Guía de estudio 4 - Acceso a Base de Datos con Node (Parte II)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo recordar y repasar los contenidos que hemos visto en clase.

Donde aprenderemos a crear una API REST con funciones para modificar y eliminar registros en una base de datos PostgreSQL capturando los posibles errores.

Y en específico aprenderemos a:

- Agregar una ruta PUT en una API REST y utilizarla para modificar registros en una tabla alojada en PostgreSQL.
- Agregar una ruta DELETE en una API REST y utilizarla para eliminar registros en una tabla alojada en PostgreSQL.
- Capturar los posibles errores en una consulta SQL realizada con el paquete pg usando la sentencia try catch.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

API REST con PostgreSQL(PUT)

Continuando con el proyecto **"Plan de Viajes"** , crearemos un endpoint que nos permita modificar los datos de un viaje. Para lograrlo agregaremos la ruta **PUT /viajes/:id** y utilizaremos **req.params** y **req.query** para recibir el identificador del viaje y el valor del nuevo presupuesto y modificarlo.

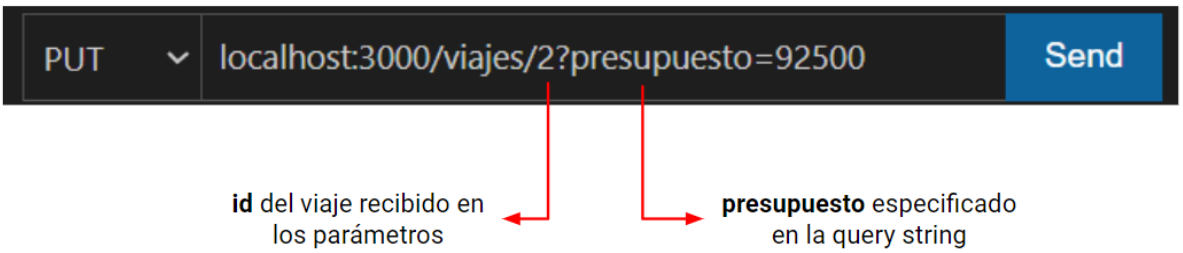

Imagen 1. Parámetros y query string en la consulta
Fuente: Desafío Latam

* query string es la parte de la URL que viene después del signo de interrogación

Lo primero será agregar en el archivo ***consultas.js*** una nueva función llamada ***modificarPresupuesto()*** que reciba en los argumentos el nuevo presupuesto y el id del viaje a modificar.

```
const modificarPresupuesto = async (presupuesto, id) => {
    const consulta = "UPDATE viajes SET presupuesto = $1 WHERE id = $2"
    const values = [presupuesto, id]
    const result = await pool.query(consulta, values)
}
```

Debes incluir en las exportaciones al final del archivo esta función para poder ser reconocida en el servidor.

```
module.exports = { agregarViaje, obtenerViajes, modificarPresupuesto }
```

Entre los archivos de esta unidad encontrarás 2 archivos "Apoyo Lectura" con las aplicaciones que hicimos en la clase anterior referente a los viajes y al equipamiento

Ahora en el archivo del servidor ***index.js*** debemos agregar en las importaciones la función ***modificarPresupuesto***

```
const { agregarViaje, obtenerViajes, modificarPresupuesto }
= require('./consultas')
```

Y al final del código agregar la ruta PUT /viajes/:id

```
app.put("/viajes/:id", async (req, res) => {
    const { id } = req.params
    const { presupuesto } = req.query
    await modificarPresupuesto(presupuesto, id)
    res.send("Presupuesto modificado con éxito")
})
```

Para probar nuestra nueva ruta, usemos Thunder Client para emitir la consulta modificando el presupuesto del viaje a Valdivia declarando en los parámetros el id 1, y en la query string el nuevo presupuesto.

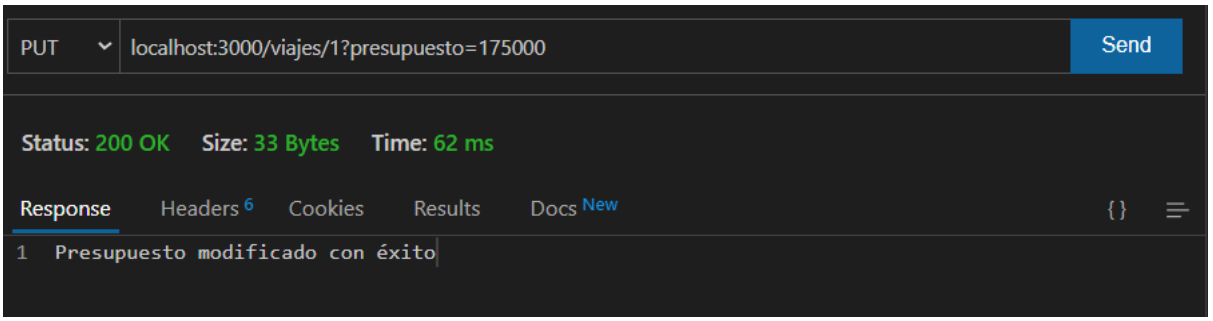

Imagen 2. Probando la ruta **PUT**
Fuente: Desafío Latam

Ahora revisemos los registros de la tabla viajes haciendo una consulta GET a nuestro servidor para verificar que el presupuesto de Valdivia fue modificado con éxito.

