

**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO**



DANIEL PINTO BARROS

**DESENVOLVIMENTO DE UM SISTEMA DE
MONITORAMENTO DE CONSUMO DE ÁGUA E
ENERGIA ELÉTRICA UTILIZANDO PRÁTICAS DE
ENGENHARIA DE SOFTWARE**

VITÓRIA-ES

Julho/2023

Daniel Pinto Barros

**DESENVOLVIMENTO DE UM SISTEMA DE
MONITORAMENTO DE CONSUMO DE ÁGUA E
ENERGIA ELÉTRICA UTILIZANDO PRÁTICAS DE
ENGENHARIA DE SOFTWARE**

Parte manuscrita do Projeto de Graduação do aluno Daniel Pinto Barros, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES

Julho/2023

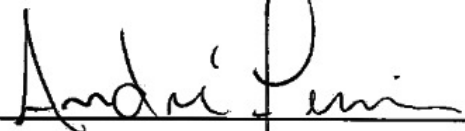
Daniel Pinto Barros

DESENVOLVIMENTO DE UM SISTEMA DE MONITORAMENTO DE CONSUMO DE ÁGUA E ENERGIA ELÉTRICA UTILIZANDO PRÁTICAS DE ENGENHARIA DE SOFTWARE

Parte manuscrita do Projeto de Graduação do aluno Daniel Pinto Barros, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

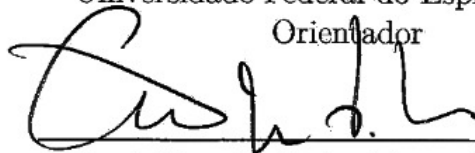
Aprovado em 14 de Julho de 2023.

COMISSÃO EXAMINADORA:



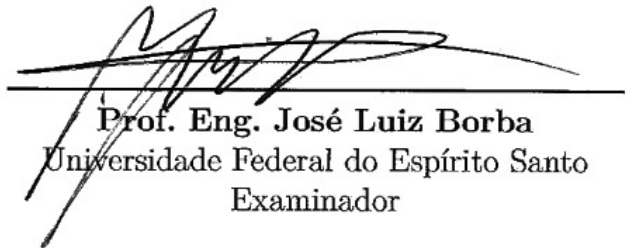
Prof. Dr. André Ferreira

Universidade Federal do Espírito Santo
Orientador



Prof. Dr. Antônio Manoel Ferreira
Frasson

Universidade Federal do Espírito Santo
Examinador



Prof. Eng. José Luiz Borba
Universidade Federal do Espírito Santo
Examinador

Vitória-ES

Julho/2023

RESUMO

Com os sucessivos avanços tecnológicos nas áreas de eletrônica, computação e de infraestrutura da internet, o sensoriamento de processos incorporando conectividade para elementos físicos de uma instalação permite que o mapeamento, a classificação e a utilização de dados gerados na produção industrial automatizada seja de forma otimizada.

Porém, somente a aquisição dos dados não é suficiente para a evolução da indústria atual à conhecida indústria inteligente ou indústria 4.0, sendo também necessárias ferramentas que permitam a acessibilidade, o estudo e o monitoramento dos dados, permitindo a tomada de ações orientadas por eles. Assim, é possível otimizar processos na cadeia produtiva aplicando inteligência e processamento digital nos processos sensoriados.

Neste cenário, com o intuito de tornar o processo produtivo otimizado, todo o seguimento de Engenharia de *Software* e desenvolvimento de sistemas que possibilitam inserir essa capacidade de processamento, vem ganhando forças.

Este trabalho visa desenvolver sistemas e serviços para realizar o monitoramento de medidores de água e energia conectados à internet desenvolvidos para serem instalados no campus de Goiabeiras da Universidade Federal do Espírito Santo (UFES), sendo capaz de facilitar a apresentação, manipulação e alerta de inconsistências dos dados obtidos através dos medidores a operadores humanos.

Palavras-chave: Engenharia de *Software*; Desenvolvimento *Full Stack*; *Docker*; *Python*; *React*.

ABSTRACT

With successive technological advances in the areas of electronics, computing and internet infrastructure, process sensing incorporating connectivity to physical elements of an installation allows the mapping, classification and use of data generated in automated industrial processes to be optimized.

However, only the data acquisition is not enough for the evolution of the current industry to the known smart industry or industry 4.0, being also necessary tools that allow the accessibility, the study and the monitoring of the data, allowing the taking of actions guided by them. Thus, it is possible to optimize processes in the production chain by applying intelligence and digital processing in the sensed processes.

In this scenario, with the aim of optimizing the production process, the entire segment of software engineering and the development of systems that make it possible to insert this processing capacity has been gaining strength.

This work aims to develop systems and services to carry out the monitoring of water and energy meters connected to the internet developed to be installed on the Goiabeiras campus of the Federal University of Espírito Santo (UFES), being able to facilitate the presentation, manipulation and alert of inconsistencies from the data obtained through the meters to human operators.

Keywords: Software Engineering; Full Stack Development; Docker; Python; React.

LISTA DE FIGURAS

| | |
|--|----|
| Figura 1 – Interação com API | 14 |
| Figura 2 – Comunicação entre <i>client</i> e <i>server</i> | 15 |
| Figura 3 – Arquitetura do <i>Docker</i> | 17 |
| Figura 4 – Comparação da arquitetura de virtualização entre Máquinas virtuais e Contêineres | 19 |
| Figura 5 – Fluxograma das práticas de CI/CD | 23 |
| Figura 6 – Arquitetura do sistema desenvolvido | 24 |
| Figura 7 – Modelo das entidades do banco de dados | 26 |
| Figura 8 – Comando para criação da tabela de definição de alarmes no banco de dados | 27 |
| Figura 9 – Exemplo de definição de rotas com Flask API | 27 |
| Figura 10 – Implementação lógica de uma rota com Flask API | 28 |
| Figura 11 – Rotas definidas para a API | 28 |
| Figura 12 – Configuração da biblioteca SQLAlchemy com o banco de dados | 29 |
| Figura 13 – Implementação dos modelos sincronizados com o banco de dados | 29 |
| Figura 14 – <i>Dockerfile</i> para criação da imagem docker do <i>Client</i> API | 31 |
| Figura 15 – Arquivo <i>docker-compose</i> para criação de uma aplicação multi-contêiner | 33 |
| Figura 16 – Docker <i>Client</i> listando os contêineres da aplicação em execução | 34 |
| Figura 17 – Página de login da aplicação | 35 |
| Figura 18 – Página de registro de novos usuários da aplicação | 36 |
| Figura 19 – Página principal, ou página de alarmes, da aplicação | 37 |
| Figura 20 – Página de grupo de medidores da aplicação | 38 |
| Figura 21 – Página de criação de um grupo de medidores da aplicação | 38 |
| Figura 22 – Página de listagem de medidores da aplicação | 39 |
| Figura 23 – Página para adicionar medidor da aplicação | 40 |
| Figura 24 – Página de detalhes de um medidor da aplicação | 41 |
| Figura 25 – Página para criação de definições de alarme da aplicação | 42 |
| Figura 26 – Página de relatórios de consumo de energia | 43 |
| Figura 27 – Página de relatórios de consumo de água | 43 |
| Figura 28 – Relatório com erro de medição | 45 |

LISTA DE TABELAS

| | |
|---|----|
| Tabela 1 – Códigos de operação do Modbus. | 15 |
|---|----|

LISTA DE ABREVIATURAS E SIGLAS

| | |
|-------|--|
| ACID | <i>Atomicity, Consistency, Isolation, and Durability</i> |
| API | <i>Application Programming Interface</i> |
| CI/CD | <i>Continuous Integration/Continuous Delivery</i> |
| GPL | <i>General Public License</i> |
| HTTP | <i>Hypertext Transfer Protocol</i> |
| IoT | <i>Internet of Things</i> |
| IP | <i>Internet Protocol</i> |
| REST | <i>Representational State Transfer</i> |
| TCP | <i>Transmission Control Protocol</i> |
| UFES | Universidade Federal do Espírito Santo |
| URI | <i>Uniform Resource Identifier</i> |
| URL | <i>Uniform Resource Locator</i> |

SUMÁRIO

| | | |
|------------|--|-----------|
| 1 | INTRODUÇÃO | 10 |
| 1.1 | Apresentação | 10 |
| 1.2 | Objetos e escopo de trabalho | 11 |
| 1.2.1 | Objeto | 11 |
| 1.2.2 | Escopo | 11 |
| 1.3 | Objetivos | 11 |
| 1.4 | Estrutura do Texto | 12 |
| 2 | REFERENCIAL TEÓRICO | 14 |
| 2.1 | Comunicação | 14 |
| 2.1.1 | Protocolo HTTP | 14 |
| 2.1.2 | API REST | 14 |
| 2.1.3 | Modbus TCP | 15 |
| 2.2 | Engenharia de Software | 16 |
| 2.2.1 | Python | 16 |
| 2.2.1.1 | Flask API | 16 |
| 2.2.2 | <i>Docker</i> | 16 |
| 2.2.2.1 | <i>Docker Client e Docker Server</i> | 17 |
| 2.2.2.2 | <i>Imagens Docker</i> | 17 |
| 2.2.2.3 | <i>Registro Docker</i> | 18 |
| 2.2.2.4 | <i>Contêiner Docker</i> | 18 |
| 2.2.2.5 | <i>Virtualização</i> | 18 |
| 2.2.3 | JavaScript | 19 |
| 2.2.3.1 | React | 20 |
| 2.2.4 | NGINX | 20 |
| 2.3 | Banco de dados | 20 |
| 2.3.1 | <i>PostgreSQL</i> | 20 |
| 3 | METODOLOGIA E ETAPAS DE DESENVOLVIMENTO | 22 |
| 3.1 | Metodologia | 22 |
| 3.2 | Etapas de desenvolvimento | 22 |
| 3.2.1 | Arquitetura do Sistema | 23 |
| 3.2.2 | Estruturação do Banco de dados | 25 |
| 3.2.3 | <i>Client-API (Back-end)</i> | 25 |
| 3.2.4 | Cliente Modbus | 30 |
| 3.2.5 | Serviço de monitoramento | 30 |

| | | |
|------------|---|-----------|
| 3.2.6 | Interface <i>web</i> (<i>Front-end</i>) | 30 |
| 3.2.7 | Dockerização | 31 |
| 4 | RESULTADOS | 34 |
| 5 | CONCLUSÃO E TRABALHOS FUTUROS | 44 |
| 5.1 | Conclusão | 44 |
| 5.2 | Trabalhos futuros | 44 |
| | REFERÊNCIAS | 46 |

1 INTRODUÇÃO

1.1 Apresentação

Devido aos sucessivos avanços tecnológicos, desenvolvimentos e inovações, o cenário industrial global transformou-se drasticamente nos últimos anos. A quarta revolução industrial (Indústria 4.0) visa transformar indústrias tradicionais em indústrias inteligentes, incorporando tecnologias inovadoras permitindo que ativos físicos sejam integrados em processos digitais e físicos interligados, criando fábricas e ambientes de fabricação inteligentes (GEORGIOS; KERSTIN; THEOFYLAKTOS, 2019). Quando se fala de indústria 4.0 é comum ouvir termos como inteligência artificial, internet das coisas, *big data* e computação em nuvem, que são tecnologias que vêm sendo empregadas em um processo contínuo acompanhando o surgimento e a atualização de tecnologias (SCHWAB, 2016).

A indústria inteligente exige a obtenção de dados de sensores relevantes e informações de processo em tempo real de todos os componentes da cadeia de valor de fabricação. Prevê-se que a indústria inteligente seja alcançada incorporando conectividade em produtos industriais, usando nuvem e Internet das Coisas (do inglês *Internet of Things* - IoT) para alavancar inteligência e conhecimento acionável para máquinas, colaboração autônoma entre máquinas e integração de produtos e serviços adicionais de valor agregado (BREIVOLD, 2017).

Como mencionado em Rose, Eldridge e Chapin (2015), O termo "Internet das Coisas" foi usado pela primeira vez em 1999 pelo pioneiro da tecnologia britânico, Kevin Ashton, para descrever um sistema no qual objetos no mundo físico poderiam ser conectados à internet através de sensores. Hoje, IoT se tornou um termo popular para descrever cenários no qual conexão de internet e capacidade de processamento são estendidos para uma variedade de objetos, dispositivos, sensores e itens do cotidiano.

Nesse sentido, a fim de desenvolver um sistema de monitoramento do consumo de água e energia no campus de Goiabeira da UFES, em Felix (2022) foi feito um sensoriamento de medidores de água e energia, no qual, os dados dos medidores são disponibilizados para acesso via internet. Porém, se vê necessária a construção de *softwares* que possam interagir com esses dados de maneira a facilitar a leitura e manipulação dos mesmos, assim, otimizando a interação do sistema com seus usuários.

Esse trabalho visa desenvolver um *software* para realizar o monitoramento dos medidores instalados de forma remota. A opção de desenvolver uma aplicação *web* para o problema proposto se mostra a escolha mais adequada, uma vez que essas aplicações podem ser

acessadas em qualquer computador que possua conexão com internet e um navegador instalado, facilitando a implementação e manutenção da aplicação, uma vez que diferentes sistemas operacionais não demandam alteração a arquitetura da aplicação. Em Pop (2002) é feito um estudo empírico, apresentando pontos negativos e o desempenho de aplicações *desktop* e *web*.

1.2 Objetos e escopo de trabalho

1.2.1 Objeto

O objeto deste projeto é a elaboração de um *software web* de produção própria que integra serviços que trabalham de forma paralela com o mesmo objetivo de criar uma central de monitoramento de consumo de água e de energia, criando uma interface que possibilite a aquisição, a manipulação, o processamento e a visualização dos dados gerados por medidores integrados na rede global de *internet*.

1.2.2 Escopo

O escopo é delimitado pela programação das aplicações *front-end* e *back-end*, e dos serviços de *modbus client* e de processamento de dados para geração de alarmes. Para validação, o *software* será integrado com os medidores desenvolvidos em Felix (2022) que já estão instalados em uma residência.

1.3 Objetivos

Objetivo Geral

- Desenvolvimento de aplicações e serviços *web* que possibilitem a aquisição e o processamento das medições de consumo energético e de consumo de água, a fim de possibilitar um monitoramento para usuários humanos, gerando uma interface amigável utilizando de tecnologias de engenharia de software presentes no mercado atual com alto desempenho e operabilidade;

- Desenvolver um sistema escalável em que cada componente seja independente, facilitando futuras migrações de tecnologias ou novos componentes.

Objetivos Específicos

- Implementar uma estrutura de banco de dados relacionais para armazenar todos os dados de sensores, usuários, medições e alarmes;
- Desenvolver uma aplicação *back-end* utilizando conceitos de *Restful API* que atue como peça central na manipulação e aquisição dos dados e possibilite uma integração multiplataforma, utilizando uma *framework* de aplicações *web* em *Python* conhecida como *Flask API*;
- Desenvolver um serviço que integre um cliente *Modbus* com a aplicação *back-end*, responsável pela aquisição de dados de medidores que possuem servidor *Modbus* integrado;
- Desenvolver um serviço que atuará no processamento dos dados de medição, acionando alarmes e informando inconsistências;
- Desenvolver uma aplicação *front-end* para construir a interface gráfica do sistema, facilitando a operação e a leitura de dados para operadores humanos, utilizando a biblioteca JavaScript chamada React;
- Subir todo o sistema desenvolvido de forma containerizada, para execução da aplicação de forma consistente em qualquer ambiente e infraestrutura, utilizando o serviço *Docker*.

1.4 Estrutura do Texto

O presente trabalho está estruturado da seguinte maneira:

- **Introdução:** o capítulo atual que visa realizar a contextualização, a apresentação inicial da proposta, assim como a visão geral do projeto e os objetivos a serem alcançados;
- **Referencial teórico:** neste capítulo serão apresentadas as tecnologias utilizadas;

- **Metodologia e etapas de desenvolvimento:** apresentação do desenvolvimento do projeto e da aplicação das tecnologias envolvidas para a elaboração das aplicações e como as mesmas se integram. O capítulo é finalizado com a proposta do projeto definido dentro do cenário estudado.
- **Resultados:** apresentação da aplicação desenvolvida.
- **Conclusão:** considerações finais sobre o projeto desenvolvido e propostas para melhoria e complementação do sistema.

2 REFERENCIAL TEÓRICO

2.1 Comunicação

2.1.1 Protocolo HTTP

Como mencionado em Berners-Lee, Fielding e Frystyk (1996), Protocolo de Transferência de Hipertexto, do inglês Hypertext Transfer Protocol (HTTP), é um protocolo com a leveza e velocidade necessárias para sistemas de informação hipermídia distribuídos. O HTTP tem sido usado pela iniciativa de informações globais na rede mundial de internet desde 1990. Ele atua na camada de aplicação e é enviado sobre o protocolo TCP/IP.

2.1.2 API REST

De um modo geral, uma API expõe um conjunto de dados e funções para facilitar as interações entre os programas de computador e permitir que eles troquem informações, conforme ilustrado na Figura 1. Uma API é a face de um serviço *web*, ouvindo e respondendo diretamente às solicitações do cliente (MASSE, 2011).

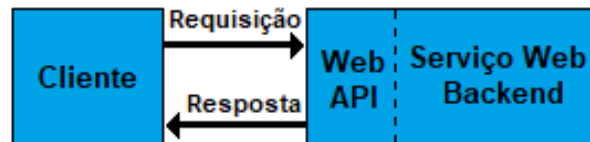


Figura 1 – Interação com API

Fonte: (MASSE, 2011)

O estilo de arquitetura REST é comumente aplicado ao design de APIs para serviços web modernos. Uma API em conformidade com o estilo de arquitetura REST é uma API REST (MASSE, 2011).

A sigla REST, do inglês *Representational State Transfer*, que significa Transferência de Estado Representacional, trata-se de um conjunto de princípios e definições necessárias para a criação de interfaces *web* bem definidas, permitindo a comunicação entre aplicações através do protocolo HTTP (WEBBER; PARASTATIDIS; ROBINSON, 2010).

2.1.3 Modbus TCP

Como mencionado em MODBUS ORGANIZATION (2006) Modbus TCP é um protocolo de comunicação serial, que se encontra sobre a camada de transporte TCP/IP, usado para transmitir informações sobre redes entre dispositivos eletrônicos. Existem dois agentes principais nessa rede: *client*, que é o dispositivo que faz requisição de informações, e o *server*, que são dispositivos que fornecem informações quando requisitados, como ilustra a Figura 2.

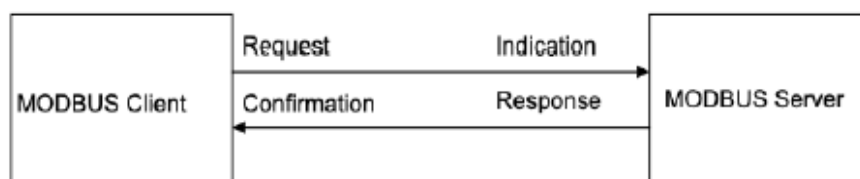


Figura 2 – Comunicação entre *client* e *server*

Fonte: (MODBUS ORGANIZATION, 2006)

Um *client* pode se conectar a 247 *servers*, cada qual com um endereço atribuído de 1 a 247. O primeiro byte de uma requisição feita pelo *client* contém o endereço de qual *server* está sendo feita a requisição, assim cada *server* sabe se deve ou não responder a requisição (MODBUS ORGANIZATION, 2006).

O segundo byte da mensagem é o código de operação que está sendo requisitada para o *server*: leitura ou escrita. A Tabela 1 possui os códigos de operação.

| Código de operação | Ação | Nome da Tabela |
|--------------------|------------------|-------------------------------|
| 01 (01 hex) | Leitura | Saídas Discretas (Bobinas) |
| 02 (02 hex) | Leitura | Entradas Discretas (Contato) |
| 03 (03 hex) | Leitura | Registro de Saída Analógico |
| 04 (04 hex) | Leitura | Registro de Entrada Analógico |
| 05 (05 hex) | Escrita Simples | Saídas Discretas (Bobinas) |
| 06 (06 hex) | Escrita Simples | Registro de Saída Analógico |
| 15 (0F hex) | Escrita Múltipla | Saídas Discretas (Bobinas) |
| 16 (10 hex) | Escrita Múltipla | Registro de Saída Analógico |

Tabela 1 – Códigos de operação do Modbus.

Fonte: (MODBUS ORGANIZATION, 2006)

Os próximos bytes são configurados de acordo com a operação que será requisitada, sendo que os últimos dois bytes são reservados para checagem de erro.

2.2 Engenharia de Software

2.2.1 Python

Como mencionado em Borges (2014), criada em 1990 por Guido van Rossum, Python é uma linguagem de alto nível orientada ao objeto, de tipagem dinâmica e forte, interpretada e interativa. A linguagem apresenta diversas estruturas de alto nível e uma vasta coleção de módulos prontos para uso, além de *frameworks* de terceiros que podem ser adicionados. Também inclui recursos encontrados em outras linguagens de programação modernas tais como geradores, introspecção, persistência, metaclasses e unidades de teste.

A linguagem trouxe mais simplicidade para o projeto desenvolvido uma vez que Python possui *frameworks* para operar banco de dados relacionais (SQLAlchemy), fazer tratativas de requisições HTTP (Django, FlaskAPI, FastAPI e *Request*), implementar modbus *client* (pyModbusTCP) e entre outros.

Python é um software de código aberto com licença compatível com a *General Public License* (GPL), porém menos restritiva, permitindo que o Python seja inclusive incorporado em produtos proprietários.

2.2.1.1 Flask API

Flask é um *framework* que pode ser usado para construir facilmente uma API *web*. As aplicações *web* geralmente precisam de algumas funcionalidades principais, como interação com as solicitações de clientes, roteamento de URLs para recursos, renderização de páginas *web* e interação com bancos de dados de *back-end*. Um *framework* como o Flask disponibiliza pacotes e módulos que executam as tarefas mais complexas e pesadas, assim um desenvolvedor precisa apenas se concentrar na sua configuração e na lógica da aplicação (CHAN; CHUNG; HUANG, 2019).

2.2.2 Docker

Docker é uma tecnologia de virtualização de contêineres que, além de construí-los, fornece um fluxo de trabalho, que ajuda desenvolvedores na criação de contêineres e aplicativos dentro deles e compartilhá-los com outras pessoas (ANDERSON, 2015).

2.2.2.1 Docker Client e Docker Server

O *Docker* pode ser explicado como um aplicativo baseado em cliente e servidor, conforme ilustrado na Figura 3. O servidor *Docker* é quem executa todas as funcionalidades dessa ferramenta, ela obtém uma solicitação de operação do cliente *Docker* e a processa de acordo através de uma API *RESTful* completa administrada pelo *Docker*. O *daemon* (ou servidor) do *Docker* e o cliente *Docker* podem ser executados na mesma máquina ou um cliente *Docker* local pode ser conectado a um *daemon* remoto, que está sendo executado em outra máquina (ANDERSON, 2015).

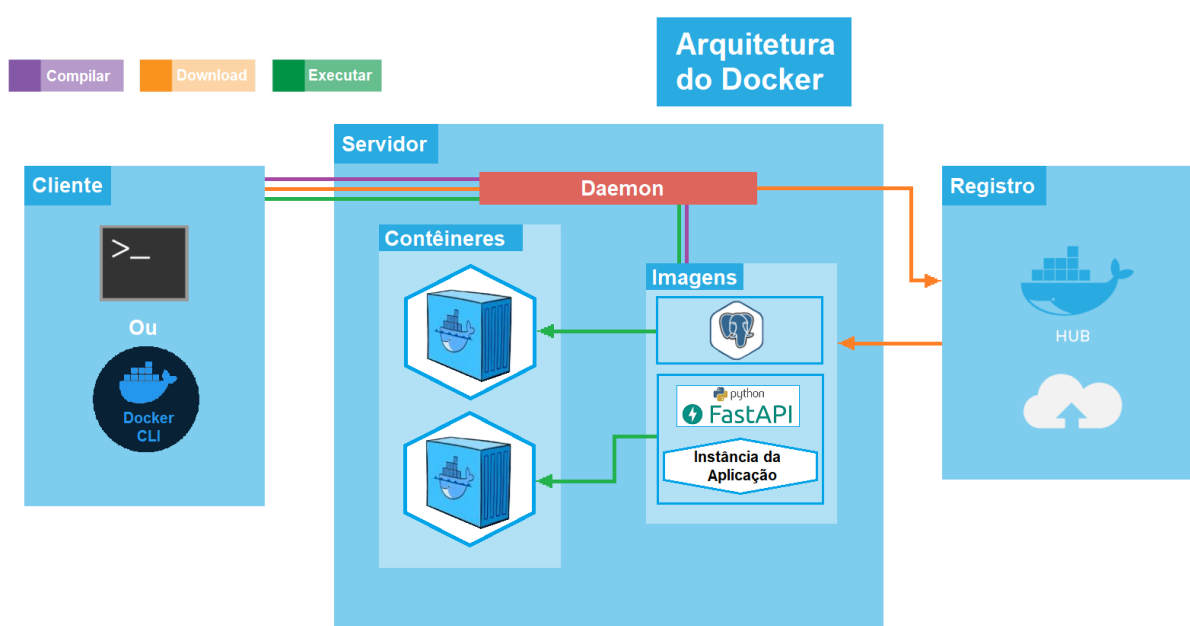


Figura 3 – Arquitetura do *Docker*

Fonte: Podução do próprio autor

2.2.2.2 Imagens *Docker*

A imagem *Docker* é um arquivo utilizado para a execução de códigos e aplicações dentro de um contêiner *Docker*, agindo como um modelo de execução, possuindo uma série de instruções para a sua construção. Essas imagens também agem como um ponto inicial para se utilizar a tecnologia *Docker*.

Existem dois métodos para a criação dessas imagens: o primeiro sendo através da utilização de um molde do tipo "somente leitura". O alicerce de todas as imagens é uma "imagem base" que, geralmente, são instâncias de um sistema operacional, *e.g.* Ubuntu 14.04 LTS e Fedora 20. Essas imagens criam um contêiner com capacidade de execução completa do sistema operacional. Os aplicativos necessários podem ser adicionados à imagem base

modificando-a, sendo necessário construir uma nova imagem. Esse processo de construção é chamado de "*committing change*".

O segundo método é a criação de um arquivo *Docker* contendo uma lista de instruções que são seguidos pelo *daemon* quando o comando "*Docker build*" é executado a partir do terminal *bash*, a fim de se criar uma imagem *Docker*. Esta é uma maneira automatizada de se produzir uma imagem (ANDERSON, 2015).

2.2.2.3 Registro *Docker*

Os registros *Docker* funcionam de forma correspondente aos repositórios de código-fonte, onde imagens podem ser enviadas ou extraídas de uma única fonte.

Existem dois tipos de registros: públicos e privados. O *Docker Hub* é um registro público onde todos podem extrair imagens disponíveis e enviar suas próprias imagens sem haver a necessidade de se criar uma imagem a partir do zero. As imagens podem ser distribuídas para uma área específica, sendo elas pública ou privada, usando o recurso do *Docker Hub* (ANDERSON, 2015).

2.2.2.4 Contêiner *Docker*

Os contêineres *Docker* são criados a partir de imagens já criadas. Os contêineres contêm todo o kit necessário para que uma aplicação possa ser executada de forma isolada. Por exemplo, suponha que haja uma imagem do sistema operacional Ubuntu com SQL SERVER, quando essa imagem for executada com o comando "*Docker run*", um contêiner será criado e o SQL SERVER será executado no sistema operacional Ubuntu (ANDERSON, 2015).

2.2.2.5 Virtualização

A virtualização é um conceito antigo, que vem sendo usado na computação em nuvem. A Figura 4 faz a comparação da arquitetura de uma máquina virtual e dos contêineres.

Na virtualização de máquinas virtuais, o *hypervisor* trabalha entre o sistema operacional e a CPU, e é uma plataforma virtual que opera mais de um sistema operacional dentro do

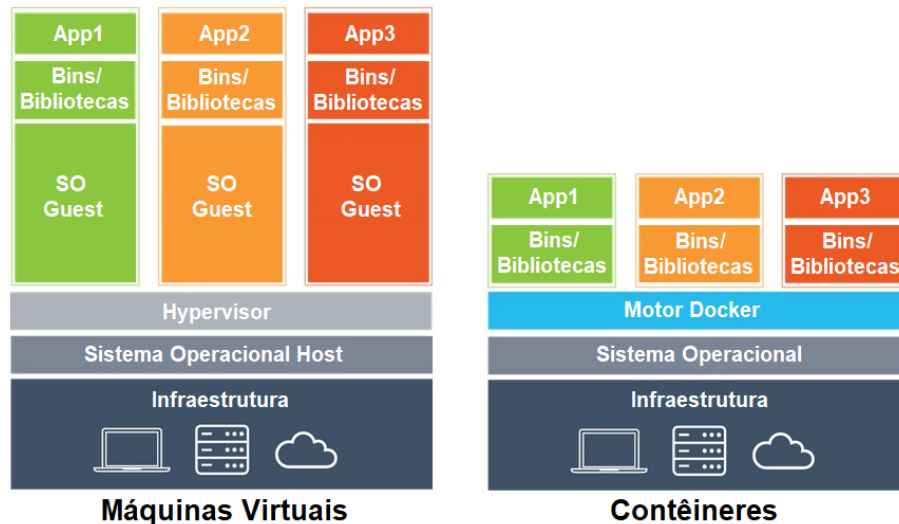


Figura 4 – Comparação da arquitetura de virtualização entre Máquinas virtuais e Contêineres

Fonte: Produção do próprio autor

servidor.

Ao contrário da virtualização de *hypervisor*, em que uma ou mais máquinas independentes são executadas virtualmente em hardware físico por meio de uma camada de intermediação, os contêineres são executados no espaço do usuário sobre o kernel de um sistema operacional. Como resultado, a visualização do contêiner é frequentemente chamada de virtualização no nível do sistema operacional. A tecnologia de contêiner permite que várias instâncias de espaço de usuário isoladas sejam executadas em um único host (TURNBULL, 2014).

A Figura 4 mostra que em um *host* de controle único existem muitos contêineres Linux, que são isolados. Recursos como rede, memória, CPU e *Block I/O* são alocados pelo kernel Linux e também lida com *cgroups* sem iniciar a máquina virtualizada (RAD; BHATTI; AHMADI, 2017).

2.2.3 JavaScript

JavaScript é uma linguagem de programação no qual, atualmente, todos os navegadores modernos possuem interpretadores, tornando-se a linguagem de programação mais onipresente da história. Essa, é uma linguagem de alto nível, dinâmica, interpretada e não tipada, conveniente para estilos de programação orientados ao objeto e funcionais (FLANAGAN, 2004).

2.2.3.1 React

O React é uma biblioteca JavaScript de código aberto com foco em criar interfaces de usuário (*front-end*) em páginas *web* (DAWSON, 2014). Ele foi desenvolvido pelo Facebook para criação de elementos de interface reutilizáveis de forma simples, intuitiva e com um ótimo desempenho, lidando com alguns dos desafios associados a sites de grande escala baseados em dados (GACKENHEIMER; PAUL, 2015).

O React é uma ferramenta poderosa que, atualmente, é utilizada por grandes empresas como Facebook, Twitter, Netflix, Uber, Spotify, Instagram e muitas outras (DAWSON, 2014).

Uma grande vantagem de utilizar React como uma ferramenta de *front-end* é o dinamismo que tal ferramenta adiciona às páginas web, atualizando dados e conteúdo sem a necessidade de atualizar a página. Ao invés do back-end retornar páginas completas em HTML, o React consegue fazer a requisição e atualizar componentes individualmente, reduzindo a transferência de dados e, conseqüentemente, aumentando a velocidade de navegação na aplicação.

2.2.4 NGINX

O NGINX é um famoso software de código aberto para servidores web lançado originalmente para navegação HTTP. Hoje, porém, ele também funciona como proxy reverso, balanceador de carga HTTP, e proxy de email para os protocolos IMAP, POP3, e SMTP.

A ideia de incluir o NGINX na aplicação é de fazer um balanceamento de carga HTTP. Dessa forma, pode-se subir várias instâncias do *back-end* a fim de aumentar o desempenho do sistema.

2.3 Banco de dados

2.3.1 PostgreSQL

O *PostgreSQL* é um sistema gerenciador de banco de dados objeto-relacional de código aberto que utiliza a linguagem SQL combinada com muitos outros recursos que escala

com uma grande variedade de dados complexos, armazenando e dimensionando dados de forma segura (POSTGRESQL ORGANIZATION, 2022).

O *PostgreSQL* conquistou uma forte reputação pela confiabilidade da sua arquitetura comprovada, pela integridade de dados, por um robusto conjunto de recursos, pela extensibilidade e dedicação da comunidade de código aberto por trás do software, para fornecer soluções inovadoras e de desempenho consistentes. O *PostgreSQL* é executado em todos os principais sistemas operacionais e é compatível com ACID desde 2001 (POSTGRESQL ORGANIZATION, 2022).

Além de ser gratuito é altamente extensível. Por exemplo, você pode definir seus próprios tipos de dados, criar funções personalizadas e até escrever código em diferentes linguagens de programação sem recompilar seu banco de dados (POSTGRESQL ORGANIZATION, 2022).

3 METODOLOGIA E ETAPAS DE DESENVOLVIMENTO

3.1 Metodologia

O trabalho proposto, esteve sujeito a muitas mudanças e trocas de requisitos durante a sua elaboração. Foram adotadas práticas de integração contínua e entrega contínua (do inglês, *Continuous Integration/Continuous Delivery*), conhecido na engenharia de software pela sigla CI/CD. Esses dois processos permitem tanto que o processo de desenvolvimento como o processo de entrega seja mais ágil.

Como se pode observar na Figura 5, as práticas de CI/CD se baseiam em um ciclo de ações para entregas de funcionalidades para um sistema ou aplicação. Em resumo do que é discutido em Hirano (2022), o ciclo se baseia nos seguintes passos:

1. Planejamento de novas funcionalidades ou identificação de defeitos;
2. Desenvolvimento de código para implementação da funcionalidade ou correção de defeitos;
3. Construção do sistema;
4. Teste do sistema;
5. Entrega e disponibilização de operação do produto gerado;
6. Monitoramento da aplicação.

Esse ciclo de ações possibilita aos desenvolvedores trabalharem incluindo funcionalidades, identificando e corrigindo falhas do sistema enquanto o mesmo já está em funcionamento.

Assim, o desenvolvimento de todo o sistema foi realizado em paralelo. Todas as aplicações foram desenvolvidas juntas à medida que novas funcionalidades eram sendo incluídas.

3.2 Etapas de desenvolvimento

Tendo entendimento dos objetivos e propósitos que a aplicação deve satisfazer, conhecido no meio de engenharia de software como "entendimento de negócio", pode-se arquitetar

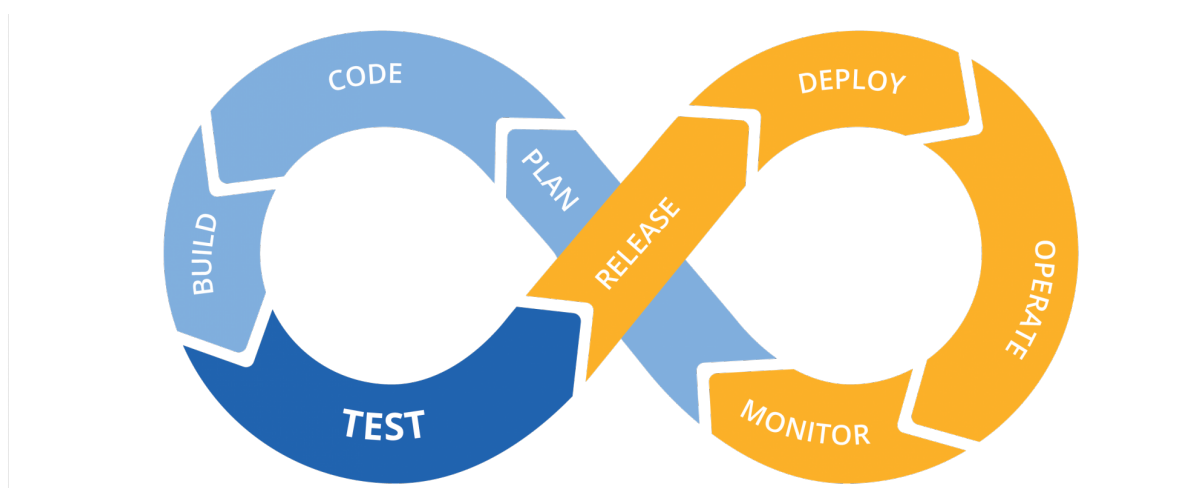


Figura 5 – Fluxograma das práticas de CI/CD

Fonte: (HIRANO, 2022)

uma estrutura de software criando processos e serviços para atender as necessidades encontradas.

3.2.1 Arquitetura do Sistema

O objetivo principal da aplicação é facilitar a visualização dos dados gerados pelos medidores a usuários humanos para monitorar o consumo de energia e água de um complexo industrial com diversos pontos de medições. No caso específico desse projeto, se planeja implementar o sistema no campus de Goiabeiras da Universidade Federal do Espírito Santo. A arquitetura planejada pode ser encontrada na Figura 6

Pensando no fluxo de utilização do sistema, o primeiro passo para que tal sistema seja viável é o desenvolvimento de aplicações *front-end*, que facilitem a interação entre os dados e o usuário. Esta aplicação *front-end* realizará requisições de dados, os quais serão armazenadas em um banco de dados hospedado em um servidor.

Além do *front-end*, deve-se implementar serviços *back-end* para preparar o servidor às regras de negócio. A fim de lidar com as requisições do usuário, limitação de acesso e operações no banco de dados, e até mesmo servir de intermédio para aquisição e persistência dos dados gerados pelos medidores, foi necessário implementar uma API (*API Client*) no lado do servidor. Também foi de fundamental importância a implementação de um banco de dados para a persistência de dados importantes para o funcionamento da aplicação.

A *API Client*, poderá receber dados dos medidores através de requisições HTTP feitas

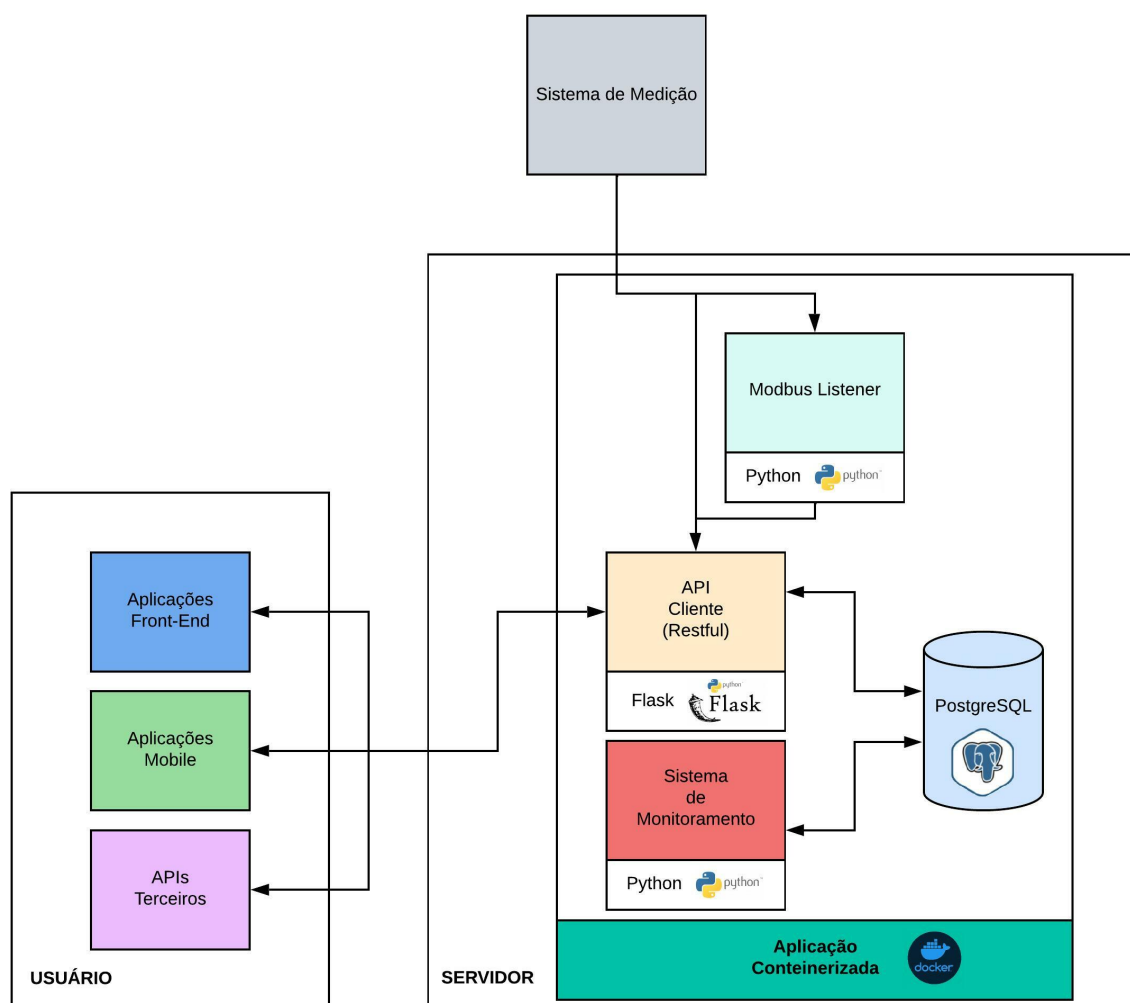


Figura 6 – Arquitetura do sistema desenvolvido

Fonte: Podução do próprio autor

pelos mesmos. Porém alguns medidores possuem o protocolo modbus integrado no seu sistema de comunicação. Para possibilitar a integração dessa tecnologia no sistema. Foi realizado o desenvolvimento de um serviço que atua como um cliente modbus, requisitando os dados dos medidores e enviando para que a *API Client* persista os dados no banco de dados.

Finalizando o pacote de serviços necessários para satisfazer a lógica de negócio, os usuários terão o poder de configurar alarmes para informar inconsistências encontradas nas medições capturadas. Assim, foi desenvolvido um serviço que fará uma varredura nos dados, procurando inconsistências definidas pelos usuários, gerando alarme quando encontradas.

Por fim, com objetivo de definir uma estratégia de implementação (do inglês, *deploy*) do sistema que facilite a integração do mesmo em diversos ambientes, todas as aplicações e

serviços foram criados e configurados para rodar em contêineres Docker.

3.2.2 Estruturação do Banco de dados

Com objetivo de realizar a persistência de dados no *back-end*, foi utilizado um banco objeto relacional PostgreSQL. Com isso se torna necessária a definição das entidades que serão armazenadas no sistema e suas relações. Na Figura 7 pode-se observar a estrutura planejada.

Para satisfazer a lógica de negócio principal do sistema, foram necessárias tabelas que guardem informações dos medidores (*registers*) e de suas medidas (*measures*). Os medidores foram divididos entre grupos para facilitar o monitoramento e a navegação nas aplicações *web*, sendo assim necessário, também, uma tabela de grupos de medidores (*register_groups*). Assim, medidores de um bloco específico do complexo industrial, podem ter uma separação lógica dos demais.

A criação de entidades para armazenar definições de alarme (*alarm_definitions*), que serão criadas pelo usuário, e para armazenar os alarmes gerados (*alarms*), também foi necessária para atender às demandas da aplicação.

Por fim, para restringir o acesso e manipulação dos dados, tanto de usuários quanto de medidores, foram adicionadas tabelas para armazenar informações dos usuários (*users*) e da confirmação de autenticidade dos mesmos (*confirmation*).

Tendo definido as entidades e suas relações conforme ilustrado na Figura 7, foi utilizada a linguagem SQL para criar as tabelas das entidades. A Figura 8 ilustra o comando para a criação da tabela da entidade *alarm_definitions*.

3.2.3 Client-API (Back-end)

O *Client-API* é a peça central da aplicação *back-end*, sendo responsável por conectar todos os serviços através do protocolo HTTP. Por detalhes técnicos mencionados no capítulo "Referencial Teórico", foi decidido implementar a API utilizando a linguagem de programação Python que oferece diversas ferramentas e bibliotecas para facilitar a implementação lógica da aplicação.

Para processar as requisições HTTP que serão realizadas para o servidor, foi utilizada

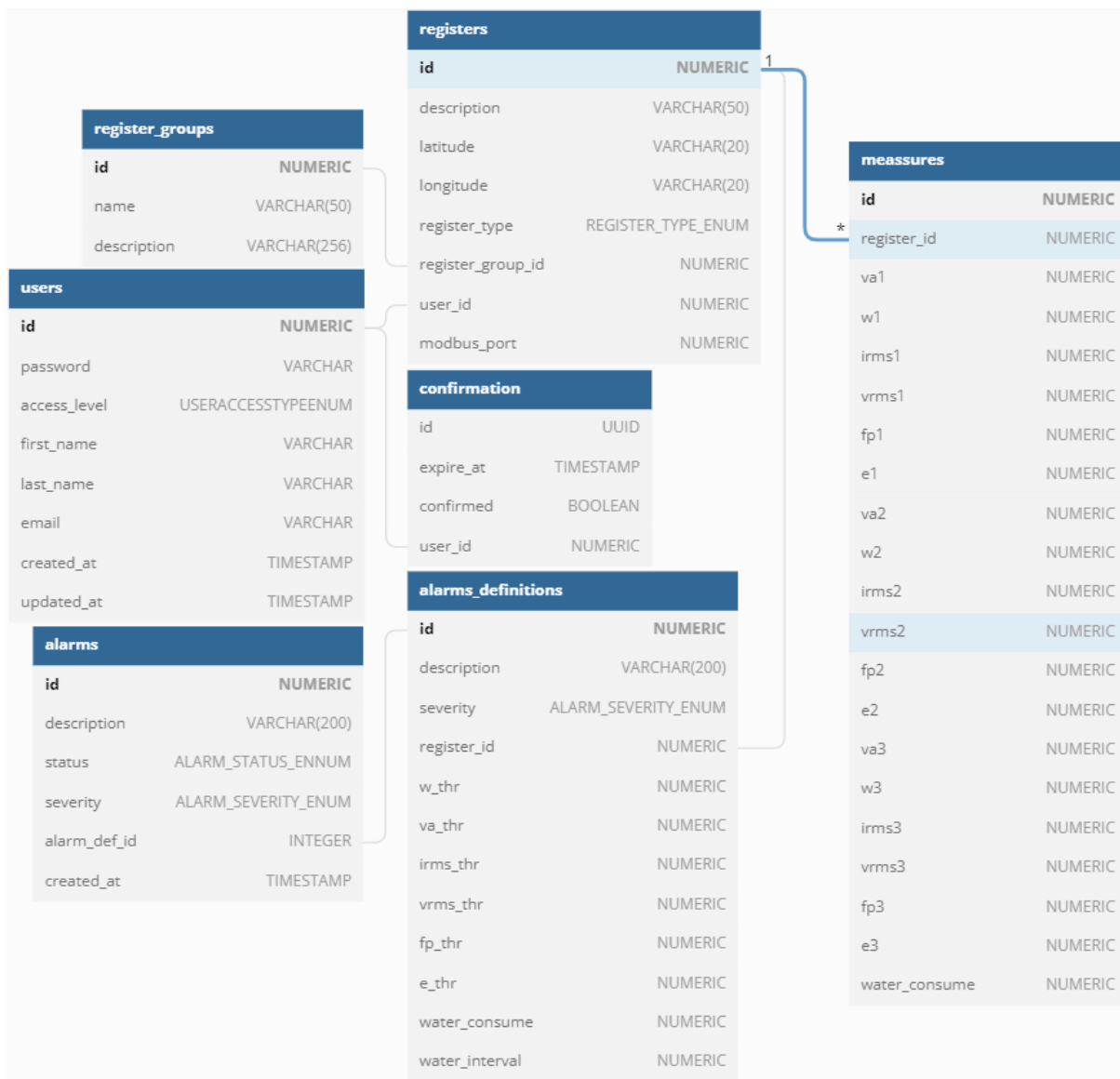


Figura 7 – Modelo das entidades do banco de dados

Fonte: Produção do próprio autor

a *framework* Flask API. Essa *framework* trata os detalhes técnicos e de alto nível do protocolo HTTP, deixando o desenvolvedor responsável, somente, pela implementação lógica da API. A Figura 9 e 10 exemplificam como realizar a implementação de uma API utilizando Flask.

Na Figura 9, estão sendo definidas três rotas para quaaais um cliente pode fazer uma requisição HTTP, assim como a configuração do IP e em qual porta do servidor a API estará exposta. Utilizando a Figura 9 como exemplo, um cliente pode fazer uma requisição HTTP para a URL "http://0.0.0.0:5000/measures" e a classe *MeasureList* será invocada para tratar a requisição. Essa, que, como ilustrada na Figura 10, tratará todas as requisições do tipo *GET* retornando uma lista de todas as medidas já recolhidas e armazenadas no banco de dados. Todas as rotas planejadas para a API estão sendo ilustradas na Figura 11.

```
CREATE TABLE IF NOT EXISTS public.alarm_definitions
(
    id integer NOT NULL DEFAULT nextval('alarm_definitions_id_seq'::regclass),
    description character varying(200) COLLATE pg_catalog."default" NOT NULL,
    severity alarmseverityenum NOT NULL,
    register_id integer NOT NULL,
    w_thr double precision,
    va_thr double precision,
    irms_thr double precision,
    vrms_thr double precision,
    fp_thr double precision,
    e_thr double precision,
    water_consume double precision,
    water_interval integer,
    CONSTRAINT alarm_definitions_pkey PRIMARY KEY (id),
    CONSTRAINT alarm_definitions_register_id_fkey FOREIGN KEY (register_id)
        REFERENCES public.registers (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
TABLESPACE pg_default;
```

Figura 8 – Comando para criação da tabela de definição de alarmes no banco de dados

Fonte: Produção do próprio autor

Além do tratamento das requisições HTTP, a API precisa de uma integração com o banco de dados. Essa integração foi realizada com a biblioteca *SQLAlchemy*, a qual o Flask possui uma extensão. Para realizar a sincronização da biblioteca com o banco de dados, foram criados os modelos das tabelas e a configuração do endereço do banco de dados, como mostram as Figuras 12 e 13. Foi escrito um modelo para cada entidade definida na Figura 7. Dessa forma, podem ser feitas buscas e alterações dos dados armazenados no PostgreSQL pelo Client-API através da sincronização configurada com o *SQLAlchemy*.

```
app = Flask(__name__)
api = Api(app)

api.add_resource(MeasureList, "/measures")
api.add_resource(Measure, "/measure/<int:id>")
api.add_resource(MeasureReport, "/reportMeasures/<int:regId>")

app.run(port=5000, debug=True, host='0.0.0.0')
```

Figura 9 – Exemplo de definição de rotas com Flask API

Fonte: Produção do próprio autor

Outras bibliotecas para serialização e desserialização (Marshmallow), para controle de acesso e autenticação de tokens (JWT *Extended*) foram de grande importância para trazer mais robustez e segurança para a API central do sistema arquitetado.

```

class MeasureList(Resource):
    @classmethod
    def get(cls):
        try:
            checkAccessAllowed([UserAccessLevelEnum.ADMIN, UserAccessLevelEnum.OPERATOR, UserAccessLevelEnum.VIEWER])
        except:
            return {"message": "User not allowed"}, 401

        return {"measures": measure_list_schema.dump(MeasureModel.find_all())}, 200

```

Figura 10 – Implementação lógica de uma rota com Flask API

Fonte: Podução do próprio autor

```

97 api.add_resource(User, "/user") # POST | DELETE
98 api.add_resource(UserRegister, "/signup") # POST
99 api.add_resource(UserLogin, "/login") # POST
100 api.add_resource(TokenRefresh, "/refresh") # POST
101 api.add_resource(UserLogout, "/logout") # POST
102 api.add_resource(UserPassword, "/user/change/password") # POST
103 api.add_resource(UserGiveAccess, "/user/change/access") # POST
104
105 api.add_resource(RegisterRegister, "/register") # POST
106 api.add_resource(Register, "/register/<int:id>") # GET | PUT | DELETE
107 api.add_resource(RegisterList, "/registers") # GET
108 api.add_resource(RegisterModbus, "/registers/modbusOn") # GET
109
110 api.add_resource(RegisterGroups, "/regGroup" ) # GET | POST | PUT | DELETE
111 api.add_resource(RegisterByRegisterGroup, "/regGroup/<int:id>/registers" ) # GET
112
113 api.add_resource(MeasureList, "/measures") # GET
114 api.add_resource(Measure, "/measure/<int:id>") # GET | DELETE
115 api.add_resource(MeasureReport, "/reportMeasures/<int:regId>") # POST
116
117 api.add_resource(MeasureRegister, "/measure") # POST
118 api.add_resource(MeasureFromRegister, "/register/<int:registerId>/lastMeasure") # GET
119 api.add_resource(MeasuresFromRegister, "/register/<int:registerId>/measures") # GET
120 api.add_resource(Confirmation, "/confirmation/<string:confirmation_id>") # GET
121 api.add_resource(ConfirmationByUser, "/resentconfirmationemail/<string:user_email>") # POST
122
123 api.add_resource(AlarmOpen, '/openAlarms') # GET
124 api.add_resource(AlarmCloseID, '/closeAlarm/<int:id>') # POST
125
126 api.add_resource(AlarmDefinition, '/alarmDefinitions') # GET | POST
127 api.add_resource(AlarmDefinitionID, '/alarmDefinitions/<int:id>') # GET | DELETE
128 api.add_resource(AlarmDefinitionRegisterID, '/register/<int:id>/alarmDefinitions') #GET
129

```

Figura 11 – Rotas definidas para a API

Fonte: Podução do próprio autor.

```
app = Flask(__name__)

pguser = os.environ.get("PG_USER")
pghost = os.environ.get("PG_HOST")
pgdatabase = os.environ.get("PG_DATABASE")
pgpassword = os.environ.get("PG_PASSWORD")
pgport = os.environ.get("PG_PORT")
postgres_url = "postgresql_address_ip://{user}:{passwd}@{host}:{port}/{database}".format(
    user = pguser,
    passwd = pgpassword,
    host = pghost,
    port = pgport,
    database = pgdatabase
)

app.config["SQLALCHEMY_DATABASE_URI"] = postgres_url
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
```

Figura 12 – Configuração da biblioteca SQLAlchemy com o banco de dados

Fonte: Produção do próprio autor

```
class RegisterGroupModel(db.Model):
    __tablename__ = "register_groups"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)
    description = db.Column(db.String(50))

    registers = db.relationship("RegisterModel", lazy="dynamic", cascade="all, delete-orphan")

    @classmethod
    def find_all(cls) -> List["RegisterGroupModel"]:
        return cls.query.all()

    @classmethod
    def find_by_id(cls, _id: int) -> "RegisterGroupModel":
        return cls.query.filter_by(id=_id).first()

    @classmethod
    def find_by_name(cls, name: str) -> "RegisterGroupModel":
        return cls.query.filter_by(name=name).first()

    def save_to_db(self) -> None:
        db.session.add(self)
        db.session.commit()

    def delete_from_db(self) -> None:
        db.session.delete(self)
        db.session.commit()
```

Figura 13 – Implementação dos modelos sincronizados com o banco de dados

Fonte: Produção do próprio autor

3.2.4 Cliente Modbus

Para possibilitar a integração de medidores com modbus integrado, foi necessária a elaboração de um serviço paralelo que faria o papel de um Cliente Modbus. Python também possui bibliotecas para desenvolver clientes modbus como o "*pyModbusTCP.client*", justificando, mais uma vez, a escolha dessa linguagem.

O serviço se comunica com o *Client* API através de requisições HTTP solicitando a lista de medidores salvos no banco de dados que se beneficiam da tecnologia Modbus. Tendo as informações dos medidores, o serviço (*modbus listener*) envia uma requisição de leitura para todos os medidores listados, processa os dados recebidos e envia os dados processados para o *Client* API.

3.2.5 Serviço de monitoramento

Usuários do sistema, terão a possibilidade de configurar alarmes para ajudar no monitoramento das medições obtidas. Por exemplo, se um medidor de energia apresentar um consumo maior de um valor especificado pelo usuário, um alarme será notificado para o mesmo. Esse comportamento tornou necessária a presença de um sistema para realizar a geração desses alarmes de forma assíncrona.

Pensando nisso, foi desenvolvido outro serviço hospedado no servidor que interage apenas com o banco de dados, recuperando as definições de alarmes criadas pelos usuários e fazendo uma varredura pelas medições a fim de encontrar inconsistências. Caso elas sejam encontradas, um alarme é criado e inserido diretamente no banco de dados.

3.2.6 Interface *web* (*Front-end*)

Com toda a regra de negócio definida e implementada no *back-end*, o próximo passo foi gerar uma interface amigável para que os usuários pudessem interagir com todo o dado gerado. Utilizando a biblioteca JavaScript chamada *React*, foi possível gerar componentes que, juntos, formam páginas web de forma dinâmica carregando apenas os dados necessários e requisitados para os componentes que serão mostrados na tela.

Todas as requisições de dados que *front-end* realiza são feitas para o *Client* API, e os dados recebidos são estruturados em componentes definidos em HTML e CSS e mostrados

no navegador do usuário.

Em conjunto, o *front-end* e o *back-end*, propiciam ao usuário a possibilidade de realizar o seu cadastro no sistema, criar e editar grupos de medidores, inserir e remover medidores no sistema, criar e remover definições de alarmes, visualizar alarmes, gerar relatórios gráficos das medidas de um medidor e acompanhar em tempo real as medidas e atualizações do sistema em monitoramento.

3.2.7 Dockerização

Com todos os componentes desenvolvidos e prontos para integração, é necessário criar as imagens docker de cada sistema desenvolvido para criação dos contêineres. Como descrito na sessão teórica sobre imagens docker, existem duas formas de criar uma imagem. A forma mais comum de se criar essas imagens é por meio de um arquivo de texto, por padrão nomeado como "*Dockerfile*" e localizado no diretório do projeto, com listas de instruções que serão executadas em um terminal bash para montar a imagem. A Figura 14 ilustra o conteúdo dentro do *Dockerfile* para criação da imagem do *Client* API.

```
1 FROM python:3.10-alpine
2
3 RUN mkdir /app
4
5 WORKDIR /app
6 RUN \
7     apk add --no-cache postgresql-libs && \
8     apk add --no-cache --virtual .build-deps gcc musl-dev postgresql-dev
9
10 ADD ./requirements.txt /app/requirements.txt
11 RUN pip install -r requirements.txt
12
13 ADD . /app
14
15 CMD [ "python", "app.py" ]
```

Figura 14 – *Dockerfile* para criação da imagem docker do *Client* API

Fonte: Produção do próprio autor

Oito instruções são executadas para a criação da imagem, a cada instrução executada são realizadas alterações na imagem base que são registradas em novas instâncias, conhecidas comumente como snapshots. As instruções executadas na Figura 14 são:

1. Acessa os registros docker que faz o *download* de uma imagem docker de um sistema linux com Python 3.10 já instalado, criando e rodando um contêiner a partir dessa

imagem base. Todos os próximos comandos são executados no terminal bash do contêiner recém criado;

2. Cria um diretório `"/app"` dentro do contêiner;
3. Define o novo diretório como o diretório de trabalho do contêiner;
4. Instala dependências e bibliotecas necessárias para integrar a API com banco de dados PostgreSQL;
5. Copia o arquivo `"requirements.txt"` do diretório do projeto para o diretório de trabalho do contêiner. Esse arquivo possui todos as *frameworks* e bibliotecas em python necessárias para rodar o sistema desenvolvido;
6. Executa um comando que instala todas as dependências do arquivo `"requirements.txt"` no contêiner;
7. Copia todo o conteúdo do projeto, ou seja, todo código desenvolvido, para dentro do diretório de trabalho do contêiner;
8. Define o comando inicial que deve ser executado quando o contêiner iniciar, iniciando a aplicação desenvolvida.

Foi escrito um Dockerfile para cada um dos elementos desenvolvidos (cliente Modbus, serviço de monitoramento, banco de dados PostgreSQL, aplicação *front-end* de interface e o *client* API), para gerar uma imagem para cada um desses sistemas. Dessa forma pode-se criar contêineres isolados para cada aplicação. Porém para realizar a integração do sistema, elemento por elemento, deve-se criar uma aplicação multi-contêiner.

O Docker *Compose* é uma ferramenta desenvolvida para ajudar a definir e compartilhar aplicativos de vários contêineres, bastando apenas um arquivo YAML para definir os serviços (DOCKER *DOCS*, 2022). Nesse arquivo de configuração, são definidos os serviços que serão utilizados, assim como suas imagens docker. Além de configurações adicionais como mapeamento de portas, dependência de contêineres, variáveis de ambiente, informações de construção de imagem, dentre outras.

A grande vantagem de usar o *Compose* é a possibilidade de definir uma pilha de aplicativos em um arquivo YAML, mantê-lo na raiz do repositório do projeto e permitir que outros desenvolvedores contribuam facilmente com o mesmo. O repositório pode ser clonado e o aplicativo de composição iniciado com apenas um comando (DOCKER *DOCS*, 2022).

```

1  version: '3'
2  services:
3    postgres:
4      image: 'postgres:latest'
5      environment:
6        POSTGRES_PASSWORD: postgres_password
7        POSTGRES_USER: postgres
8        POSTGRES_DB: postgres
9      volumes:
10     - ./pgdata:/var/lib/postgresql/data
11     restart: always
12     ports:
13     - '13000:5432'
14
15   nginx:
16     depends_on:
17       - api-client
18     restart: always
19     build:
20       dockerfile: Dockerfile.dev
21       context: ./nginx
22     ports:
23     - '5000:80'
24
25   api-client:
26     depends_on:
27       - postgres
28     build:
29       dockerfile: Dockerfile.dev
30       context: ./api-client
31     volumes:
32     - ./api-client:/app
33     restart: always
34     environment:
35       PGUSER: postgres
36       PGHOST: postgres
37       PGDATABASE: postgres
38       PGPASSWORD: postgres_password
39       PGPORT: 5432
40     expose:
41     - '5000'
42
43   modbus-listener:
44     depends_on:
45       - api-client
46       - postgres
47     restart: on-failure
48     build:
49       dockerfile: Dockerfile.dev
50       context: ./modbus-listener
51     volumes:
52     - ./modbus-listener:/app
53
54   monitoring-system:
55     depends_on:
56       - api-client
57       - postgres
58     restart: on-failure
59     environment:
60       PGUSER: postgres
61       PGHOST: postgres
62       PGDATABASE: postgres
63       PGPASSWORD: postgres_password
64       PGPORT: 5432
65     build:
66       dockerfile: Dockerfile.dev
67       context: ./monitoring-system
68     volumes:
69     - ./monitoring-system:/app
70
71   web-interface:
72     depends_on:
73       - api-client
74     build:
75       dockerfile: Dockerfile.dev
76       context: ./web-interface
77     volumes:
78     - ./web-interface:/app
79     restart: always
80     environment:
81       APICLIENURL: api-client
82     expose:
83     - '3000'

```

Figura 15 – Arquivo *docker-compose* para criação de uma aplicação multi-contêiner

Fonte: Produção do próprio autor

4 RESULTADOS

Todos os códigos desenvolvidos para a produção do sistema estão sob domínio público e versionados via Git pela plataforma GitHub. (BARROS, 2022).

Para sua execução basta realizar o *download* do repositório em uma máquina com Docker e Git instalados e executar um comando Docker para iniciar a aplicação em contêineres (BARROS, 2022). Não é necessário ter bibliotecas e outros *softwares* utilizados no projeto instalados para executar a aplicação.

Após executar o comando Docker para iniciar a aplicação, é possível observar e acompanhar os contêineres criados e em execução pelo Docker *Client* como ilustra a Figura 16.

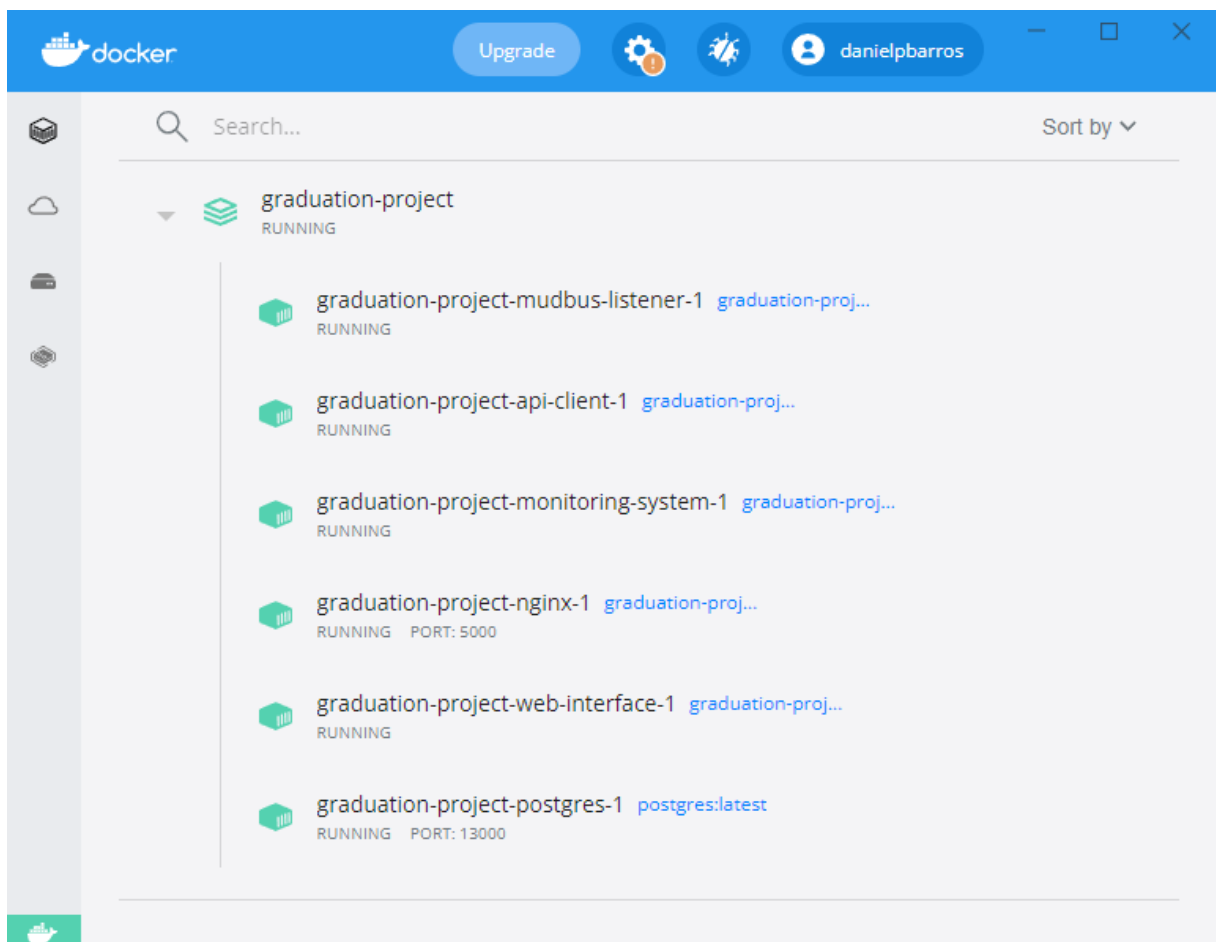


Figura 16 – Docker *Client* listando os contêineres da aplicação em execução

Fonte: Produção do próprio autor.

Com todos os contêineres em execução, inserindo a URL definida para a aplicação em algum navegador *web*, o usuário será encaminhado para a tela de *login* do sistema como

ilustra a Figura 17. Podendo ser redirecionado para tela de registro de novos usuários, como ilustra a Figura 18, ou para a tela principal da aplicação após realizar o *login*, como ilustra a Figura 19.

A tela principal da aplicação, podendo ser definida como a página de alarmes, lista todos os os alarmes acionados, estando eles ativos ou não. Alarmes ativos, são alarmes em que medições inconsistentes estão sendo obtidas em tempo real. Quando as medições voltam aos limites definidos, o alarme é desativado, porém o registro de que existe um alarme desativado continua na página de alarme da aplicação, vide alarme presente na Figura 19, até que o usuário remova o registro.

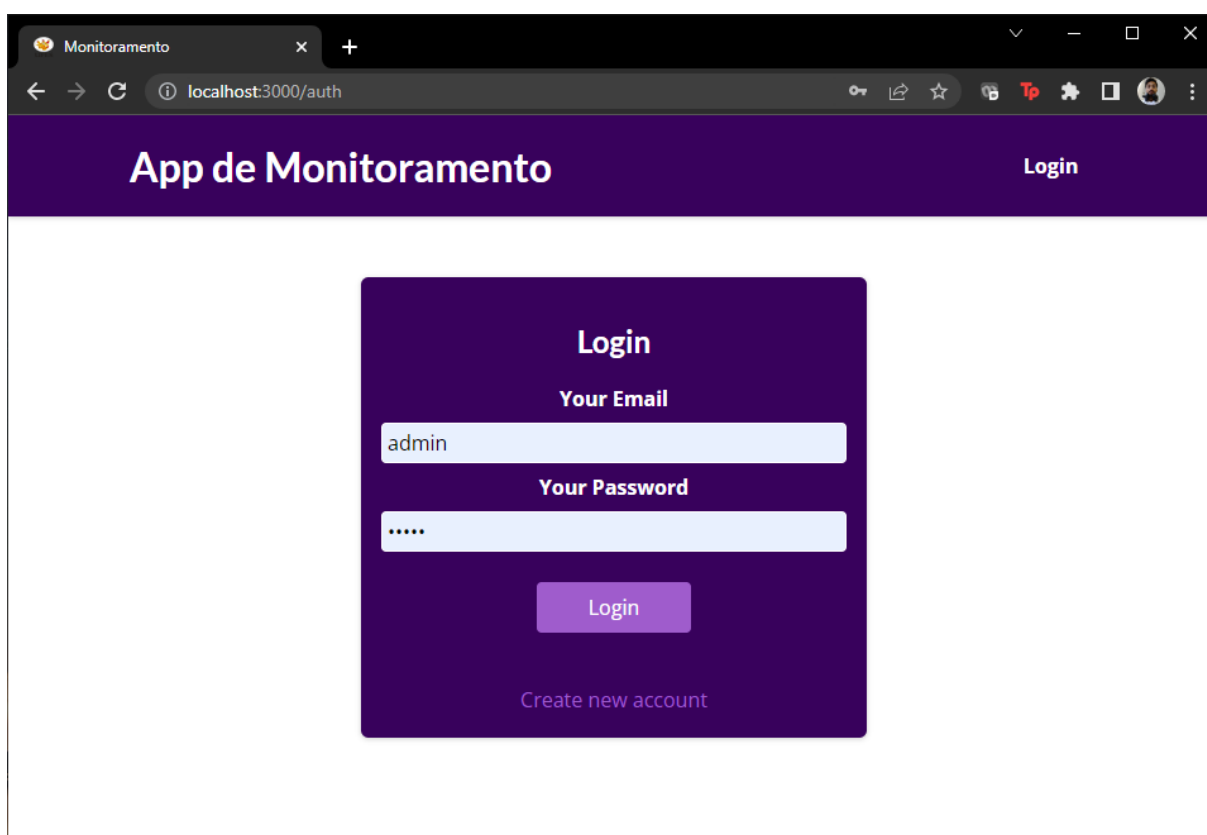


Figura 17 – Página de login da aplicação

Fonte: Produção do próprio autor.

Nas abas da página principal, pode se acessar a página de grupos de medidores, ilustrada na Figura 20. O intuito de existir grupos de medidores é de adicionar uma organização dos medidores a fim de facilitar a utilização dos usuários, por exemplo, podendo agrupar todos os medidores de um prédio ou de um setor separadamente. Nessa página, é possível criar novos grupos, Figura 21, ou listar todos os medidores dentro de um grupo específico, como ilustra a Figura 22.

Na listagem de medidores da Figura 22 é possível navegar para a página de um medidor, onde serão apresentadas todas as informações do dispositivo, além da possibilidade de

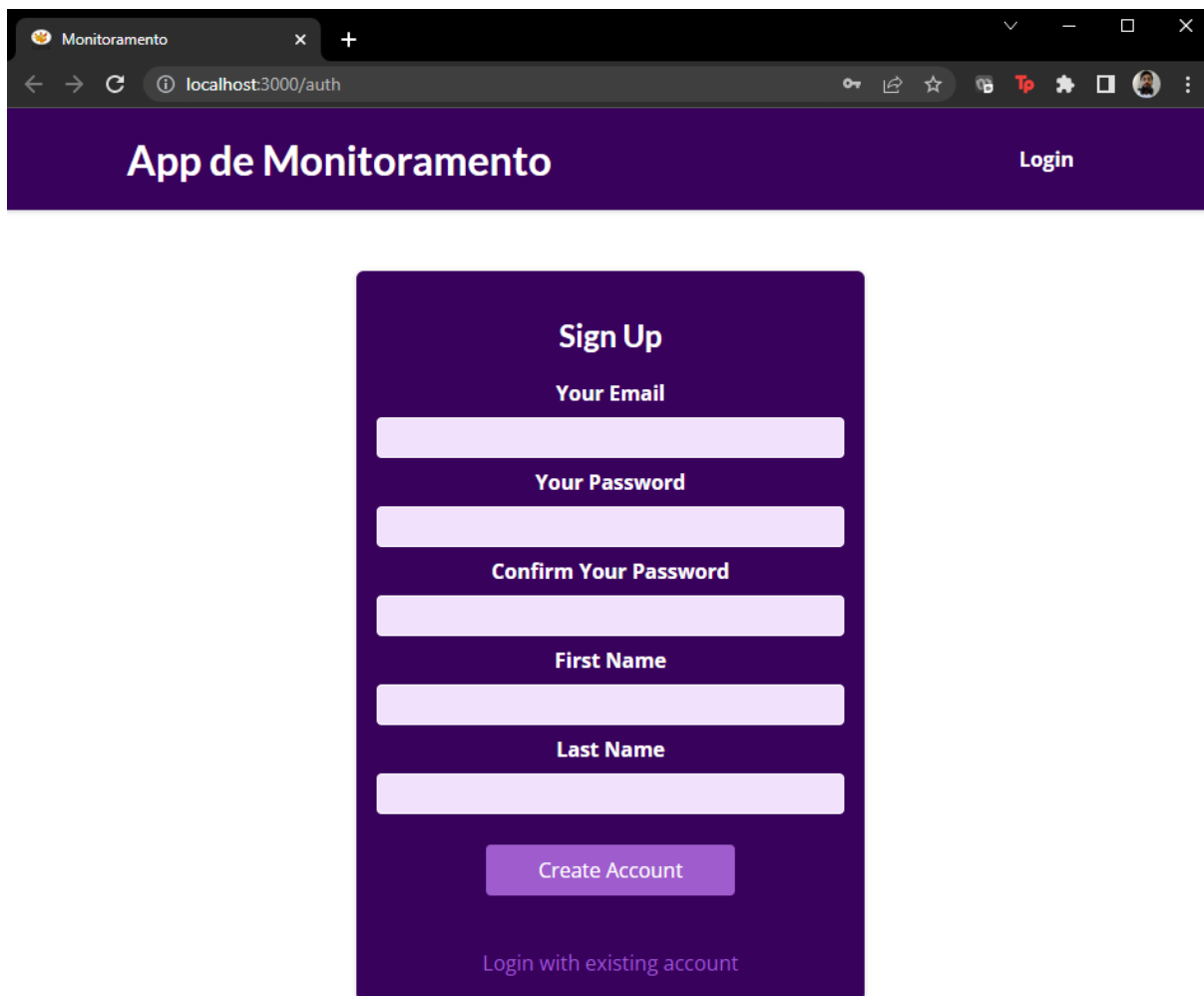


Figura 18 – Página de registro de novos usuários da aplicação

Fonte: Produção do próprio autor.

inserção e remoção de outros medidores ao sistema. A página para adicionar um novo medidor está ilustrada na Figura 23.

Ao navegar para o detalhamento de um medidor, ilustrado na Figura 24, é possível visualizar os detalhes do medidor, a última medição coletada, que é atualizada em tempo real, os alarmes configurados, além de requisitar relatório gráfico das medições e criar novas definições de alarmes (Figura 25).

Já os relatórios gráficos apresentam a mudança temporal de todas as medições de um dia selecionado, como se pode observar nas Figuras 26 e 27, que são, respectivamente, relatórios de energia e de consumo de água.

Cada usuário possui um nível de acesso entre três (visualizador, operador e administrador). Visualizadores só tem acesso aos recursos de visualizar informações de alarmes, dados



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/alarms'. The application header is purple and contains the title 'App de Monitoramento' on the left and navigation links for 'Alarms', 'Grupos de Medidores', 'Profile', and a 'Logout' button on the right. The main content area is titled 'Alarmes' and contains a table with the following data:

| Descrição | Severidade | Estado | Criado em | Medidor | Ações |
|--------------------------|------------|------------|----------------------------|-----------------------------------|-------------------------------|
| Consumo Instantâneo Alto | Crítico | Desativado | 2022-11-19T20:00:20.531088 | Medidor Energia 1 | Fechar Alarme |

Figura 19 – Página principal, ou página de alarmes, da aplicação

Fonte: Produção do próprio autor.

de medidas e relatórios gráficos. Operadores, possuem acesso aos recursos de visualizadores e também podem adicionar e remover medidores, grupos e definições de alarmes. Administradores não possuem restrição de acesso, podendo até prover acesso a outros usuários.

Outra ferramenta implementada é um menu de navegação no cabeçalho da página que possibilita que o usuário retorne a qualquer página do fluxo à medida que o mesmo navega entre elas.

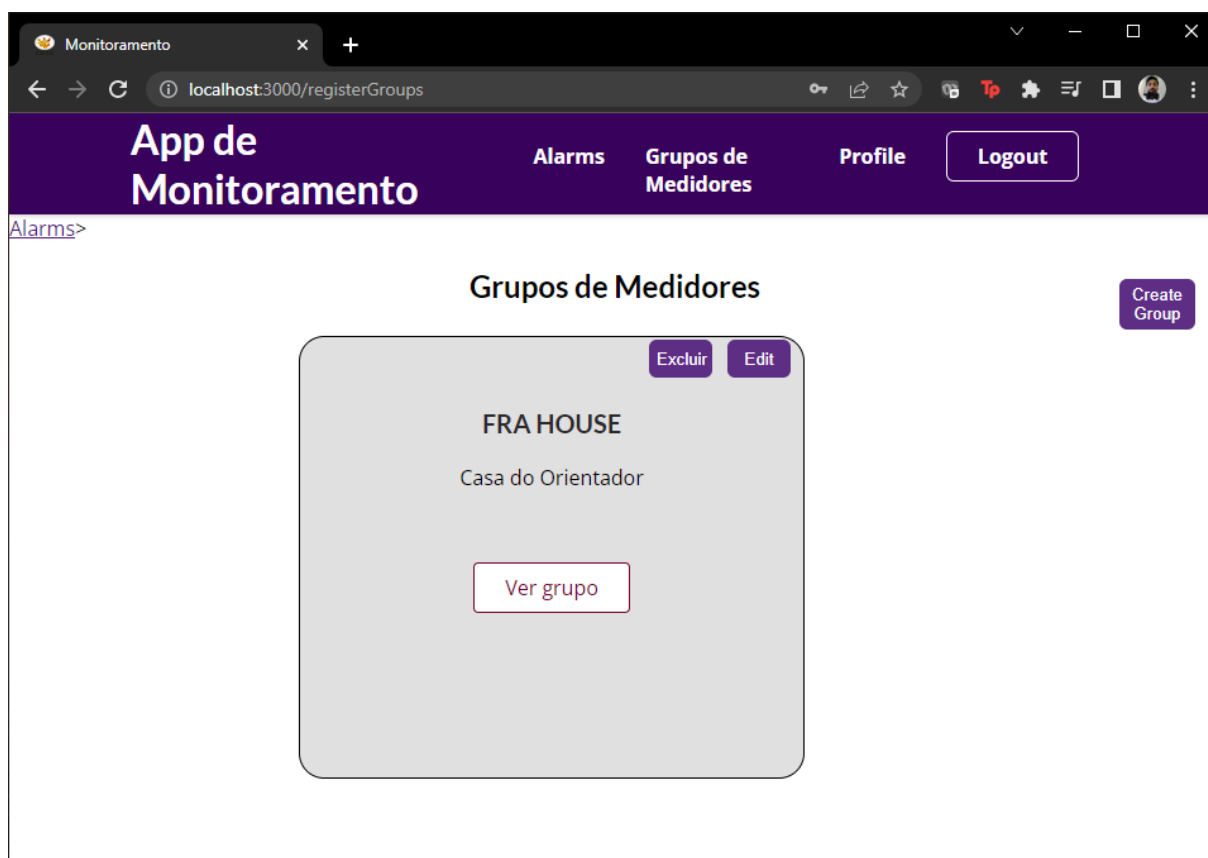


Figura 20 – Página de grupo de medidores da aplicação

Fonte: Produção do próprio autor.

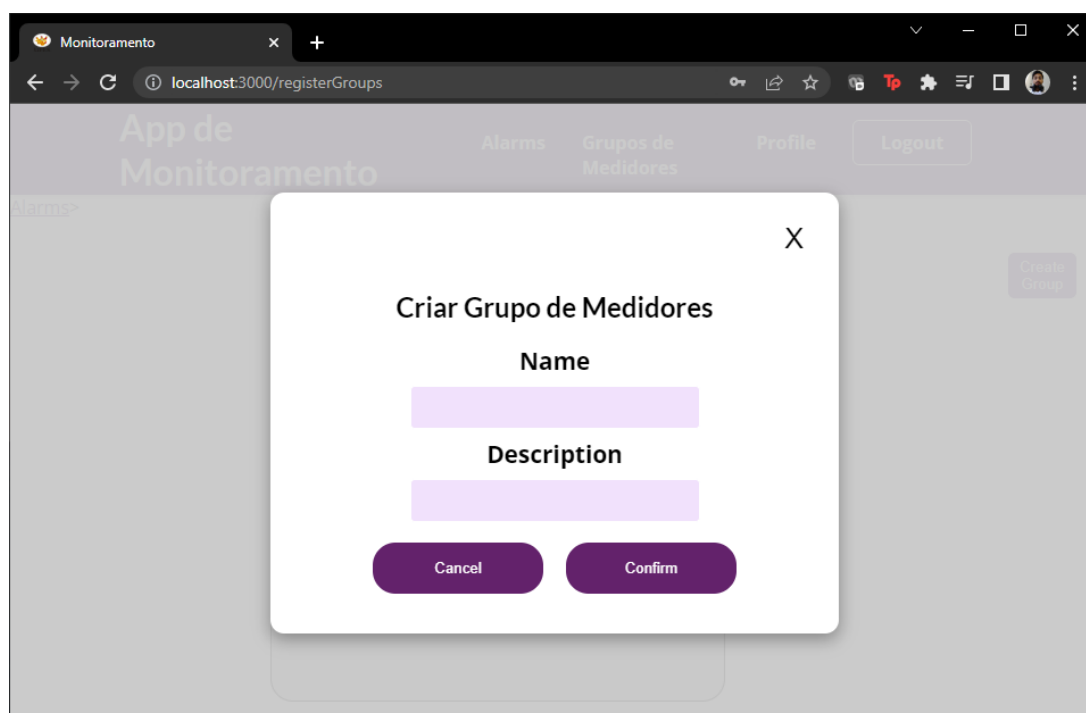


Figura 21 – Página de criação de um grupo de medidores da aplicação

Fonte: Produção do próprio autor.

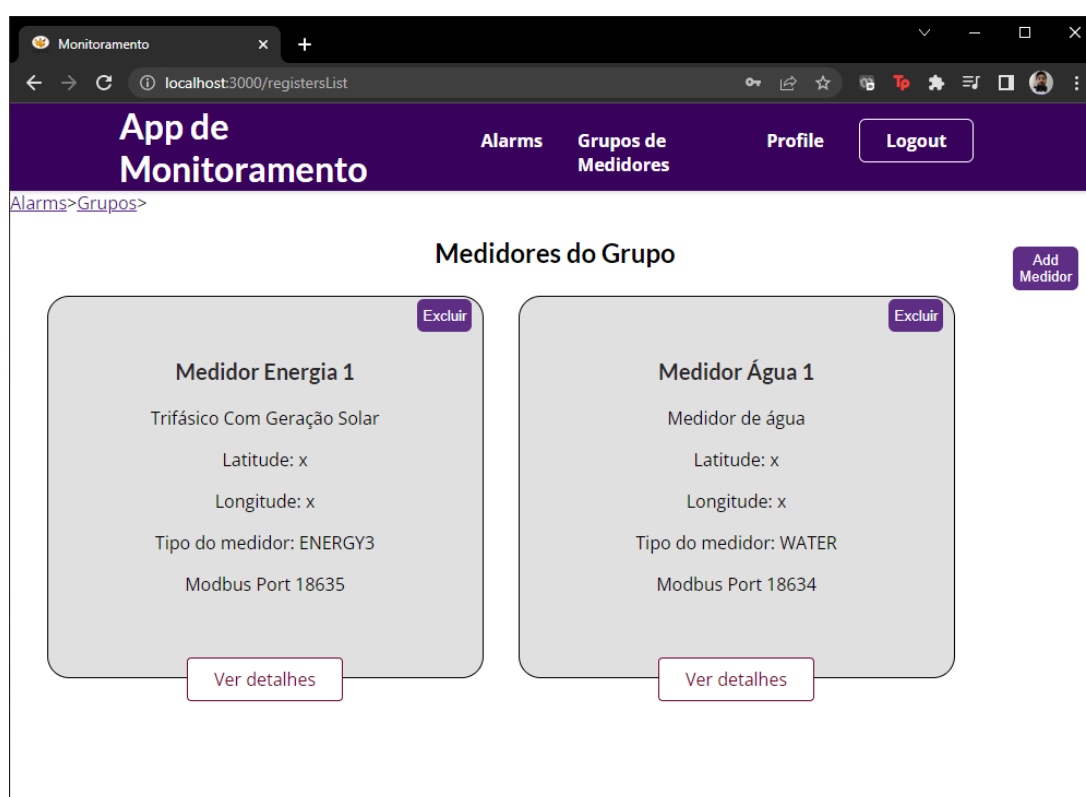


Figura 22 – Página de listagem de medidores da aplicação

Fonte: Produção do próprio autor.

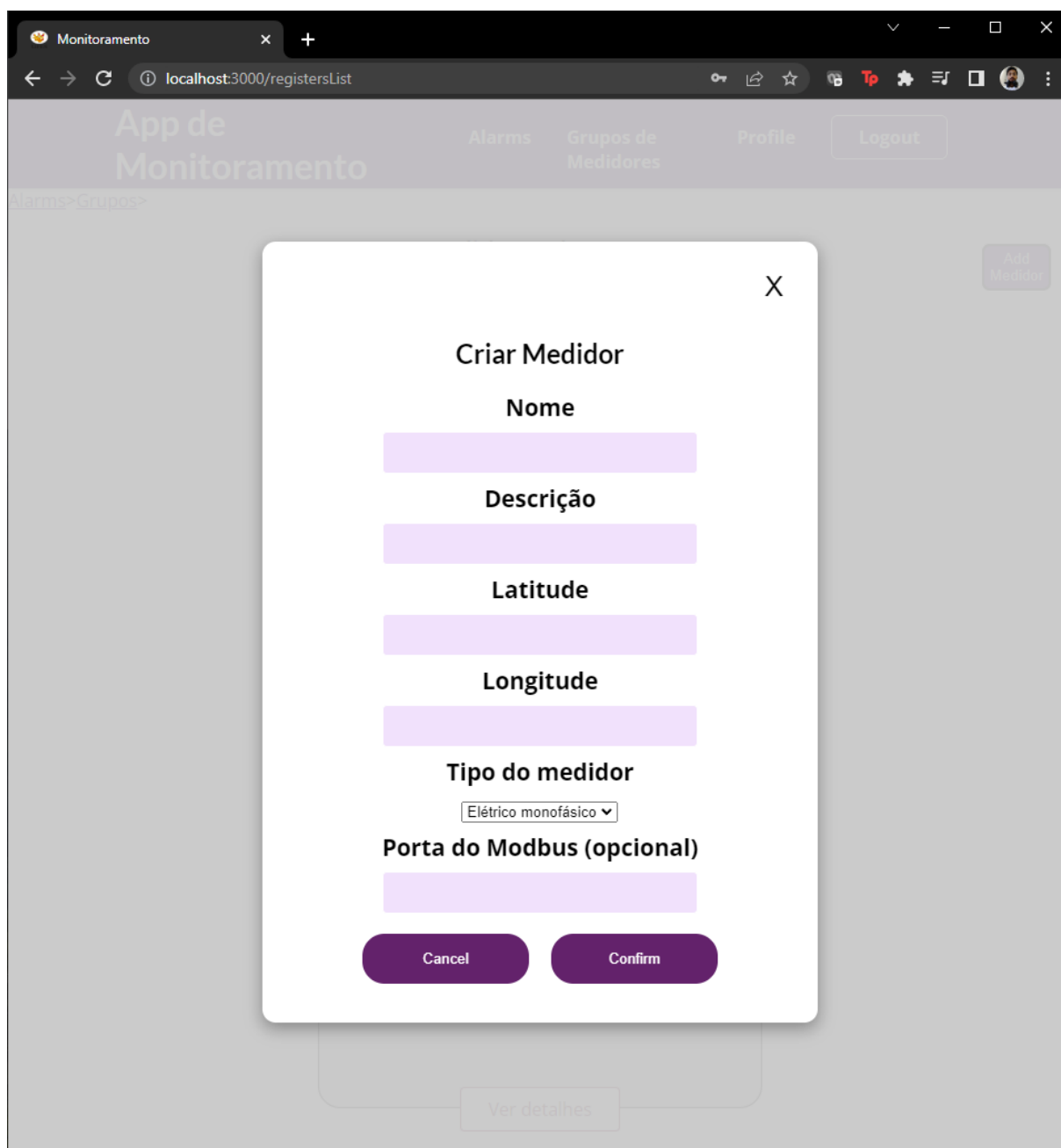


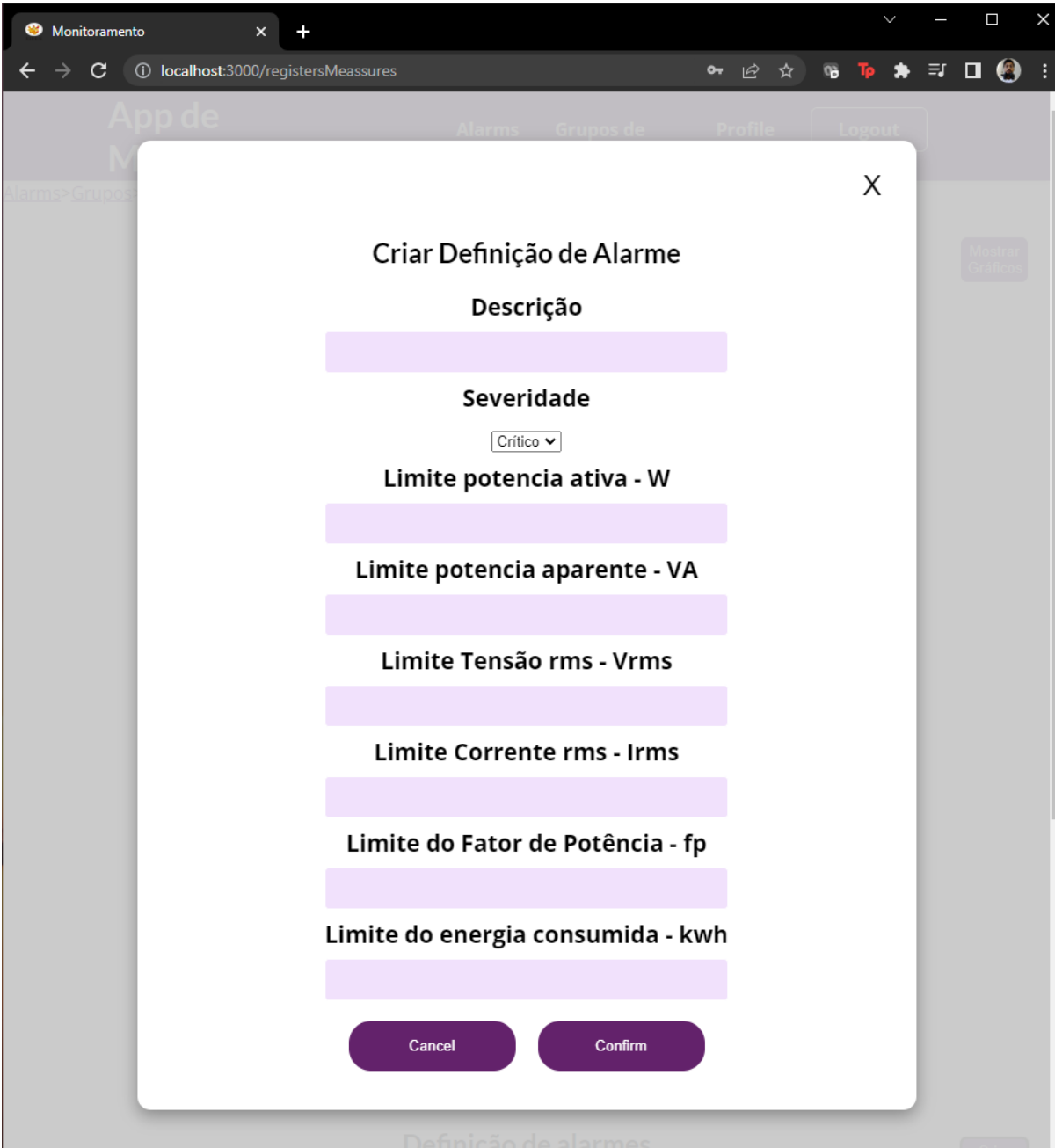
Figura 23 – Página para adicionar medidor da aplicação

Fonte: Produção do próprio autor.



Figura 24 – Página de detalhes de um medidor da aplicação

Fonte: Produção do próprio autor.



The image shows a web browser window with the URL `localhost:3000/registerMeasures`. A modal dialog titled "Criar Definição de Alarme" is displayed. The modal contains the following fields and controls:

- Descrição**: A text input field.
- Severidade**: A dropdown menu currently set to "Crítico".
- Limite potencia ativa - W**: A text input field.
- Limite potencia aparente - VA**: A text input field.
- Limite Tensão rms - Vrms**: A text input field.
- Limite Corrente rms - Irms**: A text input field.
- Limite do Fator de Potência - fp**: A text input field.
- Limite do energia consumida - kwh**: A text input field.

At the bottom of the modal are two buttons: "Cancel" and "Confirm".

Figura 25 – Página para criação de definições de alarme da aplicação

Fonte: Produção do próprio autor.

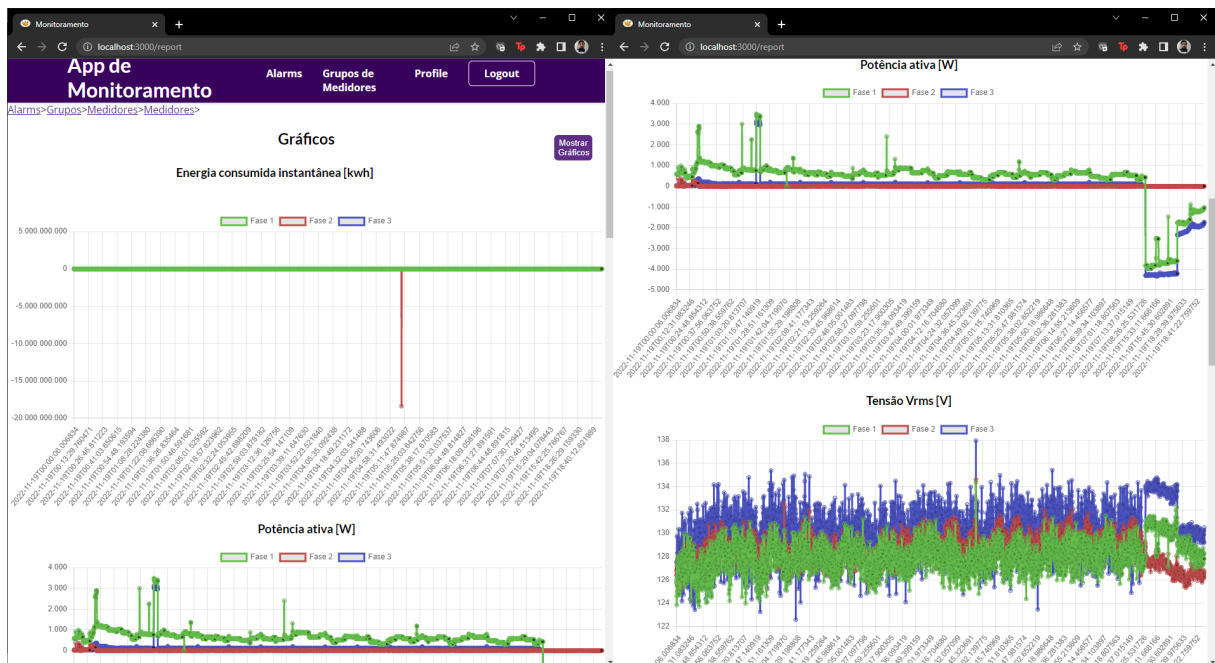


Figura 26 – Página de relatórios de consumo de energia

Fonte: Produção do próprio autor.

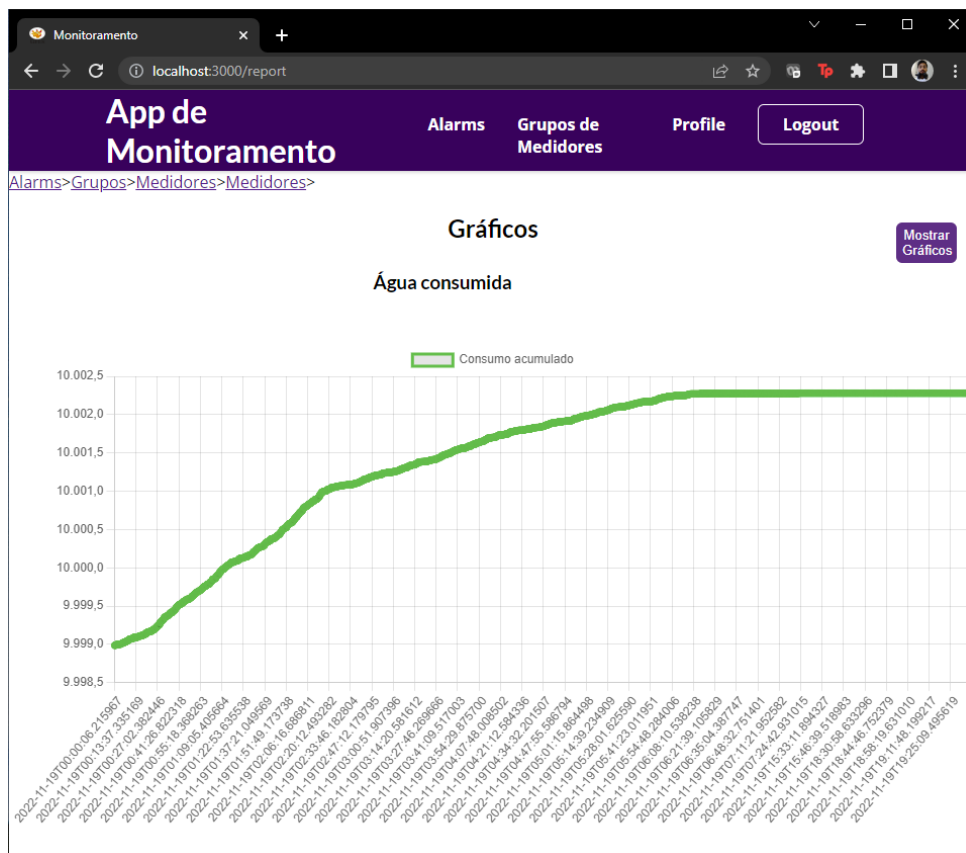


Figura 27 – Página de relatórios de consumo de água

Fonte: Produção do próprio autor.

5 CONCLUSÃO E TRABALHOS FUTUROS

5.1 Conclusão

O sistema criado é funcional e está pronto para operação. A tecnologia *Docker*, utilizada para a containerização da aplicação, é de extrema importância para facilitar a implementação do sistema em qualquer servidor de hospedagem, dado que a maioria desses servidores, atualmente, dão suporte para aplicações em contêineres. Além de tornar mais prática a tentativa de compilar o sistema em máquina local, facilitando a continuação do desenvolvimento da aplicação para futuros desenvolvedores.

Com uma interface intuitiva, tecnologias recentes e de alta performance e com suporte para medidores via protocolo HTTP e Modbus TCP, o sistema apresenta uma alta escalabilidade e flexibilidade para muitos cenários. Com sua arquitetura bem definida e uma base sólida já construída, é fácil fazer adaptações através das práticas de CI/CD para inserir novas funcionalidades e escalar o sistema para outros tipos de medições e sensores.

5.2 Trabalhos futuros

Os relatórios gráficos da aplicação indicam que os medidores apresentam, ocasionalmente, erros de leitura, gerando erros muito acentuados nas medidas como mostra a Figura 28. Um tratamento de dados pode ser inserido a fim de tornar as leituras mais confiáveis.

Ferramentas foram adicionadas ao sistema para disponibilizar dados, facilitando a aquisição de dados para treinamento de algoritmos de aprendizado de máquina. Isso permite a adição de modelos de inteligência artificial que podem fazer previsões de alarmes e gerar relatórios inteligentes.

Novas tecnologias e ferramentas estão sempre surgindo e inovando a engenharia de software. Assim, é importante que sejam realizadas atualizações no sistema pelos desenvolvedores para deixá-lo sempre com alta performance à medida com que a tecnologia avança, além da resolução de *BUGs* que podem ser encontrados a medida que o sistema é utilizado.

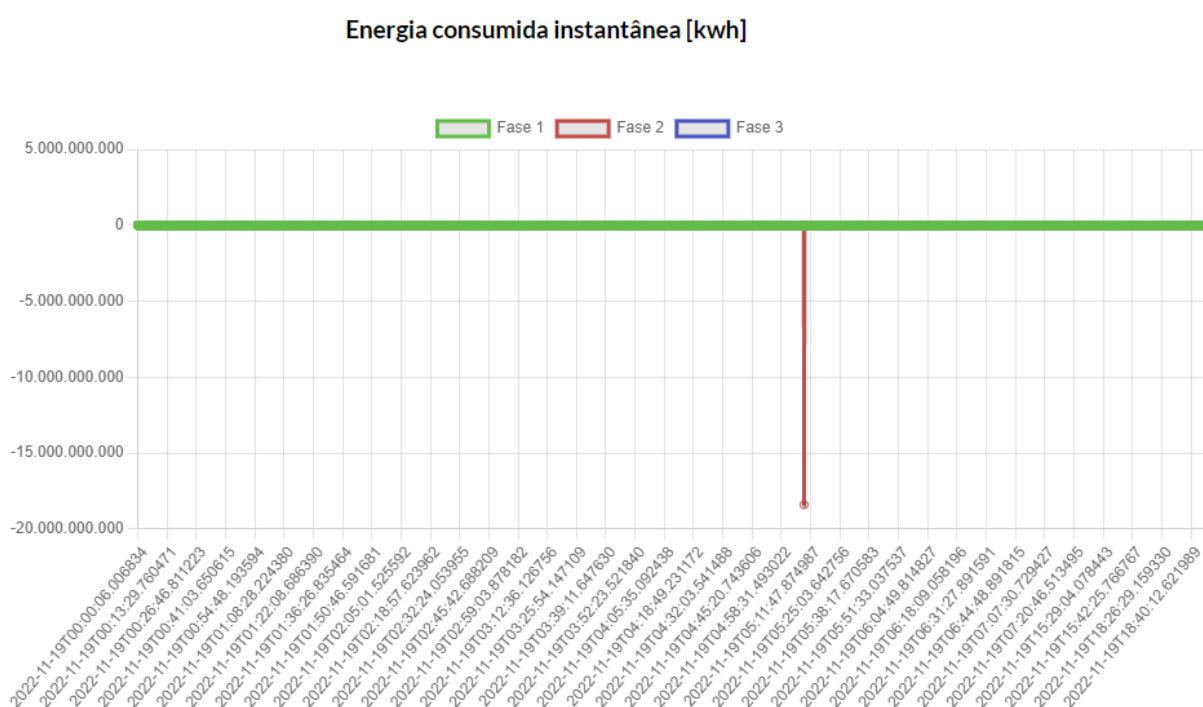


Figura 28 – Relatório com erro de medição

Fonte: Produção do próprio autor.

REFERÊNCIAS

- ANDERSON, C. Docker [software engineering]. Ieee Software, IEEE, v. 32, n. 3, p. 102–c3, 2015. Citado 3 vezes nas páginas 16, 17 e 18.
- BARROS, D. Sistema de Monitoramento - Projeto de Graduação. 2022. Disponível em: <<https://github.com/DanielPintoBarros/graduation-project>>. Acesso em: 23 jun. 2022. Citado na página 34.
- BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. Hypertext transfer protocol-HTTP/1.0. [S.l.], 1996. Citado na página 14.
- BORGES, L. E. Python para desenvolvedores: aborda Python 3.3. [S.l.]: Novatec Editora, 2014. Citado na página 16.
- BREIVOLD, H. P. Internet-of-things and cloud computing for smart industry: A systematic mapping study. In: IEEE. 2017 5th International Conference on Enterprise Systems (ES). [S.l.], 2017. p. 299–304. Citado na página 10.
- CHAN, J.; CHUNG, R.; HUANG, J. Python API Development Fundamentals: Develop a full-stack web application with Python and Flask. [S.l.]: Packt Publishing Ltd, 2019. Citado na página 16.
- DAWSON, C. Javascript’s history and how it led to reactjs. The New Stack, 2014. Citado na página 20.
- DOCKER DOCS. Use Docker Compose. 2022. Disponível em: <https://docs.docker.com/get-started/08_using_compose/>. Acesso em: 21 jun. 2022. Citado na página 32.
- FELIX, Q. Medição e consumo de energia e água com sistema supervisorio. 2022. Projeto de Graduação (Graduação em Engenharia Elétrica) - Centro Tecnológico, Universidade Federal do Espírito Santo, Vitória, 2022. Citado 2 vezes nas páginas 10 e 11.
- FLANAGAN, D. JavaScript: o guia definitivo. [S.l.]: Bookman Editora, 2004. Citado na página 19.
- GACKENHEIMER, C.; PAUL, A. Introduction to React. [S.l.]: Springer, 2015. v. 52. Citado na página 20.
- GEORGIOS, L.; KERSTIN, S.; THEOFYLAKTOS, A. Internet of things in the context of industry 4.0: An overview. International Journal of Entrepreneurial Knowledge, 2019. Citado na página 10.
- HIRANO, M. CI/CD — Continuous Integration and Continuous Delivery. 2022. Disponível em: <<https://medium.com/tecnologia-e-afins/ci-cd-continuous-integration-and-continuous-delivery-fb5d0aed4bf5>>. Acesso em: 23 jun. 2022. Citado 2 vezes nas páginas 22 e 23.
- MASSE, M. REST API design rulebook: designing consistent RESTful web service interfaces. [S.l.]: "O'Reilly Media, Inc.", 2011. Citado na página 14.

- MODBUS ORGANIZATION. MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b. 2006. Disponível em: <<https://www.modbus.org/docs/Modbus-Messaging-Implementation-Guide-V1-0b.pdf>>. Acesso em: 20 jun. 2022. Citado na página 15.
- POP, P. Comparing web application with desktop applications: An empirical study. 2002. Department of Computer and Information Science, Linköping University, Linköping, 2002. Citado na página 11.
- POSTGRESQL ORGANIZATION. PostgreSQL: The world's most advanced open source database. 2022. Disponível em: <<https://www.postgresql.org/>>. Acesso em: 21 jun. 2022. Citado na página 21.
- RAD, B. B.; BHATTI, H. J.; AHMADI, M. An introduction to docker and analysis of its performance. International Journal of Computer Science and Network Security (IJCSNS), *International Journal of Computer Science and Network Security*, v. 17, n. 3, p. 228, 2017. Citado na página 19.
- ROSE, K.; ELDRIDGE, S.; CHAPIN, L. The internet of things: An overview. The internet society (ISOC), Reston, VA, v. 80, p. 1–50, 2015. Citado na página 10.
- SCHWAB, K. The fourth industrial revolution: what it means, how to respond. 2016. Disponível em: <<https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/>>. Acesso em: 13 mar. 2022. Citado na página 10.
- TURNBULL, J. The Docker Book: Containerization is the new virtualization. [S.l.]: James Turnbull, 2014. Citado na página 19.
- WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. REST in practice: Hypermedia and systems architecture. [S.l.]: "O'Reilly Media, Inc.", 2010. Citado na página 14.