



Acceso a Base de Datos con Node (Parte II)



**Activen las cámaras los que puedan y
pasemos asistencia**





Inicio

{desafío}
latam_



En esta unidad aprenderemos a crear una API REST con funciones para **modificar** y **eliminar** registros en una base de datos PostgreSQL capturando los posibles errores

Objetivos

*/** 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 **/*

Objetivos



Activación de conceptos



```
CREATE DATABASE plan_de_viajes;  
  
\c plan_de_viajes;  
  
CREATE TABLE viajes (id, destino, presupuesto);
```

¿Cuál es el problema con estas instrucciones para crear una tabla en una base de datos?



Activación de conceptos



```
const getDate = async () => {  
  const result = await pool.query("SELECT NOW()")  
  console.log(result)  
}
```

¿ Qué hace la función getDate ?



Activación de conceptos



```
const agregarViaje = async (destino, presupuesto) => {  
  const consulta = "INSERT INTO viajes values (DEFAULT, $1, $2)"  
  const result = await pool.query(consulta, values)  
  console.log("Viaje agregado")  
}
```

¿Cuál es el problema con este código?



Activación de conceptos



```
const agregarViaje = async (destino, presupuesto) => {  
  const consulta = "INSERT INTO viajes values (DEFAULT, $1, $2)"  
  const result = await pool.query(consulta, values)  
  console.log("Viaje agregado")  
}
```

¿Cuál es el problema con este código?



Activación de conceptos



¿Por qué debemos
parametrizar las consultas?



Activación de conceptos



```
const listarViajes = async (id) => {  
  const consulta = `SELECT * FROM viajes where id = ${id}`  
  const result = await pool.query(consulta)  
  console.log(result)  
}
```

¿Cómo debemos modificar el query anterior para evitar un problema de sql injection?



Desarrollo

{desafío}
latam_



/* API REST con PostgreSQL(PUT) */

Acceso a base datos con Node

API REST con PostgreSQL(PUT)

Continuando con el proyecto “**Plan de Viajes**” iniciado en la unidad anterior, 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.



id del viaje recibido en
los parámetros

presupuesto especificado
en la query string

Acceso a base datos con Node

API REST con PostgreSQL(PUT)

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.

Acceso a base datos con Node

API REST con PostgreSQL(PUT)

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

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

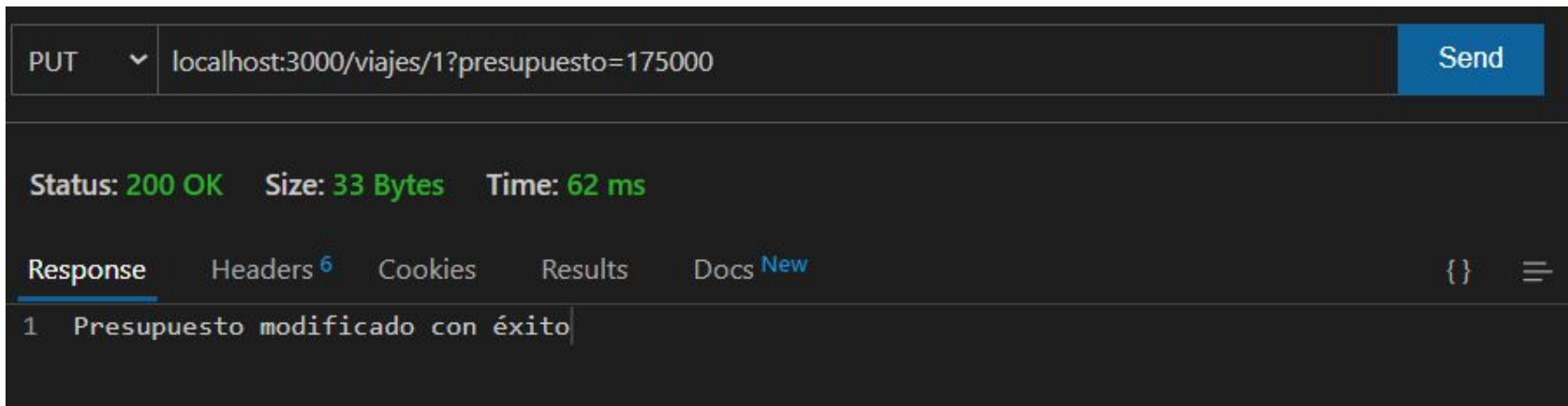
Y al final del código agregar la ruta **PUT /viajes/:id**

```
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")  
})
```

