Nome do Projeto: Otimização no Agronegócio: IoT na Avicultura

Nome do Aluno: Rodolfo Felipe Pereira dos Santos

Data: 05/11/2025

Disciplina: Projeto de IoT para o Big Data

Equipe: Big Data no Agronegócio – 5º Sem

Professor Orientador: Antônio Fernando Traina

DESENVOLVIMENTO TEÓRICO CODIFICAÇÃO PROJETO

1. Arquitetura básica do projeto

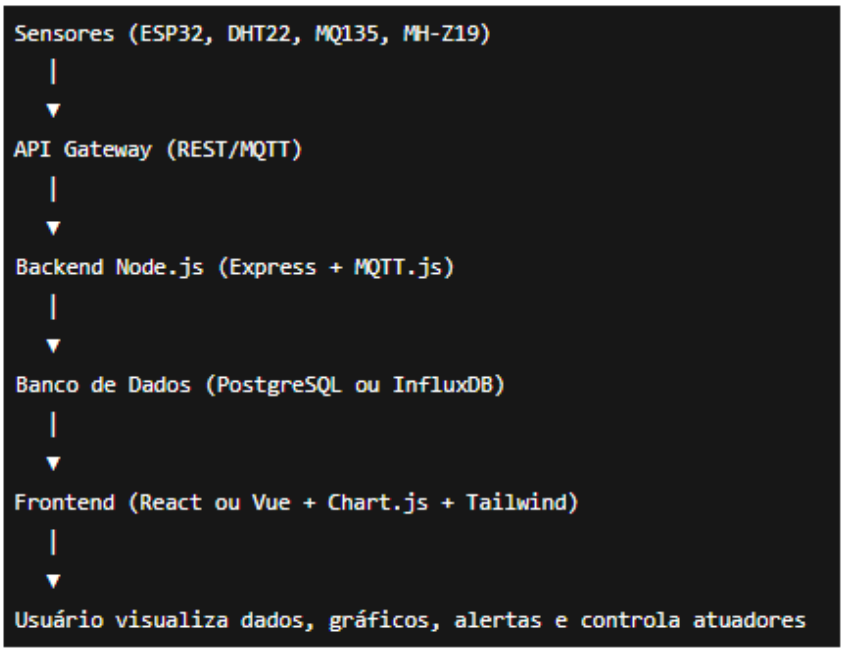


Imagem 1 - Arquitetura básica projeto (Fonte: Autor)

