cli_name||' '||app_clients.cli_lastname as  
Cliente,  
      att_date,  
      att_detail,  
      app_employee.emp_name||' '||app_employee.emp_lastname as  
Ejecutivo  
      FROM app_attentions  
      INNER JOIN app_clients  
      ON app_attentions.cli_id = app_clients.cli_id  
      INNER JOIN app_employee  
      ON app_attentions.emp_id = app_employee.emp_id  
    ,  
  }  
}
```

Como puedes notar la consulta de ejemplo es enorme y esto es porque la técnica de Prepared Statement es justamente aplicada a consultas de alto cómputo para la base de datos, en una menor escala sería imperceptible notar la diferencia.

¿Qué tanto impacto hace Prepared Statement? A continuación te mostraré la ejecución correspondiente a la consulta de ejemplo previa, en un caso sin prepared statement y en otro si. El tiempo de la consulta se encontrará al final de la tabla.

Prueba sin usar Prepared statement

La ejecución de la consulta sin utilizar Prepared Statements, como se puede ver en la imagen se demoró 6.107 milisegundos:

(index)	att_id	cliente	att_date	att_detail	ejecutivo
0	7	'Martin Verdugo'	2020-03-13T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
1	6	'Martin Verdugo'	2020-02-08T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Benjamin Inostroza'
2	5	'Martin Verdugo'	2020-02-04T03:00:00.000Z	'Sin Servicio'	'Benjamin Inostroza'
3	4	'Martin Verdugo'	2020-01-22T03:00:00.000Z	'Como descargar los manuales del telephone'	'Ana Maria Fuentes'
4	3	'Martin Verdugo'	2020-01-12T03:00:00.000Z	'Como pagar mi cuenta'	'Ana Maria Fuentes'
5	2	'Martin Verdugo'	2019-12-24T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
6	1	'Martin Verdugo'	2019-12-24T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
7	10	'Alejandra Morande'	2019-12-22T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Ana Maria Fuentes'
8	9	'Alejandra Morande'	2019-12-13T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
9	8	'Alejandra Morande'	2019-11-03T03:00:00.000Z	'Como solicito traslado de domicilio'	'Ana Maria Fuentes'
10	14	'Cesar Orellana'	2020-02-22T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
11	13	'Cesar Orellana'	2020-02-14T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
12	12	'Cesar Orellana'	2020-01-20T03:00:00.000Z	'Que es 4G'	'Ana Maria Fuentes'
13	11	'Cesar Orellana'	2019-12-18T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Ana Maria Fuentes'
14	15	'Ingrid Rivera'	2020-03-22T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'

calculate-time: 6.107ms

Imagen 6. Medición de tiempo de respuesta sin utilizar Prepared statements.

Fuente: Desafío Latam

Prueba utilizando Prepared Statement

En esta consulta se utilizó Prepared Statement y devolvió un tiempo de ejecución de 5.059 milisegundos como se puede ver en la imagen:

(index)	att_id	cliente	att_date	att_detail	ejecutivo
0	7	'Martin Verdugo'	2020-03-13T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
1	6	'Martin Verdugo'	2020-02-08T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Benjamin Inostroza'
2	5	'Martin Verdugo'	2020-02-04T03:00:00.000Z	'Sin Servicio'	'Benjamin Inostroza'
3	4	'Martin Verdugo'	2020-01-22T03:00:00.000Z	'Como descargar los manuales del telephone'	'Ana Maria Fuentes'
4	3	'Martin Verdugo'	2020-01-12T03:00:00.000Z	'Como pagar mi cuenta'	'Ana Maria Fuentes'
5	2	'Martin Verdugo'	2019-12-24T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
6	1	'Martin Verdugo'	2019-12-24T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
7	10	'Alejandra Morande'	2019-12-22T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Ana Maria Fuentes'
8	9	'Alejandra Morande'	2019-12-13T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
