UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Felipe Augusto Bortolini

Sistema de Supervisão e Aquisição de Dados para ETEs e ETAs em Condomínios

Blumenau
2024

Felipe Augusto Bortolini

Sistema de Supervisão e Aquisição de Dados para ETEs e ETAs em Condomínios

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr. Carlos Roberto Moratelli

Blumenau
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Bortolini, Felipe Augusto
Sistema de Supervisão e Aquisição de Dados para ETEs e ETAs em Condomínios / Felipe Augusto Bortolini ; orientador, Carlos Roberto Moratelli, 2024.
116 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. supervisório. 3. protocolo MQTT. 4. Firebase. 5. Flutter. I. Moratelli, Carlos Roberto. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Felipe Augusto Bortolini

Sistema de Supervisão e Aquisição de Dados para ETEs e ETAs em Condomínios

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 27 de novembro de 2024.

Banca Examinadora:

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 02/12/2024 13:32:23-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Guilherme Brasil Pintarelli
Data: 02/12/2024 13:57:22-0300
CPF: ***.523.129-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Guilherme Brasil Pintarelli
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Maiquel de Brito
Data: 02/12/2024 13:55:13-0300
CPF: ***.657.150-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Maiquel de Brito
Universidade Federal de Santa Catarina

AGRADECIMENTOS

A realização deste trabalho é fruto de muitos esforços, apoio e dedicação de várias pessoas que, direta ou indiretamente, contribuíram para a sua conclusão.

Em primeiro lugar, gostaria de expressar minha profunda gratidão à minha noiva, Júlia Pedroso Lusa, por todo o suporte emocional, compreensão e paciência ao longo desta jornada. Sua presença constante e incentivo foram fundamentais em cada etapa deste processo.

Agradeço imensamente à minha família, especialmente aos meus pais, César Augusto Martini Bortolini e Lilian Bortolini, e ao meu irmão, João Vitor Bortolini. A ajuda financeira e emocional de vocês foi essencial para que eu pudesse me concentrar nos estudos e alcançar este objetivo. Ter uma base familiar tão estável e segura me deu a tranquilidade necessária para superar os desafios, sabendo que sempre poderia contar com o apoio de vocês.

Não poderia deixar de agradecer ao meu orientador, Carlos Roberto Moratelli, pela orientação e ensinamentos valiosos. Sua dedicação e disponibilidade foram indispensáveis para a realização deste trabalho. Agradeço também por todas as contribuições e direcionamentos que me guiaram durante a execução deste projeto.

A todos, meu sincero agradecimento.

RESUMO

Este trabalho desenvolve um supervisório voltado ao monitoramento de ETA (Estação de Tratamento de Água) e ETE (Estação de Tratamento de Esgoto) em condomínios prediais. O sistema utiliza sensores conectados a dispositivos ESP32 para monitorar o nível dos tanques e o estado das bombas. A transmissão dos dados ocorre via protocolo MQTT para um broker, sendo processados em um ambiente Node.js hospedado na plataforma Heroku. Os dados são armazenados no banco de dados NoSQL do Cloud Firestore, enquanto funções no Firebase gerenciam alarmes e notificações enviadas aos usuários via Telegram e e-mail. A aplicação Web, desenvolvida em Flutter e hospedada no Firebase Hosting, permite o acompanhamento do sistema. O projeto oferece uma solução eficiente para a gestão de infraestruturas críticas, com geração automática de alertas quando o nível dos tanques está fora dos limites ou em casos de falhas nas bombas. O método adotado integra tecnologias de IoT, computação em nuvem e sistemas de notificações, proporcionando um monitoramento confiável para o gerenciamento remoto de sistemas de abastecimento e tratamento de água e esgoto.

Palavras-chave: supervisório; protocolo MQTT; Firebase; Flutter, ETA/ETE.

ABSTRACT

This work develops a supervisory aimed at monitoring WTS (Water Treatment Station) and STS (Sewage Treatment Station) in building condominiums. The system uses sensors connected to ESP32 devices to monitor tank levels and pump status. Data transmission occurs via the MQTT protocol to a broker, being processed in a Node.js environment hosted on the Heroku platform. The data is stored in the Cloud Firestore NoSQL database, while Firebase Functions manage alarms and notifications sent to users via Telegram and email. The Web application, developed in Flutter and hosted on Firebase Hosting, allows for system tracking. The project offers an efficient solution for managing critical infrastructures, with automatic alert generation when tank levels are outside the limits or in case of pump failures. The adopted method integrates IoT technologies, cloud computing, and notification systems, providing reliable monitoring for the remote management of water supply and sewage treatment systems.

Keywords: supervisory; MQTT protocol; Firebase; Flutter; WTS/STS.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AOT	Ahead Of Time
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CDN	Content Delivery Network
CEP	Complex Event Processing
CLI	Command Line Interface
CSS	Cascading Style Sheets
DBNs	Deep Belief Networks
ETA	Estação de Tratamento de Água
ETE	Estação de Tratamento de Esgoto
GB	Gigabyte
GNSS	Global Navigation Satellite System
GPRS	General Packet Radio Service
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I2C	Inter-Integrated Circuit
IBM	International Business Machines
IoT	Internet of Things
JIT	Just In Time
JSON	JavaScript Object Notation
MB	Megabyte
min	minutos
MQTT	Message Queuing Telemetry Transport
MVC	Model View Controller
NoSQL	Not Only Structured Query Language
NPM	Node Package Manager
PaaS	Platform as a Service
PDF	Portable Document Format
PWA	Progressive Web Apps
QoS	Quality of Service
SaaS	Software as a Service
SCADA	Supervisory Control and Data Acquisition
SMTP	Simple Mail Transfer Protocol
SPI	Serial Peripheral Interface
STS	Sewage Treatment Station

TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver/Transmitter
UASB	Upflow Anaerobic Sludge Blanket
URL	Uniform Resource Locator
WTS	Water Treatment Station

SUMÁRIO

1 INTRODUÇÃO

A urbanização e o crescimento demográfico das últimas décadas têm demandado novos modelos de infraestrutura para o uso e o tratamento eficiente dos recursos hídricos em áreas residenciais (HEIDARI *et al.*, 2021). Nos condomínios, onde a densidade populacional e a concentração de atividades diárias são elevadas, a manutenção contínua e a gestão eficaz dos sistemas de abastecimento de água e esgotamento sanitário são fundamentais para a saúde e o bem-estar dos moradores. Ao mesmo tempo, com o aumento das regulamentações ambientais e o avanço nas tecnologias de automação, torna-se cada vez mais viável e necessário o desenvolvimento de sistemas capazes de monitorar remotamente variáveis críticas (BORAH; KHANAL; SUNDARAVADIVEL, 2024). O nível dos tanques e o estado operacional das bombas, responsáveis pelo abastecimento e pela condução do esgoto, são exemplos disso. Este cenário demanda soluções integradas, automatizadas e inteligentes que permitam gerenciar em tempo real os processos essenciais de saneamento básico nos condomínios.

Na busca por soluções para esses desafios, o uso de sistemas supervisórios, IoT (Internet of Things) e computação em nuvem se destaca por possibilitar o monitoramento em tempo real, além da geração de alertas e notificações automáticas que contribuem para a continuidade dos processos sem a necessidade de intervenção manual constante. No entanto, a implementação dessas tecnologias exige uma abordagem multidisciplinar, envolvendo conhecimentos em engenharia de controle, redes de comunicação, desenvolvimento de software e integração de dispositivos IoT. Essa integração deve considerar a segurança e a confiabilidade do sistema, especialmente em aplicações de saneamento básico, onde qualquer interrupção no fornecimento de dados pode comprometer a saúde pública e o conforto dos moradores.

1.1 CONTEXTUALIZAÇÃO

Nos condomínios, o sistema de abastecimento de água e o sistema de esgotamento sanitário são dois dos principais elementos de infraestrutura, sendo responsáveis, respectivamente, pela distribuição de água potável e pelo tratamento do esgoto gerado (CARVALHO JÚNIOR, 2023). Cada um desses sistemas envolve o uso de tanques de armazenamento e bombas para garantir o fluxo contínuo e adequado. Para a operação segura e eficiente dos sistemas, o monitoramento em tempo real dos níveis dos tanques e do estado das bombas é necessário para evitar problemas, como o desabastecimento ou o transbordamento de esgoto, situações que podem impactar diretamente a qualidade de vida dos moradores e acarretar altos custos de manutenção.

Com o avanço das tecnologias de comunicação e a popularização dos dispositivos IoT, permitiu-se monitorar remotamente e de forma automatizada esses sistemas críticos. A utilização de sensores inteligentes e conectados a plataformas na nuvem permite a

coleta e a análise de dados em tempo real, gerando uma camada de inteligência para o gerenciamento preditivo e preventivo. Assim, a arquitetura de monitoramento que emprega IoT, computação em nuvem e sistemas supervisórios possibilita que gestores tenham uma visão contínua e precisa do estado operacional dos dispositivos instalados nos sistemas de abastecimento e esgotamento.

1.2 MOTIVAÇÃO

O desenvolvimento de um sistema supervisório para o monitoramento de ETA (Estação de Tratamento de Água) e ETE (Estação de Tratamento de Esgoto) em condomínios é motivado pela necessidade de aumentar a confiabilidade e a eficiência na operação desses sistemas, reduzindo o tempo de resposta a incidentes e os custos associados à manutenção. Em cenários convencionais, o monitoramento desses sistemas requer a presença de equipes no local para inspeções periódicas, tornando o processo custoso e pouco eficiente. Com a adoção de tecnologias de IoT e computação em nuvem, é possível automatizar a coleta e o processamento dos dados, viabilizando a criação de alertas automáticos que informam os operadores sobre a necessidade de ações corretivas antes que ocorram falhas críticas.

1.3 OBJETIVOS

O foco principal deste trabalho reside a partir do recebimento dos dados da planta física pelo *broker* MQTT (Message Queuing Telemetry Transport) para tratamento, alimentação do banco de dados e consumo pelo sistema supervisório. Dessa maneira o objetivo principal do trabalho é: desenvolver um sistema de supervisão Web e aquisição de dados, destinado aos síndicos de condomínios, para monitoramento de ETAs e ETEs.

1.3.1 Objetivos Específicos

- Desenvolver um sistema de processamento de dados com um cliente assinante no tópico MQTT, no qual serão publicadas as informações dos dispositivos no formato JSON (JavaScript Object Notation), armazenando-as no banco de dados Firestore do Firebase.
- Desenvolver um sistema supervisório Web para visualização dos dados monitorados, proporcionando ao usuário a possibilidade de acompanhar o estado das bombas e níveis dos tanques.
- Implementar um sistema de alarmes com notificações aos usuários via Telegram e e-mail, disparadas quando ocorrerem falhas nos dispositivos monitorados ou variações abaixo de níveis críticos nos tanques.
- Realizar testes e validações do sistema supervisório, com simulações de dados recebidos via MQTT.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho é estruturado em oito capítulos que descrevem de forma sequencial os conceitos, metodologias, implementação e resultados relacionados ao desenvolvimento de um sistema supervisório para monitoramento de ETAs e ETEs em condomínios residenciais.

- **Capítulo 1** - Introdução: Este capítulo apresenta uma contextualização sobre a importância do monitoramento automatizado dos sistemas de saneamento básico em condomínios, além da motivação, objetivos e estrutura do trabalho.
- **Capítulo 2** - Saneamento Básico em Condomínios Prediais: São abordados os sistemas de saneamento aplicados aos condomínios, com foco nas ETA (Estação de Tratamento de Água) e ETE (Estação de Tratamento de Esgoto), descrevendo suas funções, componentes como bombas e reservatórios, e os desafios operacionais associados a cada um.
- **Capítulo 3** - Fundamentação Teórica: Neste capítulo, são apresentados os conceitos teóricos e as tecnologias utilizadas no desenvolvimento do sistema de monitoramento, incluindo o uso de IoT para monitoramento remoto, microcontroladores ESP32, protocolo MQTT, Firebase, Flutter, Node.js e arquiteturas de software como o padrão MVC e a Arquitetura Limpa.
- **Capítulo 4** - Arquitetura e Escopo do Projeto: No capítulo se descreve a arquitetura do sistema proposto, com detalhes sobre os componentes e processos de coleta de dados, incluindo sensores e controladores ESP32, protocolo MQTT, e o papel do Node-RED na estrutura do sistema.
- **Capítulo 5** - Implementação do *Backend* para Monitoramento de ETAs e ETEs: O capítulo detalha a implementação do *backend*, abordando o uso do *Dyno Worker* no Heroku, o banco de dados Firestore e sua estrutura de coleções específicas para alarmes, histórico e usuários. São descritas também as Firebase Functions para gerenciamento de alarmes e a integração com sistemas de notificação via Telegram e e-mail.
- **Capítulo 6** - Aplicação *Frontend* com Flutter: Este capítulo apresenta a estrutura e os padrões de projeto da aplicação *frontend*, desenvolvida com o framework Flutter. São discutidos aspectos como a arquitetura de camadas, gerenciamento de estado com GetX, navegação, integração com Firebase e processos de hospedagem no Firebase Hosting.
- **Capítulo 7** - Resultados e Discussão: Neste capítulo, são avaliados os resultados obtidos com a implementação do sistema, incluindo a análise de desempenho da comunicação via MQTT, armazenamento de dados, efetividade dos alarmes

e notificações, e uma descrição das principais páginas da aplicação, como *login*, *home*, análises, histórico de dados, alarmes e relatórios.

- **Capítulo 8** - Conclusão: O último capítulo apresenta as conclusões do trabalho, ressaltando as contribuições do sistema para o monitoramento de ETAs e ETEs em condomínios e sugerindo direções para trabalhos futuros que possam aprimorar e expandir as funcionalidades do sistema.

Esta organização visa fornecer uma visão clara e progressiva sobre os aspectos técnicos e práticos do projeto, facilitando o entendimento das etapas envolvidas no desenvolvimento e avaliação do sistema supervisório.

2 SANEAMENTO BÁSICO EM CONDOMÍNIOS PREDIAIS

O saneamento básico é uma infraestrutura essencial para garantir a saúde pública e a qualidade de vida dos moradores. Em condomínios prediais, dois dos principais serviços de saneamento incluem o abastecimento de água potável e o tratamento de esgoto, ambos fundamentais para suprir necessidades básicas, prevenir doenças e evitar danos ambientais. A ausência ou falhas nesses serviços podem comprometer o bem-estar dos residentes e gerar uma série de problemas, como falta de água ou mau cheiro, afetando diretamente a qualidade de vida dos moradores (LEONETI; PRADO; BORGES DE OLIVEIRA, 2011; MAIA *et al.*, 2023).

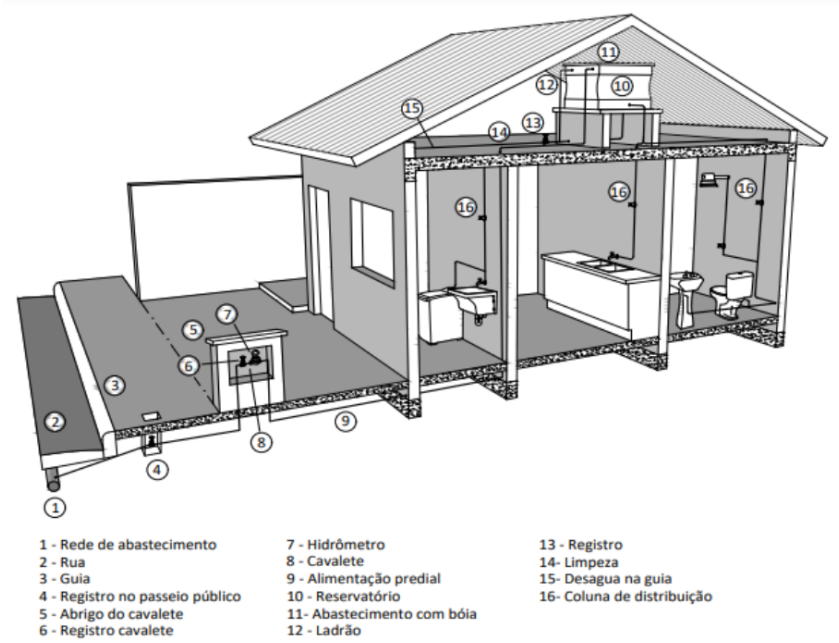
2.1 ETA (Estação de Tratamento de Água)

Os sistemas de abastecimento de água em edificações são projetados para assegurar a distribuição eficiente e contínua de água nos pontos de uso. Esses sistemas podem ser classificados em três categorias principais: direto, indireto e misto. A escolha de um sistema adequado depende de diversos fatores, como a pressão disponível na rede pública, as características da edificação e as exigências de consumo.

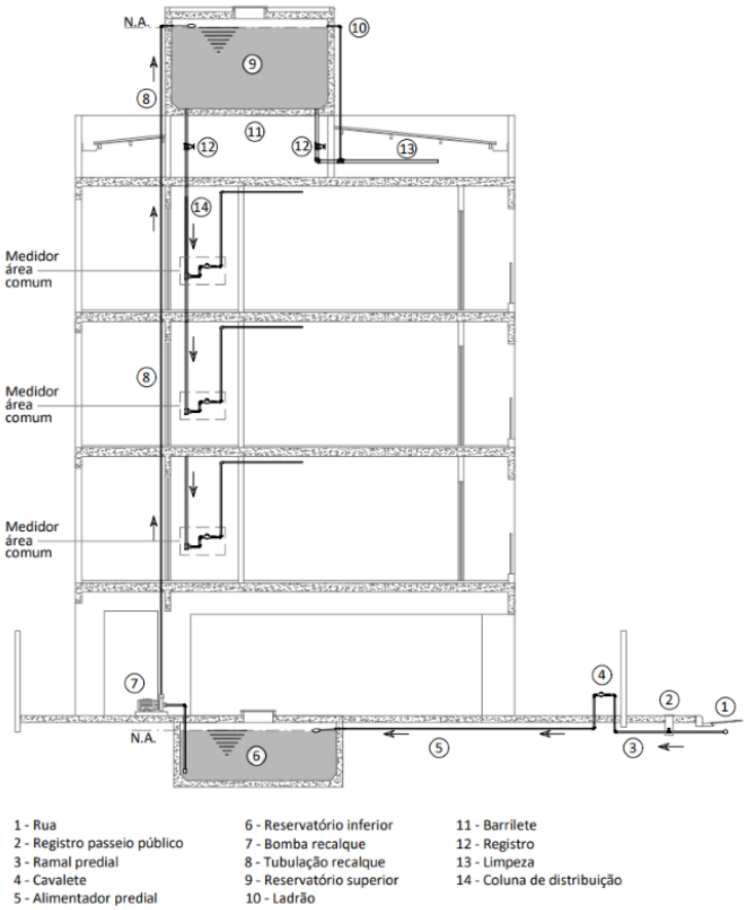
No sistema de abastecimento direto, a água é fornecida diretamente da rede pública aos pontos de utilização da edificação, sem passar por reservatórios intermediários. Essa configuração é mais comum em locais onde a pressão e o fornecimento de água são suficientes para atender a demanda da edificação sem a necessidade de armazenamento, não sendo recomendado em muitas regiões brasileiras que não atendem a esses requisitos (CARVALHO JÚNIOR, 2023). Segundo Reali *et al.* (2002), o sistema indireto de abastecimento de água predial se divide em duas possíveis implementações: por gravidade e hidropneumático.

Implementações por gravidade utilizam reservatórios localizados na parte superior das edificações, garantindo a distribuição da água por meio da pressão natural causada pela diferença de altura. Nessa implementação, podem ser encontrados sistemas sem bombeamento, exemplificado na Figura 1a, nos quais a água é enviada diretamente ao reservatório superior pela própria pressão da rede pública; e sistemas com bombeamento, exemplificado na Figura 1b, nos quais a água é inicialmente armazenada em um reservatório inferior e, em seguida, bombeada até o reservatório superior (CARVALHO JÚNIOR, 2023).

Figura 1 – Sistema de abastecimento de água indireto por gravidade.



(a) Sem bombeamento.



(b) Com bombeamento.

Fonte: Carvalho Júnior (2023).

Já o sistema hidropneumático utiliza um conjunto de bomba, injetor de ar e tanque de pressão para fornecer água. A bomba injeta a água no tanque de pressão, comprimindo o ar presente, gerando um colchão de ar que mantém a pressão constante até que o sistema precise ser acionado novamente (CARVALHO JÚNIOR, 2023).

Por fim, o sistema misto combina características dos dois anteriores. Nele, parte da água é fornecida diretamente da rede pública para determinados pontos de uso, como torneiras externas, enquanto o restante é alimentado por um reservatório superior. Esse sistema é vantajoso em residências unifamiliares ou pequenas edificações, onde a pressão da rede pública pode ser suficiente para atender a alguns pontos, mas um reservatório ainda é necessário para garantir a distribuição uniforme em toda a edificação (CARVALHO JÚNIOR, 2023).

Se tratando de sistemas indiretos de abastecimento de água predial por gravidade com bombeamento, esses sistemas dependem de uma gestão eficiente para evitar a interrupção no fornecimento de água aos residentes. Problemas podem ocorrer por falhas nos equipamentos, como o desgaste das bombas ou vazamentos nos reservatórios.

2.1.1 Bombas em ETAs

Nas ETAs, as bombas hidráulicas são responsáveis pelo transporte da água através das diversas etapas do processo de tratamento, desde a captação da água bruta até sua distribuição após o tratamento. Essas bombas podem ser centrífugas, de fluxo axial ou submersíveis, dependendo das características da estação e da vazão necessária. Elas garantem o fornecimento contínuo de água bruta até o início do tratamento e sua posterior distribuição após o processo, assegurando que a água atenda aos padrões de potabilidade (CHEIS, 2015).

No entanto, as bombas em ETAs estão sujeitas a falhas operacionais que podem comprometer a eficiência do sistema. Entre os problemas mais comuns, estão o desgaste de rotores, cavitação e sobrecarga elétrica. A cavitação, em especial, ocorre quando há formação de bolhas de vapor na bomba devido a uma queda na pressão interna, o que pode danificar os componentes e reduzir a vida útil do equipamento. Outro problema comum é o desgaste mecânico dos rotores, que pode diminuir a eficiência do bombeamento ao longo do tempo (CHEIS, 2015).

2.1.2 Reservatórios em ETAs

Nos sistemas de abastecimento de água em ETAs, os reservatórios são projetados para armazenar água tratada temporariamente antes de sua distribuição. O dimensionamento adequado desses reservatórios depende de fatores como o consumo diário e as flutuações na demanda, que podem ocorrer ao longo do dia. Níveis baixos de água nos reservatórios podem causar interrupções no fornecimento, enquanto níveis muito altos

podem sobrecarregar os sistemas de bombeamento, exigindo um controle rigoroso dos volumes armazenados (SCHMIDT, 2011).

Para monitorar o volume de água e prevenir falhas, sensores de nível, como ultrassônicos, capacitivos ou de pressão, são instalados em pontos estratégicos dos reservatórios. Esses sensores permitem o monitoramento contínuo da quantidade de água disponível e alertam para possíveis anomalias, como a presença de vazamentos ou queda no nível de água (COELHO; GLÓRIA; SEBASTIÃO, 2020; MANNSFELD *et al.*, 2010).

Vazamentos em reservatórios podem ocorrer devido ao desgaste estrutural, falhas nas juntas de vedação ou até mesmo fissuras. Esses problemas podem levar à perda de água já tratada e aumentar os custos de operação. Tecnologias de monitoramento e inspeções regulares são importantes para detectar vazamentos e realizar manutenções corretivas rapidamente, evitando maiores prejuízos e interrupções no abastecimento.

Conforme observado no estudo de Schmidt (2011), o dimensionamento correto do volume útil dos reservatórios em edificações multifamiliares é essencial para evitar desperdícios e garantir um abastecimento aperfeiçoado. Além disso, o estudo ressalta a importância de tecnologias que monitoram e ajustam automaticamente os níveis de água, garantindo uma operação mais eficiente dos sistemas de abastecimento de água.

2.2 ETE (Estação de Tratamento de Esgoto)

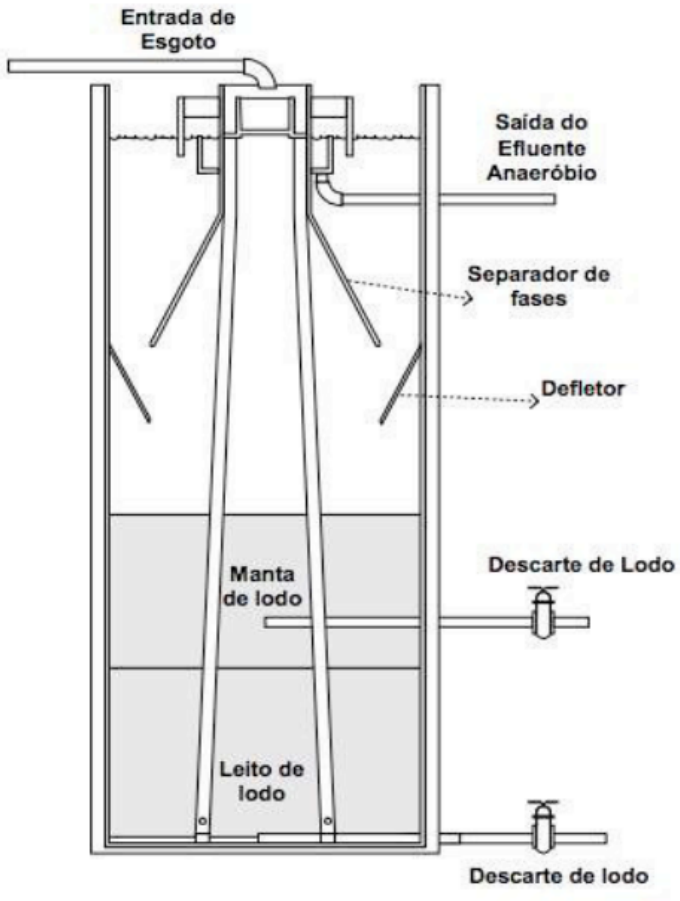
O tratamento de esgoto em condomínios prediais tem sido amplamente utilizado, especialmente em regiões onde a infraestrutura pública de saneamento é limitada ou insuficiente. Os diferentes processos de tratamento de esgoto podem ser classificados como aeróbios, anaeróbios ou mistos, sendo cada um deles apropriado para diferentes condições de operação e objetivos de tratamento. Esses sistemas são frequentemente implementados em estações de tratamento compactas, que podem oferecer soluções práticas e eficientes para empreendimentos de pequeno e médio porte, como condomínios prediais.

No processo aeróbio, microrganismos que utilizam oxigênio como agente oxidante degradam a matéria orgânica presente no esgoto, resultando na sua estabilização. Esse tipo de processo é amplamente utilizado devido à sua alta eficiência na remoção de matéria orgânica, sendo aplicável em diversos sistemas, como lodos ativados e filtros biológicos. Um dos principais sistemas aeróbios é o de lodos ativados, onde o oxigênio é constantemente suprido por meio de aeração forçada, promovendo a degradação da matéria orgânica de maneira eficaz. No entanto, o consumo energético desse tipo de sistema é elevado, uma vez que a manutenção da aeração é fundamental para a eficiência do processo (BARBOSA, J. N., 2009).

Por outro lado, os processos anaeróbios operam na ausência de oxigênio, utilizando microrganismos que convertem a matéria orgânica em biogás (principalmente metano e dióxido de carbono), que pode ser aproveitado para a geração de energia. Esses sistemas são conhecidos por sua eficiência no tratamento de esgoto com altas concentrações de

matéria orgânica e pelo baixo consumo energético, já que não requerem aeração forçada. De acordo com Lima (2017), o UASB (Upflow Anaerobic Sludge Blanket), ilustrado na Figura 2, é um exemplo amplamente utilizado em estações de tratamento compactas devido à sua eficiência na redução da carga orgânica e à produção de biogás como subproduto. Entretanto, uma das desvantagens desse processo é a menor eficiência na remoção de nutrientes, como nitrogênio e fósforo, quando comparado aos sistemas aeróbios. Além disso, sistemas anaeróbios podem apresentar emissão de odores, especialmente em áreas urbanas, o que exige um controle operacional rigoroso (BARBOSA, J. N., 2009; LIMA, 2017).

Figura 2 – Reator UASB.



Fonte: Lima (2017).

As estações de tratamento de esgoto modulares são ideais para condomínios, ao permitirem ajustes conforme a variação da demanda. Esses sistemas integram processos aeróbios e anaeróbios em módulos, e incluem bombas para o transporte dos efluentes entre os módulos e tanques para o armazenamento durante o processo de tratamento. A modularidade facilita tanto a operação quanto a manutenção, permitindo a substituição ou ampliação de módulos conforme necessário, sem comprometer o funcionamento geral da estação (LIMA, 2017).

Entretanto, alguns desafios operacionais associados às estações compactas de tratamento de esgoto envolvem, principalmente, o funcionamento das bombas e o controle dos níveis dos tanques.

2.2.1 Bombas em ETEs

Nas ETEs, as bombas têm a função de garantir o transporte adequado do efluente através das diferentes fases do processo. Elas precisam lidar com as características específicas do esgoto, como a presença de sólidos suspensos e líquidos de alta densidade. Dependendo da demanda da ETE, diferentes tipos de bombas, como as submersíveis, centrífugas ou de lóbulos, podem ser utilizadas, cada uma projetada para evitar bloqueios e garantir um bombeamento contínuo e eficiente (CHEIS, 2015).

As bombas, responsáveis pelo transporte de esgoto entre os diferentes módulos de tratamento, são suscetíveis às mesmas falhas especificadas na Seção 2.1.1 para ETAs.

2.2.2 Tanques em ETEs

Nas ETEs compactas modulares, os tanques são responsáveis pelo armazenamento e pela equalização dos efluentes ao longo das etapas de tratamento. Como esses sistemas são projetados para ocupar pouco espaço, o controle do nível dos tanques influencia diretamente na eficiência do tratamento e na continuidade do processo, evitando interrupções.

Em estações compactas, manter o nível adequado nos tanques ajuda a minimizar problemas com odores, uma vez que níveis muito baixos podem expor o esgoto ao ar, favorecendo a decomposição anaeróbia da matéria orgânica, que libera gases como o sulfeto de hidrogênio (H_2S), responsável por odores desagradáveis (SILVA, A. B. da, 2007). Isso é especialmente relevante em condomínios prediais, onde o controle de odores é importante para evitar incômodos para os moradores.

Além disso, com o nível baixo de água, o lodo no fundo do tanque pode não ser adequadamente diluído ou movimentado. À medida que esse material fermenta, pode levar à formação de crostas de material orgânico e maior liberação de odores. O acúmulo excessivo desse material pode sobrecarregar as bombas e causar desgastes mecânicos, resultando em manutenções frequentes e diminuindo a eficiência geral do sistema. Monitorar o nível dos tanques e realizar manutenções preventivas no sistema de bombeamento são medidas comuns para evitar esses problemas e garantir que o efluente seja tratado de maneira contínua e eficaz (FECAROTTA; MARTINO; MORANI, 2019).

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como foco introduzir os conceitos associados ao projeto desenvolvido nesse trabalho, concentrando-se em conceitos importantes sobre as tecnologias que podem ser utilizadas para a criação de um sistema de monitoramento remoto com um sistema supervisório.

3.1 MONITORAMENTO REMOTO COM IoT

O uso de tecnologias de IoT (Internet of Things) em sistemas de monitoramento remoto tem se expandido, especialmente em contextos industriais e de automação predial. Tecnologias como essa, permitem o monitoramento contínuo de equipamentos, gerando dados que podem ser enviados para armazenamento na nuvem ou em servidores locais para análise. A comunicação entre dispositivos IoT é realizada por diferentes protocolos de rede, como HTTP (Hypertext Transfer Protocol), MQTT e SPI (Serial Peripheral Interface), dependendo das necessidades de desempenho, confiabilidade e disponibilidade de largura de banda (SILVA, R. C. e, 2022; THOMÉ, 2023).

Em aplicações de monitoramento de sensores, os dispositivos IoT são equipados com sensores para coletar dados, como temperatura, umidade, pressão ou nível de líquidos. Esses dados são processados por microcontroladores que podem enviar informações para um servidor centralizado. No ambiente residencial ou industrial, os dados podem ser visualizados por meio de interfaces gráficas, facilitando a manutenção preditiva e evitando falhas operacionais. Em particular, o uso de plataformas em nuvem, como o Firebase ou AWS IoT, permite o armazenamento seguro e eficiente dos dados coletados, além de facilitar o escalonamento do sistema para grandes volumes de dispositivos (THOMÉ, 2023).

3.1.1 ESP32

O ESP32 é um microcontrolador popular em aplicações de IoT devido à sua versatilidade e desempenho. Ele integra conectividade Wi-Fi e Bluetooth, oferecendo suporte a múltiplos protocolos de comunicação como SPI, I2C (Inter-Integrated Circuit), UART (Universal Asynchronous Receiver/Transmitter) e MQTT, sendo ideal para monitoramento de sensores e controle de dispositivos remotamente (SILVA, R. C. e, 2022). O ESP32 também possui suporte para criptografia de dados, melhorando a segurança na comunicação, especialmente em ambientes industriais.

Além disso, o ESP32 permite a integração com plataformas de armazenamento em nuvem, como Firebase e AWS IoT graças à sua conectividade Wi-Fi integrada e suporte a protocolos como HTTP, MQTT e *WebSocket*. Embora não possua recursos específicos para essas plataformas, sua capacidade de comunicação eficiente por meio de redes Wi-Fi

e a compatibilidade com bibliotecas disponíveis para esses serviços tornam a integração mais simples. Um exemplo comum é a implementação de sistemas de monitoramento em tempo real, onde o ESP32 coleta dados de sensores, como temperatura ou nível de água, e envia esses dados para um servidor por meio de uma rede Wi-Fi (DINSE, 2022).

