

Transacciones, cursores y API REST (Parte I)

Cursores

Competencias

- Reconocer el uso de cursores para consultar registros en PostgreSQL utilizando el entorno Node.
- Construir un programa que maneje cursores sobre una base de datos PostgreSQL utilizando el entorno Node.

Introducción

En este capítulo aprenderás a utilizar los cursores que nos ofrecen las bases de datos SQL, el cual, permite manipular un grupo de datos, que se encuentran entre una aplicación web y la base de datos PostgreSQL. Se debe enfatizar que esta funcionalidad existe en el motor de base de datos y que desde Node ocupamos el paquete pg-cursor para realizar esta operación.

Otro factor importante de la utilización de cursores es que nos permiten manipular información en forma segmentada, para evitar consumos altos de memoria al ejecutar consultas que generen un alto volumen de datos, los cuales pueden afectar el tiempo de la consulta que deseamos realizar, por lo que aprender los cursores en Node con el paquete pg-cursor garantizará un menor tiempo de espera y evitará un procesamiento de cómputo innecesario.

Ejecución del cursor desde Node

Antes de iniciar con la lectura de este capítulo te recomiendo que revises el **Material Apoyo Lectura - Cursores en PostgreSQL**, ubicado en "Material Complementario", en donde aprenderás brevemente ¿Qué es un cursor?, ¿Cómo usarlos? y ¿Cómo cerrarlos? directamente en la consola psql.

En pocas palabras un cursor se puede definir como un puntero que representa una instrucción escrita con el comando SELECT y su uso radica en la mejora del rendimiento por parte del motor de base de datos, ¿Pero por qué mejora el rendimiento? Porque una vez se haya emitido una consulta con un cursor, éste será reconocido en su próxima ejecución y el motor de base de datos no necesitará hacer la lectura de la sentencia completa, sino que ya sabrá exactamente qué es lo que se está solicitando.

pg-cursor

Podrías estar haciéndote la pregunta ¿Y cómo puedo utilizar los cursores desde Node? Pues resulta que nuestro paquete protagonista "pg" es el núcleo de diferentes extensiones y paquetes creados por su autor Brian Carlson, una de estas extensiones es "pg-cursor". Es uno de los paquetes derivados de la librería "pg", utilizado para realizar consultas desde Node utilizando cursores.

Instalación

Como se trata de un paquete diferente deberemos instalarlo en nuestro proyecto y el comando para hacer esto es el siguiente:

```
npm i pg-cursor
```

Este paquete junto a "pg" será lo único necesario para utilizar los cursores con Node.

Sintaxis

La sintaxis de la extensión "pg-cursor" no es muy diferente a la que ya conoces del paquete "pg", no obstante, hay que seguir unos pasos para poder emplear los cursores correctamente, desde su importación hasta su cierre.

Importación

Para hacer una consulta con cursores debes antes que todo importar el paquete “pg-cursor”, el cual devuelve por defecto una clase, así como te muestro en el siguiente código:

```
const Cursor = require("pg-cursor")
```

Consulta

Esta clase recibe como constructor un String con la consulta SQL o un JSON como argumento, así como te muestro en el siguiente código, el cuál utilizarías en caso de querer hacer una consulta parametrizada:

```
new Cursor(sql, values)
```

De no querer parametrizar la consulta, puedes enviar como argumento al constructor la sentencia SQL en puro formato String, en el siguiente código te muestro un ejemplo de cómo sería la realización de una consulta de esta manera:

```
const consulta = new Cursor("select * from usuarios")  
const cursor = client.query(consulta);
```

Observa que estoy utilizando un objeto “cliente”, el cual puede ser la instancia de la clase cliente o el parámetro que nos ofrece la conexión con la clase Pool. Si no te sientes muy ubicado en este punto no te preocupes, más adelante con un ejercicio lo verás con más claridad.

Lectura

Para leer el resultado de esta consulta, debes utilizar el método “read” de la instancia del cursor, el cual recibe como primer parámetro un valor numérico, donde se indica la cantidad de registros que el cursor debe retornar y como segundo parámetro, una función callback que recibe un posible error durante la consulta y un arreglo con los registros solicitados en formato de objeto.