9	8	'Alejandra Morande'	2019-11-03T03:00:00.000Z	'Como solicito traslado de domicilio'	'Ana Maria Fuentes'
10	14	'Cesar Orellana'	2020-02-22T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'
11	13	'Cesar Orellana'	2020-02-14T03:00:00.000Z	'Sin Servicio'	'Ana Maria Fuentes'
12	12	'Cesar Orellana'	2020-01-20T03:00:00.000Z	'Que es 4G'	'Ana Maria Fuentes'
13	11	'Cesar Orellana'	2019-12-18T03:00:00.000Z	'Como puedo cambiarme a un plan con fibra óptica'	'Ana Maria Fuentes'
14	15	'Ingrid Rivera'	2020-03-22T03:00:00.000Z	'Como cambiar la clave del WIFI'	'Ana Maria Fuentes'

calculate-time: 5.059ms

Imagen 7. Medición de tiempo de respuesta usando Prepared statements.

Fuente: Desafío Latam

Como puedes notar si existe un cambio aunque pareciera poco, hay que pensar que esto es escalable a consultas aún más complejas y este 17,16% de mejora puede ser muy importante.

El paquete pg ofrece varias herramientas como esta, entre otra de las populares está el parseo de tipos, el cual consiste en utilizar el interpretador de tipos de PostgreSQL y formatear nuestros datos según su parser. Si quieres saber más sobre esta técnicas y otras, en la [documentación oficial de pg](#), creado por el estadounidense Brian Carlson encontrarás todo lo referente a esta librería.

Captura de errores

Competencia

- Implementar el procedimiento de captura de errores de conexión y errores de consulta utilizando el entorno Node.

Introducción

En este capítulo aprenderás a capturar los errores que puedan suceder tanto en una conexión a PostgreSQL como en una consulta SQL. Aprenderás que para capturar estos errores podemos usar los callbacks de los métodos asíncronos del paquete pg como el "connect()" y el "query()", ó ocupando la sentencia try catch de JavaScript dentro de una función async/await, para capturar cualquier instancia fallida que suceda dentro de un bloque de código.

La captura de errores es de suma importancia en el desarrollo de cualquier aplicación puesto que recibirás de manera clara un feedback sobre ¿Qué sucedió? ¿Por qué sucedió? a través de la consola, y conociendo el motivo del error podrás tomar medidas para solucionarlo.

Consideraciones previas

Así como los errores de estado HTTP, también tenemos un listado de errores en PostgreSQL que representan una situación. Para familiarizarse mejor con estos errores conociendo sus códigos y sus descripciones revisa el **Material Apoyo Lectura - Interpretación de errores PostgreSQL**, ubicado en "Material Complementario".

Además de todos los errores que existen tenemos un grupo de los errores más comunes que suceden en el desarrollo, si los quieres conocer, cuáles son y cómo solucionarlos revisa el **Material Apoyo Lectura - Errores más comunes**, ubicado en "Material Complementario".

Capturando errores de conexión

Los errores de conexión podemos capturarlos en el momento que ejecutamos el método `connect()` de la clase `Pool`, porque en su callback recibimos como primer parámetro la instancia del posible error que haya sucedido.

Ejercicio guiado: Capturando errores de conexión

Consultar todos los registros de la tabla **ropa** de la base de datos **jeans**, pero voluntariamente escribir mal el nombre de la base de datos quitándole la "s" al final de la palabra e indispensablemente escribiremos un condicional, que detecte si existe el error y devuelva por consola el código de este.

Sigue los siguientes pasos para capturar el error de conexión:

- **Paso 1:** Cambiar el nombre de la base de datos por uno incorrecto.
- **Paso 2:** Usar un condicional `if` para retornar el código de error de conexión tomando como valor el primer parámetro del callback en el método `connect()`.


