Universidade Federal de Minas Gerais

Projeto Orientado em Computação 2 - Relatório final

Servidor para sistema de Monitoramento e Controle de Estações de Tratamento de Água

Aluno: Gabriel Martins Medeiros Fialho

Matrícula: 2020006540

Tipo de Pesquisa: Tecnológica
Orientador: Thiago Ferreira de Noronha
Coorientador: Anolan Milanés

30 de junho de 2025

Conteúdo

1. Introdução

Este trabalho integra-se ao escopo do Programa Água Doce (PAD), uma iniciativa do Governo Federal coordenada pelo Ministério do Meio Ambiente, que visa fornecer acesso sustentável à água potável em comunidades do Semiárido brasileiro. Por meio da implantação de estações de dessalinização, o programa busca garantir a qualidade da água consumida, incorporando práticas de monitoramento, sustentabilidade e gestão ambiental.

No contexto desse programa, foi identificado um desafio operacional significativo: a ausência de mecanismos automatizados para o monitoramento das estações e a consequente dependência de registros manuais. Para mitigar esse problema, está sendo desenvolvido um sistema completo de monitoramento.

A arquitetura geral do sistema envolve a coleta de dados por microcontroladores ESP32 acoplados a sensores físicos (como TDS, temperatura, pressão e fluxo), o envio das leituras para um aplicativo móvel via Bluetooth Low Energy (BLE) e, por fim, a sincronização desses dados com um servidor web. Este servidor centraliza e organiza os dados recebidos, tornando-os acessíveis através de um dashboard visual e interativo projetado especialmente para gestores do sistema.

A contribuição específica deste trabalho concentra-se na construção do servidor web e da interface administrativa. O servidor é responsável por autenticar os usuários, persistir os dados recebidos em um banco de dados, expor uma API RESTful para comunicação com os clientes e garantir a segurança e integridade dos dados manipulados. Já a aplicação web permite o acesso visual aos dados coletados, a gestão de estações e sensores e a identificação de leituras anômalas por meio de alertas visuais.

A proposta busca, portanto, estabelecer um sistema robusto e escalável para o controle das estações de dessalinização, promovendo maior autonomia para os operadores locais, eficiência na análise das informações por gestores e maior confiabilidade nas ações de manutenção preventiva.

2. Escopo

Durante a graduação em Ciência da Computação, é comum lidar com projetos que exigem o desenvolvimento de sistemas completos, integrando conhecimentos de banco de dados, redes, arquitetura de software, engenharia de requisitos, interfaces e segurança. Este projeto sintetiza muitos desses aprendizados em uma aplicação concreta, voltada para o monitoramento de estações de dessalinização. A seção de escopo tem como objetivo delimitar claramente os componentes e funcionalidades contemplados no desenvolvimento, oferecendo uma visão abrangente das capacidades técnicas do sistema construído.

O escopo do trabalho contempla o desenvolvimento de um servidor web capaz de receber e armazenar dados enviados por um aplicativo móvel, estruturando e persistindo essas informações em um banco de dados relacional. Além disso, o servidor gerencia autenticação e autorização de usuários com perfis distintos (gestores e operadores), garantindo que apenas usuários devidamente credenciados possam acessar ou modificar informações. A API RESTful desenvolvida também permite a integração segura e eficiente com o dashboard web e o próprio aplicativo.

No que diz respeito ao front-end, o escopo abrange a implementação de um dashboard voltado ao perfil de gestor, permitindo a visualização gráfica dos dados históricos de leitura dos sensores, com suporte a filtros por estação, data e tipo de sensor. O dashboard também foi projetado para destacar automaticamente alertas e leituras fora dos parâmetros esperados, facilitando a tomada de decisão rápida e eficaz. Complementarmente, o sistema possibilita o cadastro e a gestão de novas estações e sensores, atividade restrita aos usuários com permissões administrativas, o que torna a plataforma escalável e alinhada com as necessidades operacionais do Programa Água Doce.

Por fim, este escopo propõe uma solução centralizada, segura e extensível para organizar os dados de sensores coletados em campo. Isso favorece a análise colaborativa entre gestores e operadores, promovendo maior controle sobre a operação das estações e maior confiabilidade na qualidade da água distribuída às comunidades atendidas.