En el siguiente código, te muestro un ejemplo de la lectura de la consulta realizada previamente:

```
cursor.read(10, (err, rows) => {  
  console.log(rows);  
});
```

Cierre del cursor

La instancia del cursor contiene un método “close()” que al ser ejecutado, se le está indicando al motor de base de datos que una vez obtenido los registros especificados no siga con la consulta.

Este cierre se recomienda ocupar dentro del callback del método “read”, así como te muestro en el siguiente código.

```
cursor.read(10, (err, rows) => {  
  console.log(rows);  
  cursor.close()  
});
```

Mi primera consulta con cursores

Ahora que conoces la sintaxis del paquete “pg-cursor”, procedamos con el desarrollo de una aplicación en Node, que al ser ejecutada realizará una consulta con el paquete pg-cursor a una base de datos. ¿Cuál base de datos? Revisa entre los contenidos de esta sesión y encontrarás el **Apoyo Lectura - Base de datos de usuarios**. Este archivo comprimido contiene una base de datos en formato csv con 500 registros de usuarios y será la base de datos que usaremos para probar los cursores.

Para iniciar con la programación de esta aplicación deberás tener una base de datos ya creada o crear una nueva, puedes usar la siguiente instrucción para crear una base de datos llamada **clientes**.

```
CREATE DATABASE clientes;
```

Ahora que tienes la base de datos creada, deberás crear una tabla **usuarios** con los campos "first_name", "last_name" y "email". ¿No recuerdas exactamente cómo se escribe la instrucción SQL para crear la tabla? No te preocupes, a continuación te muestro el código para hacerlo:

```
CREATE TABLE usuarios (first_name varchar(100), last_name varchar(100),  
email varchar(100));
```

Una vez creada la tabla, necesitarás importar el csv con los datos de los clientes y para eso puedes ejecutar la siguiente instrucción en la terminal de PostgreSQL.

```
\copy usuarios from '<dirección del archivo csv>' DELIMITER ',' CSV  
HEADER;
```

Recuerda cambiar la dirección del archivo csv según el árbol de archivos de tu sistema operativo.

Ejercicio guiado: Mi primer cursor SQL

Construir una aplicación que consulte e imprima por consola los primeros 5 clientes del restaurante "Full Belly, Happy Heart", de manera que los usuarios de la base de datos serán los clientes de este restaurante. Realizar la primera consulta con cursores desde una aplicación Node, para esto sigue los siguientes pasos:

- **Paso 1:** Importar la clase Pool del paquete pg. Utilizamos esta clase por buenas prácticas y para anticiparnos a una posible alta demanda de conexiones y peticiones a la base de datos.
- **Paso 2:** Importar la clase Cursor del paquete pg-cursor.
- **Paso 3:** Crear un objeto con la configuración de los datos de usuario, servidor, contraseña de usuario, base de datos y el puerto 5432.
- **Paso 4:** Ejecutar una conexión con el método connect de la instancia de la clase Pool.
- **Paso 5:** Crear una constante que almacene la instancia de la clase Cursor, pasando como argumento una consulta SQL que solicite todos los registros de la tabla usuarios.
- **Paso 6:** Ocupar el método read para especificar que solo necesitas 5 registros y escribe la función callback, que recibe el error de la consulta y las filas solicitadas.

- **Paso 7:** Imprimir por consola las filas recibidas en callback.
- **Paso 8:** Cerrar el cursor, libera al cliente y cierra la conexión con la base de datos.

```
// Paso 1
const { Pool } = require("pg");
// Paso 2
const Cursor = require("pg-cursor");
// Paso 3
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
};
const pool = new Pool(config);

// Paso 4
pool.connect((error_conexion, client, release) => {
  // Paso 5
  const consulta = new Cursor("select * from usuarios");
  const cursor = client.query(consulta);
  // Paso 6
  cursor.read(5, (err, rows) => {
    // Paso 7
    console.log(rows);
    // Paso 8
    cursor.close();
    release();
    pool.end();
  });
});
```

Ahora ejecuta esta aplicación y deberás obtener lo que te muestro a continuación:

```
$ node usuarios.js
[
  { first_name: 'James', last_name: 'Butt', email: 'jbutt@gmail.com' },
  {
    first_name: 'Josephine',
    last_name: 'Darakjy',
    email: 'josephine_darakjy@darakjy.org'
  },
  { first_name: 'Art', last_name: 'Venere', email: 'art@venere.org' },
  {
    first_name: 'Lenna',
    last_name: 'Paprocki',
    email: 'lpaprocki@hotmail.com'
  },
  {
    first_name: 'Donette',
    last_name: 'Foller',
    email: 'donette.foller@cox.net'
  }
]
```

Imagen 1. Primeros 5 registros de la tabla usuarios consultado con cursores.