3.2 NODE-RED

O Node-RED é uma ferramenta de programação visual projetada para facilitar a integração entre dispositivos de IoT, APIs e serviços online. Criado inicialmente pela IBM, ele permite que fluxos de trabalho sejam desenvolvidos por meio de uma interface gráfica de fácil utilização, onde componentes, chamados de "nós", são conectados para realizar tarefas específicas. Cada nó pode representar uma ação, como enviar ou receber dados, processar informações ou até mesmo controlar dispositivos físicos. Os nós, quando conectados, formam "fluxos" que determinam o comportamento da aplicação (FOUNDATION, 2023).

Uma das principais características do Node-RED é sua flexibilidade e facilidade de integração com uma ampla gama de protocolos e serviços, como MQTT, HTTP e WebSockets. Isso o torna uma escolha popular para soluções de automação residencial e industrial, especialmente quando há a necessidade de monitoramento e controle em tempo real de dispositivos e sensores conectados (MARTINS, 2019).

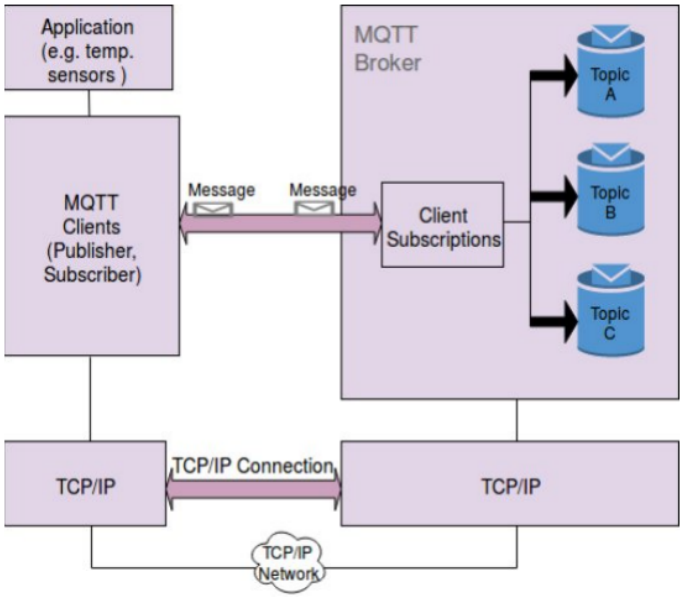
3.3 MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)

O MQTT é um protocolo de comunicação leve projetado para a troca eficiente de dados entre dispositivos com recursos limitados e em redes com largura de banda restrita ou com alta latência. Desenvolvido pela IBM (International Business Machines) em 1999, o protocolo foi criado para suportar ambientes com conectividade intermitente, sendo amplamente utilizado em sistemas de IoT devido à sua arquitetura simplificada e eficiente, baseada no modelo *publish/subscribe*. Nesse modelo, um *broker* atua como intermediário entre publicadores e assinantes, gerenciando a troca de mensagens de forma assíncrona (SONI; MAKWANA, 2017).

A arquitetura do MQTT é composta por três elementos principais: o *broker*, os clientes e os tópicos. Os clientes utilizam uma rede TCP/IP (Transmission Control Protocol/Internet Protocol) para se conectar ao *broker*, responsável por receber e distribuir mensagens aos assinantes que subscreveram aos tópicos apropriados. Os publicadores são clientes que enviam mensagens para tópicos específicos e os assinantes são clientes que recebem essas mensagens com base em sua assinatura nos tópicos (OASIS, 2019).

A Figura 3 ilustra o funcionamento básico dessa arquitetura, destacando o fluxo de mensagens entre os publicadores, o *broker* e os assinantes.

Figura 3 – Arquitetura MQTT.



Fonte: SONI e MAKWANA (2017).

O MQTT oferece três níveis de QoS (Quality of Service), apresentados no Quadro 1, os quais são escolhidos conforme a criticidade da aplicação e as condições da rede, permitindo flexibilidade no controle da confiabilidade e latência da comunicação.

Quadro 1 – Níveis de QoS do MQTT.

QoS 0 (At most once)	As mensagens são enviadas sem garantias de entrega, sendo este o nível mais rápido e eficiente em termos de desempenho, porém sem confiabilidade garantida.
QoS 1 (At least once)	As mensagens são enviadas pelo menos uma vez, com a possibilidade de ocorrerem duplicações.
QoS 2 (Exactly once)	As mensagens são garantidamente entregues uma única vez, com um protocolo de *handshake* de quatro vias que assegura a integridade da transmissão, mesmo em condições de rede adversas.

Fonte: SONI e MAKWANA (2017).

Outra funcionalidade importante do MQTT é o mecanismo de retenção de mensagens. O *broker* pode reter mensagens publicadas em um tópico para novos assinantes, garantindo que as mensagens sejam entregues mesmo que o assinante não estivesse conectado no momento da publicação inicial. Adicionalmente, o MQTT suporta sessões persistentes, nas quais as mensagens são armazenadas e entregues posteriormente aos assinantes quando eles reconectam ao *broker* (JANA, 2022).

Em termos de segurança, o MQTT implementa mecanismos como autenticação de clientes e a criptografia dos dados transmitidos. A criptografia é realizada por meio do TLS (Transport Layer Security), geralmente nas portas 8883 ou 443, protegendo os dados em

trânsito e garantindo a integridade e a confidencialidade das mensagens trocadas entre publicadores, *broker*s e assinantes. Entretanto, deve-se considerar o impacto no desempenho dos dispositivos, especialmente em redes com baixa capacidade de processamento, onde o uso da criptografia pode aumentar significativamente o consumo de recursos. Nesses casos, a conexão sem criptografia pode ser utilizada nas portas 1883 ou 80 (JANA, 2022; SONI; MAKWANA, 2017).

O protocolo MQTT é uma solução eficaz para a comunicação entre dispositivos de IoT, especialmente em cenários que exigem alta eficiência no consumo de energia e largura de banda. Suas funcionalidades, como os diferentes níveis de qualidade de serviço, retenção de mensagens e mecanismos de segurança, são flexíveis e adaptáveis às diversas necessidades de sistemas distribuídos. Dessa forma, ele pode ser aplicado em diversas áreas, que vão desde o monitoramento remoto até a automação de processos.

3.4 FIREBASE

O Firebase é uma plataforma oferecida pelo Google que fornece diversas ferramentas e serviços para desenvolvimento de aplicativos móveis e Web, permitindo a integração de funcionalidades como autenticação de usuários, banco de dados em tempo real, hospedagem, entre outros. Como uma plataforma de BaaS (Backend as a Service), o Firebase facilita o gerenciamento do *backend* da aplicação sem a necessidade de manutenção de servidores dedicados, tornando o desenvolvimento mais ágil e escalável. Esta seção explora os principais serviços utilizados no projeto, incluindo Authentication, Firestore, Realtime Database e Hosting.

3.4.1 Authentication

O Authentication é um serviço completo de autenticação que permite a implementação de login por meio de diversos provedores, como Google, Facebook, e-mail e senha, além de autenticação anônima. Entre as vantagens desse serviço, destaca-se a facilidade de uso e a segurança reforçada no gerenciamento de credenciais e *tokens* de autenticação (THOMÉ, 2023; GOOGLE, 2024b).

3.4.2 Cloud Firestore

O Cloud Firestore é um banco de dados NoSQL (Not Only Structured Query Language) que utiliza uma estrutura baseada em documentos. Diferentemente dos bancos de dados relacionais, que organizam os dados em tabelas, o Firestore armazena os dados em coleções de documentos. Cada documento consiste em um conjunto de pares chave-valor, semelhante à estrutura de objetos JSON. Além disso, as coleções podem conter subcoleções e outros documentos, criando uma estrutura hierárquica e flexível para o armazenamento de dados (SAM; CHI; SILVA, S., 2023).

Entre suas principais características, destaca-se a capacidade de sincronização em tempo real, garantindo que os dados sejam visualizados instantaneamente em todos os dispositivos conectados. Além disso, o Firestore oferece suporte a consultas avançadas, permitindo filtros, ordenações e combinações de múltiplas condições sem comprometer o desempenho. Ele também possui escalabilidade automática, sendo gerenciar grandes volumes de dados sem a necessidade de configuração manual (GOOGLE, 2024b).

3.4.3 Cloud Functions

O Firebase Functions é um serviço *serverless* que permite a execução de código *backend* em resposta a eventos dos serviços do Firebase, como atualizações no Firestore ou requisições HTTPS (Hypertext Transfer Protocol Secure). Esse serviço elimina a necessidade de gerenciar servidores, já que o Firebase cuida automaticamente da infraestrutura e escalabilidade conforme a demanda (ALI; DAUWED, 2022; GOOGLE, 2024b).

As funções são escritas em JavaScript ou TypeScript e são executadas em um ambiente controlado na plataforma de serviços em nuvem para armazenamento Google Cloud, processamento e desenvolvimento de aplicações (GOOGLE, 2024b).

3.4.4 Realtime Database

O Realtime Database é um banco de dados NoSQL que oferece sincronização de dados em tempo real entre os clientes, facilitando a comunicação instantânea em aplicações que exigem atualizações constantes. Ele utiliza uma estrutura de dados baseada em uma árvore JSON (BHAGAT *et al.*, 2022), o que permite que as informações sejam armazenadas e organizadas eficientemente em hierarquias simples (GOOGLE, 2024b).

Entre as principais vantagens do Realtime Database estão a persistência offline dos dados, permitindo que os clientes continuem a ter acesso e a enviar informações mesmo sem conexão, e a escalabilidade automática, suportando até 200 mil usuários conectados simultaneamente. Caso seja necessário acomodar mais usuários, é possível adicionar múltiplos bancos de dados ao projeto para aumentar essa capacidade manualmente (THOMÉ, 2023; GOOGLE, 2024b).

3.4.5 Hosting

O Firebase Hosting é um serviço que oferece hospedagem rápida e segura para aplicativos Web, facilitando a implantação de conteúdo estático, como HTML (HyperText Markup Language), CSS (Cascading Style Sheets) e JavaScript, além de suportar PWA (Progressive Web Apps). Um dos principais benefícios do Firebase Hosting é a facilidade de implantação, permitindo que desenvolvedores distribuam suas aplicaçõeseficientementee. A infraestrutura global e a CDN (Content Delivery Network) do Firebase garantem que

as aplicações hospedadas sejam acessíveis rapidamente em qualquer parte do mundo, proporcionando escalabilidade e baixa latência (GOOGLE, 2024b).

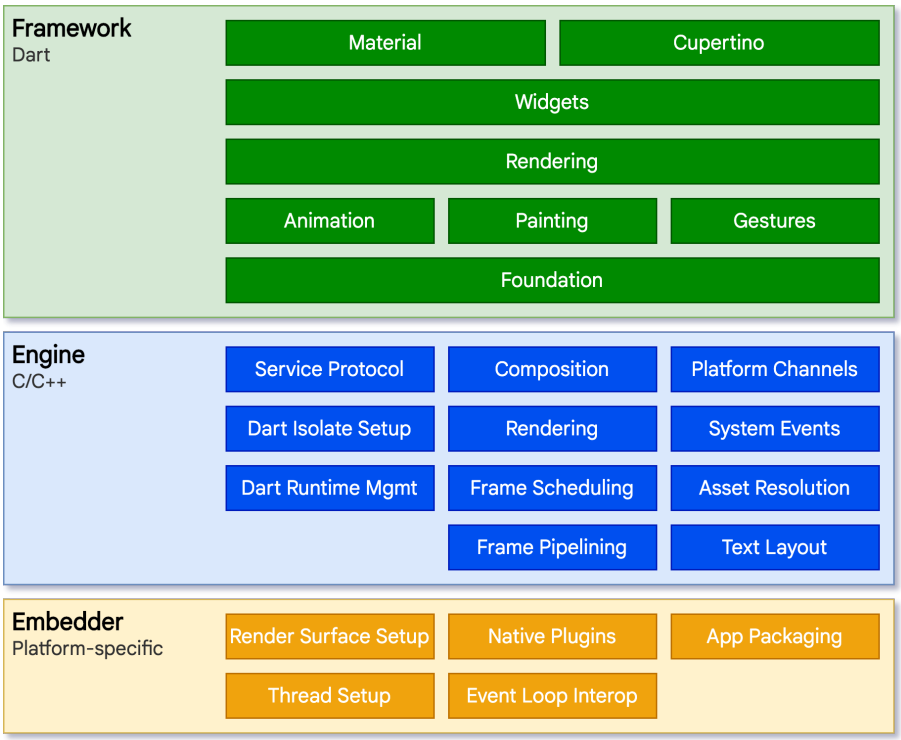
3.5 FLUTTER

O Flutter é um framework *open-source* desenvolvido pelo Google que permite a criação de aplicativos nativos de alto desempenho para diversas plataformas, como Android, iOS, Web e desktop, utilizando uma única base de código. Desde seu lançamento em 2017, o Flutter tem se consolidado como uma das ferramentas mais populares para desenvolvimento de aplicativos devido à sua flexibilidade, eficiência e comunidade ativa de desenvolvedores.

3.5.1 Arquitetura

A arquitetura do Flutter é composta por três camadas: Framework, Engine e Embedder, que trabalham juntas para permitir o desenvolvimento de aplicativos multiplataforma. A Figura 4 ilustra essa arquitetura.

Figura 4 – Camadas da arquitetura do Flutter.



Fonte: Google (2024c).

O Framework, escrito na linguagem de programação Dart, é a camada usada diretamente pelos desenvolvedores. Ele é baseado em uma estrutura de widgets, ou seja, blocos de construção da interface gráfica, permitindo a criação de interfaces multiplataforma, utilizando os estilos Material Design e Cupertino. Além disso, a camada de *widgets* inclui

subsistemas responsáveis pela renderização, animações e interações de toque, oferecendo um ambiente flexível para criar a interface de usuário (GOOGLE, 2024c).

A Engine, implementada em C++, é responsável por gerenciar a renderização gráfica usando a biblioteca Skia e a execução do código Dart. Ela também permite a comunicação entre o código Dart e as APIs nativas de cada sistema operacional, por meio dos Platform Channels, garantindo que os aplicativos possam acessar recursos específicos como sensores e câmeras, mantendo o desempenho adequada (GOOGLE, 2024c).

A camada de Embedder adapta o Flutter às diferentes plataformas, como Android, iOS, Web e desktop, gerenciando a renderização em cada sistema e capturando eventos como toques e cliques. O Embedder também cuida da integração do aplicativo com o sistema operacional e da sua distribuição, garantindo que o Flutter possa funcionar de maneira consistente em diferentes ambientes com uma única base de código (GOOGLE, 2024c).

3.5.2 Vantagens do Flutter

Um dos principais diferenciais do Flutter em relação a outras plataformas de desenvolvimento é sua capacidade de recarregamento rápido, ou *hot reload*, que permite ao desenvolvedor visualizar as alterações no código em tempo real, sem a necessidade de recompilar toda a aplicação. Essa funcionalidade acelera significativamente o processo de desenvolvimento, tornando o Flutter uma ferramenta eficiente para iteração rápida (THOMÉ, 2023).

Além disso, o Flutter possui renderização própria, não dependendo de componentes nativos do sistema operacional para desenhar sua interface, ao contrário de outras ferramentas de desenvolvimento multiplataforma. Em vez disso, ele utiliza sua própria *engine* de renderização, o que permite manter uma consistência visual e bom desempenho entre diferentes plataformas (THOMÉ, 2023).

3.5.3 Programação com Dart

O Flutter utiliza a linguagem Dart, também desenvolvida pelo Google, que combina paradigmas de programação orientada a objetos e funcional. O Dart é aprimorado para desenvolvimento de interfaces gráficas e possui suporte tanto para compilação JIT (Just In Time), usada no desenvolvimento com o *hot reload*, quanto para AOT (Ahead Of Time), usada na produção, garantindo inicialização rápida e desempenho consistente (GOOGLE, 2024a).

3.5.4 Pacotes

Os pacotes são uma das principais maneiras de reutilizar código no Flutter, facilitando a adição de funcionalidades específicas sem a necessidade de implementá-las do

zero. Um pacote Flutter contém um conjunto de bibliotecas que fornecem funcionalidades pré-desenvolvidas para aplicações, como gerenciamento de estados, manipulação de dados, animações, entre outros. Esses pacotes são distribuídos por meio do pub.dev, que funciona como um repositório central para pacotes Dart e Flutter (ALBERTO, 2024).

Um dos grandes benefícios do uso de pacotes é a simplificação no desenvolvimento. Ao utilizar pacotes, os desenvolvedores podem adicionar funcionalidades específicas de maneira rápida e eficiente, sem a necessidade de lidar com a complexidade interna dessas funcionalidades. Além disso, a comunidade Flutter é ativa, contribuindo regularmente com pacotes constantemente mantidos e atualizados, resultando em um ecossistema robusto e em constante evolução (ALBERTO, 2024).

Essas funcionalidades tornam o Flutter uma plataforma altamente flexível e eficiente, auxiliando os desenvolvedores a construir aplicações complexas de forma mais ágil e eficaz (ALBERTO, 2024).

3.6 GETX

O GetX é um pacote para Flutter que oferece ferramentas para gerenciamento de estado, injeção de dependências e navegação de forma simples e eficiente (GETX, 2024). Devido à sua abordagem leve e minimalista, o GetX tem sido amplamente utilizado por desenvolvedores, ao exigir menos código e reduzir a complexidade em comparação a outras bibliotecas tradicionais, como o *Provider* ou o *Bloc*.

Uma das principais características do GetX é a integração de três funcionalidades essenciais em uma única solução: gerenciamento de estado, injeção de dependências e roteamento. O gerenciamento de estado no GetX é baseado em observadores reativos, o que permite que a interface do usuário seja atualizada automaticamente sempre que os dados monitorados forem alterados, simplificando o processo de desenvolvimento. Além disso, o GetX facilita a injeção de dependências, permitindo o gerenciamento automático de instâncias por meio do conceito de *lazy loading*, no qual os objetos são criados apenas quando necessários, otimizando o desempenho da aplicação (GETX, 2024).

O gerenciamento de rotas também é simplificado no GetX, oferecendo uma forma mais direta de navegação entre as telas da aplicação. O pacote suporta a passagem de parâmetros entre rotas, facilitando a transição entre as diferentes interfaces e o controle da navegação (GETX, 2024).

Devido à sua simplicidade e flexibilidade, o GetX é uma ferramenta eficaz para o desenvolvimento de aplicações Flutter modulares e desacopladas.

3.7 HEROKU

O Heroku é uma PaaS (Platform as a Service) que simplifica o processo de desenvolvimento, implantação e escalabilidade de aplicações Web. A principal vantagem do

Heroku é permitir que os desenvolvedores concentrem seus esforços exclusivamente no desenvolvimento de código, enquanto a plataforma gerencia automaticamente elementos complexos, como o provisionamento de servidores, balanceamento de carga e escalabilidade. Essa solução é altamente versátil, compatível com diversas linguagens de programação populares, como Node.js, Ruby, Python e Java, o que a torna apropriada para uma ampla variedade de projetos (SILVA, L. F. da *et al.*, 2017).

3.7.1 *Dynos Workers*

No Heroku, as aplicações são executadas dentro de contêineres virtuais conhecidos como *Dynos*. Esses *Dynos* são a unidade de execução e escalabilidade da plataforma. Existem diferentes tipos de *Dynos* para diferentes propósitos: os *Web Dynos* lidam com requisições HTTP, enquanto os *Workers Dynos* processam tarefas em segundo plano, como processamento de filas de mensagens, execução de algoritmos ou outras operações que não exigem interação imediata com o usuário (SILVA, L. F. da *et al.*, 2017).

Os *Workers Dynos* realizam o processamento de tarefas assíncronas e operações em segundo plano. Eles permitem que tarefas como envio de e-mails, processamento de dados ou execução de algoritmos complexos ocorram sem comprometer a desempenho da aplicação principal. Ferramentas e frameworks como Celery, Sidekiq e RabbitMQ são frequentemente utilizados para gerenciar essas filas de tarefas, distribuindo-as para serem processadas pelos *Workers Dynos*. Dessa forma, as operações em segundo plano são executadas paralelamente à aplicação Web, garantindo que o desempenho do usuário final não seja afetado (AGGARWAL *et al.*, 2012).

O modelo de cobrança do Heroku para os *Dynos Workers* baseia-se no tempo de execução. Os desenvolvedores são cobrados pelo tempo em que os *Dynos* estão em atividade, independentemente de estarem processando tarefas ou em estado ocioso. O Heroku oferece diferentes tipos de *Dynos*, desde opções gratuitas, com limitações de uso, até *Dynos* de maior desempenho, adequados para aplicações que requerem escalabilidade mais robusta. Para aprimorar os custos, os *Dynos Workers* podem ser configurados para escalarem automaticamente, ativando-se apenas quando houver tarefas pendentes e desativando-se quando o processamento for concluído (SILVA, L. F. da *et al.*, 2017).

3.8 GIT

O Git é um sistema de controle de versão distribuído, amplamente utilizado no desenvolvimento de software, criado por Linus Torvalds em 2005. Sua principal função é gerenciar alterações em arquivos, permitindo que equipes de desenvolvimento colaborem simultaneamente em um mesmo projeto. O Git armazena localmente todo o histórico de alterações do repositório, proporcionando maior independência de servidores externos (THE GIT PROJECT, 2024).

Entre suas principais funcionalidades, destaca-se o conceito de *commit*, que registra o estado do projeto em determinado momento, possibilitando o rastreamento de alterações ao longo do tempo. Cada *commit* é identificado por um código único, garantindo a integridade e a organização do histórico do projeto. Além disso, o Git permite a criação de *branches* (ramificações), que possibilitam desenvolver funcionalidades ou correções isoladamente, sem comprometer a versão principal do projeto (THE GIT PROJECT, 2024).

Devido à sua versatilidade, o Git tornou-se essencial no desenvolvimento de software moderno. Sua capacidade de facilitar o trabalho em equipe e sua compatibilidade com metodologias ágeis, como Scrum e Kanban, consolidam seu uso como uma ferramenta padrão na engenharia de software.

3.9 PADRÃO MVC (Model View Controller)

O padrão arquitetural de software MVC tem em vista estabelecer uma clara separação de responsabilidades entre três componentes principais: Modelo (Model), Exibição (View) e Controlador (Controller). Essa divisão favorece tanto a manutenção quanto a escalabilidade do software, uma vez que cada componente desempenha uma função específica, possibilitando alterações em uma parte do sistema sem interferir diretamente nas demais (KEARNEY, 2019).

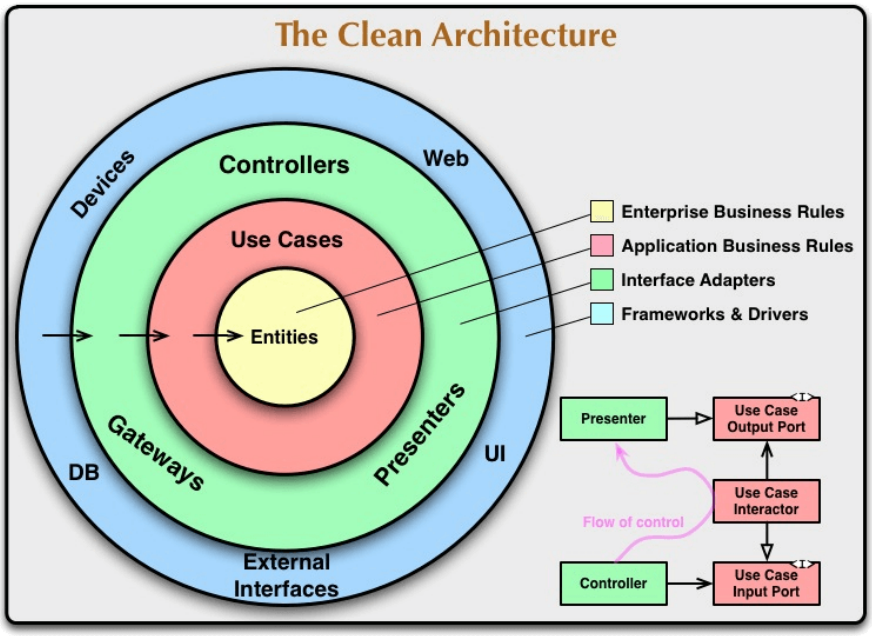
O Modelo é encarregado de gerir os dados e as regras de negócio da aplicação. Ele define como os dados são armazenados, manipulados e estruturados. A Exibição constitui a interface com o usuário, sendo responsável por exibir as informações processadas pelo Modelo de forma compreensível. O Controlador atua como intermediário entre o usuário e o sistema, recebendo as interações e atualizando tanto o Modelo quanto a Exibição com base nas ações do usuário. Dessa forma, o Controlador garante que quaisquer alterações no Modelo sejam refletidas na Exibição, facilitando a interação eficiente entre o usuário e o software (BARBOSA, A. S. R., 2022).

O padrão MVC é amplamente adotado no desenvolvimento de aplicações com interface gráfica, sendo considerado eficiente em projetos de menor escala, graças à sua simplicidade e reusabilidade. Contudo, em sistemas de grande porte, sua implementação pode se tornar mais complexa à medida que a aplicação cresce, exigindo cuidados adicionais para manter a organização e o desempenho (BARBOSA, A. S. R., 2022).

3.10 ARQUITETURA LIMPA

A Arquitetura Limpa é um conjunto de práticas que visa aprimorar a qualidade do código e simplificar a manutenção de um software, organizando o sistema em camadas concêntricas, onde cada uma tem um propósito bem definido e segue o princípio de independência. A Figura 5 ilustra as organização de camadas proposta por Martin (2017).

Figura 5 – Camadas da Arquitetura Limpa.



Fonte: Martin (2017).

Segundo Martin (2017), a arquitetura limpa é composta pelas seguintes camadas principais:

- **Entidades**: Representam as regras de negócio fundamentais e são independentes de qualquer tecnologia. Elas podem ser reutilizadas em diferentes contextos da aplicação, sem dependências externas.
- **Casos de Uso**: Contêm a lógica de aplicação, garantindo que as entidades sejam usadas para cumprir os objetivos específicos do sistema. São responsáveis por executar as operações que seguem as regras de negócio.
- **Adaptadores de Interface**: Responsáveis por intermediar a comunicação entre as camadas internas e os elementos externos, adaptando os dados de entrada e saída conforme necessário pelas entidades e casos de uso.
- **Frameworks e Drivers**: Lida diretamente com tecnologias externas, como bancos de dados e frameworks de interface. Essa camada é projetada para ser substituída sem impactar as camadas internas, assegurando a flexibilidade e modularidade do sistema.

O objetivo é que as regras de negócio, que formam o núcleo do sistema, sejam independentes de frameworks, interfaces de usuário e banco de dados, garantindo um sistema testável e possibilitando a substituição de tecnologias externas sem impactar as regras de negócio (MARTIN, 2017).

3.11 NODE.JS

Node.js é um ambiente de execução de código JavaScript construído sobre o motor V8 do Google, originalmente projetado para funcionar em navegadores, mas adaptado para rodar no lado do servidor. Lançado em 2009, o Node.js trouxe um novo paradigma para o desenvolvimento de aplicações web e servidores, ao introduzir um modelo de execução baseado em eventos e não bloqueante. Isso permite que o ambiente lide eficientemente com múltiplas conexões simultâneas (NODE.JS, 2023).

Uma de suas principais características é a arquitetura baseada em um loop de eventos assíncrono, que possibilita a realização de operações de entrada e saída sem bloquear o restante do processamento. Esse modelo assíncrono torna o Node.js especialmente adequado para sistemas que demandam alta escalabilidade, como servidores de aplicações Web em tempo real (NODE.JS, 2023).

Além disso, o Node.js oferece uma vasta biblioteca de módulos nativos acessíveis por meio do NPM (Node Package Manager), o que simplifica a adição de funcionalidades ao projeto sem a necessidade de desenvolvê-las do zero. O NPM conta com um extenso repositório de pacotes, que vão desde a criação de servidores HTTP simples até sistemas de automação complexos (SOFTWARE, 2023).

3.12 SISTEMAS SUPERVISÓRIOS

O sistema supervisório, frequentemente referido pela sigla SCADA (Supervisory Control and Data Acquisition), é uma aplicação utilizada para monitorar e controlar processos em tempo real, adquirindo, processando e exibindo dados de sensores e dispositivos de campo. Ele permite que operadores supervisionem o sistema e façam intervenções conforme as necessidades detectadas por meio dos dados coletados. Esse tipo de sistema é amplamente utilizado em diferentes áreas, como automação de processos industriais e monitoramento de sistemas de infraestrutura, proporcionando uma visão integrada dos dados e possibilitando a gestão centralizada de diversas variáveis (BOYER, 2009).

A arquitetura de um SCADA inclui camadas responsáveis pela aquisição de dados, processamento e visualização gráfica. Os sensores e dispositivos de campo enviam informações por meio de protocolos de comunicação para o sistema, que processa e apresenta os dados em tempo real. Além disso, o sistema é geralmente composto por módulos que permitem gerenciar alarmes, armazenar histórico de dados e gerar relatórios (CARRARO, 2017).

3.12.1 Módulo de Alarmes

O módulo de alarmes nos sistemas supervisórios é responsável pelo monitoramento contínuo das variáveis de processo, para alertar os operadores sempre que algum parâmetro ultrapassa os limites previamente estabelecidos. O sistema realiza verificações constantes

das variáveis, gerando um alarme sempre que os valores se encontram fora dos intervalos permitidos. Dessa forma, os operadores são notificados de forma ágil, possibilitando a tomada de medidas corretivas imediatas. Esse processamento ocorre em tempo real, garantindo uma resposta rápida a eventuais condições adversas (BOYER, 2009).

Além disso, o sistema armazena os alarmes gerados, permitindo a análise histórica das ocorrências. Esse recurso é essencial tanto para a correção imediata quanto para o acompanhamento de tendências de falhas ao longo do tempo, contribuindo para a manutenção preditiva. Em algumas situações, empresas optam por manter registros físicos dos alarmes, como uma forma adicional de segurança em caso de falhas nos sistemas de armazenamento digital. O registro e a resposta eficaz aos alarmes são fundamentais para assegurar o controle e a segurança dos processos monitorados, garantindo a continuidade operacional e minimizando possíveis impactos negativos (BOYER, 2009).

3.12.2 Módulo de Histórico

O módulo de histórico em sistemas supervisórios desempenha a função de armazenar dados coletados ao longo do tempo, permitindo o acompanhamento contínuo das variáveis monitoradas e facilitando a análise posterior. Com a evolução das interfaces gráficas, que substituíram as antigas telas monocromáticas e alfanuméricas, a visualização de tendências de dados tornou-se mais intuitiva e acessível, especialmente para operadores com diferentes níveis de experiência. Essas interfaces gráficas modernas oferecem uma representação clara de gráficos de tendência, possibilitando uma interpretação mais eficiente do comportamento dos processos monitorados (BOYER, 2009).

A capacidade de registrar e exibir dados históricos visualmente, por meio de gráficos e outras representações, potencializa o monitoramento proativo, permitindo a identificação rápida de anomalias e padrões de desempenho. Esse recurso não apenas auxilia na tomada de decisões operacionais em tempo real, como também contribui para a implementação de estratégias de manutenção preventiva e preditiva, assegurando maior confiabilidade e eficiência nos processos supervisionados (BOYER, 2009).

3.12.3 Módulo de Relatórios

O módulo de relatórios em sistemas supervisórios é responsável por consolidar e organizar informações cruciais sobre alarmes, comunicação e manutenção dos processos monitorados. Esses relatórios podem ser pré-formatados ou personalizados, com geração automática ou sob demanda, conforme as necessidades operacionais. Muitas empresas ainda optam por manter impressoras dedicadas para o registro contínuo de alarmes, assegurando a disponibilidade de uma cópia física e permanente desses dados. Além disso, relatórios operacionais, como os de comunicação e auditoria, são fundamentais para a análise de falhas e o acompanhamento detalhado das operações, contribuindo para a supervisão eficaz e a melhoria dos processos monitorados (BOYER, 2009).

3.13 TRABALHOS RELACIONADOS

O trabalho desenvolvido por Thomé (2023), intitulado Sistema para Monitoramento e Rastreamento de Embarcações Baseado em IoT, visa criar um protótipo para monitoramento e manutenção de embarcações utilizando a tecnologia IoT. O projeto foi desenvolvido para a empresa Navalcare e inclui tanto uma solução de hardware quanto uma aplicação móvel denominada Navalcare Connect, implementada com o framework Flutter. O sistema monitora diversos parâmetros, como localização via GNSS (Global Navigation Satellite System) e condições climáticas, usando tecnologias como GPRS (General Packet Radio Service) e a plataforma Google Cloud/Firebase para armazenamento e processamento dos dados. A aplicação permite que os usuários acessem as informações de suas embarcações remotamente e recebam alertas preventivos de manutenção. O foco do trabalho está na criação de um sistema modular, de baixo custo e fácil instalação, que possa ser adaptado para diferentes embarcações, com integração total entre hardware e software para garantir a segurança e a eficiência operativa (THOMÉ, 2023).

O trabalho de Souza Junior (2023), intitulado Dispositivo Inteligente para Organização e Notificação de Medicamentos em Conjunto com um Aplicativo de Healthcare, apresenta um sistema IoT voltado para auxiliar no gerenciamento de medicamentos, especialmente destinado a idosos e pessoas com doenças crônicas. O projeto desenvolveu dois dispositivos: um com 14 compartimentos para doses diárias e outro com 10 compartimentos para diferentes medicamentos. Juntamente com os dispositivos, foi criado um aplicativo que permite configurar alarmes, monitorar a ingestão de medicamentos e gerar gráficos para acompanhamento médico. O sistema utiliza o microcontrolador ESP32 para controle dos dispositivos e o framework Flutter para o desenvolvimento do aplicativo, com integração ao Firebase para armazenamento de dados e envio de notificações. O protótipo inicial mostrou potencial para melhorar a adesão a medicamentos e facilitar o acompanhamento de tratamentos (SOUZA JUNIOR, 2023).