3. Objetivos

O desenvolvimento deste sistema envolveu uma série de decisões técnicas e arquiteturais que traduzem, na prática, os fundamentos teóricos aprendidos ao longo da graduação em Ciência da Computação. Ao construir uma solução para o monitoramento remoto de estações de dessalinização, foi possível aplicar de forma integrada conhecimentos sobre engenharia de software, bancos de dados, redes, segurança, desenvolvimento web e arquitetura de sistemas distribuídos. Esta seção descreve, portanto, os objetivos do servidor web integrado ao dashboard, que constitui um dos principais componentes da arquitetura do sistema.

O servidor web assume o papel de núcleo central da aplicação, responsável por concentrar, processar e disponibilizar os dados gerados em campo. Além de garantir segurança e persistência das informações, esse componente sustenta a camada visual acessada pelos gestores, oferecendo recursos analíticos e administrativos. O detalhamento dos objetivos apresentados a seguir orienta os requisitos técnicos e funcionais que nortearam a implementação do backend.

O objetivo geral consiste em desenvolver um servidor web robusto, seguro e escalável, capaz de receber, armazenar, organizar e disponibilizar dados coletados por sensores de estações de dessalinização, permitindo sua visualização interativa por meio de um *dashboard* acessado por gestores.

Para alcançar esse objetivo, foi necessário implementar mecanismos específicos para a recepção, validação e armazenamento de dados coletados por sensores. Foram desenvolvidos endpoints RESTful capazes de receber dados de sensores como TDS, temperatura, pressão e vazão, enviados por meio de um aplicativo móvel. Esses dados são armazenados de forma estruturada em um banco de dados relacional, vinculando cada leitura à sua estação, sensor e instante de tempo. A configuração de cada sensor, incluindo nome, modelo, unidade e limites operacionais, também é registrada e versionada para manter um histórico de alterações.

Outro objetivo importante foi garantir a segurança e o controle de acesso. Para isso, foi implementada a autenticação de usuários utilizando tokens JWT, diferenciando permissões por tipo de usuário e permitindo o cadastro, autenticação e revogação de

acesso a operadores e gestores com operações auditáveis. Esse mecanismo assegura que apenas usuários autorizados possam submeter dados coletados por sensores ou interagir com o *dashboard*.

A camada de visualização do sistema, acessada via *dashboard*, também foi considerada como parte dos objetivos. O servidor fornece rotas protegidas que permitem ao *frontend* consultar dados com base em diferentes filtros, como nome da estação, tipo de sensor e intervalo de tempo. Rotas específicas foram desenvolvidas para entregar os dados já organizados para visualização em gráficos e tabelas, com agregação por variável e ordenação temporal, o que contribui para uma experiência de análise clara e eficiente.

No escopo da gestão de estações e sensores, o sistema permite o cadastro de novas estações com identificadores únicos, nome e coordenadas geográficas. Também possibilita o registro de sensores com toda a configuração necessária para o seu monitoramento. Os sensores podem ser associados a estações específicas, e suas configurações são versionadas para assegurar consistência entre o *frontend* e o *backend*. Além disso, funcionalidades para edição e exclusão controlada foram implementadas, respeitando regras de segurança e integridade dos dados.

Pensando em aplicações futuras e em pesquisas, outro objetivo do sistema é organizar os dados de forma que possam ser exportados em formatos padrão, como `JSON` e `CSV`. Isso possibilita sua integração com outras ferramentas analíticas ou algoritmos de aprendizado de máquina. Ao mesmo tempo, são mantidos critérios rigorosos de consistência, integridade e rastreabilidade para cada leitura sensorial registrada.

Em termos de sustentabilidade e manutenção do sistema, a *codebase* foi organizada segundo uma arquitetura em camadas bem definidas, incluindo *controllers*, *repositories*, *entities*, *middlewares* e esquemas de validação. O que garante modularidade e facilita tanto a manutenção quanto a extensão futura. Foram aplicadas boas práticas de separação de responsabilidades e reutilização de código, e a documentação das rotas foi mantida clara e acessível para facilitar a integração com novos sistemas ou equipes de desenvolvimento.

