Lectura - API REST (Parte II)

API REST

Competencia

- Construir una API REST básica utilizando el framework Express para resolver un problema acorde al entorno Node.

Introducción

Las API REST no son un tema nuevo para nosotros, las conocimos previamente cuando aprendimos a desarrollar con Node puro. Ahora que nos encontramos aprendiendo Express, debemos adaptar los conceptos que ya conocemos a las sintaxis de nuestro framework.

En este capítulo aprenderás a crear y utilizar una API REST en un servidor desarrollado con Express, para servir un canal de comunicación que permita conectar 2 aplicaciones creadas bajo la arquitectura cliente-servidor. Esta arquitectura está en todos lados, por ello conocerla y manejarla te permitirá modularizar aún más tus aplicaciones, separando completamente el modelo de la vista, manteniendo su comunicación e incluso creando enlaces de servidor a servidor en una petición reenviada de una API REST a otra.

API REST

Las API REST son arquitecturas basadas en el protocolo HTTP, creadas para ser usadas como un puente de comunicación entre aplicaciones independientemente de la tecnología en la que sean construidas. Son desarrolladas en el lado del servidor, a través de los métodos o verbos HTTP que disponibilizan diferentes funciones, las que comúnmente conllevan a la gestión de recursos alojados en algún motor de bases de datos, sin embargo, también sirven para emitir una consulta que represente una situación por parte de quien emite la petición. Es importante destacar que es totalmente posible que un servidor se comunique con otro a través de una API REST.

Una de las ideas principales de su creación es la construcción de estructuras lógicas reutilizables, dentro de estas se pueden encontrar los middlewares, que no son más que funciones que se ejecutan en cada consulta de la ruta a la que pertenecen.

Esta arquitectura es considerada un estándar mundial y destaca por ser consumida desde cualquier tecnología o lenguaje que realice comunicaciones por HTTP y maneje datos JSON o XML. Aunque desde hace varios años se ha establecido por excelencia que el formato de datos para estas comunicaciones idealmente es JSON, por su cómoda lectura y peso ligero en comparación con XML.

Endpoints

En pocas palabras, los endpoints son una ruta creada para el consumo de un recurso, para la activación de una función o middleware definida en la configuración de nuestro servidor. También es conocido como la suma de la URL base, el path declarado en la ruta y el método HTTP.

Un ejemplo claro puede ser la PokeApi que siendo una API pública, tiene definido diferentes endpoints que devuelven diversas respuestas. En este link encontrarás todos los endpoints disponibles en esta API.

¿Cómo se puede formar un endpoint? Su estructura es la siguiente:

```
<url base>/<path>
```

Como puedes ver un endpoint es la unión entre la URL base y el path de una ruta. El path comúnmente se crea con parámetros para la definición de un endpoints de recurso dinámico, es decir, que un endpoint puede ser variable y su respuesta dependerá de los parámetros que se declaren en la ruta.

Parámetros de los endpoints

Al momento de definir un path en una ruta, podemos establecer un espacio para el libre tipeo del usuario o la definición esperada por el servidor. Este espacio variable se considera un parámetro.

Si retomamos el ejemplo de la pokeapi, podemos obtener los datos de un solo pokémon declarando al final del path el número identificador o el nombre directamente, por ejemplo:

https://pokeapi.co/api/v2/**pokemon/pikachu**

Siendo la palabra "pokémon" una definición en la ruta y "pikachu" un valor variable, nos encontramos con el parámetro del endpoint.

En Express manipulamos los parámetros de las rutas entre las propiedades del objeto request, específicamente la propiedad "params". Por ejemplo, si suponemos que pokeapi fue construida con Express, una ruta hipotética para este endpoint sería algo parecido al siguiente código:

```
app.get('/pokemon/:id', (req, res) => {
 const { id } = req.params
 ...
 res.send(pokemon)
})
```

En donde se tiene creada una ruta **GET /pokemon/:id** que devuelve como respuesta el JSON correspondiente al pokémon solicitado.

El uso más típico que encontramos en la construcción de una API REST, es servir como apoyo de un sistema tipo CRUD (Create Read Update Delete) a través de los métodos HTTP POST, READ, UPDATE y DELETE. En esta lectura construiremos progresivamente una API REST para el manejo de datos alojados en PostgreSQL, por supuesto, con el paquete pg, pero antes debemos tener lista nuestra arquitectura base.

Ejercicio guiado: Una API REST con Express

Construir una API REST que disponibilice los 4 métodos básicos HTTP para servir el backend de una aplicación cliente. La temática de esta API REST tratará de la gestión de canales de televisión que inicialmente estarán alojados como un arreglo de objetos en el servidor, pero posteriormente serán almacenados en PostgreSQL y a través del paquete pg haremos las consultas correspondientes para cumplir con las funcionalidades CRUD, este será un ejercicio progresivo que estaremos realizando en los próximos capítulos.

Sigue los pasos para la creación de esta API REST:

- ***Paso 1:*** Crear un servidor con express, incluir la importación e integración del paquete body-parser.

- ***Paso 2:*** Crear un arreglo con 2 objetos llamados “canales”, teniendo cada canal una propiedad “nombre”. Por motivos del ejercicio agregaremos los canales TNT y ESP.

- ***Paso 3:*** Crear una ruta **GET /canales** que devuelva el arreglo de objetos creado en el paso 2.

- ***Paso 4:*** Crear una ruta **POST /canal** que ingrese (push) el objeto recibido en el cuerpo de la consulta al arreglo de canales y devuelva como respuesta al cliente el arreglo con el nuevo canal agregado.

  Cabe destacar que podremos manipular el JSON enviado desde POSTMAN gracias a la integración del paquete body-parser en nuestro servidor.

- ***Paso 5:*** Crear una ruta **PUT /canal/:canal** que reciba como parámetro el nombre del canal que se desea cambiar y un payload en formato JSON con el nuevo nombre. Se devuelve el arreglo modificado como respuesta de esta ruta al cliente.