Acceso a base datos con Node

API REST con PostgreSQL(PUT)

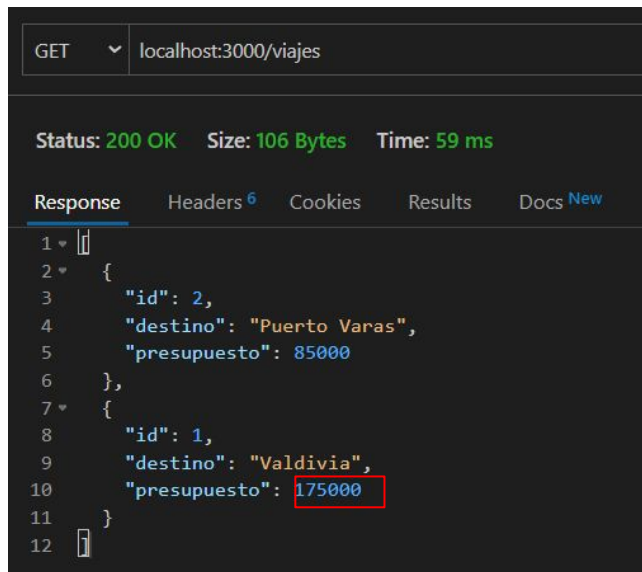
Para probar nuestra nueva ruta usamos Thunder Client para emitir la consulta modificando el presupuesto del viaje a Valdivia, declarando en los parámetros el id 1, y en la query string el nuevo presupuesto.



Acceso a base datos con Node

API REST con PostgreSQL(PUT)

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.



```
GET localhost:3000/viajes

Status: 200 OK   Size: 106 Bytes   Time: 59 ms

Response  Headers 6  Cookies  Results  Docs New

1  [
2  {
3    "id": 2,
4    "destino": "Puerto Varas",
5    "presupuesto": 85000
6  },
7  {
8    "id": 1,
9    "destino": "Valdivia",
10   "presupuesto": 175000
11 }
12 ]
```

Ejercicio

En el mismo proyecto 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.



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

Ejercicio ¡Manos al teclado!



*/** Agregar una ruta PUT en una API REST con Express para modificar registros en una tabla alojada en PostgreSQL **/* 

*/** Agregar una ruta DELETE en una API REST con Express para modificar 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 **/*

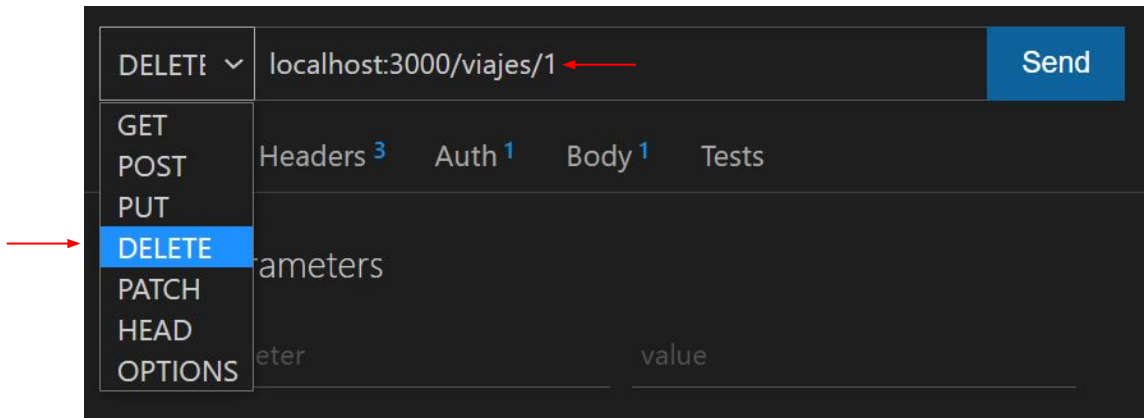
Objetivos

Acceso a base datos con Node

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



Acceso a base datos con Node

API REST con PostgreSQL(DELETE)