Fuente: Desafío Latam

Como puedes observar, obtuvimos los primeros 5 usuarios de la tablas usuarios a través de un cursor con el paquete pg-cursor.

Ejercicio propuesto (1)

Basado en el ejercicio de los clientes del restaurante, desarrolla una aplicación en Node que realice una consulta SQL con el paquete pg-cursor para obtener los primeros 20 usuarios de la tabla **usuarios**.

Transacciones

Competencias

- Reconocer el uso de transacciones en PostgreSQL utilizando el entorno Node.
- Codificar operaciones transaccionales sobre una base de datos PostgreSQL utilizando el entorno Node.

Introducción

En este capítulo aprenderemos cómo realizar operaciones transaccionales SQL, mediante la librería pg, viendo paso a paso cómo utilizar instrucciones del tipo begin, commit y rollback. Finalmente, procederemos a crear un ejercicio que ejecute transacciones con el ejemplo más común y práctico para el uso de esta herramienta, las transacciones bancarias.

Una vez que conoces y sabes utilizar las transacciones, estarás a solo un paso de implementar esta funcionalidad a tus aplicaciones con Node, abriendo las puertas al desarrollo de sistemas más seguros que garanticen integridad y consistencia en tus bases de datos.

Transacciones

Antes de iniciar con la lectura de este capítulo te recomiendo que revises el **Material Apoyo Lectura - Transacciones**, ubicado en "Material Complementario", en donde aprenderás brevemente ¿Qué es una transacción en bases de datos?, ¿Por qué son necesarias? y ¿Cuándo utilizarlas?, la integridad de una base de datos, la propiedad ACID y la técnica del Journaling.

Importancia de las transacciones en SQL

Las transacciones en SQL, resuelven un problema de alta importancia que puede suceder en el caso de necesitar realizar varias consultas cuyos impactos tienen alguna relación, el ejemplo más típico y académico es el de una transacción bancaria, donde el saldo de una persona disminuye una cantidad determinada y otra persona recibe este monto.

La pregunta que resuelve el concepto de transacción es ¿Dónde puede ocurrir el problema? Y la respuesta es que las consultas SQL podrían experimentar algún problema en su ejecución, por ejemplo:

- Error de comunicación con la base de datos.
- Error de sintaxis en la instrucción SQL.
- Error por una restricción definida en la creación de un campo.
- Error por no encontrar el registro específico.
- Entre otros.

Estos errores pueden representar problemas importantes si no se consideran en la realización de consultas de impacto relacionado. Las transacciones nos garantizan que si necesitas realizar varias consultas, todas se realizarán con éxito o en caso de alguna experimentar algún problema ninguna lo hará.

Preparando un escenario para las transacciones

Para emplear esta importante herramienta desde una aplicación Node, lo único que debemos hacer es escribir una lógica que respete y espere la finalización de una consulta, para proceder a otra sin cerrar la conexión o liberar al cliente que emite esta petición.

Para realizar un ejercicio que ocupe transacciones en Node, necesitaremos preparar un escenario bancario donde cada usuario deberá tener un campo "saldo", que representará la cantidad de dinero que tendrán, ya que tenemos la tabla creada desde el capítulo anterior, podremos ocupar la misma base de datos, no obstante necesitaremos agregar el campo "saldo" en la tabla, por lo que deberás utilizar la siguiente instrucción en la terminal de PostgreSQL:

```
ALTER TABLE usuarios ADD saldo DECIMAL CHECK (saldo >= 0);
```

Observa que tenemos una restricción “CHECK” que va a encargarse de rebotar una consulta SQL, si esta intenta actualizar el saldo de un usuario a un valor negativo.