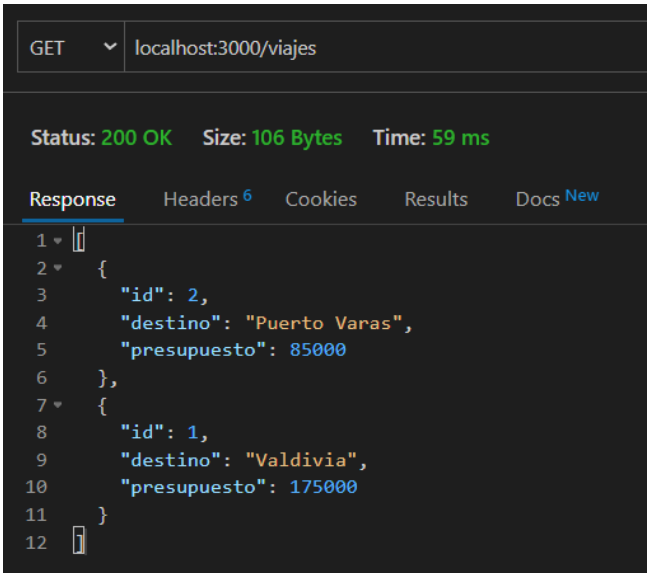

Imagen 3. Verificando que el presupuesto de un viaje se modificó
Fuente: Desafío Latam

¡Manos a la obra! - PUT

En el archivo ***equipamiento.js:***

1. Agrega una nueva función llamada ***modificarNombre*** que reciba como argumentos el nuevo nombre y el id del registro a modificar y realice la consulta SQL

2. Incluye esta función en las exportaciones del archivo *equipamiento.js*

Luego, en el archivo ***server.js:***

1. Agrega en las importaciones la función ***modificarNombre***

2. Crea una ruta **PUT /equipamientos/:id** que reciba el nuevo nombre como en la query string y el id como parámetro de la ruta y utilice la función para modificar el nombre de un equipamiento.

API REST con PostgreSQL(DELETE)

Para eliminar registros en nuestra base de datos desde nuestra aplicación debemos agregar una ruta con el método **DELETE**.

Esta ruta solo necesitará recibir el **id** del viaje a eliminar como parámetro en la URL

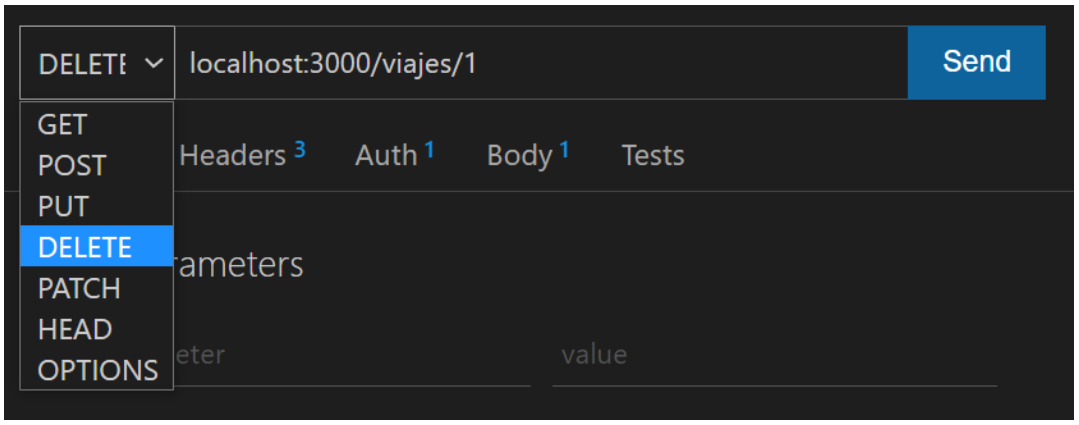

Imagen 4. Preparación de una consulta **DELETE**
Fuente: Desafío Latam

Agreguemos en el archivo ***consultas.js*** una nueva función llamada ***eliminarViaje()*** que reciba como argumento el id del viaje a eliminar.

```
const eliminarViaje = async (id) => {
    const consulta = "DELETE FROM viajes WHERE id = $1"
    const values = [id]
```

```
    const result = await pool.query(consulta, values)
}
```

Debes incluir en las exportaciones al final del archivo esta función para poder ser reconocida en el servidor.

```
module.exports =
{ agregarViaje, obtenerViajes, modificarPresupuesto, eliminarViaje }
```

Ahora en el archivo del servidor ***index.js*** debemos agregar en las importaciones la función **eliminarViaje**

Y al final del código agregar la ruta ***PUT /viajes/:id***

```
app.delete("/viajes/:id", async (req, res) => {
    const { id } = req.params
    await eliminarViaje(id)
    res.send("Viaje eliminado con éxito")
})
```

Probemos nuestra nueva ruta realizando otra consulta con Thunder Client que solicite eliminar el viaje de **Valdivia** cuyo id es el **1**