- ***Paso 6:*** Crear una ruta **DELETE /canal/:canal** que reciba como parámetro el nombre de un canal que deberás filtrar del arreglo de objetos. Se devuelve el arreglo modificado como respuesta de esta ruta al cliente.

```
// Paso 1
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000);

// Paso 2
let canales = [{ nombre: "TNT" }, { nombre: "ESP" }];

// Paso 3
app.get("/canales", (req, res) => {
  res.send(canales);
});

// Paso 4
app.post("/canal", async (req, res) => {
  const nuevo_Canal = req.body;
  canales.push(nuevo_Canal);
  res.send(canales);
});

// Paso 5
app.put("/canal/:canal", async (req, res) => {
  const { canal } = req.params;
  const { nombre } = req.body;
  canales = canales.map((c) => (c.nombre == canal ? { nombre } : c));
  res.send(canales);
});

// Paso 6
app.delete("/canal/:canal", async (req, res) => {
  const { canal } = req.params;
  canales = canales.filter((c) => c.nombre !== canal);
  res.send(canales);
});
```

Con la API REST creada procedemos con las pruebas, así que abre POSTMAN y realiza los siguientes test:

1. Realiza una consulta **GET** con la siguiente dirección http://localhost:3000/canales, deberás recibir lo que te muestro en la siguiente imagen:

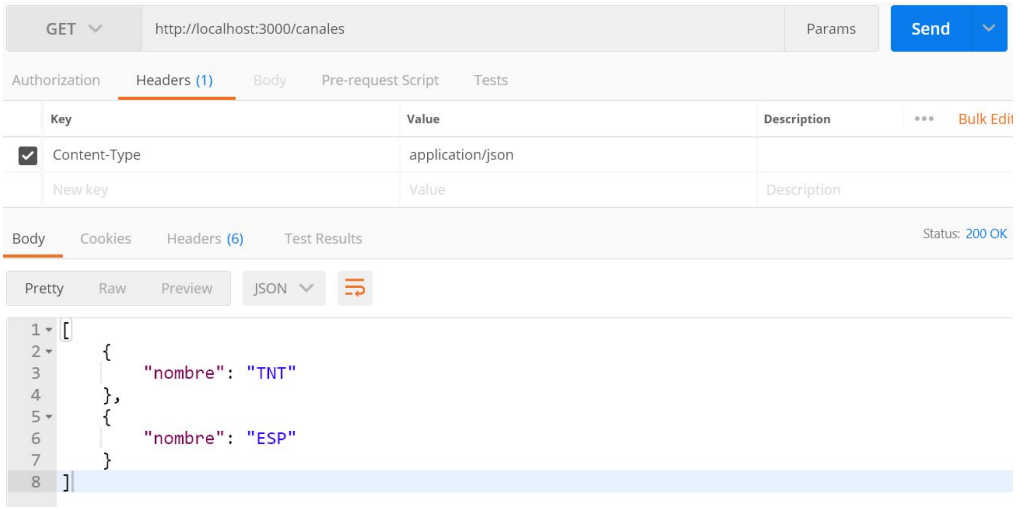

Imagen 1. Canales guardados en el servidor.
Fuente: Desafío Latam

Como ves, estamos recibiendo el arreglo de canales devuelto por nuestra API REST.

2. Realiza una consulta **POST** con la siguiente dirección http://localhost:3000/canal. Envía como cuerpo un JSON con el nombre de un canal nuevo. Deberás recibir lo que te muestro en la siguiente imagen:

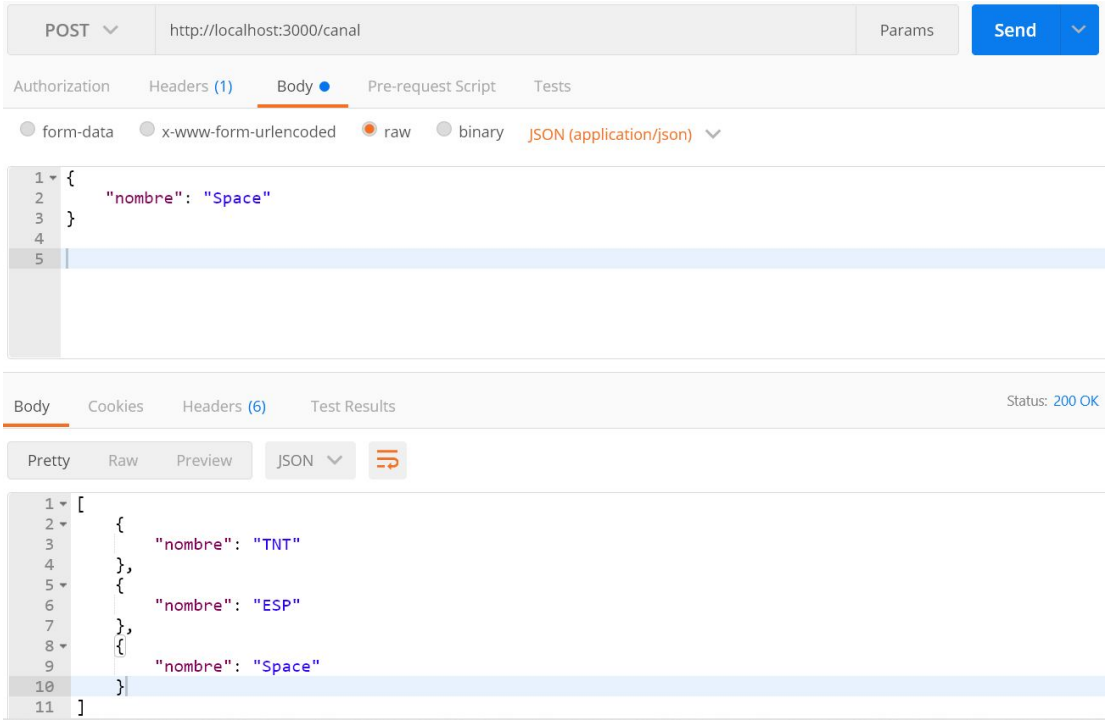

Imagen 2. Nuevo canal agregado al arreglo de canales.
Fuente: Desafío Latam

Ahora podemos observar que fue agregado al arreglo el canal que enviamos como cuerpo de la consulta **POST.**

3. Realiza una consulta **PUT** con la siguiente dirección http://localhost:3000/canal/TNT. Envía como cuerpo un JSON con el nombre de un canal que se sobrescribirá con el canal TNT que estamos declarando como parámetro. Deberás recibir lo que te muestro en la siguiente imagen:

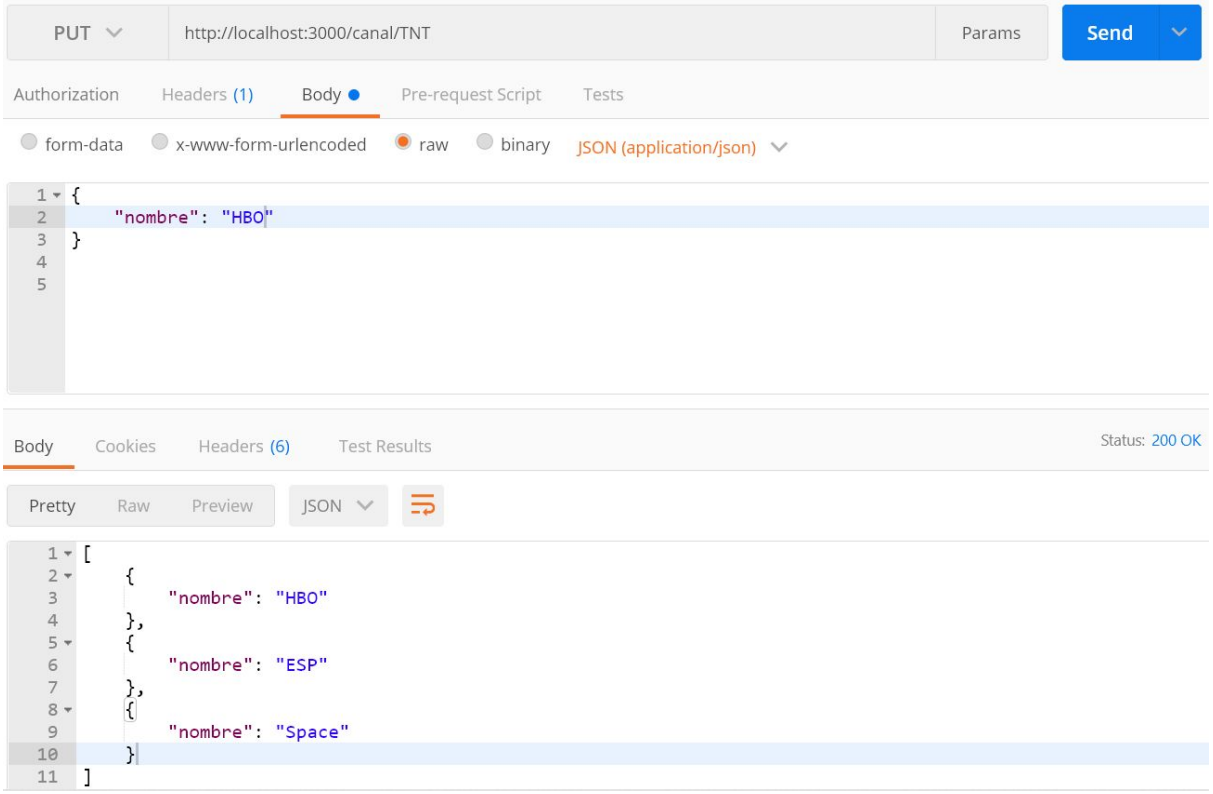

Imagen 3. Canal actualizado en el arreglo de canales.
Fuente: Desafío Latam

Con esto confirmamos que fue modificado el nombre del canal TNT por el canal HBO, ahora solo falta probar el **DELETE**.

4. Realiza una consulta **DELETE** con la siguiente dirección http://localhost:3000/canal/Space, deberás recibir lo que te muestro en la siguiente imagen:

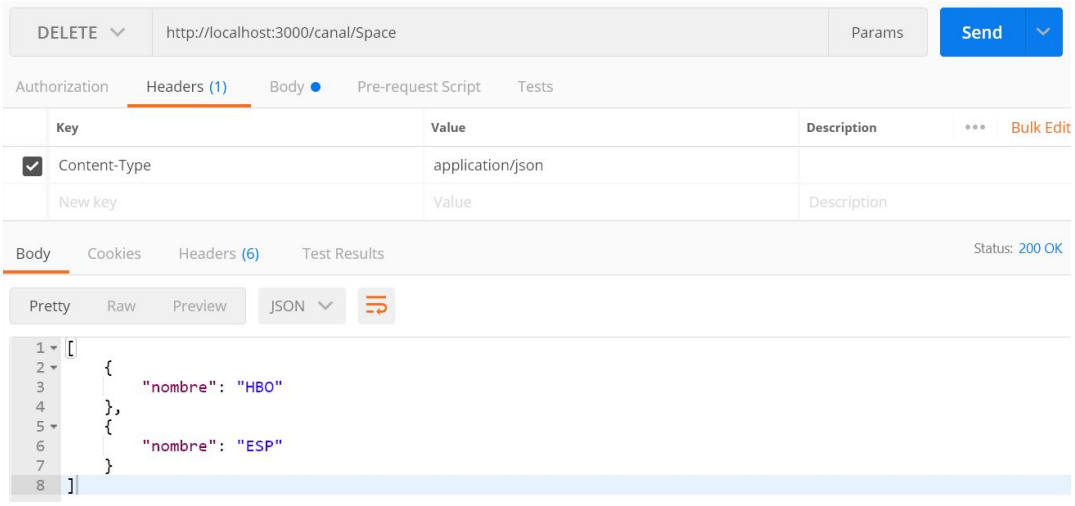

Imagen 4. Canal eliminado del arreglo de canales
Fuente: Desafío Latam

¡Excelente!, el canal fue eliminado del arreglo y nosotros hemos comprobado los 4 métodos HTTP definidos como rutas en nuestra API REST.

Ejercicio propuesto (1)

Debate con tus compañeros la siguiente pregunta:

En comparación con el desarrollo de una API REST con node puro, **¿Cuánto tiempo me he ahorrado por usar Express? ¿Cómo puedo mezclar esto con los conocimientos que he adquirido a lo largo de la carrera?**

Ejercicio propuesto (2)

Basado en el ejercicio guiado “Una API REST con Express”, desarrollar una API REST que gestione el siguiente arreglo:

```
let comidas = [{ nombre: "Pizza" }, { nombre: "Hamburguesa" }];
```

POST (CREATE) & GET(READ)

Competencias

- Construir una ruta POST para la inserción de datos en PostgreSQL utilizando el paquete pg para resolver un problema de persistencia de datos.
- Construir una ruta GET para obtener datos alojados en PostgreSQL utilizando el paquete pg para resolver un problema de persistencia de datos.

Introducción

Hoy en día las API REST son indispensables en el mundo de los web services y no hay señal alguna de que esto cambie pronto. Al dominarlas, podrás consumir y enviar datos entre distintas fuentes y en diferentes entornos, tecnologías y dispositivos.

En este capítulo utilizaremos el paquete pg con nuestra API REST desarrollada con Express, recordando que en una aplicación real la persistencia de datos es indispensable y las conexiones a las bases de datos desde el cliente son realizadas a través de servidores.

Con este conocimiento adquirido podrás hacer inserciones a tablas en PostgreSQL y persistir los datos de tus aplicaciones para próximamente consumirlas en el momento que los necesites.

Consideraciones previas

En este capítulo empezaremos con la construcción de una API REST que utiliza el paquete pg para la gestión de datos alojados en PostgreSQL, por ende, debemos considerar lo siguiente:

1. Usar el siguiente comando por la terminal para instalar el paquete pg:

```
npm i --save pg
```

2. Para un mejor orden en nuestro código, crea un archivo nuevo llamado "consultas.js" para escribir las consultas SQL, a través de "require" y el "module.exports", estaremos importando en nuestro servidor las funciones asíncronas que crearemos en este documento.