Ahora que tenemos el campo agregado a la tabla **usuarios**, necesitamos terminar de preparar el escenario para el ejercicio y nuestros usuarios no tienen dinero, así que cedamos una cantidad de saldo para todos los usuarios, para eso utiliza la siguiente instrucción:

```
UPDATE usuarios set saldo = 20000;
```

Con esta instrucción todos los usuarios tendrán un saldo de \$20.000 y ya estaríamos listos para iniciar un ejercicio aplicando transacciones.

Ejercicio guiado: Usando transacciones SQL en Node, violando restricciones

Desarrollar una aplicación en Node, que al ser ejecutada realice consultas SQL, usando las transacciones para transferir \$25.000 desde el cliente registrado en la tabla **usuarios** de correo **yuki_whobrey@aol.com**, al cliente también registrado en la tabla con el correo **fletcher.flosi@yahoo.com**, ambos son clientes del banco llamado “Dinero Azul”. Debemos esperar que esta transacción no se termine puesto que el primer usuario no tiene esa cantidad en su saldo.

Prosigue con los siguientes pasos para el desarrollo de este ejercicio:

- **Paso 1:** Realizar una consulta SQL con el comando BEGIN para iniciar la transacción.
- **Paso 2:** Realizar una consulta SQL para actualizar el saldo del usuario de correo yuki_whobrey@aol.com descontando \$25.000.
- **Paso 3:** Realizar una consulta SQL para actualizar el saldo del usuario de correo fletcher.flosi@yahoo.com acreditando \$25.000.
- **Paso 4:** Realizar una consulta SQL con el comando COMMIT para terminar la transacción.

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

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
});

pool.connect(async (error_conexion, client, release) => {
  // Paso 1
  await client.query("BEGIN");

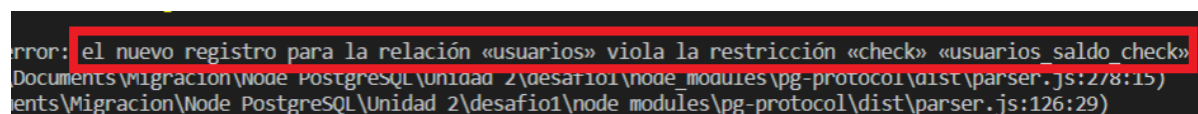
  // Paso 2
  const descontar =
    "UPDATE usuarios SET saldo = saldo - 25000 WHERE email = 'yuki_whobrey@aol.com' ";
  await client.query(descontar);

  // Paso 3
  const acreditar =
    "UPDATE usuarios SET saldo = saldo + 25000 WHERE email = 'fletcher.flosi@yahoo.com' ";
  await client.query(acreditar);

  // Paso 4
  await client.query("COMMIT");

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

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



```
error: el nuevo registro para la relación «usuarios» viola la restricción «check» «usuarios_saldo_check»
Documents\Migracion\node PostgreSQL\Unidad 2\desafio1\node_modules\pg-protocol\dist\parser.js:278:15
nts\Migracion\node PostgreSQL\Unidad 2\desafio1\node_modules\pg-protocol\dist\parser.js:126:29)
```

Imagen 2. Mensaje de error por consola.
Fuente: Desafío Latam

Como puedes notar hemos recibido un error indicando que el cambio que intentamos realizar viola la restricción del campo saldo.

¿Pero esto realmente funcionó? Utiliza tu consola de PostgreSQL y ejecuta la siguiente instrucción para revisar el saldo actual del usuario al que intentamos descontar saldo.

```
select * from usuarios where email = 'yuki_whobrey@aol.com';
```

Y deberás recibir lo que te muestro en la siguiente imagen:

```
clientes=# select * from usuarios where email = 'yuki_whobrey@aol.com';
               clientes
first_name | last_name | email | saldo
-----+-----+-----+-----
Yuki      | Whobrey  | yuki_whobrey@aol.com | 20000
(1 fila)
```

Imagen 3. Consulta a la tabla usuarios

Fuente: Desafío Latam

La restricción ha funcionado sin problema y no tenemos un saldo negativo en este usuario, pero ¿Qué pasó con el saldo del usuario acreditado en la transacción? Pues debería ser también el mismo previo a la transacción, pero si quieres comprobarlo ocupa la siguiente instrucción:

```
select * from usuarios where email = 'fletcher.flosi@yahoo.com';
```

Y deberás recibir lo que te muestro en la siguiente imagen:

```
clientes=# select * from usuarios where email = 'fletcher.flosi@yahoo.com';
               clientes
first_name | last_name | email | saldo
-----+-----+-----+-----
Fletcher   | Flosi     | fletcher.flosi@yahoo.com | 20000
(1 fila)
```

Imagen 4. Consulta a la tabla usuarios.

