UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”

INSTITUTO DE CIÊNCIA E TECNOLOGIA DE SOROCABA (ICTS)

RELATÓRIO PROJETO INTEGRADOR

Atividade 2 - Comunicação básica e protótipo de backend Descrição Geral

Amanda Segura Mendes de Oliveira

Ana Clara Godoy Ensides

Beatriz Martuscelli da Silva Prado

Felipe Pellegrini Kumagae

Guilherme Yuiti de Queiroz Barbosa

Hebert de Oliveira Brito

Maysa Gabriela Lucas Izaias

Rafael Utsunomya Machado

SOROCABA-SP

2025

SUMÁRIO

1. Introdução

Nesta segunda etapa do projeto, buscou-se estabelecer a comunicação entre o microcontrolador ESP32 e o servidor web por meio de requisições HTTP baseadas no protocolo REST, conforme a arquitetura proposta para a equipe na Atividade 1. O principal objetivo foi implementar o esqueleto funcional do backend, capaz de receber e registrar dados enviados pelo ESP32 — como leituras de sensores de nível — e retornar respostas de confirmação ao dispositivo. Esse processo é fundamental para garantir a integração entre o sistema embarcado e o servidor, formando o elo de comunicação que permitirá o controle remoto e o monitoramento do sistema de tanques. Assim, esta etapa consolida as bases de comunicação e armazenamento de dados sobre as quais serão desenvolvidas as fases seguintes, incluindo autenticação, criptografia e dashboard de supervisão.

Como ainda não foi possível realizar a conexão direta com o módulo ESP32 e o sistema físico de tanques (DTS200), a equipe optou por desenvolver e testar duas abordagens alternativas de servidor: uma utilizando a linguagem C# e outra por meio da plataforma Node-RED. Essa estratégia experimental garantiu o avanço do projeto mesmo na ausência do hardware, além de proporcionar um ambiente controlado para o teste das estruturas de API e do fluxo de dados, assegurando que o backend esteja apto a receber e tratar as informações provenientes do ESP32 na próxima fase do desenvolvimento.

2. Implantar um servidor funcional capaz de receber requisições externas

2.1 C#

https://github.com/RafaelUtsunomya/ProjetoIntegrador

Para estabelecer o fluxo de comunicação entre o sistema embarcado e o servidor, foram implementados dois *endpoints* centrais na API. A rota principal, **[HttpPost("RegistrarDados")]**, serve como o ponto de entrada de dados, projetada para receber requisições POST enviadas pelo microcontrolador ESP. Quando o ESP envia uma nova leitura (formatada como um JSON correspondente ao DTO RegistrarDadosESPRequest), este *endpoint* a captura, encaminha para a camada de serviço para processamento e a persiste no banco de dados. Essa rota foi estruturada para retornar uma resposta 200 OK (incluindo os dados salvos com seu novo Id) em

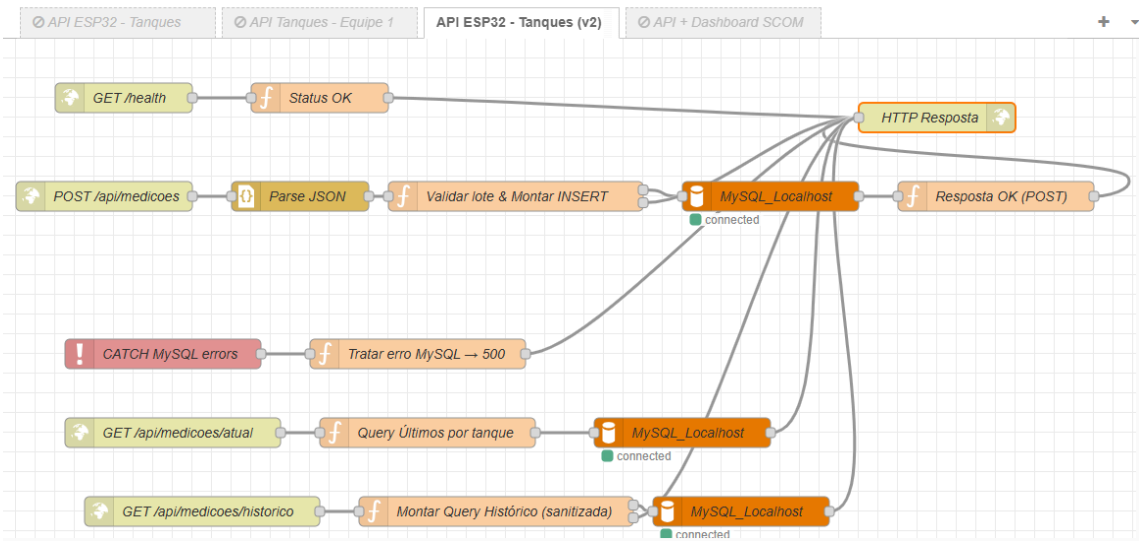
caso de sucesso, ou 400 Bad Request se ocorrerem erros de validação, garantindo um *feedback* claro ao dispositivo.

