

Proyecto3A_Server_Backend

Generado por Doxygen 1.12.0

1 Sensor Management API	1
1.1 Tabla de Contenidos	1
1.2 Características	1
1.3 Tecnologías Usadas	2
1.4 Estructura del Proyecto	2
1.5 Requisitos Previos	2
1.6 Configuración del Entorno	2
1.7 Uso	2
1.8 API Endpoints	3
1.9 Contribuciones	3
1.10 Licencia	3
2 Índice de archivos	5
2.1 Lista de archivos	5
3 Documentación de archivos	7
3.1 Referencia del archivo docker-compose-example-main/db.js	7
3.1.1 Descripción detallada	7
3.1.2 Documentación de variables	7
3.1.2.1 const	7
3.1.2.2 exports	7
3.1.2.3 pool	8
3.2 Referencia del archivo docker-compose-example-main/server.js	8
3.2.1 Descripción detallada	9
3.2.2 Documentación de funciones	10
3.2.2.1 delete() [1/3]	10
3.2.2.2 delete() [2/3]	10
3.2.2.3 delete() [3/3]	10
3.2.2.4 get() [1/3]	11
3.2.2.5 get() [2/3]	11
3.2.2.6 get() [3/3]	12
3.2.2.7 listen()	12
3.2.2.8 post() [1/2]	12
3.2.2.9 post() [2/2]	13
3.2.2.10 use() [1/2]	13
3.2.2.11 use() [2/2]	13
3.2.3 Documentación de variables	14
3.2.3.1 app	14
3.2.3.2 cors	14
3.2.3.3 express	14
3.2.3.4 pool	14
3.2.3.5 port	14
3.3 Referencia del archivo docker-compose-example-main/README.md	14

3.4 Referencia del archivo README.md	14
Índice alfabético	15

Capítulo 1

Sensor Management API

Este proyecto es una API para gestionar usuarios y sensores utilizando Node.js y PostgreSQL. Permite la creación, consulta, inserción y eliminación de datos de sensores, así como la administración de usuarios.

1.1. Tabla de Contenidos

- Características
- Tecnologías Usadas
- Estructura del Proyecto
- Requisitos Previos
- Configuración del Entorno
- Uso
- API Endpoints
- Contribuciones
- Licencia

1.2. Características

- Gestión de usuarios: crear, listar y eliminar usuarios.
- Gestión de sensores: agregar, listar y eliminar mediciones de sensores.
- API RESTful para interactuar con los datos.
- Base de datos PostgreSQL para almacenamiento persistente.
- Contenerización con Docker.

1.3. Tecnologías Usadas

- Node.js
- Express
- PostgreSQL
- Docker
- Docker Compose

1.4. Estructura del Proyecto

```
```\n/docker-compose-example-main\n- db.js # Configuración de la conexión a la base de datos.\n- server.js # Código principal de la API.\n- Dockerfile # Archivo para construir la imagen Docker de la aplicación.\n- docker-compose.yml # Archivo de configuración para Docker Compose.\n- package.json # Dependencias y scripts de la aplicación.\n- README.md # Documentación del proyecto.\n```\n
```

## 1.5. Requisitos Previos

Antes de comenzar, asegúrate de tener instalados los siguientes programas:

- `Node.js`
- `Docker`
- `Docker Compose`

## 1.6. Configuración del Entorno

1. Clona el repositorio:  

```
git clone https://github.com/tu_usuario/sensor-management-api.git\n cd sensor-management-api
```
2. Construye y ejecuta los contenedores:  