Fuente: Desafío Latam

Ahí lo tenemos, la transacción devolvió exactamente lo que esperábamos y hemos comprobado que ambos usuarios mantuvieron sus saldos.

Ejercicio propuesto (2)

Basado en el ejercicio de las transacciones del banco “Dinero Azul”, desarrolla una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera 30000 de saldo del usuario con correo **bette_nicka@cox.net** al usuario con correo **vinouye@aol.com**.

Ejercicio guiado: Usando transacciones SQL en Node, caso de éxito

Al comprobar que las transacciones con resultados fallidos no afectarán la consistencia de nuestra base de datos, sigue los siguientes pasos para hacer unas modificaciones al código anterior y realizar un caso de éxito, donde podamos comprobar que si se intenta transferir un saldo que sí está disponible en la cuenta del emisor, esta llegará sin problema a la cuenta receptora:

- **Paso 1:** Modificar la consulta SQL para actualizar el saldo del usuario de correo yuki_whobrey@aol.com descontando en este caso \$20.000. Agrega el comando RETURNING * al final de la consulta y almacena el resultado en una constante.
- **Paso 2:** Modificar la consulta SQL para actualizar el saldo del usuario de correo fletcher.flosi@yahoo.com acreditando en este caso \$20.000. Agrega el comando RETURNING * al final de la consulta y almacena el resultado en una constante.
- **Paso 3:** Imprimir ambas constantes creadas que representarán los registros afectados en ambas actualizaciones. Esto lo hacemos para visualizar los saldos actuales de ambos usuarios.

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

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
});

pool.connect(async (error_conexion, client, release) => {
  await client.query("BEGIN");

  // Paso 1
  const descontar =
    "UPDATE usuarios SET saldo = saldo - 20000 WHERE email = 'yuki_whobrey@aol.com' RETURNING *";
  const descuento = await client.query(descontar);

  // Paso 2
  const acreditar =
    "UPDATE usuarios SET saldo = saldo + 20000 WHERE email = 'fletcher.flosi@yahoo.com' RETURNING *";
  const acreditacion = await client.query(acreditar);

  // Paso 3
  console.log("Descuento realizado con éxito: ", descuento.rows[0]);
  console.log("Acreditación realizada con éxito: ",
acreditacion.rows[0]);

  await client.query("COMMIT");

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

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

```
Descuento realizado con éxito: {  
  first_name: 'Yuki',  
  last_name: 'Whobrey',  
  email: 'yuki_whobrey@aol.com',  
  saldo: '0'  
}  
Acreditación realizada con éxito: {  
  first_name: 'Fletcher',  
  last_name: 'Flosi',  
  email: 'fletcher.flosi@yahoo.com',  
  saldo: '40000'  
}
```

Imagen 5. Mensajes por consola representando luego de una transacción exitosa.

Fuente: Desafío Latam

Excelente! El saldo del usuario Yuki se redujo a \$0 puesto que inicialmente se le asignó la misma cantidad recientemente transferida, y el usuario Fletcher ahora tiene \$40.000, comprobando que la transacción hizo su trabajo sin problemas.

Ejercicio propuesto (3)

Basado en el ejercicio de las transacciones del banco “Dinero Azul”, desarrollar una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera \$10.000 de saldo del usuario con correo **albina@glick.com** al usuario con correo **shawna_palaspas@palaspas.org**.

Imprimir por consola el resultado de ambas consultas SQL para demostrar que los saldos fueron modificados.

Captura de errores en transacciones

Competencia

- Codificar un programa capturando errores de transacciones SQL de base de datos utilizando el entorno Node.

Introducción

Aunque las transacciones en base de datos cuentan con múltiples mecanismos desarrollados a lo largo del tiempo para garantizar y validar el flujo de datos, esto no quiere decir que estén exentas de no funcionar debido a que una consulta falle, que la conexión con la base de datos se vea interrumpida, entre otras. El control de estos diferentes posibles escenarios que puedan acontecer en una transacción es fundamental en las aplicaciones que hacen uso de esta funcionalidad.

