UNIVERSIDADE FEDERAL DE JUIZ DE FORA
FACULDADE DE ENGENHARIA
ENGENHARIA ELÉTRICA - HABILITAÇÃO EM ROBÓTICA E AUTOMAÇÃO INDUSTRIAL

Bernardo Capobiango de Andrade

Gateway MODBUS-MQTT para Sistemas de Automação Industrial Baseado em Plataforma Web

Juiz de Fora
2025

Bernardo Capobiango de Andrade

Gateway MODBUS-MQTT para Sistemas de Automação Industrial Baseado em Plataforma Web

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

Orientador: Prof. Dr. Guilherme Márcio Soares

Juiz de Fora

2025

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Andrade, Bernardo Capobiango de.
Gateway MODBUS-MQTT para Sistemas de Automação Industrial Baseado em Plataforma Web / Bernardo Capobiango de Andrade. – 2025.
79 f. : il.

Orientador: Guilherme Márcio Soares
Trabalho de Conclusão de Curso de Graduação – Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Engenharia Elétrica - Habilitação em Robótica e Automação Industrial, 2025.

1. Sistema IOT. 2. MQTT, Docker, Sparkplug. I. Soares,Guilherme M., orient. II. Título.

Bernardo Capobiango de Andrade

Gateway MODBUS-MQTT para Sistemas de Automação Industrial Baseado em Plataforma Web

Trabalho de conclusão de curso apresentado ao Departamento de Energia Elétrica da Universidade Federal de Juiz de Fora como requisito para aprovação na disciplina - Trabalho de Final de Curso.

Aprovado em 19 de Março de 2025

BANCA EXAMINADORA

Prof. Dr. Guilherme Márcio Soares - Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Leandro Rodrigues Manso Silva
Universidade Federal de Juiz de Fora

Dr. Matheus Alberto de Souza
Universidade Federal de Juiz de Fora

Dr. Sérgio Queiroz de Almeida
Companhia de Saneamento Municipal - Cesama

AGRADECIMENTOS

Primeiramente, agradeço a Deus, pela minha vida, e por me permitir ultrapassar todos os obstáculos encontrados ao longo da graduação e da realização deste trabalho.

Aos meus pais, Adriana e Moysés, que nunca mediram esforços para me ensinar e proporcionar o maior valor da vida: o amor. Sem vocês como inspiração e ponto de apoio nada disso seria possível, obrigado por me guiarem e serem exemplos nessa trajetória da vida.

À minha irmã mais velha Luiza, que sempre me serviu como inspiração e me guiou no caminho da Engenharia e da vida. Suas dicas, conselhos e amor foram pilares fundamentais nessa caminhada.

À minha irmã mais nova Júlia, que esteve ao meu lado nessa trajetória, suportando os dias difíceis e celebrando as conquistas.

À minha avó Maria Inez, que sempre me guiou e cuidou no caminho da empatia e carinho.

À minha namorada Isadora, que durante a graduação foi meu ponto de apoio, carinho e suporte em inúmeros momentos, me fazendo ver e resolver as dificuldades com outros olhos.

Ao grupo PET Elétrica UFJF, em especial o Prof. Danilo Pinto, que transformaram a minha experiência durante a graduação, ensinando a importância da faculdade além do campus e os valores essenciais da Engenharia, com os quais muito aprendi e continuo aprendendo.

Aos amigos que fiz durante a graduação, com os quais a caminhada foi mais divertida e enriquecedora, me ajudando nos dias mais difíceis.

Aos professores e servidores da UFJF, por proporcionarem um ambiente de educação pública de qualidade, que todos deveriam ter a oportunidade de vivenciar.

Ao meu orientador Guilherme, suas aulas não apenas ampliaram meus conhecimentos mas também me abriram os olhos para a área. Sua dedicação e orientação me tornaram um engenheiro e pessoa melhor.

RESUMO

Este trabalho apresenta o desenvolvimento de uma aplicação web fullstack para gerenciamento e encapsulamento de um gateway MQTT/Modbus destinado à CESAMA (Companhia de Saneamento Municipal). O sistema foi implementado utilizando Python com FastAPI no backend e React com Vite no frontend, estabelecendo uma arquitetura moderna e eficiente. A aplicação proporciona uma interface intuitiva para o gerenciamento dos dispositivos conectados ao gateway, facilitando a integração entre os protocolos MQTT e Modbus. Através da conteinerização com Docker, o sistema garante fácil implantação e manutenção em diferentes ambientes. As funcionalidades implementadas simplificam os processos de configuração, monitoramento e manutenção, garantindo uma experiência de usuário consistente, acessível e refinada. Este avanço tecnológico representa uma contribuição significativa para a gestão e modernização da infraestrutura industrial da CESAMA, alinhando-se aos padrões atuais do mercado.

Palavras-chave: Sistema IIOT. Docker. MQTT. SparkPlug. FastAPI. React.

ABSTRACT

This work presents the development of a fullstack web application for the management and encapsulation of an MQTT/Modbus gateway designed for CESAMA (Municipal Sanitation Company). The system was implemented using Python with FastAPI for the backend and React with Vite for the frontend, establishing a modern and efficient architecture. The application provides an intuitive interface for managing devices connected to the gateway, facilitating integration between MQTT and Modbus protocols. Through Docker containerization, the system ensures easy deployment and maintenance across different environments. The implemented features simplify the configuration, monitoring, and maintenance processes while ensuring a consistent, accessible, and refined user experience. This technological advancement represents a significant contribution to the management and modernization of CESAMA's industrial infrastructure, aligning with current market standards.

Keywords: IIOT System. Docker. MQTT. SparkPlug. FastAPI. React.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CLP	Controlador Lógico Programável
IIOT	Industrial Internet of Things (Internet das Coisas Industrial)
IOT	Internet of Things (Internet das Coisas)
IP	Internet Protocol
JWT	JSON Web Token
MQTT	Message Queuing Telemetry Transport
OPC	Open Platform Communications
ORM	Object-Relational Mapping
RBAC	Role-Based Access Control
REST	Representational State Transfer
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SPA	Single Page Application
TCP	Transmission Control Protocol
TI	Tecnologia da Informação
UI	User Interface

SUMÁRIO

1 Introdução

A indústria 4.0 tem tornado-se cada vez mais presente em sistemas da atualidade, buscando a coleta, armazenamento e análise de dados gerados pelas cadeias de produção, bem como a integração de sistemas e processos industriais de forma inteligente e eficiente. Nesse contexto, destaca-se o protocolo MQTT como uma nova alternativa para a comunicação entre dispositivos, devido à sua simplicidade, baixa complexidade e alta interoperabilidade. Neste capítulo, serão apresentados alguns conceitos gerais que mostram como a atualização tecnológica em empresas é importante, como a integração entre tecnologias, legadas e modernas, pode ser feita, bem como os desafios envolvidos na transição e quais soluções são adequadas.

1.1 Gateways de protocolos e atualizações tecnológicas

Os gateways de protocolos desempenham um papel crucial na interconexão de sistemas e dispositivos, especialmente em um cenário industrial moderno em constante evolução. Eles permitem a comunicação entre diferentes protocolos, facilitando a integração de tecnologias legadas com soluções modernas, como as que utilizam IoT e MQTT. Essa capacidade de adaptação é fundamental para empresas que buscam atualizar suas infraestruturas tecnológicas e melhorar a eficiência operacional.

O gateway de aplicação é utilizado para permitir a comunicação entre diferentes aplicações ou sistemas de software. Essencialmente, atua como um intermediário entre os sistemas, traduzindo os protocolos de comunicação utilizados por cada um deles. Isso permite que aplicações desenvolvidas em diferentes linguagens de programação ou que utilizem diferentes protocolos, possam se comunicar de forma transparente.

Como exemplo de um cenário industrial moderno, é possível a utilização de um gateway de aplicação para a integração de diferentes protocolos de comunicação. Em um caso específico, o exemplo inclui o MODBUS, um protocolo industrial tradicional com recursos limitados, e o MQTT, um protocolo contemporâneo para IoT, mas a mesma estratégia é aplicável com quaisquer outros protocolos. A função do gateway consiste na leitura dos dados em um formato (como MODBUS) e na republicação em outro (como MQTT), o que resulta em atualização tecnológica no transporte dos dados industriais sensíveis e importantes, sem dependência dos protocolos selecionados.

Ademais, é possível incrementar um gerenciamento simples, eficiente e moderno em cima dessa aplicação, tornando o sistema robusto e escalável.

1.2 Motivação e objetivos

No setor de distribuição e tratamento de água da cidade de Juiz de Fora, a CESAMA enfrenta desafios relacionados à comunicação e monitoramento remoto de suas estações de tratamento e distribuição. A necessidade de modernização das tecnologias de comunicação industrial sem interrupção no abastecimento de água constitui a principal motivação deste trabalho. O ambiente atual opera com tecnologias industriais tradicionais, porém limitadas, o que dificulta a escalabilidade e a manutenção do sistema.

A proposta inclui uma aplicação web fullstack moderna para o gerenciamento, manutenção e encapsulamento de um Gateway que permite a conexão entre as estações da planta e o software de supervisão E3. A implementação desta interface visa a substituição do protocolo industrial tradicional MODBUS por uma solução baseada no protocolo contemporâneo MQTT, mais adequado para comunicações em ambientes industriais modernos.

Este trabalho complementa a pesquisa anterior, citada em [7], que aborda aspectos técnicos da mesma problemática. A integração destas soluções resultará em um sistema completo para gerenciamento da planta da CESAMA, com capacidade de monitoramento do estado das conexões sem a necessidade de interrupções no abastecimento de água para a cidade.

1.3 Revisão do estado da técnica

Uma breve revisão da literatura revela que existem diversas soluções comerciais e acadêmicas voltadas para a conversão e gerenciamento de dados entre protocolos Modbus e MQTT.

1.3.1 Soluções comerciais

No mercado, aplicações como o vNode e NeuronEX oferecem soluções robustas para a conversão de protocolos e gerenciamento dos dados, permitindo, por exemplo, a integração de dispositivos Modbus a sistemas baseados em MQTT. Essas aplicações frequentemente fornecem interfaces web para configuração e suporte a padrões como MQTT Sparkplug.

1.3.1.1 vNode

O vNode [8] é concebido como um *Industrial IoT Gateway* que possibilita a coleta, o processamento e a entrega de informações oriundas da planta industrial. Sua arquitetura modular, que dispensa a necessidade de programação customizada, adota um conceito *plug-and-play*, permitindo que os módulos de conectividade sejam configurados de maneira rápida e intuitiva. Além disso, o vNode é compatível com diversas plataformas operacionais,

incluindo sistemas baseados em Linux, Windows e dispositivos embarcados com arquitetura ARM, o que reforça sua versatilidade em ambientes industriais heterogêneos. A arquitetura principal de funcionamento pode ser vista na Figura 1.

Figura 1 – Arquitetura de um dos usos de casos do vNode. Fonte: [1]

Para garantir a interoperabilidade entre os dispositivos de campo e os sistemas de supervisão (SCADA), o vNode integra uma ampla variedade de protocolos industriais, fundamentais para a comunicação no ambiente de automação. Entre os principais protocolos suportados, destacam-se:

- **Modbus** (TCP/IP e RTU): Protocolo tradicional e amplamente adotado para a comunicação entre dispositivos eletrônicos de automação.
- **OPC** (OPC DA e OPC UA): Padrões que permitem a interoperabilidade entre diferentes sistemas e dispositivos, facilitando a troca de dados em tempo real.
- **DNP3**: Utilizado principalmente em sistemas de controle de energia e automação de processos, oferecendo alta confiabilidade em ambientes críticos.
- **MQTT**: Protocolo leve de mensageria, ideal para a transmissão de dados em redes com restrição de banda ou alta latência.
- **REST**: Interface de comunicação que viabiliza a integração com aplicações web e plataformas de nuvem.

Adicionalmente, o vNode dispõe de módulos específicos para a comunicação com dispositivos industriais, como os controladores Siemens S7, e permite a coleta de dados a partir de arquivos em formatos TXT, CSV e XML. Na etapa de entrega, a plataforma se integra a sistemas de banco de dados (por exemplo, SQL e MongoDB), sistemas SCADA e até mesmo a serviços de nuvem como AWS, Google Cloud e Azure.

Em síntese, o vNode configura uma solução eficaz para os objetivos deste trabalho, proporcionando uma integração robusta e integrada para ambientes industriais. Entretanto,

por ser um serviço comercializado e fechado, não oferece o mesmo nível de flexibilidade e controle que uma implementação própria pode alcançar. Ademais, por se tratar de uma solução paga, os custos envolvidos podem ser exarcebados e proibitivos em determinados contextos.

1.3.1.2 NeuronEX

A *EMQ Technologies* é reconhecida mundialmente como a principal fornecedora de software para infraestrutura de dados IoT open-source. Com uma abordagem nativa em nuvem, a empresa capacita aplicações IoT à prova de futuro ao oferecer produtos integrados que conectam, movimentam, processam e integram dados em tempo real – desde o edge até a nuvem multi-cloud. Seu portfólio é liderado pelo *EMQX*, uma plataforma de mensageria MQTT open-source escalável e confiável.

Dentro do portfólio da EMQ, destaca-se o *Neuron* (também conhecido como NeuronEX [9]), uma solução avançada de *Industrial Edge Data Hub* projetada para ambientes industriais. O Neuron é uma ferramenta essencial para a transformação digital, pois permite o acesso em tempo real e a análise inteligente de dados provenientes de diferentes dispositivos e sistemas presentes na cadeia produtiva, tais como Controladores Lógicos Programáveis (CLPs), máquinas CNC, robôs, sistemas SCADA e sensores inteligentes. A arquitetura básica de utilização do NeuronEX pode ser observada na Figura 2.

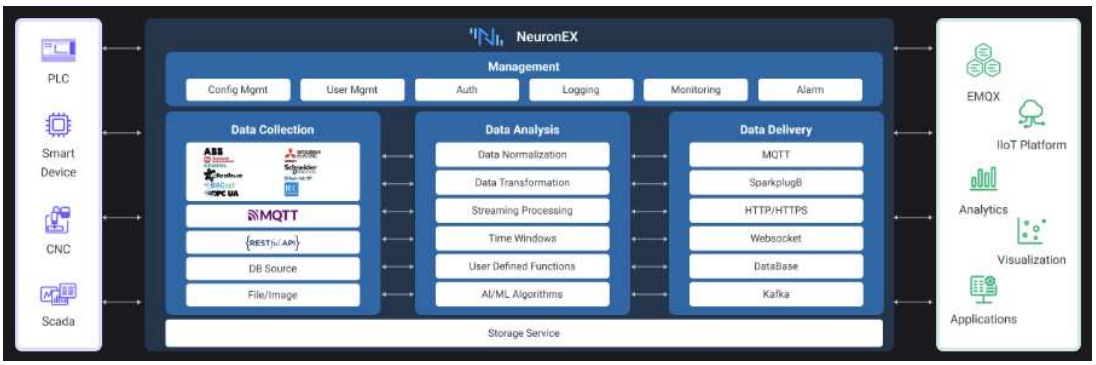


Figura 2 – Arquitetura básica do NeuronEX. Fonte: [2]

Uma das principais vantagens do Neuron é sua capacidade de integrar uma ampla variedade de protocolos industriais, o que facilita a comunicação entre os diversos equipamentos e sistemas de produção. Essa integração abrange:

- **MQTT:** Protocolo leve de mensageria, ideal para a transmissão de dados em redes com restrição de banda ou alta latência.
- **SparkPlugB:** Voltado para padronizar a comunicação entre dispositivos industriais, esse protocolo promove maior interoperabilidade e integração dos dados em ambientes de produção.

- **HTTP:** Permite a integração com serviços web e a transmissão de dados para plataformas na nuvem ou centros de dados locais, ampliando as possibilidades de conexão e análise.
- **WebSocket:** Protocolo que viabiliza a comunicação bidirecional contínua, essencial para aplicações que exigem baixa latência e atualizações em tempo real.

Além disso, o Neuron oferece suporte à coleta e processamento de dados por meio de funcionalidades como filtragem, limpeza, padronização e processamento em fluxo (streaming). Sua arquitetura modular e flexível possibilita a adaptação tanto a ambientes *on-premises* quanto a infraestruturas baseadas em contêineres (Docker, Kubernetes), contribuindo para uma integração eficiente entre as tecnologias de Informação (TI) e de Operações (OT).

Em síntese, o Neuron demonstra capacidade para operação em ambientes industriais, oferecendo funcionalidades de acesso em tempo real e análise de dados. A solução dispõe de um repositório no GitHub [10], permitindo acesso ao seu código-fonte e à colaboração da comunidade. Contudo, mesmo sendo open-source, o Neuron impõe um limite de uso de apenas 30 tags, uma restrição que pode se mostrar incompatível com o escopo de projetos mais robustos. Essa limitação pode resultar em custos adicionais que, em determinadas situações, tornam a solução economicamente inviável para o uso pretendido. Além disso, optar pelo desenvolvimento de uma solução própria pode oferecer maior controle sobre as funcionalidades necessárias e um potencial de customização superior, permitindo que os requisitos específicos do projeto sejam atendidos de maneira mais precisa e econômica.

1.3.2 Trabalhos correlatos

No âmbito acadêmico, diversas pesquisas têm se concentrado no desenvolvimento de soluções de gateway Modbus/MQTT, além de diversos trabalhos que implementam aplicações web com React. Serão exploradas as propostas feitas em [11, 12, 13, 14, 15].

1.3.2.1 Conversor Modbus/MQTT utilizando Raspberry Pi

Em [11] é apresentado o uso de um Raspberry Pi como conversor entre os protocolos Modbus e MQTT, para controle de uma planta didática.

O trabalho faz uma revisão sobre os conceitos envolvidos e mostra uma possível forma de fazer a conversão entre os protocolos. Por fim, apresenta os resultados do funcionamento e valida o conceito como viável para aplicação industrial.

1.3.2.2 Implementação de um Sistema Gateway MQTT-Modbus para Abstração de Redes Industriais

Em [12] é proposto o desenvolvimento de um gateway que integra redes industriais legadas, baseadas no protocolo Modbus RTU, ao ecossistema IoT via MQTT. Utilizando um ESP32 e um broker em JavaScript, o sistema permite que usuários, sem conhecimento aprofundado de Modbus, controlem e monitorem dispositivos industriais por meio de requisições JSON. Essa solução abstrai a complexidade do protocolo, implementando módulos de validação, conversão e sincronização de mensagens, além de oferecer controle de acesso granular. Testado em ambiente simulado, o gateway demonstra ser uma ferramenta confiável, escalável e segura para a integração de redes industriais com a Internet das Coisas.

1.3.2.3 Sistema "Meu TCC": Implementação do front-end de uma aplicação web para controle de tccs utilizando ReactJs

Em [13] é apresentada a implementação do Front-End de uma aplicação web denominada "Meu TCC", utilizando tecnologias modernas de desenvolvimento. Baseado em Javascript e React, o sistema emprega o conceito de Single Page Application (SPA), onde a interface é dividida em componentes reutilizáveis que são atualizados dinamicamente. A integração com a biblioteca Material UI proporcionou acesso a elementos pré-construídos baseados no Material Design da Google, agilizando o desenvolvimento e garantindo consistência visual.

1.3.2.4 Sistema Web IoT para Monitoramento Ambiental e Controle de Iluminação Utilizando Node-RED, MQTT e Comunicação Modbus em Plataforma Linux Embarcada

Em [14] é apresentado um sistema IoT baseado na web para monitoramento ambiental e controle de iluminação. A solução utiliza uma Raspberry Pi 4B como plataforma Linux embarcada, hospedando um broker MQTT e um servidor Node-RED para criar uma interface homem-máquina multiplataforma. Programas Python implementam múltiplas threads para eficiente troca de dados via protocolos Modbus TCP e MQTT, permitindo interação contínua entre PLCs e servidores broker. O sistema aproveita a ferramenta Node-RED para desenvolvimento de uma interface web com recursos de Publicação/Subscrição MQTT, garantindo comunicação fluida com o servidor.

Através da configuração do servidor HTTP Apache para direcionar ao diretório web do Node-RED, os pesquisadores conseguiram monitorar efetivamente as condições laboratoriais e gerenciar o sistema de iluminação. Os resultados experimentais validam a relação custo-benefício da abordagem na transformação de sistemas de controle tradicionais

em sistemas supervisórios baseados na web, ampliando a funcionalidade dos equipamentos existentes e reforçando a aplicabilidade das soluções IoT em ambientes práticos de pesquisa.

1.3.2.5 Gateway para Integração de Redes Industriais Modbus com Ecossistemas IoT via Protocolo MQTT

Em [15] é apresentado o desenvolvimento de um gateway que atua como ponte entre dispositivos industriais baseados no protocolo Modbus e aplicações de Internet das Coisas (IoT) que utilizam o protocolo MQTT. Implementado em um Raspberry Pi executando Node.js, o sistema permite que equipamentos industriais legados comuniquem-se diretamente com plataformas IoT modernas, viabilizando a coleta e análise de dados em tempo real. O gateway inclui uma interface web intuitiva para configuração e monitoramento, eliminando a necessidade de conhecimentos avançados sobre os protocolos envolvidos.

O autor validou o sistema como uma solução de baixo custo para projetos de automação industrial, conduzindo o trabalho na empresa SCADAHUB, demonstrando sua eficácia na integração de dispositivos Modbus tradicionais ao ecossistema IoT através do protocolo MQTT.

1.4 Organização do trabalho

Este trabalho foi dividido em quatro capítulos, de maneira a oferecer uma compreensão clara e sequencial das etapas desenvolvidas, desde a fundamentação teórica até a apresentação dos resultados práticos.

Inicialmente, a seção de Introdução traz o contexto acerca dos gateways de protocolos, aplicações na modernização, e apresenta a motivação que impulsionou a realização deste estudo, complementada por uma revisão do estado da téncica e análise de trabalhos correlatos e soluções comerciais existentes para problemáticas semelhantes.

O segundo capítulo foi estruturado em três partes interdependentes: Backend, Frontend e Infraestutura de TI. Em cada uma é explicada a arquitetura adotada e o ferramental utilizado, que possibilitaram a implementação do sistema.

O capítulo 3 traz os resultados obtidos ao longo do desenvolvimento, demonstrando por evidências como logs e capturas de telas, a integração e desempenho do sistema, assim como a integração com o gateway.

Por fim, o capítulo de Conclusões sintetiza os principais achados do estudo, discutindo os resultados alcançados com relação ao que foi proposto.

2 Desenvolvimento

Neste capítulo, será explicado o funcionamento e desenvolvimento do aplicativo como um todo. Existem três principais etapas: Backend, Frontend e Infraestrutura de TI. Cada uma delas será explicada em sua respectiva seção, contendo os detalhes de desenvolvimento, desafios e implementação.

2.1 Backend

O backend do sistema é a parte do aplicativo que roda no servidor, local ou hospedado na nuvem, gerenciando a lógica do aplicativo, o armazenamento, os serviços e manipulando os dados necessários. Essa parte do aplicativo também é responsável pela segurança dos usuários e é quem fornece os dados para a aplicação que interage com o usuário final.

2.1.1 Estrutura e responsabilidades do backend

No presente trabalho, a responsabilidade é gerenciar estações, dispositivos, tags, usuários e logs do sistema da CESAMA, assim como encapsular o serviço do gateway Modbus/MQTT, descrito no trabalho [7]. Ou seja, gerenciar as requisições que o frontend fará, para manipular, criar ou remover itens do banco de dados.

Com isso em vista, o backend foi estruturado conforme descrito na Figura 3.

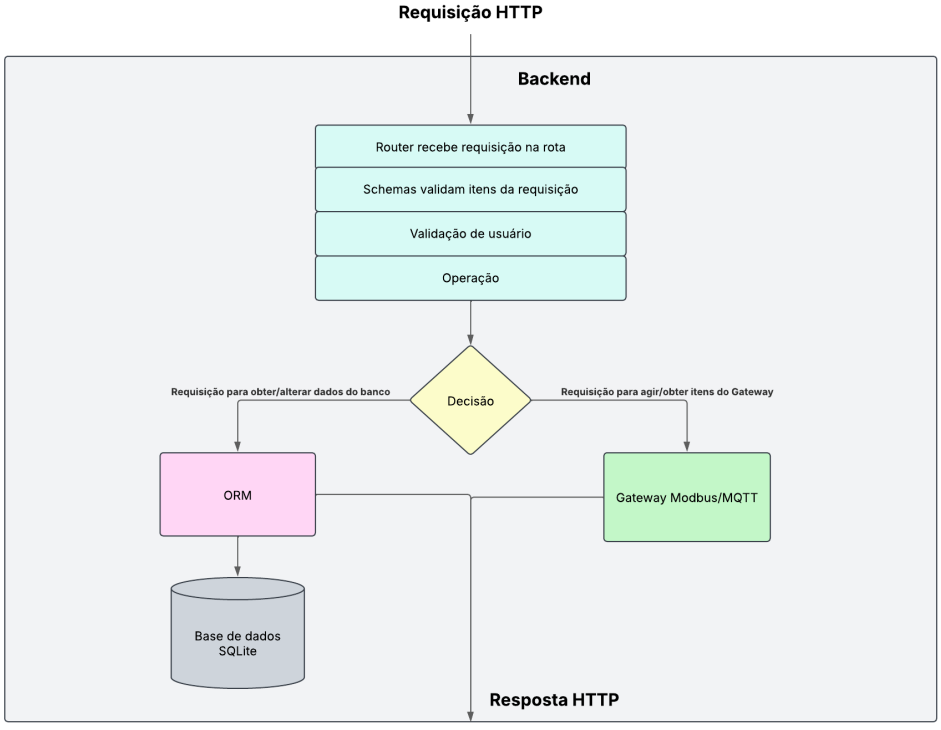


Figura 3 – Diagrama da estrutura pensada para o backend - Fonte: Elaborado pelo autor.

2.1.2 Tecnologias

Nesse tópico serão apresentadas as tecnologias e ferramentas utilizadas para o desenvolvimento do backend. Serão listadas as tecnologias base do projeto, o framework para APIs utilizado, o ferramental para controle do banco de dados e modelagem dos dados e por fim as ferramentas utilizadas para controle de segurança e geração de logs do aplicativo.

2.1.2.1 Base do projeto

Para um desenvolvimento ágil, seguro e completo, a linguagem de programação *Python* foi a escolhida para essa parte do aplicativo. A escolha se deu, principalmente, por ser a linguagem utilizada no desenvolvimento do sistema do gateway MQTT/Modbus em [7]. Assim, o backend pode encapsular o sistema facilmente como um serviço dele.

Para o gerenciamento de pacotes e bibliotecas, a fim de uma boa manutenção e gerenciamento do projeto, foi utilizado o *Poetry*, uma ferramenta para gerenciamento de dependências e pacotes do Python. A ferramenta tem seu código aberto [16]. Com ela, é possível detalhar em arquivos específicos do projeto quais versões de determinadas dependências serão usadas, e ao criar um ambiente novo a ferramenta instala tudo em um ambiente virtual específico para o projeto.

2.1.2.2 Multi-threading

Com a estrutura do backend em vista, foi necessário encontrar uma forma de desacoplar o gateway do processamento principal do backend. Assim, o sistema poderia receber e responder requisições, ler e alterar o banco de dados sem interromper ou prejudicar a execução das tarefas do gateway.

Dessa forma, foi utilizada da biblioteca *threading* do Python. Ela permite a criação e gerenciamento de múltiplas threads, que são caminhos de execução independentes dentro do mesmo processo. Então, diferentes partes do código são executadas concorrentemente. No caso do projeto, é utilizado da concorrência para executar o backend e o gateway simultaneamente, sem que um interfira no outro.

2.1.2.3 Framework - FastAPI

Dentro da linguagem foi escolhido o framework web *FastAPI* para a construção das APIs. O *FastAPI* é de código aberto [17], e considerado um dos frameworks mais rápidos na linguagem, além disso é de fácil e rápida implementação, facilitando o desenvolvimento para o trabalho. Com isso, foi possível gerenciar rotas para a aplicação de forma fácil e simples, preocupando apenas com a implementação e manipulação dos dados necessários [18].

2.1.2.4 Banco de dados e modelagem

Para o banco de dados, como a aplicação não é tão grande, com imensos volumes de dados, foi escolhida uma abordagem mais simples e de fácil implementação. Foi utilizado um banco de dados local *SQLite* em um único arquivo *database.db*.

Entretanto, o projeto não lida com comandos SQL diretos, mas sim utiliza da técnica Mapeamento Objeto-Relacional (ORM). É uma técnica de programação que vincula (ou mapeia) objetos a registros de banco de dados. Em outras palavras, um ORM permite que você interaja com seu banco de dados, como se estivesse trabalhando com objetos Python. Por essa razão, abstraem-se os registros e comandos SQL e é feita a manipulação de objetos.

Para o trabalho, foi utilizado o ORM *SQLAlchemy* [19]. Com ele foi possível modelar e persistir informações sobre as estações, dispositivos, tags e configurações do sistema.

Outra problemática é que, ao longo do desenvolvimento o banco de dados vai modificando-se, novas tabelas e colunas surgem, bem como novos indexadores, relações e restrições. Com isso, é necessário uma ferramenta para gerenciar as migrações do banco de dados. As migrações são utilizadas para modificar ou atualizar a estrutura do banco de dados, permitindo ações como a criação de novas tabelas, a adição de colunas ou a alteração do tipo de dado de um campo. Elas são fundamentais para manter um histórico das mudanças realizadas no esquema do banco ao longo do tempo. Além disso, possibilitam a reversão para uma versão anterior do esquema, caso seja necessário. Para isso, foi utilizado do *Alembic* que é uma ferramenta leve para migrações de banco de dados, especialmente para o SQLAlchemy, uma vez que foi feita pelo mesmo autor [20].

Por último, em um sistema backend é necessário utilizar da validação de dados, para garantir que os dados passados nas requisições estão corretos, gerar documentações robustas e tornar o sistema completo e robusto. Para isso, foi utilizada a biblioteca *Pydantic* [21], para validação de dados que trabalha perfeitamente com o *FastAPI* . Com o *Pydantic*, você define modelos usando classes tipadas que validam entradas, convertem tipos e geram documentação JSON. No *FastAPI*, o *Pydantic* valida requisições, assegurando conformidade dos dados antes do processamento. Esta integração melhora a segurança e facilita o desenvolvimento pela geração automática de documentação via *Swagger UI* e *ReDoc*.

2.1.2.5 Segurança e logging

No contexto do backend, a segurança das informações é extremamente importante. Nesse cenário, duas coisas básicas são necessárias: autenticação para os endpoints e encriptar senhas de usuários do sistema.

O processo de autenticação verifica a identidade do usuário, enquanto a autorização determina quais ações esse usuário pode executar. A autenticação é implementada através do JSON Web Token (JWT) e criam-se regras de autorização para controlar o acesso aos endpoints. Para o gerenciamento de tokens JWT foi usada a biblioteca *PyJWT* [22].

O JWT é um padrão (RFC 7519) que define uma maneira compacta e autônoma de transmitir informações entre as partes de maneira segura. Essas informações são transmitidas como um objeto JSON que é digitalmente assinado usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA. Ele consiste em três partes: header, payload e assinatura. Elas são separadas por pontos (.) e, juntas, formam um token JWT. É importante ressaltar que, apesar de a informação em um JWT estar codificada, ela não está criptografada. Isso significa que qualquer pessoa com acesso ao token pode decodificar e ler suas informações. No entanto, sem o segredo usado para assinar o token, não é possível alterar as informações ou forjar um novo token. Portanto, não deve-se incluir informações sensíveis ou confidenciais no payload do JWT.

A estrutura do JWT possibilita a inclusão de *claims*, que são declarações sobre uma entidade (geralmente o usuário) e informações adicionais úteis para a aplicação. As claims podem ser categorizadas em três tipos:

- **Registered Claims:** São claims pré-definidas e recomendadas pelo padrão, como `iss` (emissor), `sub` (assunto), `aud` (destinatário), `exp` (tempo de expiração), `nbf` (não antes de), `iat` (data de emissão) e `jti` (identificador único do token). Essas claims ajudam a padronizar a validação e a gerência do token.
- **Public Claims:** São claims que podem ser definidas livremente, mas para evitar colisões de nomenclatura, devem ser registradas ou conter um identificador único, como um URI.
- **Private Claims:** São claims customizadas e definidas pelas partes envolvidas (emissor e consumidor do token), que carregam informações específicas da aplicação, como privilégios de acesso, preferências de usuário, entre outros dados.

Na prática, ao gerar um JWT, o desenvolvedor define quais claims serão incluídas conforme as necessidades do sistema. Por exemplo, a inclusão da claim `exp` garante que o token possua um tempo de validade determinado, evitando seu uso prolongado ou indevido. Outras claims, como uma customizada, podem indicar outras informações, facilitando a implementação de regras de negócio. Assim, o JWT não apenas assegura a autenticação, mas também fornece informações essenciais para a aplicação.

Para as senhas dos usuários, foi adodtada a biblioteca *pwdlib* que permitirá criptografar adequadamente as senhas [23].

Ademais, é de suma importância ter rastreabilidade do sistema, bem como ferramentas para auditoria e verificações de segurança. Dessa forma, faz-se necessário o uso de tecnologias de *logging* para registrar operações e informações relevantes do sistema como um todo. Foi escolhida a biblioteca *Daiquiri*, um sistema de *logging* que fornece rastreabilidade para todas as operações do gateway, registra eventos importantes como conexões estabelecidas, erros de comunicação e operações de leitura/escrita [24].

2.1.3 Modelagem do banco de dados

A partir da problemática descrita na Seção 1.2, é identificada a estrutura das estações da CESAMA, partindo da estrutura principal, a estação de tratamento:

- Cada estação possui N dispositivos. (Relação 1:N)
- Cada dispositivo possui N tags de leitura/escrita (Relação 1:N)

Mas, antes de mencionar a modelagem dos dados e como foi pensado o banco de dados, é preciso listar os tipos de variáveis utilizadas no trabalho. Alguns tipos são básicos dos bancos de dados, outros são tipos *enum* onde apenas determinados valores são permitidos, conforme listado abaixo:

- **Tipos primitivos**
  - **Int**: Utilizado para armazenar valores inteiros.
  - **Booleano**: Utilizado para armazenar valores lógicos, como flags.
  - **Varchar**: Utilizado para armazenar strings de texto variável.
  - **Timestamp**: Utilizado para armazenar datas e horários, como data de criação e data de atualização dos registros.
- **Tipos enumerados**
  - **UserRole**: Define os papéis de usuário no sistema:
    - *admin*: Administrador com acesso completo
    - *manager*: Gerente com acesso intermediário
    - *viewer*: Visualizador de dados com acesso restrito
  - **Resource**: Define os recursos do sistema que podem estar sujeitos a permissões:
    - *station*: Estações
    - *device*: Dispositivos
    - *tag*: Tags de dados
    - *user*: Usuários do sistema

    - *permission*: Permissões de acesso
    - *bridge*: Gateway
    - *logs*: Registros de log
    - *mqtt_broker_config*: Configuração do broker MQTT
  - **Action**: Define as ações possíveis sobre os recursos:
    - *create*: Criar
    - *read*: Ler
    - *update*: Atualizar
    - *delete*: Excluir
    - *bridge_action*: Ação específica do gateway

- **Relações entre tabelas**: As relações são implementadas com o uso de chaves estrangeiras que referenciam identificadores primários:
  - *station_id*: Chave estrangeira que referencia um item da tabela de estações
  - *device_id*: Chave estrangeira que referencia um item da tabela de tags

Com os tipos de dados citados estabelecidos, é possível agora falar da modelagem das tabelas e relacionamentos dos dados.

Para a publicação dos dados pelo gateway, como citado no trabalho [7], é utilizado o protocolo Sparkplug para MQTT. Nesse cenário, o *edge of node* seria exatamente a tag de leitura/escrita, pois ela é o ponto final da estrutura.

Com todo o citado em vista, o banco de dados da aplicação foi modelado a fim de conseguir correlacionar todos esses dados estruturados em árvore, onde uma estação tem N dispositivos e um dispositivo tem N tags.

Então, foi criada uma tabela para as estações, outra para os dispositivos e outra para as tags. Em cada uma delas, os itens são armazenados de forma independente, entretanto, existe a correlação entre eles pelo uso de chaves primárias de relacionamentos. Cada dispositivo contém o id da sua estação correspondente, e cada tag tem o id do seu dispositivo correspondente. Na configuração do banco, foi feito o relacionamento de forma que, se o item pai é excluído, o item filho também é.

Ademais, para gerenciamento do sistema foi criada uma tabela de usuários, que contém os dados básicos de cadastro como login e senha criptografada, mas contém também o cargo daquele usuário (UserRole). Tal dado é relevante para que seja possível o gerenciamento de permissões do que cada usuário pode ou não fazer dentro do sistema, como ler apenas certos itens, atualizá-los ou criá-los e removê-los. Dessa forma, estrutura-se um sistema completamente controlado pelos administradores, do que os outros cargos podem ou não executar. Como complemento dessas permissões, foi criada uma tabela de

permissões, onde é configurado o recurso do permissionamento (Resource), a ação (Action), o cargo da permissão cadastrada (UserRole) e um booleano se é ou não permitido o acesso daquela ação, naquele recurso, por aquele cargo.

Por último, foi criada uma tabela para armazenar as configurações de acesso ao broker MQTT, a tabela *mqtt_broker_config*, que é pretendida ter apenas um registro com os dados de acesso ao broker como usuário e senha.

É possível ter uma visualização mais robusta e completa da modelagem do banco em um diagrama do mesmo. Este está presente na Figura 4 abaixo.

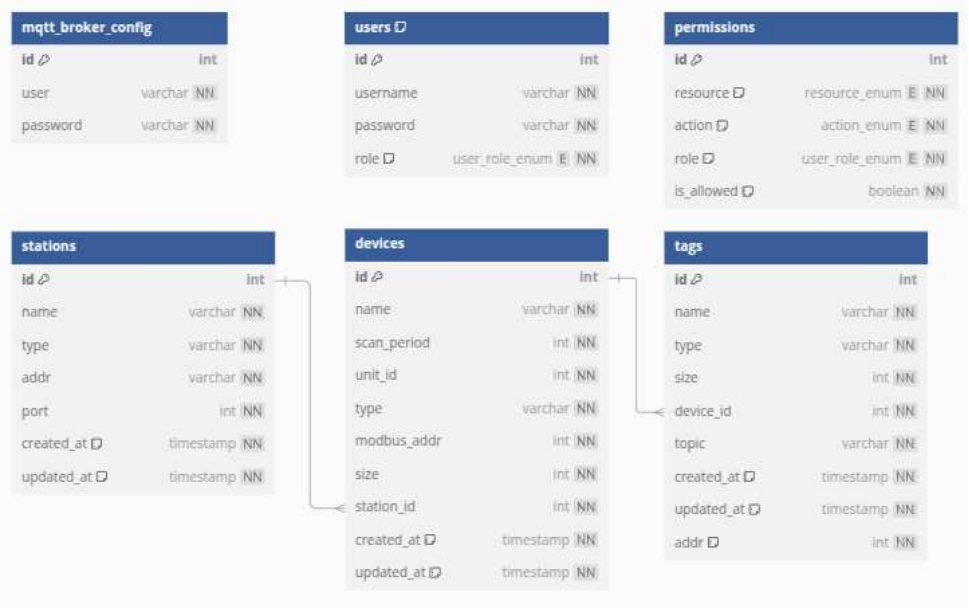


Figura 4 – Diagrama das tabelas e variáveis do banco de dados - Fonte: Elaborado pelo autor

Nesse contexto, o SQLAlchemy, mencionado na sub-seção *2.1.2.4*, atua transformando os dados dessas tabelas em objetos Python. Dessa forma, é possível, por exemplo, executar uma busca por uma estação e ter um campo desse objeto que carrega automaticamente todos os dispositivos vinculados com aquela estação, facilitando a interação com os dados e uso deles para o Gateway. Visualiza-se assim, a importância do ORM na prática, e como usufruir dos benefícios, abstraindo a camada do SQL.

2.1.4 Login e Tokens

Nessa sub-seção será abordado como foi feito a parte de login na aplicação, como o sistema gerencia os tokens para autenticação contínua do usuário e como é feito o armazenamento seguro das senhas no sistema.

Conforme descrito na sub-seção *2.1.2.5*, foi utilizado o token JWT para realizar a autenticação nos chamados.

Em uma aplicação web, o processo de autenticação geralmente tem um fluxo padrão, que foi o escolhido para este trabalho, e segue da seguinte maneira:

1. O usuário entra com as suas credenciais no cliente (frontend) e envia para o servidor em uma rota de geração de token;
2. O servidor verifica as credenciais, buscando pelo usuário no banco de usuários, e, se estiverem corretas, gera um token JWT e o envia de volta ao cliente;
3. Nas solicitações subsequentes, o cliente deve incluir esse token no cabeçalho de autorização de suas solicitações HTTP. Como, por exemplo: *Authorization: Bearer <token>*;
4. Quando o servidor recebe uma solicitação com um token JWT, ele pode verificar a assinatura e, se o token é valido e não expirou, ele processa a solicitação do cliente;
5. Caso esteja perto da expiração e o cliente queira renovar o seu token, existe uma rota para renovação que ele pode fazer uma requisição, passando o token antigo (ainda válido), e o servidor deve gerar um novo token e retorná-lo;

Este fluxo pode ser também exemplificado pela Figura 5 abaixo.

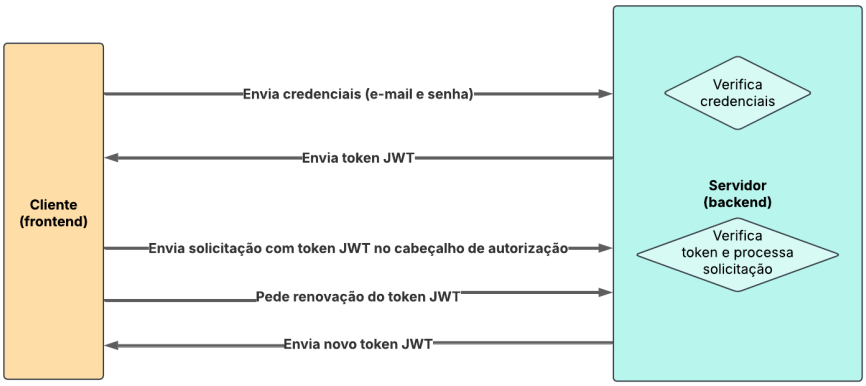


Figura 5 – Fluxo de autenticação da aplicação - Fonte: Elaborado pelo autor

Com o fluxo de autenticação esclarecido, deve ser explicado o armazenamento seguro das senhas, antes de explicar como é feito o login e gerenciamento dos tokens.

Para as senhas, conforme citado na sub-seção *2.1.2.5* é utilizada de uma biblioteca para gerenciar hashs, encriptar e validar hashs. No projeto, foi optado pelas recomendações padrões da bibliteca para o *PasswordHash*, assim cria-se o contexto que encripta e valida os hashs de forma fácil e eficiente.

Então, basicamente o fluxo na aplicação é:

1. Ao criar um usuário, é utilizado o contexto da biblioteca para criar um hash da senha preenchida (simplesmente com o comando *pwd_context.hash(password)*) e é armazenado no banco de dados esse hash da senha.

2. Ao executar o login, busca-se o usuário no banco de dados. Então, é utilizado um comando da biblioteca (*pwd_context.verify(plain_password, hashed_password)*) para validar se a senha preenchida confere com o hash armazenado. É retornado verdadeiro ou falso e o login é efetuado ou não.

Dessa forma, é garantida a segurança das senhas do usuário, pois a senha nunca é salva inteira no banco de dados, apenas um hash criptografado dela, que não pode ser decriptografado, apenas comparado.

Com o processo de login feito, o sistema então gera um token para o cliente poder se autenticar nas suas requisições. O login é feito na rota */auth/token* - as rotas serão explicadas na seção **2.1.5** - e retorna o token JWT, conforme já explicado.

Como uma camada a mais de segurança na aplicação, utiliza-se do arquivo *.env* para armazenar as variáveis críticas do sistema, como tempo de expiração do token JWT (informação de payload do token), chave screta para encriptação do token JWT (assinatura do token) e o algoritmo usado para codificação do token. Assim, é garantido que tais dados não estejam expostos em código e a assinatura, por exemplo, não seja exposta e não comprometa os tokens que o cliente recebe e usa.

Para a geração do token JWT, são utilizadas apenas as claims 'sub' para informar o *username* do usuário e 'exp' para informar o tempo de expiração do token. Para codificar o token, é utilizada a função *encode* da biblioteca, passando o payload, a assinatura e o algoritmo para codificação. Ao final, é retornado o token para o cliente poder utilizar, caso o login seja realizado com sucesso.

2.1.5 Rotas

Em uma aplicação web, a URL principal funciona como a porta de entrada para o servidor, definindo a base a partir da qual os diversos caminhos (ou rotas) são estruturados. Essa URL base, por exemplo `http://seuservidor.com`, estabelece o ponto de partida para acessar os recursos e funcionalidades da aplicação, como informações de usuários, produtos, pedidos e outros serviços essenciais. A partir dela, cada rota adiciona um segmento que direciona a requisição para uma função específica no backend.

Por exemplo, ao configurar a rota `/usuarios` para responder a uma requisição GET, o acesso a `http://seuservidor.com/usuarios` pode retornar uma lista de usuários cadastrados. Da mesma forma, a rota `/produtos` pode ser utilizada para cadastrar, atualizar ou excluir produtos, dependendo do método HTTP empregado (GET, POST, PUT, DELETE). Essa abordagem modular, característica do *FastAPI*, torna o sistema

mais organizado e intuitivo, facilitando tanto o desenvolvimento quanto a manutenção e a escalabilidade do projeto.

A estruturação do backend foi realizada utilizando o `APIRouter` do FastAPI, o que possibilitou uma organização modular das rotas. Todos os arquivos que definem os endpoints foram agrupados na pasta `routers`, localizada dentro da pasta `src`. Dessa forma, cada conjunto de funcionalidades possui seu próprio módulo, o que facilita a manutenção, o reaproveitamento de código e a escalabilidade do projeto. Um exemplo de uso é demonstrado em [25], e exemplificado pela Figura 6 abaixo.

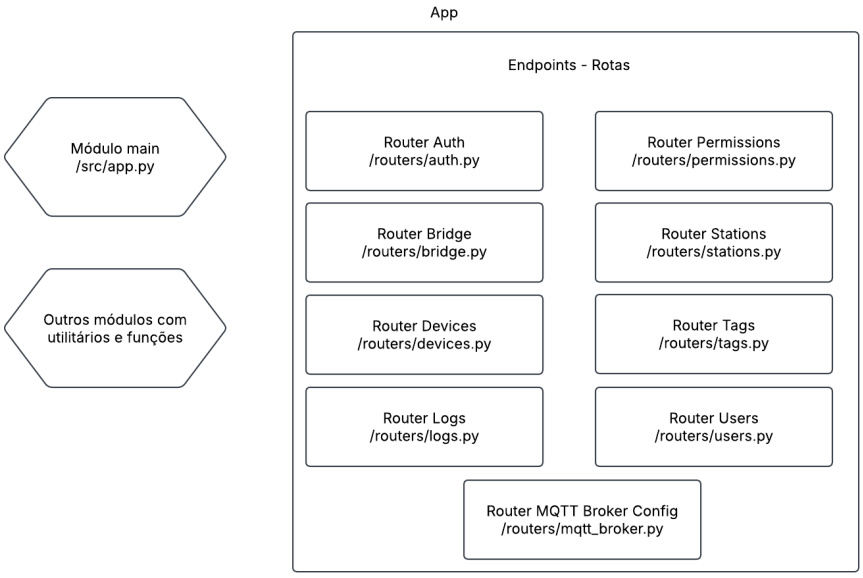


Figura 6 – Estrutura de módulos e rotas do backend - Fonte: Elaborado pelo autor

No arquivo `app.py`, presente na pasta `src`, o aplicativo principal é instanciado e as rotas definidas nos módulos da pasta `routers` são integradas por meio do `APIRouter`. Essa abordagem promove uma clara separação entre a lógica de negócio e a interface de comunicação com o cliente, permitindo que novas funcionalidades sejam adicionadas de maneira simples e organizada. Além disso, o uso do FastAPI contribui para a geração automática de uma documentação interativa da API, agilizando os testes e a compreensão dos recursos expostos, pontos fundamentais para um projeto robusto e de alta qualidade.

No presente projeto, as rotas foram estruturadas da seguinte maneira:

- **Autenticação (/auth/*)**: Contém rotas relacionadas à autenticação e autorização de usuários. Rotas internas:
  - `POST /auth/token`: Realiza o login de um usuário e retorna um token JWT.
  - `POST /refresh_token`: Atualiza o token JWT de um usuário
- **Bridge (/bridge/*)**: Contém rotas para gerenciamento do Gateway MQTT/Modbus. Rotas internas:

  - `POST /bridge/start`: Inicializa o gateway;
  - `POST /bridge/stop`: Para o gateway;
  - `POST /bridge/restart`: Reinicia o gateway;
  - `GET /bridge/status`: Obtém o status do gateway
  - `GET /bridge/station-status`: Obtém o status atual da conexão de todas as estações

- **Dispositivos (/devices/*)**: Contém rotas para gerenciamento de dispositivos. Rotas internas:
  - `POST /devices/`: Cria um novo dispositivo.
  - `GET /devices/{device_id}`: Obtém os dados de um dispositivo específico.
  - `GET /devices/`: Lista múltiplos dispositivos com suporte a paginação.
  - `PUT /devices/{device_id}`: Atualiza os dados de um dispositivo específico.
  - `DELETE /devices/{device_id}`: Exclui um dispositivo específico.

- **Usuários (/users/*)**: Contém rotas para gerenciamento de usuários. Rotas internas:
  - `POST /users/`: Cria um novo usuário.
  - `GET /users/`: Lista múltiplos usuários com suporte a paginação.
  - `GET /users/{user_id}`: Obtém os dados de um usuário específico.
  - `PUT /users/{user_id}`: Atualiza os dados de um usuário específico.
  - `DELETE /users/{user_id}`: Exclui um usuário específico.

- **Tags (/tags/*)**: Contém rotas para gerenciamento de tags. Rotas internas:
  - `POST /tags/`: Cria uma nova tag.
  - `GET /tags/{tag_id}`: Obtém os dados de uma tag específica.
  - `GET /tags/`: Lista múltiplas tags com suporte a paginação.
  - `PUT /tags/{tag_id}`: Atualiza os dados de uma tag específica.
  - `DELETE /tags/{tag_id}`: Exclui uma tag específica.

- **Estações (/stations/*)**: Contém rotas para gerenciamento de estações. Rotas internas:
  - `POST /stations/`: Cria uma nova estação.
  - `GET /stations/{station_id}`: Obtém os dados de uma estação específica.
  - `GET /stations/`: Lista múltiplas estações com suporte a paginação.

  - `PUT /stations/{station_id}`: Atualiza os dados de uma estação específica.
  - `DELETE /stations/{station_id}`: Exclui uma estação específica.

- **Permissões (/permissions/*)**: Contém rotas para gerenciamento de permissões. Rotas internas:
  - `POST /permissions/`: Cria uma nova permissão.
  - `GET /permissions/`: Lista múltiplas permissões com suporte a paginação.
  - `GET /permissions/{permission_id}`: Obtém os dados de uma permissão específica.
  - `PUT /permissions/{permission_id}`: Atualiza os dados de uma permissão específica.
  - `DELETE /permissions/{permission_id}`: Exclui uma permissão específica.

- **Broker MQTT (/mqtt_broker/*)**: Contém rotas para gerenciamento da configuração do broker MQTT. Rotas internas:
  - `POST /mqtt_broker/`: Cria uma nova configuração do broker MQTT.
  - `GET /mqtt_broker/`: Obtém a configuração atual do broker MQTT.
  - `PUT /mqtt_broker/`: Atualiza a configuração do broker MQTT.

- **Logs (/logs/*)**: Contém rotas para acesso e visualização dos logs da aplicação. Rotas internas:
  - `GET /logs/categories`: Lista todas as categorias de logs disponíveis.
  - `GET /logs/files/{main_category}/{subcategory}`: Lista todos os arquivos de log disponíveis para uma categoria específica.
  - `GET /logs/content/{main_category}/{subcategory}/{filename}`: Retorna o conteúdo de um arquivo de log específico.

Assim, a organização das responsabilidades e endpoints fica bem separadas, respeitando as requisições HTTP e sendo fácil de manter o sistema. Ainda, existe um sistema de validação no início de cada endpoint, para garantir que o usuário solicitando tenha permissão para tal.

2.1.5.1 Validação de Permissões nos endpoints

O sistema implementa um mecanismo de controle de acesso baseado em papéis (conforme já explicado) através de uma função `validate_user`, que é invocada em todos os endpoints antes da execução da operação principal. Esta função verifica se o usuário autenticado possui as permissões necessárias para realizar a ação solicitada.

A função consulta o banco de dados em busca de uma permissão que corresponda ao trio (recurso, ação, papel do usuário). Caso não exista uma permissão correspondente ou esta não permita a ação, uma exceção HTTP 403 (Forbidden) é lançada, impedindo o acesso não autorizado.

Esta abordagem garante a aplicação consistente das políticas de autorização em todas as operações da API e assim, proporciona uma camada de segurança eficaz e centralizada para o controle de acesso aos recursos do sistema.

2.1.6 Gerenciamento automático dos tópicos MQTT

No sistema, as tags possuem um campo `topic` que segue o padrão SparkplugB para comunicação MQTT. Este campo é crítico para o correto funcionamento da infraestrutura de comunicação, pois define como os dispositivos e seus dados são descobertos e acessados na rede.

O padrão SparkplugB utilizado segue uma estrutura que permite a organização hierárquica das informações e facilita o roteamento de mensagens entre os componentes do sistema:

```
spBv1.0/{tipo_estação}/DDATA/{nome_estação}/{nome_dispositivo}/{nome_tag}
```

Para garantir a consistência dos tópicos, o sistema implementa um mecanismo automático de gerenciamento que:

1. Gera o tópico para cada nova tag criada, baseando-se nas informações da tag, do dispositivo associado e da estação à qual o dispositivo pertence.

2. Mantém os tópicos atualizados quando ocorrem alterações em qualquer nível da hierarquia. Quando uma estação tem seu nome ou tipo alterado, todos os tópicos de todas as tags associadas aos dispositivos conectados a esta estação são automaticamente atualizados.

3. De forma similar, quando um dispositivo tem seu nome modificado, os tópicos de todas as suas tags são recalculados para refletir a nova nomenclatura.

4. O sistema utiliza eventos de sessão do SQLAlchemy para interceptar operações de persistência (flush) e garantir que os tópicos estejam sempre sincronizados antes de qualquer alteração ser efetivamente registrada no banco de dados.

Esta abordagem elimina a necessidade de atualização manual dos tópicos e reduz significativamente o risco de inconsistências na comunicação MQTT, que poderiam resultar em falhas na coleta ou distribuição de dados no sistema de supervisão industrial.

2.1.7 Gerenciamento do Gateway Modbus/MQTT

O sistema implementa o Gateway Modbus/MQTT que atua como intermediário na comunicação entre dispositivos industriais e aplicações externas. Para garantir o correto funcionamento deste componente crítico, foi desenvolvido um mecanismo robusto de gerenciamento do ciclo de vida do Gateway.

A arquitetura de gerenciamento do Gateway é composta por três componentes principais: o `BridgeManager`, uma instância global deste gerenciador, e um gerenciador de ciclo de vida integrado à aplicação FastAPI. Esta estrutura permite o controle eficiente do Gateway, facilitando sua inicialização automática, monitoramento de estado e encerramento controlado.

Durante a inicialização da aplicação, o sistema executa automaticamente o processo de configuração e inicialização do Gateway através de um contexto assíncrono (`lifespan`). Este processo inclui a consulta ao banco de dados para obter as estações configuradas, juntamente com seus dispositivos e tags associados, além das credenciais do broker MQTT. Com estes dados, o Gateway é iniciado em uma thread separada, permitindo que opere independentemente do servidor web principal.

O mecanismo de gerenciamento implementa um padrão thread-safe para controlar o Gateway, utilizando locks para evitar condições de corrida durante operações críticas. Isto garante que apenas uma instância do Gateway esteja em execução a qualquer momento, evitando duplicação de conexões e conflitos de recursos.

Para fins de monitoramento, o sistema mantém informações sobre o status de conectividade de cada estação configurada, permitindo que os administradores verifiquem rapidamente o estado operacional do Gateway. Estas informações são acessíveis através de endpoints dedicados na API, conforme explicitado na sub-seção **2.1.5**.

O encerramento controlado do Gateway é outro aspecto importante do gerenciamento. Quando a aplicação é finalizada, o sistema garante que todas as conexões sejam adequadamente encerradas e os recursos liberados, evitando vazamentos de memória e conexões pendentes.

Uma descrição mais detalhada sobre o funcionamento interno do Gateway Modbus/MQTT, incluindo seus mecanismos de comunicação e processamento de dados, pode ser encontrada em [7].

2.1.8 Documentação interativa com Swagger

O sistema disponibiliza documentação interativa através do Swagger UI, recurso integrado ao framework FastAPI. Esta interface permite visualizar, testar e interagir com todos os endpoints da API sem a necessidade de ferramentas adicionais. Para acessar a documentação, basta navegar até o caminho `/docs` da aplicação (por exemplo,

`http://localhost:8000/docs`). A interface apresenta todos os endpoints organizados por tags, conforme definido nos roteadores da aplicação. Cada endpoint exibe informações detalhadas sobre:

- Método HTTP utilizado (GET, POST, PUT, DELETE)
- Parâmetros esperados (path, query, body)
- Esquemas de dados de entrada e saída
- Códigos de status possíveis e seus significados
- Requisitos de autenticação

A funcionalidade de "Try it out"permite testar os endpoints diretamente pelo navegador, preenchendo os parâmetros necessários e visualizando as respostas em tempo real. Para endpoints que exigem autenticação, é possível utilizar o botão "Authorize"para fornecer um token JWT válido ou fazer login com usuário e senha. Esta documentação automática é gerada a partir dos tipos, docstrings e anotações de tipo presentes no código, garantindo que esteja sempre sincronizada com a implementação atual da API. Isto facilita significativamente o desenvolvimento, testes e integração com outros sistemas.

2.1.9 Gerenciamento de Migrações com Alembic

O sistema utiliza o Alembic, uma ferramenta de migração de banco de dados para SQLAlchemy, para gerenciar as alterações no esquema do banco de dados de forma controlada e versionada. As migrações são essenciais para a manutenção da integridade dos dados durante o ciclo de vida da aplicação, permitindo evoluir o esquema do banco sem perder dados existentes.

No contexto da aplicação, o Alembic é configurado para trabalhar diretamente com os modelos de dados definidos no módulo `models.py`, aproveitando a definição de metadados fornecida pelo registro de tabelas SQLAlchemy (`table_registry`). Esta abordagem garante que qualquer alteração nos modelos, como adição de novas entidades, modificação de campos existentes ou criação de relacionamentos, seja automaticamente detectada e incorporada nas migrações.

O processo de migração ocorre em duas etapas principais: primeiramente, o Alembic compara o estado atual do banco de dados com a definição dos modelos no código, gerando scripts de migração que representam as diferenças encontradas. Em seguida, estes scripts são aplicados ao banco de dados, executando as alterações necessárias de forma ordenada e segura.

Uma característica importante da configuração do Alembic no sistema é a capacidade de funcionar tanto em modo "online"quanto "offline". No modo online, as migrações

são executadas diretamente contra um banco de dados em execução, enquanto no modo offline os scripts são gerados para execução posterior, o que é útil em ambientes de produção onde o acesso direto ao banco pode ser restrito.

Esta abordagem de gerenciamento de esquema proporciona diversos benefícios:

- Controle de versão do esquema de banco de dados, permitindo rastrear mudanças ao longo do tempo
- Capacidade de reverter alterações problemáticas através de migrações de downgrade
- Sincronização automática entre o modelo de dados no código e sua representação no banco
- Facilitação da colaboração entre desenvolvedores, minimizando conflitos de esquema
- Implantação controlada de mudanças em ambientes de produção

O Alembic trabalha em conjunto com o SQLAlchemy para interpretar as classes mapeadas como `Station`, `Device`, `Tag` e demais entidades, incluindo seus relacionamentos e restrições, garantindo que o esquema do banco de dados permaneça sempre coerente com a lógica de negócios implementada na aplicação.

2.1.10 Gerenciamento de Logs

O sistema implementa uma infraestrutura robusta de logs, fundamental para o monitoramento operacional, diagnóstico de problemas e auditoria das atividades realizadas na aplicação.

2.1.10.1 Arquitetura de Logs

A arquitetura de logs do sistema segue uma abordagem modular e hierárquica, com separação clara entre os logs da API REST e os logs do Gateway Modbus/MQTT. Os arquivos de log são organizados em diretórios específicos (`/app/logs/api` e `/app/logs/bridge`), facilitando a localização e gestão por parte dos administradores do sistema. Para cada componente funcional do backend, é configurado um logger dedicado com rotação diária de arquivos e período de retenção parametrizável (padrão de 30 dias). Esta estratégia de rotação previne o crescimento descontrolado dos arquivos de log, mantendo apenas os registros mais relevantes e recentes.

2.1.10.2 Categorização e Níveis de Log

O sistema implementa uma categorização refinada dos logs através de loggers específicos para cada módulo:

- **Logs de API**: Registram as operações CRUD realizadas nos endpoints, como criação, leitura, atualização e exclusão de estações, dispositivos, tags e usuários. Cada endpoint registra o autor da ação e os detalhes da operação, conforme visto no módulo de estações.
- **Logs do Gateway**: Documentam o funcionamento do Gateway Modbus/MQTT, incluindo conexões estabelecidas, leituras de dados dos dispositivos, publicações em tópicos MQTT e eventuais erros de comunicação.
- **Logs de Estações**: Para cada estação configurada, um logger específico é criado dinamicamente através da função `get_station_logger`, permitindo o acompanhamento individualizado de cada estação industrial.

Os logs utilizam níveis de severidade padronizados (INFO, WARNING, ERROR, CRITICAL) para classificar as mensagens conforme sua importância e urgência.

2.1.10.3 Integração com os Componentes do Sistema

Nos controladores de rota, como exemplificado no módulo de estações, cada operação CRUD é acompanhada de registros de log que documentam a ação realizada, o usuário responsável e o resultado da operação. Por exemplo, quando uma estação é criada, atualizada ou excluída, o sistema registra automaticamente detalhes como o nome da estação e o usuário que executou a ação.

No Gateway Modbus/MQTT, os logs são utilizados para monitorar o estado das conexões com as estações industriais, registrar as leituras de dados realizadas e documentar as publicações nos tópicos MQTT. Cada estação possui seu próprio arquivo de log, o que permite o diagnóstico preciso de problemas específicos.

2.1.10.4 Acesso aos Logs via API

O sistema disponibiliza um endpoint específico para acesso aos logs, implementado no roteador `logs`. Este endpoint permite que o frontend obtenha acesso aos registros de log para visualização e análise pelo administrador do sistema. As operações disponíveis incluem:

- Listar os arquivos de log disponíveis
- Recuperar o conteúdo de arquivos de log específicos
- Filtrar logs por data, componente ou nível de severidade

O acesso a estes endpoints é controlado pelo mesmo mecanismo de autorização utilizado nos demais recursos do sistema, garantindo que apenas usuários com as permissões

adequadas possam acessar estas informações sensíveis. A validação de permissões é realizada através da função `validate_user`, que verifica se o usuário atual possui o papel necessário para acessar o recurso `LOGS`.

2.1.10.5 Segurança e Integridade

O sistema de logs foi projetado considerando aspectos de segurança importantes:

- Sanitização de informações sensíveis antes do registro nos logs
- Controle de acesso baseado em papéis para visualização dos logs
- Preservação da integridade dos registros para fins de auditoria
- Rotação automática para evitar exposição prolongada de dados históricos

O sistema de gerenciamento de logs implementado constitui um componente crítico para a operação e manutenção da aplicação, proporcionando visibilidade detalhada sobre todas as atividades realizadas no sistema. A abordagem estruturada e granular adotada facilita o diagnóstico de problemas, permite o monitoramento proativo do funcionamento do sistema e fornece uma trilha de auditoria completa para fins de segurança e conformidade.

2.2 Frontend

O desenvolvimento do frontend para o sistema Gateway MODBUS/MQTT constitui um componente essencial deste projeto, sendo responsável pela interface com a qual os usuários interagem para gerenciar estações, dispositivos e tags. Esta seção apresenta os aspectos técnicos e metodológicos empregados na construção da camada de apresentação do sistema, abordando desde a arquitetura geral até os componentes específicos implementados.

A abordagem adotada prioriza a experiência do usuário, a modularidade do código e a manutenibilidade, elementos fundamentais para garantir a longevidade e a evolução do sistema em contextos industriais. As tecnologias selecionadas e as estratégias de implementação foram criteriosamente escolhidas para atender aos requisitos funcionais e não-funcionais estabelecidos, considerando o contexto operacional da CESAMA e as melhores práticas de desenvolvimento web moderno.

2.2.1 Visão Geral da Arquitetura

A arquitetura do frontend foi projetada seguindo o paradigma de Single Page Application (SPA), no qual o carregamento inicial do aplicativo ocorre uma única vez, e as subsequentes interações do usuário resultam em atualizações dinâmicas do conteúdo sem a necessidade de recarregamentos completos da página. A Figura 7 abaixo exemplifica a

diferença do ciclo de uma aplicação SPA para uma aplicação tradicional. Na aplicação tradicional, após cada requisição uma página inteira nova é carregada, já na aplicação SPA, como já mencionado, ocorre apenas a troca de informações e/ou views/urls já carregadas.

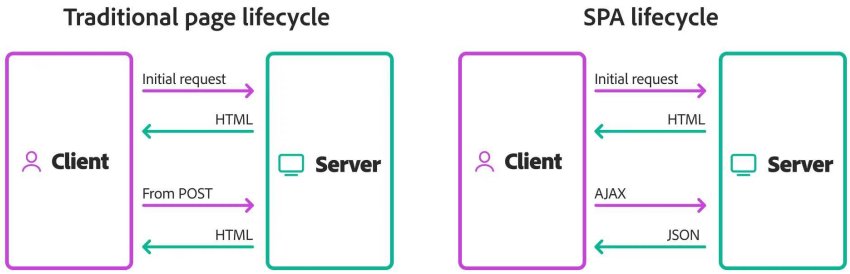


Figura 7 – Comparativo entre ciclo de vida de uma página tradicional e uma SPA - Fonte: [3]

Esta abordagem proporciona uma experiência mais fluida e responsiva ao usuário final, aproximando-se da experiência de uso de aplicações desktop tradicionais.

A biblioteca React foi selecionada como fundamento tecnológico principal por sua eficiência no gerenciamento do ciclo de vida dos componentes e pela renderização otimizada através do Virtual DOM. A utilização do React possibilita a construção de interfaces modulares e reativas, onde cada componente encapsula sua própria lógica, estado e apresentação. Esta característica favorece a manutenibilidade e a testabilidade do código, aspectos cruciais para sistemas que demandam evolução contínua.

A estruturação do código-fonte segue um modelo organizacional baseado em responsabilidades funcionais, conforme ilustrado na hierarquia de diretórios:

```
src/
 assets/         # Recursos estáticos (imagens, ícones)
 components/     # Componentes UI reutilizáveis
 context/       # Provedores de Contexto React
 pages/         # Componentes de página
 api/           # Configuração e instâncias do Axios
 routes/        # Configuração de rotas
 theme/         # Configuração do tema Material-UI
```

Esta organização favorece a separação de responsabilidades, onde cada diretório agrupa arquivos com propósitos similares. As pastas e arquivos podem ser visualizados na Figura 8 abaixo.

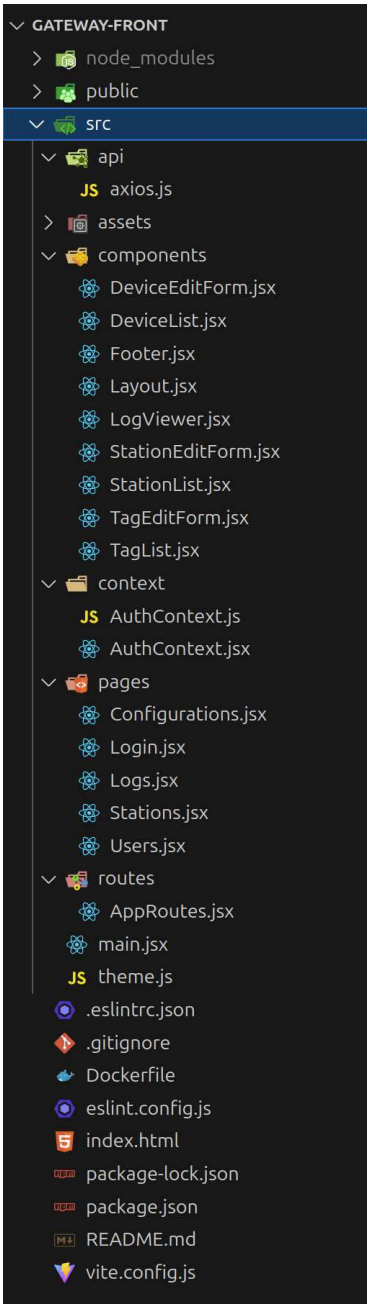


Figura 8 – Estrutura das pastas e arquivos visualizado pelo editor Visual Studio Code - Fonte: Elaborado pelo autor

O diretório `components` contém elementos reutilizáveis da interface do usuário, enquanto `pages` abriga os componentes que representam visões completas correspondentes a rotas específicas. O diretório `context` centraliza os provedores de contexto do React, fundamentais para o gerenciamento de estado global da aplicação.

Em termos de padrões de design, adotou-se extensivamente o conceito de componentes funcionais com hooks, uma abordagem moderna do React que substituiu os componentes baseados em classes. Esta decisão favorece a escrita de código mais conciso, facilita o compartilhamento de lógica entre componentes e otimiza o processo de renderização. Os principais hooks utilizados incluem:

- `useState`: Para gerenciamento de estado local dos componentes
- `useEffect`: Para execução de efeitos colaterais sincronizados com o ciclo de vida do componente
- `useContext`: Para acesso ao estado global disponibilizado pelos provedores de contexto
- `useCallback`: Para memoização de funções, evitando recriações desnecessárias em renderizações subsequentes

A comunicação com o backend é centralizada através de um cliente HTTP baseado em Axios, configurado com interceptores para inclusão automática de tokens de autenticação. Este padrão garante que todas as requisições sigam um formato consistente e facilita a implementação de mecanismos globais de tratamento de erros.

Para o gerenciamento de rotas, implementou-se o React Router, que possibilita a navegação entre diferentes visões sem recarregamento completo da página.

As rotas são protegidas por um componente de alto nível (`ProtectedRoute`) que verifica a autenticação e as permissões do usuário antes de renderizar o conteúdo solicitado.

A arquitetura também incorpora o conceito de "lifting state up"(elevação de estado), onde o estado compartilhado por múltiplos componentes é mantido no ancestral comum mais próximo. Este padrão reduz a complexidade do fluxo de dados e facilita o rastreamento de mudanças de estado durante o desenvolvimento e a depuração.

Em síntese, a arquitetura frontend do sistema Gateway MODBUS/MQTT foi concebida com foco na modularidade, manutenibilidade e experiência do usuário, utilizando tecnologias e padrões modernos de desenvolvimento web para criar uma interface robusta, eficiente e evolutiva. Com base nos arquivos do projeto, será elaborada a subseção sobre Tecnologias e Bibliotecas Utilizadas, considerando a versão mais recente do React utilizada.

2.2.2 Tecnologias e Bibliotecas Utilizadas

O desenvolvimento do frontend do Gateway MODBUS/MQTT foi fundamentado em um conjunto criteriosamente selecionado de tecnologias e bibliotecas, escolhidas com base em sua maturidade, suporte da comunidade, desempenho e adequação aos requisitos do projeto. As mesmas são brevemente representadas na Figura 9. A seguir, apresenta-se uma análise detalhada de cada componente tecnológico empregado.

Figura 9 – Stack tecnológica utilizada no desenvolvimento do frontend. Fonte: Elaborado pelo autor

2.2.2.1 React

O React, desenvolvido e mantido pelo Meta (anteriormente Facebook), foi escolhido como biblioteca principal para a construção da interface devido à sua abordagem declarativa, que permite a criação de interfaces de usuário complexas a partir de componentes modulares [26]. Para este projeto, utilizou-se a versão 19, que representa uma das iterações mais recentes e estáveis da biblioteca. A natureza baseada em componentes do React ofereceu várias vantagens significativas para o projeto:

- **Componentização**: A capacidade de dividir a interface em componentes reutilizáveis reduziu a duplicação de código e aumentou a manutenibilidade do sistema.
- **Virtual DOM**: Uma representação em memória da estrutura do DOM (Document Object Model, a estrutura hierárquica de elementos HTML da página). Enquanto o DOM real é manipulado diretamente pelo navegador e tem alto custo computacional em cada atualização, o Virtual DOM atua como uma camada intermediária que

permite ao React identificar precisamente quais elementos precisam ser atualizados, minimizando manipulações no DOM real e resultando em melhor desempenho.

  - **DOM (Document Object Model)**: É uma representação estruturada em forma de árvore de todos os elementos HTML de uma página web. O navegador utiliza esta estrutura para renderizar a página e cada modificação nela desencadeia uma nova renderização, processo que pode ser computacionalmente custoso, especialmente em aplicações complexas.

- **Fluxo de dados unidirecional**: O modelo de fluxo de dados do React, onde as informações fluem de componentes pais para filhos, torna o código mais previsível e facilita a depuração. Tal fluxo pode ser exemplificado pela Figura 10 abaixo, onde o componente "filho"recebe dados do componente "pai", mas pode utilizar alguma função *callback* para atualizar o componente "pai".

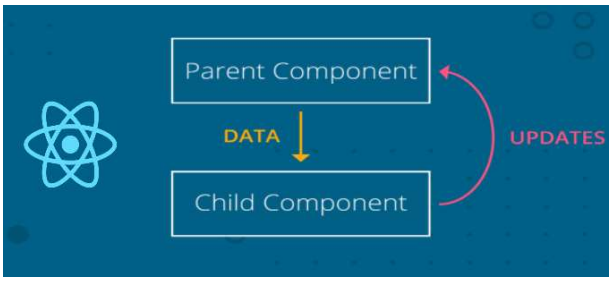


Figura 10 – Exemplo do fluxo de dados unidirecional do React - Fonte: [4]

- **Ecossistema rico**: O amplo ecossistema do React proporcionou acesso a diversas bibliotecas complementares que aceleraram o desenvolvimento.

A versão utilizada introduziu melhorias significativas na API de renderização e no desempenho geral, particularmente em relação ao gerenciamento de efeitos colaterais e renderização condicional. Este projeto fez uso extensivo dos Hooks do React, como `useState`, `useEffect`, `useContext` e `useCallback`, que representam o paradigma moderno de desenvolvimento em React, facilitando a gestão de estado e ciclo de vida dos componentes sem a necessidade de classes [27].

2.2.2.2 Vite

O Vite foi adotado como ferramenta de build e servidor de desenvolvimento, em substituição ao tradicional Create React App, por oferecer vantagens significativas em termos de desempenho [28]. A configuração do projeto em `vite.config.js` demonstra uma implementação minimalista, porém eficaz, desta ferramenta:

- **Inicialização instantânea**: O Vite, em sua versão 6.0.5, implementa um sistema de divisão de módulos de aplicação e dependências, permitindo que o servidor de

desenvolvimento inicie quase instantaneamente, independentemente do tamanho da aplicação.

- **Hot Module Replacement (HMR)**: A substituição de módulos em tempo real é significativamente mais rápida no Vite devido à sua arquitetura baseada em ESM (ECMAScript Modules).
- **Otimizações de produção**: Para builds de produção, o Vite utiliza Rollup, oferecendo excelentes otimizações de tree-shaking e splitting de código.
- **Configuração simplificada**: A configuração mínima necessária para iniciar o desenvolvimento facilitou o setup inicial do projeto, como evidenciado pela configuração concisa do arquivo `vite.config.js`.
- **Plugin ESLint integrado**: A utilização do plugin `vite-plugin-eslint` permitiu a verificação de qualidade de código durante o desenvolvimento, aumentando a confiabilidade e consistência do código produzido.

A experiência de desenvolvimento aprimorada proporcionada pelo Vite resultou em ciclos de desenvolvimento mais rápidos, especialmente durante a fase de implementação e testes da interface do usuário.

2.2.2.3 Material-UI

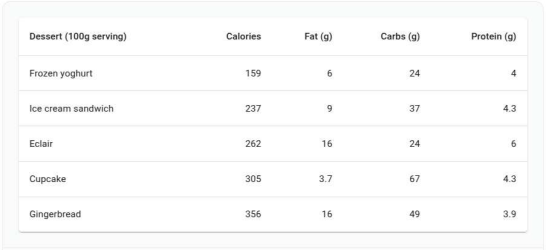
O Material-UI (MUI) foi selecionado como framework de componentes de interface do usuário [5]. A escolha foi baseada nos seguintes fatores:

- **Design System consolidado**: Baseado nas diretrizes do Material Design do Google, o MUI oferece uma linguagem visual consistente e familiar aos usuários.
- **Componentes ricos**: A biblioteca fornece uma ampla gama de componentes pré-construídos que atendem às necessidades de interfaces industriais, como tabelas, diálogos, formulários e navegação.
- **Responsividade**: Os componentes do MUI são nativamente responsivos, adaptando-se a diferentes tamanhos de tela sem configuração adicional extensiva.
- **Personalização**: O sistema de temas do MUI permite a customização consistente de cores, tipografia e outros aspectos visuais através de configuração centralizada, como explicitado em [29].
- **Acessibilidade**: Os componentes seguem as diretrizes WCAG, garantindo que a interface seja utilizável por pessoas com diferentes necessidades.

No projeto, utilizou-se extensivamente componentes como `Box`, `Grid2`, `Table`, `Dialog`, `Snackbar` e `Card` para estruturar a interface de gerenciamento de estações, dispositivos e tags. A biblioteca `@mui/icons-material` forneceu um conjunto abrangente de ícones utilizados para enriquecer a experiência do usuário e melhorar a usabilidade da interface. Alguns dos exemplos dos componentes dados na documentação oficial estão na Figura 11 abaixo.

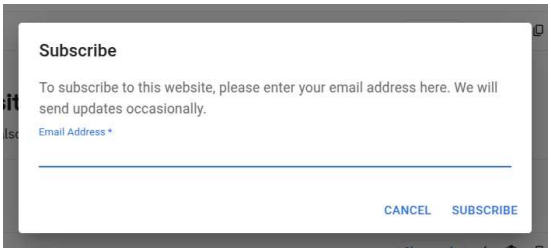
(a) Componente Card exemplo

(b) Componente Grid exemplo



Dessert (100g serving)	Calories	Fat (g)	Carbs (g)	Protein (g)
Frozen yoghurt	159	6	24	4
Ice cream sandwich	237	9	37	4.3
Eclair	262	16	24	6
Cupcake	305	3.7	67	4.3
Gingerbread	356	16	49	3.9

(c) Componente Table exemplo



(d) Componente Dialog exemplo

Figura 11 – Componentes de exemplo do Material UI - Fonte: [5]

2.2.2.4 React Router

Para o gerenciamento de navegação na aplicação, o React Router [30] foi implementado devido a:

- **Roteamento declarativo**: Permite definir rotas como componentes React, facilitando a organização da navegação.
- **Roteamento aninhado**: Suporta a criação de layouts aninhados com rotas filhas, essencial para implementar o padrão de navegação hierárquica do sistema.
- **Navegação programática**: Oferece APIs para navegação baseada em eventos ou lógica de negócios.

- **Parâmetros de rota**: Facilita a captura de parâmetros dinâmicos da URL, útil para exibição contextual de recursos.
- **Melhorias de desempenho**: A versão 7 introduz otimizações de desempenho significativas em relação às versões anteriores.

No sistema desenvolvido, o React Router foi fundamental para implementar o componente `ProtectedRoute`, que verifica a autenticação e autorização do usuário antes de renderizar páginas protegidas, redirecionando usuários não autorizados para a página de login.

2.2.2.5 Axios

O Axios foi escolhido como cliente HTTP para comunicação com a API backend [31] pelos seguintes fatores:

- **API baseada em Promises**: Facilita o trabalho com operações assíncronas utilizando sintaxe moderna de JavaScript.
- **Interceptores**: Permite a interceptação de requisições e respostas para manipulação global, utilizado no projeto para inclusão automática de tokens de autenticação em cabeçalhos.
- **Transformação de dados**: Oferece funções para transformação automática de dados enviados e recebidos.
- **Cancelamento de requisições**: Suporta o cancelamento de requisições em andamento, útil para evitar atualizações de estado em componentes desmontados.
- **Tratamento de erros consistente**: Proporciona um formato padronizado para tratamento de erros HTTP.

No projeto, o Axios foi configurado como uma instância centralizada no diretório `api`, com interceptores para autenticação e tratamento uniforme de erros, garantindo uma comunicação robusta e consistente com o backend.

2.2.2.6 Context API

Para gerenciamento de estado global, optou-se pela Context API nativa do React em detrimento de bibliotecas externas [32]. Esta escolha foi motivada por:

- **Simplicidade**: A API de contexto oferece uma solução mais direta para compartilhamento de estado entre componentes distantes na árvore de renderização.

- **Integração nativa**: Sendo parte do próprio React, não introduz dependências adicionais ou paradigmas conflitantes.
- **Suficiência**: Para as necessidades de estado global deste projeto, principalmente relacionadas à autenticação e autorização, a Context API mostrou-se suficiente sem a complexidade adicional de soluções como Redux.

O projeto utiliza principalmente o `AuthContext` para gerenciar o estado de autenticação e as permissões do usuário, tornando estas informações acessíveis a qualquer componente.

2.2.2.7 Bibliotecas Auxiliares

Além das tecnologias principais, o projeto também fez uso de bibliotecas auxiliares que contribuíram para sua robustez e funcionalidade:

- **PropTypes**: Utilizada para validação de tipos em tempo de desenvolvimento, aumentando a confiabilidade dos componentes através da definição clara de suas interfaces [33].
- **File-Saver**: Implementada para facilitar o download de arquivos gerados pelo sistema, como relatórios ou logs [34].
- **ESLint**: Ferramenta de análise estática que garantiu a qualidade e consistência do código, configurada com plugins específicos para React (`eslint-plugin-react` e `eslint-plugin-react-hooks`) [35].

A combinação destas tecnologias e bibliotecas resultou em uma base sólida para o desenvolvimento do frontend do Gateway MODBUS/MQTT, possibilitando a criação de uma interface de usuário moderna, responsiva e eficiente, alinhada às necessidades dos usuários finais e aos requisitos técnicos do projeto.

2.2.3 Sistema de Autenticação e Autorização

O sistema implementa autenticação baseada em JSON Web Tokens (JWT), onde as credenciais do usuário são validadas pelo servidor, que retorna um token contendo informações sobre identidade e permissões. Este token é armazenado localmente e incluído automaticamente em todas as requisições subsequentes, permitindo acesso contínuo sem necessidade de reautenticação frequente.

A gestão do estado de autenticação é centralizada através do Context API do React, que disponibiliza informações sobre o usuário logado e suas permissões para todos

os componentes da aplicação. Esta abordagem simplifica o gerenciamento de sessão e facilita a adaptação da interface conforme o nível de acesso.

O controle de acesso baseado em papéis (RBAC) estabelece três níveis de usuários: administradores com acesso irrestrito, gerentes com permissão para gerenciar estações e dispositivos, e visualizadores com acesso somente leitura. A implementação de rotas protegidas através do React Router garante que páginas restritas só possam ser acessadas por usuários com as permissões adequadas.

2.2.4 Gerenciamento de Estado

O gerenciamento de estado da aplicação utiliza uma combinação estratégica da Context API para estados globais (como autenticação) e o hook useState para estados locais de componentes. Esta abordagem equilibra a necessidade de compartilhamento de informações entre componentes distantes com a modularidade e independência de estados específicos. Na Figura 12 abaixo é exemplificada a passagem de informações entre componentes distantes a partir de um Contexto (Context API), a partir dele é possível fornecer dados para componentes distantes, sem necessariamente ter que passar entre os seus componentes pais.

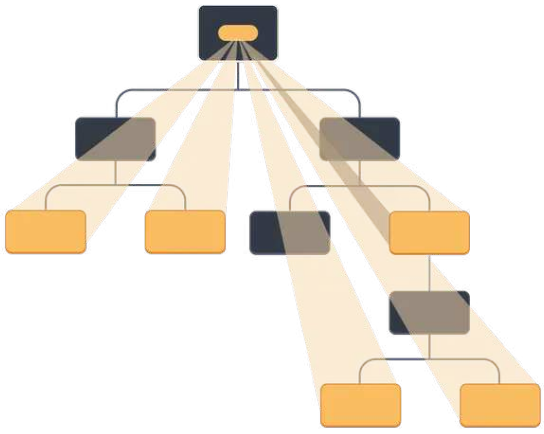


Figura 12 – Exemplo de passagem de informações para componentes distantes pelo Context API - Fonte: [6]

Informações críticas como tokens de autenticação são persistidas no localStorage do navegador, permitindo restauração automática da sessão em recarregamentos ou reabertura da aplicação. A verificação de validade destes dados durante a inicialização garante segurança sem comprometer a experiência do usuário.

Na tela de configuração de estações, implementou-se um sistema de "transações pendentes"onde alterações são mantidas inicialmente apenas na interface, permitindo ao usuário visualizar o impacto das mudanças antes de confirmar todas em uma única operação ou descartá-las completamente. Esta abordagem melhora a consistência dos

dados e reduz o risco de configurações parciais ou inconsistentes. A Figura 13 demonstra o modal para confirmar as alterações feitas durante o "modo de edição".

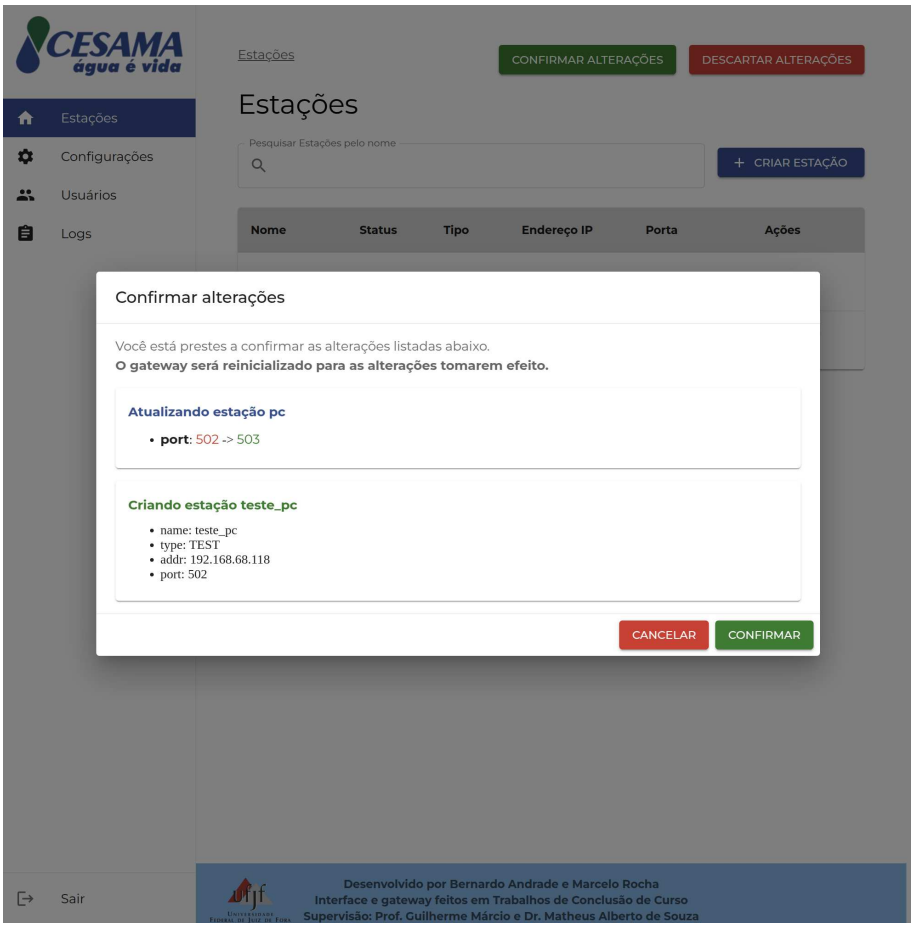


Figura 13 – Exemplo de transações pendentes - Fonte: Elaborado pelo autor

2.2.5 Interface do Usuário

A interface foi construída sobre o Material-UI, implementando um design system consistente com paleta de cores funcionais e inspiradas na paleta da CESAMA, tipografia hierárquica e densidade de informação balanceada. Elementos visuais específicos como indicadores de status e navegação hierárquica foram desenvolvidos para atender às necessidades particulares do monitoramento industrial. As cores e tipografias utilizadas foram:

- **Cor primária**:
  - Principal: #32508F
  - Clara: #C2E0F6
- **Cor secundária**:
  - Principal: #8F7232

- **Tipografia**: Montserrat, sans-serif

Adotou-se uma abordagem responsiva que adapta layouts e componentes a diferentes tamanhos de tela, garantindo usabilidade desde estações de trabalho industriais até dispositivos móveis para monitoramento remoto. Estruturas em grid fluido e pontos de quebra estratégicos mantêm a consistência visual enquanto otimizam o espaço disponível.

Grande atenção foi dedicada ao feedback visual durante interações, com estados interativos claramente definidos, animações funcionais que guiam a atenção, e confirmações visuais imediatas que complementam o sistema de notificações. Esta redundância informativa previne incertezas operacionais, aspecto crucial em ambientes industriais.

2.2.6 Estrutura de Navegação

A navegação do sistema organiza-se em quatro áreas principais acessíveis via menu lateral: Estações (ponto focal operacional), Usuários e Configurações (áreas administrativas) e Logs (visualização de registros). Esta estruturação estabelece clara separação entre tarefas cotidianas e administrativas, refletindo diferentes frequências de uso. O fluxo principal da navegação no sistema pode ser visualizado na Figura 14 abaixo.

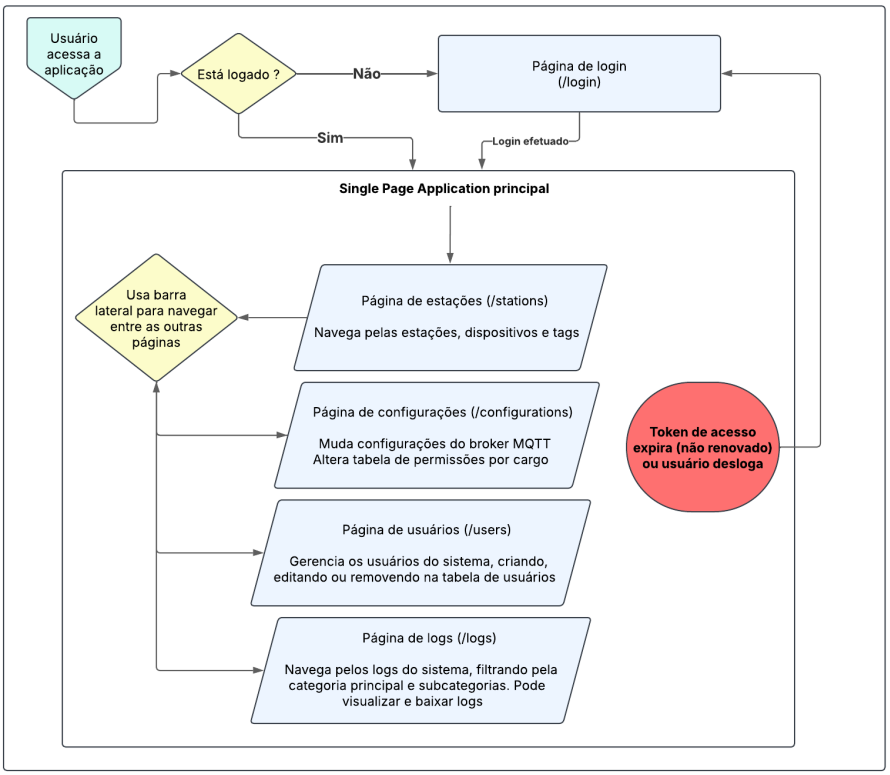


Figura 14 – Fluxograma de navegação na aplicação - Fonte: Elaborado pelo autor

Serão apresentadas as principais páginas da aplicação, conforme ilustrado na Figura 14, nas sub-seções a seguir.

2.2.6.1 Página de login

A página de login na aplicação é a página para qual qualquer usuário não autenticado é redirecionado. Nela o intuito era ter uma interação simples com usuário para login via usuário e senha do sistema. Após efetuado o login, o usuário é redirecionado para a Página "principal"das estações (/stations). A Figura 15 traz uma captura de tela da página.

A parte do login foi pensada para que o usuário possa ocultar/desocultar a senha que está sendo digitada.

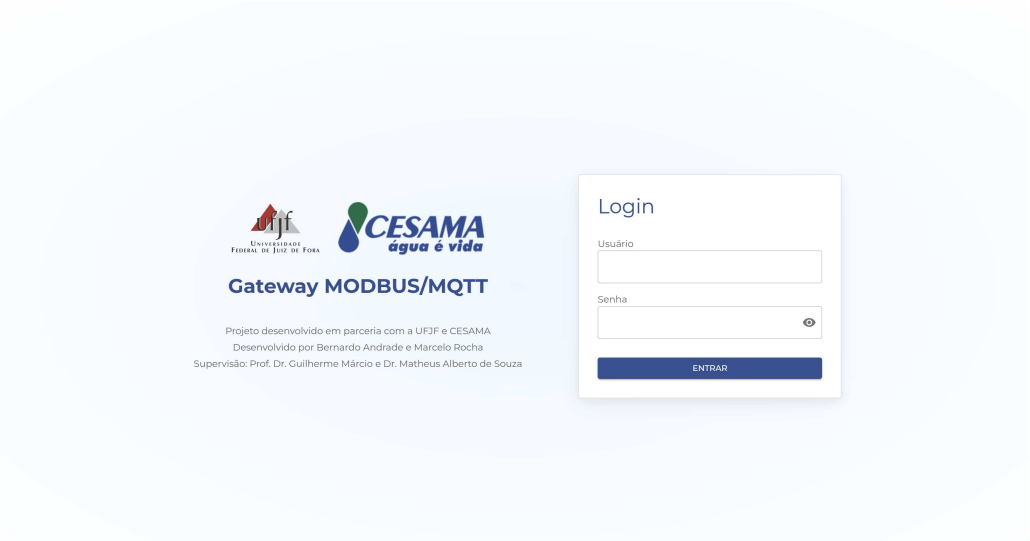


Figura 15 – Página de login da aplicação - Fonte: Elaborado pelo autor.

Caso as credenciais sejam inválidas (não cadastradas no banco do sistema), um aviso aparecerá, conforme ilustrado na Figura 16

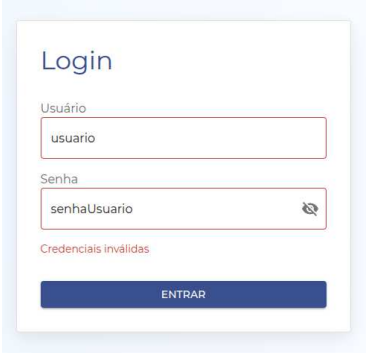


Figura 16 – Exemplo de erro no login - Fonte: Elaborado pelo autor

2.2.6.2 Página de estações

A página de estações é considerada a página principal, pois nela se agrupam todas as informações das estações, dispositivos e tags cadastrados nos sistema.

Agora, é possível também observar um padrão (após usuário cadastrado) no qual a barra lateral existe para navegação do usuário entre as diversas páginas da aplicação e

logout, e a parte "principal"da página, que concentra de fato as informações principais daquela página. Ao navegar de página para página, o que mudará será apenas o conteúdo "principal". Os exemplos irão esclarecer essa ideia.

A Figura 17 traz uma captura de tela da página de estações, em que pode-se observar uma tabela principal das Estações, a barra lateral de navegação e um footer com informações básicas do projeto.

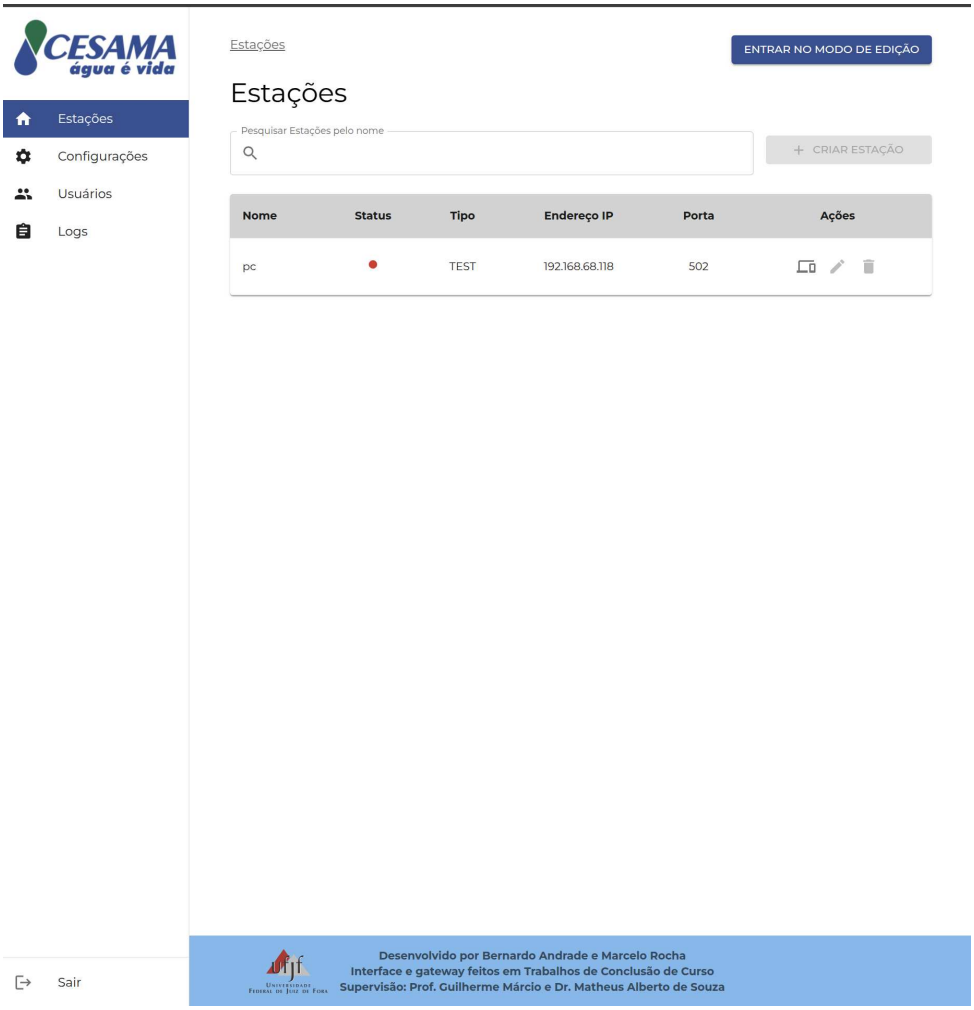
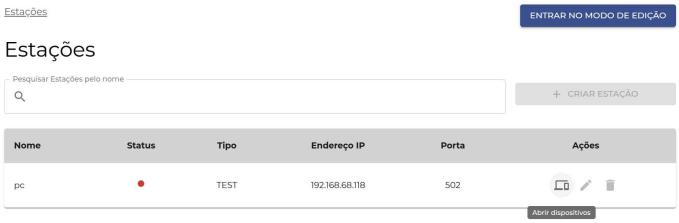
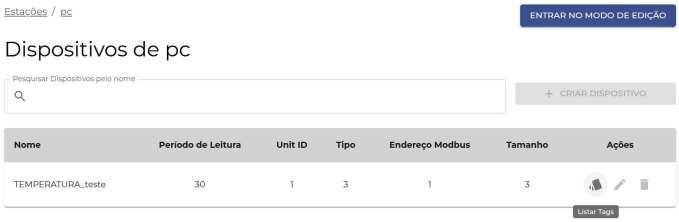


Figura 17 – Exemplo da página de estações - Fonte: Elaborado pelo autor

Dentro da seção de Estações, implementou-se navegação multinível com sistema de drill-down progressivo, onde o usuário navega da lista de estações para dispositivos associados e finalmente para tags específicas. Breadcrumbs interativos e integração com o histórico do navegador facilitam a orientação e o retorno a contextos anteriores. A Figura 18 demonstra a sequência da navegação multinível, visualizando os dispotivos da estação "pc", depois as tags do dispositivo "TEMPERATURA_teste". Na parte superior, é possível clicar nos Breadcrumbs para voltar a qualquer momento para alguma das páginas anteriores. Ademais, na tabela existem as *Ações* que permitam que o usuário interaja com aquela linha da tabela, podendo expandir o item para ver os dispositivos ou tags, ou editar/excluir aquele item da linha (caso esteja em modo de edição).



(a) Página das estações.



(b) Página expandida do dispositivos da estação 'pc'.

(c) Página expandida das tags do dispotivo 'TEMPERATURA_teste'.

Figura 18 – Exemplo de navegação entre itens da estação - Fonte: Elaborado pelo autor.

Para operações complexas, adotou-se navegação modal através de diálogos de edição e fluxos de confirmação para operações críticas. Esta abordagem isola temporariamente o contexto durante edições, mantendo o estado da visualização principal intacto e garantindo retorno preciso ao concluir a operação.

2.2.6.3 Página de configurações

A página de configurações permite ao usuário configurar as credenciais do broker MQTT e as permissões dos cargos no sistema. Dessa forma, o administrador tem controle total sobre a aplicação e o gateway. A Figura 19 traz uma captura de tela da página.

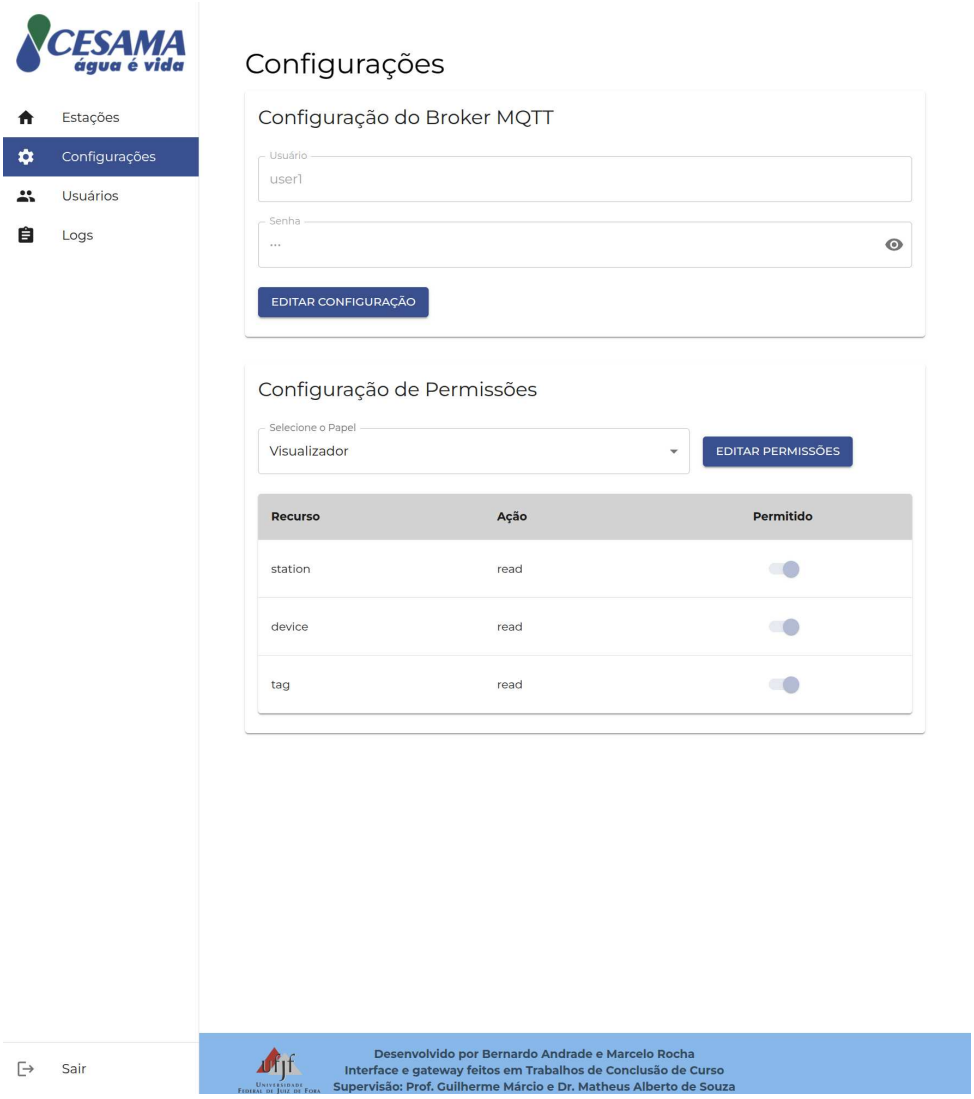


Figura 19 – Exemplo da página de configurações - Fonte: Elaborado pelo autor

2.2.6.4 Página de usuários

A página de usuários, permite ao administrador configurar os usuários do sistema. Seja cadastrando novos usuários, editando os existentes ou removendo-os. A Figura 20 traz uma captura de tela da página.

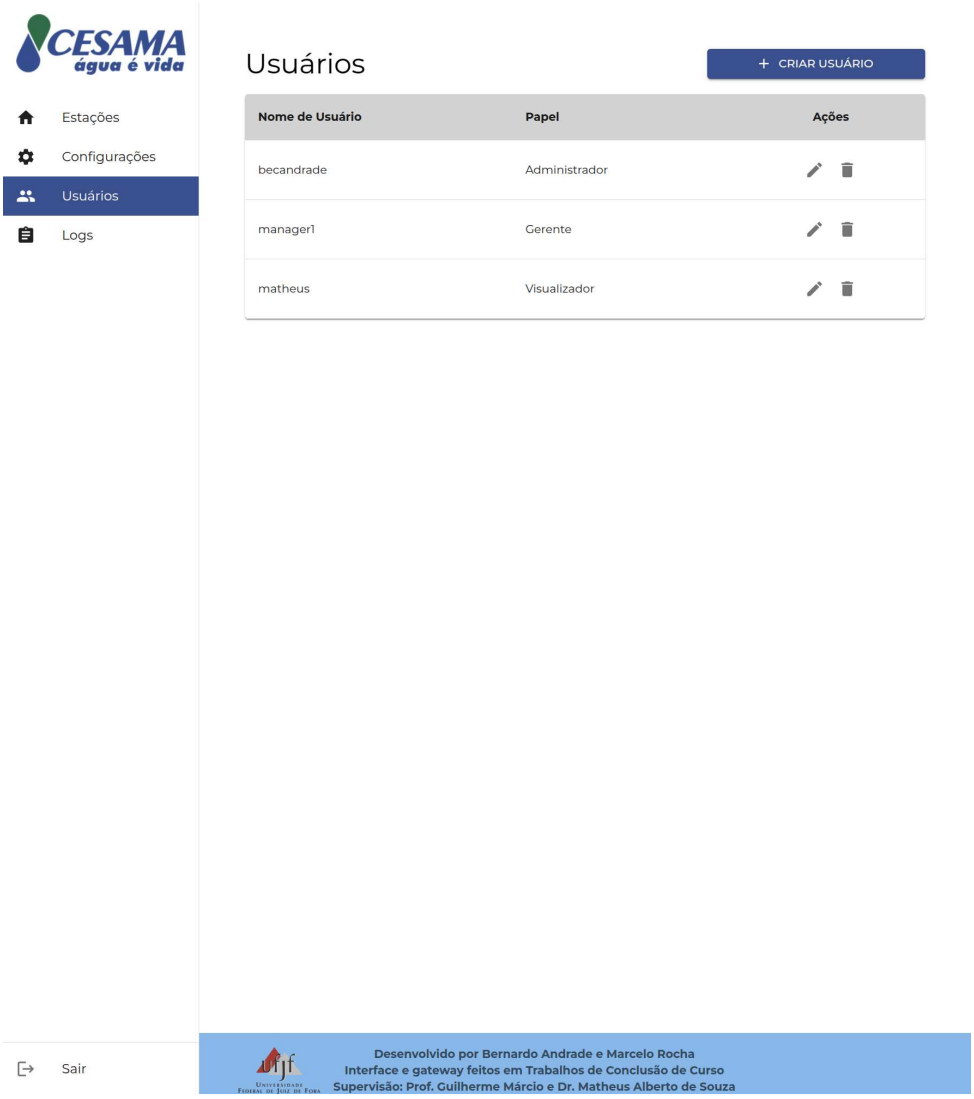


Figura 20 – Exemplo da página de usuários - Fonte: Elaborado pelo autor.

2.2.6.5 Página de logs

A página de logs, permite ao usuário visualizar/baixar logs de diferentes categorias da aplicação, desde logs de alterações de cadastros simples como novas estações cadastradas ou alteradas, até registros de operações do Gateway Modbus/MQTT, verificando se as publicações, leituras e conexões estão funcionando ou não. A Figura 21 traz uma captura de tela da página.

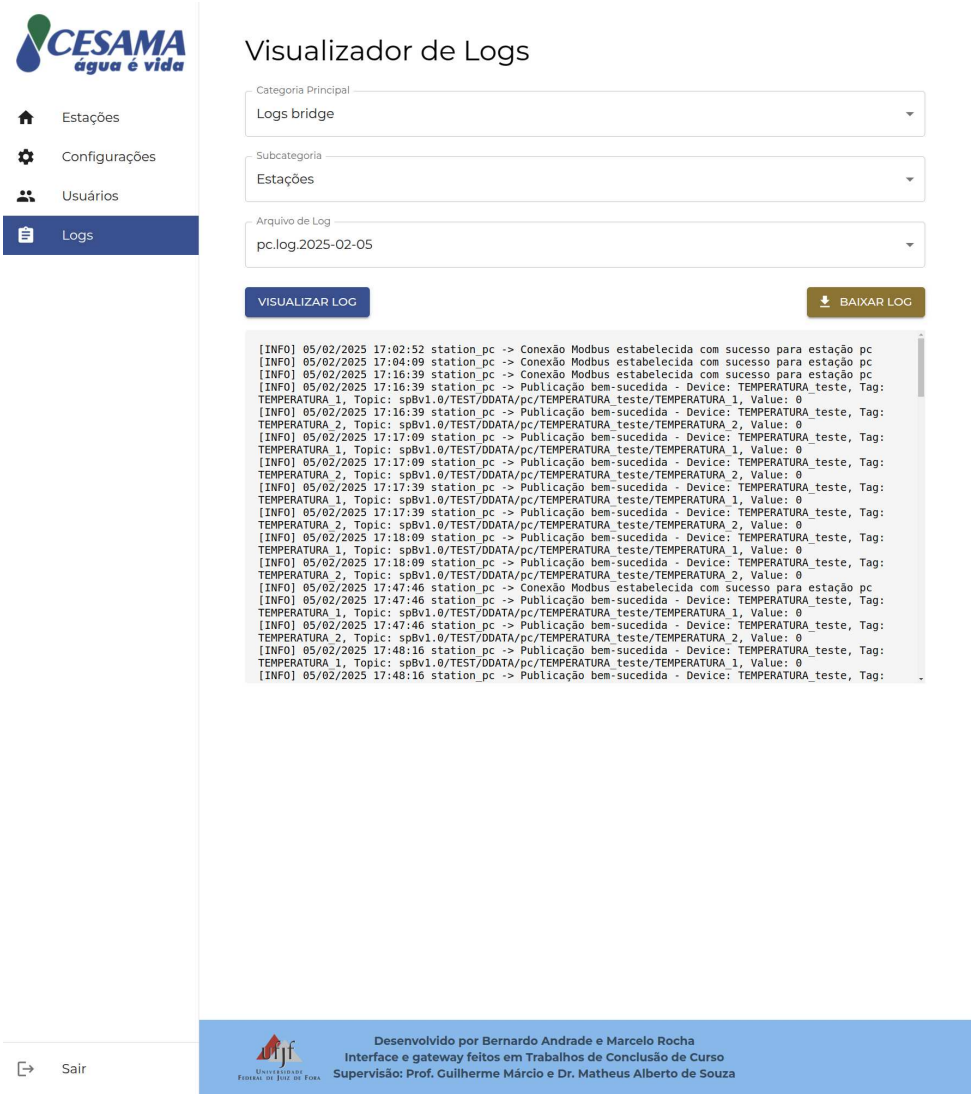


Figura 21 – Exemplo da página de logs - Fonte: Elaborado pelo autor.

2.2.7 Comunicação com o Backend

A comunicação com o backend é realizada através de API RESTful utilizando o cliente Axios, com instância centralizada que padroniza cabeçalhos, transformações de dados e tratamento de erros. As requisições são organizadas em serviços específicos para cada domínio da aplicação, com validação em duas etapas para operações críticas.

Implementou-se tratamento abrangente de erros, com mensagens específicas para falhas previsíveis e sistema de tratamento de exceções em múltiplas camadas para erros inesperados. Durante operações assíncronas, indicadores de progresso e estados de desabilitação de controles previnem ações duplicadas e fornecem feedback claro.

A arquitetura de comunicação foi otimizada para responsividade em condições variáveis de rede, com técnicas como agrupamento de requisições, cache local e carregamento progressivo. Para monitoramento em tempo real, implementou-se polling inteligente que ajusta a frequência de atualização conforme atividade do usuário e disponibilidade do

servidor.

2.2.8 Otimizações de Performance

A performance da interface foi otimizada através de técnicas como memoização de componentes e funções utilizando useCallback e useMemo, especialmente em listas de estações, dispositivos e tags. O gerenciamento criterioso do ciclo de vida com implementação disciplinada do useEffect e carregamento sob demanda de dados hierárquicos contribui significativamente para a responsividade.

Adotou-se carregamento preguiçoso (lazy loading) para rotas menos frequentes e avaliação criteriosa de dependências externas, reduzindo o tamanho do pacote inicial e acelerando a inicialização da aplicação. A ferramenta Vite proporciona divisão de código e tree shaking adicionais durante o processo de build.

Para contextos industriais específicos, implementou-se frequência adaptativa de atualização baseada na visibilidade da aplicação, virtualização para listas extensas, e mecanismos de estabilidade como retry automático e sincronização periódica. Estas otimizações resultam em interface responsiva e confiável, mesmo em condições operacionais desafiadoras.

2.2.9 Tratamento de Erros e Feedback ao Usuário

O sistema implementa uma abordagem multicamada para tratamento de erros, combinando validação preventiva nos formulários com captura e apresentação adequada de exceções durante operações. Validações em tempo real nos campos de entrada fornecem feedback imediato sobre problemas potenciais, como valores fora das faixas permitidas para parâmetros MODBUS, enquanto validações mais complexas são executadas antes da submissão do formulário. Na Figura 22 é demonstrada a validação do campo de endereço IP no formulário de criação/edição de uma estação.

Figura 22 – Exemplo de validação do campo de Endereço IP - Fonte: Elaborado pelo autor

Erros de comunicação com o backend ou falhas operacionais são capturados e apresentados através de um sistema centralizado de notificações baseado no componente

Snackbar do Material-UI. As mensagens são categorizadas por severidade (informação, sucesso, alerta ou erro) com codificação visual por cores e ícones, facilitando a compreensão rápida da natureza do problema. Para operações críticas como exclusão de estações, diálogos de confirmação explícita previnem ações irreversíveis acidentais.

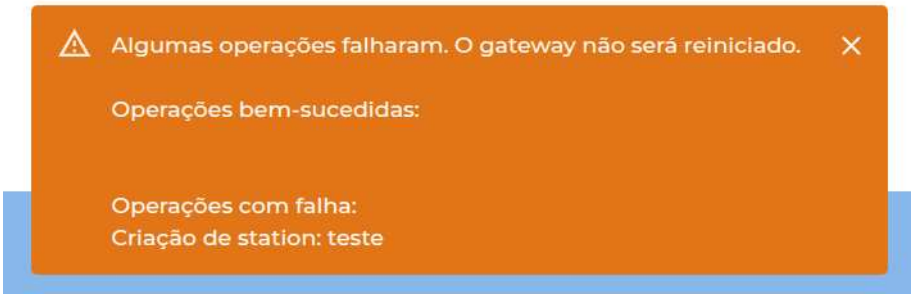


Figura 23 – Snackbar apresentando falha na criação de estação - Fonte: Elaborado pelo autor.

O feedback positivo também recebe atenção especial, com confirmações visuais para operações bem-sucedidas e indicadores de progresso para ações de longa duração como reinicialização do gateway. Esta estratégia equilibrada de feedback negativo e positivo reduz a frustração do usuário, aumenta a confiança no sistema e diminui a curva de aprendizado, aspectos particularmente importantes em ambientes industriais onde a eficiência operacional é crítica.

Figura 24 – Snackbar apresentando sucesso nas alterações - Fonte: Elaborado pelo autor

2.2.10 Considerações sobre Experiência do Usuário

A experiência do usuário no sistema Gateway MODBUS/MQTT foi projetada considerando o contexto operacional industrial, onde clareza, eficiência e consistência sobrepõem-se a elementos estéticos. Os fluxos de trabalho foram estruturados para minimizar o número de interações necessárias para tarefas frequentes, como verificação de status de estações, enquanto operações complexas são subdivididas em etapas lógicas com estado preservado entre transições.

A consistência visual e comportamental foi priorizada em toda a aplicação, com padrões de interação uniformes para operações similares e terminologia técnica precisa. Esta abordagem reduz a carga cognitiva dos operadores, especialmente importante em ambientes industriais onde o sistema pode ser utilizado por profissionais com diferentes níveis de familiaridade tecnológica e em condições de atenção dividida com outros equipamentos.

Aspectos de acessibilidade foram considerados através da implementação de contraste adequado, tamanhos de fonte ajustáveis e suporte a navegação por teclado. O

feedback imediato para ações do usuário, além de sua função informativa, também contribui para a percepção de responsividade do sistema, criando uma experiência que inspira confiança mesmo quando operações de backend possuem latência inerente. Estas considerações resultam em uma interface que, embora tecnicamente sofisticada, apresenta-se ao usuário como uma ferramenta intuitiva e confiável.

2.3 Infraestrutura de TI

A infraestrutura de TI do sistema Gateway MODBUS/MQTT foi projetada visando facilidade de implantação, portabilidade e manutenção. A adoção de tecnologias de conteinerização, especificamente o Docker, permitiu encapsular todos os componentes do sistema em ambientes isolados e reproduzíveis, simplificando significativamente o processo de desenvolvimento e implantação.

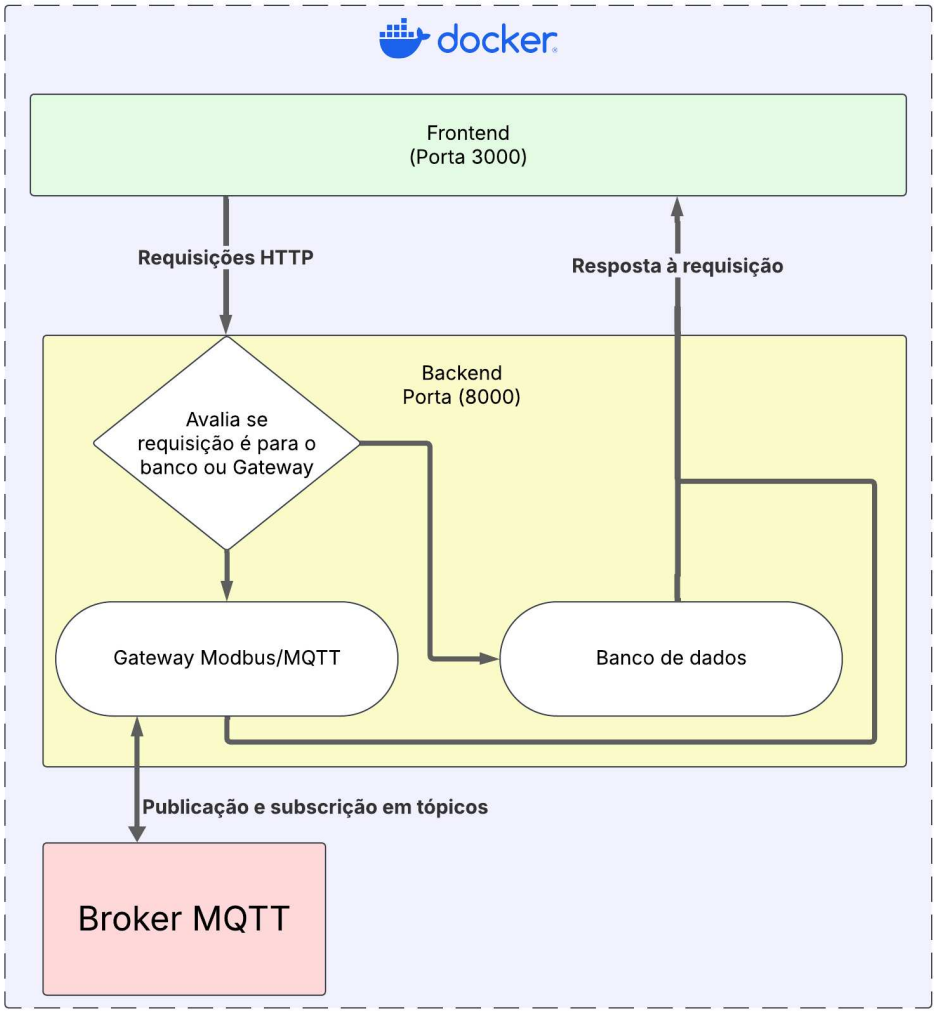


Figura 25 – Visão geral da infraestrutura baseada em Docker - Fonte: Elaborado pelo autor.

2.3.1 Conteinerização com Docker

O Docker foi escolhido como plataforma de conteinerização por proporcionar ambientes leves, portáteis e consistentes para execução dos diferentes componentes do sistema [36]. Cada serviço da aplicação é executado em um contêiner independente, permitindo isolamento de recursos e dependências específicas, além de facilitar a escalabilidade horizontal quando necessário.

A conteinerização traz benefícios substanciais para um sistema industrial como o Gateway MODBUS/MQTT, incluindo:

- **Consistência entre ambientes**: Elimina o tradicional problema "funciona na minha máquina", garantindo comportamento idêntico nos ambientes de desenvolvimento, teste e produção.
- **Isolamento de dependências**: Cada componente opera com suas próprias bibliotecas e dependências, evitando conflitos e simplificando atualizações.
- **Otimização de recursos**: Os contêineres compartilham o kernel do sistema operacional, resultando em sobrecarga significativamente menor comparada a máquinas virtuais tradicionais.

2.3.2 Arquitetura de Contêineres

A arquitetura do sistema é composta por três contêineres principais, cada um com responsabilidades específicas:

2.3.2.1 Frontend

O contêiner do frontend encapsula a aplicação React e utiliza uma abordagem multi-estágio no Dockerfile para otimizar o tamanho e a performance. No primeiro estágio, a aplicação é construída em um ambiente Node.js:

```
FROM node:20-alpine as builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
```

No segundo estágio, apenas os artefatos de build são transferidos para um servidor Nginx leve, responsável por servir a aplicação estática:

```
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

A configuração personalizada do Nginx implementa redirecionamento apropriado para aplicações de página única (SPA), garantindo que todas as rotas sejam gerenciadas corretamente pelo React Router.

2.3.2.2 Backend

O contêiner do backend encapsula a API FastAPI desenvolvida em Python e utiliza Poetry para gerenciamento de dependências:

```
FROM python:3.11-slim
RUN pip install "poetry==1.8.2"
COPY pyproject.toml poetry.lock ./
RUN poetry config virtualenvs.create false
RUN poetry install --without dev --no-interaction
```

Esta configuração garante instalação consistente das dependências e otimização para ambiente de produção, excluindo pacotes de desenvolvimento. O servidor Uvicorn é utilizado para execução da aplicação FastAPI, oferecendo alto desempenho para aplicações assíncronas.

2.3.2.3 MQTT Broker

O terceiro contêiner implementa o broker MQTT utilizando a imagem oficial do Eclipse Mosquitto, um broker leve e de código aberto que implementa o protocolo MQTT:

```
mosquitto:
  image: eclipse-mosquitto
  volumes:
    - ./Mosquitto/conf:/mosquitto/config
    - ./Mosquitto/data:/mosquitto/data
    - ./Mosquitto/log:/mosquitto/log
```

A configuração é montada a partir do host, permitindo personalização das políticas de acesso, autenticação e persistência conforme os requisitos específicos da CESAMA.

2.3.3 Orquestração com Docker Compose

O Docker Compose é utilizado para definir e gerenciar a execução dos múltiplos contêineres que compõem a aplicação [37]. Esta ferramenta simplifica significativamente o

processo de desenvolvimento e implantação, permitindo iniciar toda a infraestrutura com um único comando.

O arquivo `docker-compose.yml` define a configuração de cada serviço, suas dependências, mapeamentos de porta e volumes:

```
services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - ./backend:/app
      - ./backend/database.db:/app/database.db
      - ./backend/logs:/app/logs
    environment:
      - DATABASE_URL=sqlite:///database.db
    networks:
      - app-network

  frontend:
    build:
      context: ./frontend/gateway-front
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - backend
    networks:
      - app-network

  mosquitto:
    image: eclipse-mosquitto
    container_name: mosquitto
    volumes:
      - ./Mosquitto/conf:/mosquitto/config
      - ./Mosquitto/data:/mosquitto/data
      - ./Mosquitto/log:/mosquitto/log
```

```
    ports:
      - "1883:1883"
      - "9001:9001"
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

Aspectos importantes desta configuração incluem:

- **Volumes persistentes**: Dados críticos como o banco de dados SQLite e logs são armazenados em volumes mapeados para o host, garantindo persistência mesmo após a recriação dos contêineres.
- **Rede dedicada**: Uma rede bridge isolada (`app-network`) é criada para comunicação entre os contêineres, melhorando a segurança e simplificando a descoberta de serviços.
- **Definição de dependências**: Através da diretiva `depends_on`, garante-se que os contêineres sejam iniciados na ordem correta, com o frontend dependendo do backend.

2.3.4 Considerações sobre Implantação

A infraestrutura baseada em Docker proporciona flexibilidade para implantação em diversos ambientes, desde servidores on-premises até plataformas de nuvem como máquinas virtuais.

Os requisitos de hardware são relativamente modestos devido à eficiência dos contêineres:

- **CPU**: 2+ cores para operação regular
- **Memória**: Mínimo de 2GB RAM, recomendado 4GB
- **Armazenamento**: 20GB+ para sistema, logs e dados

2.3.5 Implantação em Novo Ambiente

A implantação do sistema Gateway MODBUS/MQTT em um novo ambiente é simplificada através do uso de Docker. A seguir, são apresentados os passos necessários para executar a aplicação em uma nova máquina, assumindo que Docker e Docker Compose estejam previamente instalados (instruções de instalação em [38]).

2.3.5.1 Requisitos Preliminares

Antes de iniciar a implantação, deve ser verificado se o ambiente atende aos seguintes requisitos:

- Docker Engine (versão 20.10 ou superior)
- Docker Compose (versão 2.0 ou superior)
- Acesso à internet (para download inicial das imagens)
- Portas 3000 (Frontend), 8000 (Backend) e 1883 (MQTT) disponíveis

2.3.5.2 Procedimento de Implantação

O processo de implantação consiste em cinco etapas simples:

1. **Obtenção do Código-Fonte**:

   O usuário deve clonar o repositório do projeto ou copiar os arquivos para a máquina de destino:

   ```
   git clone https://repositorio.do.projeto/gateway-modbus-mqtt.git
   cd gateway-modbus-mqtt
   ```

2. **Configuração do Ambiente**:

   Criar os diretórios necessários para os volumes persistentes:

   ```
   mkdir -p Mosquitto/conf Mosquitto/data Mosquitto/log
   mkdir -p backend/logs
   ```

   Configurar o broker MQTT criando o arquivo `Mosquitto/conf/mosquitto.conf` com o seguinte conteúdo:

   ```
   persistence true
   persistence_location /mosquitto/data
   log_dest file /mosquitto/log/mosquitto.log

   listener 1883
   ## Authentication ##
   allow_anonymous false
   password_file /mosquitto/config/password.txt
   ```

Criar também o arquivo de senhas para autenticação do broker MQTT:

```
# Crie um arquivo vazio para senhas
touch Mosquitto/conf/password.txt

# Execute o comando dentro do contêiner para adicionar um usuário
# Substitua 'usuario' e 'senha' pelos valores desejados
docker compose exec mosquitto mosquitto_passwd -b
    /mosquitto/config/password.txt usuario senha

# Ou, se o contêiner ainda não estiver em execução, use:
docker run --rm -v $(pwd)/Mosquitto/conf:/mosquitto/config
    eclipse-mosquitto \ mosquitto_passwd -b
    /mosquitto/config/password.txt usuario senha
```

Estas configurações habilitam a persistência de dados, registro de logs e, importante para ambientes de produção, autenticação obrigatória para conexões ao broker MQTT, aumentando a segurança do sistema.

3. **Construção dos Contêineres**:

   Executar o comando abaixo para construir as imagens Docker definidas no docker-compose:

   ```
   docker compose build
   ```

   Este processo pode levar alguns minutos na primeira execução, pois serão baixadas as imagens base e instaladas todas as dependências.

4. **Inicialização do Sistema**:

   Iniciar todos os serviços com o comando:

   ```
   docker compose up -d
   ```

   A flag `-d` executa os contêineres em modo destacado (background).

5. **Verificação da Execução**:

   Confirmar se todos os contêineres estão em execução:

```
docker compose ps
```

Verificar os logs de cada serviço para identificar possíveis problemas:

```
docker compose logs backend
docker compose logs frontend
docker compose logs mosquitto
```

Após concluir estes passos, o sistema estará acessível através dos seguintes endereços:

- Frontend: `http://localhost:3000` ou `http://[IP-DA-MÁQUINA]:3000`
- Backend API: `http://localhost:8000` ou `http://[IP-DA-MÁQUINA]:8000`
- MQTT Broker: `mqtt://localhost:1883` ou `mqtt://[IP-DA-MÁQUINA]:1883`

2.3.5.3 Operações Comuns de Manutenção

Para facilitar a administração do sistema em produção, a seguir são listados os comandos mais comuns para operações de manutenção:

- **Parar todos os serviços**:

```
docker compose down
```

- **Reiniciar um serviço específico**:

```
docker compose restart backend
```

- **Visualizar logs em tempo real**:

```
docker compose logs -f
```

- **Atualizar após alterações no código**:

```
docker compose build frontend
docker compose up -d --no-deps frontend
```

Esta configuração facilita tanto ambientes de desenvolvimento quanto de produção, permitindo que equipes de TI com diferentes níveis de experiência possam gerenciar o sistema com facilidade.

Esta arquitetura de infraestrutura proporciona um equilíbrio entre robustez operacional e facilidade de manutenção, permitindo que a equipe técnica da CESAMA gerencie o sistema com recursos mínimos de TI, enquanto mantém a possibilidade de evolução e escala conforme necessário.

3 Resultados

Este capítulo apresenta os resultados obtidos no desenvolvimento e implementação do sistema Gateway MODBUS/MQTT, demonstrando o funcionamento dos diferentes componentes da solução e sua integração, dentro de um ambiente simulado.

3.1 Inicialização do sistema

Nesta seção, será demonstrado como o sistema foi inicializado e quais os resultados obtidos nessa inicialização, bem como os passos iniciais necessários para acessar o aplicativo web.

Seguindo os passos presentes na Seção *2.3.5.2*, o Docker é utilizado para inicializar o sistema. Ao executar o comando *docker compose up* o sistema é executado com os logs no terminal ativos, para poder visualizar e mostrar os resultados.

Serão analisadas as inicializações separadamente, nas suas respectivas sub-seções a seguir.

3.1.1 Análise da inicialização do backend

Ao inicializar o sistema, o backend fica disponível na porta 8000 da máquina, esperando por requisições. O Gateway também é inicializado junto com a aplicação, mas em uma thread separada.

As Figuras 26 e 27 mostram os logs, no terminal da máquina, de inicialização do backend.

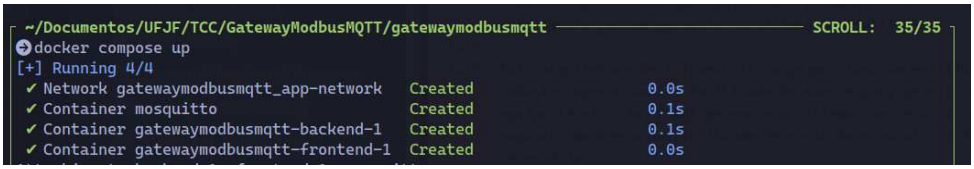


Figura 26 – Inicialização do container no terminal - Fonte: Elaborado pelo autor.

Figura 27 – Inicialização do backend no terminal - Fonte: Elaborado pelo autor.

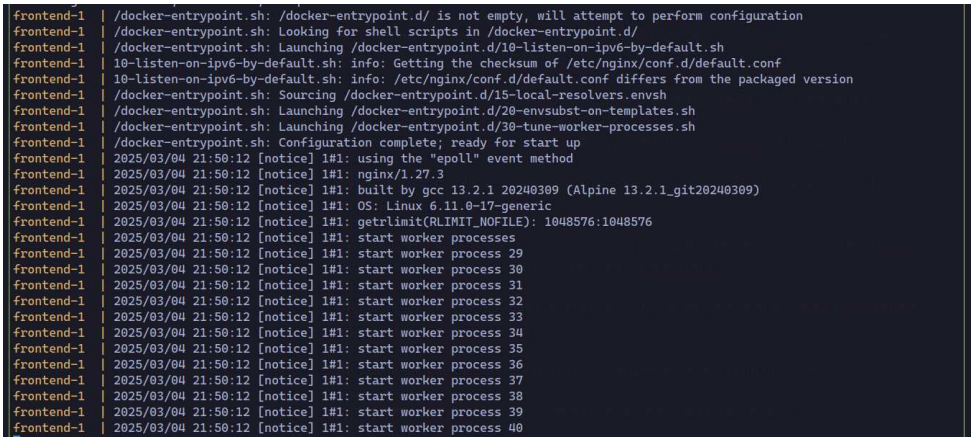


Figura 28 – Inicialização do frontend no terminal - Fonte: Elaborado pelo autor.

3.1.2 Análise da inicialização do frontend

Ao inicializar o sistema, o frontend fica disponível na porta 3000 da máquina, servindo a aplicação frontend.

As Figuras 26 e 28 mostram os logs, no terminal da máquina, de inicialização do frontend.

Ao tentar acessar o sistema na máquina via *localhost* pela url *https://localhost:3000* a aplicação já redireciona para a página de login. Os logs no terminal são mostrados na Figura 29.

Figura 29 – Exemplo de requisição de página pelo frontend no terminal - Fonte: Elaborado pelo autor.

3.2 Resultados Simulados

Para poder desenvolver e demonstrar o funcionamento do projeto como um todo, foi feito o uso de duas máquinas distintas. Uma foi responsável por rodar o container do aplicativo fullstack de gerenciamento do sistema. A segunda máquina utilizou a versão de demonstração do software MODBUS Slave, da empresa *modbus tools,* para simular um dispositivo escravo sendo acessado remotamente, como se fossem as estações da CESAMA. A Figura 30 ilustra a arquitetura da simulação, onde o Laptop executa o simulador MODBUS Slave e o PC executa o container da aplicação.

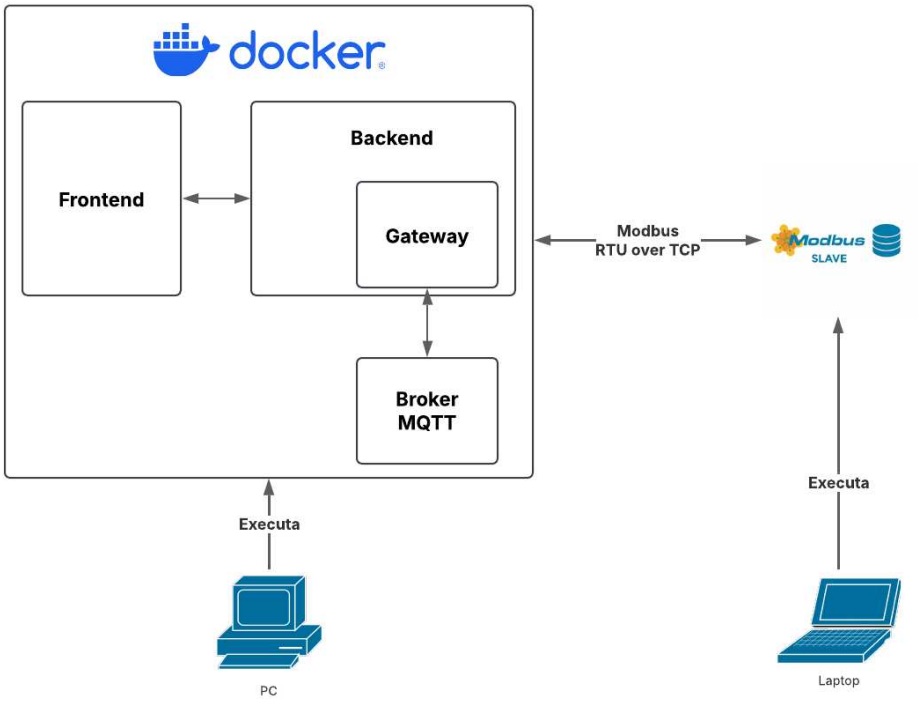


Figura 30 – Diagrama de estrutura da simulação - Fonte: Elaborado pelo autor.

3.2.1 Configuração do simulador

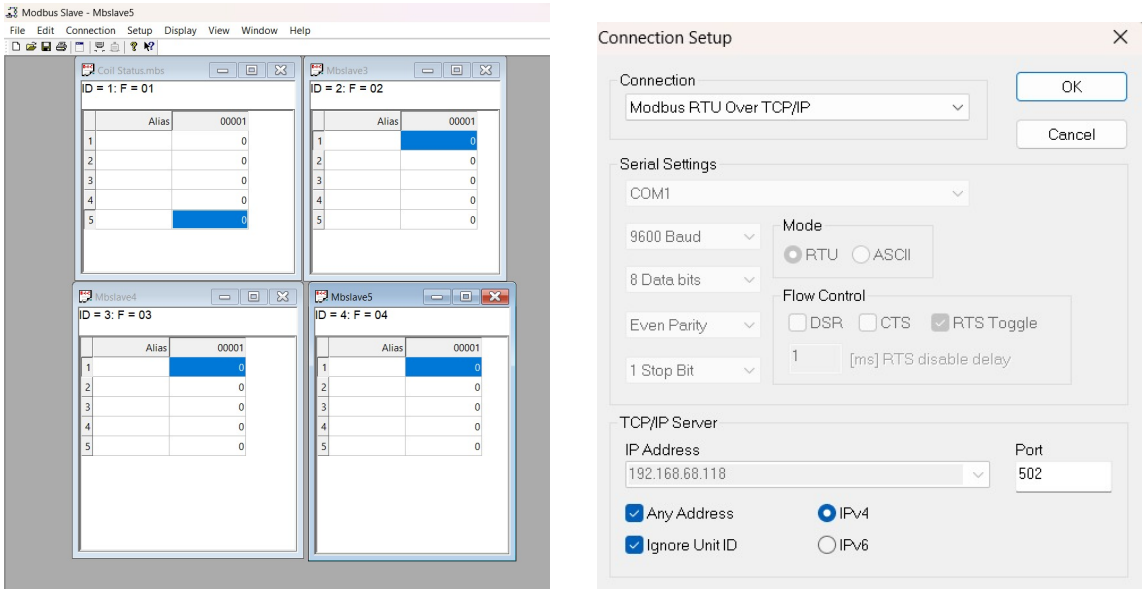
No simulador MODBUS Slave, a configuração é feita como se ele fosse uma estação. Dentro dele, são configurados diferentes dispositivos com diferentes tags e valores nas tags para serem lidos.

Para simplificar a demonstração, foram utilizados apenas 4 dispositivos com 5 tags cada. Cada dispositivo é de um tipo de dado, sendo:

- **Coil Status**: Tipo 1, Coil Status;
- **Mbslave3**: Tipo 2, Input Status;
- **Mbslave4** : Tipo 3, Holding Register;
- **Mbslave5** : Tipo 4, Input Register;

Todas as 5 tags de cada dispositivo foram mantidas com o valor 0.

A Figura 31 mostra a configuração feita dentro do simulador, junto com as nomenclaturas de cada dispositivo e a configuração de conexão do simulador.



(a) Configuração dos escravos simulados.

(b) Configuração da conexão Modbus simulada.

Figura 31 – Configurações da simulação - Fonte: Elaborado pelo autor.

3.2.2 Configuração do aplicativo web com os dados do simulador

Feitas as configurações no simulador no Laptop e conectado na rede, é necessário cadastrar a estação no aplicativo executado no PC, bem como os dispositivos e tags de cada um. A lista de cadastro segue a seguinte hierarquia:

- **Estação**: teste_simulado com IP do Laptop e porta 502
  - **Dispositivos:**
  - Coil Status
    - **Tags:**
    - tag1
    - tag2
    - tag3
    - tag4
    - tag5
  - Mbslave3
    - **Tags:**
    - tag1
    - tag2
    - tag3

    - tag4
    - tag5
- Mbslave4
    - **Tags:**
    - tag1
    - tag2
    - tag3
    - tag4
    - tag5
- Mbslave5
    - **Tags:**
    - tag1
    - tag2
    - tag3
    - tag4
    - tag5

Para configurar o sistema, primeiro é necessário fazer login. Para isso, foi preparado previamente um usuário no banco de dados com papel de "Administrador"do sistema. Existe também o exemplo de um usuário "Gerente"do sistema que terá algumas restrições.

3.2.2.1 Login

Após preenchimento das credenciais e confirmação o backend recebe a requisição de login e retorna com sucesso, além de já buscar pelas estações e status das mesmas para exibir na página, conforme observado na Figura 32.

Figura 32 – Logs de requisições ao backend no terminal - Fonte: Elaborado pelo autor.

3.2.2.2 Cadastro de estação

Efetuado o login, a aplicação entra na página de estações. Nela, deve-se realizar o cadastro de uma estação de teste, com as mesmas configurações efetuadas no simulador (conforme hierarquia na seção **3.2.2**). Para isso, basta entrar no modo de edição e ir em "Criar estação".

A Figura 33 exemplifica o formulário de cadastro.

Figura 33 – Formulário de cadastro de estação - Fonte: Elaborado pelo autor

O registro de criação pode ser verificado pelo log das estações, conforme mostrado na Figura 34.

Figura 34 – Log de cadastro da estação - Fonte: Elaborado pelo autor

3.2.2.3 Cadastro de dispositivo

Após cadastrar a estação, é feito o cadastro de um dispositivo dela. Basta abrir os dispositivos da estação criada e ir em "Criar dispositivo".

A Figura 35 exemplifica o formulário de cadastro.

O registro de criação pode ser verificado pelo log dos dispositivos, conforme mostrado na Figura 36.

Figura 35 – Formulário de cadastro de dispositivo - Fonte: Elaborado pelo autor

Figura 36 – Log de cadastro de dispositivo - Fonte: Elaborado pelo autor

3.2.2.4 Cadastro de tag

Após cadastrar o dispositivo, é feito o cadastro de uma tag dele. Basta abrir as tags do dispositivo criado e ir em "Criar tag".

A Figura 37 exemplifica o formulário de cadastro.

O registro de criação pode ser verificado pelo log das tags, conforme mostrado na Figura 38.

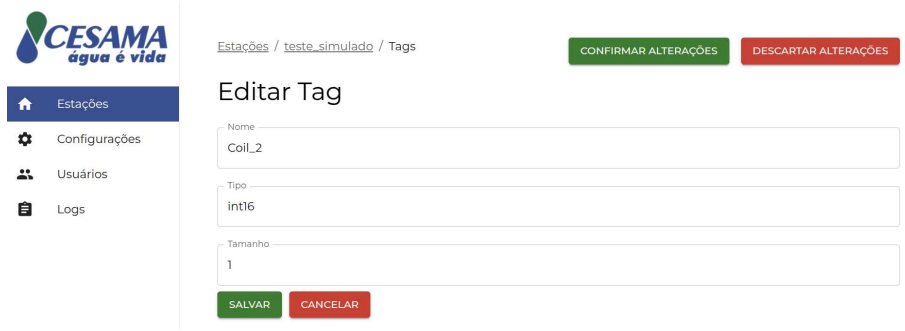


Figura 37 – Formulário de cadastro de tag - Fonte: Elaborado pelo autor

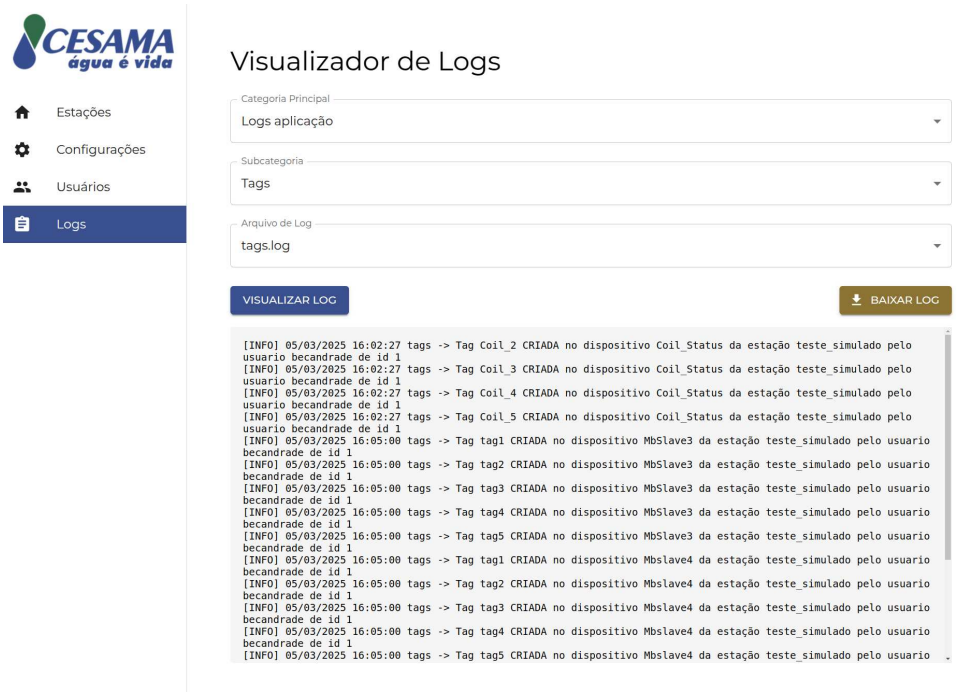


Figura 38 – Log de cadastro de tag - Fonte: Elaborado pelo autor

3.2.3 Permissionamento de páginas para gerente e visualizador

Conforme mencionado anteriormente, o acesso às páginas de Configuração e Usuários é restrito apenas para usuários com papel de Administrador no sistema.

Ao fazer login com um usuário com papel de Gerente, a barra de navegação lateral mostra apenas as páginas Estações e Logs, conforme mostra a Figura 39.

O comportamento se estende para usuários que tiverem o papel Visualizador.

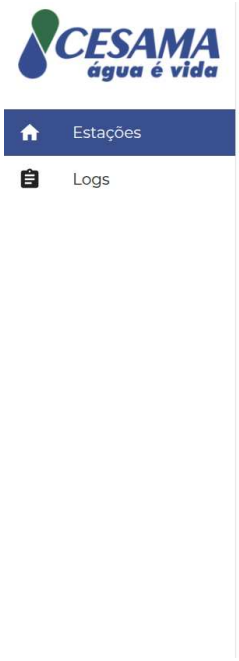


Figura 39 – Exemplo da barra lateral em visualização do Gerente - Fonte: Elaborado pelo autor

3.2.4 Monitoramento da estação e funcionamento do Gateway encapsulado

Com os dados cadastrados, monitora-se o status da estação cadastrada a partir da tela de Estações, na coluna *Status*.

Quando o simulador está conectado e funcionando, um indicador verde mostra que a conexão está adequada, conforme ilustrado na Figura 40 .

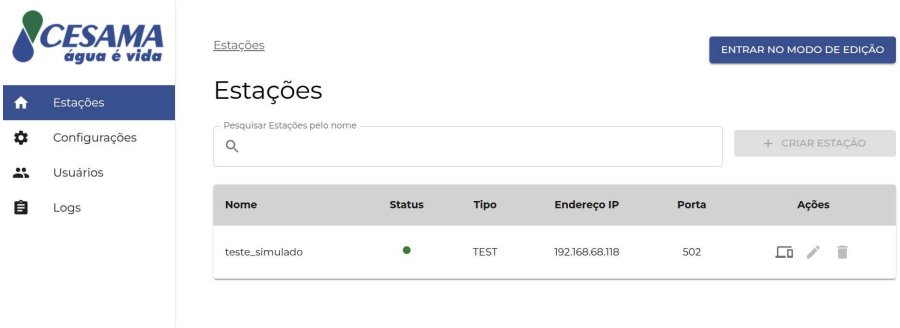


Figura 40 – Exemplo de estação conectada - Fonte: Elaborado pelo Autor

Já quando o simulador é desconectado, o indicador muda para vermelho, mostrando que a conexão com aquela estação foi perdida, conforme ilustrado na Figura 41.

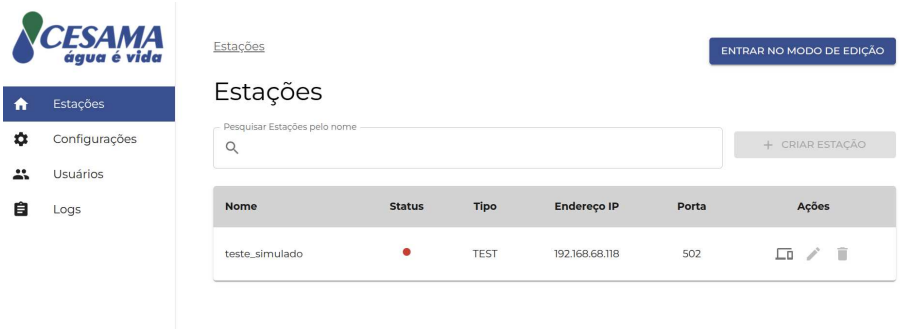


Figura 41 – Exemplo de estação desconectada - Fonte: Elaborado pelo Autor

Ademais, é possível verificar o log da estação, na categoria do bridge (Gateway) para verificar logs de publicação de dados no broker MQTT e status da conexão Modbus com a estação. Isto pode ser feito a partir da página de Logs, conforme ilustrado na Figura 42, a partir do registro no tempo `"05/03/2025 17:35:12"` é possível visualizar a conexão sendo estabelecida com sucesso e as publicações de dados no broker MQTT sendo feitas com sucesso.

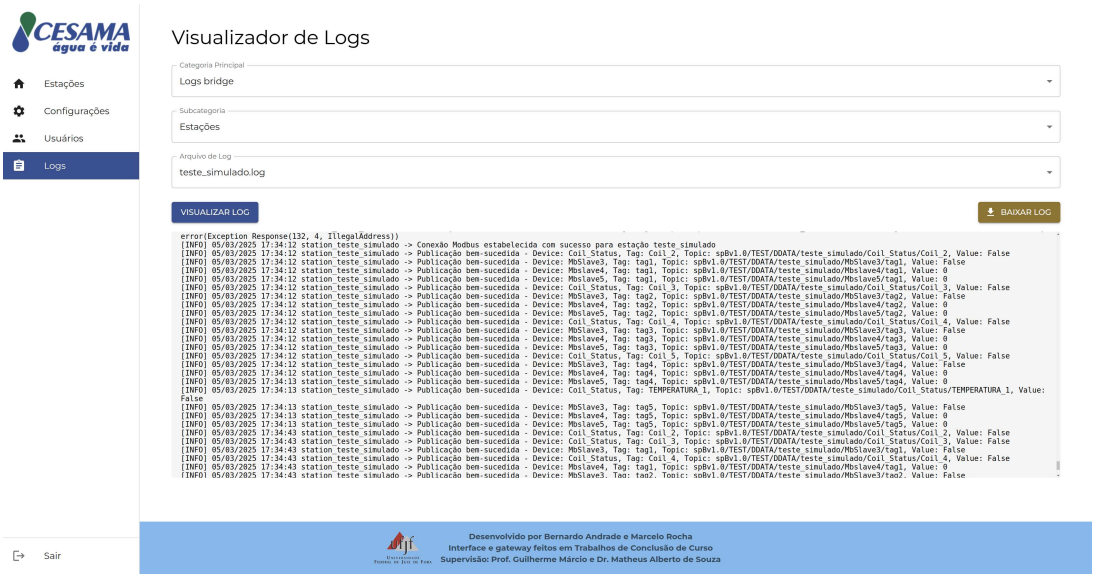


Figura 42 – Log de funcionamento do gateway - Fonte: Elaborado pelo autor

4 Conclusões

O presente trabalho propôs o desenvolvimento de uma aplicação web fullstack para gerenciamento e encapsulamento de um gateway MQTT/Modbus para a CESAMA. O sistema foi implementado utilizando Python com FastAPI no backend e React com Vite no frontend, estabelecendo uma arquitetura moderna e eficiente. Ademais, foi possível encapsular de forma integrada o gateway desenvolvido em [7].

4.1 Objetivos alcançados

Ao término do desenvolvimento e simulação realizada, é possível concluir que o aplicativo atendeu plenamente às demandas e expectativas, proporcionando uma interface intuitiva para o gerenciamento dos dispositivos conectados ao gateway e facilitando a integração entre os protocolos MQTT e Modbus. As funcionalidades implementadas permitiram simplificar o processo de configuração, monitoramento e manutenção do sistema.

A utilização do FastAPI no backend mostrou-se uma escolha acertada, proporcionando uma API de alta performance e fácil manutenção, enquanto o React com Vite no frontend, em conjunto com a biblioteca Material UI para componentes, garantiu uma experiência de usuário consistente, acessível e esteticamente refinada. A integração do Material UI permitiu a implementação de uma interface gráfica profissional com componentes reutilizáveis, acelerando o desenvolvimento e melhorando a usabilidade do sistema. A arquitetura adotada demonstrou-se escalável e modular, permitindo futuras expansões e adaptações conforme as necessidades da CESAMA evoluam.

4.2 Sugestão para Estudos Futuros

Como trabalhos futuros, sugere-se a implementação de recursos adicionais de análise de dados e visualização em tempo real, além da possibilidade de integração com outros protocolos industriais expandindo a versatilidade do sistema.

Por fim, conclui-se que o desenvolvimento deste aplicativo contribuiu significativamente para a gestão e modernização da planta industrial da CESAMA, representando um avanço tecnológico importante e adequado com os padrões atuais do mercado.

REFERÊNCIAS

1 VNODE. *vNode use Cases v2.5*. 2022. Disponível em: <https://vnodeautomation.com/wp-content/uploads/vNode-v121-UseCases-v2.5.pdf>. Acesso em: 15/02/2025.

2 INC, E. T. *Neuron Documentation*. 2024. Disponível em: <https://docs.emqx.com/en/neuronex/latest/>. Acesso em: 15/02/2025.

3 USTYMENKO, V. *A/B testing on single-page applications with Adobe Target*. 2025. Disponível em: <https://business.adobe.com/blog/how-to/your-guide-to-successfully-implementing-a-b-testing-in-a-single-page-application>. Acesso em: 01/03/2025.

4 ARMSTRONG, N. *Passing Data From Child to Parent Component in TypeScript React*. 2021. Disponível em: <https://plainenglish.io/blog/passing-data-from-child-to-parent-component-in-typescript-react>. Acesso em: 03/02/2025.

5 MATERIALUI. *Material UI - Overview*. 2025. Disponível em: <https://mui.com/material-ui/getting-started/>. Acesso em: 15/02/2025.

6 REACT. *Passing Data Deeply with Context*. 2025. Disponível em: <https://pt-br.react.dev/learn/passing-data-deeply-with-context#context-an-alternative-to-passing-props>. Acesso em: 01/03/2025.

7 REIMAO, M. R. Gateway mqtt para sistemas de automação industrial. *UFJF*, 2024.

8 S.L., V. B. *Meet the new vNode v1.21 Powerful Edge Platform For IIoT*. 2024. Disponível em: <https://vnodeautomation.com>. Acesso em: 15/02/2025.

9 INC, E. T. *NeuronEX - Industrial Edge Data Hub*. 2024. Disponível em: <https://www.emqx.com/en/products/neuronex>. Acesso em: 15/02/2025.

10 INC, E. T. *Neuron Github*. 2024. Disponível em: <https://github.com/emqx/neuron>. Acesso em: 15/02/2025.

11 SOBRINHO, A. M. et al. Conversor embarcado de protocolos modbus/mqtt para rede iot utilizando raspberry pi. *Brazilian Journal of Development*, v. 9, n. 6, p. 19327–19337, 2023.

12 SILVA, C.; MUNIZ, F. An iot gateway for modbus and mqtt integration. In: . [S.l.: s.n.], 2019.

13 MATOS, E. D. V. de. Sistema meu tcc: implementação do front-end de uma aplicação web para controle de tccs utilizando reactjs. *UFSC*, 2023. Disponível em: <https://repositorio.ufsc.br/handle/123456789/248929>.

14 CHEN, C.-Y. et al. Web-based internet of things on environmental and lighting control and monitoring system using node-red, mqtt and modbus communications within embedded linux platform. *Internet of Things*, v. 27, p. 101305, 2024. ISSN 2542-6605. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2542660524002464>.

15 BORDIGNON, G. D. Desenvolvimento de um gateway de protocolo com suporte a modbus e mqtt. *UFSC*, 2024. Disponível em: <https://repositorio.ufsc.br/handle/123456789/256860>.

16 POETRY. *Poetry Documentation*. 2025. Disponível em: <https://python-poetry.org/docs/>. Acesso em: 15/02/2025.

17 FASTAPI. *FastAPI Source Code*. 2025. Disponível em: <https://github.com/fastapi/fastapi>. Acesso em: 15/02/2025.

18 FASTAPI. *FastAPI Documentation*. 2025. Disponível em: <https://fastapi.tiangolo.com/>. Acesso em: 15/02/2025.

19 SQLALCHEMY. *The Python SQL Toolkit and Object Relational Mapper*. 2025. Disponível em: <https://www.sqlalchemy.org/>. Acesso em: 15/02/2025.

20 ALEMBIC. *A database migrations tool written by the author of SQLAlchemy*. 2025. Disponível em: <https://alembic.sqlalchemy.org/>. Acesso em: 15/02/2025.

21 PYDANTIC. *The most widely used data validation library for Python*. 2025. Disponível em: <https://docs.pydantic.dev/latest/>. Acesso em: 15/02/2025.

22 PYJWT. *Python library to encode and decode JSON Web Tokens*. 2025. Disponível em: <https://pyjwt.readthedocs.io/en/stable/>. Acesso em: 15/02/2025.

23 PWDLIB. *Password hasher helper for the modern Python era*. 2025. Disponível em: <https://frankie567.github.io/pwdlib/>. Acesso em: 15/02/2025.

24 DAIQUIRI. *Python logging setup helper*. 2025. Disponível em: <https://daiquiri.readthedocs.io/en/latest/>. Acesso em: 15/02/2025.

25 FASTAPI. *Bigger Applications - Multiple files*. 2025. Disponível em: <https://fastapi.tiangolo.com/tutorial/bigger-applications/#an-example-file-structure>. Acesso em: 15/02/2025.

26 META. *React Reference Overview*. 2025. Disponível em: <https://react.dev/reference/react>. Acesso em: 15/02/2025.

27 META. *Built-in React Hooks*. 2025. Disponível em: <https://react.dev/reference/react/hooks>. Acesso em: 15/02/2025.

28 VITE. *Getting Started with Vite*. 2025. Disponível em: <https://vite.dev/guide/>. Acesso em: 15/02/2025.

29 MATERIALUI. *Material UI - Theming*. 2025. Disponível em: <https://mui.com/material-ui/customization/theming/?srsltid=AfmBOorta5ywxHPJnFF8LbkSOUfXreQ1t7dZNkoDCbWUNFIXOEjQNqyv>. Acesso em: 15/02/2025.

30 ROUTER, R. *React Router Home*. 2025. Disponível em: <https://reactrouter.com/home>. Acesso em: 15/02/2025.

31 AXIOS. *Axios - Getting Started*. 2025. Disponível em: <https://axios-http.com/docs/intro>. Acesso em: 15/02/2025.

32 REACT. *Creating a context*. 2025. Disponível em: <https://react.dev/reference/react/createContext>. Acesso em: 15/02/2025.

33 TYPES prop. *Runtime type checking for React props and similar objects*. 2025. Disponível em: <https://www.npmjs.com/package/prop-types>. Acesso em: 15/02/2025.

34 SAVER file. *Solution to saving files on the client-side*. 2025. Disponível em: <https://www.npmjs.com/package/file-saver>. Acesso em: 15/02/2025.

35 ESLINT. *Find and fix problems in your JavasScript code*. 2025. Disponível em: <https://eslint.org/docs/latest/>. Acesso em: 15/02/2025.

36 DOCKER. *Docker Docs*. 2025. Disponível em: <https://docs.docker.com/>. Acesso em: 01/03/2025.

37 DOCKER. *Docker Compose*. 2025. Disponível em: <https://docs.docker.com/compose/>. Acesso em: 01/03/2025.

38 DOCKER. *Get Docker*. 2025. Disponível em: <https://docs.docker.com/get-started/get-docker/>. Acesso em: 01/03/2025.