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 }
```

Acceso a base datos con Node

API REST con PostgreSQL(DELETE)

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

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

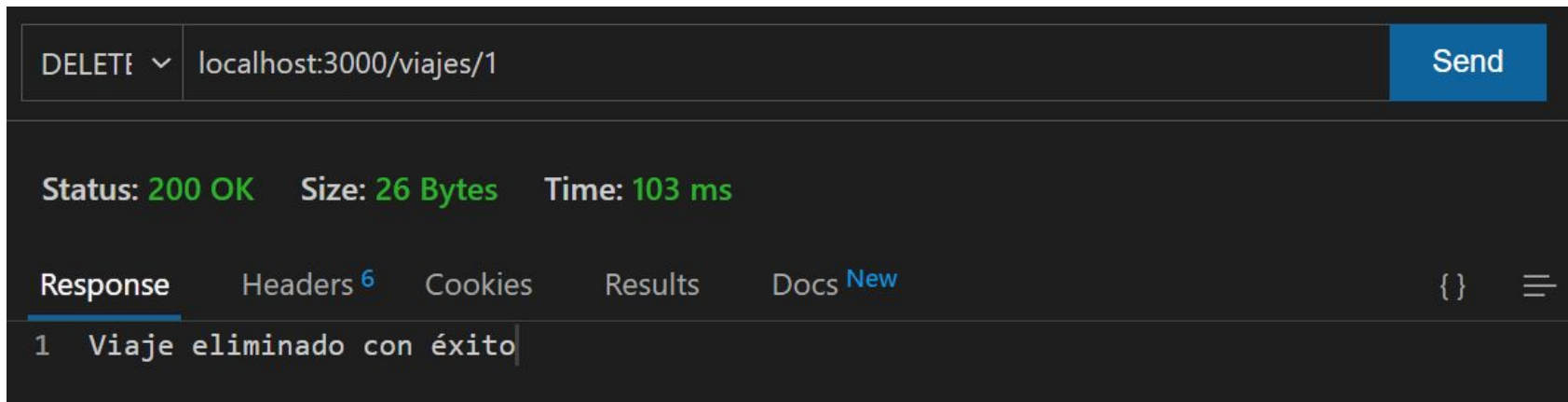
Y al final del código agregar la ruta **PUT /viajes/:id**

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


Acceso a base datos con Node

API REST con PostgreSQL(DELETE)

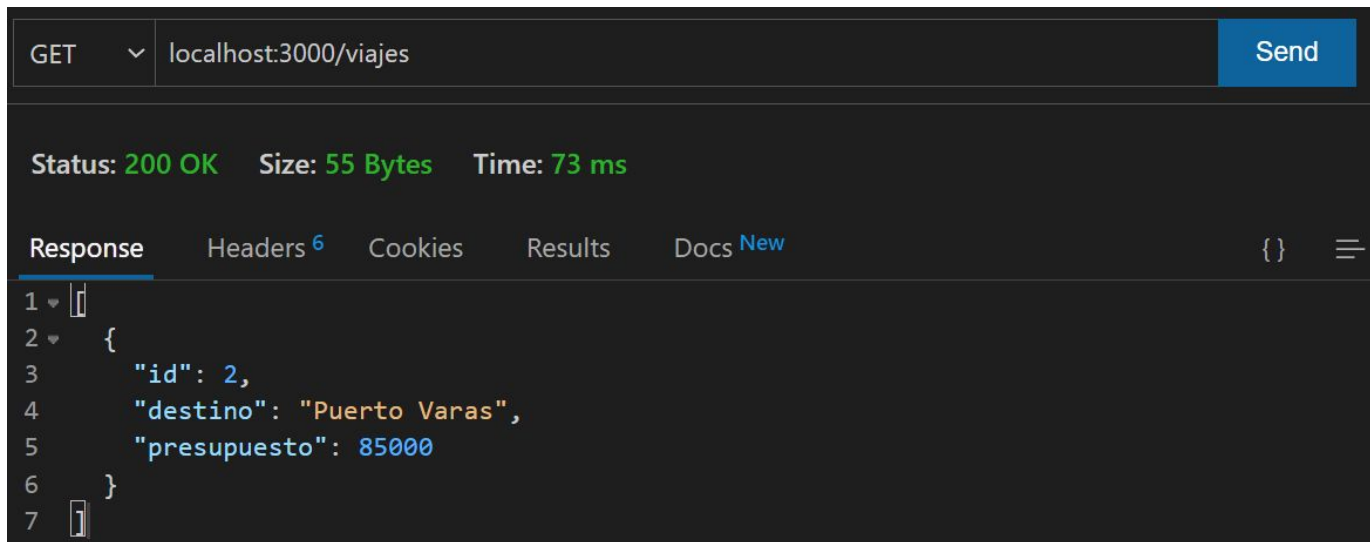
Probemos nuestra nueva ruta realizando otra consulta con Thunder Client que solicite eliminar el viaje de **Valdivia** cuyo id es el **1**



Acceso a base datos con Node

API REST con PostgreSQL(DELETE)

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.