Complementando a entrada de dados, a rota **[HttpGet("ObterDados")]** atua como o ponto de saída ou consulta do sistema. Utilizando o método GET, este *endpoint* permite que aplicações consumidoras—como *dashboards* de visualização ou ferramentas de teste—solicitem os dados que foram previamente armazenados. Ao ser chamada, a API consulta o banco de dados através da camada de serviço e retorna uma lista formatada em JSON com os registros das leituras. Em conjunto, essas duas rotas (POST para registrar e GET para consultar) criam a base funcional da API, consolidando o ciclo completo de persistência e recuperação de dados do projeto.

2.2 NodeRed

No Node-RED, foi implementado um servidor web completo para simular a API responsável pela comunicação entre o ESP32 e o banco de dados MySQL. O fluxo desenvolvido possui quatro rotas principais: **GET /health**, **POST /api/medicoes**, **GET /api/medicoes/atua**l e **GET /api/medicoes/historico.** A Figura 1 exibe os blocos utilizados no Node-RED.

Figura 1 - Node-Red



Fonte: Elaboração própria.

A primeira rota, destinada ao “health check”, foi criada para verificar a disponibilidade do servidor, retornando um objeto JSON com o status “ok” e o horário atual do sistema, como mostrado na Figura 2. Essa etapa permitiu confirmar o correto funcionamento do servidor e serviu como base para os testes das demais rotas.

A segunda rota (**POST /api/medicoes**) foi o núcleo do backend, responsável por receber os dados simulados das medições em formato JSON, validar a estrutura recebida e inserir os registros no banco de dados. Para isso, foi utilizado um nó function que verifica se o payload contém um array de objetos com os campos tanque, nível e timestamp, construindo automaticamente a instrução SQL de inserção em lote. Os dados validados são enviados ao nó MySQL_Localhost, configurado para se conectar ao banco “scom” no servidor local. Ao final da operação, uma mensagem de resposta confirma o número de registros inseridos com sucesso. Também foi adicionado um nó catch para capturar e tratar possíveis erros de comunicação com o banco, retornando uma resposta HTTP com código 500 e descrição do erro.

As duas últimas rotas — **GET /api/medicoes/atual** e **GET /api/medicoes/historico** — foram implementadas para permitir a consulta dos dados armazenados. A primeira retorna o valor mais recente de nível registrado para cada tanque, utilizando uma consulta SQL com JOIN e MAX(timestamp). Já a segunda rota permite filtrar o histórico por tanque e intervalo de tempo, aplicando sanitização aos parâmetros recebidos para evitar injeção de código SQL. Com isso, o fluxo do Node-RED consolidou um backend funcional e seguro, pronto para receber os dados provenientes do ESP32 e disponibilizá-los ao frontend do sistema supervisório.

3. Definir e testar rotas REST para inserção e consulta de dados

3.1 C#

Para a etapa de definição das rotas, a arquitetura da API foi projetada seguindo os padrões RESTful. Foram estabelecidos dois *endpoints* principais no *controller*: uma rota de inserção (**[HttpPost("RegistrarDados")]**) e uma rota de consulta (**[HttpGet("ObterDados")]**). A rota de inserção utiliza o método POST, pois é semanticamente usada para "criar" um novo recurso (um novo registro de dados) no servidor, e foi configurada para esperar os dados do ESP no formato JSON, definidos pelo DTO RegistrarDadosESPRequest. A rota de consulta utiliza o método GET, apropriado para a "leitura" de dados, e foi programada para retornar uma lista de todos os registros armazenados, também em formato JSON.

O teste dessas rotas foi realizado para simular a comunicação do ESP e validar o fluxo de dados. Utilizando uma ferramenta de cliente de API, como o Postman, disparamos requisições POST manuais para o *endpoint* RegistrarDados, enviando um JSON com dados de exemplo. O sucesso foi validado pela recepção de um status HTTP 200 OK do servidor e pela confirmação visual, no banco de dados, de que o novo registro foi corretamente persistido. Em seguida, disparamos requisições GET para o *endpoint* ObterDados para verificar se a lista de dados, incluindo o registro recém-criado, era retornada com sucesso, confirmando o ciclo completo de escrita e leitura.