3. Abre tu terminal psql y usa las siguientes sentencias SQL para la construcción de una base de datos y una tabla llamada **canales.**

```
CREATE DATABASE canales;

\c canales

CREATE TABLE canales (id SERIAL PRIMARY KEY, nombre VARCHAR(25));
```

4. Los ejercicios que desarrollaremos estarán divididos en 2 partes correspondientes al archivo "consultas.js" y la construcción de la API REST en el servidor.

Ahora sí, estamos listos para iniciar con nuestro ejercicio y la construcción progresiva de una API REST que utilizará el paquete pg para la gestión de datos en PostgreSQL.

Ejercicio guiado: Insertando canales

Construir un servidor con Express que disponibilice una ruta **POST /canal** para la inserción de un nuevo registro en la tabla **canales**.

Sigue los pasos para realizar este ejercicio:

Consultas SQL:

- ***Paso 1:*** Importar la clase Pool y crear una instancia declarando en el objeto de configuración la base de datos “canales”.

- ***Paso 2:*** Crear una función asíncrona de nombre “nuevoCanal” que haga lo siguiente:

    - Recibir un parámetro “canal”, el cual contendrá el nombre de un nuevo canal a agregar en la tabla **canales**.
    - Retornar el arreglo “rows” del objeto “result”, luego de hacer la consulta SQL con el método “query”.
    - Incluir el comando RETURNING * al final de la sentencia SQL para obtener el registro que se está creando en la tabla.

- ***Paso 3:*** Exportar un objeto con la función asíncrona creada en el paso 2.

```
// Paso 1
const { Pool } = require("pg");
const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "canales",
  port: 5432,
});

// Paso 2
async function nuevoCanal(canal) {
  try {
    const result = await pool.query(
      `INSERT INTO canales (nombre) values ('${canal}') RETURNING *`
    );
    return result.rows;
  } catch (e) {
    return e;
  }
}

// Paso 3
module.exports = {
  nuevoCanal,
};
```

Servidor:

- ***Paso 1:*** Crear un servidor con Express, incluir la importación e integración del paquete body-parser.
- ***Paso 2:*** Importar la función "nuevoCanal" del archivo "consultas.js".
- ***Paso 3:*** Crear una ruta **POST /canal.**
- ***Paso 4:*** Extraer en una constante la propiedad "nombre" del cuerpo de la consulta a través del objeto request.
- ***Paso 5:*** Guardar en una constante "respuesta" el resultado de la función asíncrona "nuevoCanal" cuyo argumento deberá ser la constante creada en el paso 4.
- ***Paso 6:*** Devolver la respuesta al cliente.

```
// Paso 1
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000);

// Paso 2
const { nuevoCanal } = require("./consultas");

// Paso 3
app.post("/canal", async (req, res) => {
  // Paso 4
  const { nombre } = req.body;
  // Paso 5
  const respuesta = await nuevoCanal(nombre);
  // Paso 6
  res.send(respuesta);
});
```

Ahora probamos nuestra API REST y la lógica del archivo "consultas.js". Repite la misma consulta **POST** que realizaste en el capítulo anterior, pero en esta ocasión, agrega un canal llamado "A&E" y deberás recibir lo que te muestro en la siguiente imagen:

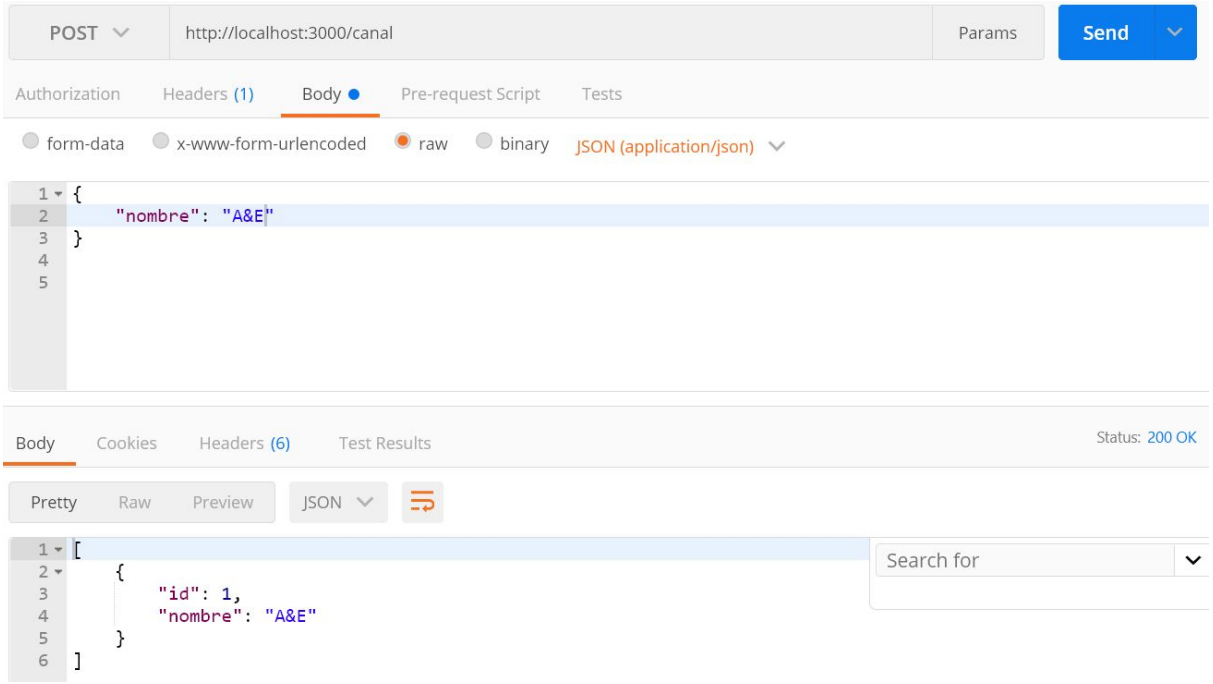

Imagen 5. Canal agregado a la tabla canales.
Fuente: Desafío Latam

¡Muy bien! la respuesta que hemos recibido por parte de nuestra API REST es señal de que fue creado un registro en la tabla **canales** dentro de la base de datos de PostgreSQL.

Ejercicio guiado: Consultando canales

Crear una ruta **GET /canales** que al ser consultada devuelva los registros de la tabla **canales**. Sigue los pasos para la solución de este ejercicio.

Consultas SQL:

- ***Paso 1:*** Crear una función asíncrona de nombre "getCanales" que realice una consulta SQL para obtener y retornar todos los registros de la tabla **canales**.