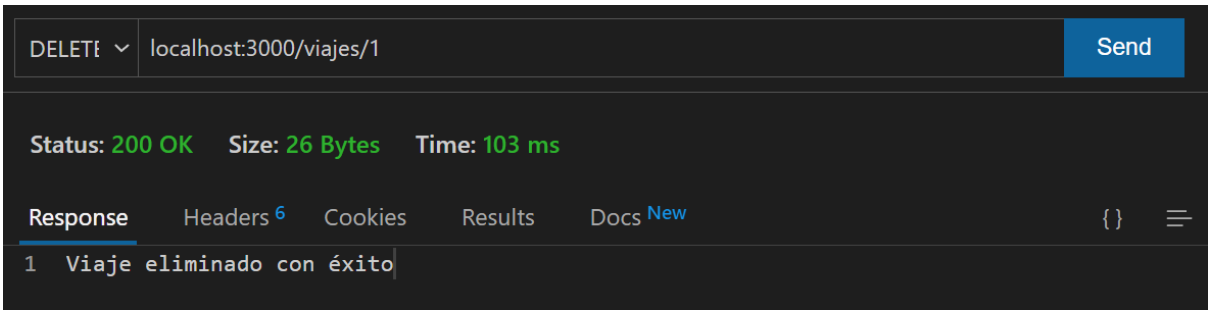

Imagen 5. Consulta **DELETE** para eliminar un viaje
Fuente: Desafío Latam

Si revisamos nuevamente los registros de la tabla ***viajes*** haciendo una consulta **GET** a nuestro servidor podremos verificar que el viaje a Valdivia ya no está.

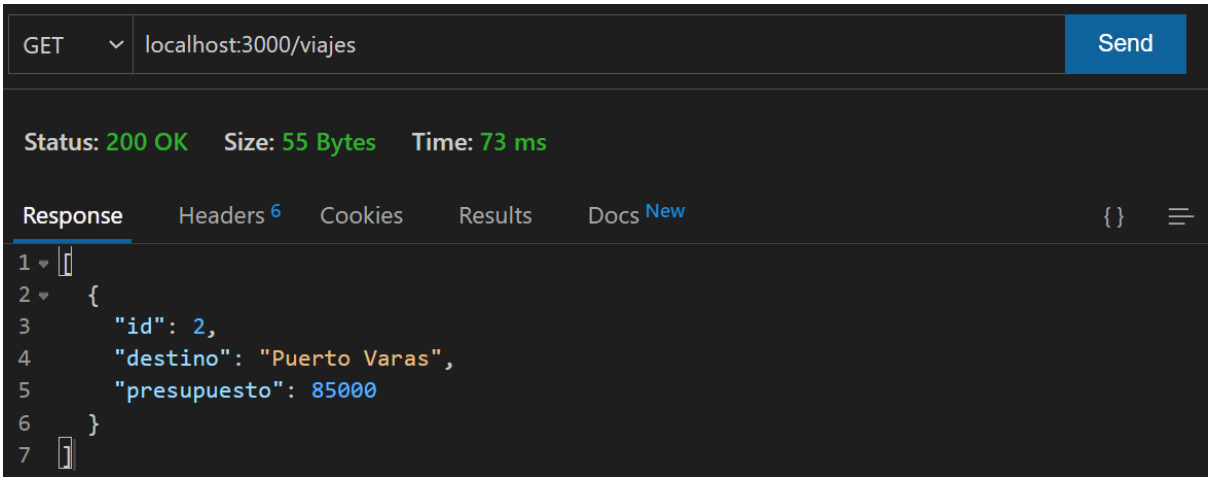

Imagen 6. Verificando que el viaje a **Valdivia** se eliminó
Fuente: Desafío Latam

¡Manos a la obra! - DELETE

En el archivo ***equipamiento.js:***

1. Agrega una nueva función llamada ***eliminarEquipamiento*** que reciba como argumento el id del equipamiento a eliminar

2. Incluye esta función en las exportaciones del archivo *equipamiento.js*

Luego, en el archivo ***server.js:***

1. Agrega en las importaciones la función ***eliminarEquipamiento***

2. Crea una ruta **DELETE /equipamientos/:id** que reciba el id del equipamiento como parámetro en la ruta y utilice la función para eliminar el registro de la base de datos.

Captura de errores

Durante una consulta SQL pueden ocurrir diversos errores que representen diferentes situaciones.

Por mencionar solo algunas es posible que:

1. No existan registros en una tabla.

    a. Suponiendo que existe solo 1 registro y 2 usuarios diferentes realizan simultáneamente la eliminación del mismo registro, la segunda consulta debería notificar que el registro no existe

2. Se intenta registrar un valor en un formato que no es compatible con un campo de la tabla

3. Existe un error en la sintaxis de la consulta SQL
4. Ningún registro fue modificado
5. Se viola una restricción definida en la tabla
    a. Se asigna un valor negativo en un campo que solo puede tener valores mayor o igual a cero

Cada una de estas situaciones son representadas por un código en específico y lo profundizaremos en la guía

Podemos capturar un error producido en la consulta SQL utilizando la sentencia try catch

```
app.method(<path>, (req,res) => {
    try {
        // función que emite una consulta SQL
    } catch (error) {
        // Captura del error producido
    }
})
```

En donde en el bloque **try** escribimos toda la lógica que idealmente se ejecutará sin problemas en la consulta, mientras que en el bloque **catch** escribimos toda la lógica que se deberá ejecutar en caso de que algún problema ocurra.