CSharp

```
[ApiController]
[Route("api/[controller]")]
public class ESPController : ControllerBase
{
    private readonly IDadosESPService _DadosEspService;

    public ESPController(IDadosESPService DadosEspService)
    {
        _DadosEspService = DadosEspService;
    }

    [HttpPost("RegistrarDados")]
    public async Task<IActionResult> RegistrarLeitura([FromBody]
RegistrarDadosESPRequest request)
    {
        try
        {
            var resposta = await
_DadosEspService.RegistrarLeituraAsync(request);

            // Retorna 200 OK com os dados registrados (incluindo Id e
Timestamp)
            return Ok(resposta);
        }
        catch (Exception ex)
        {
            // Retorna 400 Bad Request se algo der errado (ex: dados
inválidos)
            return BadRequest(new { error = ex.Message });
        }
    }

    [HttpGet("ObterDados")]
```

```
    public async Task<IActionResult> ObterLeituras()
    {
        var leituras = await _DadosEspService.ObterLeiturasAsync();
        return Ok(leituras);
    }
}
```

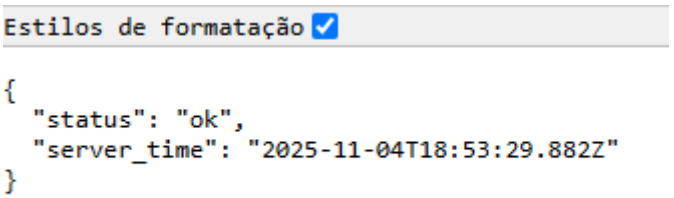
3.2 NodeRed

As rotas REST definidas e testadas no Node-RED permitem estabelecer um ciclo completo de comunicação entre o servidor e os dispositivos clientes. A rota **GET /health** (Figura 2) foi criada para verificar o funcionamento do servidor, retornando um status de disponibilidade e o horário atual, servindo como teste inicial de conectividade. A rota **POST /api/medicoes** é responsável por receber os dados de medições enviados em formato JSON — contendo tanque, nível e timestamp — e registrá-los no banco de dados MySQL. Já as rotas **GET /api/medicoes/atual** (Figura 3) e **GET /api/medicoes/historico** (Figura 4) permitem consultar os dados armazenados: a primeira retorna o valor mais recente de nível de cada tanque, enquanto a segunda possibilita visualizar o histórico de medições filtrando por tanque e intervalo de tempo. Em conjunto, essas rotas consolidam a estrutura fundamental da API REST, garantindo a inserção, verificação e recuperação de dados de forma padronizada e eficiente.

```
None
### HEALTH CHECK
GET http://localhost:1880/health
Accept: application/json


### POST /api/medicoes (envio de dados)
POST http://localhost:1880/api/medicoes
Content-Type: application/json

[
  {"tanque": 1, "nivel": 25.4, "timestamp": "2025-11-04 02:10:00"},
  {"tanque": 2, "nivel": 33.7, "timestamp": "2025-11-04 02:10:00"},
  {"tanque": 3, "nivel": 41.2, "timestamp": "2025-11-04 02:10:00"}
```

```
]


### GET /api/medicoes/atual
GET http://localhost:1880/api/medicoes/atual
Accept: application/json


### GET /api/medicoes/historico (tanque 3, últimas 24h)
GET
http://localhost:1880/api/medicoes/historico?tanque=3&de=2025-11-03%2000
:00:00&ate=2025-11-05%2000:00:00
Accept: application/json
```