- ***Paso 2:*** Incluir la función en la exportación del documento.

```
// Paso 1
async function getCanales() {
  try {
    const result = await pool.query(`SELECT * FROM canales`);
    return result.rows;
  } catch (e) {
    return e;
  }
}

// Paso 2
module.exports = {
  nuevoCanal,
  getCanales,
};
```

Servidor:

- ***Paso 1:*** Crear una ruta **GET /canales.**
- ***Paso 2:*** Almacenar en una constante la respuesta de la función asíncrona "getCanales()". Recuerda agregar esta función en la importación de documento "consultas.js".
- ***Paso 3:*** Devolver al cliente la respuesta almacenada en la constante creada en el paso 2.

```
// Paso 1
app.get("/canales", async (req, res) => {
  // Paso 2
  const respuesta = await getCanales();
  // Paso 3
  res.send(respuesta);
});
```

Antes de probar esta ruta, agreguemos un canal más a través de una consulta **POST**. En este caso incorporamos el canal llamado "History Channel" ¿Por qué hacemos esto? Para comprobar que estamos recibiendo todos los registros de la tabla **canales**, con esta última inserción esperamos obtener los 2 canales que estarán en la base de datos.

Al hacer esta segunda inserción deberás obtener la respuesta que te muestro en la siguiente imagen:

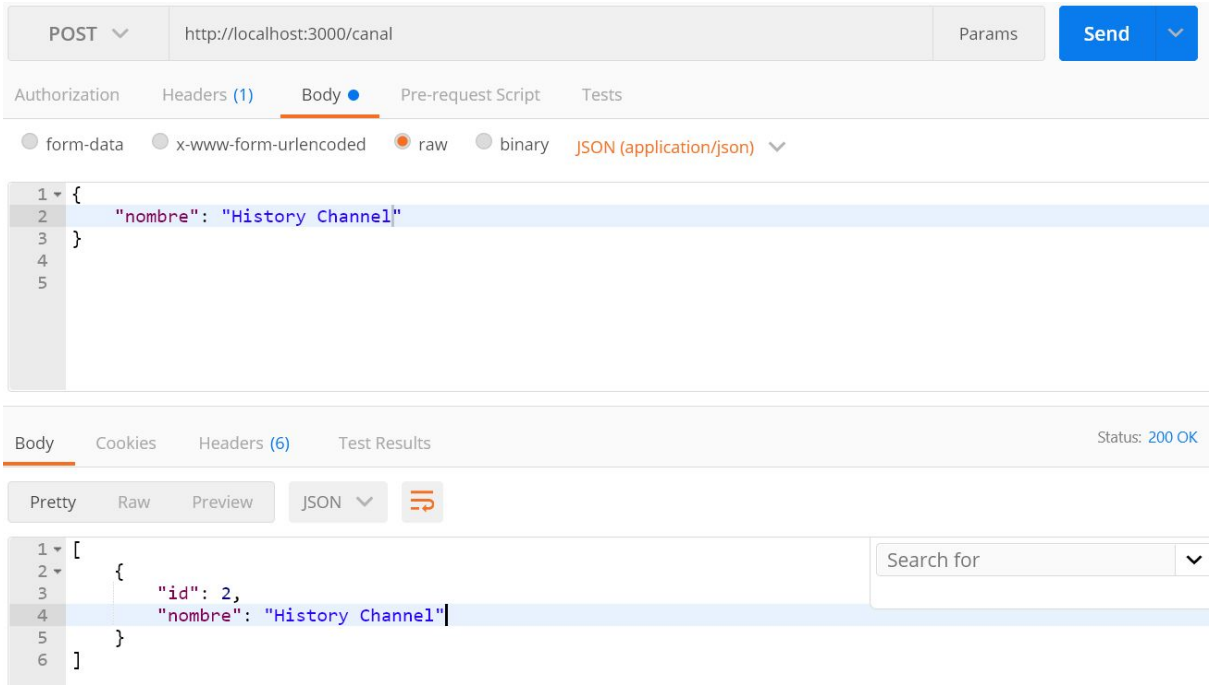

Imagen 6. Canal “History Channel” agregado a la base de datos.
Fuente: Desafío Latam

Ahora que tenemos 2 canales registrados, realiza una consulta **GET** a la ruta http://localhost:3000/canales y deberás obtener lo que te muestro en la siguiente imagen:

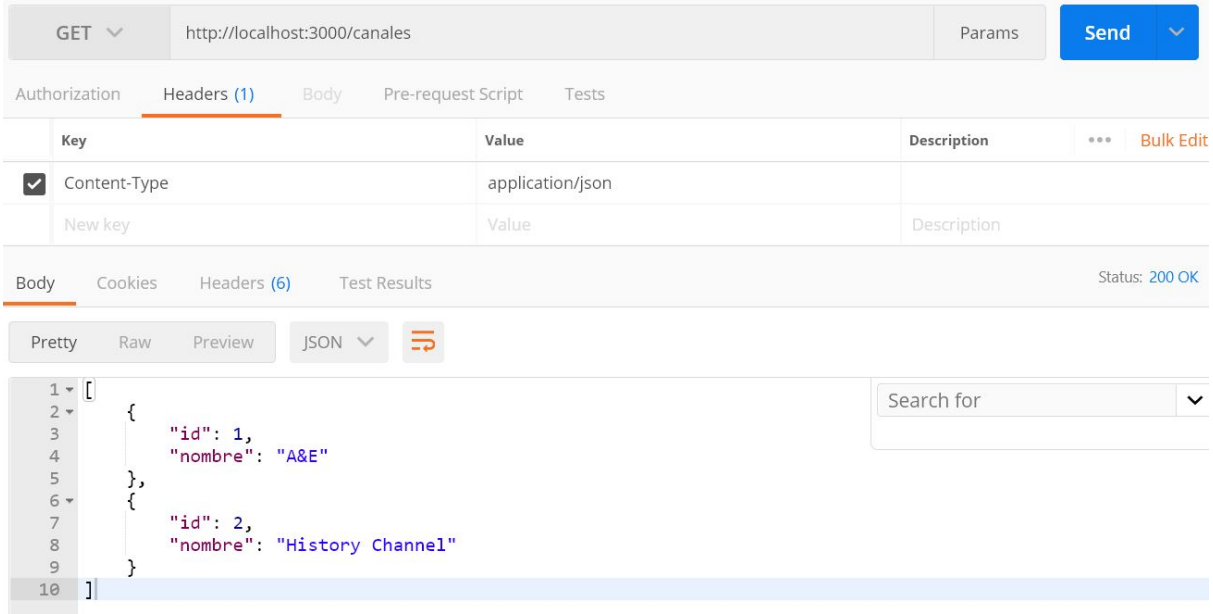

Imagen 7. Canales registrados en la tabla “canales”.
Fuente: Desafío Latam

Ahí lo tenemos, la respuesta de la consulta nos devolvió los 2 canales registrados en la tabla **canales**, ¡Sigue así, lo estás haciendo muy bien!

Ejercicio propuesto (3)

Basado en los ejercicios “Insertando canales” y “Obteniendo canales”, construir una API REST con Express que disponibilice las rutas GET y POST para la obtención e inserción de datos registrados en la siguiente tabla SQL:

```
CREATE TABLE actores (id SERIAL PRIMARY KEY, nombre VARCHAR(25));
```

PUT (UPDATE) & DELETE (DELETE)

Competencias

- Construir una ruta PUT para la actualización de datos en PostgreSQL utilizando el paquete pg para resolver un problema de persistencia de datos.
- Construir una ruta DELETE para eliminar registros en PostgreSQL utilizando el paquete pg para resolver un problema de persistencia de datos.

Introducción

En este punto de la lectura hemos logrado crear un servidor con 2 rutas correspondientes a la creación y obtención de registros alojados en tabla **canales**. Con lo anterior hemos cumplido con las funcionalidades CREATE y READ de un sistema de gestión básico, no obstante, aún falta programar la posibilidad de hacer UPDATE y DELETE, de esa manera completar el CRUD. En este capítulo, agregaremos las rutas PUT y DELETE para tener lista una API REST que dispone de los 4 métodos HTTP básicos.

Al finalizar este capítulo habremos terminado con nuestro ejercicio progresivo con temática de canales de televisión y como consecuencia de esto habrás fortalecido tus habilidades en el desarrollo de API REST, ahora con el framework Express como arquitectura base del servidor.

Ejercicio guiado: Actualizando canales

Crear una ruta **PUT /canal/:id** que utilice una función asíncrona para actualizar el registro del canal que tenga como id el número 1.

Consultas SQL:

- ***Paso 1:*** Crear una función asíncrona de nombre "editCanal", que reciba como primer parámetro el id del canal que se desea actualizar y como segundo el nuevo nombre que usaremos para actualizarlo. El objetivo de esta función será devolver un registro actualizado luego de emitir una consulta SQL a la tabla **canales**.

- ***Paso 2:*** Incluir la función en la exportación del documento.

```
// Paso 1
async function editCanal(id, nuevoNombre) {
  try {
    const res = await pool.query(
      `UPDATE canales SET nombre = '${nuevoNombre}' WHERE id = '${id}'
RETURNING *`
    );
    return res.rows;
  } catch (e) {
    console.log(e);
  }
}

// Paso 2
module.exports = {
  nuevoCanal,
  getCanales,
  editCanal
};
```

Servidor:

- ***Paso 1:*** Crear una ruta **PUT /canal/:id.**

- **Paso 2:** Usar destructuring con el objeto request para crear una constante a partir del parámetro de la ruta llamado "id".

- ***Paso 3:*** Usar destructuring con el payload de la consulta para crear una constante usando la propiedad "nombre".

- ***Paso 4:*** Almacenar en una constante la respuesta de la función asíncrona "editCanal()" cuyos argumentos serán el id y el nombre almacenados en las constantes creadas en los pasos 2 y 3. Recuerda agregar esta función en la importación de documento "consultas.js".

- ***Paso 5:*** Devolver al cliente la respuesta almacenada en la constante creada en el paso 4.

```
// Paso 1
app.put("/canal/:id", async (req, res) => {
  // Paso 2
  const { id } = req.params;
  // Paso 3
  const { nombre } = req.body;
  // Paso 4
  const respuesta = await editCanal(id, nombre);
  // Paso 5
  res.send(respuesta);
});
```

Para probar esta ruta y actualizar un canal en la base de datos, abre POSTMAN y consulta la ruta http://localhost:3000/canal/1 para sobrescribir el nombre del canal de id 1. Como payload envía un JSON con la propiedad "nombre" y define como valor "FOX", deberás recibir la respuesta que te muestro en la siguiente imagen:

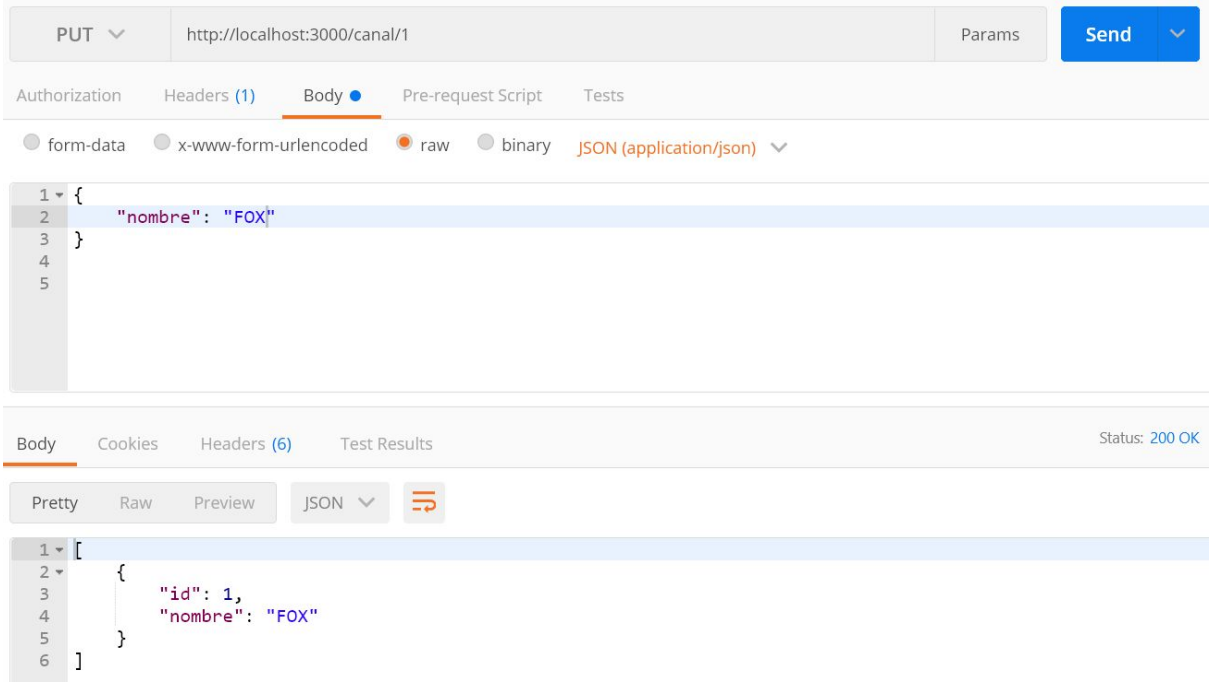

Imagen 8. Canal actualizado al consultar la ruta PUT.
Fuente: Desafío Latam

Muy buen trabajo, con esta respuesta estamos confirmando que fue actualizado el registro de id 1 y su campo nombre fue sobrescrito por el valor que enviamos como payload en la consulta **PUT**.

Ejercicio guiado: Eliminando canales

Crear una ruta **DELETE /canal/:id** que utilice una función asíncrona para eliminar el registro de un canal en la base de datos y en caso de no eliminar ningún registro, devolver un mensaje indicando que no existe ningún canal registrado con ese id.

Consultas SQL:

- ***Paso 1:*** Crear una función asíncrona de nombre "deleteCanal" que ejecute una consulta SQL para eliminar el registro de la tabla **canales** que coincida con un id recibido como parámetro. La función deberá retornar el rowCount del objeto result.

- ***Paso 2:*** Incluir la función en la exportación del documento.

```
// Paso 1
async function deleteCanal(id) {
  try {
    const result = await pool.query(`DELETE FROM canales WHERE id =
'${id}'`);
    return result.rowCount;
  } catch (e) {
    return e;
  }
}

// Paso 2
module.exports = {
  nuevoCanal,
  getCanales,
  editCanal,
  deleteCanal
};
```

Servidor:

- ***Paso 1:*** Crear una ruta **DELETE /canal/:id.**
- **Paso 2:** Usar destructuring con el objeto request para crear una constante a partir del parámetro de la ruta llamado "id".
- ***Paso 3:*** Almacenar en una constante la respuesta de la función asíncrona "deleteCanal()" cuyos argumento serán el id recibido como parámetro en la ruta. Recuerda agregar esta función en la importación de documento "consultas.js".
- ***Paso 4:*** Realizar un operador ternario para condicionar que el valor de la respuesta es mayor a 0, con esto lo que hacemos es validar que fue eliminado algún canal en la base de datos, en caso contrario devolver un mensaje indicando que no existe ningún canal registrado con ese id.

```
// Paso 1
app.delete("/canal/:id", async (req, res) => {
  // Paso 2
  const { id } = req.params;
  // Paso 3
  const respuesta = await deleteCanal(id);
  // Paso 4
  respuesta > 0
    ? res.send(`El canal de id ${id} fue eliminado con éxito`)
    : res.send("No existe un canal registrado con ese id");
});
```

Probemos esta ruta eliminado el registro de id 1, a través de una consulta **DELETE** a la ruta http://localhost:3000/canal/1. Deberás recibir lo que te muestro en la siguiente imagen:

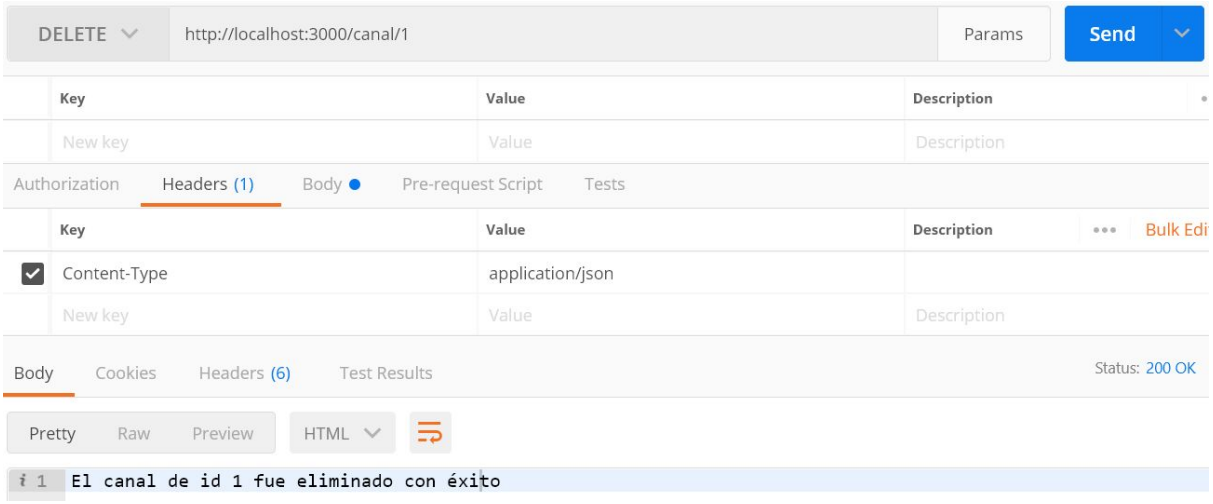

Imagen 9. Canal eliminado luego de hacer una consulta a la ruta DELETE.
Fuente: Desafío Latam

Recibimos como respuesta el mensaje preparado en el caso de éxito del operador ternario, puesto que sí existía un registro con ese id, pero ¿Qué sucede si volvemos a hacer la consulta? Deberás recibir lo que te muestro en la siguiente imagen:

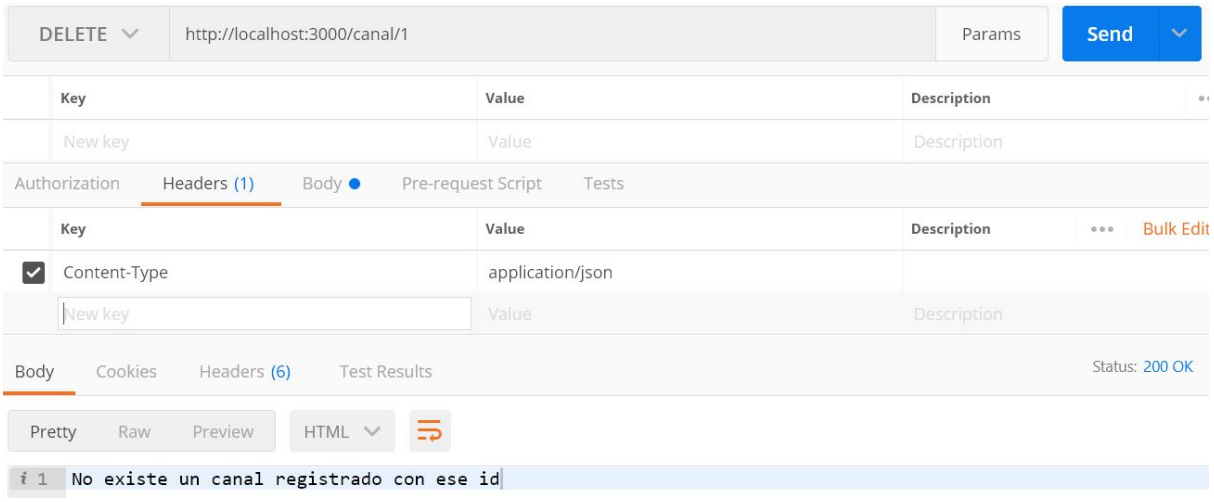

Imagen 10. Mensaje obtenido por intentar eliminar un canal con un id no identificado.
Fuente: Desafío Latam

¡Espectacular!, ahí lo tenemos, un mensaje devuelto basado en una condición.