Pongamos a prueba la captura de errores modificando la ruta **POST /viajes** que creamos en la unidad anterior.

```
app.post("/viajes", async (req, res) => {
    try {
        const { destino, presupuesto } = req.body
        await agregarViaje(destino, presupuesto)
        res.send("Viaje agregado con éxito")
    } catch (error) {
        res.status(500).send(error)
    }
})
```

En el bloque catch usamos el objeto response(**res**) para devolver el error junto con un código de estado http 500 que se describe como **Internal Server Error.**

Ahora realicemos una consulta con Thunder Client en donde solo enviamos un objeto con el atributo destino pero **no** con el atributo presupuesto

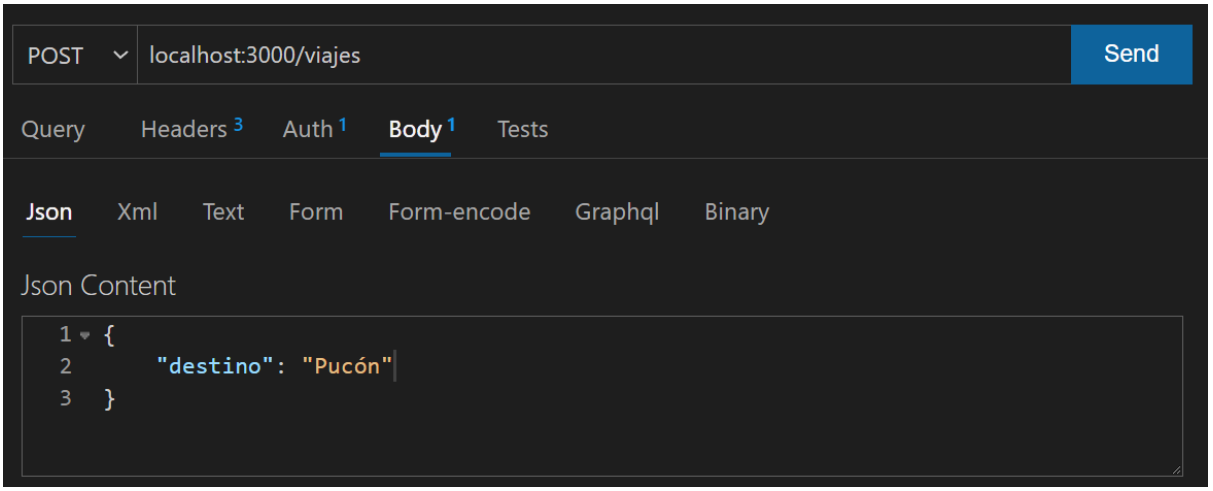

Imagen 7. Provocando la captura de un error a partir de una consulta **POST**
Fuente: Desafío Latam

Esta consulta nos devolverá un error puesto que todos los campos de la tabla deben contener un valor distinto de **Null**

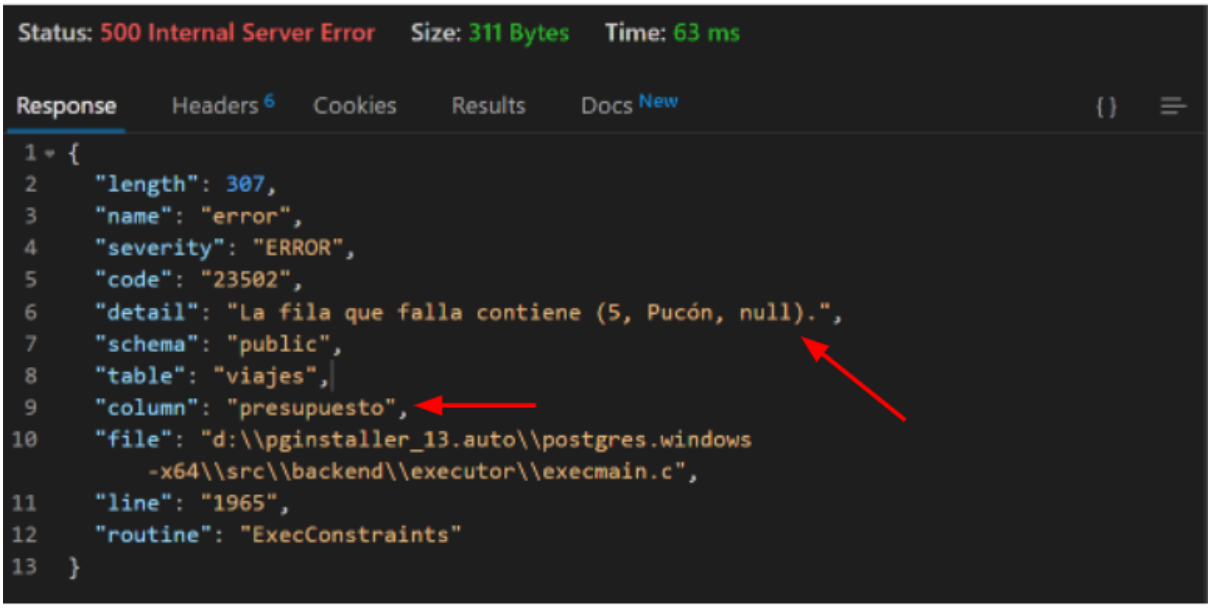

Imagen 8. Objeto del error
Fuente: Desafío Latam

El error queda en este objeto y contiene el detalle producido por PostgreSQL referente a lo sucedido.

¡Manos a la obra! - Ruta POST

En el archivo ***server.js:***

1. Agrega en la ruta **POST** captura de errores usando la sentencia **try catch**

2. Prueba la ruta **POST** enviando un objeto vacío y verificando que se recibe un objeto de error como respuesta de la consulta

Personalizando los errores

Entendiendo que los errores aparecen durante una operación y que éstos representan una situación en específico, podríamos ser más informativos al momento de responderle a un cliente que ocurrió un problema durante su consulta.