Figura 2 - /health

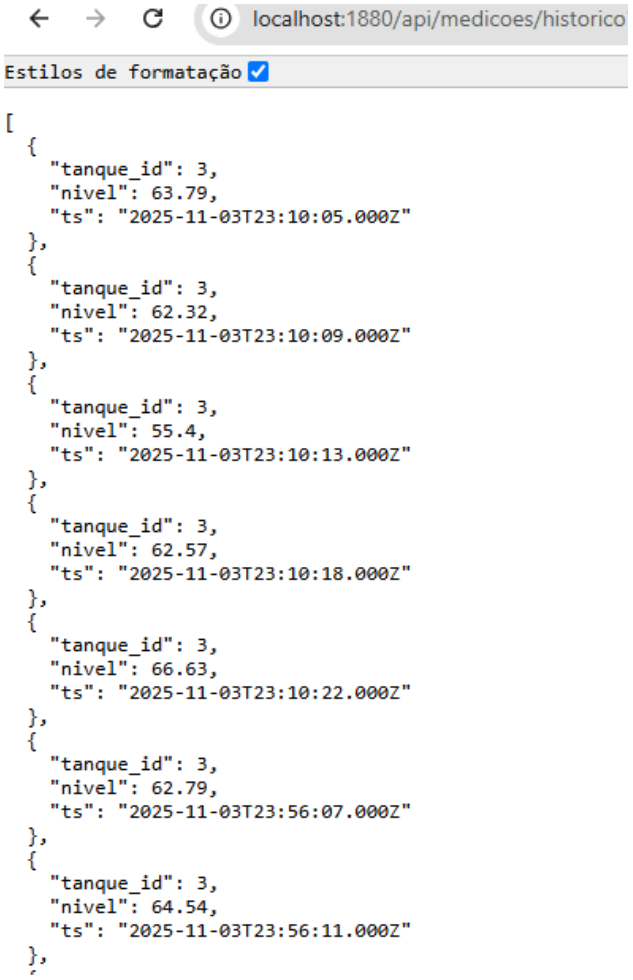


Fonte: Elaboração própria.

Figura 3 - /api/medicoes/atual

Fonte: Elaboração própria.

Figura 4 - /api/medicoes/historico



Fonte: Elaboração própria.

4. Registrar os dados recebidos em um banco de dados relacional

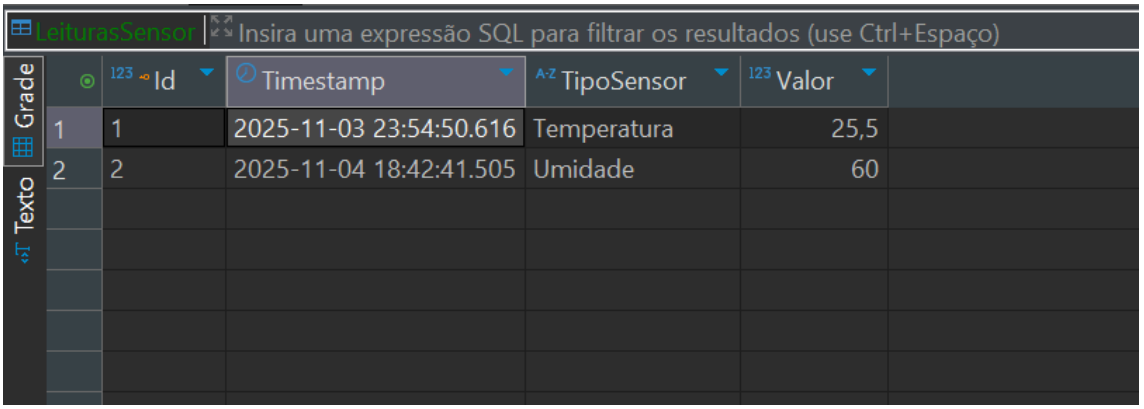
4.1 C#

Para implementar o registro de dados em um banco relacional, foi utilizado o SQL Server em conjunto com o *Object-Relational Mapper* (ORM) Entity Framework (EF) Core. Essa abordagem permitiu abstrair a complexidade da linguagem SQL, tratando os dados como objetos C#. Concretamente, quando a API recebe os dados, a camada de serviço instancia a entidade de domínio (LeituraSensor) e a repassa para a camada de repositório. O repositório, por sua vez, utiliza o AppDbContext (a representação do banco em código) para adicionar esta nova entidade ao DbSet correspondente. Finalmente, a chamada ao método SaveChangesAsync() instrui o EF Core a analisar o objeto C# e gerar automaticamente o comando INSERT apropriado, executando-o contra o SQL Server para persistir a nova leitura de forma estruturada e confiável na tabela LeiturasSensor. Na Figura 5 pode-se ver o SQL Server.

CSharp

```
public class DadosESPService : IDadosESPService
{
    private readonly IDadosESPRepository _repository;

    public DadosESPService(IDadosESPRepository repository)
    {
        _repository = repository;
    }

    public async Task<IEnumerable<DadosESPResponse>>
ObterLeiturasAsync()
    {
        var leituras = await _repository.ObterTodasAsync();

        // Usando .Select() para transformar cada item da lista
        return leituras.Select(leitura => new DadosESPResponse
        {
            Id = leitura.Id,
            Timestamp = leitura.Timestamp,
            TipoSensor = leitura.TipoSensor,
            Valor = leitura.Valor
        });
    }

    public async Task<DadosESPResponse>
RegistrarLeituraAsync(RegistrarDadosESPRequest request)
    {
        // 1. Converte o DTO de requisição para a entidade de domínio
        var novaLeitura = new DadosESP(request.TipoSensor,
request.Valor);

        // 2. Salva no banco através do repositório
        var leituraSalva = await
_repository.AdicionarAsync(novaLeitura);


        return new DadosESPResponse
        {
            Id = leituraSalva.Id,
            Timestamp = leituraSalva.Timestamp,
            TipoSensor = leituraSalva.TipoSensor,
            Valor = leituraSalva.Valor
        };
    }
}
```

Figura 5 - SQL Server.