O artigo Smart IoT-based Water Treatment with a Supervisory Control and Data Acquisition (SCADA) System, de Dwarakanath *et al.* (2023) e colaboradores, propõe um sistema integrado de monitoramento e controle de processos de tratamento de água utilizando IoT e SCADA. O sistema usa sensores conectados via IoT para medir, em tempo real, parâmetros críticos como qualidade, temperatura e pressão da água, além de realizar análises preditivas com algoritmos de DBNs (Deep Belief Networks). Uma das inovações discutidas no artigo é o uso de MQTT, um protocolo leve de comunicação que facilita a transmissão de dados entre dispositivos IoT e o sistema SCADA, proporcionando eficiência e baixo consumo de banda. Além disso, o artigo destaca o uso de CEP (Complex Event Processing) para análise de grandes volumes de dados gerados pelos sensores em tempo real, permitindo a automação de decisões e ações corretivas no tratamento da água. O sistema visa garantir a conformidade com os limites de emissões químicas e melhorar a eficiência operacional, reduzindo a necessidade de intervenções humanas e prevenindo

falhas no sistema por meio de manutenções preditivas. A integração dessas tecnologias oferece uma solução robusta para o gerenciamento de recursos hídricos em estações de tratamento (DWARAKANATH *et al.*, 2023).

O trabalho de Isaque Vilson Batista da Costa, intitulado Desenvolvimento de um Software Supervisório para Monitoramento da Qualidade da Água na Escola Superior de Tecnologia da UEA através da Aquisição de Dados de Sensores, desenvolve um sistema para monitorar a qualidade da água no campus da Universidade do Estado do Amazonas. Utilizando sensores que captam parâmetros físico-químicos da água, o software recebe os dados e os exibe em tempo real, permitindo que os usuários acompanhem a qualidade da água consumida na instituição. A comunicação entre os dispositivos e o sistema é feita por meio de protocolos como MQTT, e os dados são armazenados em uma base de dados MongoDB, com o monitoramento sendo realizado por um sistema desenvolvido em Node.js e React. O projeto aspira auxiliar a universidade a melhorar sua posição no ranking GreenMetric, que avalia as universidades em termos de sustentabilidade (COSTA, 2022).

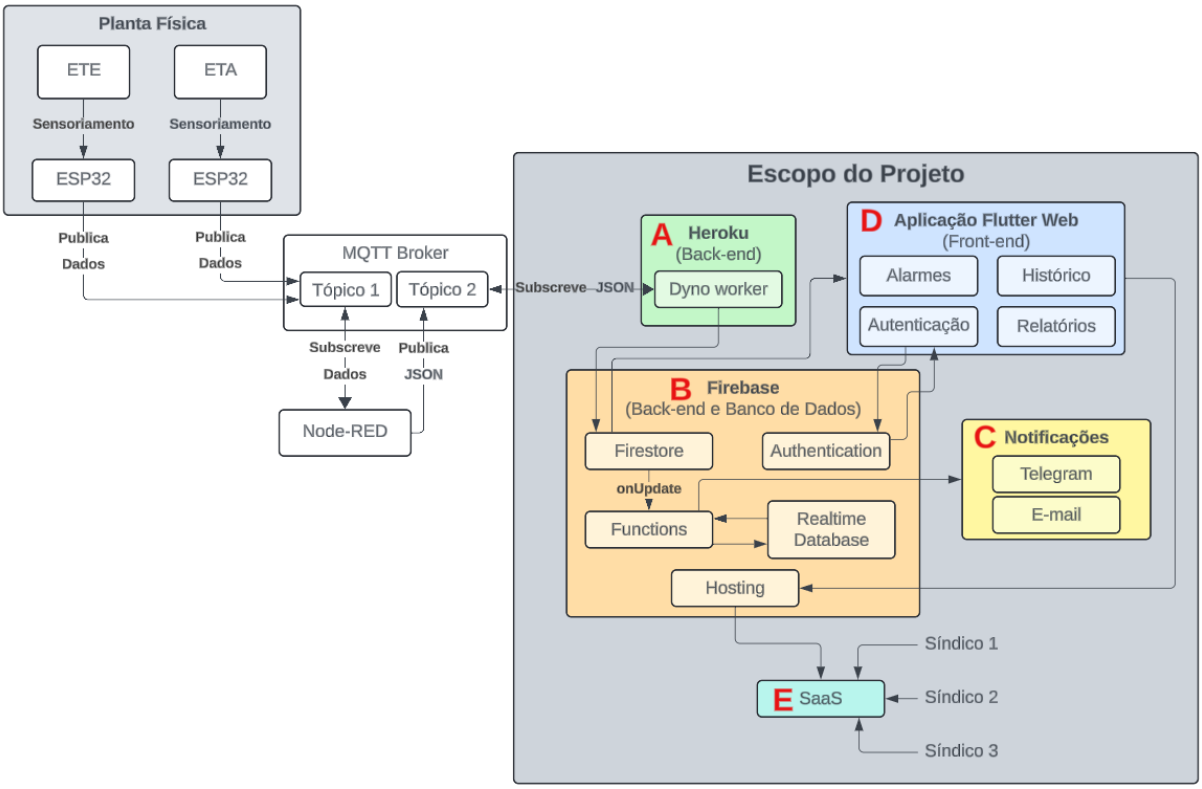
4 PROJETO E ARQUITETURA DO SISTEMA

O presente capítulo aborda a arquitetura e a implementação do sistema supervisório para monitoramento de ETEs e ETAs em condomínios. Serão descritos a organização da arquitetura, incluindo a coleta de dados dos sensores, o processamento no *backend* e o armazenamento no Firestore. Além disso, serão apresentadas as funções automatizadas para geração de alertas e notificações, bem como a integração das tecnologias utilizadas no desenvolvimento da aplicação Web.

4.1 ARQUITETURA PROPOSTA

A arquitetura deste sistema foi concebida com uma abordagem modular, utilizando uma combinação de serviços em nuvem e tecnologias de comunicação para o monitoramento de sistemas de abastecimento de água e estações de tratamento de esgoto. O escopo do projeto se concentra principalmente na infraestrutura de software, cuja função é processar e gerenciar os dados obtidos por sensores distribuídos que monitoram o nível dos reservatórios e o estado das bombas. A arquitetura proposta para o projeto em questão, bem como a delimitação de escopo, são ilustrados na Figura 6.

Figura 6 – Arquitetura do projeto.



Fonte: Elaborado pelo Autor (2024).

Embora o projeto inclua uma camada física de IoT, composta por sensores conecta-

dos ao microcontrolador ESP32 que coletam e transmitem os dados via protocolo MQTT, como especificado na Seção 1.3, o foco principal deste trabalho reside na arquitetura do *backend* a partir do momento em que os dados são recebidos pelo *broker* MQTT. A responsabilidade central do sistema inicia-se com um *worker* (Figura 6-A), implementado em Node.js e hospedado na plataforma Heroku, que atua como subscritor de um tópico MQTT. Esse *worker* processa as mensagens recebidas e as armazena no banco de dados Firestore, da plataforma Firebase (Figura 6-B), onde as informações são organizadas e utilizadas para o gerenciamento de alertas e notificações.

O sistema utiliza Firebase Functions para automação de eventos e controle de alarmes. Essas funções são disparadas sempre que há atualizações em coleções do Firestore que guardam dados recebidos dos dispositivos físicos, possibilitando a detecção de níveis de água abaixo do mínimo permitido ou falhas nas bombas. Em resposta a esses eventos, as funções enviam notificações automáticas aos usuários por meio de Telegram e e-mail (Figura 6-C), assegurando uma comunicação eficiente em situações que exijam atenção imediata.

A escolha do Firestore como banco de dados em detrimento de outro banco de dados com modelo relacional, foi motivada pela facilidade de uso, praticidade e familiaridade com a plataforma. Em determinadas situações, o Realtime Database é utilizado como complemento para algumas funcionalidades específicas do sistema, servindo como um cache de dados e evitando a sobrecarga de leituras realizadas no Firestore.

A interface de usuário é desenvolvida utilizando o Flutter Web (Figura 6-D), oferecendo aos usuários funcionalidades como alarmes, relatórios e histórico dos eventos monitorados. Além disso, a autenticação de usuários é gerenciada através dos serviços do Firebase Authentication, garantindo a segurança no acesso ao sistema.

Por fim, o sistema possui um modelo de distribuição como SaaS (Software as a Service) (Figura 6-E), o que permite que o sistema seja acessado pelos síndicos remotamente e centralizada. Com isso, os síndicos conseguem monitorar suas respectivas instalações de forma independente, sem a necessidade de múltiplas instâncias do sistema.

Apesar da relevância da camada física, composta pelos sensores e controladores IoT, essa parte será abordada de maneira resumida, uma vez que o foco deste trabalho está no desenvolvimento do sistema supervisório e na maneira como os dados são processados, armazenados e utilizados para a geração de alertas. Assim, esta seção apresenta uma visão detalhada da arquitetura de software do sistema, abordando as tecnologias empregadas, a organização dos dados e os componentes responsáveis pela comunicação e controle de eventos.

Os sistemas de supervisão e aquisição de dados podem ser classificados em diferentes categorias, dependendo de suas funcionalidades e do nível de interação permitido com a planta física. Entre essas categorias, destacam-se os sistemas de monitoramento passivo, que apenas coletam e exibem dados ao usuário, sem permitir ações diretas sobre o processo

físico, e os sistemas que incluem controle local ou remoto, permitindo intervenções nos dispositivos da planta. O sistema desenvolvido neste trabalho se enquadra na categoria de monitoramento passivo, uma vez que coleta informações sensoriadas da planta física e as exibe ao usuário, sem oferecer a possibilidade de intervenção direta. Essa abordagem é particularmente útil em cenários onde a prioridade é informar o estado do sistema de maneira confiável, evitando riscos associados a intervenções inadequadas.

Para a validação do sistema de supervisão e aquisição de dados, são realizadas simulações utilizando uma aplicação desenvolvida em Node.js. Essas simulações envolvem o envio sequencial de mensagens para o tópico MQTT, para avaliar o comportamento e o desempenho do *worker* na análise das informações recebidas em formato JSON e na inserção dos dados processados no banco de dados Firestore.

Além disso, é avaliado o desempenho do consumo de dados pela aplicação *frontend* desenvolvida em Flutter Web. Essa análise garantirá que a interface do usuário seja constantemente alimentada com dados atualizados da planta física. Simultaneamente, verifica-se o funcionamento das funções do Firebase, assegurando que notificações em tempo real sejam disparadas corretamente, especialmente em situações que exigem a ativação de novos alarmes.

Para o sistema supervisório, ETAs e ETEs são generalizadas como “centrais”, havendo esses dois tipos de centrais no sistema. Similarmente, tanques e bombas são agrupados sob a denominação de “dispositivos”, sendo que os dispositivos são localizados em centrais.

4.2 PROTOCOLO MQTT

O sistema de comunicação utiliza o protocolo MQTT, permitindo que os dados coletados sejam publicados em um tópico específico no *broker* MQTT. As mensagens publicadas por cada ESP32 possuem informações sobre a central, o dispositivo e o valor lido pelo sensor.

Para facilitar o tratamento dos dados recebidos no tópico MQTT, de maneira que o *worker* possa processar esses dados recebidos em conjunto e registrar na coleção do Firestore, as informações de todos os dispositivos de todas as centrais do condomínio devem ser consolidadas em uma única mensagem, a ser publicada no tópico *values_from_devices* do *broker*. Esse tópico será subscrito pelo *worker*, responsável por registrar os dados no banco de dados.

Para agrupar todas as informações recebidas dos ESP32 instalados em cada dispositivo, utiliza-se o Node-RED para subscrever o tópico MQTT onde os ESP32 publicam os dados separadamente. O Node-RED reúne essas informações em uma estrutura JSON, que será enviada como conteúdo de uma mensagem publicada em um tópico específico, o qual é subscrito pelo *worker* responsável pelo registro dos dados no banco de dados. Dessa maneira, o Node-RED facilita o gerenciamento dos fluxos de dados, realizando

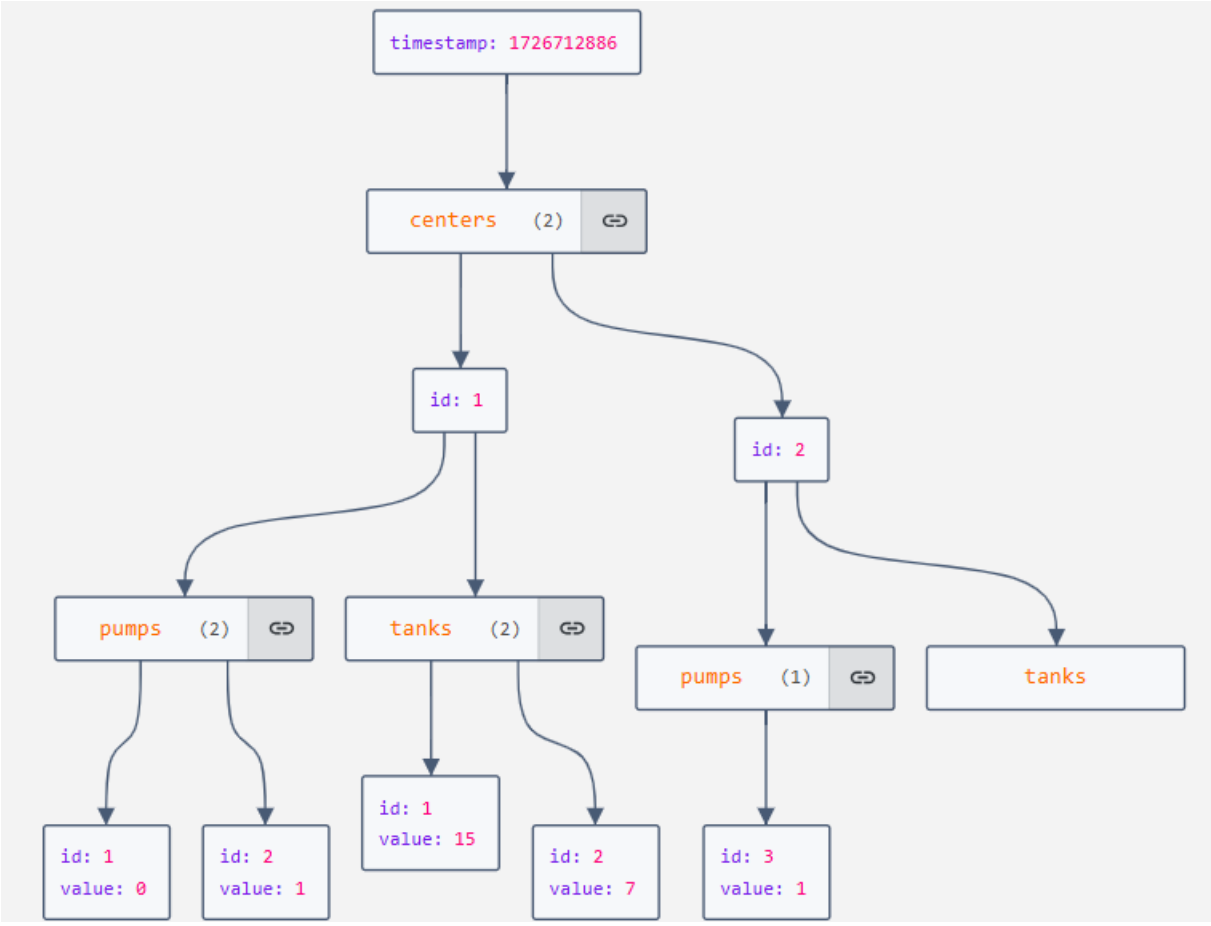
a agregação e o processamento inicial das mensagens antes de sua distribuição para o *backend*. A conversão das mensagens enviadas pelos ESP32 para o formato JSON também contribui para a integração eficiente com outras partes do sistema.

4.2.1 Estrutura da mensagem JSON

As mensagens enviadas ao sistema através do broker MQTT seguem um formato JSON, o qual é amplamente utilizado devido à sua flexibilidade e compatibilidade com diversas tecnologias. A estrutura do JSON contém informações importantes sobre o sistema, como o nível dos reservatórios e o estado das bombas, organizados de maneira a facilitar a interpretação e o processamento no *backend*. O arquivo JSON proposto pode ser visto no Código A.1 presente no Apêndice A.

A Figura 7 ilustra uma visualização gráfica da estrutura JSON projetada para o sistema, sendo dinâmica conforme a quantidade de centrais e dispositivos em cada uma delas.

Figura 7 – Visualização gráfica da estrutura JSON proposta.



Fonte: Elaborado pelo Autor (2024).

A estrutura JSON apresentada contém informações organizadas em dois níveis principais: um campo de “timestamp”, que registra o momento da coleta dos dados em

formato de marca temporal, e um vetor chamado “centers”, que armazena dados de diferentes centrais. Cada objeto dentro de “centers” possui um identificador único de cada central, o campo “id”, além de vetores de “pumps” (bombas) e “tanks” (reservatórios).

Em “pumps”, as bombas têm seus próprios identificadores “id” e um campo “value” que indica seu estado atual (ligada ou desligada, por exemplo). Em “tanks”, os reservatórios são identificados por “id” e possuem o campo “value”, que informa o nível de água no respectivo tanque. Essa estrutura permite organizar de maneira eficiente as informações de múltiplas centrais, com dados específicos sobre bombas e tanques.

4.2.2 *Broker* MQTT

O *broker* MQTT é responsável por gerenciar a comunicação entre os dispositivos IoT, e o sistema supervisório. Ele segue o modelo de publicação/assinatura (*publish/subscribe*), no qual os dados coletados pelos sensores são publicados em tópicos específicos e distribuídos para os subscritores interessados. No contexto deste projeto, foi utilizado o *broker* MQTT integrado ao *Node-RED*, uma escolha que facilita o desenvolvimento e teste do sistema devido à sua integração com fluxos de dados e à capacidade de monitoramento em tempo real.

O *broker* foi configurado para assegurar a entrega confiável das mensagens entre os dispositivos e o *backend*. Para isso, o nível de qualidade de serviço foi definido como 1, garantindo que cada mensagem seja entregue pelo menos uma vez. Esse nível de QoS é suficiente para assegurar que os dados críticos, como o estado das bombas e o nível dos tanques, sejam recebidos pelo sistema supervisório sem duplicação ou perda.

Além disso, foi ativada a retenção de mensagens para que novos clientes que se conectam ao *broker* recebam a última mensagem publicada, assegurando que não haja perda de dados importantes durante eventuais desconexões. A persistência de sessão também foi configurada para manter o estado dos dispositivos conectados, mesmo após desconexões temporárias, garantindo que as mensagens sejam processadas corretamente após a reconexão.

5 DESENVOLVIMENTO DO *BACKEND*

Este capítulo aborda a implementação do *backend* para monitoramento de ETAs e ETEs, com ênfase no uso de um *Dyno Worker* no Heroku. Este worker processa dados de sensores recebidos via MQTT, atualizando coleções no Firestore para monitoramento e histórico. A solução garante a captura de informações em tempo real e a geração de alertas automáticos em caso de falhas ou níveis críticos.

5.1 *DYNO WORKER* NO HEROKU

O sistema de monitoramento dos dispositivos exige uma solução que permita a captura contínua de dados enviados pelos sensores instalados. Esses dados, como os níveis de tanques e o estado de funcionamento das bombas, precisam ser processados e armazenados em um banco de dados para garantir o acompanhamento em tempo real e o registro histórico para futuras análises.

Inicialmente, considerou-se a possibilidade de utilizar um servidor próprio para realizar essa tarefa de subscrição no tópico MQTT e registro dos dados no Firestore. Essa solução permitiria um controle total da infraestrutura, sem a necessidade de depender de provedores de serviços em nuvem. No entanto, essa abordagem apresentou alguns desafios significativos, especialmente relacionados aos custos operacionais com energia e à alta probabilidade de inatividade do servidor por motivos inesperados, como quedas de energia, falhas no hardware ou outros problemas técnicos. Essas questões tornaram essa solução menos confiável, uma vez que qualquer interrupção na disponibilidade do servidor poderia resultar em falhas no recebimento e armazenamento dos dados, impactando diretamente a eficiência do sistema de monitoramento.

Diante desses fatores, foram avaliadas alternativas em nuvem, que oferecem maior confiabilidade e menor necessidade de manutenção física. Entre as opções analisadas e descartadas, estavam:

- **Google Cloud Run**: Oferece execução de processos sem a necessidade de manter servidores, porém, apresentou custos elevados, especialmente para o estágio inicial do projeto.
- **CloudFlare Workers**: Outra possibilidade, com boa escalabilidade, mas também com custos mais altos e maior complexidade de configuração.
- **Vercel**: Focada em implantação de aplicações, essa plataforma apresentou algumas limitações na execução contínua de *workers*, além de custos que também poderiam se tornar um fator limitante conforme o uso do sistema aumentasse.

Por fim, foi analisada a viabilidade de utilização da plataforma Heroku para a implementação de um *Dyno Worker*, que daqui para frente será chamada apenas de worker, como explicado na Seção 3.7.1. Essa opção foi escolhida por sua simplicidade de

configuração e pelo custo relativamente baixo para operações. O Node.js, utilizado para desenvolver o worker, é ideal para este tipo de aplicação devido ao seu suporte à detecção de eventos, permitindo uma execução eficiente com múltiplas conexões simultâneas. Portanto, o Heroku permite a execução contínua de workers sem a necessidade de gerenciamento de infraestrutura física, assegurando maior confiabilidade e reduzindo o risco de inatividade.

5.1.1 Implementação do *Dyno Worker*

A implementação do worker no Heroku, tem como principal objetivo o gerenciamento de duas coleções de documentos no Firestore. Essas coleções são:

1. *last_mqtt_update*: Responsável por armazenar os dados mais recentes enviados pelos dispositivos de campo.
2. *historian_mqtt*: Onde se armazena o histórico de dados enviados pelos dispositivos, para análises posteriores.

5.1.1.1 Gerenciamento da coleção last_mqtt_update

A coleção *last_mqtt_update* visa registrar os dados mais atuais recebidos dos dispositivos conectados, permitindo que o sistema sempre tenha as informações mais recentes disponíveis para consulta. Para isso, o worker implementa um *subscriber* MQTT que ouve o tópico *values_from_devices* com os dados processados pelo Node-RED. Quando uma mensagem é recebida, os dados são extraídos e registrados na coleção *last_mqtt_update*. O código para realizar essa tarefa pode ser descrito da seguinte maneira:

- **Conexão ao *broker* MQTT**: O worker se conecta ao *broker* MQTT e assina o tópico de interesse, como apresentado no Código 5.1 escrito em JavaScript.

Código 5.1 – Conexão do worker ao broker MQTT.

```
mqttClient.on('connect', () => {
  console.log('Connected to MQTT broker');
  mqttClient.subscribe('values_from_devices', (err) => {
    if (!err) {
      console.log('Subscribed to values_from_devices topic.');
    } else {
      console.error('Subscription error:', err);
    }
  });
});
```

- **Recepção e processamento das mensagens**: Toda vez que uma mensagem é recebida, os dados são convertidos de formato JSON e atualizados no Firestore. A função atualiza a coleção *last_mqtt_update* com os dados mais recentes, como apresentado no Código 5.2, onde o primeiro argumento *message* é o identificador do evento, enquanto *topic* e *message* são os argumentos da função programada para execução no gatilho do evento.

Código 5.2 – Recepção e processamento das mensagens pelo worker.

```
mqttClient.on('message', (topic, message) => {
  if (topic === 'values_from_devices') {
    console.log('Message received:', message.toString());
    const data = JSON.parse(message.toString());
    firestore.collection('last_MQTT_update').doc('1').set(data, { merge: true
        })
      .then(() => {
        console.log('Firestore document updated successfully!');
      })
      .catch((error) => {
        console.error('Error updating Firestore document:', error);
      });
  }
});
```

5.1.1.2 Gerenciamento da coleção historian_mqtt

Além de registrar os dados mais recentes, o worker é responsável por armazenar os dados históricos na coleção *historian_mqtt*, permitindo a criação de um banco de dados detalhado para futuras análises. O gerenciamento dessa coleção é feito principalmente por duas funções: *saveHistorianData* e *deleteOldDocuments*.

A função *saveHistorianData* é executada periodicamente e pretende registrar, a cada intervalo definido, os dados coletados dos dispositivos em documentos diários. Esses documentos são organizados por data e subdivididos em duas subcoleções: *pumps* (bombas) e *tanks* (tanques). Cada subcoleção contém um documento denominado *daily_data*, que armazena múltiplos timestamps, com as informações dos dispositivos naquele período.

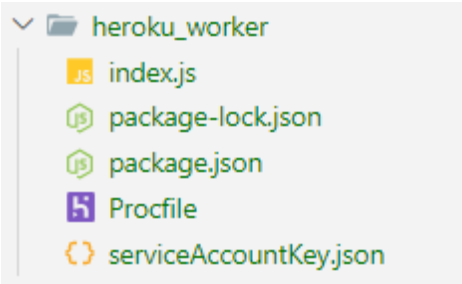
A segunda função, *deleteOldDocuments*, é responsável pela manutenção do banco de dados. Ela é executada diariamente e garante que os dados mais antigos, que ultrapassam o período de retenção definido, sejam excluídos, prevenindo a sobrecarga do banco de dados e respeitando as limitações de armazenamento do Firestore.

O intervalo mínimo entre as amostras, no caso, é de 5 minutos, garantindo uma frequência de registro que equilibra o detalhamento necessário para o monitoramento com a capacidade de armazenamento e as limitações do Firestore. A justificativa para essa escolha será discutida em mais detalhes na Seção 7.2.

5.1.2 Implantação do *Dyno Worker* no Heroku

O processo de implantação no Heroku envolve a configuração de arquivos e o uso de comandos para subir o worker em produção. A Figura 8 mostra os arquivos utilizados para essa implementação.

Figura 8 – Arquivos para implantação do *Dyno Worker* no Heroku.



Fonte: Elaborado pelo Autor (2024).

A seguir serão descritos os arquivos fonte utilizados para a implantação do *Dyno Worker* no Heroku.

1. **Arquivo index.js**: Contém o código principal do worker, incluindo a configuração do cliente MQTT, a conexão com o Firestore e as funções de processamento de dados.
2. **Arquivo Procfile**: Define o tipo de processo que será executado. No caso deste worker, o conteúdo do arquivo é `worker: node index.js`:
3. **Arquivo package.json e package-lock.json**: Define as dependências do projeto e suas versões. Durante a implantação, o Heroku utiliza esses arquivos para garantir que todas as bibliotecas necessárias sejam instaladas corretamente.
4. **Arquivo serviceAccountKey.json**: Contém as credenciais para acesso ao Firestore em ambos os ambientes (produção e desenvolvimento). Esse arquivo é carregado no projeto para permitir a comunicação segura com o banco de dados.

O processo de implantação do worker no Heroku envolve uma série de etapas, começando com a inicialização do repositório Git, seguida pela criação do aplicativo no Heroku e finalizando com o envio do código para o servidor.

Primeiramente, o repositório Git é criado e configurado executando os comandos `$ git init` para inicializar o repositório local, seguido por `$ git add .` para adicionar todos os arquivos ao controle de versão. Em seguida, o comando `$ git commit -m "Initial commit"` realiza o primeiro *commit*, registrando as alterações.

Com o repositório Git preparado, o próximo passo é criar o aplicativo no Heroku com o comando `$ heroku create supervisorio`. Isso configura o ambiente na plataforma Heroku e associa o repositório local ao aplicativo recém-criado. Em seguida, é necessário especificar o *buildpack* do Heroku para Node.js, garantindo que a plataforma saiba como construir e executar a aplicação. Isso é feito utilizando o comando `$ heroku buildpacks:set heroku/nodejs`.

Após essas configurações, o código é enviado para o servidor Git, acessado pelo Heroku quando necessário, através do comando `$ git push heroku master`, fazendo

o upload dos arquivos e iniciando o processo de *deploy*. Finalmente, para garantir que o worker seja executado continuamente, o comando `$ heroku ps:scale worker=1` é utilizado para configurar o processo do worker, ativando-o e permitindo que ele comece a monitorar e processar os dados recebidos dos dispositivos monitorados.

A interação entre o Git e o Heroku se dá de maneira que no contexto do Heroku, o Git não apenas organiza o código, mas também serve como ponte para a comunicação com a plataforma, permitindo que as atualizações feitas no repositório local sejam enviadas e aplicadas no ambiente remoto de forma rápida e eficiente.

Essa integração proporciona diversas vantagens. A possibilidade de utilizar comandos Git para executar a implantação simplifica o fluxo de trabalho e reduz a complexidade da manutenção do aplicativo, especialmente em equipes de desenvolvimento colaborativo. Além disso, como o Heroku consegue acessar o histórico do repositório Git, é possível reverter mudanças ou restaurar versões anteriores do código diretamente no ambiente de produção, aumentando a segurança e a confiabilidade do processo de *deploy*. Dessa forma, o Git e o Heroku trabalham em conjunto para oferecer um processo de implantação ágil, flexível e alinhado às melhores práticas do desenvolvimento moderno.

Com esses passos concluídos, o worker estará implantado e funcionando corretamente no Heroku, pronto para desempenhar suas funções de monitoramento e processamento de dados em tempo real.

5.2 BANCO DE DADOS FIRESTORE

O Firestore, utilizado como o banco de dados principal no projeto, organiza os dados em coleções e documentos no formato de pares chave-valor, permitindo uma estruturação flexível e eficiente. O Firestore foi escolhido devido à sua capacidade de armazenar dados de forma organizada e escalável, alinhando-se às necessidades do sistema de monitoramento de infraestruturas de saneamento.

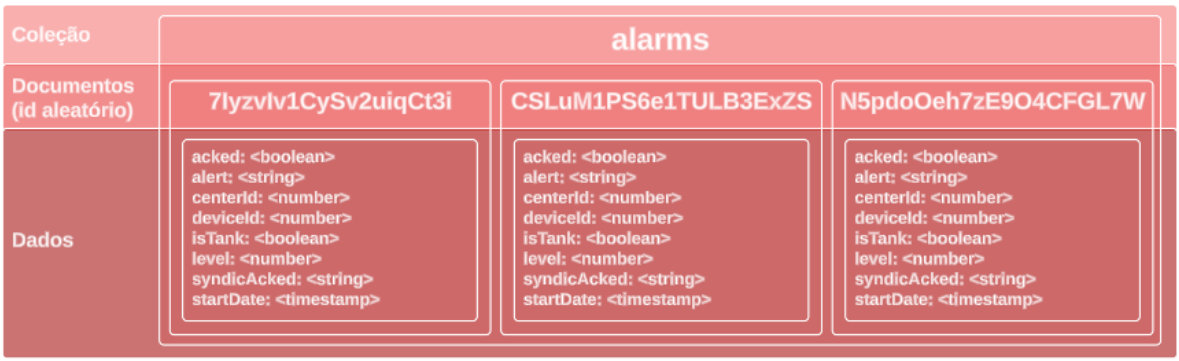
O banco de dados foi estruturado em diferentes coleções, cada uma com uma função específica no sistema. As coleções *alarms*, *alarms_history*, *centers*, *pumps*, *tanks*, *last_mqtt_update*, *historian_mqtt* e *users* armazenam as informações essenciais para o gerenciamento e operação das estações de tratamento. A seguir, detalha-se a função e a estrutura de cada uma dessas coleções.

5.2.1 Coleção *alarms*

A coleção denominada *alarms* é utilizada para armazenar os alarmes ativos no sistema supervisório. Cada documento presente nessa coleção corresponde a um alarme gerado por um dispositivo monitorado, contendo informações detalhadas sobre o tipo de alarme, o dispositivo responsável pela sua geração, o estado de reconhecimento, o estado de resolução do problema e uma breve descrição sobre o motivo do alarme. O estado de

reconhecimento refere-se ao momento em que um usuário comunica ao sistema que tomou ciência do problema e que se responsabilizará por ir ao local adotar as medidas necessárias para sua resolução. Esse procedimento também possibilita que outros usuários saibam quem está encarregado de tratar o alarme. Por sua vez, o estado de resolução indica se o problema foi solucionado ou se o alarme permanece ativo, sinalizando que a situação ainda demanda atenção.

Figura 9 – Estrutura da coleção *alarms*.



Fonte: Elaborado pelo Autor (2024).

A Figura 9 ilustra a estruturação da coleção *alarms* no Firestore. Cada alarme é representado por um documento identificado por um ID gerado aleatoriamente, que contém informações específicas sobre o evento que gerou o alarme. Cada campo de informação presente no documento é descrito a seguir:

- ***acked (boolean)***: Indica se o alarme foi reconhecido por algum usuário do sistema, informando aos demais usuários que uma pessoa já está a caminho para resolver o problema. O valor *true* representa que o alarme foi reconhecido, enquanto *false* indica que ainda não houve reconhecimento.
- ***alert (string)***: Descrição do evento que disparou o alarme, como “Falha na bomba” ou “Nível baixo do tanque”.
- ***centerId (number)***: Identificador do centro de monitoramento ao qual o dispositivo pertence.
- ***deviceId (number)***: Identificador do dispositivo que gerou o alarme.
- ***isTank (boolean)***: Informa se o dispositivo relacionado ao alarme é um tanque. O valor *true* indica que o dispositivo é um tanque, enquanto *false* indica que o dispositivo é uma bomba.
- ***level (number)***: Representa o valor numérico associado ao nível do tanque no momento em que o alarme foi acionado, caso o alarme tenha sido gerado por um tanque específico. Caso contrário, o valor permanece como zero.