Tener los 4 métodos básicos HTTP servidos con rutas en nuestro servidor es solo una parte de las "buenas prácticas" de API REST, las cuales profundizaremos y conoceremos en la próxima sesión.

Ejercicio propuesto (4)

Basado en los ejercicios "Actualizando canales" y "Eliminando canales", incorpora a la API REST que creaste en el ejercicio propuesto 3, las rutas PUT y DELETE para la actualización y eliminación de datos registrados en la tabla **actores**.

Resumen

En esta lectura aprendimos a crear una API REST que disponibiliza cuatro rutas correspondientes a los métodos básicos para comunicaciones bajo el protocolo HTTP. En conjunto con el paquete pg, logramos la gestión y persistencia de datos almacenados en PostgreSQL manejados por un servidor desarrollado con Express.

En síntesis hemos abordado los siguientes contenidos:

- Qué es una API REST y su utilidad en la comunicación de aplicaciones independientemente de la tecnología en la que fueron desarrolladas.
- Qué es un endpoint, cómo se construyen a partir de la base URL y la definición de un path variable para la creación de un endpoint que devuelve recursos de forma dinámica.
- Cuáles son los parámetros en los endpoints y cómo se obtienen del objeto request en un servidor con Express.
- Cómo crear rutas GET y POST para la obtención e inserción de datos en una tabla alojada en PostgreSQL.
- Cómo crear rutas PUT y DELETE para la actualización y eliminación de datos en una tabla alojada en PostgreSQL.

Solución de los ejercicios propuesto

1. Desarrollar un servidor que permita la carga de videos recibidos por medio de un formulario HTML cuyo input está declarado con el nombre “video”. Deberás usar el paquete express-fileupload y desarrollar la lógica que almacene los archivos en una carpeta llamada “uploads”, además de tener definido un límite de 2 MB por archivo.

   El ahorro de tiempo es notoriamente mejor por la estructura minimalista de express.

   Sin inconvenientes es totalmente posible aplicar todos los conocimientos adquiridos hasta ahora puesto que express corresponde a la creación del servidor, es decir la aplicación de los paquetes: nodemailer, nodemon, chalk, JIMP, axios, yargs, mocha, lodash, UUID, pg y todo el manejo asíncrono de Node sigue siendo igual.

2. Basado en el ejercicio guiado “Una API REST con express”, desarrollar una API REST que gestione el siguiente arreglo:

```
let comidas = [{ nombre: "Pizza" }, { nombre: "Hamburguesa" }];
```

```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000);

let comidas = [{ nombre: "Pizza" }, { nombre: "Hamburguesa" }];

app.get("/comidas", (req, res) => {
  res.send(comidas);
});

app.post("/comida", async (req, res) => {
  const nueva_comida = req.body;
  comidas.push(nueva_comida);
  res.send(comidas);
});
```

```
app.put("/comida/:comida", async (req, res) => {
  const { comida } = req.params;
  const { nombre } = req.body;
  comidas = comidas.map((c) => (c.nombre == comida ? { nombre } : c));
  res.send(comidas);
});

app.delete("/comida/:comida", async (req, res) => {
  const { comida } = req.params;
  comidas = comidas.filter((c) => c.nombre !== comida);
  res.send(comidas);
});
```

3. Basado en los ejercicios “Insertando canales” y “Obteniendo canales”, construir una API REST con express que disponibilice las rutas GET y POST para la obtención e inserción de datos registrados en siguiente tabla SQL:

```
CREATE TABLE actores (id SERIAL PRIMARY KEY, nombre VARCHAR(25));
```

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

async function nuevoActor(actor) {
  try {
    const result = await pool.query(
      `INSERT INTO actores (nombre) values ('${actor}') RETURNING *`
    );
    return result.rows;
  } catch (e) {
    console.log(e);
    return e;
  }
}
```

```
async function getActores() {
  try {
    const result = await pool.query(`SELECT * FROM actores`);
    return result.rows;
  } catch (e) {
    return e;
  }
}

module.exports = {
  nuevoActor,
  getActores,
};
```

```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000);

const { nuevoActor, getActores } = require("./consultasActores");

app.post("/actor", async (req, res) => {
  const { nombre } = req.body;
  const respuesta = await nuevoActor(nombre);
  res.send(respuesta);
});

app.get("/actores", async (req, res) => {
  const respuesta = await getActores();
  res.send(respuesta);
});
```

4. Basado en los ejercicios “Actualizando canales” y “Eliminando canales”, agregarle a la API REST que creaste en el ejercicio propuesto 2 las rutas PUT y DELETE para la actualización y eliminación de datos registrados en la tabla “actores”.

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

async function nuevoActor(actor) {
  try {
    const result = await pool.query(
      `INSERT INTO actores (nombre) values ('${actor}') RETURNING *`
    );
    return result.rows;
  } catch (e) {
    console.log(e);
    return e;
  }
}

async function getActores() {
  try {
    const result = await pool.query(`SELECT * FROM actores`);
    return result.rows;
  } catch (e) {
    console.log(e);
    return e;
  }
}

async function editActor(id, nuevoNombre) {
  try {
    const res = await pool.query(
      `UPDATE actores SET nombre = '${nuevoNombre}' WHERE id = '${id}'
RETURNING *`
    );
    return res.rows;
  } catch (e) {
    console.log(e);
    return e;
  }
}
```

```
async function deleteActor(id) {
  try {
    const result = await pool.query(`DELETE FROM actores WHERE id =
'${id}'`);
    return result.rowCount;
  } catch (e) {
    return e;
  }
}
module.exports = {
  nuevoActor,
  getActores,
  editActor,
  deleteActor
};
```

```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000);

const { nuevoActor, getActores, editActor, deleteActor} =
require("./consultasActores");

app.post("/actor", async (req, res) => {
  const { nombre } = req.body;
  const respuesta = await nuevoActor(nombre);
  res.send(respuesta);
});

app.get("/actores", async (req, res) => {
  const respuesta = await getActores();
  res.send(respuesta);
});

app.put("/actor/:id", async (req, res) => {
  const { id } = req.params;
  const { nombre } = req.body;
  const respuesta = await editActor(id, nombre);
```

```
  res.send(respuesta);
});

app.delete("/actor/:id", async (req, res) => {
  const { id } = req.params;
  const respuesta = await deleteActor(id);
  respuesta > 0
    ? res.send(`El actor de id ${id} fue elimado con éxito`)
    : res.send("No existe un actor registrado con ese id");
});
```