```
GET localhost:3000/viajes Send

Status: 200 OK Size: 55 Bytes Time: 73 ms

Response Headers 6 Cookies Results Docs New {} ≡

1 {
2   {
3     "id": 2,
4     "destino": "Puerto Varas",
5     "presupuesto": 85000
6   }
7 }
```

Ejercicio

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.

Ejercicio ¡Manos al teclado!



*/** Agregar una ruta PUT en una API REST con Express para modificar registros en una tabla alojada en PostgreSQL **/* ✓

*/** Agregar una ruta DELETE en una API REST con Express para modificar 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 **/*

Objetivos

Acceso a base datos con Node

Captura de errores

Durante una consulta SQL pueden ocurrir diversos errores o situaciones especiales que tenemos que resolver.

Por mencionar solo algunas:

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.

Acceso a base datos con Node

Captura de errores

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 el bloque **try** escribimos toda la lógica que idealmente se ejecutará sin problemas en la consulta

En el bloque **catch** escribimos toda la lógica que se deberá ejecutar en caso de que algún problema ocurra

Acceso a base datos con Node

Captura de errores

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

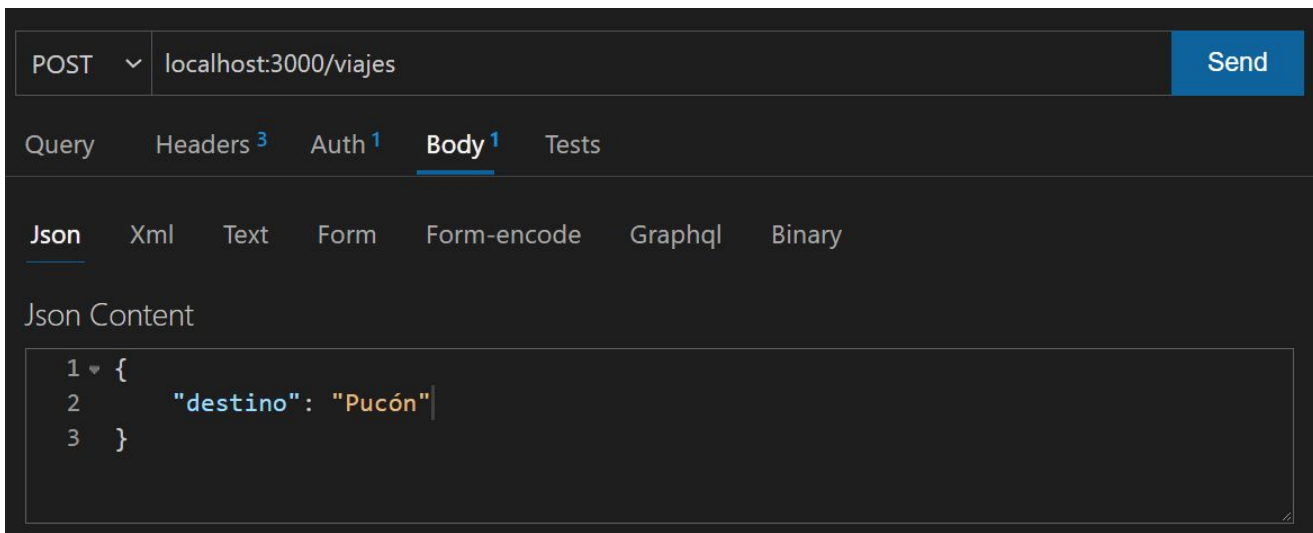
```
app.post("/viajes", async (req, res) => {  
  try {  
    const { destino, presupuesto } = req.body  
    await agregarViaje(destino, presupuesto)  
    res.send("Viaje agregado con éxito")  
  } catch (error) {  
    res.status(500).send(error)  
  }  
})
```

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**

Acceso a base datos con Node

Captura de errores

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



Acceso a base datos con Node

Captura de errores

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