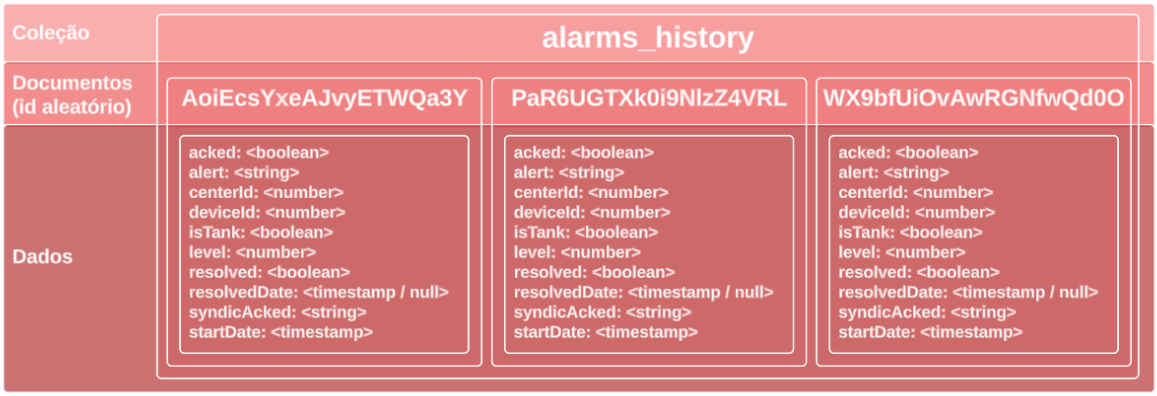
- ***syndicAcked (string)***: Nome do síndico ou responsável que reconheceu o alarme.
- ***startDate (timestamp)***: Data e hora do início do alarme, registrando o momento em que a condição anômala foi detectada.

Os alarmes permanecem nessa coleção até que a condição que os gerou seja resolvida. Após a resolução, o documento correspondente é removido da coleção por meio de uma função na nuvem executada pelo Firebase.

5.2.2 Coleção *alarms_history*

A coleção *alarms_history* armazena o histórico de todos os alarmes que já ocorreram no sistema de monitoramento. Diferente da coleção *alarms*, que registra apenas os alarmes ativos, essa coleção mantém um registro permanente dos alarmes, mesmo após a resolução dos problemas. Cada documento na coleção *alarms_history* contém informações sobre um alarme específico, incluindo detalhes que foram mantidos durante sua ocorrência. A Figura 10 apresenta a estrutura da coleção.

Figura 10 – Estrutura da coleção *alarms_history*.



Fonte: Elaborado pelo Autor (2024).

Cada documento desta coleção possui os mesmos campos presentes na coleção *alarms*, com exceção de dois campos adicionais que registram informações sobre a resolução do alarme:

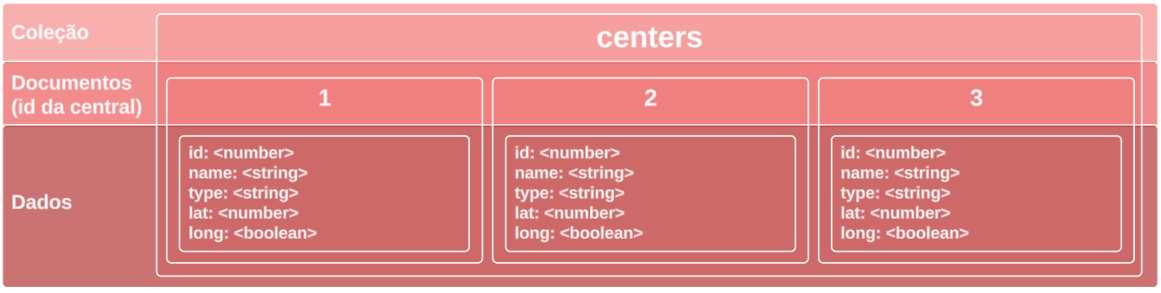
- ***resolved (boolean)***: Indica se o problema que gerou o alarme foi resolvido. Um valor *true* significa que o alarme foi resolvido, enquanto *false* indica que o problema continua pendente no momento em que o histórico foi registrado.
- ***resolvedDate (timestamp / null)***: Este campo registra a data e hora em que o alarme foi resolvido. Se o problema ainda não foi resolvido no momento do registro no histórico, o campo permanece com valor nulo.

Esses campos adicionais permitem que o sistema registre não apenas o início e o reconhecimento dos alarmes, mas também o momento exato em que a situação foi resolvida. Isso garante uma visão detalhada do ciclo de vida de cada alarme, facilitando a análise posterior e o acompanhamento das resoluções de problemas.

5.2.3 Coleção *centers*

A coleção *centers* é responsável por armazenar as informações sobre as centrais de tratamento monitoradas no sistema supervisório, sejam elas ETAs ou ETEs. Cada documento nesta coleção representa uma central específica e contém dados que descrevem sua identificação, localização geográfica e tipo. Esses dados garantem que os dispositivos e alarmes sejam corretamente associados às suas respectivas unidades. A Figura 11 apresenta a estrutura da coleção.

Figura 11 – Estrutura da coleção *centers*.



Fonte: Elaborado pelo Autor (2024).

Os principais campos presentes em cada documento da coleção *centers* são:

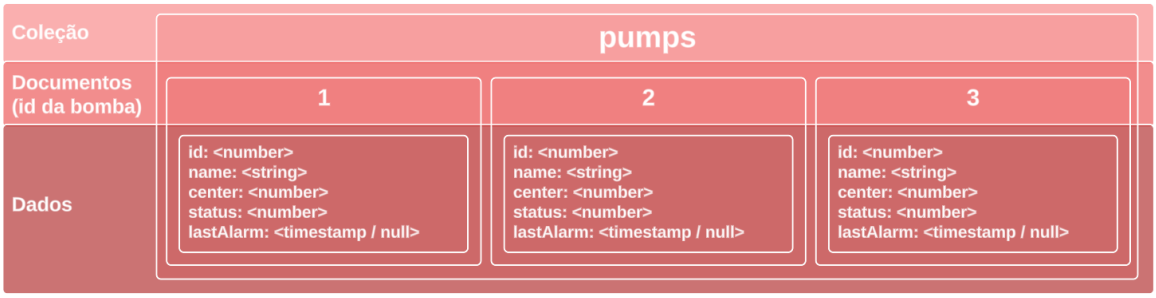
- ***centerId (number)***: Identificador único de cada central no sistema, que permite a vinculação de dispositivos, bombas, tanques e alarmes a uma central específica.
- ***name (string)***: Nome da central de tratamento, facilitando sua identificação no sistema e tornando o monitoramento mais intuitivo para os usuários.
- ***lat (number)***: Latitude da central, representada em coordenadas geográficas (graus decimais). Esse campo é usado para mapear a posição exata da central.
- ***long (number)***: Longitude da central, também representada em graus decimais. Esse campo, juntamente com o campo lat, permite que a posição da central seja visualizada em um mapa no sistema.
- ***type (string)***: Tipo de central, que pode ser “ETA” para Estações de Tratamento de Água ou “ETE” para Estações de Tratamento de Esgoto, permitindo a distinção entre diferentes tipos de centrais.

Os dados de latitude (lat) e longitude (long) são fundamentais para a visualização das centrais no sistema supervisório, ao permitirem que as unidades sejam mapeadas precisamente em interfaces gráficas. A exibição geográfica das centrais facilita a identificação rápida da localização das estações monitoradas, auxiliando na tomada de decisões operacionais, especialmente em cenários com múltiplas ETAs ou ETEs distribuídas geograficamente. Isso proporciona uma visão abrangente e espacial das instalações, permitindo que os síndicos monitorem e ajustem o funcionamento das centrais de forma mais eficiente, com base em sua localização no mapa.

5.2.4 Coleção *pumps*

A coleção *pumps* é responsável por armazenar as informações sobre as bombas instaladas nas centrais de tratamento monitoradas pelo sistema supervisório. Cada documento nesta coleção representa uma bomba específica e contém dados que descrevem sua identificação, estado operacional e informações sobre eventos de alarme. Esses dados garantem que as bombas estejam corretamente associadas às suas respectivas centrais e que o sistema monitore o desempenho e possíveis falhas. A Figura 12 apresenta a estrutura da coleção.

Figura 12 – Estrutura da coleção *pumps*.



Fonte: Elaborado pelo Autor (2024).

Os principais campos presentes em cada documento da coleção *pumps* são:

- ***id (number)***: Identificador único da bomba, gerado automaticamente em ordem crescente à medida que novos dispositivos são adicionados ao sistema. Isso facilita a identificação sequencial de cada bomba.
- ***name (string)***: Nome da bomba, facilitando sua identificação pelos operadores do sistema, especialmente quando múltiplas bombas estão em operação em uma mesma central.
- ***center (number)***: Identificador da central à qual a bomba está vinculada. Esse campo referencia o ID da central na coleção *centers*, garantindo que cada bomba seja associada à sua respectiva central.

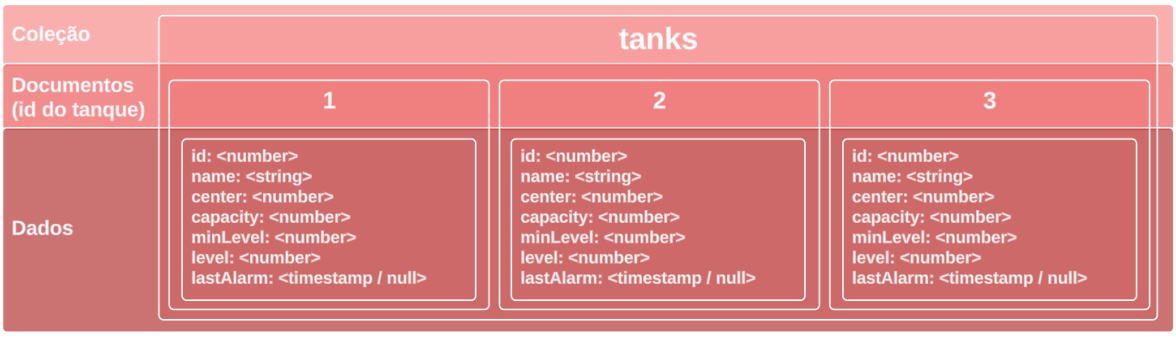
- ***status (number)***: Estado atual da bomba, representado por um número que indica se a bomba está operando normalmente, com valor um, ou se está com falha, com valor zero.
- ***lastAlarm (timestamp / null)***: Data e hora do último alarme associado à bomba. Se nenhum alarme tiver sido gerado, o valor será nulo.

Os dados de *status* e *lastAlarm* são importantes para o controle e monitoramento das bombas, permitindo que os operadores detectem de forma rápida qualquer falha ou interrupção no funcionamento. A coleta contínua dessas informações possibilita a análise das condições operacionais, facilitando a identificação de padrões que indiquem a necessidade de manutenções preventivas. Dessa forma, é possível manter as bombas operando nos parâmetros estabelecidos e reduzir o tempo de inatividade causado por problemas inesperados.

5.2.5 Coleção *tanks*

A coleção *tanks* é responsável por armazenar as informações sobre os tanques nas centrais de tratamento, também monitorados pelo sistema supervisório. Cada documento nesta coleção representa um tanque específico e contém dados sobre sua capacidade, níveis de operação e alarmes. Assim como nas bombas, esses dados garantem que os tanques estejam corretamente associados às suas centrais e permitam o monitoramento contínuo. A Figura 13 apresenta a estrutura da coleção.

Figura 13 – Estrutura da coleção *tanks*.



Fonte: Elaborado pelo Autor (2024).

Os principais campos presentes em cada documento da coleção tanks são:

- ***id (number)***: Identificador único do tanque, gerado automaticamente em ordem crescente à medida que novos dispositivos são adicionados ao sistema. Isso permite uma numeração sequencial dos tanques conforme são instalados.
- ***name (string)***: Nome do tanque, que facilita sua identificação pelos operadores do sistema.

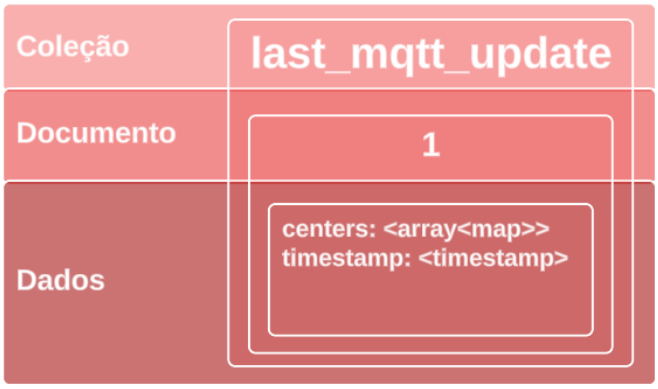
- ***center (number)***: Identificador da central à qual o tanque está vinculado. Esse campo referencia o ID da central na coleção *centers*, garantindo que cada tanque esteja corretamente associado à sua central.
- ***capacity (number)***: Capacidade total do tanque, medida em litros. Esse campo é utilizado para garantir que o tanque não exceda seu limite operacional.
- ***minLevel (number)***: Nível mínimo de segurança do tanque, que indica o ponto no qual o sistema deve emitir um alerta para evitar níveis críticos.
- ***level (number)***: Nível atual de água ou efluente no tanque. Esse campo é monitorado continuamente para garantir que o tanque opere nos limites estabelecidos.
- ***lastAlarm (timestamp / null)***: Data e hora do último alarme associado ao tanque. Se nenhum alarme tiver sido gerado, o valor será nulo.

Os campos *capacity*, *minLevel* e *level* são específicos da coleção *tanks*, diferenciando-se da coleção *pumps*, que não possui dados sobre armazenamento de líquidos. Esses campos são fundamentais para o monitoramento do volume dos tanques, ajudando a identificar situações críticas, como níveis muito baixos. A coleta e o monitoramento desses dados são essenciais para manter o equilíbrio operacional das centrais de tratamento e garantir a segurança e continuidade das operações.

5.2.6 Coleção *last_mqtt_update*

A coleção *last_mqtt_update* é responsável por armazenar as informações da última atualização recebida dos dispositivos por meio do protocolo MQTT no sistema supervisório. Essa coleção contém um único documento, que é constantemente atualizado com os dados mais recentes recebidos, substituindo os dados anteriores. O documento organiza as informações por centrais de monitoramento e inclui um registro de tempo que indica o momento exato da última atualização. A Figura 14 apresenta a estrutura da coleção.

Figura 14 – Estrutura da coleção *last_mqtt_update*.



Fonte: Elaborado pelo Autor (2024).

Os principais campos presentes em cada documento da coleção *last_mqtt_update* são:

- ***centers (array<map>)***: Este campo armazena uma lista de mapas contendo informações sobre cada central que recebeu uma atualização via MQTT. Cada item do *array* representa os dados relacionados a uma central específica.
- ***timestamp (timestamp)***: Registra o momento exato em que a última atualização foi recebida.

Os dados contidos no campo *centers* representam um *array* de mapas, onde cada mapa armazena informações detalhadas sobre as centrais que enviaram dados ao sistema por meio de mensagens MQTT. Esse campo permite que o sistema supervisório rastreie múltiplas centrais simultaneamente, garantindo a atualização em tempo adequado de todas as unidades monitoradas.

A Figura 7 e o Apêndice A mostram a estrutura JSON das mensagens enviadas via protocolo MQTT, detalhando como esses dados são organizados. A presença do campo *timestamp* é essencial para o sistema identificar falhas ou interrupções na comunicação das centrais e notifique os operadores, caso os dados não estejam sendo atualizados em um período esperado.

O uso desses campos permite que o sistema identifique rapidamente possíveis falhas de comunicação e assegure que as informações transmitidas pelos dispositivos e centrais estejam sempre atualizadas, garantindo um monitoramento contínuo.

5.2.7 Coleção *historian_mqtt*

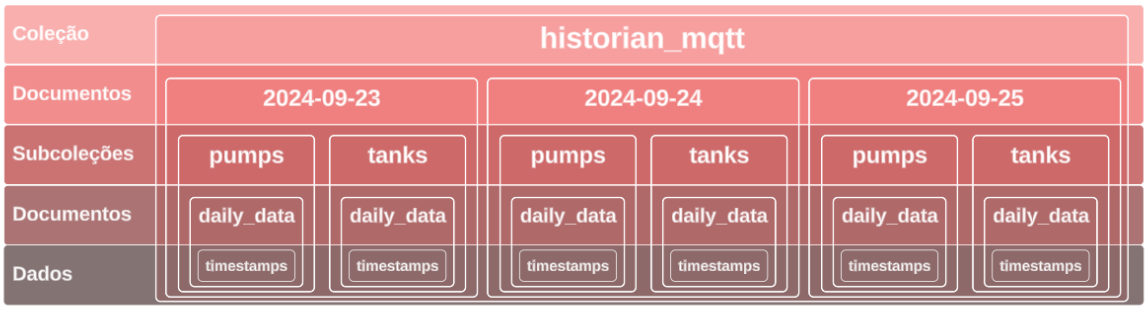
Devido a restrições inerentes ao Firestore, como o limite de 1 MB por documento, além dos custos associados ao armazenamento e operações de leitura e escrita que aumentam conforme se ultrapassam os limites do plano gratuito, torna-se necessário organizar a coleção de dados eficientemente, assim como definir um intervalo mínimo adequado entre as amostras. O objetivo é permitir o registro contínuo dos dados ao longo de todo o dia, ao mesmo tempo que se facilita o acesso e a leitura do histórico pelo sistema supervisório.

Conforme o esquema apresentado na Figura 15, a coleção *historian_mqtt* é organizada da seguinte maneira:

- Coleção: *historian_mqtt* contém os documentos que representam os dados diários.
- Documentos: Cada documento é nomeado com a data (por exemplo, 2024-09-23) e armazena os dados de um único dia.
- Subcoleções: Dentro de cada documento diário, há duas subcoleções: *pumps* (bombas) e *tanks* (tanques).

- Documentos das Subcoleções: Dentro de cada subcoleção, há um documento chamado *daily_data*, onde são armazenados todos os dados referentes às bombas ou tanques de um dia específico.
- Dados (timestamps): O documento *daily_data* contém múltiplos timestamps, que registram os dados de diferentes amostras ao longo do dia.

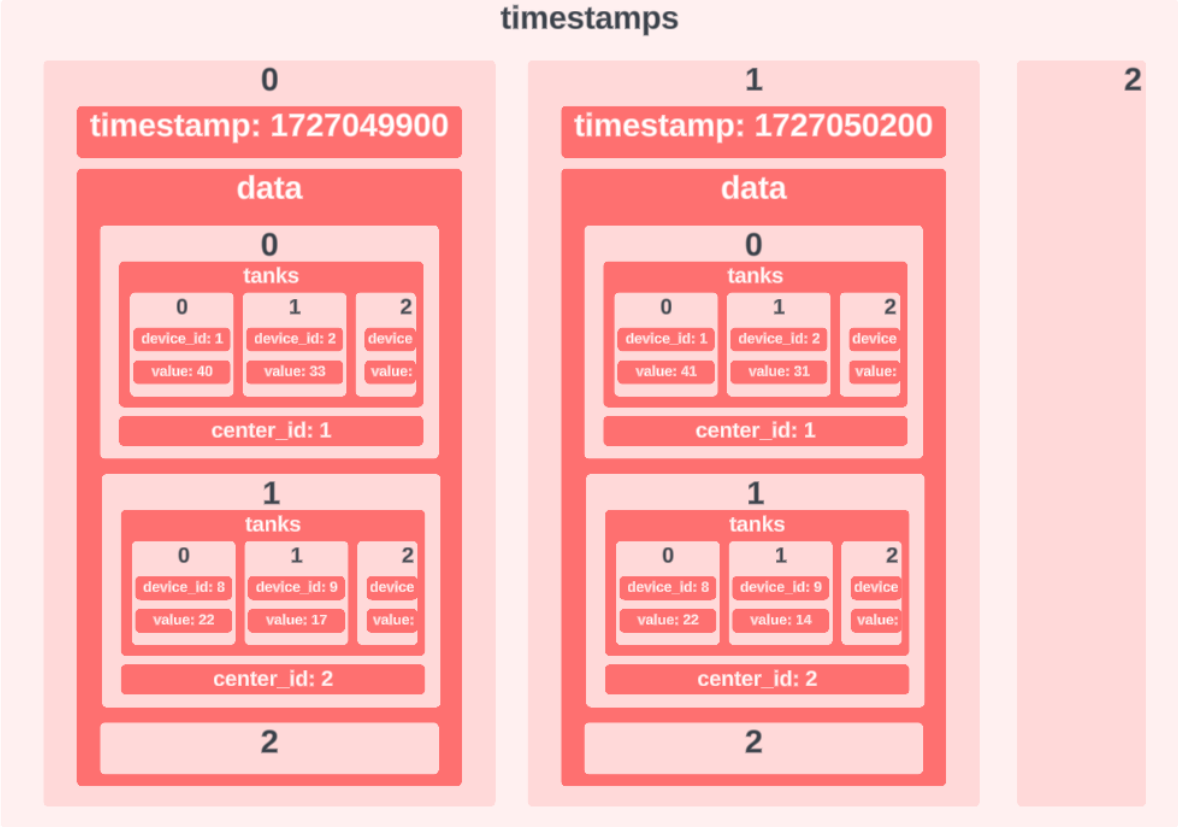
Figura 15 – Arquitetura para a coleção *historian_mqtt*.



Fonte: Elaborado pelo Autor (2024).

Os dados (timestamps) presentes no documento *daily_data* seguem a estrutura JSON esquematizada na Figura 16 para tanques e apresentada no Apêndice B, onde cada timestamp registra as informações de um momento específico. Dentro de cada timestamp, os dados são organizados em listas hierárquicas que podem conter vários elementos, como indicado pelos índices (0, 1, 2, ...) na figura.

Figura 16 – Visualização gráfica da estrutura JSON inserida em *historian_mqtt*.



Fonte: Elaborado pelo Autor (2024).

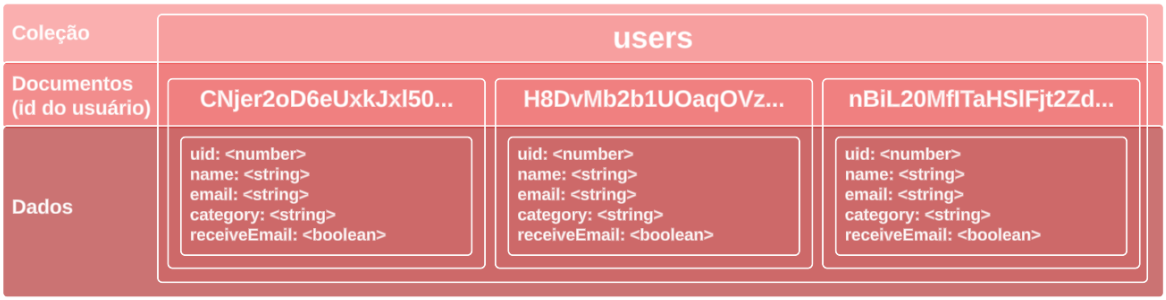
Cada timestamp inclui um vetor de objetos, denominado data, o qual armazena os dados de diferentes centrais (*centers*). Cada central é identificado por um *center_id* e contém um conjunto de dispositivos, que podem ser tanques (*tanks*) ou bombas (*pumps*). Dentro dessas listas, cada dispositivo é representado por um *device_id* e o valor correspondente (*value*), como o nível de um tanque ou o estado de uma bomba naquele momento.

Essa estrutura permite armazenar os dados de várias centrais e dispositivos em um único timestamp, facilitando a organização das informações e permitindo consultas mais eficientes para o sistema supervisório. As listas são flexíveis e podem acomodar diversos dispositivos, garantindo que o sistema consiga lidar com um número significativo de sensores e atuadores distribuídos entre diferentes centrais.

5.2.8 Coleção *users*

A coleção *users* é utilizada para armazenar as informações dos usuários que possuem acesso ao sistema supervisório. Cada documento na coleção representa um usuário específico e contém dados que descrevem sua identidade, papel no sistema e preferências de recebimento de notificações por e-mail. A Figura 17 apresenta a estrutura da coleção.

Figura 17 – Estrutura da coleção *users*.



Fonte: Elaborado pelo Autor (2024).

- ***uid (number)***: Identificador único do usuário, que permite associar as ações realizadas no sistema a um usuário específico.
- ***name (string)***: Nome completo do usuário, utilizado para identificação no sistema.
- ***email (string)***: Endereço de e-mail do usuário, utilizado para autenticação e também para o envio de notificações ou alertas do sistema.
- ***category (string)***: Define o papel do usuário no sistema, podendo ser "Host" ou "Síndico". O campo *category* é utilizado para definir os níveis de acesso e permissões no sistema, onde usuários com o perfil de "Host" possuem permissões para criar e editar componentes do sistema, enquanto os "Síndico" têm acesso mais restrito.
- ***receiveEmail (boolean)***: Define se o usuário receberá notificações de alertas por e-mail. Quando definido como *true*, o usuário recebe alertas diretamente em sua caixa de e-mail. No entanto, caso essa opção seja mantida como *true* de forma contínua, o usuário poderá ter sua caixa de entrada sobrecarregada com e-mails indesejados. Quando *false*, o usuário não recebe alertas por e-mail.

O controle exercido pelos campos *category* e *receiveEmail* permite uma gestão eficiente de permissões e preferências no sistema. Com isso, é possível garantir que as funcionalidades mais sensíveis, como a criação e edição de componentes, sejam acessíveis apenas a usuários com permissões adequadas, enquanto o envio de alertas é personalizado conforme as preferências de cada usuário, evitando notificações excessivas.

5.2.9 Índices

Os índices no Firestore são mecanismos utilizados para tornar mais eficiente a recuperação de dados em consultas que utilizam múltiplos filtros, ordenações ou combinações de campos. Eles permitem que o banco de dados acesse informações de forma mais rápida e eficiente, especialmente em consultas complexas que utilizam cláusulas *where*, *orderBy*

e *range*. Sem os índices, as consultas que filtram por mais de um campo podem ser lentas ou até mesmo não suportadas pelo Firestore.

No sistema supervisório desenvolvido, os índices foram configurados para melhorar o desempenho das buscas na coleção de histórico de alarmes (*alarms_history*), onde as consultas precisam filtrar dados com base em critérios como central, dispositivo, tipo de alarme e intervalos de datas. Esses índices são essenciais para permitir que as consultas realizadas no aplicativo Flutter sejam rápidas e eficientes, principalmente quando se trata de grandes volumes de dados históricos.

A Figura 18 apresenta os índices configurados no Firestore para a coleção *alarms_history*.

Figura 18 – Índices configurados na coleção *alarms_history* do Firestore.

ID da coleção	Campos indexados	Escopo da consulta	Status
alarms_history	deviceId Crescente startDate Crescente __name__ Crescente	Coleta	Ativado
alarms_history	centerId Crescente deviceId Crescente startDate Crescente __name__ Crescente	Coleta	Ativado
alarms_history	isTank Crescente startDate Crescente __name__ Crescente	Coleta	Ativado
alarms_history	id Crescente center Crescente startDate Crescente __name__ Crescente	Coleta	Ativado
alarms_history	centerId Crescente startDate Crescente __name__ Crescente	Coleta	Ativado

Fonte: Firestore - Firebase Google (2024).

As principais combinações de campos indexados incluem:

- ***deviceId* e *startDate***: utilizados para buscar alarmes de um dispositivo específico em um intervalo de datas.
- ***centerId*, *deviceId* e *startDate***: tornam mais eficiente a busca quando é necessário filtrar os alarmes por central, dispositivo e data.
- ***isTank* e *startDate***: utilizados para buscar alarmes relacionados a tanques em um período específico.

A criação de índices compostos no Firestore é indispensável para a execução de consultas que envolvem múltiplos filtros simultâneos, como a ilustrada no Código 5.3.

Código 5.3 – Exemplo de utilização de múltiplos filtros.

```
Query query = alarmHistorycollectionReference
    .where('startDate', isGreaterThanOrEqualTo: Timestamp.fromDate(startDate!))
    .where('startDate', isLessThanOrEqualTo: Timestamp.fromDate(endDate!))
    .where('centerId', isEqualTo: centerId)
    .orderBy('startDate')
.startAt([Timestamp.fromDate(startDate)]).endAt([Timestamp.fromDate(endDate)]);
```

Essa consulta utiliza dois filtros aplicados aos campos: *centerId*, *startDate* e a ordenação por *startDate*. Sem um índice composto que inclua esses campos, a consulta

poderia se tornar ineficiente ou até mesmo resultar em erros de limite de consulta no Firestore.

5.3 UTILIZAÇÃO DO REALTIME DATABASE PARA GERAÇÃO DE ALARMES

O Firebase Realtime Database é utilizado como um cache de dados, ou seja, uma cópia de dados para acelerar a recuperação dos mesmos, o qual armazena dados estáticos, como nome de centrais, nome de bombas, nome capacidade e limites de tanques, além dos e-mails dos usuários que devem receber notificações. Essa abordagem visa tornar mais eficiente o desempenho do sistema, evitando leituras constantes no Firestore, que podem gerar custos adicionais e impactar a eficiência do sistema.

A escolha pelo uso do Realtime Database em conjunto com o Firestore foi motivada pela necessidade de recuperar dados frequentemente acessados pelas funções na nuvem. Esses dados, em sua maioria, são considerados estáticos na aplicação ou sofrem poucas alterações após a primeira inserção. Com o uso do Realtime Database aliado a um sistema de cache, as funções conseguem acessar essas informações de forma rápida e eficiente, reduzindo a necessidade de novas leituras no Firestore. Essa abordagem é especialmente vantajosa devido ao limite de leituras gratuitas no Firestore, otimizando o desempenho e os custos da aplicação.

Na estrutura apresentada na Figura 19, o Realtime Database é utilizado para armazenar as seguintes informações importantes para o funcionamento do sistema:

- **Centrais**: No nó *centers*, são armazenados os nomes das centrais. Essas informações são utilizadas para que as notificações enviadas aos usuários contenham os nomes legíveis das centrais, evitando consultas repetidas ao Firestore sempre que uma notificação precisa ser enviada.
- **E-mails**: No nó emails, são mantidos os e-mails dos usuários que devem receber alertas por e-mail. Isso permite que, ao disparar um alarme, o sistema acesse rapidamente os e-mails corretos, sem precisar fazer novas consultas ao Firestore para identificar quais usuários optaram por receber notificações.
- **Bombas**: No nó *pumps*, o sistema armazena o nome da bomba e o estado de alerta mais recente, representado pelo campo *lastAlertStatus*. Esse campo indica se o último estado da bomba gerou um alerta (valor *true*) ou não (valor *false*). Essa informação é importante para evitar a geração de notificações repetitivas. Um alerta só é disparado se o estado anterior da bomba era "sem alerta" (ou seja, $lastAlertStatus = false$) e o novo estado indica uma falha ou problema.
- **Tanques**: No nó *tanks*, os dados armazenados incluem o nome do tanque, o último estado de alerta (*lastAlertStatus*), e o nível mínimo aceitável para aquele tanque (*minLevel*). O campo *lastAlertStatus* tem a mesma função das bombas, evitando a duplicação de alertas. O campo *minLevel*, por sua vez, é utilizado

Figura 19 – Estrutura do Realtime Database para o cache de dados.

Fonte: Elaborado pelo Autor (2024).

para verificar se o nível de água no tanque caiu abaixo do limite aceitável, acionando um alerta caso necessário.

Esse cache no Realtime Database possibilita que o sistema realize verificações e envie notificações de maneira rápida e eficiente, sem a necessidade de consultar constantemente o Firestore, o que reduziria o desempenho e aumentaria os custos de operação. Ao manter essas informações no Realtime Database, o sistema consegue monitorar continuamente os níveis de tanques e o estado de bombas, acionando alarmes somente quando necessário e garantindo que as informações sejam consistentes e atualizadas em tempo real.

5.4 FIREBASE FUNCTIONS

As Firebase Functions foram implementadas utilizando Node.js sendo responsáveis por automatizar diversas tarefas no sistema, desde a verificação e geração de alarmes até a atualização de dados e o envio de notificações. Essas funções permitem a execução de código de *backend* em resposta a eventos específicos, como atualizações de dados no Firestore e no Realtime Database, utilizado como um cache para tornar o processo mais eficiente.

As Seções 5.4.1 a 5.4.5 detalham as principais funções criadas, explicando o objetivo

de cada uma, os gatilhos que as disparam e a lógica por trás de seu funcionamento.

5.4.1 *checkAlarms*

Essa função é a mais crítica para o monitoramento dos dispositivos, sendo responsável por verificar os alarmes ativos. Ela é acionada sempre que há uma atualização no documento da coleção *last_mqtt_update*, que contém os dados mais recentes de dispositivos, como bombas e tanques.

A função realiza as seguintes tarefas:

- Sincronização de dados: A função primeiro sincroniza os dados dos dispositivos armazenados no Realtime Database, garantindo que os nomes e estados dos dispositivos (bombas e tanques) estejam atualizados.
- Verificação de condições de alarme: Para cada bomba e tanque presentes nas centrais, a função compara o estado atual dos dispositivos com o estado anterior armazenado no Realtime Database. Se o estado anterior não indicar falhas e o estado atual de uma bomba ou tanque indicar uma falha, como nível de água baixo ou falha na bomba, um novo alarme é gerado.
- Registro de alarmes no Firestore: Caso um novo alarme seja detectado, a função registra o alarme na coleção *alarms_history*, garantindo que o histórico dos eventos seja mantido.
- Envio de notificações: Para cada alarme gerado, a função prepara mensagens para serem enviadas via *bot* de Telegram e e-mail, notificando os responsáveis pelo sistema.
- Resolução de alarmes: Ao detectar que um alarme passou de estado ativo para resolvido, o registro desse alarme na coleção *alarms_history* tem seu campo de resolvido alterado para verdadeiro, além de registrar a data de resolução em outro campo.