```
docker-compose up --build
```

Esto levantará los contenedores para la base de datos y la aplicación.

## 1.7. Uso

Una vez que los contenedores estén en funcionamiento, la API estará disponible en `http://localhost↵:13000`.

## 1.8. API Endpoints

- GET /setup: Crea las tablas de usuarios y sensores en la base de datos.
- GET /latest: Obtiene las últimas mediciones de temperatura y CO2.
- GET /: Devuelve todos los usuarios y sensores.
- POST /: Inserta un nuevo sensor.
  - Body: { "type": "temperature", "value": 25.5, "timestamp": "2024-09-22T12:00:00Z", "userId": 1 }
- POST /users: Crea un nuevo usuario.
  - Body: { "username": "nuevo\_usuario" }
- DELETE /users/:id/measurements: Elimina todas las mediciones de un usuario específico.
- DELETE /reset: Reinicia las tablas de la base de datos.
- DELETE /erase: Elimina todas las tablas.

## 1.9. Contribuciones

Las contribuciones son bienvenidas. Si deseas contribuir a este proyecto, por favor abre un issue o envía un pull request.

## 1.10. Licencia

Este proyecto está bajo la Licencia MIT. Consulta el archivo LICENSE para más información.





## Capítulo 2

# Índice de archivos

### 2.1. Lista de archivos

Lista de todos los archivos con breves descripciones:

docker-compose-example-main/ <a href="#">db.js</a>	
Configuración de la conexión a la base de datos PostgreSQL . . . . .	7
docker-compose-example-main/ <a href="#">server.js</a>	
API para gestionar usuarios y sensores, incluyendo funcionalidades de creación, consulta, inserción y eliminación de datos . . . . .	8



## Capítulo 3

# Documentación de archivos

### 3.1. Referencia del archivo docker-compose-example-main/db.js

Configuración de la conexión a la base de datos PostgreSQL.

#### Variables

- `const { Pool } = require('pg')`
- `const pool`  
*Instancia de conexión a la base de datos PostgreSQL.*
- `module exports = pool`  
*Exporta el pool de conexiones para que otros módulos lo utilicen.*

#### 3.1.1. Descripción detallada

Configuración de la conexión a la base de datos PostgreSQL.

Este archivo configura y exporta un pool de conexiones a PostgreSQL utilizando el módulo `pg` para gestionar las interacciones con la base de datos.

#### 3.1.2. Documentación de variables

##### 3.1.2.1. `const`

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

##### 3.1.2.2. `exports`

```
module exports = pool
```

Exporta el pool de conexiones para que otros módulos lo utilicen.

### 3.1.2.3. pool

```
const pool
```

#### Valor inicial:

```
= new Pool({
 host: 'db',
 port: 5432,
 user: 'user123',
 password: 'password123',
 database: 'db123'
})
```

Instancia de conexión a la base de datos PostgreSQL.

El pool de conexiones permite reutilizar conexiones a la base de datos para mejorar el rendimiento. La configuración incluye el host, puerto, usuario, contraseña y nombre de la base de datos.

#### Parámetros

<code>{string}</code>	host - Dirección del host de la base de datos.
<code>{number}</code>	port - Puerto en el que escucha la base de datos.
<code>{string}</code>	user - Nombre de usuario para la conexión.
<code>{string}</code>	password - Contraseña para la conexión.
<code>{string}</code>	database - Nombre de la base de datos.

## 3.2. Referencia del archivo docker-compose-example-main/server.js

API para gestionar usuarios y sensores, incluyendo funcionalidades de creación, consulta, inserción y eliminación de datos.

#### Funciones

- `app use (cors())`  
*Habilita CORS para permitir solicitudes desde diferentes dominios.*
- `app use (express.json())`  
*Habilita el middleware para procesar JSON en las solicitudes.*
- `app get ('/setup', async(req, res)=> { try { await pool.query(` CREATE TABLE IF NOT EXISTS users(id SERIAL PRIMARY KEY, username VARCHAR(100) NOT NULL);CREATE TABLE IF NOT EXISTS sensors(id SERIAL PRIMARY KEY, type VARCHAR(100), value FLOAT, timestamp TIMESTAMP, user_id INTEGER REFERENCES users(id) ON DELETE CASCADE);`);res.status(200).send({ message:"Successfully created users and sensors tables" });} catch(err) { console.log(err);res.sendStatus(500);} })`  
*Ruta para crear las tablas 'users' y 'sensors' en la base de datos si no existen.*
- `app get ('/latest', async(req, res)=> { try { const temperatureQuery=await pool.query(` SELECT *FROM sensors WHERE type='temperature' ORDER BY timestamp DESC LIMIT 1 `);const co2Query=await pool.query(` SELECT *FROM sensors WHERE type='CO2' ORDER BY timestamp DESC LIMIT 1 `);const responseData={ temperature:temperatureQuery.rows[0]||null, co2:co2Query.rows[0]||null };res.↵ status(200).send(responseData);} catch(err) { console.log(err);res.sendStatus(500);} })`  
*Ruta para obtener las últimas mediciones de temperatura y CO2.*
- `app get ('/', async(req, res)=> { try { const sensorsQuery=await pool.query('SELECT *FROM sensors');const usersQuery=await pool.query('SELECT *FROM users');const responseData={ sensors:sensorsQuery.↵ rows, users:usersQuery.rows };res.status(200).send(responseData);} catch(err) { console.log(err);res.send↵ Status(500);} })`

*Ruta principal para obtener todos los sensores y usuarios.*

- `app.post('/', async(req, res)=> { const { type, value, timestamp, userId }=req.body;try { const userExists=await pool.query('SELECT *FROM users WHERE id=$1', [userId]);if(userExists.rows.length===0) { return res.status(400).send({ message:"User does not exist" });} await pool.query('INSERT INTO sensors(type, value, timestamp, user_id) VALUES($1, $2, $3, $4)', [type, value, timestamp, userId]);res.status(200).send({ message:"Successfully added sensor data" });} catch(err) { console.log(err);res.sendStatus(500);} })`

*Ruta para insertar datos de un sensor.*

- `app.post('/users', async(req, res)=> { const { username }=req.body;try { await pool.query('INSERT INTO users(username) VALUES($1)', [username]);res.status(200).send({ message:"Successfully added user" });} catch(err) { console.log(err);res.sendStatus(500);} })`

*Ruta para insertar un nuevo usuario.*

- `app.delete('/users/:id/measurements', async(req, res)=> { const { id }=req.params;try { const result=await pool.query('DELETE FROM sensors WHERE user_id=$1', [id]);res.status(200).send({ message:"Successfully deleted measurements for user" });} catch(err) { console.log(err);res.sendStatus(500);} })`

*Ruta para eliminar todas las mediciones asociadas a un usuario.*

- `app.delete('/reset', async(req, res)=> { try { await pool.query('DROP TABLE IF EXISTS sensors');await pool.query('DROP TABLE IF EXISTS users');await pool.query(' CREATE TABLE users(id SERIAL PRIMARY KEY, username VARCHAR(100) NOT NULL);CREATE TABLE sensors(id SERIAL PRIMARY KEY, type VARCHAR(100), value FLOAT, timestamp TIMESTAMP, user_id INTEGER REFERENCES users(id) ON DELETE CASCADE);');await pool.query('INSERT INTO users(username) VALUES($1),($2)', ['user1', 'user2']);await pool.query(' INSERT INTO sensors(type, value, timestamp, user_id) VALUES($1, $2, $3, $4),($5, $6, $7, $8) ', ['temperature', 25.5, '2024-09-22T12:00:00Z', 1, 'humidity', 60.0, '2024-09-22T12:00:00Z', 2]);res.status(200).send({ message:"Successfully reset users and sensors tables with default data" });} catch(err) { console.log(err);res.sendStatus(500);} })`

*Ruta para reiniciar las tablas de la base de datos.*

- `app.delete('/erase', async(req, res)=> { try { await pool.query('DROP TABLE IF EXISTS sensors');await pool.query('DROP TABLE IF EXISTS users');res.status(200).send({ message:"Successfully reset users and sensors tables" });} catch(err) { console.log(err);res.sendStatus(500);} })`

*Ruta para borrar las tablas de la base de datos.*

- `app.listen(port,()=> console.log(`Server running on port:${port}`))`

*Inicia el servidor en el puerto 3000.*

## Variables

- `const express = require('express')`
- `const cors = require('cors')`
- `const pool = require('./db')`
- `const port = 3000`
- `const app = express()`

### 3.2.1. Descripción detallada

API para gestionar usuarios y sensores, incluyendo funcionalidades de creación, consulta, inserción y eliminación de datos.

Este servidor Express gestiona las tablas de usuarios y sensores, proporcionando rutas para agregar datos de sensores, obtener los datos más recientes y reiniciar las tablas en una base de datos PostgreSQL.

### 3.2.2. Documentación de funciones

#### 3.2.2.1. delete() [1/3]

```
app.delete (
 '/erase' ,
 async(req, res) ,
 { try { await pool.query('DROP TABLE IF EXISTS sensors');await pool.query('DROP
TABLE IF EXISTS users');res.status(200).send({ message:"Successfully reset users and sensors
tables" });} catch(err) { console.log(err);res.sendStatus(500);} })
```

Ruta para borrar las tablas de la base de datos.

Borra las tablas 'users' y 'sensors' si existen.

@route DELETE /erase

Devuelve

{object} 200 - Éxito al eliminar las tablas.

{object} 500 - Error interno del servidor.

#### 3.2.2.2. delete() [2/3]