```
Status: 500 Internal Server Error   Size: 311 Bytes   Time: 63 ms

Response   Headers 6   Cookies   Results   Docs New   {}   ≡

1 {
2   "length": 307,
3   "name": "error",
4   "severity": "ERROR",
5   "code": "23502",
6   "detail": "La fila que falla contiene (5, Pucón, null).",
7   "schema": "public",
8   "table": "viajes",
9   "column": "presupuesto",
10  "file": "d:\\pginstaller_13.auto\\postgres.windows
        -x64\\src\\backend\\executor\\execmain.c",
11  "line": "1965",
12  "routine": "ExecConstraints"
13 }
```

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

Ejercicio

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

Ejercicio ¡Manos al teclado!



*/** Agregar una ruta PUT en una API REST con Express para modificar registros en una tabla alojada en PostgreSQL **/* ✓

*/** Agregar una ruta DELETE en una API REST con Express para modificar 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 **/* ✓

Objetivos

Acceso a base datos con Node

Captura de errores

Una de las preguntas más comunes de este tema es: **¿En dónde debería escribir la captura de errores?**

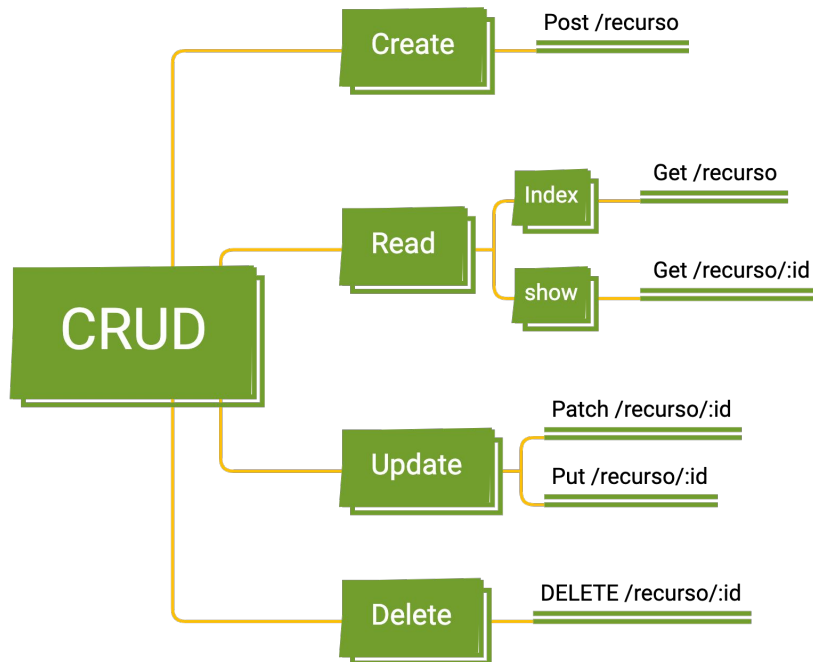
Esta capa de validación puede escribirse tanto en aplicaciones Frontend como Backend, no obstante, es recomendable que en ambas aplicaciones nos preocupemos por escribir la captura de errores.

Desde el backend capturamos un error y podemos preparar el detalle de lo sucedido como respuesta a una consulta emitida por una aplicación Frontend.

De igual manera, en una aplicación Frontend podemos preparar la lógica para reaccionar de una u otra manera en base a los errores capturados en pro de darle siempre una interacción a los usuarios finales.

Convenciones de endpoints

Cuando nos pidan construir un CRUD utilizaremos las siguientes rutas y métodos



Donde el recurso puede ser viajes, equipajes, usuarios o cualquier otro asociado a una tabla de la base de datos que estemos trabajando.

Puedes encontrar más detalles de estas convenciones en la guía de estudio.



Cierre

{desafío}
latam_



¿Existe algún concepto que no
hayas comprendido?

Reflexionemos

- Revisar la guía donde adicionalmente al contenido se aborda cómo identificar los tipos de errores, el formato en el que se entregan al cliente y utilizar middleware para realizar validaciones
- Revisar en conjunto el desafío.

¿Qué sigue?



*Academia de
talentos digitales*

www.desafiolatam.com



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam