

# Transacciones, cursores y API REST (Parte II)

## Levantando un servidor con conexión a PostgreSQL

### Competencia

- Construir un servidor que se conecte con una base de datos PostgreSQL utilizando el entorno Node.

### Introducción

Saber SQL y cómo ocuparlo directamente en la terminal es bueno, saber como crear una aplicación en Node que se conecte con el paquete pg a PostgreSQL y realice consultas con cursores e incluso transacciones es mejor que solo saber SQL, pero incluso es mucho mejor e importante saber crear un entorno completo full stack, que disponibilice a través de una interfaz gráfica de usuario las funcionalidades programadas desde una aplicación en Node.

En este capítulo, aprenderás y empezarás a crear un sistema que mezcle las bases y fundamentos aprendidos hasta ahora en el módulo, creando un servidor en Node que disponibilice una ruta que al ser consultada reciba un dato de tipo fecha emitido desde PostgreSQL. Con esto podrás empezar a compactar tus conocimientos de desarrollo frontend, backend y bases de datos teniendo como resultado un perfil más competente.

## Preparando el servidor

Ya hemos desarrollado varios servidores a lo largo de este módulo y del anterior, no obstante, no está demás recordar que la construcción de un servidor se realiza en Node gracias al módulo "http" y su método "createServer". Partamos directamente con la construcción de un servidor básico de Node.

### Ejercicio guiado: Preparando mi servidor

Construir un servidor en Node que utilice el paquete pg para consultar la fecha actual emitida desde PostgreSQL y devuelta a un cliente a través de una ruta.

Prosigue con los siguientes pasos para la realización de este ejercicio:

- **Paso 1:** Importar el módulo http.
- **Paso 2:** Crear un servidor con el método createServer.
- **Paso 3:** Disponibilizar una ruta GET orientada a la raíz del servidor.
- **Paso 4:** Ocupar el método listen para disponibilizar al servidor en el puerto 3000.

```
// Paso 1
const http = require("http");
// Paso 2
http
  .createServer((req, res) => {
    // Paso 3
    if (req.url == "/" && req.method === "GET") {
      res.end("Servidor funcionando =D !");
    }
  })
// Paso 4
  .listen(3000);
```

Ahora abre tu navegador y entra a tu servidor a través de la ruta: <http://localhost:3000/>, deberás ver lo que te muestro en la siguiente imagen:

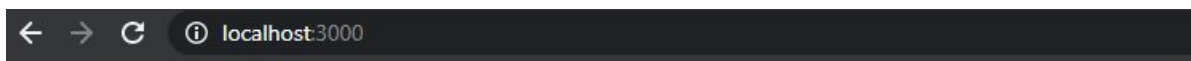


Imagen 1. Probando servidor básico en Node.  
Fuente: Desafío Latam

¡Perfecto!, ya hemos levantado el servidor sin problemas, aprovecho para recordarte que la práctica hace al maestro, si hoy en día has logrado levantar un servidor sin problemas y además entiendes como funciona, se debe a que ya lo hemos hecho varias veces y en cada repetición adquirimos más dominio en las tecnologías y herramientas que estemos usando.

## Consultando con el paquete pg desde el servidor

Ahora que tenemos nuestro servidor, podemos enfocarnos en la consulta a PostgreSQL con el paquete “pg”, para esto por debes instalar el paquete con siguiente comando:

```
npm i pg
```

Una vez instalado, procedemos con la lógica a consultar al motor de bases de datos, pero hagámoslo de una forma modular y organizada.

## Ejercicio guiado: Probando conexión

Crear un archivo nuevo llamado “consultas.js”, en donde escribiremos una función asíncrona que al ser ejecutada realice una consulta SQL con la instrucción “SELECT NOW()”, la cual recordemos procesa una petición al motor de base de datos, que devuelve un dato de tipo Date con la fecha actual.

Prosigue los siguientes pasos para la realización de este ejercicio:

- **Paso 1:** Importar el paquete pg y crear una instancia de la clase Pool, definiendo las propiedades básicas para una consulta. No es necesario definir ninguna base de datos.
- **Paso 2:** Crear una función asíncrona que devuelva el objeto result de una consulta SQL con la instrucción “SELECT NOW()”.
- **Paso 3:** Exportar un módulo en forma de objeto que contenga la función asíncrona creada en el paso anterior.

```
// Paso 1
const { Pool } = require("pg");

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

// Paso 2
const getDate = async () => {
  const result = await pool.query("SELECT NOW()");
  return result;
};

// Paso 3
module.exports = { getDate };
```

Muy bien, con este código escrito en el archivo “consultas.js”, ahora debemos importarlo en nuestro servidor, por lo que deberás realizar los siguientes cambios al archivo principal en donde desarrollamos el servidor:

- **Paso 1:** Importar la función asíncrona getDate del archivo “consultas.js”.
- **Paso 2:** Dentro de la ruta raíz, almacena en una constante el resultado de la función asíncrona importada.
- **Paso 3:** Devolver al cliente a través del método “end()” del objeto “res” un JSON con el objeto result de la función asíncrona.

```
const http = require("http");
// Paso 1
const { getDate } = require("./consultas");
http
  .createServer(async (req, res) => {
    if (req.url == "/" && req.method === "GET") {
      // Paso 2
      const result = await getDate();
      // Paso 3
      res.end(JSON.stringify(result));
    }
  })
  .listen(3000);
```

Ahora abre tu navegador y entra al servidor. Deberás recibir algo como lo que te muestro en la siguiente imagen:



```
1 // 20201101110751
2 // http://localhost:3000/
3
4 {
5   "command": "SELECT",
6   "rowCount": 1,
7   "oid": null,
8   "rows": [
9     {
10      "now": "2020-11-01T14:07:51.263Z"
11    }
12  ],
13  "fields": [↔],
14  "_parsers": [↔],
15  "_types": {↔},
16  "RowCtor": null,
17  "rowFromArray": false
18 }
```

Imagen 2. Objeto result de PostgreSQL recibido desde el servidor.

Fuente: Desafío Latam

Ahí lo tenemos, estamos recibiendo el objeto result enviado desde PostgreSQL a nuestro servidor, disponibilizado a través de la ruta raíz. Con esto haz logrado un importante paso, porque podremos de ahora en adelante utilizar los mecanismos de un servidor Node, para programar una API REST y ofrecerle a un cliente los métodos para la gestión de datos en nuestras bases de datos.

## Ejercicio propuesto (1)

Desarrollar un servidor en Node que disponibilice una ruta GET **/fecha** que devuelva un string al cliente con una fecha enviada desde PostgreSQL.

## Insertando registros

### Competencia

- Construir un servidor en Node que disponibilice una ruta para crear registros de PostgreSQL a través del paquete pg.

### Introducción

Ahora que sabes crear un servidor que disponibiliza una ruta al cliente para consultar la fecha devuelta desde PostgreSQL, podrás empezar a programar una API REST que le permita a la aplicación cliente ejecutar las funcionalidades básicas de un sistema administrativo, me refiero a un CRUD (Create Read Update Delete).

En este capítulo, comenzaremos con la creación de una API REST en Node, que utilice el paquete pg y PostgreSQL como motor de bases de datos e iniciaremos con la función INSERT, para agregar un nuevo registro en una tabla a través de una interfaz en HTML.

## Insertando datos a PostgreSQL desde el servidor

Con lo aprendido en el capítulo anterior, podemos iniciar con la creación de una API REST para disponibilizar las funcionalidades de un sistema CRUD a nuestras aplicaciones clientes, y eso es justo lo que haremos ahora con el siguiente ejercicio.

### Ejercicio guiado: POST & CREATE (CREATE)

El gimnasio Heart Strong, necesita desarrollar un servidor en Node que devuelva varios formularios HTML, para una gestión de tipo CRUD de los registros almacenados en una tabla llamada **ejercicios**, de su base de datos llamada **gym**.

El ejercicio que realizaremos a lo largo de esta lectura será tu primer desarrollo backend que utiliza un motor de bases de datos para persistir información. Es un paso muy importante que darás como full stack developer puesto que en el mundo laboral realizamos este tipo de prácticas constantemente. Junto a tus conocimientos en el desarrollo frontend, podrás utilizar este ejercicio como base o maqueta para partir con próximos proyectos y agilizar el desarrollo de tus aplicaciones full stack.

Para iniciar con este ejercicio necesitarás hacer lo siguiente:

- Abre tu terminal de PostgreSQL.
- Crea la base de datos con la siguiente instrucción:

```
CREATE DATABASE gym;
```

- Conéctate a la base de datos **gym** con la siguiente instrucción.

```
\c gym;
```

- Crea la tabla **ejercicios** con los campos: nombre, series, repeticiones y descanso. Para esto utiliza la siguiente instrucción:

```
CREATE TABLE ejercicios (nombre varchar(30), series varchar(30),  
repeticiones varchar(30), descanso varchar(30));
```

Con la tabla **ejercicios** creada ahora si podemos iniciar a construir la API REST, partiremos con la modificación de la ruta raíz para que devuelva un documento HTML (index.html), que encontrarás como material de apoyo llamado **Apoyo Lectura - Gimnasio Heart Strong** en el



que está desarrollada una interfaz básica pero funcional, que consultará rutas a nuestro servidor.

- **Paso 1:** Importar la función asíncrona insertar del archivo "consultas.js". Esta función aún no la creamos pero no hay problema, simplemente estamos adelantándonos y tenemos escrito todo lo que necesitaremos en el servidor, posteriormente la crearemos en el archivo "consultas.js".
- **Paso 2:** Importar el módulo fs y el módulo url.
- **Paso 3:** Usar el objeto response del servidor para devolver la cabecera especificando que el contenido que se devolverá será HTML.
- **Paso 4:** Usar el método readFileSync del módulo File System para leer el documento index.html y almacenarlo en una variable. Esta variable debe ser devuelta al cliente.

```
const http = require("http");
// Paso 1
const { insertar } = require("./consultas");
// Paso 2
const fs = require("fs");
http
  .createServer(async (req, res) => {
    if (req.url == "/" && req.method === "GET") {
      // Paso 3
      res.setHeader("content-type", "text/html");
      // Paso 4
      const html = fs.readFileSync("index.html", "utf8");
      res.end(html);
    }
  })
  .listen(3000);
```

Ahora abre tu navegador y entra al servidor. Deberás recibir algo similar la siguiente imagen:

The screenshot shows a web browser at localhost:3000. It contains three forms for managing exercises:

- Agregar nuevo ejercicio:** Fields for Nombre, Series, Repeticiones, and Descanso, with an 'Agregar' button.
- Editar ejercicio:** A dropdown for Nombre, and fields for Series, Repeticiones, and Descanso, with an 'Editar' button.
- Eliminar ejercicio:** A dropdown for Nombre and an 'Eliminar' button.

Below the forms is a table with the following columns:

Nombre	Series	Repeticiones	Descanso
--------	--------	--------------	----------

Imagen 3. Formularios HTML devueltos por el servidor en la ruta raíz.

Fuente: Desafío Latam

Perfecto, ahora que tenemos los formularios a nuestra disposición en el lado del cliente, debemos crear la función asíncrona en el archivo “consultas.js”, encargada de recibir un payload enviado desde el cliente en el formulario con leyenda “Agregar nuevo ejercicio”, te recomiendo que te tomes unos 5 minutos para revisar con detención el código escrito en el HTML y luego continúes con esta lectura.

### Función Insertar

Ahora que estás más contextualizado con el ejercicio, procedamos con modificar el archivo “consultas.js” para incluir la base de datos creada y la creación de la función “insertar”, la cual se encargará de recibir como parámetro los datos enviados por método POST, desde el cliente para realizar la consulta SQL que registre este nuevo ejercicio en la tabla.

Prosigue con los siguientes pasos para el desarrollo de esta etapa:

- **Paso 1:** Agregar la base de datos “gym” al objeto de configuración de la clase Pool.
- **Paso 2:** Crear una función asíncrona llamada “insertar” que reciba un parámetro “datos”. El cual será un arreglo enviado desde el servidor.
- **Paso 3:** Generar con try catch una consulta parametrizada con un JSON como argumento definiendo como “values” el parámetro “datos” de la función y retornando el objeto “result”.
- **Paso 4:** Dentro del bloque catch, devuelve el código del error obtenido.
- **Paso 5:** Exportar un objeto con la función “insertar”

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

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

// Paso 2
const insertar = async (datos) => {
  // Paso 3
  const consulta = {
    text: "INSERT INTO ejercicios values($1, $2, $3, $4)",
    values: datos,
  };
  try {
    const result = await pool.query(consulta);
    return result;
  } catch (error) {
    // Paso 4
    console.log(error.code);
    return error;
  }
};

// Paso 5
module.exports = { insertar };
```

## Ejercicio propuesto (2)

Desarrollar una función asíncrona que realice una consulta SQL para insertar un registro con los datos recibidos como parámetros y retorna el último registro insertado en forma de arreglo.

## Ruta POST

Con lo anterior terminado, ahora necesitamos crear en nuestro servidor una ruta **POST /ejercicios**, que obtenga los datos enviados desde el cliente y se la pase como argumento, en formato de arreglo a la función "insertar" para finalmente devolver el resultado.

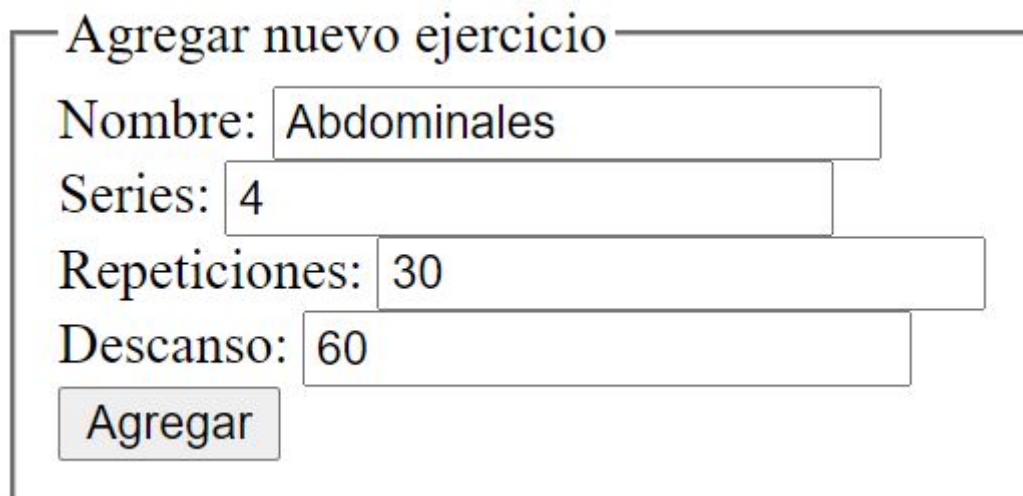
- **Paso 1:** Crear una ruta **POST /ejercicios** que reciba la data enviada desde el cliente.
- **Paso 2:** Al terminar de recibir la data del cliente, utiliza el `Object.values()` para almacenar en una constante un arreglo con los valores del objeto body.
- **Paso 3:** Guardar en una constante la respuesta de la ejecución de la función asíncrona "insertar" pasándole como argumento el arreglo generado en el paso anterior.
- **Paso 4:** Devolver al cliente un JSON con el objeto de respuesta creado en el paso anterior

```
const http = require("http");
const { insertar } = require("./consultas");
const fs = require("fs");
http
  .createServer(async (req, res) => {
    if (req.url === "/" && req.method === "GET") {
      res.setHeader("content-type", "text/html");
      const html = fs.readFileSync("index.html", "utf8");
      res.end(html);
    }

    // Paso 1
    if ((req.url === "/ejercicios" && req.method === "POST")) {
      let body = "";
      req.on("data", (chunk) => {
        body += chunk;
      });
      req.on("end", async () => {
        // Paso 2
        const datos = Object.values(JSON.parse(body));
        // Paso 3
        const respuesta = await insertar(datos);
        // Paso 4
        res.end(JSON.stringify(respuesta));
      });
    }
  })
```

```
});  
}  
})  
.listen(3000);
```

Ahora que tenemos la ruta creada, preparada en el servidor y la función insertar desarrollada en el archivo “consultas.js”, es momento de probar esto. Intenta insertar un nuevo ejercicio a través del formulario HTML, como te muestro en la siguiente imagen:



Agregar nuevo ejercicio

Nombre:

Series:

Repeticiones:

Descanso:

Imagen 4. Ingresando datos al formulario para probar el INSERT.

Fuente: Desafío Latam

Ahora con los datos escritos en los inputs, presiona el botón “Agregar” y consulta en la terminal de PostgreSQL los datos de la tabla **ejercicios** con la siguiente instrucción:

```
SELECT * FROM ejercicios;
```

```
gym=# select * from ejercicios;  
      nombre      | series | repeticiones | descanso  
-----+-----+-----+-----  
 Abdominales | 4      | 30           | 60  
(1 fila)
```

Imagen 5. Tabla ejercicios mostrada por la terminal de PostgreSQL.

Fuente: Desafío Latam

¡Felicitaciones!! ahí está, como puedes observar en la imagen anterior, se ha agregado un registro a la tabla ejercicios con los datos escritos en el formulario HTML.

### Ejercicio propuesto (3)

Desarrollar una ruta POST que reciba el payload del cliente y ejecute la función “insertar”, pasándole como argumento la data recibida del cliente y devuelva un JSON con el resultado de la consulta SQL. Además debe especificar un código de estado HTTP de 201.

## Consultado registros

### Competencia

- Construir un servidor en Node que disponibilice una ruta para consultar los registros de una tabla en PostgreSQL a través del paquete pg.

### Introducción

En el capítulo anterior aprendiste generar un nuevo registro en la tabla **ejercicios**, creando una función asíncrona y una ruta en el servidor, en este capítulo haremos algo parecido pero nos enfocaremos en la segunda sigla del CRUD, me refiero al “READ”, el cual tiene como objetivo leer los registros existentes en la base de datos.

Teniendo esta funcionalidad desarrollada estaremos a medio camino de la construcción de nuestra API REST, ya podremos crear sistemas que agreguen y obtengan datos, usando un servidor en Node y PostgreSQL como motor de bases de datos.

## Consultando registros de una tabla

Con lo aprendido en el capítulo anterior, podemos continuar con el desarrollo de nuestra API REST, en esta ocasión nos enfocaremos en la consulta de registros de nuestra tabla **ejercicios**.

### Ejercicio guiado: GET & SELECT (READ)

Al igual que la función “insertar” necesitaremos una función “consultar” en el archivo “consultas.js” que contenga la lógica para realizar la consulta SQL y obtener todos los registros de la tabla. Prosigue con los siguientes pasos para agregar la función “insertar” e iniciar con el desarrollo de esta etapa.

- **Paso 1:** Crear una función asíncrona llamada “consultar”.
- **Paso 2:** Generar con try una consulta SQL que solicite todos los registros de la tabla **ejercicios** y retorna la respuesta.
- **Paso 3:** Retornar en el bloque catch el código de error.
- **Paso 5:** Incluir entre los módulos para exportar la función creada.

```
// Paso 1
const consultar = async () => {
  // Paso 2
  try {
    const result = await pool.query("SELECT * FROM ejercicios");
    return result;
  } catch (error) {
    // Paso 3
    console.log(error.code);
    return error;
  }
};
// Paso 4
module.exports = { insertar, consultar };
```



## Ejercicio propuesto (4)

Desarrollar una función asíncrona que realice una consulta SQL para obtener todos los registros de la tabla **ejercicios** que tengan en el campo de "repeticiones" un número igual a 20.

### Ruta GET

Con la función "consultar" creada, ahora procedemos con la inclusión en el servidor de la ruta **GET /ejercicios** que utilizará esta función para devolver los registros de la tabla **ejercicios**.

- **Paso 1:** Crear la ruta **GET /ejercicios**.
- **Paso 2:** Almacenar en una constante la respuesta de la función asíncrona "consultar". No olvides agregar la importación de esta función.
- **Paso 3:** Devolver al cliente en formato JSON todos los registros obtenidos de la función "consultar".

```
// Paso 1
if (req.url == "/ejercicios" && req.method === "GET") {
  // Paso 2
  const registros = await consultar();
  // Paso 3
  res.end(JSON.stringify(registros));
}
```

Ahora veamos si hicimos todo bien, para esto simplemente vuelve a consultar el servidor, porque en el documento HTML tiene escrita una función que consultará esta ruta para cargar la tabla automáticamente al cargar el sitio web. Si entras al servidor en la ruta raíz deberás ver lo que te muestro en la siguiente imagen:

**Agregar nuevo ejercicio**

Nombre:   
 Series:   
 Repeticiones:   
 Descanso:   
 Agregar

**Editar ejercicio**

Nombre: Seleccione un ejercicio ▾  
 Series:   
 Repeticiones:   
 Descanso:   
 Editar

**Eliminar ejercicio**

Nombre: Seleccione un ejercicio ▾  
 Eliminar

Nombre	Series	Repeticiones	Descanso
Abdominales	4	30	60 segundos

Imagen 6. Tabla HTML imprimiendo los registros de la tabla **ejercicios**.

Fuente: Desafío Latam

Como puedes observar la tabla HTML se ha cargado con los registros de la tabla **ejercicios**. Si quieres comprobar de otra manera que todo salió bien, abre pestaña nueva en el navegador y consultando la dirección <http://localhost:3000/ejercicios> deberás ver lo que te muestro en la siguiente imagen:

```

1 // 20201101175335
2 // http://localhost:3000/ejercicios
3
4 {
5   "command": "SELECT",
6   "rowCount": 1,
7   "oid": null,
8   "rows": [
9     {
10      "nombre": "Abdominales",
11      "series": "4",
12      "repeticiones": "30",
13      "descanso": "60"
14    }
15  ],
16  "fields": [],
17  "_parsers": [],
18  "_types": {},
19  "RowCtor": null,
20  "rowAsArray": false
21 }
  
```

Imagen 7. JSON obtenido al consultar la ruta /ejercicios.

Fuente: Desafío Latam

Como puedes observar en la imagen anterior, estamos recibiendo el objeto "result" que nos devuelve PostgreSQL cuando realizamos una consulta SQL con el comando SELECT, este JSON es el que estamos devolviendo desde el servidor y el que contiene en la propiedad "rows" todos los registros.

### Ejercicio propuesto (5)

Desarrollar una ruta GET que devuelva un JSON con los registros de la tabla **ejercicios** usando la función "consultar". Además debe especificar un código de estado HTTP de 200.

## Actualizando registros

### Competencia

- Construir un servidor en Node que disponibilice una ruta para actualizar los registros de una tabla en PostgreSQL a través del paquete pg.

### Introducción

Actualizar un registro es indispensable para un usuario que puede humanamente equivocarse al guardar un recurso o simplemente por factores diferentes necesite cambiar algún valor. Para esto tenemos el verbo HTTP PUT, el cual coincide con el UPDATE del CRUD y tiene el objetivo de modificar los valores guardados en una o más filas de la base de datos.

Al finalizar este capítulo, estaremos a solo un paso de finalizar la creación de nuestra API REST y por consecuencia mejorar nuestra competencias como desarrolladores full stack developers.

## Actualizando registros de una tabla

Ahora que sabes como insertar y consultar registros, llega el momento de la 3era letra del famoso CRUD, me refiero al UPDATE. Recordemos que en una API REST la actualización se acostumbra a relacionar con el verbo PUT, el cual actualiza de forma definitiva un recurso, no obstante, no está de más que sepas que la actualización podría ser solo parcial, pero lo correcto técnicamente sería ocupar el verbo PATCH para esto. En nuestro caso usaremos la actualización total y por eso ocuparemos el método PUT.

Entre los formularios disponibles en el HTML tenemos uno con leyenda “Editar ejercicio”, el cual como campo de entrada para el nombre del ejercicio tiene un selector, este selector ya está programado para que se cargue a través de la ruta **GET /ejercicios**, esto es así para identificar el ejercicio que se desea editar.

### Función editar

Entonces necesitamos preparar una función asíncrona que llamaremos “editar” y una ruta que reciban los datos escritos en el formulario correspondiente y emitan la consulta SQL para actualizar ese ejercicio.

## Ejercicio guiado: PUT & UPDATE (UPDATE)

Agregar una función en el archivo “consultas.js” que reciba un arreglo con los nuevos valores del ejercicio que se desea editar, siendo el campo “nombre” el que usaremos para el condicional en la instrucción SQL.

- **Paso 1:** Crear una función asíncrona llamada editar que reciba como parámetros los datos.
- **Paso 2:** Preparar un objeto para hacer una consulta parametrizada, que actualice un registro utilizando el nombre como identificador en el comando “WHERE”. La propiedad “values” de este objeto debe ser igual al parámetro “datos”.
- **Paso 3:** Utilizar try catch para emitir la consulta SQL devolviendo un error en caso de existir.
- **Paso 4:** Incluir la función “editar” dentro del objeto que estamos exportando en “consultas.js”.

```
// Paso 1
const editar = async (datos) => {

  // Paso 2
  const consulta = {
    text: `UPDATE ejercicios SET
      nombre = $1,
      series = $2,
      repeticiones = $3,
      descanso = $4
    WHERE nombre = $1 RETURNING *`,
    values: datos,
  };

  // Paso 3
  try {
    const result = await pool.query(consulta);
    console.log(result);
    return result;
  } catch (error) {
    console.log(error);
    return error;
  }
};

// Paso 4
module.exports = { insertar, consultar, editar };
```

## Ejercicio propuesto (6)

Desarrollar una función asíncrona que realice una consulta SQL sin parámetros para actualizar algún registro de la tabla **ejercicios**.

Ruta PUT

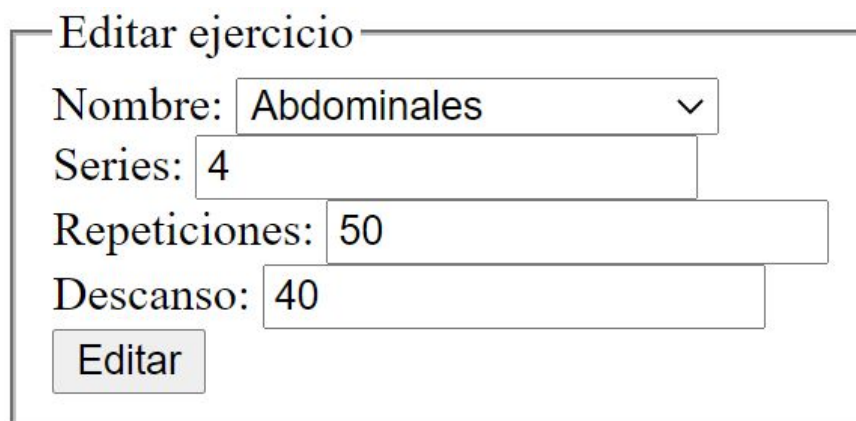
Bien, tenemos con esto la función “editar” creada, ahora vayamos con la ruta **PUT /ejercicios** en nuestro servidor. Lo que haremos básicamente será replicar la ruta **POST** cambiando solamente el método de la ruta y la función que ejecutaremos una vez tengamos la data enviada desde el cliente.

Prosigue con los siguientes pasos para el desarrollo de esta ruta:

- **Paso 1:** Crear una ruta **PUT /ejercicios** que consuma un payload enviado desde el cliente.
- **Paso 2:** Guardar en una constante la respuesta de la función asíncrona “editar” pasándole como argumento los datos convertidos en arreglo. No olvides agregar la importación de esta función.

```
// Paso 1
if (req.url == "/ejercicios" && req.method == "PUT") {
  let body = "";
  req.on("data", (chunk) => {
    body += chunk;
  });
  req.on("end", async () => {
    const datos = Object.values(JSON.parse(body));
    // Paso 2
    const respuesta = await editar(datos);
    res.end(JSON.stringify(respuesta));
  });
}
```

Ok, es momento de probar nuestra ruta para actualizar un registro en la tabla **ejercicios** basado en el campo “nombre”. Para esto ocupa la interfaz en HTML, específicamente en el formulario de leyenda “Editar ejercicio” con los datos que te muestro en la siguiente imagen:



Editar ejercicio

Nombre: Abdominales

Series: 4

Repeticiones: 50

Descanso: 40

Editar

Imagen 8. Formulario para editar con valores para probar la ruta PUT.  
Fuente: Desafío Latam

Ahora presionando el botón deberías ver en la tabla que se cambiaron los datos de “Abdominales”, así como te muestro en la siguiente imagen.

← → ↻ localhost:3000

Agregar nuevo ejercicio

Nombre:

Series:

Repeticiones:

Descanso:

Agregar

Editar ejercicio

Nombre: Seleccione un ejercicio ▾

Series:

Repeticiones:

Descanso:

Editar

Eliminar ejercicio

Nombre: Seleccione un ejercicio ▾

Eliminar

Nombre	Series	Repeticiones	Descanso
Abdominales	4	50	40 segundos

Imagen 9. Formulario para editar con valores para probar la ruta PUT.

Fuente: Desafío Latam

Ahí lo tenemos, el cambio se realizó sin problema, con la actualización lista ya llevamos 3 letras del CRUD, las cuales son Create, Read y Update. En el siguiente capítulo aprenderás a realizar la eliminación de un ejercicio almacenado en la tabla **ejercicios** y una vez logrado podremos celebrar un CRUD finalizado.

## Ejercicio propuesto (7)

Desarrollar una ruta PUT que reciba el payload enviado por la aplicación cliente y utilice la función “editar”, para emitir una consulta SQL que actualice la información de un registro y devuelva un mensaje que diga “Recurso editado con éxito”. Además, debe especificar un código de estado HTTP de 200.



## Eliminando registros

### Competencia

- Construir un servidor en Node que disponibilice una ruta para eliminar los registros de una tabla en PostgreSQL a través del paquete pg.

### Introducción

En este último capítulo aprenderemos a desarrollar la lógica para finalizar la construcción de nuestra API REST, quedando solo pendiente la funcionalidad de DELETE, es decir eliminar un registro basado en un campo especificado en el lado del cliente, con apoyo de la interfaz en HTML cedida como apoyo de la lectura.

Al finalizar el capítulo, habrás obtenido las competencias necesarias para crear el tipo de sistema más popular y cotidiano al que nos enfrentamos en el trabajo día a día, me refiero a un sistema que cree, lea, actualice y elimine registros de una base de datos, en nuestro caso en PostgreSQL.

## Eliminando registros de una tabla

Has llegado a la última etapa de la construcción de nuestra API REST y ésta se refiere al método DELETE, el cual comparte su nombre con la última letra de las siglas del CRUD.

### Ejercicio guiado: DELETE & DELETE (DELETE)

Para eliminar un registro de la tabla **ejercicios** necesitamos solamente 1 dato para identificar al ejercicio que queremos eliminar, este dato será en nuestra temática de gimnasio el campo “nombre”, es decir que con solo el nombre del ejercicio podremos emitir la consulta SQL y eliminar ese registro de nuestra base de datos.

#### Función eliminar

Ya que solo necesitamos el nombre, podremos ocupar las query strings para recibir este campo como parámetro de la consulta **DELETE**. Prosigue los siguientes pasos para el desarrollo de una función “eliminar” en nuestro archivo “consultas.js”.

- **Paso 1:** Crear una función asíncrona llamada “eliminar” que reciba un parámetro llamado “nombre”.
- **Paso 2:** Utilizar el try catch para hacer una consulta SQL que tenga por interpolación el parámetro nombre y retorne un error en caso de existir.
- **Paso 3:** Incluir la función “eliminar” en el objeto que estamos exportando del archivo “consultas.js”.

```
// Paso 1
const eliminar = async (nombre) => {
  // Paso 2
  try {
    const result = await pool.query(
      `DELETE FROM ejercicios WHERE nombre = '${nombre}'`
    );
    return result;
  } catch (error) {
    console.log(error.code);
    return error;
  }
}
```

```
};  
  
// Paso 3  
module.exports = { insertar, consultar, editar, eliminar };
```

## Ejercicio propuesto (8)

Desarrollar una función asíncrona que realice una consulta SQL para eliminar un registro de la tabla **ejercicios** e imprime por consola la cantidad de filas eliminadas.

### Ruta DELETE

Con la función “eliminar” lista, ahora solo queda desarrollar la ruta en el servidor que reciba como query string un parámetro “nombre” en la petición emitida desde el cliente. Para el desarrollo e inclusión de esta ruta y última etapa de nuestra API REST, sigue los siguientes pasos.

- **Paso 1:** Incluir la importación del módulo “url” de node para el consumo de query strings. Todas las importaciones en nuestro servidor quedaría como te muestro en el siguiente código.

```
// Paso 1  
const http = require("http");  
const url = require("url");  
const { insertar, consultar, editar, eliminar } =  
require("./consultas");  
const fs = require("fs");
```

- **Paso 2:** Crear una ruta **DELETE /ejercicios**. Considera utilizar el método “startsWith” en esta ocasión ya que no será una URL fija debido a que el valor del parámetro va a cambiar en cada ejercicio.
- **Paso 3:** Extraer de las query string la propiedad “nombre”.
- **Paso 4:** Enviar como argumento de la función asíncrona “eliminar” el nombre extraído de las query strings.

```
// Paso 2
if (req.url.startsWith("/ejercicios?") && req.method == "DELETE") {
  // Paso 3
  const { nombre } = url.parse(req.url, true).query;
  // Paso 4
  const respuesta = await eliminar(nombre);
  res.end(JSON.stringify(respuesta));
}
```

¿Cómo probar esta ruta? El tercer formulario HTML tiene un selector en el que encontrarás el ejercicio “Abdominales”, selecciónalo así como te muestro en la siguiente imagen.

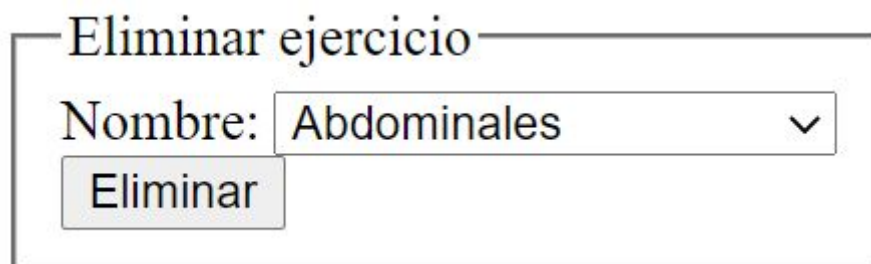


Imagen 10. Formulario para eliminar con un ejercicio seleccionado.  
Fuente: Desafío Latam

Ahora solo queda presionar el botón “eliminar” y tu tabla de ejercicios quedará de nuevo totalmente vacía, así como te muestro en la siguiente imagen:

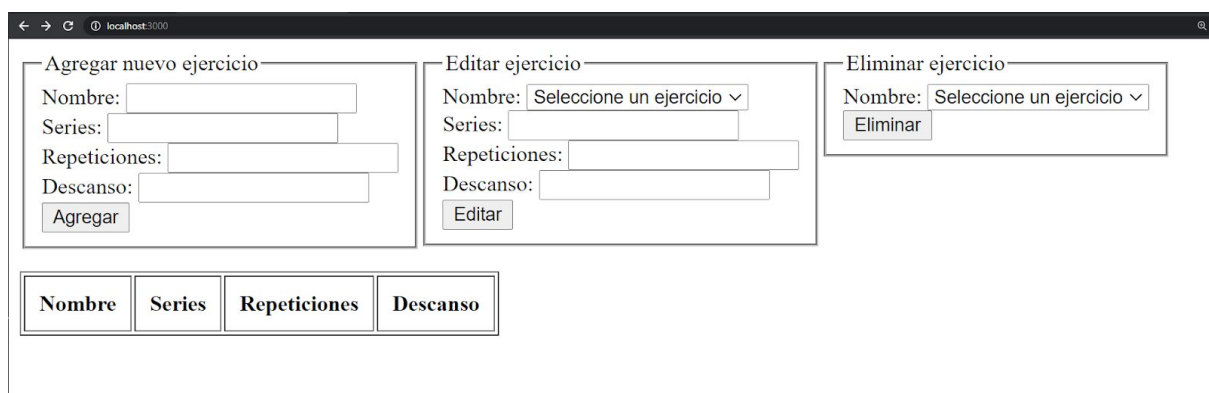


Imagen 11. Vista general de la interfaz en HTML demostrando que el ejercicio fue eliminado.  
Fuente: Desafío Latam

¡Excelente!, en el ejercicio fue eliminada de la tabla y tú has logrado terminar tu primer CRUD usando una API REST desarrollada en un servidor Node ocupando PostgreSQL como motor de base de datos.

## Ejercicio propuesto (9)

Desarrollar una ruta DELETE que reciba un parámetro “nombre” con las query strings y ejecute la función “eliminar” para eliminar un registro de la tabla **ejercicios**, además de devolver un mensaje diciendo “Registro eliminado con éxito!” y un status 200.

Con el conocimiento adquirido en esta lectura podrás empezar a considerar preparar interfaces con más funcionalidades a tus usuarios, estas funcionalidades podrían ejecutar una consulta SQL para crear un nuevo registro, actualizar datos en una tabla alojada en una base de datos, crear nuevos campos e inclusive de una nueva tabla. Sacándole todo el provecho que nos ofrece el entorno de Node y PostgreSQL, te convertirás en un profesional más capaz de dar solución a distintos problemas que se presentan en la cotidianidad laboral.

## Resumen

En esta lectura aprendimos a construir un servidor que disponibiliza rutas para gestionar registros de PostgreSQL y comenzar a programar una API REST que le permite a la aplicación cliente ejecutar funcionalidades como crear, consultar, actualizar y eliminar datos alojados en una base de datos, hemos abordamos los siguientes contenidos:

- Preparación de un servidor para la creación de una API REST que permita ejecutar las funciones de un CRUD.
- Desarrollo de una función asíncrona “insertar” y una ruta **POST** para la inserción de un nuevo registro en la tabla, complementando el comportamiento del CREATE.
- Desarrollo de una función asíncrona “consultar” y una ruta **GET** para la obtención de los registros de una tabla, complementando el comportamiento del READ.
- Desarrollo de una función asíncrona “editar” y una ruta **PUT** para la actualización de un registro de la tabla , complementando el comportamiento del UPDATE.
- Desarrollo de una función asíncrona “eliminar” y una ruta **DELETE** para la eliminación de un registro de la tabla , complementando el comportamiento del DELETE.

## Solución de los ejercicios propuesto

1. Desarrollar un servidor en Node que disponibilice una ruta GET **/fecha** que devuelva un string al cliente con una fecha enviada desde PostgreSQL

```
// consultas.js
const { Pool } = require("pg");

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

const getDate = async () => {
  const result = await pool.query("SELECT NOW()");
  return result.rows[0].now;
};

module.exports = { getDate };
```

```
// index.js
const http = require("http");
const { getDate } = require("./consultas");
http
  .createServer(async (req, res) => {
    if (req.url == "/" && req.method === "GET") {
      const result = await getDate();
      res.end(result.toString());
    }
  })
  .listen(3000);
```

2. Desarrollar una función que reciba asíncrona que realice una consulta SQL para insertar un registro con los datos recibidos como parámetros y retorna el último registro insertado en forma de arreglo.

```
const insertar = async (datos) => {
  const consulta = {
    text: "INSERT INTO ejercicios values($1, $2, $3, $4) RETURNING *",
    values: datos,
    rowMode: "array"
  };
  try {
    const result = await pool.query(consulta);
    console.log(result.rows[0])
    return result;
  } catch (error) {
    console.log(error.code);
    return error;
  }
};
```

3. Desarrollar una ruta POST que reciba el payload del cliente y ejecute la función "insertar" pasándole como argumento la data recibida del cliente y devuelve un JSON con el resultado de la consulta SQL. Además debe especificar un código de estado HTTP de 201.

```
if (req.url == "/ejercicios" && req.method == "POST") {
  let body = "";
  req.on("data", (chunk) => {
    body += chunk;
  });
  req.on("end", async () => {
    const datos = Object.values(JSON.parse(body));
    const respuesta = await insertar(datos);
    res.statusCode = 201
    res.end(JSON.stringify(respuesta));
  });
}
```

4. Desarrollar una función asíncrona que realice una consulta SQL para obtener todos los registros de la tabla **ejercicios** que tengan en el campo de "repeticiones" un número igual a 20.

```
const consultar = async () => {
  try {
    const result = await pool.query(
      "SELECT * FROM ejercicios WHERE repeticiones = '20';"
    );
    return result;
  } catch (error) {
    console.log(error);
    return error;
  }
};
```

5. Desarrollar una ruta GET que devuelva un JSON con los registros de la tabla **ejercicios** usando la función "consultar". Además debe especificar un código de estado HTTP de 200.

```
if (req.url == "/ejercicios" && req.method === "GET") {
  const registros = await consultar();
  res.statusCode = 200;
  res.end(JSON.stringify(registros));
}
```

6. Desarrollar una función asíncrona que realice una consulta SQL sin parámetros para actualizar algún registro de la tabla **ejercicios**.

```
const editar = async (datos) => {
  console.log(datos)
  try {
    const result = await pool.query(`UPDATE ejercicios SET
      nombre = '${datos[0]}',
      series = '${datos[1]}',
      repeticiones = '${datos[2]}',
      descanso = '${datos[3]}'
      WHERE nombre = '${datos[0]}' RETURNING *`);
    return result;
  } catch (error) {
    console.log(error);
    return error;
  }
};
```



7. Desarrollar una ruta PUT que reciba el payload enviado por la aplicación cliente y utilice la función “editar” para emitir una consulta SQL que actualice la información de un registro y devuelva un mensaje que diga “Recurso editado con éxito” . Además debe especificar un código de estado HTTP de 200.

```
if (req.url == "/ejercicios" && req.method == "PUT") {
  let body = "";
  req.on("data", (chunk) => {
    body += chunk;
  });
  req.on("end", async () => {
    const datos = Object.values(JSON.parse(body));
    await editar(datos);
    res.statusCode = 200;
    res.end("Recurso editado con éxito!");
  });
}
```

8. Desarrollar una función asíncrona que realice una consulta SQL para eliminar un registro de la tabla **ejercicios** y retorne la cantidad de filas eliminadas.

```
const eliminar = async (nombre) => {
  try {
    const result = await pool.query(
      `DELETE FROM ejercicios WHERE nombre = '${nombre}' RETURNING *`
    );
    console.log(result.rowCount)
    return result.rowCount;
  } catch (error) {
    console.log(error.code);
    return error;
  }
};
```

9. Desarrollar una ruta DELETE que reciba un parámetro “nombre” con las query strings y ejecute la función “eliminar” para eliminar un registro de la tabla **ejercicios**, además de devolver un mensaje diciendo “Registro eliminado con éxito!” y un status 200.

```
if (req.url.startsWith("/ejercicios?") && req.method == "DELETE") {  
  const { nombre } = url.parse(req.url, true).query;  
  await eliminar(nombre);  
  res.statusCode = 200;  
  res.end("Registro eliminado con éxito!");  
}
```