Códigos HTTP equivalentes

En PostgreSQL existe una tabla de diversos códigos de errores que podemos utilizar e interpretar para personalizar nuestras respuestas ante un error.

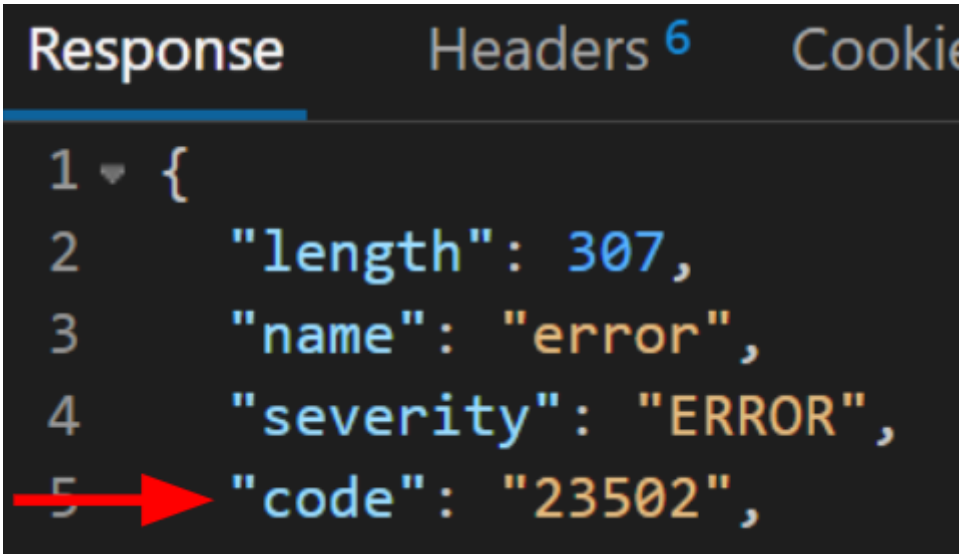

Imagen 9. Código de error dentro del objeto de error
Fuente: Desafío Latam

El código que se muestra en la imagen anterior (23502) representa una violación a una restricción **NOT NULL** en uno de los campos de la tabla y se produjo porque la aplicación cliente nos envió el cuerpo de su consulta incompleto.

En este caso el código HTTP que corresponde devolver al cliente sería el 400 (Bad Request).

Para adaptar a nuestro servidor a esta personalización del error debemos modificar la función la ruta **POST /viajes** para ahora considerar y condicionar la respuesta del error al código recibido.

```
app.post("/viajes", async (req, res) => {
    try {
        const { destino, presupuesto } = req.body
        await agregarViaje(destino, presupuesto)
        res.send("Viaje agregado con éxito")
    } catch (error) {
        const { code } = error
        if (code == "23502") {
            res.status(400)
                .send("Se ha violado la restricción NOT NULL en uno de los campos de la tabla")
        } else {
            res.status(500).send(error)
        }
    }
})
```

De esta manera si emitimos de nuevo la consulta **POST** sin especificar un presupuesto obtendremos lo siguiente:

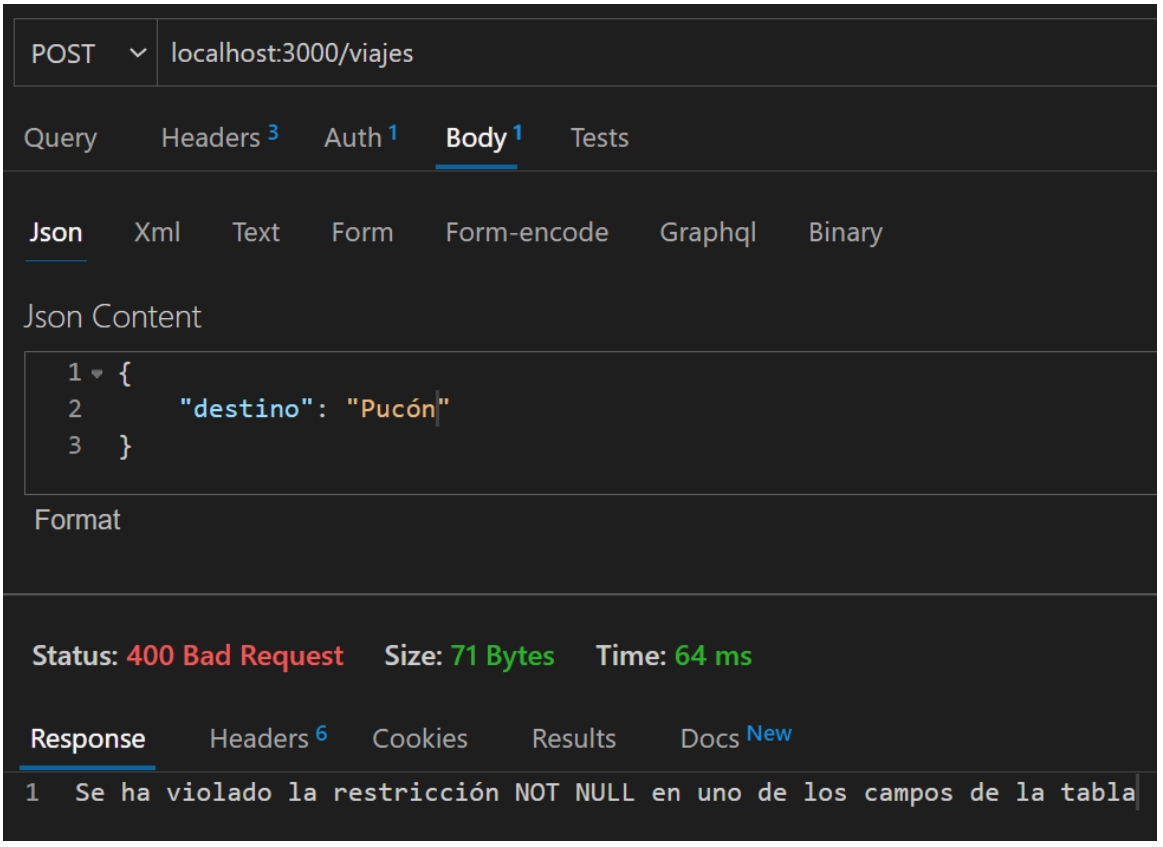

Imagen 10. Error HTTP equivalente
Fuente: Desafío Latam

Con esto el cliente sabrá que el error fue provocado por el formato de su consulta, logrando una mejor comunicación entre nuestra aplicación backend y la aplicación frontend.

¡Manos a la obra! - Condicional en la ruta POST

En el archivo ***server.js:***

1. Agrega un condicional en la ruta **POST /equipamientos** que evalúe si el código de error de PostgreSQL es igual a **23502**
2. Devuelve un código de estado HTTP 400 indicando que se ha violado la restricción NOT NULL en la base de datos.
3. Prueba nuevamente la ruta enviando un equipamiento incompleto o un objeto vacío

Provocando el código 404 (Not Found)

En el objeto **result** de cada consulta podemos acceder al atributo ***rowCount*** para conocer cuántas filas estamos obteniendo o han sido afectadas por la consulta. PostgreSQL no considera como un error la devolución de 0 filas, sin embargo en la comunicación cliente - servidor es de gran valor notificar desde un servidor que no se encontró ningún recurso.

Veamos un caso de uso modificando la función ***modificarPresupuesto*** haciendo uso del **Throw** para emitir una excepción en caso de que el valor del ***rowCount*** sea **0**.

```
const modificarPresupuesto = async (presupuesto, id) => {
    const consulta = "UPDATE viajes SET presupuesto = $1 WHERE id = $2"
    const values = [presupuesto, id]
    const { rowCount } = await pool.query(consulta, values)
    if (rowCount === 0) {
        throw { code: 404, message: "No se consiguió ningún viaje con este id" }
    }
}
```

En caso de cumplirse el condicional, le indicaremos a la ruta que ocurrió una excepción representado por un objeto con el código HTTP y la descripción a devolver.

La ruta **PUT /viajes/:id** entonces deberá estar preparada para capturar esta excepción de la siguiente manera:

```
app.put("/viajes/:id", async (req, res) => {
    const { id } = req.params
    const { presupuesto } = req.query
    try {
```

```
        await modificarPresupuesto(presupuesto, id)
        res.send("Presupuesto modificado con éxito")
    } catch ({ code, message }) {
        res.status(code).send(message)
    }
})
```

Realicemos una consulta al servidor intentando modificar el presupuesto de un viaje de id **22**, el cuál no existe en nuestra tabla *viajes*.

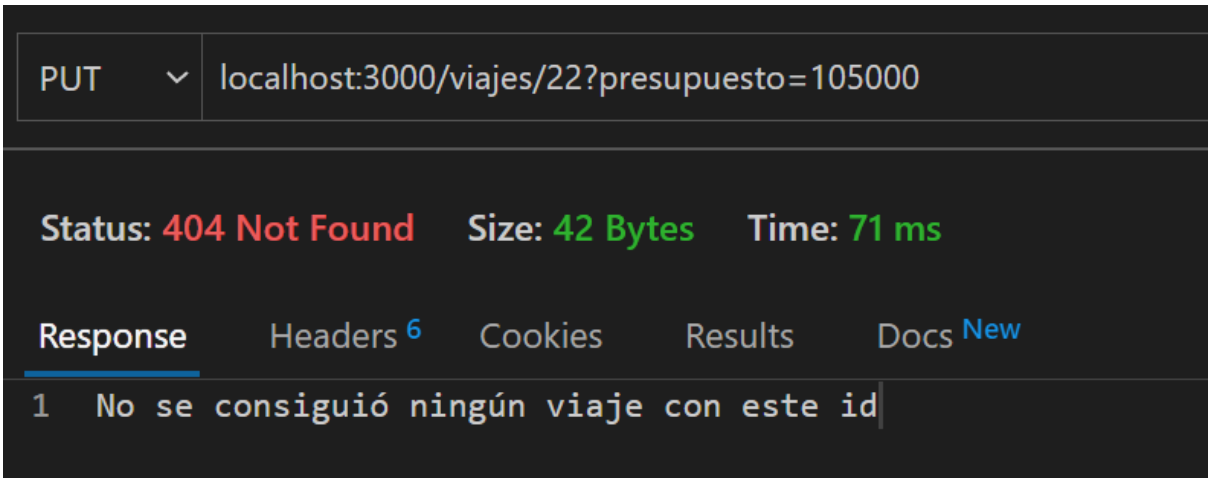

Imagen 11. Provocando el código 404
Fuente: Desafío Latam

¡Manos a la obra! - Provocando el código 404

En los archivos ***server.js*** y ***equipamiento.js:***

1. Realiza las modificaciones necesarias para considerar el ***rowCount*** obtenida en la función ***modificarNombre***

2. Devolver un código 404 al cliente de ruta **PUT /equipamientos/:id** en caso de no afectar ninguna fila.

Validaciones usando middlewares

Aunque es posible escribir todas las validaciones directamente en la lógica de cada ruta, esto generaría que nuestro archivo **index.js** termine siendo muy grande y más complicado de mantener.