2. Estrutura esperada de pastas

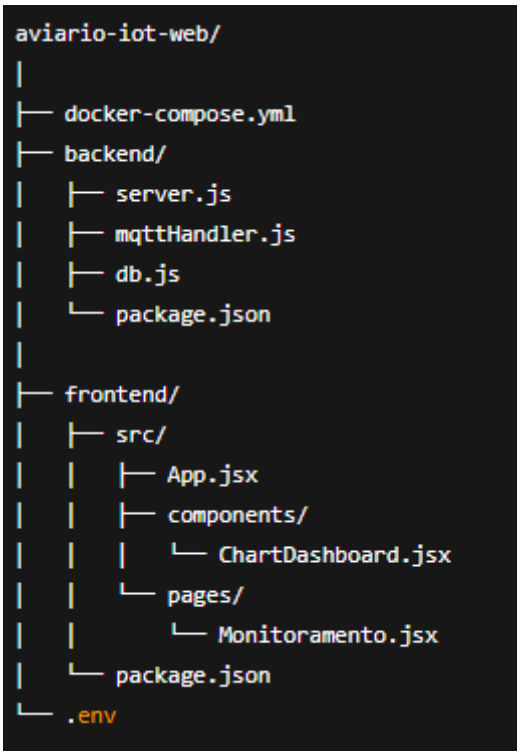


Imagem 2 - Estrutura esperada de pastas

3. Código inicial do Projeto

3.1. Frontend

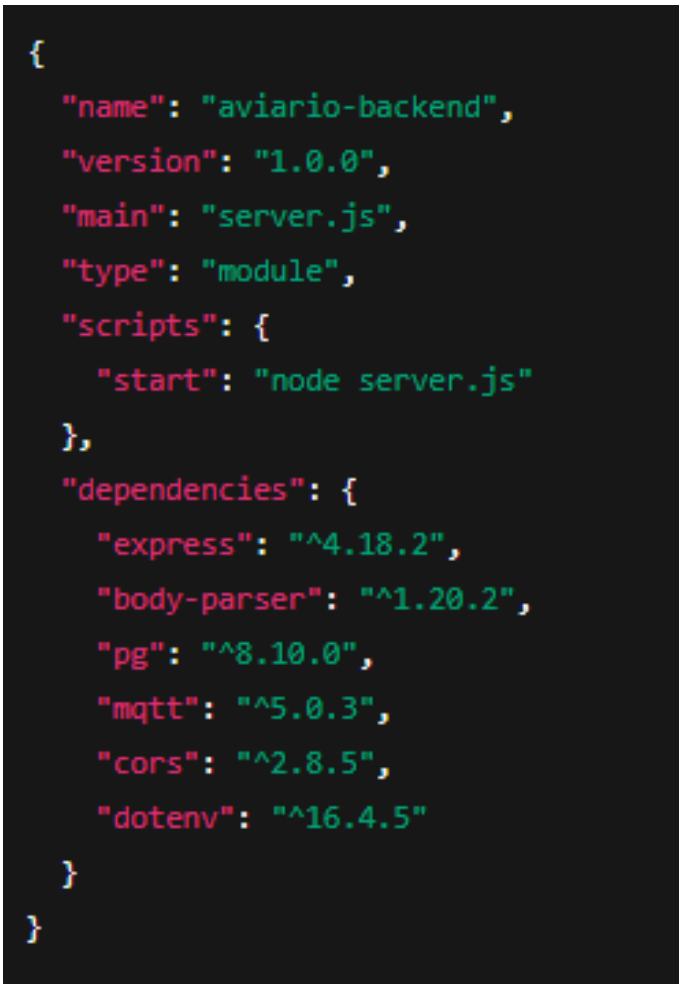
```
import { useEffect, useState } from "react";
import axios from "axios";
import ChartDashboard from "./components/ChartDashboard";

function App() {
 const [leituras, setLeituras] = useState([]);

 useEffect(() => {
  const fetchData = async () => {
   const res = await axios.get("http://localhost:3000/leituras");
   setLeituras(res.data);
  };
  fetchData();
  const interval = setInterval(fetchData, 5000);
  return () => clearInterval(interval);
 }, []);

 return (
  <div className="p-4 font-sans bg-gray-50 min-h-screen">
```

```
      <h1 className="text-3xl font-bold text-center mb-6">📊 Painel do Aviário IoT</h1>
      <ChartDashboard dados={leituras} />
    </div>
  );
}

export default App;
```

3.2. **Backend**

```
import express from "express";
import bodyParser from "body-parser";
import { connectDB, insertReading } from "./db.js";
import { initMQTT } from "./mqttHandler.js";
import cors from "cors";
const app = express();
app.use(bodyParser.json());
app.use(cors());
connectDB();
initMQTT(insertReading);
// Rotas REST
app.get("/", (req, res) => res.json({ message: "API Aviário IoT ativa" }));
app.get("/leituras", async (req, res) => {
 const result = await global.client.query("SELECT * FROM leituras ORDER BY id DESC LIMIT 50");
 res.json(result.rows);
});

const PORT = 3000;
app.listen(PORT, () => console.log(`Servidor rodando na porta ${PORT}`));
```

4. Dependências
   4.1. Backend



```
{
  "name": "aviario-backend",
  "version": "1.0.0",
  "main": "server.js",
  "type": "module",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "body-parser": "^1.20.2",
    "pg": "^8.10.0",
    "mqtt": "^5.0.3",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5"
  }
}
```