Fonte: Elaboração própria.

4.2 NodeRed

Para o armazenamento e gerenciamento dos dados coletados, foi utilizado um banco de dados MySQL hospedado localmente (localhost), configurado especificamente para o projeto. No script SQL, foram criadas três tabelas principais e uma *view* auxiliar. A tabela **tanques** registra as informações estáticas de identificação de cada tanque do sistema, como nome e descrição. A tabela **medicoes** armazena os dados enviados pelo ESP32, contendo o identificador do tanque, o nível medido e o respectivo timestamp, permitindo o registro histórico das leituras. Já a tabela **eventos** foi destinada ao armazenamento de alertas e logs do sistema, como notificações de níveis críticos, falhas e inserção de novos comandos de controle. Por fim, a *view* **vw_ultimas_medicoes** foi criada para facilitar consultas às últimas medições de cada tanque, otimizando a integração entre o backend e o dashboard de supervisão.

SQL utilizado:

```sql
SQL
-- ============================================================
-- BANCO DE DADOS PARA PROJETO INTEGRADOR - Etapa 02
-- Comunicação ESP ↔ Servidor Node-RED via REST API
-- ============================================================

-- Criação do banco
CREATE DATABASE IF NOT EXISTS scom
DEFAULT CHARACTER SET utf8mb4
DEFAULT COLLATE utf8mb4_general_ci;

USE scom;

-- ============================================================
```

```sql
-- 1. Tabela de Tanques
-- ============================================================
CREATE TABLE IF NOT EXISTS tanques (
  id INT PRIMARY KEY,
  nome VARCHAR(50) NOT NULL,
  descricao VARCHAR(255) DEFAULT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Inserção de tanques iniciais (ajuste conforme o projeto)
INSERT IGNORE INTO tanques (id, nome, descricao) VALUES
  (1, 'Tanque 1', 'Tanque de controle principal'),
  (2, 'Tanque 2', 'Tanque intermediário'),
  (3, 'Tanque 3', 'Tanque auxiliar');

-- ============================================================
-- 2. Tabela de Medições (dados recebidos do ESP32)
-- ============================================================
CREATE TABLE IF NOT EXISTS medicoes (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  tanque_id INT NOT NULL,
  nivel DOUBLE NOT NULL,
  ts DATETIME NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX idx_tanque_ts (tanque_id, ts),
  FOREIGN KEY (tanque_id) REFERENCES tanques(id)
    ON DELETE CASCADE ON UPDATE CASCADE
);

-- ============================================================
-- 3. Tabela de Eventos (alertas, logs, etc.)
-- ============================================================
CREATE TABLE IF NOT EXISTS eventos (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  tipo VARCHAR(30) NOT NULL,          -- exemplo: 'ALTA', 'BAIXA',
'ERRO'
  descricao VARCHAR(255) NOT NULL,
  ts DATETIME NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- ============================================================
-- 4. View auxiliar: últimas medições por tanque
-- ============================================================
CREATE OR REPLACE VIEW vw_ultimas_medicoes AS
SELECT
  m.tanque_id,
  t.nome AS tanque_nome,
```

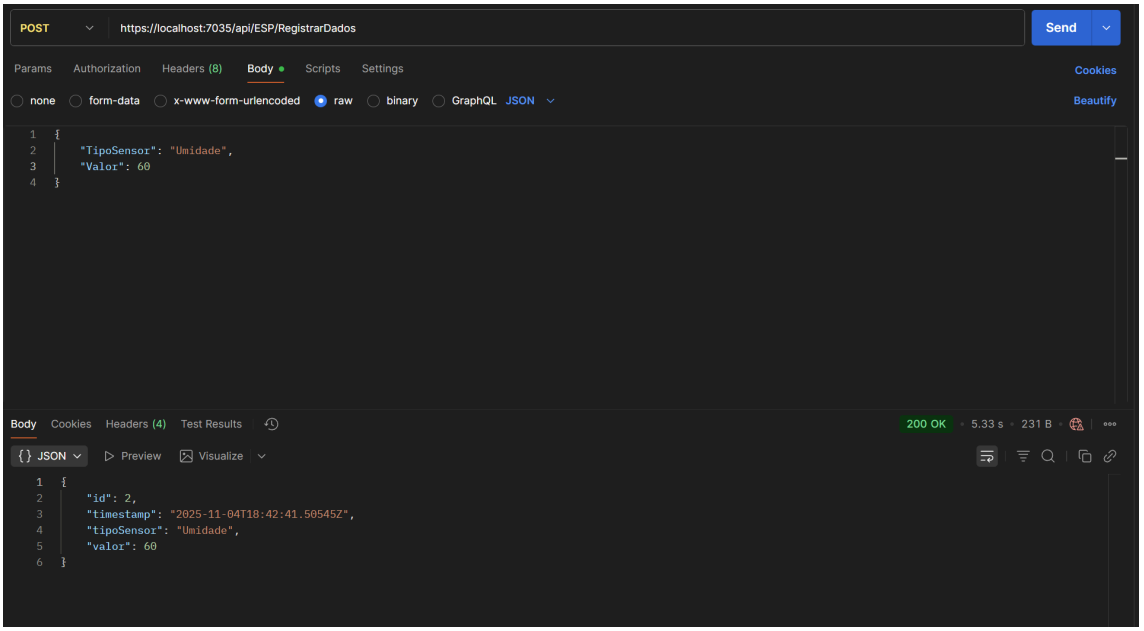
```
  m.nivel,
  m.ts
FROM medicoes m
JOIN tanques t ON t.id = m.tanque_id
WHERE m.ts = (
  SELECT MAX(m2.ts)
  FROM medicoes m2
  WHERE m2.tanque_id = m.tanque_id
);
```