A lógica da função evita a duplicação de notificações, gerando novos alarmes apenas se o estado anterior do dispositivo não indicava um problema, ou seja, o alarme anterior estava resolvido.

5.4.2 *updatePumpOnEdit* e *updateTankOnEdit*

Essas funções são disparadas quando ocorre uma alteração em um documento nas coleções *pumps* ou *tanks*, respectivamente. O objetivo dessas funções é manter os dados de dispositivos atualizados no Realtime Database para que a verificação de alarmes ocorra eficientemente, sem a necessidade de realizar leituras constantes no Firestore.

- ***updatePumpOnEdit***: Atualiza os nomes e estados das bombas quando um documento na coleção *pumps* é modificado, sincronizando os dados no Realtime Database.
- ***updateTankOnEdit***: Realiza a mesma tarefa para os tanques, garantindo que os nomes e limites mínimos de nível estejam sempre sincronizados no Realtime Database.

Ambas as funções são fundamentais para manter os dados atualizados, evitando leituras desnecessárias do Firestore durante as verificações de alarme.

5.4.3 *scheduleAlertUpdates*

Essa função é executada periodicamente, a cada uma hora, para verificar o estado dos alarmes ativos no sistema. O objetivo é garantir que, caso algum alarme permaneça ativo por um período prolongado, os responsáveis sejam notificados repetidamente até que a situação seja resolvida.

- Verificação de alarmes não resolvidos: A função consulta a coleção *alarms* e verifica se existem alarmes ainda em aberto (não resolvidos).
- Envio de atualizações: Para cada alarme não resolvido, novas notificações são enviadas aos usuários via *bot* de Telegram e e-mail, a fim de reforçar o alerta.

A execução periódica dessa função garante que nenhum alarme passe despercebido ou fique sem a devida atenção dos responsáveis.

5.4.4 *onUserUpdate*

Essa função é acionada sempre que há uma atualização no documento de um usuário na coleção *users*. O principal objetivo é garantir que, quando um usuário atualiza suas preferências, como o recebimento de e-mails, os dados sejam sincronizados no Realtime Database.

- Atualização das preferências de e-mail: Quando um usuário altera seu e-mail ou sua preferência de receber notificações por e-mail, a função atualiza o Realtime Database, garantindo que as próximas notificações sejam enviadas corretamente.

Essa função é importante para manter os dados de contato dos usuários atualizados, permitindo que as notificações sejam enviadas eficientemente.

5.4.5 *updateCenterOnEdit*

A função *updateCenterOnEdit* é executada sempre que um documento na coleção *centers* é atualizado. Seu objetivo é garantir que os nomes das centrais de tratamento (ETEs ou ETAs) estejam sincronizados corretamente no Realtime Database.

- Sincronização dos nomes das centrais: A função atualiza o nome da central no Realtime Database, garantindo que as notificações enviadas aos usuários utilizem os nomes corretos e legíveis das centrais.

Isso evita que seja necessário consultar o Firestore sempre que uma notificação precisa ser enviada, otimizando o desempenho do sistema.

5.5 INTEGRAÇÃO COM TELEGRAM E E-MAIL

A integração com o Telegram e o E-mail permite que os usuários recebam notificações automáticas sempre que um alarme é gerado. As Seções 5.5.1 e 5.5.2 detalham a implementação.

5.5.1 Envio de Notificações via Telegram

A integração com o Telegram foi implementada utilizando a biblioteca *Telegraf*, que permite o envio de mensagens para um *bot* específico. Sempre que um alarme é detectado, a função *checkAlarms* ou *scheduleAlertUpdates* gera uma mensagem enviada ao Telegram.

- Identificação do chat: A função utiliza o ID do chat configurado para enviar mensagens para o grupo ou usuário correto.
- Mensagens detalhadas: As mensagens contêm detalhes sobre o alarme gerado, como o nome da central, o nome do dispositivo (bomba ou tanque), e o problema detectado e o nível atual do tanque se for o caso.

A função auxiliar *sendTelegramMessage* cuida do envio das mensagens, consolidando todos os alertas gerados em uma única mensagem e enviando-os de uma vez.

5.5.2 Envio de Notificações por E-mail

A integração de e-mail foi implementada utilizando a biblioteca *nodemailer*, que se conecta a um servidor SMTP (Simple Mail Transfer Protocol), como o Gmail, para enviar e-mails aos usuários cadastrados.

- Verificação de usuários optados por receber e-mails: O sistema consulta o Realtime Database para verificar quais usuários têm a preferência de receber notificações por e-mail. Se o usuário tiver ativado a opção *receiveEmail*, ele será incluído na lista de destinatários.
- Mensagens personalizadas: O corpo do e-mail possui informações incluindo o nome da central, o dispositivo afetado, e o estado atual do alarme.

A função *sendMail* é responsável por compor e enviar o e-mail, utilizando o serviço de SMTP configurado. Além disso, o sistema garante que os e-mails sejam enviados apenas para os usuários que optaram por receber alertas, evitando o envio de e-mails desnecessários.

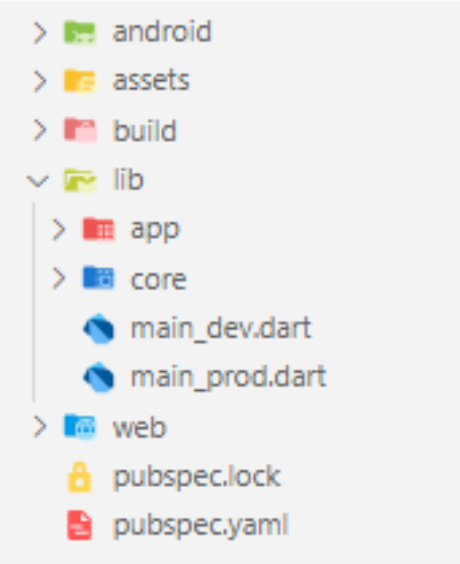
6 DESENVOLVIMENTO DO *FRONTEND*

Neste capítulo, será apresentada a implementação do *frontend* da aplicação utilizando o Flutter, abordando as principais decisões arquiteturais e práticas adotadas no desenvolvimento. A aplicação foi inicialmente desenvolvida para ambiente Web, com a estrutura preparada para uma adaptação futura para Android, aproveitando a flexibilidade do Flutter para criar interfaces multiplataforma com um único código base.

6.1 ESTRUTURA DO PROJETO

Na Figura 20, é apresentada a estrutura de diretórios do projeto Flutter. Nela, observa-se a organização do código, incluindo os diretórios *android*, que contém os arquivos de configuração para a futura adaptação para Android, e *web*, dedicada ao ambiente Web atual. Destaca-se o diretório *assets*, responsável por armazenar arquivos estáticos, como imagens, ícones, fontes e outros recursos visuais utilizados pela aplicação. O diretório *build* contém os artefatos gerados durante o processo de compilação do projeto.

Figura 20 – Estrutura de diretórios do projeto Flutter.



Fonte: Elaborado pelo Autor (2024).

O diretório *lib* abriga o código-fonte e a arquitetura da aplicação, como o diretório *app*, onde estão localizadas as páginas e controladores, e o diretório *core*, que centraliza as entidades e lógica de negócios. Além disso, os arquivos *main_dev.dart* e *main_prod.dart* permitem diferenciar ambientes de desenvolvimento e produção.

Outro ponto importante da estrutura são os arquivos de configuração *pubspec.yaml* e *pubspec.lock*. O arquivo *pubspec.yaml* é importante para a configuração do projeto Flutter por definir as dependências e pacotes utilizados, além de especificar os recursos estáticos incluídos no diretório *assets*. Já o arquivo *pubspec.lock* é gerado automaticamente e per-

mite garantir que todas as dependências utilizadas estejam na versão correta, garantindo consistência entre os ambientes de desenvolvimento e produção.

O sistema segue a Arquitetura Limpa (ver Seção 3.10) para garantir uma clara separação de responsabilidades e facilitar futuras evoluções. Foram aplicados padrões de projeto como o MVC, além de utilizar o GetX para o gerenciamento de estados e rotas, simplificando a navegação e a manipulação dos dados na aplicação.

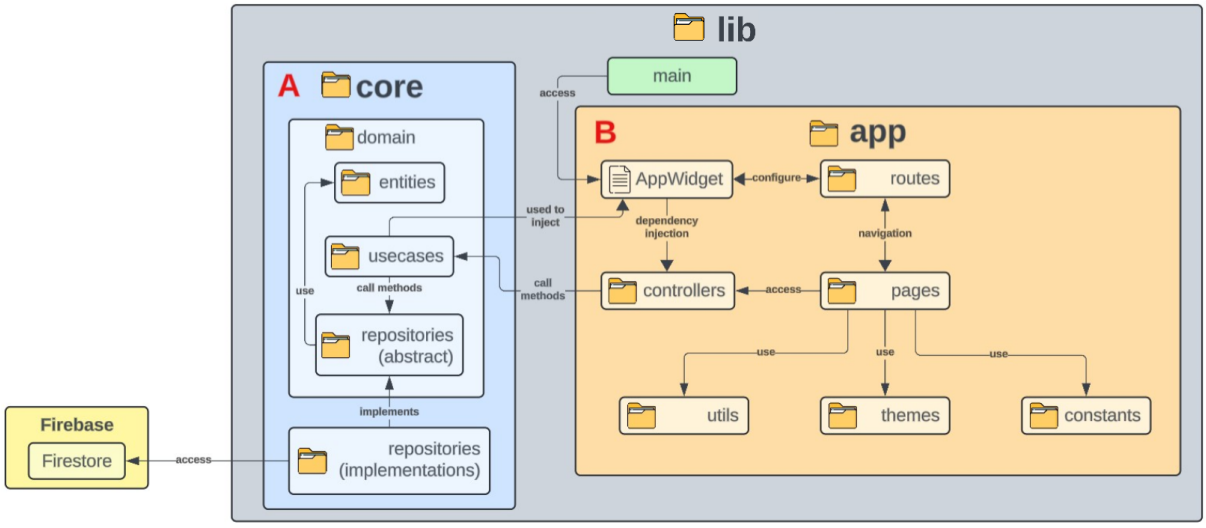
A seguir, serão detalhados os principais componentes da aplicação, incluindo a estrutura das páginas, a organização do código, a integração com serviços externos, como o Firebase, e o uso de pacotes e bibliotecas que suportam as funcionalidades do sistema.

6.2 ARQUITETURA DA APLICAÇÃO

A arquitetura do aplicativo foi projetada utilizando o conceito de Arquitetura Limpa, que oferece uma separação clara de responsabilidades entre as camadas da aplicação, facilitando a manutenção, a escalabilidade e a testabilidade do código. Essa escolha foi motivada pela necessidade de criar uma aplicação modular, com foco em alta coesão e baixo acoplamento, permitindo que cada camada do sistema evolua de forma independente.

A Figura 21 apresenta a arquitetura da aplicação, que divide o sistema em duas pastas principais: core (Figura 21-A) e *app* (Figura 21-B).

Figura 21 – Arquitetura da Aplicação Flutter.



Fonte: Elaborado pelo Autor (2024).

A pasta *core* (Figura 21-A) contém os componentes responsáveis pela lógica de negócio e abstração de dados. No interior desta, encontra-se o diretório *domain*, que armazena as *entities*, as quais representam os objetos do domínio, e os *usecases*, que encapsulam as regras de negócio da aplicação. Os *usecases* são responsáveis por realizar chamadas às interfaces definidas na camada abstrata de *repositories*, localizadas em

domain/repositories, sendo implementadas no diretório *repositories*. Vale destacar que a aplicação é completamente desacoplada de qualquer banco de dados específico, possibilitando a integração com diferentes serviços de armazenamento de dados por meio da substituição da implementação dos *repositories*, sem que haja a necessidade de modificar a lógica do restante do sistema.

A pasta *app* (Figura 21-B) reúne os elementos relacionados à interface do usuário, navegação e configuração visual. O fluxo de execução da aplicação tem início no arquivo *main*, que constitui o ponto de entrada do sistema, sendo responsável por acessar a classe *AppWidget* ao ser executado.

A classe *AppWidget* é encarregada de configurar a aplicação e gerenciar a injeção de dependências, utilizando o GetX (ver Seção 3.6) para injetar os *usecases* nos *controllers*. As *pages*, responsáveis pelas telas da aplicação, acessam os *controllers* para obter dados ou acionar funcionalidades. Os *controllers*, por sua vez, fazem chamadas diretas aos métodos dos *usecases*, sendo os responsáveis por executar a lógica de negócio. Em seguida, os *usecases* interagem com os *repositories*, os quais se encarregam de realizar o acesso ao banco de dados ou a outra fonte externa.

A navegação entre as diferentes telas da aplicação é gerenciada pela camada de *routes*, que define as rotas do sistema, facilitando as transições entre as diversas *pages*. Além disso, a pasta *app* inclui outros diretórios como *utils*, que contém funções e utilitários de uso geral; *themes*, que define as configurações visuais e o tema da aplicação; e *constants*, que armazena as constantes de configuração utilizadas em todo o sistema. Esses diretórios adicionais reforçam a modularidade da aplicação, permitindo que ela mantenha uma estrutura organizada e reutilizável, além de garantir a escalabilidade e a consistência visual do sistema.

6.2.1 Arquitetura Limpa

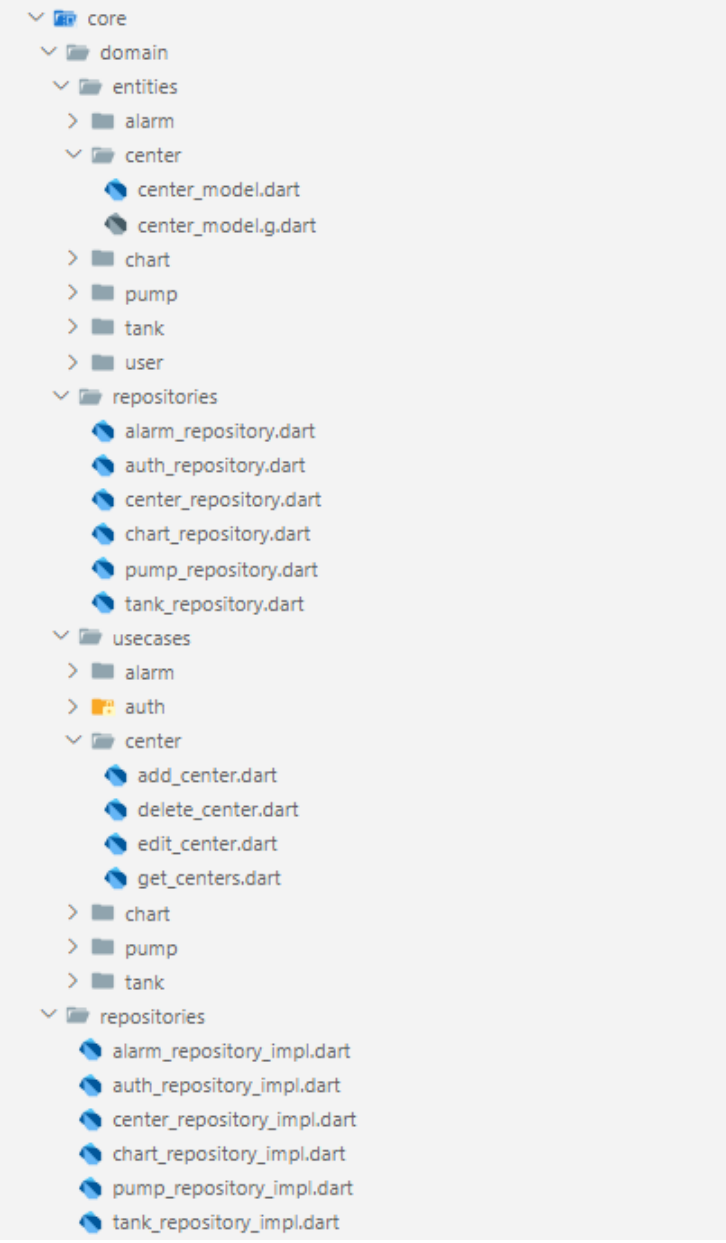
A Arquitetura Limpa é implementada em cada um dos módulos do sistema. A Figura 22 apresenta a expansão dos arquivos utilizados para as centrais, como exemplo de como essa arquitetura é aplicada na prática. Nela, observa-se a organização em três camadas:

- ***Entities* (Entidades)**: Contém as definições dos modelos de dados, como *CenterModel*, que define os atributos e comportamentos associados às centrais do sistema. As classes de entidades utilizam o pacote json_serializable, que gera automaticamente o arquivo *.g.dart* correspondente, responsável por converter os objetos entre JSON e os modelos definidos. No caso do *CenterModel*, o arquivo *center_model.g.dart* é gerado para facilitar a serialização e desserialização de dados com o Firestore.
- ***Use Cases* (Casos de Uso)**: Representam as operações que podem ser realizadas sobre as entidades. Para as centrais, há casos de uso como adicionar,

deletar, editar e ler as centrais.

- ***Repositories* (Repositórios)**: São responsáveis por intermediar o acesso aos dados e garantir que a camada de domínio não precise conhecer os detalhes de implementação do banco de dados ou de APIs externas.

Figura 22 – Diretório e arquivos utilizados para a Arquitetura Limpa das centrais.



Fonte: Elaborado pelo Autor (2024).

O diretório *core/repositories*, conforme mostrado na estrutura, contém as implementações responsáveis pela interação com o Firestore. Cada repositório tem sua interface definida na camada de domínio e uma implementação correspondente. Por exemplo:

- **alarm_repository_impl.dart**: Implementa todas as operações relacionadas aos alarmes, como a leitura de alarmes ativos ou históricos e a atualização do

estado de reconhecimento pelo síndico.

- **tank_repository_impl.dart**: Contém as funções que permitem a leitura e manipulação dos dados dos tanques, como o nível atual e a capacidade mínima.
- **center_repository_impl.dart**: Lida com os dados das centrais de monitoramento, garantindo que a aplicação possa acessar e atualizar informações sobre as centrais cadastradas.

Esses repositórios garantem que a aplicação possa realizar operações no banco de dados sem que a lógica de negócios precise conhecer detalhes de como os dados são armazenados ou recuperados. Além disso, eles seguem os princípios da Arquitetura Limpa, mantendo o sistema organizado e facilmente escalável.

6.3 PADRÕES DE PROJETO UTILIZADOS

No desenvolvimento desta aplicação, foram adotados padrões de projeto como o MVC e o GetX para o gerenciamento de estados e rotas. Esses padrões foram escolhidos para organizar o código de maneira eficiente, facilitando a manutenção e escalabilidade da aplicação.

A seguir, serão detalhados esses padrões e como foram aplicados no contexto do projeto.

6.3.1 Padrão MVC

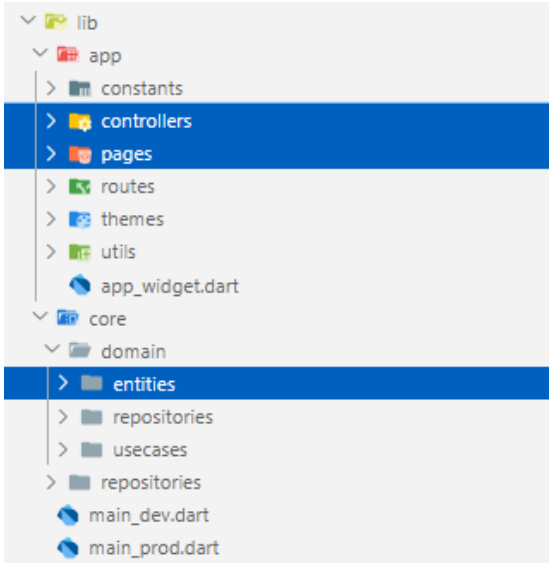
No desenvolvimento deste projeto, o padrão MVC foi adotado para organizar a aplicação em camadas, permitindo uma separação clara das responsabilidades e facilitando a manutenção e evolução do código. Embora o Flutter não siga estritamente o padrão MVC, a aplicação foi estruturada de forma que cada parte do padrão seja representada por diferentes componentes do projeto.

A Figura 23 mostra a estrutura do diretório *lib/app* onde os controladores e páginas estão organizados, além do diretório *lib/core/domain/entities* com os modelos. Esse layout reflete a aplicação do padrão MVC, com a separação dos arquivos que controlam a lógica da aplicação e a interface de usuário.

O Modelo é representado pelas classes de entidades e modelos que contêm os dados e a lógica de negócio. No contexto da aplicação, esses modelos estão localizados no diretório *core/domain/entities*. Por exemplo, o *CenterModel* define a estrutura dos dados de uma central, contendo atributos como id, *name*, *type*, entre outros, além de métodos para serialização e desserialização dos dados. Esses modelos servem como a camada de dados da aplicação, abstraindo como os dados são armazenados e manipulados.

A Visão é representada pelas telas da aplicação, que estão organizadas no diretório *pages*. Cada página exibe informações ao usuário e reage às interações, buscando dados da camada de controladores. Por exemplo, o diretório *general_analysis* contém a tela

Figura 23 – MVC no projeto Flutter.



Fonte: Elaborado pelo Autor (2024).

que exibe a análise geral das centrais monitoradas, incluindo a apresentação de alarmes e dados em tempo real.

Cada página segue a estrutura do Flutter, onde widgets são usados para construir as interfaces de usuário reativas e dinâmicas. As páginas são compostas de diferentes widgets que interagem com os dados recebidos da camada de controladores.

Os Controladores são responsáveis por gerenciar a interação entre o Modelo e a Visão, centralizando a lógica da aplicação. Eles estão localizados no diretório *controllers* e lidam com a lógica do fluxo de dados e eventos. Por exemplo, o *alarm_controller.dart* é responsável por buscar e gerenciar os dados de alarmes, interagindo com a camada de repositórios para acessar o Firestore e com as páginas para exibir os dados ao usuário.

Os controladores acessam diretamente os casos de uso localizados no diretório *core/domain/usecases*, os quais contêm a lógica específica das operações de negócio. Dessa forma, os controladores asseguram que a camada de Visão seja atualizada conforme os dados do Modelo são modificados, utilizando ferramentas como o GetX para facilitar a comunicação e o gerenciamento do estado da aplicação. Assim, atuam como intermediários, evitando que a camada de Visão precise lidar diretamente com os dados ou com a lógica de negócios.

6.3.2 Gerenciamento de Estado com GetX

O gerenciamento de estado na aplicação foi implementado utilizando o GetX (ver Seção 3.6), a injeção de dependências e o gerenciamento de rotas eficientemente. A simplicidade do GetX é especialmente valiosa em aplicações como esta, onde há múltiplos controladores e camadas de lógica. A integração do GetX com a Arquitetura Limpa e o padrão MVC proporciona uma separação clara de responsabilidades, garantindo que a

lógica de negócio, a interface de usuário e o controle de estados sejam tratados de maneira organizada e reativa.

Com GetX, a atualização do estado das diferentes partes da interface é feita de maneira reativa. Assim que algum dado importante é alterado, o GetX atualiza automaticamente as interfaces correspondentes sem a necessidade de gerenciar manualmente os ciclos de vida ou adicionar lógica complexa para observar as mudanças. Isso reduz significativamente a quantidade de código repetitivo e potencialmente falho que precisaria ser escrito para manter a interface atualizada com os dados mais recentes. Essa funcionalidade é habilitada pela classe *AppWidget*, que atua como o ponto de entrada da aplicação e utiliza a injeção de dependências do GetX para gerenciar e fornecer os controladores necessários globalmente.

A classe *AppWidget* é a primeira a ser chamada pelo arquivo *main.dart*, sendo a classe de inicialização da aplicação onde os controladores são injetados e disponibilizados globalmente na aplicação. Através do método *Get.put*(), os controladores são instanciados e colocados no contexto da aplicação. Esses controladores são, na verdade, responsáveis por encapsular os casos de uso, sendo as ações específicas do domínio da aplicação, conforme estabelecido na Arquitetura Limpa.

Os parâmetros passados para cada controlador são os casos de uso, que contêm toda a lógica de negócio relevante. Ao injetar os controladores com o GetX, garante-se que cada controlador tenha acesso direto aos casos de uso necessários para realizar ações específicas. O Código 6.1 exemplifica a instanciação do controlador para as centrais com o método *Get.put*(), utilizando a Arquitetura Limpa na classe *AppWidget*.

Código 6.1 – Exemplo de instanciação de controladores com o método Get.put().

```
1  // Instacionando a colecao "centers" do Firestore no Flutter
2  final CollectionReference _centerCollection =
       FirebaseFirestore.instance.collection("centers");
3
4  // Instacionando a implementacao do repositorio de centrais
5  CenterRepositoryImpl _centerRepositoryImpl = CenterRepositoryImpl(collectionReference:
       _centerCollection);
6
7  // Instacionando os casos de uso das centrais
8  AddCenterUseCase addCenter = AddCenterUseCase(_centerRepositoryImpl);
9  EditCenterUseCase editCenter = EditCenterUseCase(_centerRepositoryImpl);
10 DeleteCenterUseCase deleteCenter = DeleteCenterUseCase(_centerRepositoryImpl);
11 GetCentersUseCase getCenters = GetCentersUseCase(_centerRepositoryImpl);
12
13 // Instacionando o controlador de centrais com o metodo Get.put()
14 Get.put(
15     CenterController(
16       addCenter: addCenter,
17       editCenter: editCenter,
18       deleteCenter: deleteCenter,
19       getCenters: getCenters,
20     )
21 );
```

Os casos de uso seguem o princípio da Arquitetura Limpa, onde cada caso de uso representa uma operação ou ação específica da aplicação. Ao serem passados como parâmetros para os controladores, eles permitem que o controlador realize operações específicas, mantendo a separação clara entre a lógica de negócio e a interface de usuário. O Código 6.2 exemplifica a classe *CenterController* e seu construtor exigindo os casos de uso como parâmetros.

Código 6.2 – Casos de Uso na classe CenterController.

```
class CenterController extends GetxController {
  final AddCenterUseCase addCenter;
  final EditCenterUseCase editCenter;
  final DeleteCenterUseCase deleteCenter;
  final GetCentersUseCase getCenters;

  // Construtor do controlador, que recebe os use cases como parametros
  CenterController({
    required this.addCenter,
    required this.editCenter,
    required this.deleteCenter,
    required this.getCenters,
  });
}
```

Nas diferentes páginas da aplicação, os controladores são acessados utilizando o método *Get.find*(), que recupera o controlador já injetado pelo *Get.put*(). Por exemplo, para acessar o controlador de centrais em uma página de centrais, pode-se utilizar `final centerController = Get.find<CenterController>();`.

Isso permite que as páginas interajam com os controladores eficientemente, sem a necessidade de passar instâncias manualmente. O GetX garante que o controlador adequado seja recuperado de forma centralizada, permitindo uma separação clara entre a interface de usuário e a lógica de negócio.

Além disso, como os controladores utilizam os casos de uso, qualquer interação com os dados ou com a lógica de negócio, como buscar alarmes ou atualizar o estado de um dispositivo, passa por essa camada de controle, mantendo a integridade e reatividade da aplicação.

Com isso, o GetX torna o gerenciamento de estado e dependências simples e eficiente, contribuindo para um código mais limpo, modular e escalável.

6.4 GERENCIAMENTO DE ROTAS E MIDDLEWARES

O gerenciamento de rotas foi implementado utilizando o GetX para garantir uma navegação eficiente e segura entre as páginas. O GetX simplifica o processo de navegação, proporcionando uma forma centralizada de definir rotas e aplicar *middlewares*, usados para proteger determinadas rotas com verificações adicionais.

6.4.1 Gerenciamento de Rotas com GetX

As rotas da aplicação foram definidas no arquivo *get_pages.dart*, utilizando o método *GetPage*(). Este método permite criar rotas com facilidade, associando cada página a uma URL única. Adicionalmente, o GetX permite a utilização de *middlewares* para rotas específicas, garantindo que apenas usuários autorizados ou com determinadas permissões possam acessar certas páginas.

O Código 6.3 apresenta a criação de uma rota sem *middlewares*, onde a página de Login é acessada sem restrições. O atributo estático *route* da classe *Login* contém o nome do recurso usado para referenciar essa página.

Código 6.3 – Criação da rota para a página de Login.

```
GetPage(
  name: LoginPage.route,
  page: () => const LoginPage(),
);
```

Os *middlewares* são utilizados para adicionar camadas de proteção ou verificação antes que o usuário possa acessar determinadas páginas da aplicação.

- ***AuthMiddleware***: Verifica se o usuário está autenticado.
- ***HostMiddleware***: Garante que apenas usuários com a categoria "Host" tenham acesso a certas rotas administrativas.
- ***HomeMiddleware***: Verifica se o controlador de centrais conseguiu obter as centrais cadastradas no Firestore antes de permitir o acesso à página principal da aplicação, caso contrário tenta realizar a sincronização com o Firestore novamente.

Esses *middlewares* são aplicados utilizando a propriedade *middlewares* em cada rota. O Código 6.4 apresenta a criação da rota para a página de *Home* com *middlewares* aplicados, onde a rota da página inicial *HomePage* só poderá ser acessada se o usuário estiver autenticado (verificado pelo *AuthMiddleware*) e se houver centrais no controlador de centrais (verificado pelo *HomeMiddleware*).

Código 6.4 – Criação da rota para a página de Home.

```
GetPage(
  name: HomePage.route,
  page: () => const HomePage(),
  middlewares: [AuthMiddleware(), HomeMiddleware()],
);
```

6.4.2 Navegação com *Get.toNamed*() e *Get.offAllNamed*()

A navegação entre as páginas é facilitada pelas funções do GetX: *Get.toNamed*() e *Get.offAllNamed*(). Estas funções permitem a navegação com parâmetros, além de

possibilitar o gerenciamento do histórico de navegação. Enquanto *Get.toNamed*() mantém o histórico de navegação, *Get.offAllNamed*() remove todas as páginas anteriores, garantindo que o usuário não possa voltar para páginas anteriores após a navegação.

O trecho `Get.toNamed(HomePage.route);` exemplifica a navegação utilizando *Get.toNamed*(), onde o usuário é direcionado para a página inicial *HomePage* e o histórico de navegação é mantido.

A utilização de *Get.offAllNamed*() é mais adequada em cenários onde não se deseja que o usuário retorne a telas anteriores, como após o processo de login, onde o usuário deve ser redirecionado para a página inicial e não mais ter acesso à tela de login ou de autenticação.

6.5 INTEGRAÇÃO COM SERVIÇOS EXTERNOS

Nesta subseção, será descrita a integração com os serviços externos utilizados no aplicativo, com destaque para os serviços do Firebase, além dos pacotes que aprimoram a funcionalidade e a usabilidade do sistema.

6.5.1 Integração com Firebase Services

O aplicativo faz uso extensivo dos serviços fornecidos pelo Firebase, sendo fundamentais para o funcionamento do sistema de monitoramento:

- **Firebase Auth**: O pacote `firebase_auth` é usada para gerenciar o login e a autenticação dos usuários. Através desse serviço, o sistema permite a criação de contas e o controle de acesso baseado em e-mail e senha, garantindo a segurança das informações.
- **Firebase Core**: O pacote `firebase_core` é a base para inicializar os serviços Firebase no aplicativo. Sem ele, as integrações com o Firestore, Firebase Auth e outros serviços do Firebase não poderiam ser utilizados.
- **Cloud Firestore**: O pacote `cloud_firestore` é utilizado como a base de dados principal do aplicativo, onde são armazenadas as informações sobre centrais, dispositivos, alarmes e dados históricos. A comunicação com o Firestore permite a consulta e a atualização em tempo real desses dados.

6.5.2 Uso de Outros Pacotes

Além da integração com o Firebase, o projeto utiliza uma série de pacotes que auxiliam na construção de uma interface rica e no desempenho geral da aplicação:

- **get**: O get é usado para gerenciamento de estado e navegação. Ele permite o controle eficiente de fluxos de navegação entre as páginas, além de facilitar a atualização da interface com base nas alterações de estado.

- **lottie**: O pacote `lottie` é utilizada para animações no aplicativo. Com suporte para arquivos JSON animados, ela permite criar interações visuais dinâmicas e modernas para melhorar a experiência do usuário.
- **dio**: Para realizar requisições HTTP, o pacote `dio` é utilizado, proporcionando uma interface robusta e eficiente para comunicação com APIs externas.
- **syncfusion_flutter_pdf**: Usado para a criação e exportação de relatórios em PDF, o pacote `syncfusion_flutter_pdf` facilita a geração de relatórios detalhados com base nos dados de alarmes e históricos do sistema.
- **google_fonts**: Para garantir consistência visual no design, o `google_fonts` é utilizado para acessar e aplicar fontes personalizadas no aplicativo.
- **json_serializable**: Esse pacote é utilizado para serializar e desserializar os dados de modelos para JSON, facilitando a comunicação com o Firestore e o Realtime Database. Ele gera automaticamente código para mapear os objetos Dart para JSON, como nas entidades de alarmes, centrais e dispositivos.
- **flutter_map** e **latlong2**: Utilizados para a exibição de mapas interativos, permitindo a visualização geográfica das centrais e dispositivos no sistema de monitoramento. O `flutter_map` fornece a base para a exibição do mapa, enquanto o `latlong2` facilita o cálculo de coordenadas e distâncias entre pontos.
- **dropdown_button2**: Esse pacote oferece um componente aprimorado de menus suspensos (*dropdowns*), utilizado em várias partes do aplicativo para facilitar a seleção de opções pelos usuários.
- **msh_checkbox**: Utilizado para criação de caixas de seleção personalizadas (*checkboxes*), usadas em formulários e na interface de configuração de preferências, proporcionando uma experiência visual mais interativa e amigável.
- **flutter_animate**: Para a aplicação de animações dinâmicas, o pacote `flutter_animate` é usado, adicionando transições suaves e interações mais fluídas entre elementos da interface, melhorando a experiência do usuário.
- **fl_chart**: O pacote `fl_chart` é empregado na exibição de gráficos, permitindo a visualização de dados históricos e tendências no aplicativo. Gráficos de barras, linhas e outros tipos de representações visuais auxiliam os usuários a analisar os dados de maneira clara e intuitiva.