Imagem 3 - Dependências backend (Fonte: Autor)

4.2. Frontend

```
{
  "name": "aviario-frontend",
  "version": "1.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "axios": "^1.7.2",
    "chart.js": "^4.4.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-chartjs-2": "^5.2.0"
  },
  "devDependencies": {
    "vite": "^5.2.0"
  }
}
```

Imagem 4 - Dependências frontend (Fonte: Autor)

5. Observações

Com o uso de Docker como fazendo parte do ambiente de desenvolvimento, há integração de outros arquivos para rodar a aplicação. Arquivos gerados automaticamente por templates como Vite, Dockerfile, Docker-compose.yml, entre outros.

6. Execução

6.1. Estrutura Final do Projeto

Você deve ter essa estrutura de diretórios (após montar os arquivos):

aviario-iot-web/

│

├── docker-compose.yml

├── .env

│

├── backend/

```
|   ├── Dockerfile
|   ├── server.js
|   ├── db.js
|   ├── mqttHandler.js
|   └── package.json
|
├── frontend/
|   ├── Dockerfile
|   ├── package.json
|   ├── vite.config.js
|   └── src/
|       ├── App.jsx
|       ├── components/
|       └── pages/
|
└── mosquitto/
    └── config/mosquitto.conf
```

6.2. Criar o arquivo .env

Na **raiz do projeto**, crie um arquivo .env:

```
# Backend
DATABASE_URL=postgres://admin:admin123@db:5432/aviario
MQTT_BROKER_URL=mqtt://mqtt:1883

# Frontend
VITE_API_URL=http://api:3000
```

Esse arquivo será lido automaticamente pelo docker-compose.yml e pelos serviços Node.js.

6.3. Subir os containers com Docker Compose

Abra o terminal (na pasta do projeto) e rode:

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

💡 **O que acontece aqui:**

- Docker cria **4 containers**: db, mqtt, api, web.
- Os serviços são conectados em uma rede interna.
- O backend se conecta ao banco e ao broker MQTT automaticamente.
- O frontend se comunica com o backend via http://api:3000.

6.4. Verificar se tudo subiu corretamente

Após o docker-compose terminar de buildar, você verá logs como:

✅ Conectado ao PostgreSQL

📡 Conectado ao broker MQTT

Servidor rodando na porta 3000

Verifique também os containers:

docker ps

Deve aparecer algo como:

CONTAINER	IMAGE	STATUS	PORTS
aviario_web	node:20-alpine	Up	5173->5173
aviario_api	node:20-alpine	Up	3000->3000
aviario_db	postgres:15	Up	5432->5432
aviario_mqtt	eclipse-mosquitto	Up	1883->1883

6.5. Acessar a aplicação no navegador

Abra seu navegador e acesse:

🔶 **Frontend (dashboard web):**
👉 http://localhost:5173

🔶 **API Backend:**
👉 http://localhost:3000

Você deverá ver a mensagem:

{ "message": "API Aviário IoT ativa" }

6.6. Testar comunicação e visualização de dados

Caso você já tenha o ESP32 com o código IoT:

- Configure o ESP32 para usar o mesmo broker (MQTT_BROKER_URL = mqtt://mqtt:1883)
- Ele publicará dados em aviario/dados
- O backend escuta o tópico e salva no PostgreSQL
- O frontend atualiza os gráficos automaticamente

Caso ainda não tenha o hardware:

Você pode **simular publicações MQTT** com o comando:

```
docker exec -it aviario_mqtt mosquitto_pub -h mqtt -t "aviario/dados" -m '{"temperatura":28,"umidade":62,"co2":450,"amonia":8}'
```

O backend receberá e salvará essa leitura automaticamente 💾
O dashboard exibirá o ponto novo em poucos segundos.

6.7. Parar e limpar os containers

Quando quiser encerrar:

```
docker-compose down
```

Para remover volumes e dados do banco (reset completo):

```
docker-compose down -v
```

7. Referências

Pesquisas de conteúdos web