5. **Simular a comunicação do ESP com o servidor utilizando ferramentas de teste**

   5.1 C#

Para simular a comunicação que o microcontrolador ESP fará com o servidor, foi utilizada a ferramenta Postman. Nela, criamos uma requisição HTTP do tipo POST e a direcionamos para a URL completa do *endpoint* de registro (ex: https://localhost:7123/api/SensorData/RegistrarDados). No corpo ("Body") desta requisição, foi selecionado o formato raw com o tipo JSON e inserido manualmente um pacote de dados de exemplo que mimetiza o formato do DTO RegistrarDadosESPRequest, como {"TipoSensor": "Temperatura", "Valor": 25.5}. Ao enviar essa requisição, o Postman nos permitiu validar instantaneamente a resposta do servidor: o recebimento de um *status code* 200 OK, juntamente com os dados persistidos (incluindo o Id e o Timestamp), confirmou que a API estava funcionando corretamente e que a comunicação, como simulada pelo ESP, foi bem-sucedida.

Figura 6 - Teste feito com Postman.



Fonte: Elaboração própria.

5.2 NodeRed

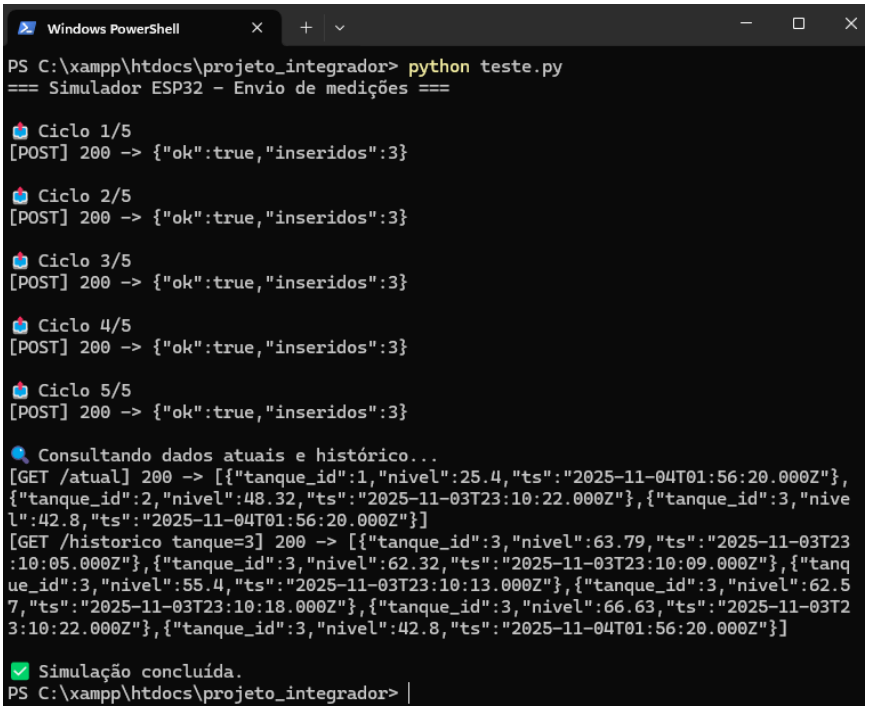
Para simular a comunicação entre o ESP32 e o servidor implementado no Node-RED, foi desenvolvido um script em Python utilizando as bibliotecas requests, json e random. O programa gera medições simuladas dos três tanques do sistema, criando valores aleatórios de nível e timestamps atuais, que são enviados ao endpoint **POST /api/medicoes** no formato JSON. Após o envio no prompt de comando (Figura 7), o simulador realiza requisições GET para os endpoints **/api/medicoes/atual** e **/api/medicoes/historico**, permitindo verificar se os dados foram corretamente armazenados e podem ser consultados no banco de dados (Figura 8). Esse processo reproduz o comportamento esperado do microcontrolador ESP32, validando o funcionamento completo da API REST e a integração entre o backend e o sistema de armazenamento.

```python
"""
Simulador de envio de medições (ESP32 → Node-RED)
-------------------------------------------------
Envia dados JSON simulados para o endpoint:
    POST http://localhost:1880/api/medicoes
E consulta:
```

```
    GET  http://localhost:1880/api/medicoes/atual
    GET  http://localhost:1880/api/medicoes/historico
"""
import requests
import json
import random
