

## Guía de estudio 3 - Acceso a Base de Datos con Node (Parte I)



¡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.

Dentro de los que se encuentran:

- Insertar datos en una tabla alojada en PostgreSQL usando el paquete pg.
- Mostrar por consola datos alojados en PostgreSQL usando el paquete pg.
- Crear una ruta GET con Express para devolver los registros de una tabla alojada en PostgreSQL.
- Crear una ruta POST con Express que reciba y almacene en PostgreSQL un nuevo registro.

**¡Vamos con todo!**



## Tabla de contenidos

Arquitectura de una aplicación con bases de datos	3
El paquete pg	4
Instalación del paquete pg	4
Primera consulta SQL con Node	5
El objeto result	6
Primer registro en PostgreSQL desde Node	7
¡Manos a la obra! - Plan de viajes	8
Obteniendo registros de PostgreSQL desde Node	9
¡Manos a la obra! - Equipamiento.js	9
API REST con PostgreSQL(GET)	9
¡Manos a la obra! - Server.js	11
API REST con PostgreSQL(POST)	12
¡Manos a la obra! - Server.js (Parte 2)	13
¿Qué es un SQL Injection?	13
Veamos un ejemplo	14
¿Cómo se previene un ataque de inyección SQL?	15
<b>¡Manos a la obra!</b>	<b>15</b>
Preguntas para una entrevista laboral	16



**¡Comencemos!**

En esta unidad aprenderemos a crear un servidor Rest con ExpressJS que lea y guarde los datos de un pedido en una base de datos.

## Arquitectura de una aplicación con bases de datos

En una aplicación con node y express usualmente atenderemos pedidos de un cliente como lo sería una página web creada con React o una aplicación creada en Android o IOS. También es muy probable que este pedido incluya guardar información, por ejemplo cuando se crea un usuario nuevo en un sistema, o listar información, como por ejemplo cuando mostramos todos los artículos de un blog. Esta información se encuentra dentro de una base de datos y con Express nos conectaremos a esta base de datos, realizaremos la operación deseada y enviaremos al cliente los datos pedidos.

En esta unidad trabajaremos con PostgreSQL, pero se pueden ocupar otros motores de bases de datos como por ejemplo MySQL, Oracle o incluso bases de datos no relacionales como MongoDB.

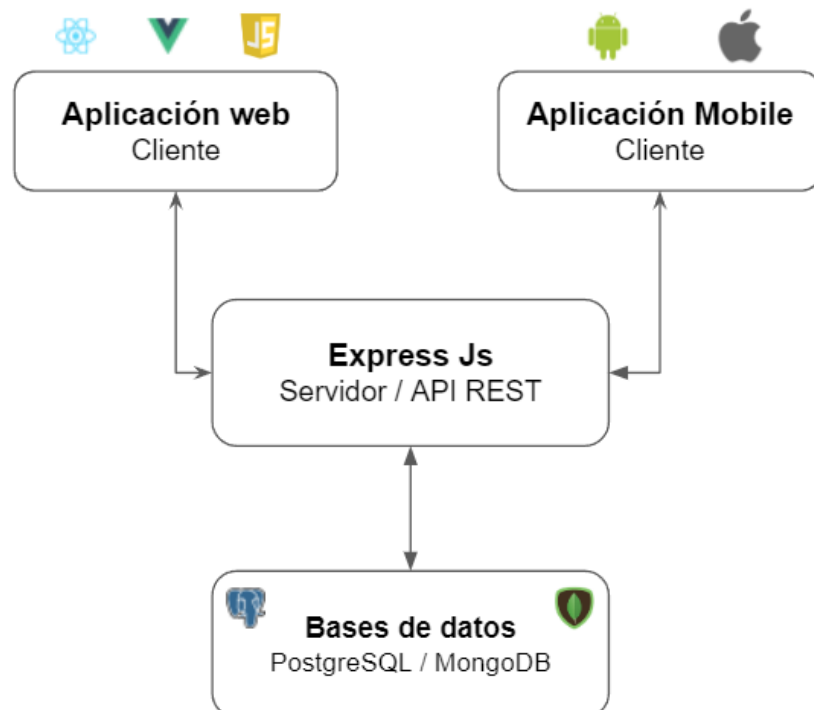


Imagen 1. Arquitectura básica de una aplicación con base de datos.  
Fuente: Desafío Latam

## El paquete pg

El paquete pg nos permite conectarnos e interactuar con una base de datos PostgreSQL y está disponible en NPM.