Por fim, os requisitos de segurança e estabilidade foram tratados como prioridade. Todas as requisições passam por autenticação e controle de acesso, e são submetidas a tratamento de erros centralizado. Práticas seguras foram adotadas para acesso ao banco de dados, uso de variáveis de ambiente e verificação de permissões em todos os *endpoints*, respeitando os diferentes perfis de usuário existentes no sistema.

Esse conjunto de objetivos interligados forma a base técnica do servidor web, promovendo um sistema robusto, confiável e preparado para o crescimento futuro e para sua aplicação em cenários reais de monitoramento hídrico.

4. Metodologia

A metodologia adotada neste projeto reflete a aplicação prática dos conhecimentos desenvolvidos ao longo da graduação em Ciência da Computação, particularmente nos campos de engenharia de software, arquitetura de sistemas e desenvolvimento web. Com base em situações reais envolvendo o monitoramento de estações de dessalinização, buscou-se estruturar uma solução que integrasse diferentes componentes tecnológicos com foco em modularidade, escalabilidade e confiabilidade. A metodologia foi

escolhida não apenas para guiar a execução do projeto, mas também para consolidar habilidades relacionadas à análise de requisitos, modelagem de sistemas e integração de subsistemas independentes.

Para o desenvolvimento do servidor web, foi utilizada uma abordagem baseada em metodologia ágil adaptada, inspirada em práticas do Scrum, mas sem a formalização completa de suas cerimônias tradicionais, como reuniões diárias. Em vez disso, optou-se por checkpoints semanais com os orientadores, nos quais o progresso técnico era revisado, ajustes estratégicos eram realizados e novas metas eram definidas. Essa abordagem possibilitou maior flexibilidade no desenvolvimento, permitindo a adaptação contínua às necessidades do sistema e à disponibilidade de tempo dos envolvidos, alinhando-se à natureza iterativa e incremental das metodologias ágeis descritas por Valente et al. [Valente 2020].

Para a organização e refinamento dos requisitos, foi adotada a estratégia baseada em épico e histórias de usuário, prática consolidada em projetos ágeis por favorecer a priorização de funcionalidades a partir da perspectiva do usuário final [Raharjana et al. 2021]. Todas as histórias de usuário abordadas nesta seção estão relacionadas ao único épico definido para o servidor: o gerenciamento e visualização de dados coletados por sensores em estações de dessalinização.

Esse épico concentra a responsabilidade do backend em receber, validar, armazenar, organizar e disponibilizar os dados transmitidos por dispositivos de campo. Dentre as principais histórias desenvolvidas, destaca-se inicialmente o recebimento e armazenamento de dados enviados pelo aplicativo móvel utilizado por operadores. Essa funcionalidade exigiu a criação de uma API RESTful segura, estruturada com rotas protegidas, capaz de armazenar leituras de sensores como TDS, pressão, temperatura e vazão, devidamente associadas à estação de origem e à data e hora da coleta.

Outra história fundamental refere-se à consulta histórica de dados. A partir do backend, gestores podem recuperar informações sensoriais vinculadas a estações específicas, com datação precisa e indicadores visuais de anomalias com base nos limites definidos. A infraestrutura de dados foi desenhada para otimizar esse tipo de consulta e facilitar análises contínuas de operação.

Em seguida, foi implementado o suporte à filtragem dos dados históricos com base em critérios como estação, variável e intervalo de datas. Essa funcionalidade permitiu a construção de endpoints mais flexíveis e preparados para atender às necessidades de análise exploratória e comparativa, especialmente em sistemas com múltiplas estações.

A metodologia também contemplou o desenvolvimento de funcionalidades administrativas, como o gerenciamento de módulos sensores. Os gestores podem cadastrar novos módulos no sistema, modificar propriedades como localização e sensores acoplados, e remover módulos obsoletos. Cada alteração é registrada com versionamento, permitindo rastreabilidade completa da configuração de cada estação.

Outro aspecto essencial abordado foi a gestão de operadores. O sistema permite o credenciamento de operadores com base em informações fornecidas pelo gestor, como nome e CPF, gerando um token de acesso para uso no aplicativo móvel. A revogação de acesso também está prevista, com controle de permissões por perfil.

A consolidação e exibição de dados em gráficos e tabelas foi planejada e implementada para facilitar a análise dos sensores em cada estação. Os dados são organizados de forma agregada e temporal, com rotas específicas voltadas para a visualização no dashboard.