import time
from datetime import datetime
BASE_URL = "http://localhost:1880"
# Configuração dos tanques simulados
TANQUES = [1, 2, 3]
def gerar_medicoes():
    """Gera um array JSON simulando leitura dos tanques."""
    agora = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    medicoes = []
    for t in TANQUES:
        nivel = round(random.uniform(20 + 10t, 40 + 10t), 2)  # níveis
distintos
        medicoes.append({
            "tanque": t,
            "nivel": nivel,
            "timestamp": agora
        })
    return medicoes

def enviar_medicoes():
    """Faz POST com as medições simuladas."""
    url = f"{BASE_URL}/api/medicoes"
    dados = gerar_medicoes()
    try:
        r = requests.post(url, json=dados, timeout=15)
        print(f"[POST] {r.status_code} -> {r.text}")
    except Exception as e:
        print(f"[ERRO POST] {e}")

def consultar_atuais():
    """Consulta os últimos valores registrados."""
    url = f"{BASE_URL}/api/medicoes/atual"
    try:
        r = requests.get(url, timeout=10)
        print(f"[GET /atual] {r.status_code} -> {r.text}")
    except Exception as e:
        print(f"[ERRO GET atual] {e}")

def consultar_historico(tanque=1):
    """Consulta histórico das últimas 24h para um tanque."""
    agora = datetime.now()
```

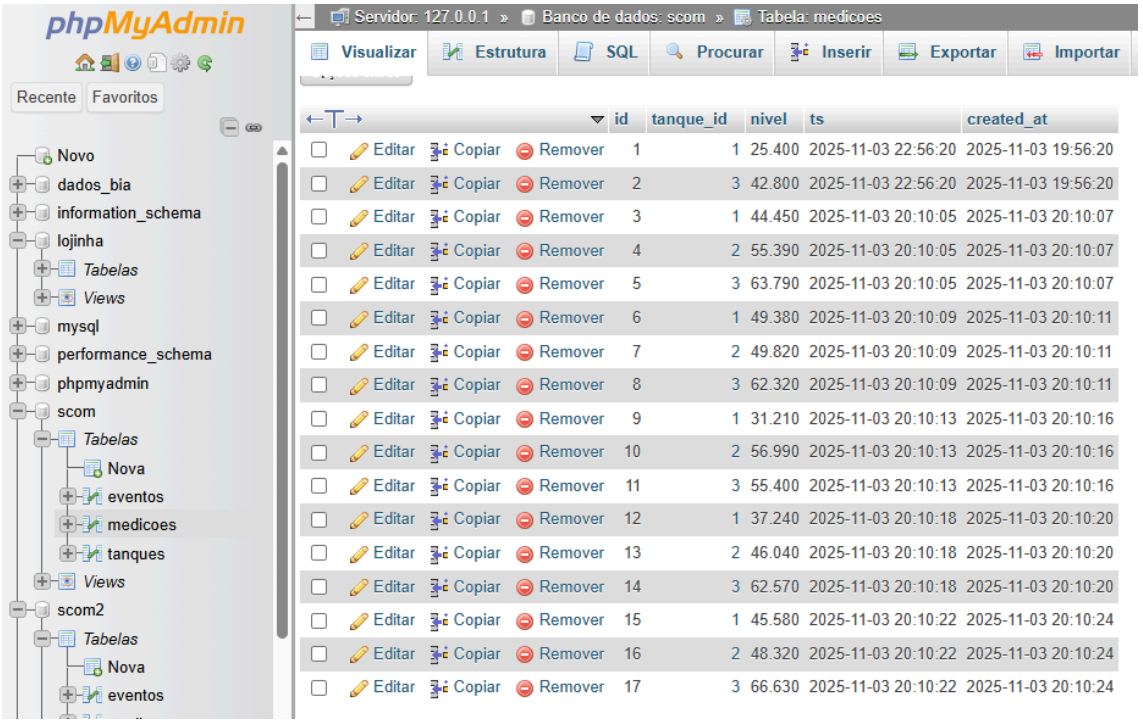
```
    de = (agora.replace(hour=0, minute=0, second=0)).strftime("%Y-%m-%d
%H:%M:%S")
    ate = (agora.replace(hour=23, minute=59,
second=59)).strftime("%Y-%m-%d %H:%M:%S")
    url =
f"{BASE_URL}/api/medicoes/historico?tanque={tanque}&de={de}&ate={ate}"
    try:
        r = requests.get(url, timeout=10)
        print(f"[GET /historico tanque={tanque}] {r.status_code} ->
{r.text}")
    except Exception as e:
        print(f"[ERRO GET historico] {e}")

if __name__ == "__main__":
    print("=== Simulador ESP32 - Envio de medições ===")
    ciclos = 5           # número de ciclos de envio
    intervalo = 2        # segundos entre envios
    for i in range(ciclos):
        print(f"\n📤 Ciclo {i+1}/{ciclos}")
        enviar_medicoes()
        time.sleep(intervalo)
    print("\n🔍 Consultando dados atuais e histórico...")
    consultar_atuais()
    consultar_historico(3)
    print("\n✅ Simulação concluída.")
```