Esses pacotes fornecem as bases para um sistema robusto e visualmente atraente, garantindo uma melhor experiência ao usuário.

6.6 HOSPEDAGEM E *DEPLOY* NO FIREBASE HOSTING (SAAS)

A utilização de serviços de hospedagem baseados em *cloud* tem se tornado uma escolha popular em projetos que demandam escalabilidade e facilidade de manutenção. O

Firebase Hosting, uma solução SaaS, oferece uma plataforma confiável para a publicação de aplicações web, integrando de forma eficiente com outros serviços do Firebase, como banco de dados e funções em nuvem. Sua simplicidade de configuração e ferramentas automatizadas permitem uma gestão ágil do processo de *deploy*, reduzindo o tempo e o esforço necessários para colocar a aplicação em produção.

6.7 FIREBASE HOSTING

A hospedagem da aplicação foi realizada utilizando o Firebase Hosting, conforme explicado na Seção 3.4.5. O processo de implantação começou com a instalação da ferramenta Firebase CLI, através do seguinte comando:

```
$ npm install -g firebase-tools
```

Após a instalação da CLI (Command Line Interface), foi configurado o ambiente de hospedagem utilizando o comando:

```
$ firebase init hosting
```

Nesse estágio, foram feitas as configurações iniciais, como a seleção da pasta de distribuição do aplicativo (`build/web`), onde ficam os arquivos gerados pelo *build* do Flutter para a Web. A seguir, o comando `$ firebase deploy` foi utilizado para realizar o deploy da aplicação no Firebase Hosting:

```
$ firebase deploy
```

Com esse comando, os arquivos do projeto foram enviados para os servidores do Firebase e a aplicação passou a estar acessível ao público por meio de uma URL gerada pelo próprio serviço. O Firebase também oferece suporte a HTTPS automaticamente, garantindo segurança no tráfego de dados.

A configuração de hospedagem é gerenciada pelo arquivo `firebase.json`, onde foram definidas diretivas adicionais para controle de rotas e redirecionamentos. Esse arquivo permite a personalização de aspectos como cache, reescritas de URL e segurança da aplicação.

Ao final do processo, o deploy foi concluído com sucesso, e a aplicação passou a ser servida diretamente pelos servidores do Firebase, sem a necessidade de configurar servidores físicos ou máquinas virtuais.

Essa abordagem de hospedagem é classificada como SaaS, já que o Firebase Hosting gerencia automaticamente a infraestrutura, a escalabilidade e a manutenção, permitindo que a equipe de desenvolvimento foque exclusivamente no código da aplicação.

6.8 CICLO DE DEPLOY E ATUALIZAÇÕES

O ciclo de deploy e atualizações do sistema segue um fluxo padronizado e eficiente, garantindo a entrega contínua de novas funcionalidades e correções, com o mínimo de impacto para os usuários. O processo inicia-se com a implementação das melhorias ou ajustes no código-fonte, seguido de testes locais para assegurar a qualidade e o funcionamento adequado da aplicação. Após essa validação, a próxima etapa consiste na geração dos arquivos de produção da aplicação para a versão web, através do comando:

```
$ flutter build web
```

Esse comando compila o projeto Flutter e gera arquivos como HTML, CSS e JavaScript, necessários para a execução da aplicação no ambiente web. Com os arquivos prontos, inicia-se o processo de deploy para o Firebase Hosting, utilizando o seguinte comando:

```
$ firebase deploy
```

Esse comando envia a versão mais recente da aplicação para os servidores do Firebase Hosting, substituindo automaticamente os arquivos da versão anterior. O Firebase Hosting, além de controlar o cache de maneira eficiente, gerencia as rotas de navegação, garantindo que a nova versão da aplicação esteja disponível de forma imediata para os usuários, sem causar interrupções significativas no serviço.

O ciclo de deploy é desenhado para ser ágil e frequente, possibilitando atualizações constantes à medida que novas funcionalidades são implementadas ou correções são necessárias. O Firebase Hosting, com sua infraestrutura baseada na nuvem, facilita não apenas a implantação de novas versões, mas também garante a escalabilidade do sistema.

Por fim, vale ressaltar que esse processo de deploy e atualização contínua é característico de uma solução de *Software as a Service* (SaaS), na qual a infraestrutura, a escalabilidade e a manutenção da aplicação são geridas por um provedor de serviços em nuvem, permitindo que os usuários tenham sempre acesso à versão mais recente do sistema, sem necessidade de intervenção manual.

7 RESULTADOS E DISCUSSÃO

Este capítulo apresenta os resultados obtidos com a implementação do sistema supervisório, incluindo a comunicação MQTT, armazenamento de dados no Firestore e interface de monitoramento. São discutidas as funcionalidades principais, como a geração de alarmes e notificações, além da escalabilidade e desempenho da aplicação. Por fim, aborda-se uma análise de custos e o plano de negócios da solução desenvolvida.

7.1 COMUNICAÇÃO MQTT E ARMAZENAMENTO DE DADOS NO FIRESTORE

A validação mínima da comunicação MQTT foi realizada para verificar a correta recepção e processamento dos dados enviados pelos dispositivos ao sistema supervisório. Para a condução dos testes, utilizou-se um *broker* público de livre acesso para o envio constante de dados simulados em determinado intervalo de tempo.

A configuração do teste permitiu a simulação do fluxo de dados provenientes de sensores físicos, representando as condições reais de operação e possibilitando a observação da integridade da comunicação entre os dispositivos e o banco de dados. Cada dispositivo e central foram previamente registrados no sistema, recebendo identificadores únicos que permitiram a associação correta de cada mensagem aos componentes monitorados no sistema supervisório.

Para garantir que todas as mensagens foram tratadas adequadamente, foi implementado um controle de mensagens enviadas e recebidas. Uma aplicação em Node.js foi configurada para enviar mensagens contendo a estrutura JSON ao tópico do MQTT, simulando cenários reais de operação dos dispositivos. Essa aplicação gerava dados aleatórios para os dispositivos e centrais, publicando mensagens em intervalos de tempo constantes no *broker* MQTT. Esse processo permitiu avaliar o comportamento do sistema em condições controladas e garantir que todas as mensagens fossem tratadas corretamente pelo *subscriber*.

O Código 7.1 apresenta uma das estruturas JSON publicadas no tópico do *broker* para o teste.

Código 7.1 – Estrutura JSON publicada no *broker* para teste.

```
{
    "timestamp": 1729968253,
    "centers": [
        {
            "id": 2,
            "pumps": [
                {
                    "id": 1,
                    "value": 0
                },
                {
                    "id": 2,
                    "value": 1
```

```
                }
            ],
            "tanks": [
                {
                    "id": 1,
                    "value": 15
                },
                {
                    "id": 2,
                    "value": 7
                }
            ]
        },
        {
            "id": 3,
            "pumps": [
                {
                    "id": 3,
                    "value": 1
                }
            ],
            "tanks": [
                {
                    "id": 3,
                    "value": 45
                }
            ]
        }
    ]
}
```

Durante os testes, foi verificado que o número de mensagens publicadas pelo *broker* coincidia exatamente com o número de mensagens processadas pelo *subscriber*, conforme registrado nos logs gerados pelo Heroku. Esses logs detalham cada etapa do processamento, permitindo confirmar que nenhuma mensagem foi perdida ou descartada durante a execução.

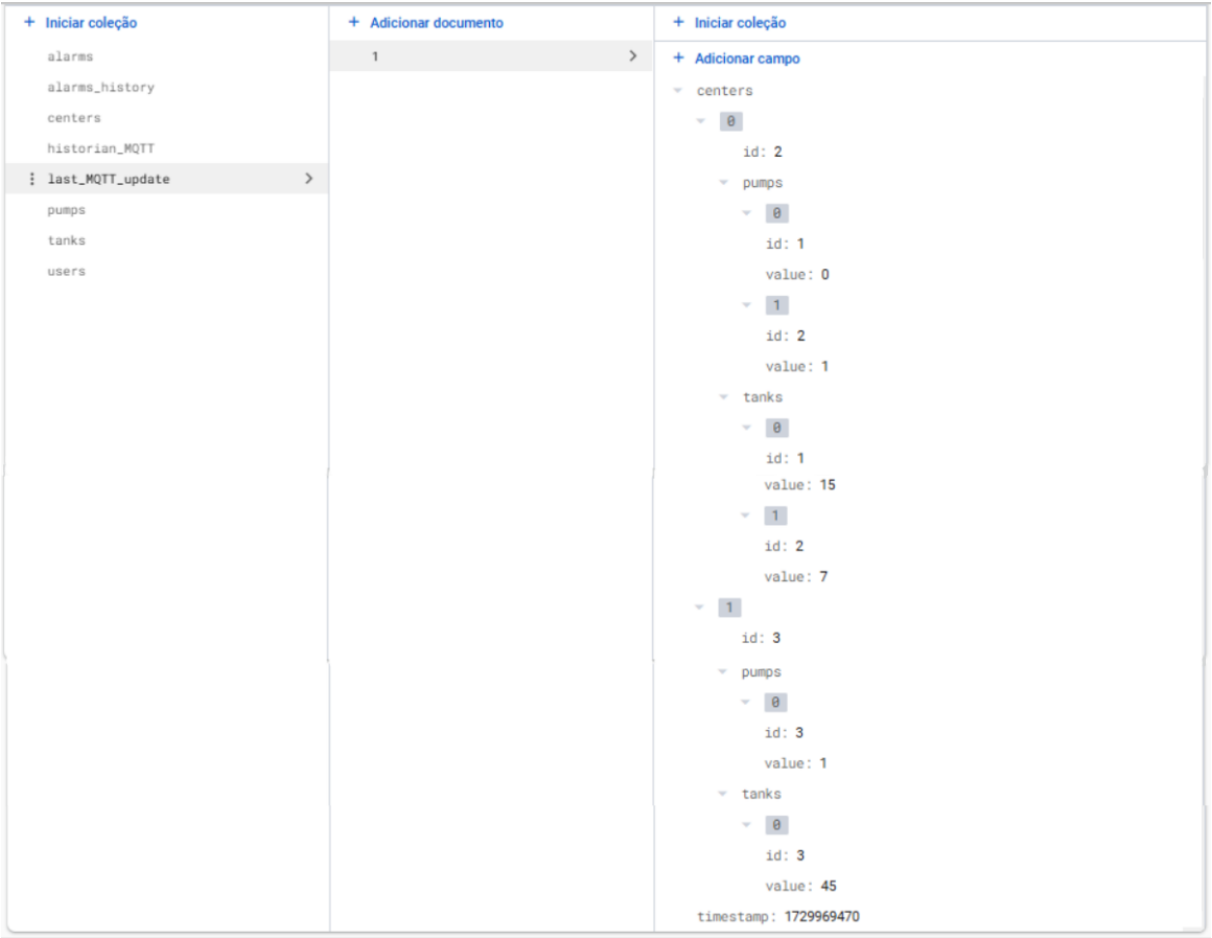
O sistema conseguiu processar todas as mensagens recebidas, respeitando a ordem de envio e mantendo o registro atualizado no Firestore. A Figura 24 ilustra os logs gerados durante esses testes, enquanto a Figura 25, que exibe a coleção *last_mqtt_update* no Firestore, confirma que os dados foram armazenados corretamente, preservando as informações de cada dispositivo e central.

Os resultados da validação indicam que o sistema apresenta capacidade de gerenciar a comunicação MQTT, registrando as informações necessárias para o monitoramento dos dispositivos em operação. A estrutura JSON utilizada e o processo de tratamento dos dados pelo *subscriber* demonstraram-se adequados para o gerenciamento de informações provenientes de diferentes dispositivos e centrais, garantindo a continuidade do monitoramento.

Figura 24 – Logs gerados pelo Heroku ao receber dados do MQTT.

Fonte: Elaborado pelo Autor (2024).

Figura 25 – Coleção *last_mqtt_update* no Firestore.



Fonte: Elaborado pelo Autor (2024).

7.2 INTERVALO DE AMOSTRAS NO HISTÓRICO

Após a inserção dos dados recebidos dos dispositivos no banco de dados, torna-se necessário gerenciar de maneira eficiente o armazenamento e a organização dessas informações para evitar custos adicionais no uso do Firestore. Nesse contexto, um dos aspectos mais críticos é a gestão do histórico de dados.

A escolha de um intervalo de tempo entre amostras no histórico está diretamente relacionada com as limitações impostas pelo Firestore, como o limite de 1 MB por documento, além dos custos associados ao armazenamento e operações de leitura e escrita que aumentam conforme se ultrapassam os limites do plano gratuito.

Dessa maneira, realiza-se uma análise detalhada da quantidade de informações que seriam armazenadas no histórico com base na estrutura JSON de *timestamps* comentada anteriormente.

A análise apresenta a quantidade de adições de dados ao longo de diferentes períodos (diário, semanal, mensal, anual e dois anos), além de mostrar o impacto que o intervalo de tempo (*timeframe*) entre as amostras exerce no consumo de armazenamento no Firestore.

Os cálculos foram realizados considerando que cada adição ao histórico ocupa 1.428 bytes para um conjunto de 30 dispositivos monitorados.

A Tabela 1 mostra a quantidade de vezes que os dados são adicionados ao Firestore, dependendo do intervalo de tempo escolhido para capturar as amostras (1 minuto, 5 minutos, etc.). Quanto maior o intervalo, menor o número de adições de dados, impactando diretamente o uso do armazenamento.

Tabela 1 – Quantidade de adições de dados no Firestore para diferentes *timeframes*.

Timeframe (min)	**Diária**	**Semanal**	**Mensal**	**Anual**	**2 anos**
1	1.440	10.080	40.320	483.840	967.680
5	288	2.016	8.064	96.768	193.536
10	144	1.008	4.032	48.384	96.768
15	96	672	2.688	32.256	64.512
30	48	336	1.344	16.128	32.256
45	32	224	896	10.752	21.504
60	24	168	672	8.064	16.128
120	12	84	336	4.032	8.064

A Tabela 2 calcula o total de armazenamento (em MB) necessário para manter essas adições no Firestore. A relação entre o intervalo de amostragem e o consumo de armazenamento fica clara: intervalos menores aumentam significativamente o uso de espaço ao longo do tempo.

Tabela 2 – Quantidade de armazenamento utilizado (em MB) no Firestore para diferentes *timeframes*.

Timeframe (min)	**Diária**	**Semanal**	**Mensal**	**Anual**	**2 anos**
1	2,05632	14,39424	57,57696	690,92352	1381,84704
5	0,411264	2,878848	11,515392	138,184704	276,369408
10	0,205632	1,439424	5,757696	69,092352	138,184704
15	0,137088	0,959616	3,838464	46,061568	92,123136
30	0,068544	0,479808	1,919232	23,030784	46,061568
45	0,045696	0,319872	1,279488	15,353856	30,707712
60	0,034272	0,239904	0,959616	11,515392	23,030784
120	0,017136	0,119952	0,479808	5,757696	11,515392

Os dados apresentados na Tabela 2 mostram que, a partir de um intervalo de 5 minutos entre as amostras, o tamanho diário dos documentos armazenados no Firestore permanece abaixo de 1 MB, com um consumo diário de aproximadamente 0,41 MB. Isso garante que o limite de 1 MB por documento imposto pelo Firestore não seja ultrapassado, mesmo ao longo de um dia completo de monitoramento de 30 dispositivos.

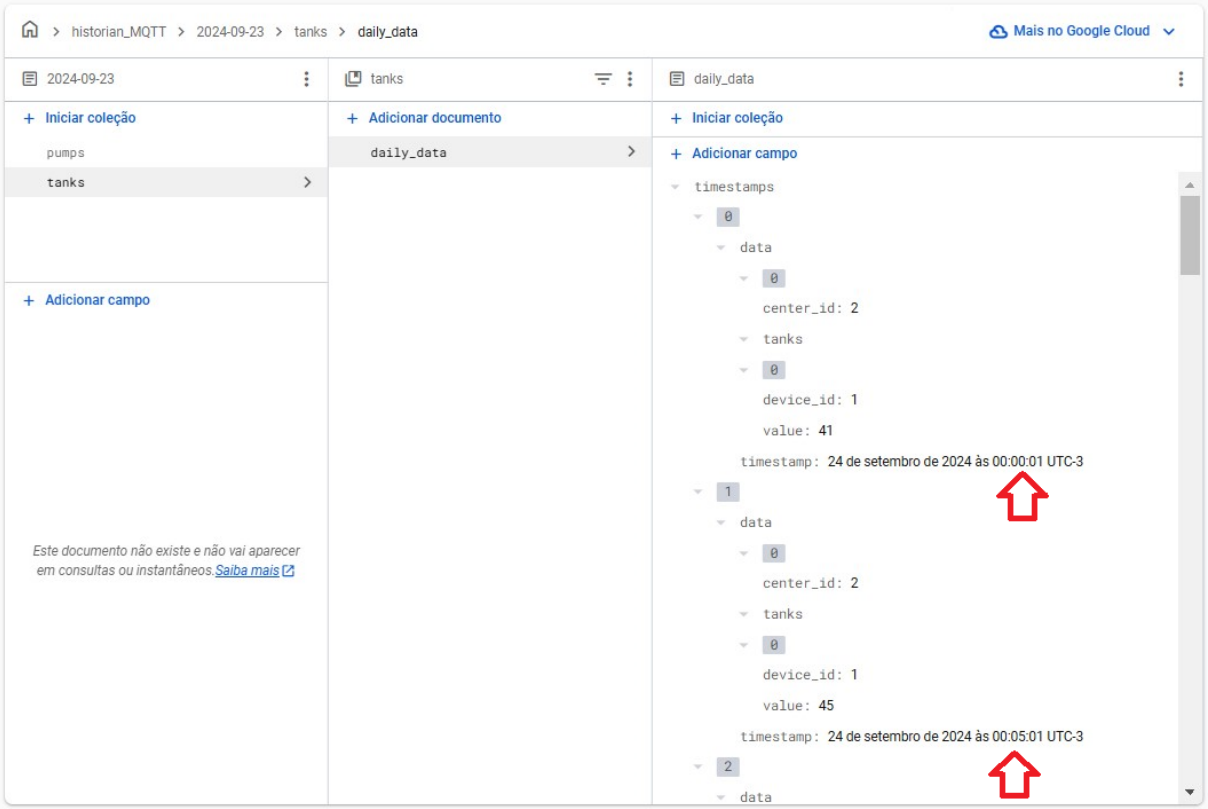
A escolha do intervalo mínimo de 5 minutos para o registro das amostras foi, portanto, baseada na necessidade de evitar o risco de que o documento diário atinja o limite de armazenamento do Firestore. Intervalos menores, como 1 minuto, resultariam em um uso diário de 2,05 MB, o que rapidamente ultrapassaria esse limite.

Além disso, para garantir ainda mais espaço dentro de cada documento *daily_data*, foi implementada a divisão dos dados em subcoleções separadas: *pumps* (bombas) e *tanks* (tanques). Essa separação evita que todos os dados de dispositivos sejam armazenados em uma única subcoleção, o que poderia levar ao aumento do tamanho do documento mais rapidamente. Com essa divisão, os dados de cada tipo de dispositivo são distribuídos entre as subcoleções, proporcionando uma margem de segurança nos documentos diários e reduzindo a probabilidade de que eles atinjam o limite de 1 MB.

A Figura 26 apresenta os registros do tanque com ID 1 cadastrado na central com ID 2, espaçados pelo intervalo de tempo de 5 minutos.

Essa organização permite que o sistema supervisório registre de forma contínua os dados de todos os dispositivos monitorados ao longo do dia, ao mesmo tempo, em que garante conformidade com as restrições de armazenamento do Firestore e preserva o desempenho do banco de dados ao longo do tempo.

Figura 26 – Intervalo de 5 minutos entre registros na coleção *historian_mqtt* no Firestore.



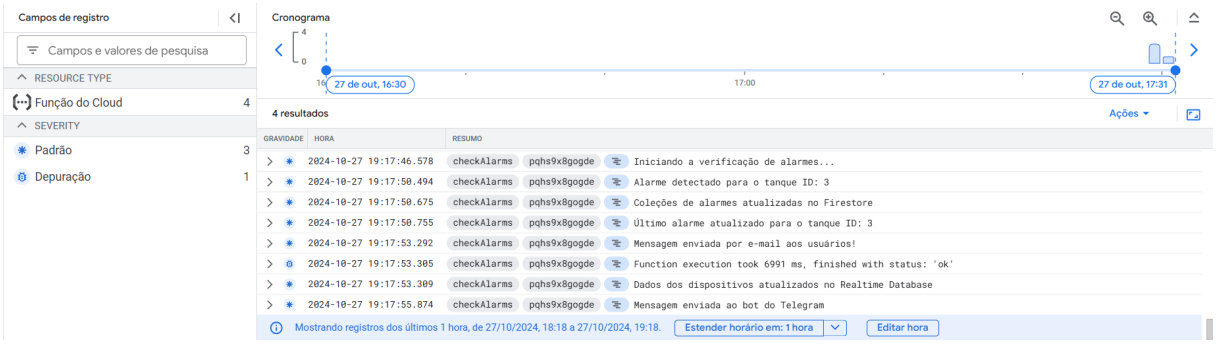
Fonte: Elaborado pelo Autor (2024).

7.3 GERAÇÃO DE ALARMES E NOTIFICAÇÕES

Os testes de geração de alarmes e notificações foram conduzidos para avaliar a efetividade e a precisão do sistema em identificar e comunicar eventos críticos, como falhas em bombas ou níveis de tanque abaixo dos mínimos aceitáveis.

Durante os testes, as Firebase Functions mostraram-se eficazes em detectar atualizações na coleção monitorada e em acionar notificações de forma rápida. Como demonstrado na Figura 27, que exibe os *logs* da execução da função *checkAlarms* no Console Google Cloud, a função é ativada após atualizações de dados no Firestore. Durante sua execução, verifica a necessidade de ativação ou resolução de alarmes com base nos dados mais recentes. Ao detectar um nível baixo no tanque de ID 3, o alarme correspondente foi acionado, atualizando as coleções de alarmes no Firestore e a data do último alarme gerado para o dispositivo. Além disso, mensagens foram enviadas via Telegram e e-mail para os usuários.

Figura 27 – *Logs* da execução da função *checkAlarms* no Console Google Cloud.



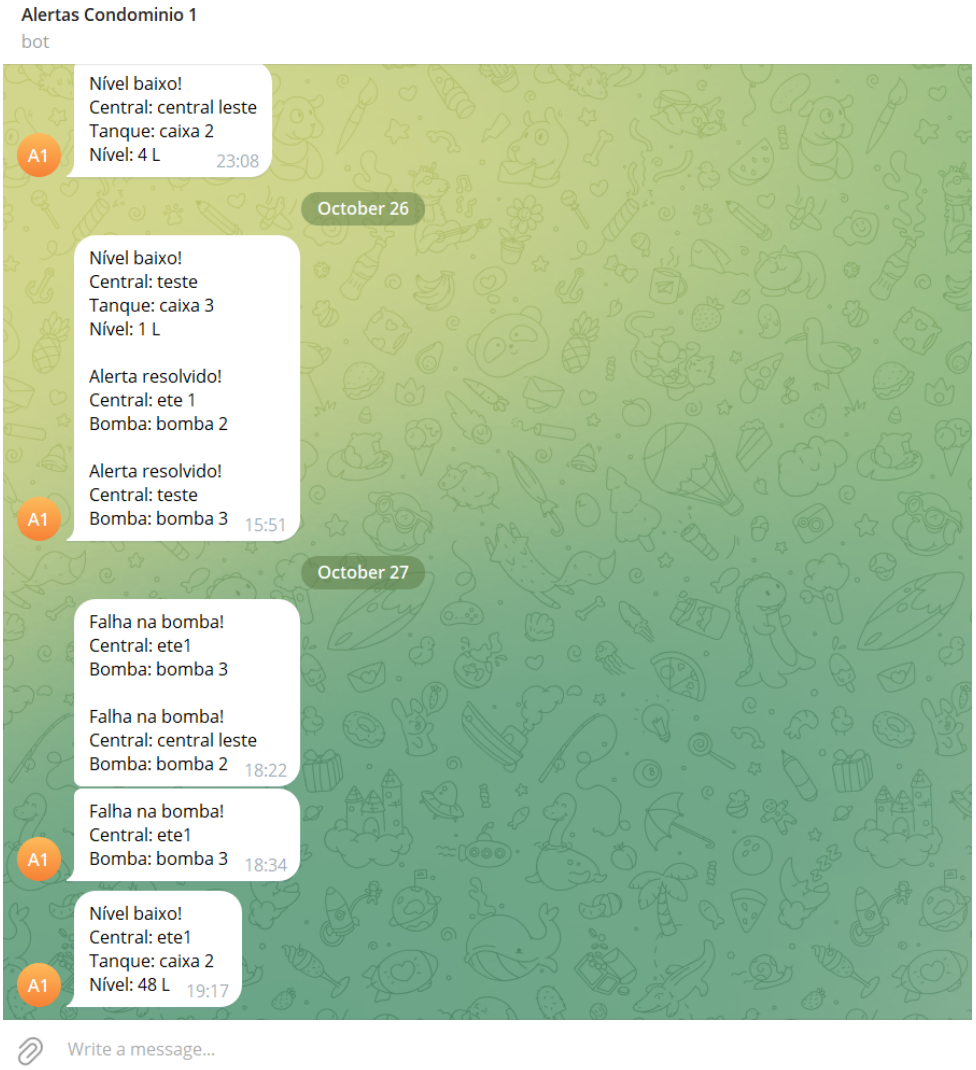
Fonte: Elaborado pelo Autor (2024).

A Figura 28 apresenta capturas de tela das mensagens enviadas ao *bot* no Telegram, onde alertas e resoluções são publicados em tempo real, gerando notificações diretamente nos dispositivos móveis dos usuários. Durante os testes, observou-se que cada alerta foi enviado com informações detalhadas sobre o evento, incluindo o tipo de falha, a central e o dispositivo afetados, bem como o nível exato onde o alerta foi acionado nos casos de monitoramento de tanques.

As mensagens de resolução mostraram-se igualmente eficazes, sendo disparadas automaticamente quando as condições de alarme retornavam aos níveis normais. Esse processo garantiu aos usuários um acompanhamento contínuo das condições do sistema e uma confirmação precisa da resolução dos eventos críticos.

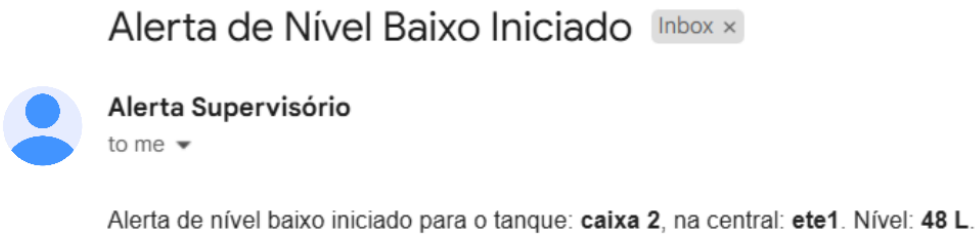
Além das notificações por Telegram, os alertas enviados por e-mail (Figura 29) também se mostraram eficientes. Cada e-mail contém informações claras sobre o evento, para proporcionar aos usuários uma alternativa adicional de recebimento de alertas. Durante os testes, verificou-se que as notificações por e-mail foram entregues consistentemente num intervalo de 5 a 10 segundos após o acionamento da Firebase Function correspondente, dependendo das condições da rede, assim como as mensagens no Telegram.

Figura 28 – Alertas enviados no *bot* do Telegram.



Fonte: Elaborado pelo Autor (2024).

Figura 29 – Alerta enviados no e-mail.



Fonte: Elaborado pelo Autor (2024).

Para evitar redundâncias e controlar o uso do Firestore, a configuração das Firebase Functions foi ajustada para monitorar exclusivamente as alterações nos estados dos alertas. Esse ajuste garantiu que o sistema de alarmes focasse apenas na geração de notificações quando um novo alerta fosse disparado, evitando o envio de notificações repetitivas para dispositivos que já estivessem em estado de falha ou com nível abaixo do permitido. Dessa forma, um alarme é emitido apenas quando o estado de alerta é inicializado ou, caso o dispositivo tivesse seu alarme resolvido e voltasse a apresentar um problema, a notificação seria reativada.

Esse ajuste auxiliou no gerenciamento do histórico de eventos no Firestore, sendo que apenas os alarmes ativos permaneceram na coleção *alarms*, enquanto os alarmes resolvidos eram automaticamente transferidos para a coleção *alarms_history*. Esse mecanismo de arquivamento minimizou o impacto sobre o desempenho do sistema e possibilitou uma consulta rápida e eficiente aos alarmes ativos.

Em conclusão, os testes de geração de alarmes e notificações demonstraram que o sistema possui a capacidade de monitorar condições críticas de forma contínua e precisa, enviando notificações confiáveis por múltiplos canais. A integração eficiente das Firebase Functions com o Firestore e o uso de canais de comunicação múltiplos asseguraram que os usuários fossem informados rapidamente sobre quaisquer alterações, contribuindo para uma resposta ágil às necessidades de manutenção e segurança operacional.

7.4 PÁGINAS DA APLICAÇÃO

Como resultado da implementação da aplicação Web, obteve-se um sistema com diversas páginas, cada uma responsável por uma funcionalidade específica no ambiente de supervisão e controle das centrais e dispositivos. Observou-se que a organização das páginas permite ao usuário monitorar alarmes, consultar históricos, gerar relatórios e configurar preferências intuitivamente, cumprindo o que o sistema se propõem.

A seguir, serão descritas as principais páginas da aplicação, explicando suas funcionalidades e a integração com os serviços externos para uma operação eficiente do sistema supervisório.

7.4.1 *Login*

A página de *Login* é a porta de entrada do sistema, onde os usuários precisam se autenticar para acessar o ambiente de supervisão e controle. Nela, são utilizados campos para inserir e validar o e-mail e a senha do usuário. O controle da página é gerenciado por diversos controladores do GetX, como o *AuthController*, que cuida da autenticação, o *ThemeController*, responsável por controlar o tema da aplicação, e o *SimpleUIController*, que gerencia ações relacionadas à interface, como a visibilidade do campo de senha.

A página adapta seu *layout* conforme o tamanho da tela, oferecendo uma interface

adequada tanto para dispositivos pequenos quanto grandes. Quando o formulário de *login* é submetido, ele realiza a validação dos campos e, em caso de sucesso, tenta autenticar o usuário por meio do *AuthController*. Se a autenticação for bem-sucedida, o usuário é redirecionado para a página de sincronização. Caso contrário, uma mensagem de erro é exibida na tela.

Para garantir uma boa experiência de usuário, a página exibe um indicador de carregamento enquanto a autenticação está sendo processada. A validação dos dados é feita diretamente no formulário, incluindo a verificação de um e-mail válido e uma senha com pelo menos 6 caracteres.

O fluxo de navegação utiliza o método *Get.offAllNamed*() para remover as páginas anteriores do histórico, garantindo que o usuário não possa retornar à página de *Login* após a autenticação bem-sucedida.

A Figura 30 ilustra a página de *Login* referenciada acima.

Figura 30 – Página de *Login*.

Fonte: Elaborado pelo Autor (2024).

7.4.2 *Home*

A implementação da página *Home* resultou em uma estrutura flexível e adaptável que serve como contêiner para as demais páginas do sistema. A *Home* é mais do que apenas uma página de exibição. Ela atua como uma estrutura que contém o cabeçalho, uma barra lateral de navegação (*sidebar*) e um espaço dedicado ao conteúdo das páginas, como a Análise Geral, Análise Individual, Alarmes, Histórico de Dados, entre outras.

A *sidebar* exibe opções de navegação principais, com menus que permitem ao usuário acessar rapidamente diferentes seções da aplicação. Ela também conta com itens visíveis apenas para os usuários com a função de “Host”, como a administração de centrais e dispositivos, além da criação de novos usuários.

O cabeçalho é dinâmico e fornece informações como a data atual, além de botões para acessar o canal do Telegram onde são enviados os alarmes e um ícone visual para indicar as notificações de alarmes quando ativos.

O conteúdo exibido na página *Home* varia conforme a opção selecionada na *sidebar*. Dependendo da escolha, diferentes páginas como análise de dispositivos, relatórios ou suporte são carregadas no espaço de conteúdo da *Home*. Assim, a *Home* funciona como um contêiner para essas páginas secundárias, facilitando a navegação e permitindo uma visão geral do sistema de monitoramento.

A Figura 31 ilustra a página *Home* com comentários do que já foi citado acima.

Figura 31 – Página de *Home*.

Fonte: Elaborado pelo Autor (2024).

7.4.3 Análise Geral

A página de Análise Geral mostrou ser uma das principais funcionalidades do sistema, permitindo a visualização do estado dos dispositivos monitorados. Sua interface é dividida em dois componentes centrais: o mapa do condomínio e a tabela de alarmes ativos.