```
const { Pool } = require("pg");

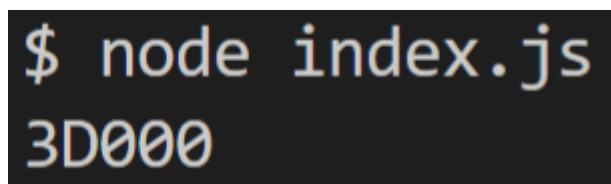
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  // Paso 1
  database: "jean",
  port: 5432,
};

const pool = new Pool(config);

pool.connect(async (error_conexion, client, release) => {
  // Paso 2
  if (error_conexion) return console.error(error_conexion.code);

  const res = await client.query("select * from ropa");
  console.log(res.rows);
  release();
  pool.end();
});
```

Ahora ejecuta esta aplicación y deberás recibir lo que te muestro en la siguiente imagen.



```
$ node index.js
3D000
```

Imagen 8. Obteniendo el código de error de conexión por consola.

Fuente: Desafío Latam

El error es "3D000", por lo tanto pertenece a la clase 3D, el cual se interpreta como "Nombre de catálogo no válido".

Capturando errores de consultas

Al igual que los errores de conexión los errores de consultas los podemos tomar por medio del primer parámetro que devuelve el callback, en este caso el método query().

Ejercicio guiado: Capturando errores en una consulta

Consultar a una tabla que no existe en la base de datos **jeans**, en este caso simplemente agregarle una "s" al final del nombre de la tabla, prosigue con los siguientes pasos:

- **Paso 1:** Realizar una consulta con el método query, ocupando su función callback para definir el parámetro de error correspondiente al primer parámetro. La consulta la debes hacer con un nombre de tabla incorrecto, en este caso consultar los registros de una tabla "ropas".
- **Paso 2:** Usar un condicional if para retornar el código de error de consulta tomando como valor el primer parámetro del callback en el método query().

```
const { Pool } = require("pg");

const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

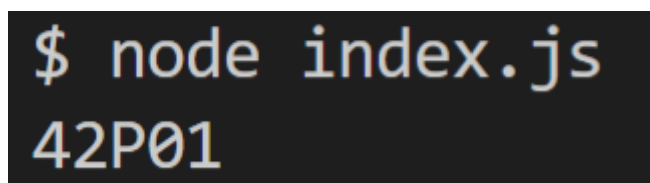
const pool = new Pool(config);

pool.connect(async (error_conexion, client, release) => {
  if (error_conexion) return console.error(error_conexion.code);

  // Paso 1
  client.query("select * from ropas", (error_consulta, res) => {
    // Paso 2
    if (error_consulta) return console.error(error_consulta.code);
    console.log(res.rows);
  });
});
```

```
release();  
pool.end();  
});  
});
```

Ejecuta la aplicación y deberás recibir el error que te muestro en la siguiente imagen:



```
$ node index.js  
42P01
```

Imagen 9. Obteniendo el código de error de una consulta.
Fuente: Desafío Latam

El error resultante fue “42P01” el cual pertenece a la clase 42 que se interpreta como “Error de sintaxis o infracción de la regla de acceso”

Capturando error de consultas con try catch y async await

Otra forma de capturar este error es por medio del try catch, el cual se implementa a un bloque de instrucciones y en caso de que este devuelva un error lo podrás capturar por medio del catch.

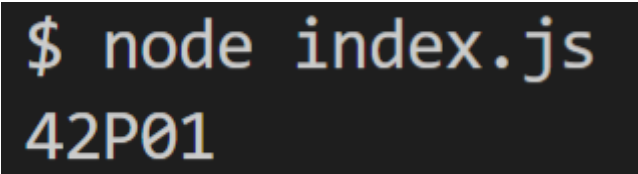
Ejercicio guiado: Capturando errores con try catch

Consultar una tabla que no existe pero tomando el error por medio de un try catch, sigamos los siguientes pasos:

- **Paso 1:** Utilizar la sentencia try para ejecutar una consulta con el método query, acompañada con la instrucción await, almacenando su retorno en una constante “res”.
- **Paso 2:** Utilizar la sentencia catch recibiendo como parámetro el error de la consulta.
- **Paso 3:** Imprimir dentro del bloque de código del catch el error de la consulta.

```
pool.connect(async (error_conexion, client, release) => {  
  if (error_conexion) return console.error(error_conexion.code);  
  
  // Paso 1  
  try {  
    const res = await client.query("select * from ropas");  
    console.log(res.rows);  
  } catch (error_consulta) {  
    // Paso 3  
    console.log(error_consulta.code);  
  }  
  release();  
  pool.end();  
});
```

Ahora ejecuta tu aplicación y deberás recibir lo que te muestro en la siguiente imagen:



```
$ node index.js  
42P01
```

Imagen 10. Obteniendo el código de error de una consulta a través del try catch.
Fuente: Desafío Latam

Como puedes notar seguimos recibiendo el error sin problemas y en este caso estaríamos escribiendo un código más legible y limpio.

Ejercicio propuesto (4)

Realizar una consulta a la tabla **usuarios** que intente realizar una inserción con un registro que contenga un id existente en otro registro de la tabla. Realizar la consulta con un JSON como argumento y capturar el error de la consulta con try catch e imprimiendo su código por consola.

Resumen

A lo largo de esta lectura aprendimos a realizar consultas SQL aplicando el concepto de Pooling para ofrecer la gestión de múltiples consultas de manera simultánea, así como consultas con texto parametrizado y la captura de errores que pueden aparecer en una conexión a PostgreSQL o una consulta SQL, abordando lo siguiente:

- El concepto de Pooling para las multiconexión de clientes y consultas SQL.
- Consulta a la base de datos con la clase Pool conectándonos con el método "connect()" y usando el parámetro "client" para realizar consultas SQL con su método "query".
- Consultas SQL con parámetros para particionar los datos involucrados en la consulta definiendo los valores en un segundo argumento del método "query".
- JSON como argumento de una consulta para la emisión de consultas SQL utilizando un objeto compuesto de propiedades para una mayor personalización de la consulta y evitar un ataque SQL Injection.
- La propiedad rowMode en el JSON como argumento de una consulta para recibir los registros en formato de arreglo.
- La propiedad "name" en el JSON como argumento de una consulta para usar el prepared statements en PostgreSQL y agilizar una misma consulta a partir de su segunda emisión.
- Captura de errores de conexión y consultas para recibir un feedback claro sobre ¿Qué sucedió? o ¿Por qué sucedió?.

Solución de los ejercicios propuestos

1. Utilizar la tabla **usuario** que creaste en la lectura anterior e ingresa por medio de la clase Pool un nuevo registro imprimiendo por consola la inserción de la consulta.

```
const { Pool } = require("pg");
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

const pool = new Pool(config);

pool.connect((error_conexion, client, release) => {
  client.query(
    "insert into usuarios (id, nombre, telefono) values (4, 'Brian', '12345678') RETURNING *;",
    (error_query, resul) => {
      console.log(error_query);
      release();

      console.log("Ultimo registro agregado: ", resul.rows[0]);
    }
  );
});

pool.end();
});
```

2. Realizar una nueva inserción a la tabla usuarios con los siguientes datos:

- Id: 5
- Nombre: Jonathan
- Teléfono: 989786545

Realizar la consulta usando texto plano parametrizado y async/await.

```
const { Pool } = require("pg");
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

const pool = new Pool(config);

pool.connect(async (error_conexion, client, release) => {
  const res = await client.query(
    "insert into usuarios (id, nombre, telefono) values ($1, $2, $3) RETURNING *;",
    [5, "Jonathan", "87654321"]
  );

  release();
  console.log("Ultimo registro agregado: ", res.rows[0]);

  pool.end();
});
```

3. Realizar una consulta a la tabla **usuarios** para obtener todos los registros pero en esta ocasión pasa un JSON como argumento a la consulta y define el Row Mode con el valor "array". Imprimir por consola los registros obtenidos.

```
const { Pool } = require("pg");
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

const pool = new Pool(config);

pool.connect(async (error_conexion, client, release) => {
  const SQLQuery = {
    rowMode: "array",
    text:
      "SELECT * FROM usuarios",
  };

  const res = await client.query(SQLQuery);

  release();
  console.log("Ultimo registro agregado: ", res.rows);

  pool.end();
});
```


4. Realizar una consulta a la tabla **usuarios** que intente realizar una inserción con un registro que contenga un id existente en otro registro de la tabla. Debes realizar la consulta con un JSON como argumento y capturar el error de la consulta con try catch e imprimiendo su código por consola.

```
const { Pool } = require("pg");
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "jeans",
  port: 5432,
  max: 20,
  idleTimeoutMillis: 5000,
  connectionTimeoutMillis: 2000,
};

const pool = new Pool(config);

pool.connect(async (error_conexion, client, release) => {
  const SQLQuery = {
    rowMode: "array",
    text:
      "insert into usuarios (id, nombre, telefono) values ($1, $2, $3)
RETURNING *;",
    values: [5, "Jonathan", "87654321"],
  };

  try {
    const res = await client.query(SQLQuery);
    console.log("Ultimo registro agregado: ", res.rows);
  } catch (error) {
    console.log(error.code);
  }

  release();

  pool.end();
});
```