```
app.delete (
 '/reset' ,
 async(req, res) ,
 { try { await pool.query('DROP TABLE IF EXISTS sensors');await pool.query('DROP
TABLE IF EXISTS users');await pool.query(` CREATE TABLE users(id SERIAL PRIMARY KEY, username
VARCHAR(100) NOT NULL);CREATE TABLE sensors(id SERIAL PRIMARY KEY, type VARCHAR(100), value
FLOAT, timestamp TIMESTAMP, user_id INTEGER REFERENCES users(id) ON DELETE CASCADE);`);await
pool.query('INSERT INTO users(username) VALUES($1),($2)', ['user1', 'user2']);await pool.↵
query(` INSERT INTO sensors(type, value, timestamp, user_id) VALUES($1, $2, $3, $4),($5, $6,
$7, $8) `, ['temperature', 25.5, '2024-09-22T12:00:00Z', 1, 'humidity', 60.0, '2024-09-22T12↵
:00:00Z', 2]);res.status(200).send({ message:"Successfully reset users and sensors tables with
default data" });} catch(err) { console.log(err);res.sendStatus(500);} })
```

Ruta para reiniciar las tablas de la base de datos.

Borra las tablas 'users' y 'sensors', y las recrea con datos predeterminados.

@route DELETE /reset

Devuelve

{object} 200 - Éxito al reiniciar las tablas.

{object} 500 - Error interno del servidor.

#### 3.2.2.3. delete() [3/3]

```
app.delete (
 '/users/:id/measurements' ,
 async(req, res) ,
 { const { id }=req.params;try { const result=await pool.query('DELETE FROM sensors
WHERE user_id=$1', [id]);res.status(200).send({ message:"Successfully deleted measurements for
user" });} catch(err) { console.log(err);res.sendStatus(500);} })
```

Ruta para eliminar todas las mediciones asociadas a un usuario.

Borra todas las entradas en la tabla 'sensors' relacionadas con un usuario específico.

@route DELETE /users/:id/measurements

### Parámetros

<code>{integer}</code>	id - ID del usuario.
------------------------	----------------------

### Devuelve

{object} 200 - Éxito al eliminar las mediciones del usuario.

{object} 500 - Error interno del servidor.

#### 3.2.2.4. `get()` [1/3]

```
app.get('/', async (req, res) => {
 try {
 const sensorsQuery = await pool.query('SELECT *FROM sensors');
 const usersQuery = await pool.query('SELECT *FROM users');
 const responseData = { sensors: sensorsQuery.rows, users: usersQuery.rows };
 res.status(200).send(responseData);
 } catch (err) {
 console.log(err);
 res.sendStatus(500);
 }
});
```

Ruta principal para obtener todos los sensores y usuarios.

Devuelve una lista de todos los sensores y usuarios registrados en la base de datos.

@route GET /

### Devuelve

{object} 200 - Datos de todos los sensores y usuarios.

{object} 500 - Error interno del servidor.

#### 3.2.2.5. `get()` [2/3]

```
app.get('/latest', async (req, res) => {
 try {
 const temperatureQuery = await pool.query(`SELECT *FROM sensors WHERE type='temperature' ORDER BY timestamp DESC LIMIT 1`);
 const co2Query = await pool.query(`SELECT *FROM sensors WHERE type='CO2' ORDER BY timestamp DESC LIMIT 1`);
 const responseData = { temperature: temperatureQuery.rows[0] || null, co2: co2Query.rows[0] || null };
 res.status(200).send(responseData);
 } catch (err) {
 console.log(err);
 res.sendStatus(500);
 }
});
```

Ruta para obtener las últimas mediciones de temperatura y CO2.

Retorna los valores más recientes de los sensores de tipo 'temperature' y 'CO2'.

@route GET /latest

### Devuelve

{object} 200 - Datos de los sensores de temperatura y CO2.

{object} 500 - Error interno del servidor.

### 3.2.2.6. get() [3/3]

```
app.get (
 '/setup' ,
 async(req, res) ,
 { try { await pool.query(` CREATE TABLE IF NOT EXISTS users(id SERIAL PRIMARY
KEY, username VARCHAR(100) NOT NULL);CREATE TABLE IF NOT EXISTS sensors(id SERIAL PRIMARY
KEY, type VARCHAR(100), value FLOAT, timestamp TIMESTAMP, user_id INTEGER REFERENCES users(id)
ON DELETE CASCADE);`);res.status(200).send({ message:"Successfully created users and sensors
tables" });} catch(err) { console.log(err);res.sendStatus(500);} })
```

Ruta para crear las tablas 'users' y 'sensors' en la base de datos si no existen.

Esta ruta se usa para inicializar las tablas necesarias en la base de datos.

@route GET /setup

Devuelve

{object} 200 - Éxito en la creación de las tablas.

{object} 500 - Error interno del servidor.

### 3.2.2.7. listen()