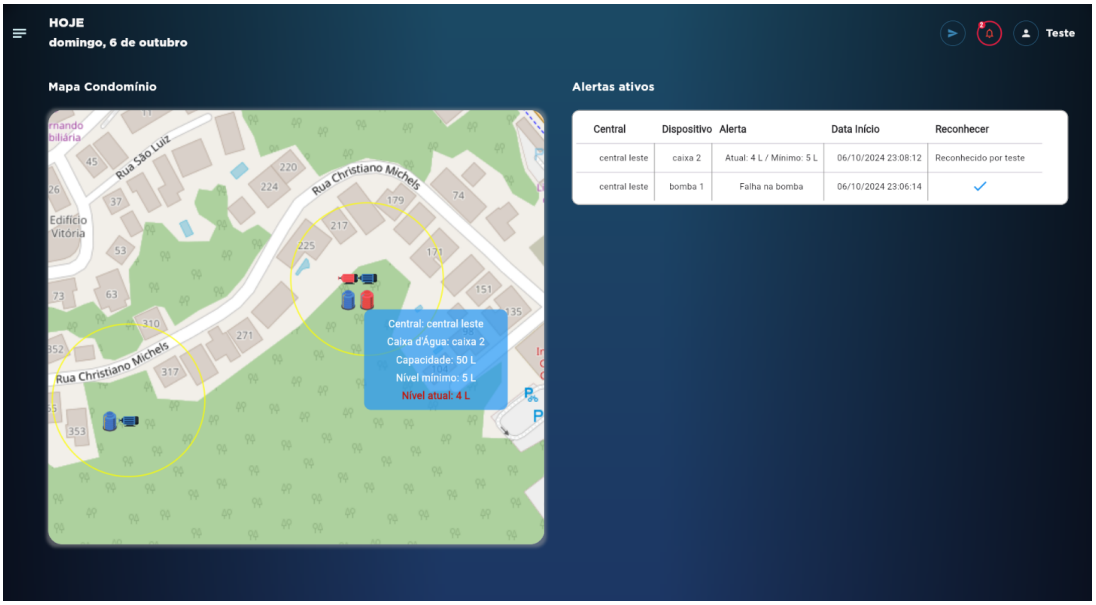
O mapa exibe as centrais e seus dispositivos de forma geográfica, utilizando o pacote *flutter_map* para renderização. As centrais são representadas por áreas circulares, e seus dispositivos são dispostos dentro desses círculos correspondentes. A interação com o mapa inclui funcionalidades como *zoom*, movimentação e rotação. Além disso, *pop-ups* são exibidos ao posicionar o cursor sobre os dispositivos, fornecendo informações detalhadas sobre o estado atual de cada um.

Ao lado do mapa, a tabela de alarmes ativos apresenta uma lista dos dispositivos que estão em estado de alerta, contendo informações como a central, o dispositivo, o

tipo de alerta e a data de início. Essa tabela é atualizada em tempo real por meio de dados transmitidos via Firebase (*stream*). Ademais, a interface permite que usuários com permissões adequadas reconheçam os alertas, indicando que será o responsável por ir ao local do dispositivo em questão a fim de resolver o problema.

A Figura 32 apresenta a interface resultante da página de Análise Geral, conforme descrito nesta seção. Nela, observa-se um exemplo de *pop-up* exibido ao posicionar o cursor do mouse sobre o ícone de um dos tanques cadastrados no sistema. No caso ilustrado, o tanque encontra-se em estado de alerta, uma vez que o nível atual de água registrado é de 4 litros, enquanto o nível mínimo estabelecido para operação segura é de 5 litros.

Figura 32 – Página de Análise Geral.



Fonte: Elaborado pelo Autor (2024).

Além disso, observa-se que o alarme exibido no primeiro registro da tabela de alarmes ativos foi devidamente reconhecido pelo usuário “teste”, que agora se torna o responsável pela análise e resolução do caso. Em contrapartida, o segundo alarme permanece sem reconhecimento por parte de qualquer usuário. Para uma visualização mais ampla da página, a *sidebar* foi temporariamente ocultada, podendo ser reexibida a qualquer momento mediante clique no ícone localizado no canto superior esquerdo da tela.

7.4.4 Análise Individual

A página de Análise Individual resultou em uma interface voltada para o monitoramento detalhado dos dispositivos cadastrados no sistema, oferecendo uma visão individualizada de cada equipamento. Essa página permite que o usuário selecione uma central específica, visualize os dispositivos conectados e obtenha informações detalhadas sobre o desempenho e o estado de cada um.

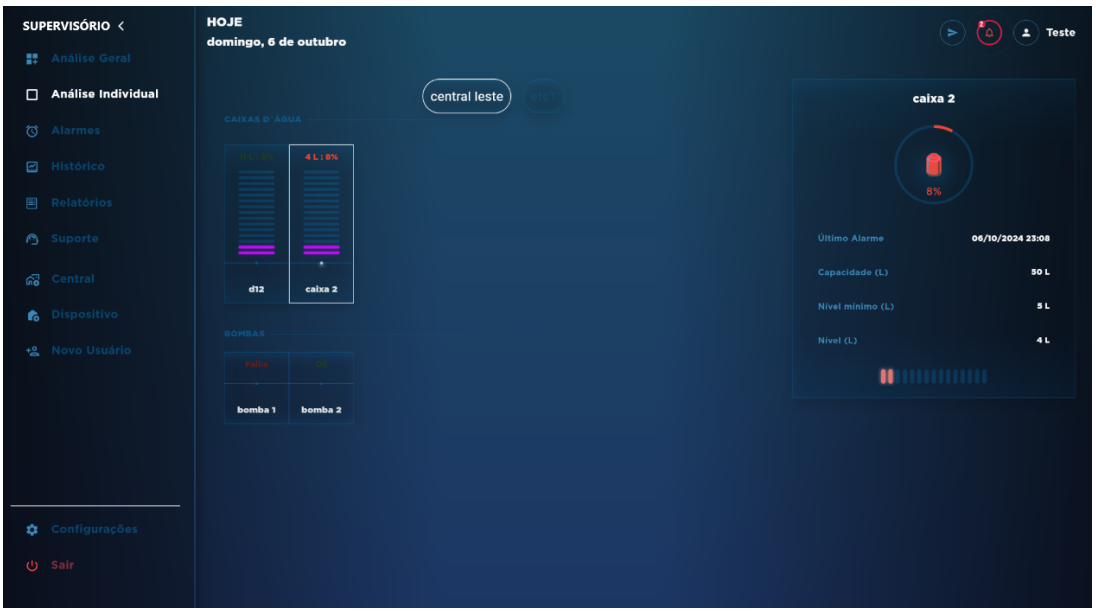
A organização da interface inicia-se com a exibição dos nomes das centrais cadastradas, dispostas horizontalmente, facilitando a navegação e a seleção. Cada central selecionada é destacada visualmente, com mudanças nas cores e nas bordas, oferecendo um indicativo claro de qual central está em análise no momento.

Abaixo dessa lista, são apresentados os dispositivos vinculados à central selecionada. Esses dispositivos, que podem incluir tanques e bombas, são exibidos horizontalmente, com indicadores visuais específicos para cada tipo. Para os tanques, barras de progresso indicam o nível atual de água em relação à capacidade total, enquanto para as bombas, são exibidos indicadores de estado, como "OK" ou "Falha". A interação do usuário com esses elementos permite a seleção de dispositivos individuais, possibilitando uma análise mais aprofundada.

No painel de detalhes, localizado à direita da tela, o usuário encontra informações detalhadas sobre o dispositivo escolhido. Para tanques, é exibido um gráfico circular que mostra o nível atual em relação à sua capacidade total, proporcionando uma visão clara da situação de armazenamento. No caso das bombas, são apresentadas informações sobre o estado operacional, como se estão funcionando adequadamente ou se há falhas. Além disso, dados complementares, como a data do último alarme gerado e a capacidade total dos dispositivos, são apresentados de maneira a facilitar o monitoramento e a tomada de decisões por parte dos usuários.

A Figura 33 apresenta o resultado visual da página de Análise Individual conforme detalhado nessa seção.

Figura 33 – Página de Análise Individual.



Fonte: Elaborado pelo Autor (2024).

7.4.5 Histórico de Alarmes

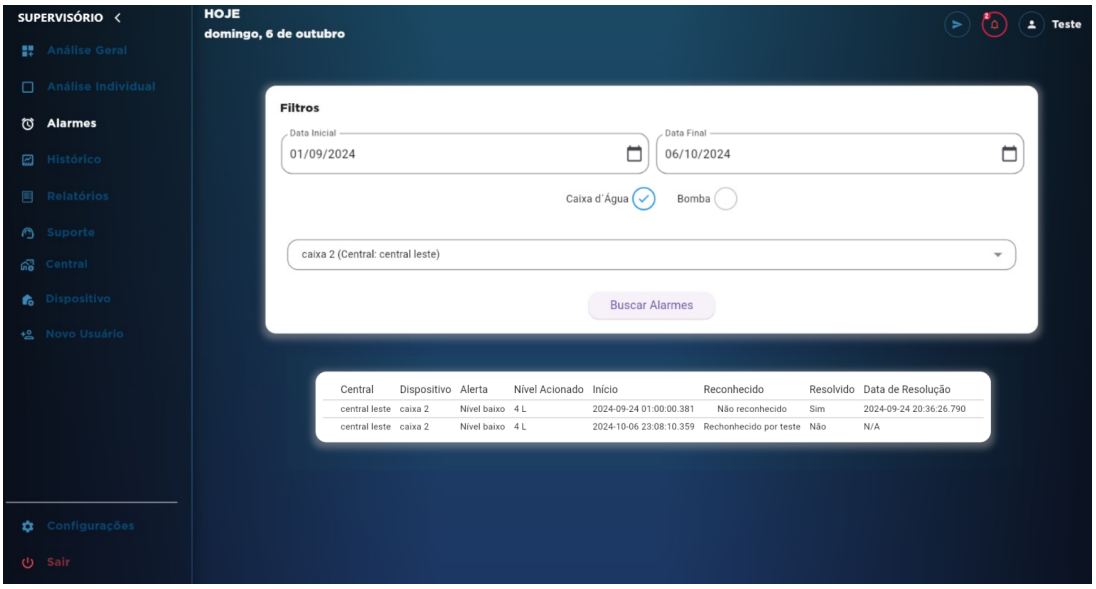
O resultado da página de Histórico de Alarmes permite a exibição do histórico dos alarmes gerados pelos dispositivos cadastrados no sistema. Sua função é permitir que o usuário visualize e analise os eventos críticos ocorridos em dispositivos, oferecendo ferramentas de busca e filtragem para facilitar o acompanhamento dos registros históricos. Por meio dessa página, o usuário pode configurar filtros que auxiliam na localização dos alarmes relevantes, organizando os dados de maneira clara e acessível.

A interface da página inicia-se com um formulário de filtros de data, permitindo que o usuário defina um intervalo de tempo para a busca dos alarmes. Além das datas, é possível selecionar o tipo de dispositivo a ser monitorado (tanques ou bombas), ajustando a exibição dos dados conforme o tipo de equipamento escolhido. Após a definição dos parâmetros, o usuário pode acionar a busca dos alarmes registrados no sistema, cujos resultados são apresentados em uma tabela contendo as informações pertinentes de cada alarme.

Os dados apresentados na tabela de alarmes incluem o nome da central onde o evento ocorreu, o dispositivo envolvido, o tipo de alerta gerado, o nível acionado (para tanques) ou o estado (para bombas), além da data e hora de início do alarme. Outros detalhes relevantes, como o reconhecimento do alarme por algum operador e a sua resolução, também são exibidos. Caso o alarme tenha sido resolvido, a data da resolução é registrada, fornecendo uma visão clara da sequência de eventos.

A página mostrou oferecer funcionalidades interativas, como a possibilidade de expandir e contrair a área de filtros, permitindo que o usuário oculte essa seção após configurar os parâmetros, otimizando o uso do espaço disponível para a exibição da tabela

Figura 34 – Página de Histórico de Alarmes.



Fonte: Elaborado pelo Autor (2024).

de alarmes. A tabela é ajustável, com rolagem horizontal e vertical, facilitando a navegação por grandes quantidades de dados.

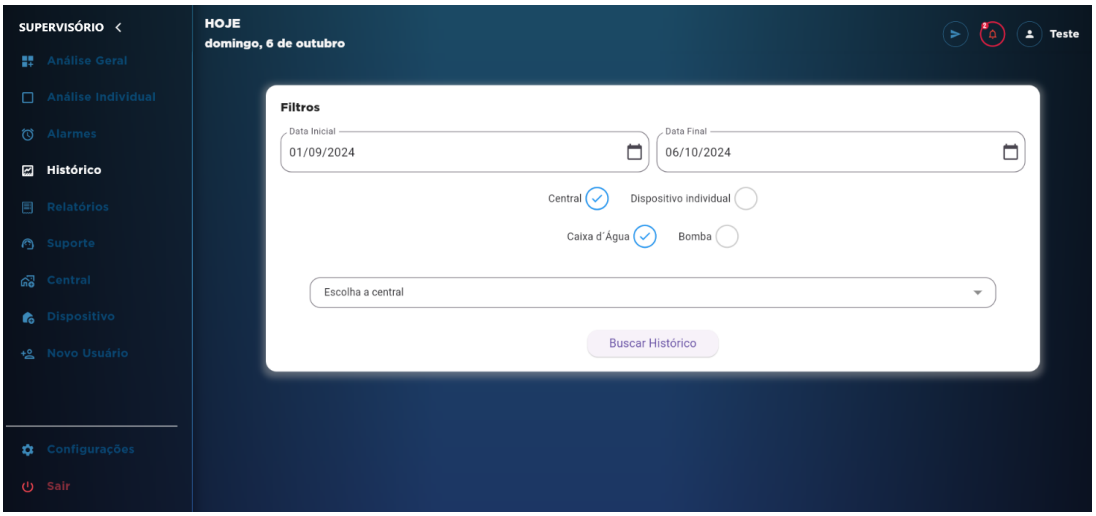
Outro aspecto relevante da página é a opção de filtrar alarmes por dispositivo específico, possibilitando uma análise mais direcionada dos eventos associados a um determinado equipamento. Isso facilita o acompanhamento do histórico de um único dispositivo em uma central.

A Figura 34 apresenta a página de Histórico de Alarmes conforme descrita nessa seção.

7.4.6 Histórico de Dados

A página de Histórico de Dados possui uma interface projetada para fornecer ao usuário uma visão detalhada dos dados coletados pelos dispositivos do sistema. Essa página permite a visualização dos registros históricos, facilitando o acompanhamento do desempenho dos dispositivos em um determinado período. Conforme apresentado na Figura 35, o usuário pode definir filtros de data, selecionar dispositivos individuais ou centrais inteiras, e escolher entre tanques ou bombas, ajustando assim a exibição dos dados conforme as suas necessidades de análise.

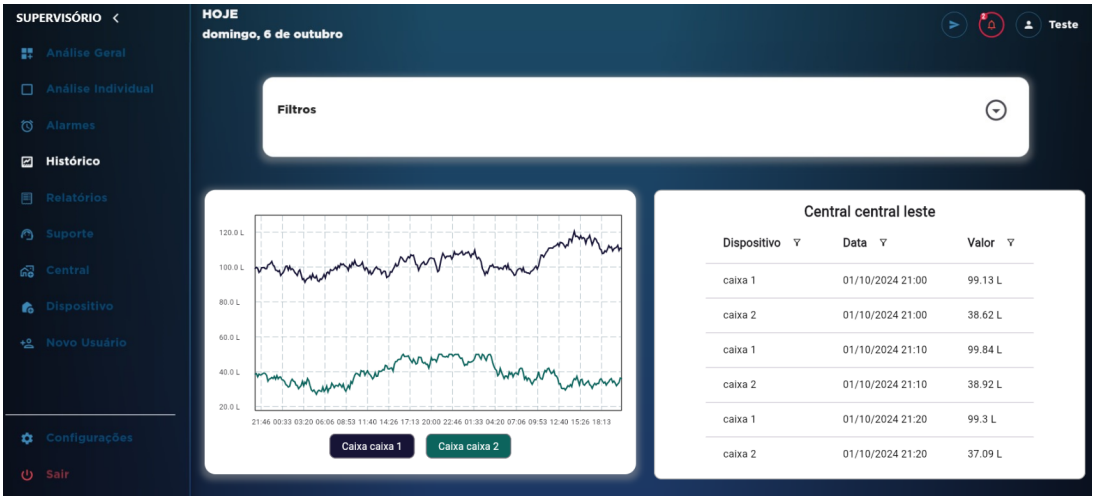
Figura 35 – Filtro da página de Histórico de Dados.



Fonte: Elaborado pelo Autor (2024).

Os gráficos exibem as variações nos níveis de água dos tanques ou o estado operacional das bombas. Como ilustrado na Figura 36, ao escolher o histórico por central no filtro, o usuário poderá visualizar as curvas de dados de vários tanques ou motores no mesmo gráfico, facilitando a análise comparativa de todos os equipamentos da central. Além disso, a tabela associada ao gráfico organiza os dados detalhados, apresentando o dispositivo, a data e o valor registrado, como o nível de água ou o estado do equipamento, permitindo uma análise mais detalhada.

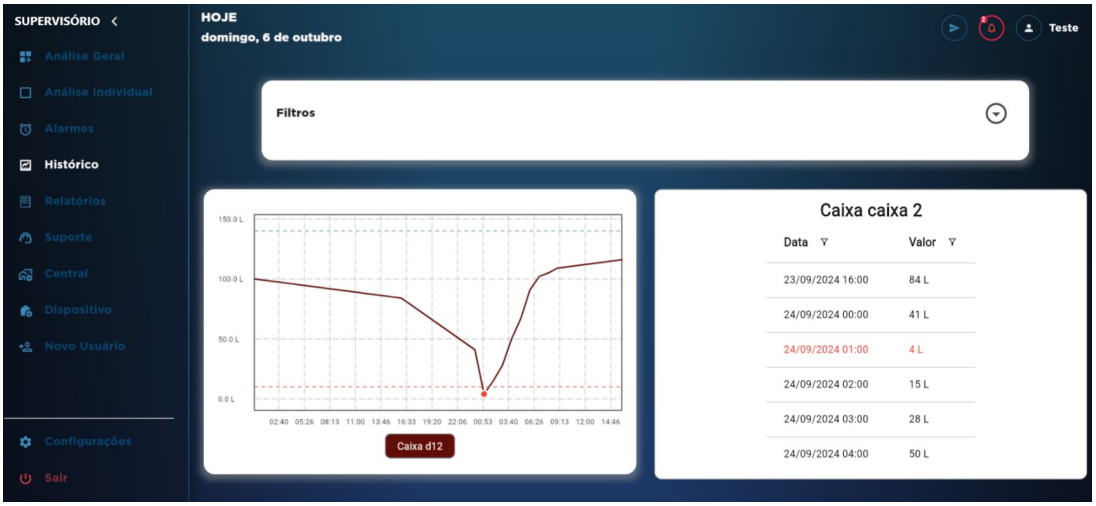
Figura 36 – Histórico de dados de tanques em uma central.



Fonte: Elaborado pelo Autor (2024).

A Figura 37 apresenta o caso de gráficos para análise individual de tanques, onde são criados gráficos de controle com limites superiores e inferiores, correspondentes à capacidade máxima e ao nível mínimo do tanque. Quando os níveis de água ultrapassam esses limites, especialmente quando o nível fica abaixo do mínimo, um ponto vermelho é adicionado ao gráfico, indicando uma situação de alerta.

Figura 37 – Histórico de dados para tanque individual.



Fonte: Elaborado pelo Autor (2024).

A página também oferece uma tabela interativa ao lado do gráfico, onde os dados históricos podem ser visualizados de maneira tabular. Os dados apresentados incluem informações como o nome do dispositivo (no caso de histórico por central), a data e o valor registrado. A tabela permite a aplicação de filtros específicos, como o dispositivo ou o valor medido, diretamente pelo cabeçalho de cada coluna, possibilitando uma análise detalhada e segmentada conforme as necessidades do usuário. Isso permite uma navegação

eficiente pelos dados e facilita a identificação de padrões ou problemas operacionais.

A lógica de exibição das amostras é ajustada automaticamente conforme o intervalo de tempo selecionado pelo usuário. Para tonar mais eficiente a visualização e garantir que os dados sejam apresentados claramente, o sistema implementa uma lógica de amostragem baseada na duração do período selecionado. Para períodos de até 1 dia, as amostras são adquiridas com intervalo de 5 minutos; para até 7 dias, as amostras são exibidas a cada 10 minutos; períodos de até 30 dias têm amostras a cada 30 minutos; para até 365 dias, a frequência é de 60 minutos; e, para intervalos superiores a 1 ano, as amostras são espaçadas a cada 120 minutos. Esse ajuste automático na frequência das amostras melhora a apresentação dos dados, tornando a visualização mais prática e eficiente.

A Figura 38 apresenta a visualização do histórico de dados de uma bomba individual. Percebe-se que ao detectar o estado de falha, com a curva chegando no valor zero, há um alerta no gráfico com um ponto vermelho, indicando a anormalidade naquele instante.

Figura 38 – Histórico de dados para bomba individual.

Fonte: Elaborado pelo Autor (2024).

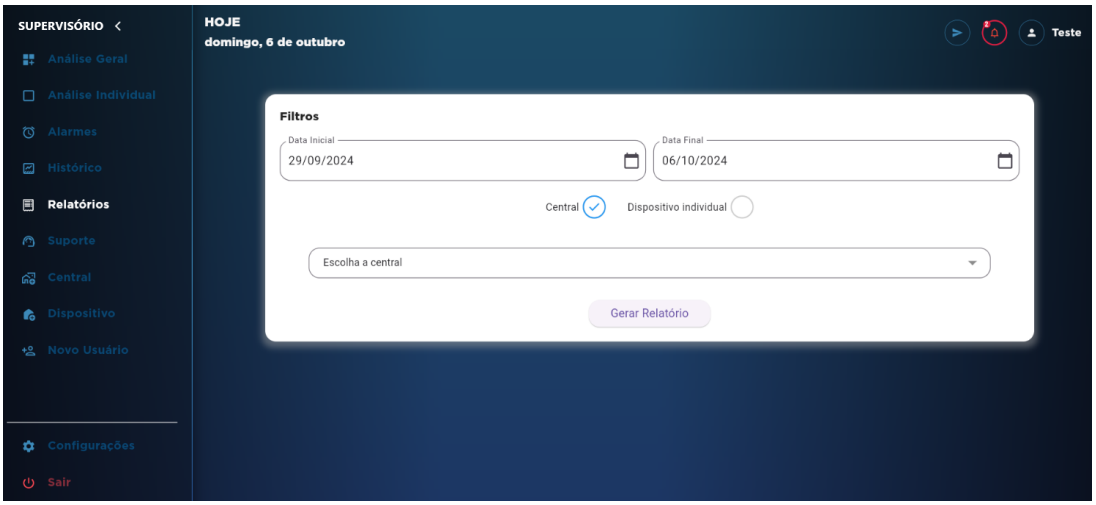
7.4.7 Relatórios

A página de Relatórios garante ao usuário poder gerar e visualizar relatórios detalhados sobre os dispositivos e centrais cadastrados no sistema. Essa funcionalidade possibilita a criação de relatórios em formato PDF tanto para uma central inteira quanto para dispositivos individuais, auxiliando na análise de eventos e dados históricos relevantes de maneira detalhada.

Conforme apresentado na Figura 39, a interface da página oferece ao usuário a possibilidade de configurar filtros de data para definir o período desejado para o relatório. Além disso, é possível optar por gerar um relatório para uma central completa ou para dispositivos individuais. No caso de dispositivos individuais, o usuário escolhe entre gerar o relatório para um tanque ou uma bomba, ajustando o conteúdo do relatório conforme o

tipo de dispositivo selecionado.

Figura 39 – Página de Relatórios.



Fonte: Elaborado pelo Autor (2024).

Para o relatório de uma central, o sistema apresenta informações detalhadas sobre os eventos registrados para todos os dispositivos daquela central, exibindo dados como os níveis acionados, se os alarmes foram reconhecidos e, em caso positivo, por quem, além de detalhes sobre a resolução dos alarmes. Também são exibidos os estados atuais dos dispositivos, com informações sobre o nível de água dos tanques ou o estado de funcionamento das bombas. Um exemplo deste tipo de relatório pode ser visto na Figura 40a, que apresenta um relatório gerado para uma central.

Por outro lado, ao gerar um relatório para um dispositivo individual, como um tanque ou uma bomba, o sistema oferece uma visão ainda mais detalhada. Além dos alarmes registrados para aquele dispositivo específico, o relatório inclui um histórico das últimas 24 horas, apresentando amostras horárias dos dados coletados. Para os tanques, por exemplo, são exibidos o nível mínimo, a capacidade total e o nível atual de água, como pode ser observado na Figura 40b, onde se apresenta um relatório gerado para um tanque de água, mostrando os alarmes e o registro de nível de água a cada hora nas últimas 24 horas.

Assim, a página de Relatórios oferece ao usuário uma ferramenta eficiente para gerar relatórios completos e bem estruturados, seja para monitoramento de uma central inteira ou de dispositivos específicos.

Figura 40 – Relatórios PDF gerados.

(a) Central.

Central - central leste

Data de geração do relatório: 17/10/2024 02:15:39
Intervalo de datas dos alarmes: 01/09/2024 - 07/10/2024
Tipo de central: Abastecimento de Água

Alarmes do período:

Dispositivo	Alarme	Nível Acionado	Reconhecido por	Início	Resolvido em
Bomba bomba 1	Falha na bomba	N/A	Não reconhecido	23/09/2024 02:00:00	23/09/2024 13:03:04
Caixa caixa 2	Nível baixo	4 L	Não reconhecido	24/09/2024 01:00:00	24/09/2024 20:36:26
Bomba bomba 1	Falha na bomba	N/A	Não reconhecido	24/09/2024 06:00:00	25/09/2024 23:07:58
Bomba bomba 1	Falha na bomba	N/A	FLV Techs	24/09/2024 15:00:00	25/09/2024 23:37:24
Bomba bomba 1	Falha na bomba	N/A	Não reconhecido	06/10/2024 23:06:12	06/10/2024 23:07:58
Caixa caixa 2	Nível baixo	4 L	teste	06/10/2024 23:08:10	Não resolvido

Estado atual dos dispositivos:

Dispositivo	Nível/Status
Bomba bomba 1	Falha
Bomba bomba 2	OK
Caixa caixa 1	11 L
Caixa caixa 2	4 L

(b) Dispositivo individual.

Caixa - caixa 2

Data de geração do relatório: 07/10/2024 00:09:15
Intervalo de datas do relatório: 01/09/2024 - 06/10/2024
Central: central leste
Tipo de central: Abastecimento de Água
Último alarme: 06/10/2024 23:08:10
Capacidade: 50 L
Nível mínimo: 5 L
Nível atual: 4 L

Alarmes do período:

Alarme	Reconhecido por	Início	Resolvido em	Nível Acionado
Nível baixo	Não reconhecido	23/09/2024 20:35:33	23/09/2024 20:36:26	4 L
Nível baixo	teste	06/10/2024 23:08:10	Não resolvido	4 L

Registros das últimas 24 horas:

Data e hora	Nível
06/10/2024 01:07:48	99 L
06/10/2024 02:07:48	98 L
06/10/2024 03:07:48	97 L
06/10/2024 04:07:48	96 L
06/10/2024 05:07:48	95 L
06/10/2024 06:07:48	94 L
06/10/2024 07:07:48	93 L
06/10/2024 08:07:48	92 L
06/10/2024 09:07:48	91 L
06/10/2024 10:07:48	90 L
06/10/2024 11:07:48	89 L
06/10/2024 12:07:48	88 L
06/10/2024 13:07:48	87 L
06/10/2024 14:07:48	86 L

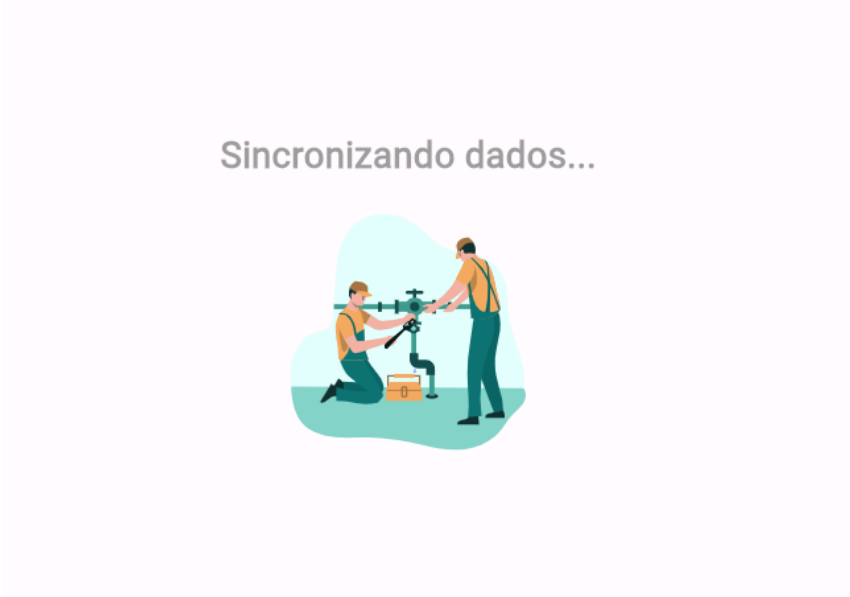
Fonte: Elaborado pelo Autor (2024).

7.4.8 Sincronização

A página de Sincronização de Dados é uma etapa com certa relevância no sistema, por ser responsável pela obtenção e carregamento de todas as informações relevantes que serão utilizadas nas demais funcionalidades da aplicação. Esse processo é essencial para garantir que os dados apresentados ao usuário estejam atualizados e prontos para análise, permitindo a visualização e monitoramento adequados das centrais, bombas e tanques cadastrados.

Durante o carregamento da página, o sistema inicia a sincronização dos dados, conforme ilustrado pela animação de trabalhadores ajustando um encanamento na Figura 41, que simboliza a preparação dos dados para o uso.

Figura 41 – Página de Sincronização.



Fonte: Elaborado pelo Autor (2024).

A sequência de operações executadas segue uma ordem cuidadosamente definida: primeiro, o sistema sincroniza os dados das centrais; em seguida, sincroniza os tanques e as bombas associados a cada central. Posteriormente, as localizações dos dispositivos são associadas às centrais, e, por fim, o controlador de mapas realiza a sincronização dos dados geográficos dos dispositivos, completando o ciclo de inicialização. Essa sequência é importante para garantir que as informações estejam organizadas e prontas para uso no sistema.

Como resultado, esta página garante que todos os dados estejam corretos antes que o usuário acesse as funcionalidades principais, como as análises de desempenho ou a visualização dos alarmes. Após a conclusão bem-sucedida da sincronização, o usuário é automaticamente redirecionado para a página principal do sistema. Caso a sincronização falhe, o sistema impede o acesso às outras páginas, até que os dados sejam corretamente

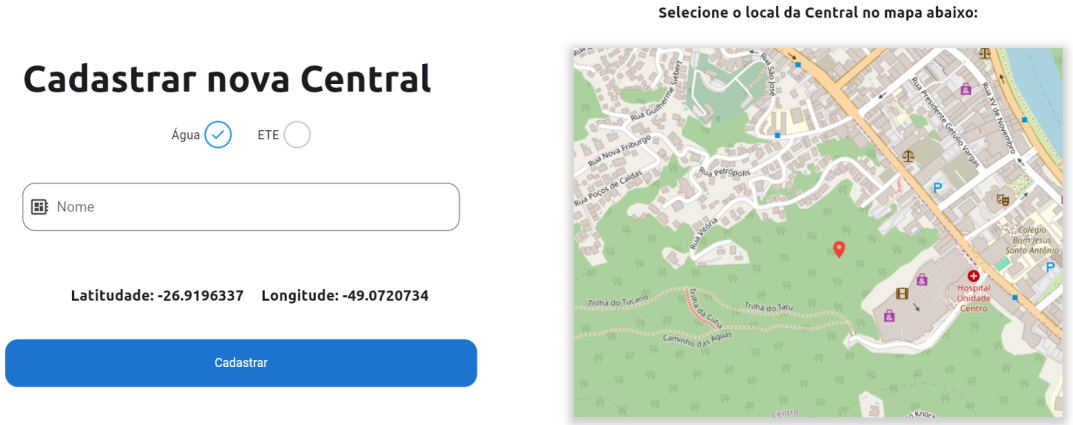
carregados.

O fluxo de trabalho garante que todas as informações essenciais estejam preparadas para o usuário interagir com o sistema, evitando possíveis inconsistências ou dados desatualizados. Isso assegura um ambiente de monitoramento confiável, essencial para a tomada de decisões precisas sobre os dispositivos monitorados.

7.4.9 Acesso Restrito

Nas páginas restritas aos administradores, é possível gerenciar centrais, dispositivos e usuários, realizando operações como adição, edição e exclusão. A página de adição de central permite cadastrar novas centrais no sistema, sendo necessário escolher entre os tipos “água” ou “ETE”, além de localizar a central no mapa e inserir suas coordenadas geográficas. O formulário de cadastro também exige que o nome da central seja único para evitar duplicidade. Na Figura 42, é possível ver a interface onde o administrador insere o nome, o tipo e define a localização da central diretamente no mapa.

Figura 42 – Página de adição de central.

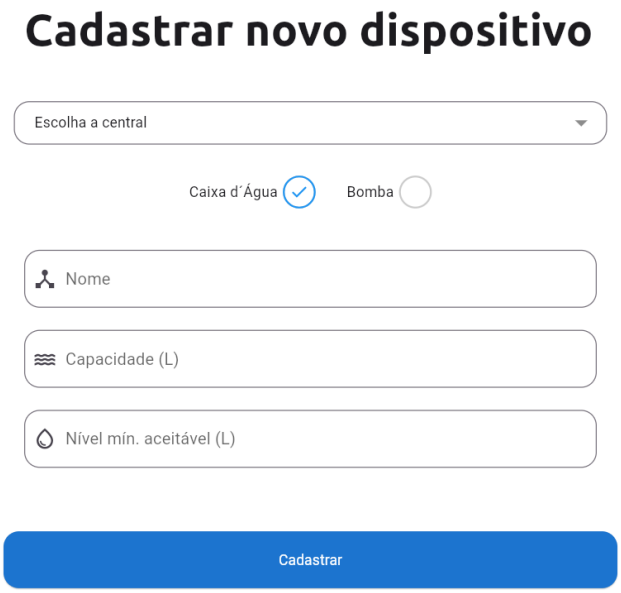


Fonte: Elaborado pelo Autor (2024).