En este capítulo, veremos cómo capturar los errores generados en consecuencia de la ejecución de transacciones. Para esto, desarrollaremos aplicaciones que sean capaces de poder controlar dichas incidencias y poder actuar de mejor manera, lo cual nos permitirá tener las herramientas necesarias para que nuestros desarrollos cuenten con mecanismos proactivos y mejor preparados frente a eventuales errores, como fallos de conexión, uso incorrecto de datos, instrucciones SQL defectuosas, entre otras.

Errores en transacciones SQL desde Node

En el capítulo anterior pudimos realizar transacciones SQL desde Node y comprobar su efectividad, no obstante cuando la transacción falló, la lectura del código se detuvo y el error se manifestó por terminal, ¿Cuál es el problema con esto? El problema es que el error no está siendo impreso por consola de forma controlada, sino que nuestra aplicación se estrelló con el error (conocido en inglés como crashed) y esto puede traer varias consecuencias, siendo la más importante el hecho de que al ser detenida la lectura del código, no se alcanza a ejecutar la instrucción del método “end()”, por lo que no cierra la conexión con PostgreSQL, dejando un tiempo muerto de espera, es decir, que todo quedará detenido y solo nos queda acabar con el proceso cancelando en la terminal.

Pero entonces ¿Cómo puedo capturar el error en una transacción? Hay que sacar el máximo provecho de nuestro lenguaje JavaScript y entre sus cientos de herramientas nos ofrece la sentencia “try catch”, para capturar errores que puedan ocurrir dentro de un bloque de códigos y poder manipular el error para imprimirlo por consola o enviarlo a una aplicación cliente, pensando a futuro si quisiéramos servir esta funcionalidad a través de un servidor.

Ejercicio guiado: Capturando errores en transacciones SQL desde Node

Ahora que sabemos que podemos usar try catch para capturar el error que puede suceder en una transacción, la siguiente pregunta podría ser ¿En dónde escribirlo? Para esto sigue los pasos donde modificaremos la aplicación con la transacción que intentamos realizar en el capítulo anterior, en este caso estará acompañada del try catch para capturar el error.

- **Paso 1:** Iniciar el bloque del try justo antes de iniciar la transacción.
- **Paso 2:** Luego de la consulta SQL con el comando COMMIT cierra el bloque del try e inicia el bloque catch recibiendo como parámetro el error.
- **Paso 3:** Dentro del bloque catch ejecuta una consulta SQL con el comando ROLLBACK para cerrar la transacción.
- **Paso 4:** Imprimir por consola el error disponible como parámetro de la sentencia catch.

```
pool.connect(async (error_conexion, client, release) => {  
  // Paso 1  
  try {  
    await client.query("BEGIN");  
    const descontar =  
      "UPDATE usuarios SET saldo = saldo - 25000 WHERE email =  
'yuki_whobrey@aol.com' ";  
    await client.query(descontar);  
  
    const acreditar =  
      "UPDATE usuarios SET saldo = saldo + 25000 WHERE email =  
'fletcher.flosi@yahoo.com' ";  
    await client.query(acreditar);  
  
    await client.query("COMMIT");  
    // Paso 2  
  } catch (e) {  
    // Paso 3  
    await client.query("ROLLBACK");  
  
    // Paso 4  
    console.log(e);  
  }  
  
  release();  
  pool.end();  
});
```