Figura 7 - Prompt de comando rodando o código python.



Fonte: Elaboração própria.

Figura 8 - Banco de dados **scom** na pasta **medicoes**.



Fonte: Elaboração própria.

6. Documentar a API e o formato dos pacotes trocados (JSON)

6.1 C#

A documentação da API foi gerada dinamicamente utilizando a ferramenta Swagger, que implementa a especificação OpenAPI. Esta abordagem foi integrada diretamente ao projeto ASP.NET Core (AddSwaggerGen()), permitindo que a documentação seja criada automaticamente a partir do código-fonte, incluindo os *controllers*, as rotas ([HttpPost], [HttpGet]) e os modelos de dados (DTOs). Ao executar a aplicação, o Swagger UI fornece uma página web interativa que lista todos os *endpoints* disponíveis, detalhando o método HTTP necessário, os parâmetros, o formato exato do corpo da requisição e os possíveis códigos de resposta (como 200 OK ou 400 Bad Request). Esta documentação interativa foi crucial, pois serviu não apenas como um guia de referência, mas também como uma plataforma inicial de testes para validar o comportamento de cada rota.

O formato de troca de pacotes de dados (payload) entre o ESP e o servidor é o JSON (JavaScript Object Notation), escolhido por sua simplicidade, legibilidade e por ser um padrão leve, ideal para dispositivos com recursos limitados. A estrutura desses pacotes foi definida de forma estrita no *backend* através de classes DTO (Data Transfer Objects). Para o envio de dados, o pacote de requisição (RegistrarDadosESPRequest) exige um JSON

contendo as chaves TipoSensor (string) e Valor (double). Em resposta, o servidor retorna um JSON (baseado no LeituraSensorResponse) que confirma os dados salvos, incluindo o Id gerado pelo banco e o Timestamp do servidor, garantindo um ciclo de comunicação claro e estruturado.

```JSON
{
	"TipoSensor":,
	"Umiidade":
}
```

```JSON
{
	"Id":,
	"Timestamp":,
	"TipoSensor":,
	"Valor":
}
```

6.2 NodeRed

A API implementada no Node-RED segue o padrão **RESTful** e utiliza o formato **JSON** para troca de informações entre o cliente (ESP32) e o servidor. O principal pacote de envio é realizado pela rota **POST /api/medicoes**, onde o dispositivo envia um array JSON contendo objetos com os campos **tanque**, **nível** e **timestamp** — representando as medições coletadas em tempo real. Cada pacote enviado contém múltiplos registros e é processado pelo servidor, que valida os dados e insere as informações no banco de dados relacional MySQL.

Como resposta, o servidor devolve pacotes JSON padronizados contendo mensagens de confirmação ou erro. Em caso de sucesso, a resposta inclui o campo **ok: true** e o número de registros inseridos (**inseridos: n**). As rotas de consulta, **GET /api/medicoes/atual** e **GET /api/medicoes/historico**, retornam listas de objetos JSON com os níveis registrados e seus respectivos timestamps, permitindo que o frontend ou o simulador visualizem os dados armazenados. Dessa forma, a API garante comunicação bidirecional confiável e estruturada entre o sistema embarcado e o servidor web.