```
app.listen (
 port ,
 () ,
 console.log(`Server running on port:${port}`))
```

Inicia el servidor en el puerto 3000.

Muestra un mensaje en la consola cuando el servidor está en funcionamiento.

### 3.2.2.8. post() [1/2]

```
app.post (
 '/' ,
 async(req, res) ,
 { const { type, value, timestamp, userId }=req.body;try { const userExists=await
pool.query('SELECT *FROM users WHERE id=$1', [userId]);if(userExists.rows.length===0) { return
res.status(400).send({ message:"User does not exist" });} await pool.query('INSERT INTO sensors(type,
value, timestamp, user_id) VALUES($1, $2, $3, $4)', [type, value, timestamp, userId]);res.↵
status(200).send({ message:"Successfully added sensor data" });} catch(err) { console.log(err);res.↵
sendStatus(500);} })
```

Ruta para insertar datos de un sensor.

Inserta una nueva medición de sensor asociada a un usuario en la base de datos.

@route POST /



**Parámetros**

<code>{string}</code>	type - Tipo de sensor (ej: 'temperature', 'CO2').
<code>{float}</code>	value - Valor de la medición.
<code>{string}</code>	timestamp - Marca de tiempo de la medición.
<code>{integer}</code>	userId - ID del usuario asociado.

**Devuelve**

{object} 200 - Éxito al insertar la medición del sensor.

{object} 400 - El usuario no existe.

{object} 500 - Error interno del servidor.

**3.2.2.9. post() [2/2]**

```
app.post('/', async (req, res) => {
 const { username } = req.body;
 try {
 await pool.query('INSERT INTO users (username) VALUES ($1)', [username]);
 res.status(200).send({ message: 'Successfully added user' });
 } catch (err) {
 console.log(err);
 res.sendStatus(500);
 }
});
```

Ruta para insertar un nuevo usuario.

Crea un nuevo usuario en la tabla 'users'.

@route POST /users

**Parámetros**

<code>{string}</code>	username - Nombre de usuario.
-----------------------	-------------------------------

**Devuelve**

{object} 200 - Éxito al agregar el usuario.

{object} 500 - Error interno del servidor.

**3.2.2.10. use() [1/2]**

```
app.use(cors());
```

Habilita CORS para permitir solicitudes desde diferentes dominios.

**3.2.2.11. use() [2/2]**

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

Habilita el middleware para procesar JSON en las solicitudes.

### 3.2.3. Documentación de variables

#### 3.2.3.1. app

```
const app = express()
```

#### 3.2.3.2. cors

```
const cors = require('cors')
```

#### 3.2.3.3. express

```
const express = require('express')
```

#### 3.2.3.4. pool

```
const pool = require('./db')
```

#### 3.2.3.5. port

```
const port = 3000
```

### 3.3. Referencia del archivo docker-compose-example-main/README.md

### 3.4. Referencia del archivo README.md

# Índice alfabético

- app
  - server.js, [14](#)
- const
  - db.js, [7](#)
- cors
  - server.js, [14](#)
- db.js
  - const, [7](#)
  - exports, [7](#)
  - pool, [7](#)
- delete
  - server.js, [10](#)
- docker-compose-example-main/db.js, [7](#)
- docker-compose-example-main/README.md, [14](#)
- docker-compose-example-main/server.js, [8](#)
- exports
  - db.js, [7](#)
- express
  - server.js, [14](#)
- get
  - server.js, [11](#)
- listen
  - server.js, [12](#)
- pool
  - db.js, [7](#)
  - server.js, [14](#)
- port
  - server.js, [14](#)
- post
  - server.js, [12](#), [13](#)
- README.md, [14](#)
- Sensor Management API, [1](#)
- server.js
  - app, [14](#)
  - cors, [14](#)
  - delete, [10](#)
  - express, [14](#)
  - get, [11](#)
  - listen, [12](#)
  - pool, [14](#)
  - port, [14](#)
  - post, [12](#), [13](#)
  - use, [13](#)
- use
  - server.js, [13](#)