Proyecto3A_Server_Backend

Generado por Doxygen 1.12.0

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

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

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

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
   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

Lista de pruebas

Miembro describe ('API Routes',()=> { beforeEach(()=> { pool.query.mockClear();});it('should set up tables successfully', async()=> { pool.query.mockResolvedValue({});const res=await request(app).get('/setup');expect(res.statusCode).toEqual(200);expect(res.body).toHave↩ Property('message', "Successfully created users and sensors tables");});it('should create a new user', async()=> { pool.query.mockResolvedValue({});const res=await request(app) .post('/users') .send({ username:'testuser' });expect(res.statusCode).toEqual(200);expect(res.body).toHave↩ Property('message', "Successfully added user");});it('should not insert sensor data if user does not exist', async()=> { pool.query.mockResolvedValue({ rows:[] });const res=await request(app) .post('/') .send({ type:'temperature', value:25, timestamp:new Date().toISOString(), userId:999 });expect(res.statusCode).toEqual(400);expect(res.body).toHaveProperty('message', "User does not exist");});it('should reset the tables successfully', async()=> { pool.query.mockResolvedValue({});const res=await request(app).delete('/reset');expect(res.statusCode).toEqual(200);expect(res.body).toHave↩ Property('message', "Successfully reset users and sensors tables with default data");});})

Prueba para la ruta GET /setup.

Prueba para la ruta POST /users.

Prueba para la ruta POST /.

Prueba para la ruta DELETE /reset.

Capítulo 3

Índice de archivos

3.1. Lista de archivos

Lista de todos los archivos con breves descripciones:

Capítulo 4

Documentación de archivos

4.1. 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.

4.1.1. 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.

4.1.2. Documentación de variables

4.1.2.1. const

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

4.1.2.2. exports

```
module exports = pool
```

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

4.1.2.3. 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

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

4.2. 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()
- module exports = app

4.2.1. 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.

4.2.2. Documentación de funciones

4.2.2.1. 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.

4.2.2.2. 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.

4.2.2.3. 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

{integer}	id - ID del usuario.

Devuelve

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

{object} 500 - Error interno del servidor.

4.2.2.4. get() [1/3]

```
app get (
            '/' ,
            async(req, res) ,
            { try { const sensorsQuery=await pool.query('SELECT *FROM sensors');const users↩
Query=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.

4.2.2.5. 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(response↩
Data);} 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.

4.2.2.6. 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.

4.2.2.7. 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.

4.2.2.8. 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

{string}	type - Tipo de sensor (ej: 'temperature', 'CO2').
{float}	value - Valor de la medición.
{string}	timestamp - Marca de tiempo de la medición.
{integer}	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.

4.2.2.9. post() [2/2]

```
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.

Crea un nuevo usuario en la tabla 'users'.

@route POST /users

Parámetros

{string}	username - Nombre de usuario.

Devuelve

{object} 200 - Éxito al agregar el usuario.

{object} 500 - Error interno del servidor.

4.2.2.10. use() [1/2]

```
app use (
            cors() )
```

Habilita CORS para permitir solicitudes desde diferentes dominios.

4.2.2.11. use() [2/2]

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

Habilita el middleware para procesar JSON en las solicitudes.

4.2.3. Documentación de variables

4.2.3.1. app

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

4.2.3.2. cors

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

4.2.3.3. exports

```
module exports = app
```

4.2.3.4. express

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

4.2.3.5. pool

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

4.2.3.6. port

```
const port = 3000
```

4.3. Referencia del archivo docker-compose-example-main/server.test.js

Pruebas automáticas para las rutas de la API del servidor utilizando Supertest y Jest.

Funciones

- jest mock ('./db')

  Se crea un mock del módulo de la base de datos para evitar conexiones reales.
- describe ('API Routes',()=> { beforeEach(()=> { pool.query.mockClear();});it('should set up tables successfully', async()=> { pool.query.mockResolvedValue({});const res=await request(app).get('/setup');expect(res.↩ statusCode).toEqual(200);expect(res.body).toHaveProperty('message', "Successfully created users and sensors tables");});it('should create a new user', async()=> { pool.query.mockResolvedValue({});const res=await request(app) .post('/users') .send({ username:'testuser' });expect(res.statusCode).toEqual(200);expect(res.↩ body).toHaveProperty('message', "Successfully added user");});it('should not insert sensor data if user does not exist', async()=> { pool.query.mockResolvedValue({ rows:[] });const res=await request(app) .post('/') .send({ type:'temperature', value:25, timestamp:new Date().toISOString(), userId:999 });expect(res.status↩ Code).toEqual(400);expect(res.body).toHaveProperty('message', "User does not exist");});it('should reset the tables successfully', async()=> { pool.query.mockResolvedValue({});const res=await request(app).delete('/reset');expect(res.↩ statusCode).toEqual(200);expect(res.body).toHaveProperty('message', "Successfully reset users and sensors tables with default data");});})

  Conjunto de pruebas para las rutas de la API.

Variables

- const request = require('supertest')
- const app = require('./server')
- const pool = require('./db')

  Se simula el pool de la base de datos con un mock.

4.3.1. Descripción detallada

Pruebas automáticas para las rutas de la API del servidor utilizando Supertest y Jest.

Este archivo contiene una serie de pruebas unitarias que simulan las interacciones con las rutas de la API del servidor sin necesidad de conectarse a una base de datos real, utilizando mocks de Jest.

4.3.2. Documentación de funciones

4.3.2.1. describe()

```
describe (
            'API Routes' ,
            () ,
            { beforeEach(()=> { pool.query.mockClear();});it('should set up tables successfully',
async()=> { pool.query.mockResolvedValue({});const res=await request(app).get('/setup');expect(res.↩
statusCode).toEqual(200);expect(res.body).toHaveProperty('message', "Successfully created
users and sensors tables");});it('should create a new user', async()=> { pool.query.mock↩
ResolvedValue({});const res=await request(app) .post('/users') .send({ username:'testuser'
});expect(res.statusCode).toEqual(200);expect(res.body).toHaveProperty('message', "Successfully
added user");});it('should not insert sensor data if user does not exist', async()=> { pool.↩
query.mockResolvedValue({ rows:[] });const res=await request(app) .post('/') .send({ type↩
:'temperature', value:25, timestamp:new Date().toISOString(), userId:999 });expect(res.↩
statusCode).toEqual(400);expect(res.body).toHaveProperty('message', "User does not exist");});it('should
reset the tables successfully', async()=> { pool.query.mockResolvedValue({});const res=await
request(app).delete('/reset');expect(res.statusCode).toEqual(200);expect(res.body).toHave↩
Property('message', "Successfully reset users and sensors tables with default data");});} )
```

Conjunto de pruebas para las rutas de la API.

Se describen diversas pruebas que comprueban el comportamiento de las rutas del servidor.

4.3.2.2. mock()

```
jest mock (
            './db' )
```

Se crea un mock del módulo de la base de datos para evitar conexiones reales.

4.3.3. Documentación de variables

4.3.3.1. app

```
const app = require('./server')
```

4.3.3.2. pool

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

Se simula el pool de la base de datos con un mock.

4.3.3.3. request

```
const request = require('supertest')
```

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

4.5. Referencia del archivo README.md

Índice alfabético