A adição de dispositivos também pode ser realizada por meio de uma página específica, que permite associar o novo dispositivo a uma central existente. Nessa página, o administrador pode escolher entre cadastrar uma caixa d’água ou uma bomba. Para as caixas d’água, o formulário requer informações sobre a capacidade em litros e o nível mínimo aceitável. A Figura 43 mostra a interface que possibilita o cadastro de novos dispositivos, onde são inseridos dados como nome, capacidade e nível mínimo, no caso de caixas d’água.

A página de adição de usuário permite o registro de novos usuários no sistema, exigindo informações como nome, e-mail, senha e categoria do usuário. Essa página também possui validações para garantir que os dados sejam corretamente inseridos, como a verificação de um e-mail válido e a criação de uma senha com os requisitos mínimos de

Figura 43 – Página de adição de dispositivo.



Fonte: Elaborado pelo Autor (2024).

segurança. A Figura 44 ilustra essa interface, onde o administrador preenche os campos necessários para adicionar um novo usuário à plataforma.

Figura 44 – Página de adição de usuário.

Fonte: Elaborado pelo Autor (2024).

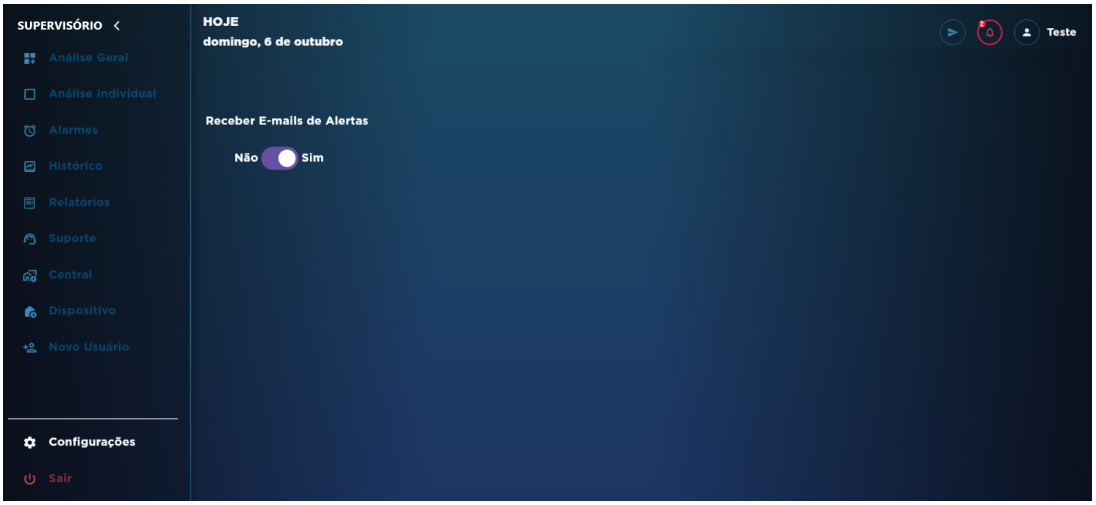
7.4.10 Outras Páginas

Além das páginas principais do sistema, existem também outras páginas importantes para a experiência de usuário, como as páginas de configurações, *splash* e suporte.

A página de configurações, mostrada na Figura 45, atualmente permite ao usuário ativar ou desativar o recebimento de e-mails de alertas. Como trabalho futuro, pretende-se expandir esta página com novas funcionalidades, como a escolha entre temas claro e escuro e a possibilidade de configurar o *broker* MQTT diretamente no software, entre

outras opções que vão facilitar a personalização do sistema conforme as preferências do usuário.

Figura 45 – Página de configurações.



Fonte: Elaborado pelo Autor (2024).

A página *splash* é exibida brevemente durante a inicialização do sistema e tem como função preparar o ambiente de trabalho enquanto sincroniza os dados necessários para o funcionamento correto da aplicação. Esta página exibe uma animação informativa enquanto o sistema carrega, garantindo que o usuário saiba que a plataforma está em processo de inicialização.

Por fim, a página de suporte foi desenvolvida para fornecer orientações e canais de comunicação caso o usuário precise de ajuda técnica. Ela facilita o contato com o suporte técnico e orienta sobre as possíveis formas de resolução de problemas relacionados ao software e dispositivos do sistema, além de servir como um ponto de referência para a obtenção de assistência.

7.5 CONSIDERAÇÕES SOBRE ESCALABILIDADE E DESEMPENHO

A escalabilidade do sistema foi planejada para garantir que o aumento no número de usuários e no volume de dados não comprometa o desempenho da aplicação. A arquitetura utilizada, baseada na Arquitetura Limpa, facilita a manutenção e a adição de novas funcionalidades, sem a necessidade de grandes mudanças no código existente.

A escolha do Firebase Cloud Firestore como banco de dados foi feita por sua capacidade de escalabilidade horizontal. O Firestore, sendo um banco de dados NoSQL, permite o crescimento distribuído dos dados, garantindo que a aplicação continue a responder de maneira eficiente mesmo com um grande volume de transações. Além disso, o modelo de precificação do Firestore, baseado no consumo (número de leituras, gravações e armazenamento), possibilita a adequação dos custos de operação à escala real da aplicação, promovendo maior controle e previsibilidade financeira.

O Firebase Hosting, responsável pela hospedagem da aplicação, utiliza uma rede de distribuição de conteúdo CDN que distribui os arquivos estáticos por servidores em diferentes regiões geográficas. Isso contribui para a redução de latência e melhora o desempenho da aplicação para usuários em várias localizações. Da mesma forma, os custos de hospedagem variam conforme o uso e a demanda global, permitindo que o sistema escale horizontalmente sem necessidade de investimentos iniciais em infraestrutura.

Para gerenciar o aumento no número de requisições, o sistema utiliza o Firebase Functions. Esse recurso permite que o *backend* responda de maneira eficiente às demandas da aplicação, mesmo durante picos de utilização, sem a necessidade de intervenção manual na infraestrutura. Assim como outros serviços do Firebase, o custo das funções é diretamente relacionado à quantidade de invocações e ao tempo de execução, garantindo flexibilidade na operação e alinhamento com as necessidades da aplicação.

O sistema está estruturado para suportar o crescimento contínuo, tanto em número de usuários quanto em volume de dados, de maneira eficiente e controlada. A utilização de serviços escaláveis como o Firestore e o Firebase Hosting, aliados a modelos de precificação baseados no consumo, assegura que a aplicação possa ser expandida de forma sustentável, sem comprometer o desempenho ou gerar custos desproporcionais.

7.6 ANÁLISE DE CUSTOS

Para a realização da análise de custos do sistema supervisório no Firebase e Heroku, considera-se a configuração de monitoramento de 30 dispositivos, com amostras de dados atualizadas a cada um minuto e armazenamento de histórico a cada cinco minutos. Essa configuração específica ajuda a estimar os custos do Firestore, Cloud Functions e Heroku.

No caso do Firestore, cada atualização de amostras para todos os dispositivos ocorrem em intervalos de um minuto, resultando em uma escrita por minuto para um documento que representa os 30 dispositivos. Com um mês típico de 30 dias, isso gera um total de 43.200 escritas mensais. Atualmente, o Firebase oferece gratuitamente um limite diário de 20.000 escritas, ou seja, 600.000 escritas por mês, que cobre integralmente as necessidades de nosso sistema, sem custo adicional.

Além disso, o armazenamento de histórico no Firestore, com uma amostragem a cada cinco minutos, gera 8.640 registros mensais. Cada registro de histórico para os 30 dispositivos ocupa aproximadamente 1.428 bytes, resultando em um consumo mensal de cerca de 11,78 MB, ou aproximadamente 141,36 MB anuais. O Firestore possui um limite gratuito de 1 GB de armazenamento, e o uso anual do histórico permanece dentro desse limite, sem custos adicionais. Para garantir que o sistema não ultrapasse esse limite, foi implementada uma política de retenção de dados que mantém o histórico de no máximo dois anos, com deleção automática dos dados mais antigos que esse período. Dessa forma, o sistema evita o acúmulo excessivo de dados e permanece no armazenamento gratuito do Firestore, mesmo em uma operação contínua e prolongada, assegurando um monitoramento

eficiente sem custos adicionais de armazenamento.

As leituras de documentos no Firestore, que ocorrem durante a visualização de dados, também ficam dentro dos limites gratuitos. O Firebase permite até 50.000 leituras por dia, totalizando 1,5 milhões de leituras mensais, um volume que cobre as necessidades do sistema. Esse limite é suficiente para as consultas realizadas pelos usuários durante o monitoramento e visualização de dados. No caso das deleções de documentos, que acontecem principalmente para o gerenciamento de histórico e a resolução de alarmes, o Firestore permite até 20.000 deleções diárias, ou 600.000 por mês, o que também atende ao uso projetado, sem gerar custos adicionais.

Para o uso das Cloud Functions, a função *checkAlarms*, responsável por verificar as atualizações no *last_mqtt_update*, é acionada a cada minuto, resultando em 43.200 invocações mensais. O Firebase oferece gratuitamente dois milhões de invocações mensais, cobrindo tranquilamente o número de invocações dessa função, portanto, sem custos adicionais. Em termos de tempo de CPU, estimando uma execução conservadora de um segundo para cada invocação, o uso mensal de CPU seria de 43.200 segundos, o equivalente a cerca de 12 horas. Como o Firebase fornece 200.000 segundos de CPU por mês gratuitamente (aproximadamente 55 horas), esse consumo de CPU também permanece dentro do limite gratuito.

Quanto ao Firebase Hosting, que suporta a interface do sistema, o uso é limitado a uma aplicação por condomínio, reduzindo o número total de acessos simultâneos e mantendo o uso dentro dos limites gratuitos. O Firebase oferece gratuitamente 10 GB de armazenamento e 360 MB de transferência diária, valores suficientes para a aplicação, especialmente com um número limitado de usuários por condomínio. No caso do Realtime Database, utilizado como cache para reduzir leituras do Firestore, o uso esperado também se mantém dentro dos limites gratuitos: o Firebase permite até um GB de armazenamento e 10 GB de download mensalmente, cobrindo a função de cache sem custos adicionais.

O Heroku é utilizado para executar o worker em Node.js, responsável pelo monitoramento e processamento das mensagens MQTT. O custo mensal do Heroku é fixo em U$7,00, e permite a consolidação de vários condomínios no mesmo worker, sem elevação do valor. Esse modelo favorece a escalabilidade, pois à medida que mais condomínios aderem ao sistema, o custo do Heroku é dividido entre eles, reduzindo o valor por condomínio.

Em resumo, o sistema de monitoramento de 30 dispositivos com a configuração proposta permanece totalmente dentro dos limites gratuitos do Firebase, sem custos adicionais além do valor fixo de U$7,00 mensais do Heroku. Anualmente, isso resulta em um custo de U$84,00, considerando apenas o Heroku.

No entanto, se o sistema for expandido, alguns limites gratuitos podem ser ultrapassados. Por exemplo, considerando o número de escritas, se a quantidade de dispositivos monitorados aumentar significativamente, chegando a um ponto onde cada escrita a cada minuto representa dados para mais de 416 dispositivos, o limite gratuito de 600.000 escri-

tas mensais do Firestore começaria a ser excedido. No caso de armazenamento, o limite gratuito de 1 GB suportaria o histórico de até aproximadamente 83 dispositivos por 1 ano, considerando o mesmo volume de dados de 1.428 bytes por amostra a cada 5 minutos. Para as invocações das Cloud Functions, o limite de 2 milhões de invocações mensais suportaria até 1.388 dispositivos (considerando a função *checkAlarms* acionada a cada minuto). Assim, o sistema é escalável, mas o aumento considerável do número de dispositivos pode começar a gerar custos adicionais conforme esses limites são ultrapassados.

A análise de custos de desenvolvimento do sistema supervisório considera a atuação de um único desenvolvedor, com um custo estimado de R$35,00 por hora. O projeto, totalizando 250 horas, abrangeu todas as fases essenciais: planejamento, desenvolvimento e testes da aplicação, resultando em um custo total de R$8.750,00.

O desenvolvimento foi estruturado em três etapas principais. A fase de planejamento e definição de requisitos demandou aproximadamente 50 horas, com um custo de R$1.750,00, concentrando-se no estabelecimento das funcionalidades e da arquitetura do sistema. A fase de desenvolvimento e implementação, que envolveu a construção da interface e integração com o Firebase, consumiu cerca de 150 horas, representando um custo de R$5.250,00. Por fim, a etapa de testes e validação, com duração de 50 horas e um custo de R$1.750,00, garantiu a funcionalidade mínima da aplicação para o contexto proposto, ainda sendo necessárias validações complementares em casos reais.

Dessa forma, o custo total de R$8.750,00 cobre o processo completo de desenvolvimento do sistema, desde o planejamento até a validação final, com uma alocação adequada de recursos para atender às necessidades do projeto.

7.7 PLANO DE NEGÓCIOS

A análise de custos apresentada na Seção 7.6 permitiu identificar os recursos necessários para a manutenção e expansão do sistema supervisório, bem como estimar os limites e custos associados à infraestrutura no Firebase e Heroku. Com base nesses dados, elaborou-se um plano de negócios para comercializar o sistema de monitoramento em condomínios, oferecendo uma estrutura flexível de pacotes que permite aos clientes ajustar as funcionalidades e o escopo de acordo com suas necessidades.

Para maximizar a acessibilidade e escalabilidade do sistema, o modelo de negócios foi estruturado com base em uma configuração modular. Essa abordagem permite que os condomínios optem por funcionalidades adicionais mediante um custo específico, ajustando-se ao número de dispositivos e aos requisitos de monitoramento de cada cliente. As opções de expansão incluem desde o aumento do número de dispositivos monitorados até o ajuste da frequência de amostragem e opções de notificações adicionais.

A estrutura do plano de negócios para venda do sistema supervisório está organizada em pacotes opcionais e serviços extras, descritos nas tabelas 3.

Tabela 3 – Estrutura de Custos do Sistema Supervisório

Categoria	**Custo Básico**	**Expansão Opcional**	**Custo Total (Plano Máximo)**
Pacote de Monitoramento	R$ 5.687,50	-	R$ 5.687,50
Dispositivos Adicionais	Inclui 30	R$ 50,00 para cada 10 extras	R$ 50,00 x N (número de grupos de 10)
Histórico de Dados	6 meses	1 ano: R$ 150,00	2 anos: R$ 250,00
Resolução de Dados no Histórico	30 minutos	10 minutos: R$ 120,00	5 minutos: R$ 200,00
Notificações	E-mail	Telegram: R$ 200,00	R$ 200,00
Geração de Relatórios	Não Incluso	Relatórios em PDF: R$ 100,00	R$ 100,00

Fonte: Elaborado pelo Autor (2024).

O valor de R$5.687,50 foi definido com base nos custos totais de desenvolvimento do sistema supervisório, que inicialmente foram calculados em R$8.750,00. Para facilitar a viabilidade comercial e garantir um retorno adequado ao longo de múltiplas vendas, optou-se por dividir esse valor de desenvolvimento por dois, resultando em um custo-base de R$4.375,00.

A este valor foi então adicionada uma margem de lucro de 30%, estabelecendo o preço final de R$5.687,50. Esse valor permite recuperar o investimento de desenvolvimento e obter rentabilidade a partir da segunda venda do sistema. A estratégia de precificação também considera a competitividade do mercado, oferecendo um valor atrativo para os condomínios e, ao mesmo tempo, proporcionando ao desenvolvedor um retorno que cobre os custos iniciais e viabiliza futuras expansões e melhorias no sistema.

O cálculo do custo mensal de manutenção do sistema supervisório foi estruturado para cobrir os principais aspectos operacionais e assegurar a estabilidade do serviço prestado, sendo apresentado na Tabela 4. Um dos elementos essenciais é o custo do Heroku, plataforma utilizada para o processamento das mensagens MQTT e a execução de funções no *backend*. Atualmente esse custo é de aproximadamente U$7,00 mensais, devendo ser monitorados alterações futuras para reajustes do cálculo de custos. A fim de se proteger contra oscilações futuras do câmbio, foi considerada uma conversão com o dólar a R$7,00, resultando em R$49,00 mensais. Essa margem visa garantir estabilidade financeira, mesmo diante de eventuais aumentos na taxa de câmbio.

Além disso, embora o Firebase tenha sido dimensionado para operar dentro dos limites gratuitos, foi incluída uma reserva de R$10,00 mensais para cobrir possíveis excedentes, garantindo que o sistema possa lidar com aumentos de uso inesperados sem comprometer a operação.

Outro componente do custo de manutenção é o suporte técnico mensal, estimado em uma média de uma hora de trabalho ao custo de R$35,00. Esse valor cobre ajustes menores, resolução de problemas pontuais e monitoramento do sistema para assegurar seu bom funcionamento.

Por fim, é aplicada uma margem de lucro de 20% sobre o total dos custos mensais. Essa margem permite que o projeto mantenha sua viabilidade econômica e reforce o suporte contínuo. Dessa forma, o custo mensal de manutenção cobre não apenas os gastos operacionais, mas também oferece uma proteção financeira, visando estabilidade e

eficiência a longo prazo para o sistema supervisório.

Tabela 4 – Custo Mensal de Manutenção

Descrição	**Custo Mensal**
Worker no Heroku	R$ 49,00
Uso do Firebase	R$ 10,00
Manutenção e Suporte	R$ 35,00 por hora
Margem de Lucro (20%)	Incluída

Fonte: Elaborado pelo Autor (2024).

A estrutura apresentada nas tabelas possibilita a personalização do sistema conforme as necessidades específicas dos condomínios, assegurando a eficiência do monitoramento sem incorrer em custos desnecessários. A utilização do worker no Heroku para processamento de mensagens *MQTT* e a opção de escalabilidade no número de dispositivos monitorados demonstram a flexibilidade do sistema. Dessa forma, condomínios de diferentes portes podem utilizar a solução com uma estrutura de custos proporcional à sua demanda de monitoramento e ao nível de funcionalidade requerido.

7.8 MELHORIAS FUTURAS

Entre as melhorias futuras planejadas para o sistema, destaca-se a inclusão de notificações via SMS como alternativa ao Telegram. Considerando a possível instabilidade do Telegram no Brasil, o SMS asseguraria a entrega das notificações, mesmo em casos de bloqueios ou restrições em aplicativos de mensagens. Embora traga custos adicionais, o SMS garante maior confiabilidade quando a conectividade com redes sociais está comprometida.

Outra melhoria significativa envolve a substituição do *Dyno Worker* no Heroku, que atualmente representa um custo mensal de 7 dólares americanos. Como alternativa, sugere-se implementar um fluxo em Node-RED capaz de se comunicar diretamente com o Firestore, inserindo e tratando dados sem a necessidade do worker. Esse ajuste traria uma redução de custos e simplificaria o fluxo de processamento, eliminando uma camada intermediária, o que pode resultar em menor latência e maior eficiência no armazenamento de dados.

Por fim, a existência de alarmes com diferentes níveis de criticidade e prioridade seria uma melhoria significativa no sistema, uma vez que a maioria dos sistemas de supervisão e aquisição de dados possuem essa funcionalidade.

8 CONCLUSÃO

Este trabalho apresentou o desenvolvimento de um sistema supervisório para monitoramento de sistemas de abastecimento de água e estações de tratamento de esgoto em condomínios, visando garantir a continuidade operacional e tornar mais eficiente a supervisão desses processos críticos.

Os objetivos estabelecidos para este trabalho foram satisfatoriamente alcançados. O objetivo geral, de desenvolver um sistema supervisório Web para monitoramento de ETEs e ETAs em condomínios, foi concretizado de maneira satisfatória, ao permitir que síndicos acompanhem o estado dos dispositivos de forma prática e em tempo real, conforme planejado. Os objetivos específicos também foram atendidos integralmente: o sistema foi implementado com um processamento de dados eficiente, utilizando a integração via MQTT e o armazenamento no Firestore, com uma interface intuitiva desenvolvida em Flutter. Além disso, um sistema de alarmes com notificações automáticas via Telegram e e-mail foi incorporado, proporcionando alertas em tempo hábil aos responsáveis em casos de falhas nos dispositivos monitorados ou de variações abaixo de níveis críticos nos tanques. A realização de testes e validações com simulações de dados recebidos confirmou a funcionalidade do sistema, demonstrando sua viabilidade para aplicação no contexto proposto.

Os resultados demonstram que a arquitetura escolhida é adequada para atender às demandas de monitoramento em condomínios. O sistema alcançou um desempenho satisfatório na comunicação, armazenamento e resposta aos eventos, confirmando a viabilidade de sua aplicação em ambientes que demandam o monitoramento de ETEs e ETAs aliado a um baixo custo operacional.

REFERÊNCIAS

AGGARWAL, Vibhor; SENGUPTA, Shubhashis; SHARMA, Vibhu Saujanya; SANTHARAM, Aravindan. A Scalable Master-Worker Architecture for PaaS Clouds. *In*: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. [*S.l.*: *s.n.*], 2012. P. 1268–1275.

ALBERTO, Matheus. **Pacotes no Flutter**. [*S.l.*: *s.n.*], 2024. Acessado em: 23 de abril de 2024. Disponível em: https://www.alura.com.br/artigos/pacotes-no-flutter?srsltid=AfmBOopQXFwyA2yqdhL3_4NdPGYzhShC8ObK2pJv-yeYiJOFwRJ93wT6.

ALI, Ali Ahmed Abed; DAUWED, Mohammed. Development of a Hybrid Mobile App for Student Management System. **Journal of Advanced Sciences and Nanotechnology**, Canadian University of Dubai, v. 1, n. 1, p. 37–42, 2022.

BARBOSA, Adriley Samuel Ribeiro. **Análise Comparativa entre os Padrões MVC, MVP, MVVM e MVI na Plataforma Android**. 2022. Trabalho de Conclusão de Curso (TCC) – Instituto Federal de Educação, Ciência e Tecnologia Goiano - Campus Urutaí, Urutaí, GO, Brasil. Orientador: Dr. Júnio César de Lima.

BARBOSA, Julmar Nunes. **Estudo da aplicação de estações de tratamento de esgoto compactas em pequenos municípios de Minas Gerais**. 2009. Monografia (Especialização) – Universidade Federal de Minas Gerais, Belo Horizonte. Monografia (Especialização em Engenharia Sanitária).

BHAGAT, Shreya A.; DUDHALKAR, Sakshi G.; KELAPURE, Prathmesh D.; KOKARE, Aniket S.; BACHWANI, Sudesh A. Review on Mobile Application Development Based on Flutter Platform. **International Journal for Research in Applied Science & Engineering Technology (IJRASET)**, v. 10, n. 1, p. 803–809, 2022.

BORAH, Shekhar Suman; KHANAL, Aaditya; SUNDARAVADIVEL, Prabha. Emerging Technologies for Automation in Environmental Sensing: Review. **Applied Sciences**, v. 14, n. 8, p. 3531, abr. 2024.

BOYER, Stuart A. **SCADA: Supervisory Control and Data Acquisition**. Research Triangle Park, NC: ISA - The Instrumentation, Systems, e Automation Society, 2009. ISBN 978-1-936007-09-7.

CARRARO, Murilo Ramos. **Ferramenta para Geração de Telas de Supervisão**. 2017. Trabalho de Conclusão de Curso (Graduação) – Universidade Federal de Santa Catarina, Florianópolis.

CARVALHO JÚNIOR, Roberto de. **Sistemas Prediais Hidráulicos e Sanitários: Princípios Básicos para Elaboração de Projetos**. 5ª edição revista e ampliada. São Paulo, SP: Edgard Blücher, 2023. Livro utilizado como referência para sistemas de abastecimento de água em edificações. ISBN 978-65-5506-403-2.

CHEIS, Daiana. **Tipos de Sistemas de Bombeamento e Aplicações para ETE's e ETA's**. Acesso em: 23 ago. 2024. abr./mai. 2015. Disponível em: https://www.revistatae.com.br/Artigo/517/tipos-de-sistemas-de-bombeamento-e-aplicacoes-para-etes-e-etas.

COELHO, João Alves; GLÓRIA, André; SEBASTIÃO, Pedro. Precise Water Leak Detection Using Machine Learning and Real-Time Sensor Data. **IoT**, MDPI, v. 1, n. 2, p. 474–493, 2020.

COSTA, Isaque Vilson Batista da. **Desenvolvimento de um Software Supervisório para Monitoramento da Qualidade da Água na Escola Superior de Tecnologia da UEA através da Aquisição de Dados de Sensores**. outubro 2022. Graduação em Engenharia Elétrica – Universidade do Estado do Amazonas, Manaus, Brasil. Orientador: Prof. Dr. Fábio de Sousa Cardoso.

DINSE, Gabriel. **Sistemas IoT com ESP32 e MQTT para Monitoramento Remoto**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação) – Universidade Federal de Santa Catarina, Brasil.

DWARAKANATH, B.; KALPANA DEVI, P.; RANJITH KUMAR, A.; AHMED SAYED, M. Metwally; GHULAM ABBAS, Ashraf; BHEEMA LINGAIAH, Thamineni. Smart IoT-based Water Treatment with a Supervisory Control and Data Acquisition (SCADA) System. **Journal of Water Reuse and Desalination**, v. 13, n. 3, p. 411–429, 2023.

FECAROTTA, Oreste; MARTINO, Riccardo; MORANI, Maria Cristina. Wastewater Pump Control under Mechanical Wear. **Water**, MDPI, v. 11, n. 6, p. 1210, 2019.

FOUNDATION, OpenJS. **About Node-RED**. [*S.l.*], 2023. Disponível em https://nodered.org/about/.

GETX. **GetX - Flutter Package**. [*S.l.*], 2024. Acesso em: 20 out. 2024. Disponível em: https://pub.dev/packages/get.

GOOGLE. **Dart Overview**. [*S.l.*], 2024. Acesso em: 23 ago. 2024. Disponível em: https://dart.dev/overview.

GOOGLE. **Firebase**. Acesso em: 23 ago. 2024. 2024. Disponível em: https://firebase.google.com/.

GOOGLE. **Flutter Architectural Overview**. [*S.l.*], 2024. Acesso em: 23 ago. 2024. Disponível em: https://docs.flutter.dev/resources/architectural-overview.

HEIDARI, Hadi; ARABI, Mazdak; WARZINIACK, Travis; SHARVELLE, Sybil. Effects of Urban Development Patterns on Municipal Water Shortage. **Frontiers in Water**, v. 3, jul. 2021.

JANA, Gabriel. **Sistema de Segurança Residencial Baseado em IoT com Protocolo MQTT para Pessoas Idosas**. 2022. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Eletrônica) – Universidade Federal de Santa Catarina, Florianópolis, SC.

KEARNEY, Adam. **The MVC Schema**. Acessado em: 17 set. 2024. Abr. 2019. Disponível em: https://medium.com/@adamkearney124/model-view-controller-f2bdbf1ee999.

LEONETI, Alexandre Bevilacqua; PRADO, Eliana Leão do; BORGES DE OLIVEIRA, Sonia Valle Walter. Saneamento básico no Brasil: considerações sobre investimentos e sustentabilidade para o século XXI. **Revista de Administração Pública**, v. 45, n. 2, p. 383–396, 2011.

LIMA, Marcos Vinícius. **Avaliação técnica e ambiental da eficiência de ETE em condomínio no Porto das Dunas - Fortaleza**. 2017. Monografia (Graduação) – Universidade Federal do Ceará, Fortaleza. Monografia (Graduação em Engenharia Ambiental).

MAIA, Larissa Helena Silva; SILVA, Marielle Ádrian Freitas; SOUZA, Tamires da Silva; NECA, Cinthia Silva Moura. Esgotamento Sanitário: A Importância do Tratamento do Esgoto Doméstico e Industrial em Prol da Qualidade de Vida e Sustentabilidade

Ambiental: Uma Revisão de Literatura. **Ciências Biológicas, Ciências Sociais**, v. 27, n. 123, jun. 2023.

MANNSFELD, SCB; TEE, BC-K; STOLTENBERG, RM; CHEN, CVH-H *et al.* Water Level Monitoring Sensor Based on Iontronic Piezo-Capacitance Effect. **Nature Materials**, v. 9, p. 859–864, 2010.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Upper Saddle River, NJ, USA: Prentice Hall, 2017.

MARTINS, Victor Ferreira. **Automação Residencial Usando Protocolo MQTT, Node-RED e Mosquitto Broker com ESP32 e ESP8266**. 2019. Trabalho de Conclusão de Curso (TCC) – Universidade Federal de Uberlândia, Uberlândia, Brasil. Orientador: Prof. Dr. Renato Santos Carrijo.

NODE.JS. **About Node.js**. [*S.l.*], 2023. Available at https://nodejs.org/pt/about.

OASIS. **MQTT Version 5.0 Standard**. [*S.l.*], 2019. Acesso em: 24 ago. 2024. Disponível em: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

REALI, Marco Antônio Penalva; MORUZZI, Rodrigo Braga; PICANÇO, Aurélio Pessôa; CARVALHO, Karina Querne de. **Instalações Prediais de Água Fria**. São Carlos, ago 2002.

SAM, Michal; CHI, Hoi; SILVA, Shivon. An Enhanced Android Application for E-Rental System. **International Journal of Worldwide Engineering Research (IJWER)**, v. 01, n. 03, p. 16–24, dez. 2023.

SCHMIDT, Jeferson Luiz. **Reservatórios de água em edificações multifamiliares: análise do dimensionamento do volume útil**. 2011. Monografia (Graduação) – Universidade Federal do Rio Grande do Sul, Porto Alegre. Acesso em: 23 ago. 2024.

SILVA, Alcione Batista da. **Avaliação da produção de odor na estação de tratamento de esgoto Paranoá e seus problemas associados**. 2007. Dissertação de Mestrado – Universidade de Brasília, Faculdade de Tecnologia, Departamento de Engenharia Civil e Ambiental, Brasília, DF. Publicação PTARH.DM-105/2007.

SILVA, Lucas Ferreira da; CHARÃO, Andrea S.; RUIZ, Renata S. R.; VELHO, Haroldo F. Campos. Implantação em Nuvem de um Portal Web para Execução

Remota de Algoritmos de Computação Científica. *In*: ANAIS da XVII Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul. Ijuí: SBC, 2017.

SILVA, Renê Couto e. **Aplicação de IoT para conversores estáticos de potência**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica) – Universidade Federal de Santa Catarina, Brasil.

SOFTWARE, Opus. Node.js: O que é, quais as vantagens e como funciona, 2023. Available at https://www.opus-software.com.br/insights/node-js/.

SONI, Dipa; MAKWANA, Ashwin. A Survey on MQTT: A Protocol of Internet of Things (IoT). **Conference Paper, Charotar University of Science and Technology**, 2017.

SOUZA JUNIOR, Sidnei de. **Dispositivo inteligente para organização e notificação de medicamentos em conjunto a um aplicativo de healthcare**. Jun. 2023. Graduação em Engenharia de Computação – Universidade Federal de Santa Catarina, Araranguá, SC, Brasil. Orientadora: Profa. Analúcia Schiaffino Morales, Dra. Coorientador: Prof. Jim Lau, Dr.

THE GIT PROJECT. **Git Documentation**. [*S.l.*], 2024. Accessed: 2024-12-01. Disponível em: https://git-scm.com/docs.

THOMÉ, Vitor Furtado. **Sistema para Monitoramento e Rastreamento de Embarcações Baseado em IoT**. 2023. Trabalho de Conclusão de Curso (Graduação em Engenharia Eletrônica) – Universidade Federal de Santa Catarina, Florianópolis. Acesso em: 23 ago. 2024.

APÊNDICE A – Arquivo JSON

Código A.1 – Estrutura JSON proposta.

```
{
    "timestamp": 1726712886,
    "centers": [
        {
            "id": 1,
            "pumps": [
                {
                    "id": 1,
                    "value": 0
                },
                {
                    "id": 2,
                    "value": 1
                }
            ],
            "tanks": [
                {
                    "id": 1,
                    "value": 15
                },
                {
                    "id": 2,
                    "value": 7
                }
            ]
        },
        {
            "id": 2,
            "pumps": [
                {
                    "id": 3,
                    "value": 1
                }
            ],
            "tanks": []
        }
    ]
}
```

APÊNDICE B – Estrutura JSON inserida na coleção *historian_mqtt* do Firestore.

Código B.1 – Estrutura JSON inserida na coleção *historian_mqtt* do Firestore.

```
{
  "timestamps": {
    "0": {
      "data": [{
        "0": {
          "center_id": 1,
          "tanks": [{
            "0": {
              "device_id": 1,
              "value": 40
            },
            "1": {
              "device_id": 2,
              "value": 33
            }
          }]
        },
        "1": {
          "center_id": 2,
          "tanks": [
            {
              "0": {
                "device_id": 8,
                "value": 22
              },
              "1": {
                "device_id": 9,
                "value": 17
              }
            }
          ]
        }
      }],
      "timestamp": 1727049900
    },
    "1": {
      "data": [
        {
          "0": {
            "center_id": 1,
            "tanks": [
              {
                "0": {
                  "device_id": 1,
                  "value": 41
                },
                "1": {
                  "device_id": 2,
                  "value": 31
                }
              }
            ]
          },
```

```
          "1": {
            "center_id": 2,
            "tanks": [
              {
                "0": {
                  "device_id": 8,
                  "value": 22
                },
                "1": {
                  "device_id": 9,
                  "value": 14
                }
              }
            ]
          }
        }
      ],
      "timestamp": 1727050200
    }
  }
}
```