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

```
$ node transacciones.js
error: el nuevo registro para la relación «usuarios» viola la restricción «check» «usuarios_saldo_check»
    at Parser.parseErrorMessage (C:\Users\Brian\Documents\Migracion\node-postgresql\Unidad 2\desafio1\node_modules\node-postgresql\lib\parser.js:271:9)
    at Parser.handlePacket (C:\Users\Brian\Documents\Migracion\node-postgresql\Unidad 2\desafio1\node_modules\node-postgresql\lib\parser.js:186:23)
    at Parser.parse (C:\Users\Brian\Documents\Migracion\node-postgresql\Unidad 2\desafio1\node_modules\node-postgresql\lib\parser.js:102:29)
    at Socket.<anonymous> (C:\Users\Brian\Documents\Migracion\node-postgresql\Unidad 2\desafio1\node_modules\node-postgresql\lib\client.js:252:14)
    at Socket.emit (events.js:315:20)
    at addChunk (_stream_readable.js:295:12)
    at readableAddChunk (_stream_readable.js:271:9)
    at Socket.Readable.push (_stream_readable.js:212:10)
    at TCP.onStreamRead (internal/stream_base_commons.js:186:23) {
  length: 347,
  severity: 'ERROR',
  code: '23514',
  detail: 'La fila que falla contiene (Yuki, Whobrey, yuki_whobrey@aol.com, -5000).',
  hint: undefined,
  position: undefined,
  internalPosition: undefined,
  internalQuery: undefined,
  where: undefined,
  schema: 'public',
  table: 'usuarios',
  column: undefined,
  dataType: undefined,
  constraint: 'usuarios_saldo_check',
  file: 'd:\\pginstaller_13.auto\\postgres.windows-x64\\src\\backend\\executor\\execmain.c',
  line: '2013',
  routine: 'ExecConstraints'
}
```

Imagen 6. Mensaje por consola con el error de la transacción en forma de objeto.

Fuente: Desafío Latam

Como puedes notar, ahora estamos recibiendo en formato de objeto el error provocado por la transacción. ¿Pero de qué me sirve esto? Sirve bastante porque si observamos con detención, el objeto nos entrega diferentes propiedades importantes que detallan y describen el error sucedido.

Ahora pudiéramos por ejemplo desmenuzar este objeto y preparar diferentes mensajes que expliquen lo que sucedió y tal vez lo más importante, el código de error que interpreta esta situación. Sigue los siguientes pasos para modificar el bloque catch y especificar por consola diferentes detalles del error:

- **Paso 1:** Imprimir por consola el código de error con la propiedad “code”.
- **Paso 2:** Imprimir por consola el detalle del error “detail” .
- **Paso 3:** Imprimir por consola el nombre de la tabla donde se originó el error con la propiedad “table”.
- **Paso 4:** Imprimir por consola el campo donde está declarada la restricción con la propiedad “constraint”.

```
pool.connect(async (error_conexion, client, release) => {
  await client.query("BEGIN");

  try {
    const descontar =
      "UPDATE usuarios SET saldo = saldo - 25000 WHERE email =
'yuki_whobrey@aol.com' ";
    await client.query(descontar);

    const acreditar =
      "UPDATE usuarios SET saldo = saldo + 25000 WHERE email =
'fletcher.flosi@yahoo.com' ";
    await client.query(acreditar);

    await client.query("COMMIT");
  } catch (e) {
    await client.query("ROLLBACK");
    // Paso 1
    console.log("Error código: " + e.code);
    // Paso 2
    console.log("Detalle del error: " + e.detail);
    // Paso 3
    console.log("Tabla originaria del error: " + e.table);
    // Paso 4
    console.log("Restricción violada en el campo: " + e.constraint);
  }

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

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

```
$ node transacciones.js
Error código: 23514
Detalle del error: La fila que falla contiene (Yuki, Whobrey, yuki_whobrey@aol.com, -5000).
Tabla originaria del error: usuarios
Restricción violada en el campo: usuarios_saldo_check
```

Imagen 7. Mensaje detallado por consola de los detalles del error de la transacción.

Fuente: Desafío Latam

Ahí lo tenemos, ahora podemos presentar los detalles del error porque podemos controlarlo gracias al try catch. Esto será crucial de considerar cuando desarrolles aplicaciones grandes que puedan enfrentarse a diferentes errores.

Ejercicio propuesto (4)

Basado en el ejercicio de las transacciones del banco “Dinero Azul”, desarrolla una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera \$50.000 de saldo del usuario con correo **willard@hotmail.com** al usuario con correo **mroyster@royster.com**.

Imprimir por consola el código, detalle, tabla originaria y el nombre del campo que tiene la restricción que impidió que culminara la transacción.

Resumen

A lo largo del capítulo aprendimos a manejar cursores sobre una base de datos y cómo esto nos servirá para evitar consumos altos de memoria. También aprendimos a realizar operaciones transaccionales SQL, mediante la librería pg y capturar errores en las transacciones SQL de una base de datos utilizando el entorno Node, abordamos lo siguiente:

- Cursores en SQL y cómo utilizarlos desde Node a través del paquete pg-cursor para realizar consultas de una manera más óptima evitando el procesamiento innecesario de información.
- Importancia de las transacciones SQL en el desarrollo de software para proteger la consistencia de nuestras bases de datos.
- Usando transacciones SQL en Node en ejercicios con temática bancaria en donde se puso a prueba la efectividad de las transacciones a través de un caso de éxito y otro de fracaso
- Capturando errores en una transacción SQL usando try catch para evitar que la aplicación se estrelle.
- Imprimiendo por consola el detalle de un error ocurrido durante una transacción gracias a la manipulación del objeto que devuelve PostgreSQL como respuesta de una consulta SQL fallida.

Solución de los ejercicios propuestos

1. Basado en el ejercicio de los clientes del restaurante, desarrollar una aplicación en Node que realice una consulta SQL con el paquete pg-cursor para obtener e imprimir por consola los primeros 20 usuarios de la tabla **usuarios**.