Con Node y Express podemos crear aplicaciones que pidan o guarden datos en distintos motores de bases de datos. En esta unidad aprenderemos a hacerlo con PostgreSQL



Imagen 2. Comunicación entre Node y PostgreSQL a través del paquete pg  
Fuente: Desafío Latam

## Instalación del paquete pg

Para empezar a utilizar el paquete pg primero hay que instalarlo por npm, para esto crea una nueva carpeta, dentro de la carpeta ejecuta la siguiente línea de comando para iniciar un proyecto NPM:

```
npm init -y
```

Ahora instalemos el paquete pg con la siguiente línea de comando:

```
npm install pg
```

Luego de instalarse deberás ver “pg” en el archivo package.json

```
"dependencies": {  
  "pg": "^8.7.3"  
}
```

Imagen 3. Instalación del paquete pg en un proyecto npm  
Fuente: Desafío Latam

## Primera consulta SQL con Node

Para empezar a hacer consultas SQL desde Node, crearemos una base de datos que tenga como objetivo almacenar los destinos y el presupuesto de un plan de viajes vacacionales.

Abre la terminal **psql** y escribe las siguientes instrucciones para crear una base de datos **plan\_de\_viajes** y una tabla **viajes**:

```
CREATE DATABASE plan_de_viajes;

\c plan_de_viajes;

CREATE TABLE viajes (id SERIAL, destino VARCHAR(50) NOT NULL,
presupuesto INT NOT NULL);
```

Ahora que tenemos la base de datos, realicemos nuestra primera consulta SQL desde Node haciendo lo siguiente:

1. Crear un archivo **consultas.js**
2. Importar la clase Pool del paquete pg
3. Crear una instancia de la clase Pool usando un objeto de configuración con las credenciales.
4. Crear una función **getDate**
5. Usar el método **query()** para emitir una consulta que devuelva la fecha actual con la función NOW()
6. Mostrar el resultado de la consulta por consola.

```
const { Pool } = require('pg')

const pool = new Pool({
  host: 'localhost',
  user: 'postgres',
  password: 'postgres',
  database: 'plan_de_viajes',
  allowExitOnIdle: true
})

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

Entendiendo el código anterior:

- La clase Pool nos permite soportar multiconexiones y un mejor rendimiento en las consultas

```
const { Pool } = require('pg')
```

- Esta propiedad le indicará a PostgreSQL que cierre la conexión luego de cada consulta

```
allowExitOnIdle: true
```

- Cada consulta devuelve un objeto result con el detalle obtenido en su ejecución

```
const result = await pool.query("SELECT NOW()")
```

## El objeto result

Cada consulta que realizamos nos devuelve un objeto result, en el cual resaltan las siguientes propiedades:

- **command:** El comando SQL que se utilizó en la consulta.
- **rowCount:** Cantidad de filas procesadas en la consulta.
  - La cantidad de filas procesadas no es lo mismo que la cantidad de filas devueltas, por ejemplo un comando update puede afectar múltiples filas, pero devolver un arreglo vacío.
- **rows:** Un arreglo de objetos con todos los resultados o filas obtenido en la consulta.
- **fields:** La estructura de cada uno de los campos o columnas en las filas obtenidas.