Para evitar esto podemos optar por utilizar middlewares como capa de validación que tenga como objetivo confirmar la existencia de un recurso antes de intentar modificarlo o eliminarlo.

Los middleware son funciones que se ejecutan antes que el código definido en en el endpoint y comparten el acceso al objeto **request**, **response** y un argumento más llamado **next**.

El argumento **next** es una función que deberá ser ejecutada para que la consulta continúe de lo contrario el cliente quedará en espera. Para ver un ejemplo de un middleware procedamos a crear una nueva ruta **GET /viajes/:id** que tiene como objetivo devolver el registro de un viaje en específico.

```
app.get("/viajes/:id", async (req, res) => {
    const { id } = req.params
    const viaje = await obtenerViaje(id)
    res.json(viaje)
})
```

La función **obtenerViaje** también será necesaria crearla en el archivo ***consultas.js*** y hacer las exportaciones e importaciones correspondientes.

```
const obtenerViaje = async (id) => {
    const consulta = "SELECT * FROM viajes WHERE id = $1"
    const values = [id]
    const result = await pool.query(consulta, values)
    const [viaje] = result.rows
    return viaje
}
```

Ahora en el servidor creemos un middleware que tenga como objetivo reportar por la terminal una consulta recibida.

```
const reportarConsulta = async (req, res, next) => {
    const parametros = req.params
    const url = req.url
    console.log(`
    Hoy ${new Date()}
    Se ha recibido una consulta en la ruta ${url}
    con los parámetros:
    `, parametros)
    next()
}
```

Para agregar esta función como middleware, solo debemos incluirla entre los argumentos de la ruta de la siguiente manera:

```
app.get("/viajes/:id", reportarConsulta, async (req, res) => {
    const { id } = req.params
    const viaje = await obtenerViaje(id)
    res.json(viaje)
})
```

Ahora realicemos una consulta para obtener alguno de los viajes que conservemos en la base de datos utilizando thunderclient.

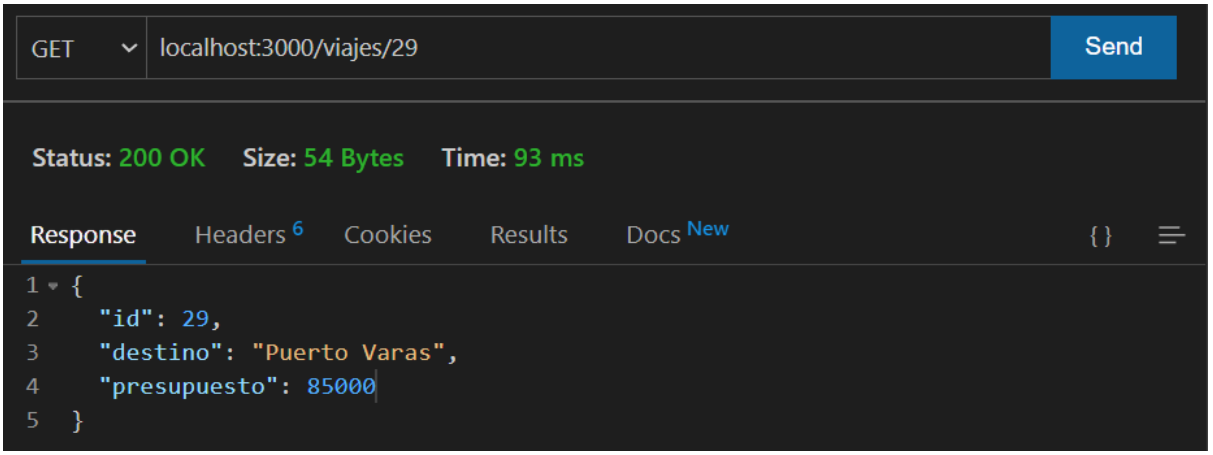

Imagen 12. Obteniendo un viaje por id
Fuente: Desafío Latam

Y si observamos la terminal, notaremos que se ha reportado la consulta realizada:

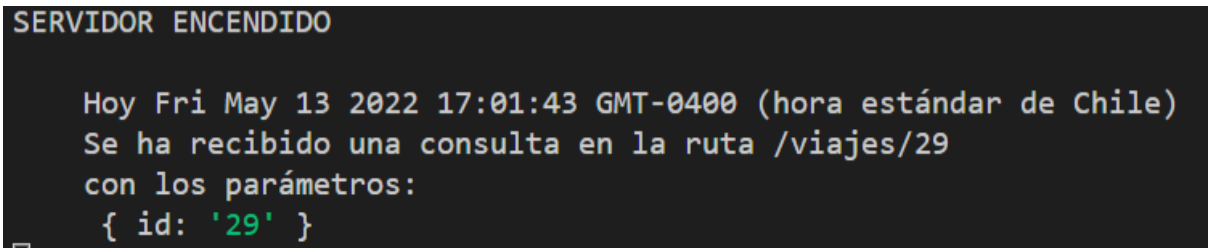

Imagen 13. Reporte de consulta por medio de un middleware
Fuente: Desafío Latam

De esta misma manera podemos hacer todas las validaciones que necesitemos y utilizar la captura de errores para permitir el consumo de recursos o entregar una respuesta al cliente con el código HTTP correspondiente a la situación.