```
const { Pool } = require("pg");
const Cursor = require("pg-cursor");
const config = {
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
};
const pool = new Pool(config);

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

  const consulta = new Cursor("select * from usuarios");
  const cursor = client.query(consulta);

  cursor.read(20, (err, rows) => {
    console.log(rows);
    cursor.close();
    release();
    pool.end();
  });
});
```

2. Basado en el ejercicio de las transacciones del banco "Dinero Azul", desarrollar una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera \$30000 de saldo del usuario con correo **bette_nicka@cox.net** al usuario con correo **vinouye@aol.com**.

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

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
});

pool.connect(async (error_conexion, client, release) => {
  await client.query("BEGIN");
  const descontar =
    "UPDATE usuarios SET saldo = saldo - 30000 WHERE email = 'bette_nicka@cox.net' ";
  await client.query(descontar);
  const acreditar =
    "UPDATE usuarios SET saldo = saldo + 30000 WHERE email = 'vinouye@aol.com' ";
  await client.query(acreditar);
  await client.query("COMMIT");

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

3. Basado en el ejercicio de las transacciones del banco “Dinero Azul”, desarrollar una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera \$10000 de saldo del usuario con correo **albina@glick.com** al usuario con correo **shawna_palaspas@palaspas.org**.

Imprimir por consola el resultado de ambas consultas SQL para demostrar que los saldos fueron modificados.

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

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
});

pool.connect(async (error_conexion, client, release) => {
  await client.query("BEGIN");

  const descontar =
    "UPDATE usuarios SET saldo = saldo - 20000 WHERE email = 'albina@glick.com' RETURNING *";
  const descuento = await client.query(descontar);

  const acreditar =
    "UPDATE usuarios SET saldo = saldo + 20000 WHERE email = 'shawna_palaspas@palaspas.org' RETURNING *";
  const acreditacion = await client.query(acreditar);

  console.log("Descuento realizado con éxito: ", descuento.rows[0]);
  console.log("Acreditación realizada con éxito: ",
acreditacion.rows[0]);

  await client.query("COMMIT");

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

- Basado en el ejercicio de las transacciones del banco “Dinero Azul”, desarrollar una aplicación en Node que realice consultas SQL para ejecutar una transacción que transfiera \$50.000 de saldo del usuario con correo **willard@hotmail.com** al usuario con correo **mroyster@royster.com**.

Imprimir por consola el código, detalle , tabla originaria y el nombre del campo que tiene la restricción que impidió que culminara la transacción


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

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "clientes",
  port: 5432,
});

pool.connect(async (error_conexion, client, release) => {
  await client.query("BEGIN");

  try {
    const descontar =
      "UPDATE usuarios SET saldo = saldo - 50000 WHERE email = 'willard@hotmail.com' ";
    await client.query(descontar);

    const acreditar =
      "UPDATE usuarios SET saldo = saldo + 50000 WHERE email = 'mroyster@royster.com' ";
    await client.query(acreditar);

    await client.query("COMMIT");
  } catch (e) {
    await client.query("ROLLBACK");
    console.log("Error código: " + e.code);
    console.log("Detalle del error: " + e.detail);
    console.log("Tabla originaria del error: " + e.table);
    console.log("Restricción violada en el campo: " + e.constraint);
  }

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