```
Result {
  command: 'SELECT',
  rowCount: 1,
  oid: null,
  rows: [ { now: 2022-05-02T20:00:56.197Z } ],
  fields: [
    Field {
      name: 'now',
      tableID: 0,
      columnID: 0,
      dataTypeID: 1184,
      dataTypeSize: 8,
      dataTypeModifier: -1,
      format: 'text'
    }
  ],
  _parsers: [ [Function: parseDate] ],
  _types: TypeOverrides {
    _types: {
      getTypeParser: [Function: getTypeParser],
      setTypeParser: [Function: setTypeParser],
      arrayParser: [Object],
      builtins: [Object]
    },
    text: {},
    binary: {}
  },
  RowCtor: null,
  rowAsArray: false
}
```

Imagen 4. El objeto Result  
Fuente: Desafío Latam

## Primer registro en PostgreSQL desde Node

Registremos un primer viaje agregando en nuestro script la siguiente función:

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

En esta ocasión estamos usando el método **query()** para hacer una consulta parametrizada que nos ayuda a evitar un problema llamado SQL Injection, el cual se discute en la guía de esta unidad.

En donde cada parámetro se representa por el símbolo del dólar(\$) seguido del orden en el que se declaran sus valores en un segundo argumento(values).

El método **query()** recibe en esta ocasión estos 2 argumentos.

```
pool.query(<consulta parametrizada>, <arreglo de valores>)
```

Ejecutemos la función `agregarViaje` pasando como argumentos el destino "Valdivia" y como presupuesto 150000

```
agregarViaje("Valdivia", 150000)
```

Y luego revisamos manualmente en la terminal `psql` si el registro se realizó con éxito usando la siguiente consulta:

```
plan_de_viajes=# SELECT * FROM viajes;
 id | destino  | presupuesto
----+-----+-----
  1 | Valdivia |    150000
(1 fila)
```

Imagen 5. Verificando el primer registro en PostgreSQL  
Fuente: Desafío Latam



## ¡Manos a la obra! - Plan de viajes

Ejecuta la siguiente consulta SQL en la base de datos **plan\_de\_viajes** para crear una tabla que registre un inventario del equipamiento que se piensa llevar en las vacaciones:

```
CREATE TABLE equipamiento (id SERIAL, nombre VARCHAR(50));
```

Luego:

1. Crea un nuevo script de nombre **equipamiento.js**
2. Importa la clase **Pool** del paquete `pg`
3. Crea una instancia de la clase **Pool** con las credenciales de la base de datos
4. Crea una función llamada **agregarEquipamiento(nombre)** que realice una consulta parametrizada para hacer un nuevo registro en la tabla correspondiente.
5. Ejecuta la función y revisa manualmente si el registro se logró con éxito.



## Obteniendo registros de PostgreSQL desde Node

Para obtener los registros almacenados en PostgreSQL crea una nueva función en el archivo consultas.js que muestre por consola y retorne las filas de la tabla *viajes*:

```
const obtenerViajes = async () => {  
  const { rows } = await pool.query("SELECT * FROM viajes")  
  console.log(rows)  
  return rows  
}  
  
obtenerViajes()
```

Ejecutando esta función veremos por consola el viaje a Valdivia registrado anteriormente

```
$ node consultas.js  
[ { id: 1, destino: 'Valdivia', presupuesto: 150000 } ]
```

Imagen 6. Mostrando registros de viajes por consola  
Fuente: Desafío Latam



## ¡Manos a la obra! - Equipamiento.js

En el archivo **equipamiento.js** crea una función que muestre por consola y retorne los equipamientos registrados en la tabla equipamiento.

Luego ejecuta la función y observa por consola si se muestra el equipamiento que registraste en el ejercicio anterior.