Entre los archivos de esta unidad encontrarás un ejemplo de la API REST que hemos creado completa y basada en el proyecto **Plan de Viajes** que utiliza Middlewares en sus diferentes rutas como capa de validación y reporte.

Convenciones de endpoints

Parte frecuente de nuestro trabajo en el lado de backend será crear nuevos endpoints, a medida que un proyecto crece el número de endpoints también irá en aumento. Para mantener el orden del proyecto se recomienda seguir ciertas convenciones con el propósito de hacer más sencilla la mantención del proyecto.

En medida de lo posible los endpoints que creemos serán del tipo CRUD o sea create (crear), read (leer), update (actualizar) o delete (borrar).

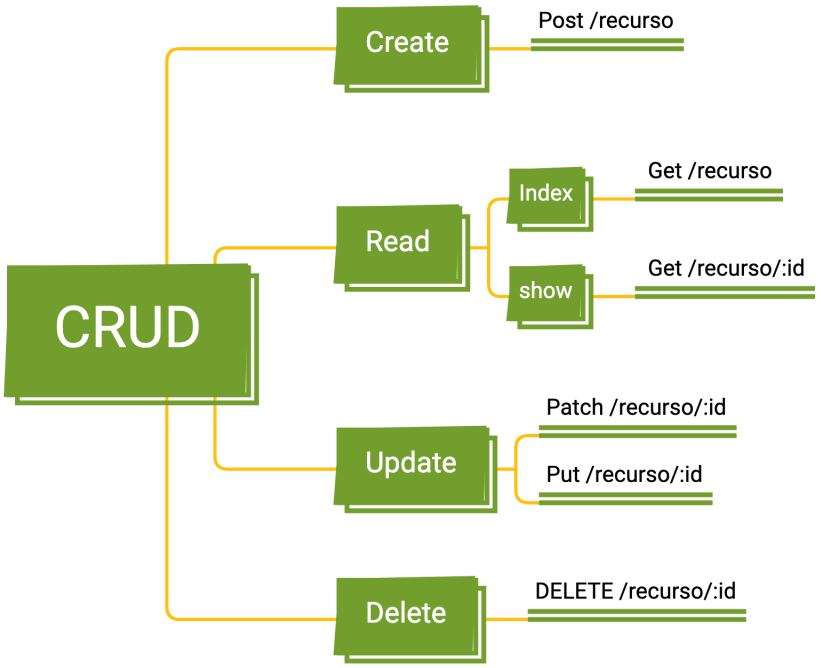

Imagen 14. Convenciones de endpoints
Fuente: Desafío Latam

Donde el recurso puede ser viajes, equipajes, usuarios o cualquier otro asociado a una tabla de la base de datos que estemos trabajando. En todos estos ejemplos nos estamos refiriendo a colecciones de elementos, es por esto mismo que se recomienda frecuentemente utilizar un sujeto plural.

Entonces si nos piden crear un endpoint para crear un usuario, deberíamos utilizar la ruta /usuarios con el método POST.

Al leer un recurso usualmente estaremos en dos escenarios, cuando nos piden todos los elementos, o cuando nos piden mostrar el detalle de uno solo.

- Si nos piden todos los elementos de un recurso, como por ejemplo todos los artículos de un blog deberíamos utilizar la ruta /articulos con el método GET
    - A veces también nos pedirán solo algunos elementos de una colección. En este caso también lo implementaremos como un index pero dentro del endpoint filtraremos o seleccionaremos los elementos utilizando algún parámetro enviado en el querystring, ejemplo: /articulos?favoritos=true
- Si nos piden el detalle de un artículo, en ese caso utilizaremos la ruta get /articulos/:id.
    - **Nota:** Hay quienes para este endpoint utilizan el sujeto en singular, lo importante es ser consistente a lo largo de nuestra API.

Al actualizar un recurso podemos ocupar el método patch o el método put. La convención más utilizada es utilizar patch cuando se cambian todos los elementos de la entidad y put cuando solo son algunos campos especificados.

Utilizando estas convenciones utilizarás menos tiempo pensando cómo nombrar cada endpoint y escoger el verbo adecuado y además será más simple mantener tu API.

Preguntas y repaso:

El siguiente código está incompleto. Asumiendo que la función eliminarViaje está implementada de la misma forma vista en la guía y elimina un elemento correctamente ¿Qué falta al siguiente endpoint para que funcione bien?

```
app.delete("/viajes", async (req, res) => {
    await eliminarViaje()
    res.send("Viaje eliminado con éxito")
})
```

Implementa control de errores y envía un error 500 al usuario en caso de que no se pueda ingresar un viaje

```
app.post("/viajes", async (req, res) => {
    const { destino, presupuesto } = req.body
    await agregarViaje(destino, presupuesto)
    res.send("Viaje agregado con éxito")
})
```

¡Manos a la obra! - Proyecto de ejemplo

Descarga el proyecto de ejemplo con Middlewares disponible entre los archivos de la unidad y replica esta buena práctica con la tabla de Equipamientos y los archivos **server.js** y **equipamiento.js**