```
$ node equipamiento.js  
[ { id: 1, nombre: 'Selfie Stick' } ]
```

Imagen 7. Mostrando registros de equipamientos por consola  
Fuente: Desafío Latam

## API REST con PostgreSQL(GET)

Ahora que podemos interactuar con una base de datos PostgreSQL para obtener y hacer registros, unamos los conocimientos previos para crear una API REST con Express que ofrezca una ruta GET /viajes que devuelva todos los viajes registrados en la tabla.

Para esto será necesario que el archivo `consultas.js` exporte las funciones que creamos recientemente. Agrega la siguiente línea de código al final del archivo.

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

Ahora que podemos acceder a las funciones como módulos, procedamos con la creación de la API REST haciendo lo siguiente:

1. Crea un archivo ***index.js***
2. Instala express
3. Importa el paquete express y las funciones de ***consultas.js***
4. Crea un servidor en el puerto 3000
5. Crea una ruta **GET /viajes** que utilice la función **obtenerViajes** para devolver los registros a una aplicación cliente.

```
const { agregarViaje, obtenerViajes } = require('./consultas')
const express = require('express');
const app = express();

app.listen(3000, console.log("SERVIDOR ENCENDIDO"))

app.get("/viajes", async (req, res) => {
  const viajes = await obtenerViajes()
  res.json(viajes)
})
```

Levanta el servidor ejecutando el script **index.js** por la terminal y utiliza la extensión Thunder Client para consultar la ruta:

**localhost:3000/viajes**

Y deberás recibir un arreglo de objetos con el viaje de Valdivia registrado anteriormente

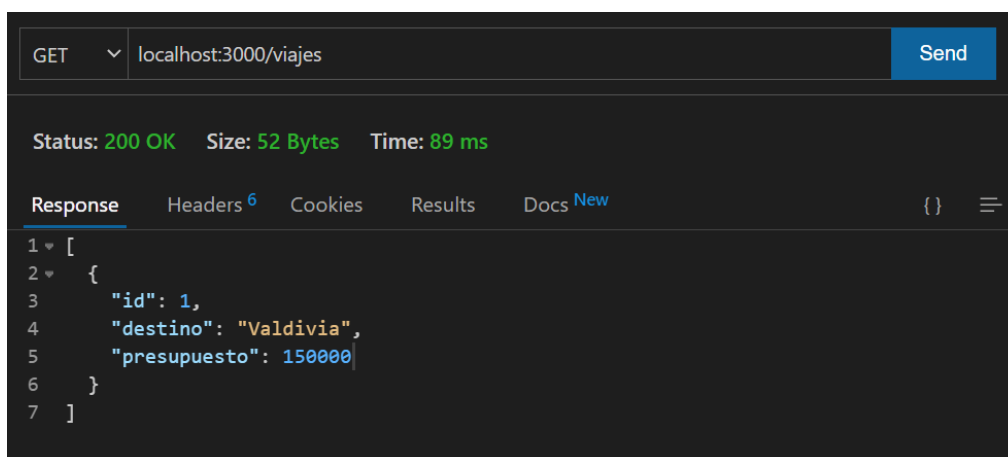


Imagen 8. Consultando ruta **GET /viajes**  
Fuente: Desafío Latam



## ¡Manos a la obra! - Server.js

En el archivo **equipamiento.js** exporta las funciones creadas para agregar y obtener los registros de la tabla equipamiento.

Luego crea un archivo **server.js** en donde deberás:

1. Importar las funciones del archivo **equipamiento.js**
2. Importar el paquete express

3. Crear un servidor en el puerto 3001
4. Crear una ruta **GET /equipamientos** que devuelva los equipamientos registrados en la base de datos.

## API REST con PostgreSQL(POST)

Para permitir desde nuestro servidor la posibilidad de registrar nuevos equipamientos procedamos a crear una ruta **POST /viajes** que utilice la función **agregarViaje()** y un payload recibido en la consulta.

Al inicio del código agrega el middleware que nos permite parsear el cuerpo de la consulta:

```
app.use(express.json())
```

Y al final del script creamos la ruta **POST /viajes**:

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

Para probar la ruta creada realicemos otra consulta con Thunder Client usando el método **POST** y agreguemos como cuerpo un objeto con un destino y un presupuesto:

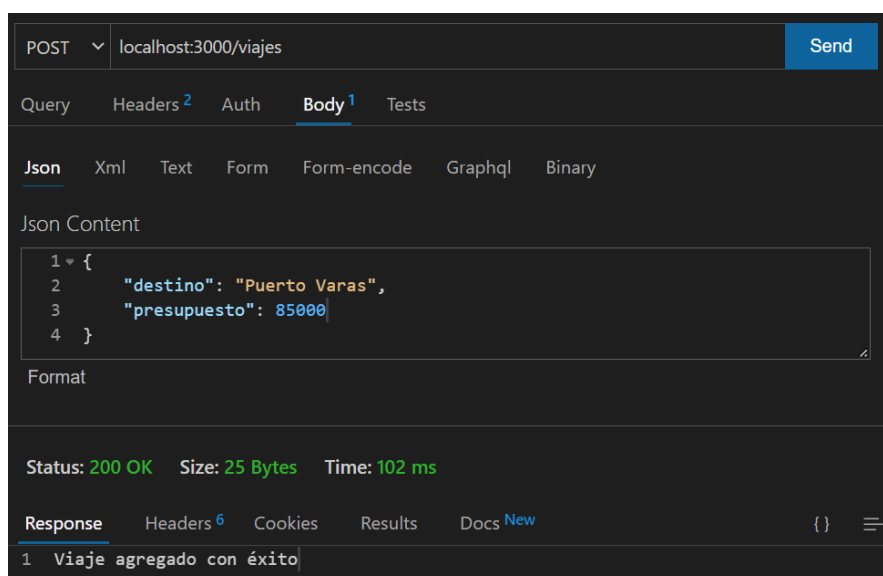
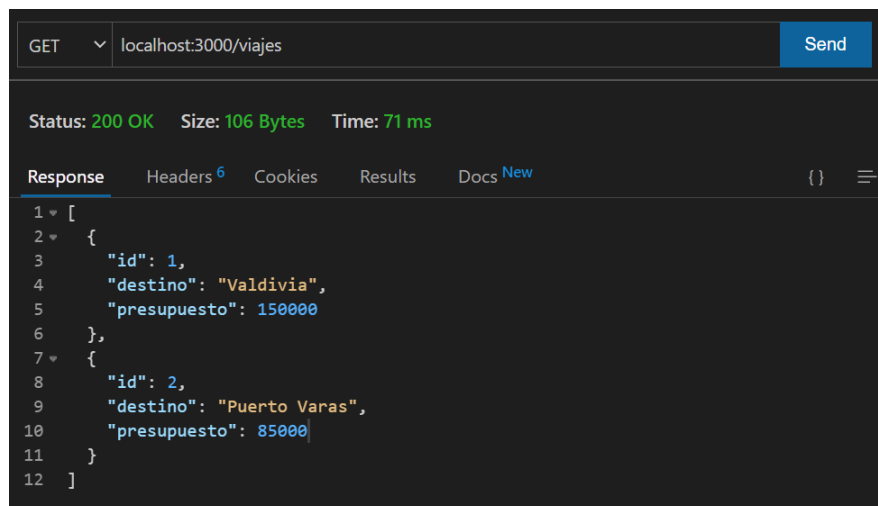


Imagen 9. Probando la ruta **POST /viajes**  
Fuente: Desafío Latam

Podemos confirmar que el viaje fue agregado con éxito revisando manualmente en la terminal **psql** o realizando otra consulta **GET /viajes**



```
GET localhost:3000/viajes Send

Status: 200 OK Size: 106 Bytes Time: 71 ms

Response Headers Cookies Results Docs New {} ≡

1 [
2   {
3     "id": 1,
4     "destino": "Valdivia",
5     "presupuesto": 150000
6   },
7   {
8     "id": 2,
9     "destino": "Puerto Varas",
10    "presupuesto": 85000
11  }
12 ]
```

Imagen 10. Verificando nuevo viaje registrado al consultar la ruta **GET /viajes**  
Fuente: Desafío Latam.



## ¡Manos a la obra! - Server.js (Parte 2)

En el archivo **server.js**:

1. Agrega el middleware para parsear el payload de la consulta.
2. Crea una ruta **POST /equipamientos**
3. Usa la función creada para agregar un nuevo equipamiento en base de datos.
4. Consulta la ruta **GET /equipamientos** para verificar que el registro fue realizado con éxito.

## ¿Qué es un SQL Injection?

La inyección de SQL es una técnica que consiste en inyectar código SQL a partir de un input, este puede ser un formulario de una página web o un valor que enviemos a una API REST desde un cliente. El código inyectado se ejecuta en el servidor y puede modificar accesos de usuarios, insertar registros o incluso borrar tablas.

## Veamos un ejemplo

Supongamos que tenemos un query sencillo donde buscamos un registro de nuestra base de datos a partir de uno de sus campos, por ahora probaremos con el id, pero podría ser cualquier otro.

```
const { Pool } = require('pg')

const pool = new Pool({
  host: 'localhost',
  user: 'postgres',
  password: 'postgres',
  database: 'plan_de_viajes',
  allowExitOnIdle: true
})

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

Si llamamos a la función listarViajes con un id que exista en nuestros registros, obtendremos el resultado deseado. En este caso el objeto correspondiente al primer registro de mi base de datos.

```
rows: [
  { id: 1, destino: 'Valdivia', presupuesto: 15000 }
]
```

Pero aquí viene la parte interesante. Si llamamos a la función de la siguiente forma: `listarViajes('1 or 1=1')`. Obtendremos todos los registros de la tabla, en lugar de uno solo. En mi caso sería el siguiente:

```
rows: [
  { id: 1, destino: 'Valdivia', presupuesto: 15000 },
  { id: 2, destino: 'Puerto Montt', presupuesto: 17000 }
]
```

Incluso ni siquiera necesitamos un id existente en la base de datos, podríamos buscar cualquiera, puesto que se tiene que cumplir una condición o la otra, y 1 siempre será igual a 1.

Este tipo de problemas podrían introducirse en nuestra API y podrían permitir fácilmente que alguien ingrese sin un password válido con la cuenta de cualquier usuario o incluso borre todos los registros de nuestra base de datos.

Para evitar este problema tenemos que siempre utilizar consultas con parámetros.

## ¿Cómo se previene un ataque de inyección SQL?

Sanear los parámetros. Sanear quiere decir limpiar aquellos elementos que puedan ser perjudiciales. Es por esto mismo que al momento de hacer la consulta en Node utilizamos el método `.query` y le pasamos los valores dentro de un arreglo, de esta forma evitamos potenciales inyecciones SQL.

```
const listarViajes = async (id) => {  
  const consulta = 'SELECT * FROM viajes WHERE id = $1'  
  const result = await pool.query(consulta, [id])  
  console.log(result)  
}
```

Luego, si intentamos llamar a la función con `id = 1` obtendremos el registro buscado, pero si intentamos llamar a la función con la inyección SQL obtendremos un error. Prueba intentándolo con `listarViajes('1 or 1=1')`



## ¡Manos a la obra!

Repliquemos el problema pero esta vez ocupando Express.

- Crea una base de datos nuevos con la tabla usuarios con id, email y password.
- Crea una aplicación en node con express y pg.
- Crea el archivo Usuarios.js con la función buscarUsuario que busque un usuario por email, no utilices queries parametrizados.
- Crea el archivo index.js con el endpoint buscar que reciba como parámetro un email.
- Desde index.js carga la función buscarUsuario y devuelve los resultados obtenidos en la función buscarUsuario como JSON.
- Llama al endpoint *buscar* utilizando como parámetro un usuario con email existente dentro de la BD para probar que funcione.
- Intenta hacer un SQL injection buscando un email no existente o `1=1`

## Preguntas para una entrevista laboral

- ¿Qué es una inyección SQL?
- ¿Cómo se previene una inyección SQL en node?
- ¿Cuál es el problema con el siguiente código?

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