Por fim, foi implementado o controle de acesso privilegiado aos gestores do sistema. Somente administradores podem cadastrar ou descredenciar gestores, e todo acesso sensível passa por autenticação via token JWT, conforme previsto na arquitetura de segurança do sistema.

A aplicação desta metodologia permitiu alinhar o desenvolvimento técnico com os requisitos funcionais e não funcionais do sistema. Cada história de usuário foi mapeada em tarefas concretas da implementação, resultando em um backend robusto, documentado, testável e escalável.

5. Servidor Web

O servidor web desempenha um papel central na arquitetura do sistema, funcionando como o elo de integração entre os dados coletados em campo pelas estações de dessalinização e os usuários do sistema (operadores, gestores e público geral). Seu desenvolvimento foi orientado pela necessidade de garantir robustez, escalabilidade, segurança e clareza na organização dos dados. Ao longo da implementação, conceitos fundamentais da Ciência da Computação foram aplicados na prática, como arquitetura em camadas, controle de acesso, persistência de dados, abstração de entidades e boas práticas de programação. Além disso, a construção do servidor permitiu o aprofundamento em ferramentas modernas utilizadas amplamente no mercado, como `Node.js`[Node.js contributors 2024], `Express`[Holowaychuk and Contributors 2024], `TypeORM`[Contributors 2024d], `MySQL`[Corporation 2024] e `Zod`[McDonnell 2024], resultando em uma aplicação sólida e extensível.

5.1. Arquitetura do Sistema

A arquitetura do servidor segue o padrão de camadas, separando claramente os papéis de roteamento, controladores, serviços, repositórios, middlewares, validações e entidades de banco de dados. Essa separação não apenas facilita a manutenção e a escalabilidade, mas também favorece a legibilidade do código, a reutilização de componentes e a testabilidade.

O núcleo do servidor é construído com o framework `Express`, que gerencia as requisições HTTP de maneira eficiente. A inicialização do servidor ocorre no arquivo `src/index.ts`, onde são aplicados middlewares fundamentais como `express.json()` para parsing do corpo das requisições, `cors()` para permitir acesso externo, e middlewares customizados para autenticação, controle de permissões e tratamento de erros.

O acesso ao banco de dados relacional `MySQL` é abstraído por meio do `TypeORM`, que mapeia as entidades do domínio como usuários, sensores, estações, leituras e papéis. Cada entidade é descrita com decorators que representam suas colunas e relacionamentos. A lógica de persistência é encapsulada em repositórios especializados, como `UserRepository`, `StationRepository` e `SensorRepository`.

A segurança do sistema é garantida por meio de autenticação baseada em JWT (JSON Web Token), implementada no AuthMiddleware.ts. A autorização, por sua vez, é gerenciada com base nos papéis atribuídos aos usuários (administrador, gestor ou operador), utilizando o middleware RequireRoleMiddleware.ts. As senhas dos usuários são armazenadas de forma segura utilizando bcrypt.

A biblioteca Zod é utilizada para validar os dados de entrada e saída de todas as rotas da API. Os schemas de validação estão organizados por controlador, na pasta src/app/schemas/API, e contribuem para garantir a integridade dos dados transmitidos. Além disso, os próprios schemas servem de base para geração automática da documentação OpenAPI através da biblioteca @asteasolutions/zod-to-openapi, sendo o processo de geração centralizado no script scripts/generateDocs.ts, que exporta o arquivo openapi.json.

Regras de negócio mais complexas, como login e verificação de permissões, foram extraídas para serviços independentes, como AuthService.ts e AccessControlService.ts. A organização modular facilita testes unitários e promove alta coesão e baixo acoplamento entre as camadas.

A camada de middlewares inclui componentes reutilizáveis como AuthMiddleware.ts, RequireRoleMiddleware.ts, ValidationMiddleware.ts para verificação dos schemas do Zod e ErrorMiddleware.ts para captura e formatação de erros.

A configuração do TypeORM é definida em src/database/data-source.ts, com uso de variáveis de ambiente (armazenadas no arquivo .env) para permitir flexibilidade entre os ambientes de desenvolvimento e produção. O sistema também conta com scripts de populadores iniciais, como UserSeed.ts e seeds/init.ts, que criam dados simulados para testes.

A pasta docs/ centraliza a documentação gerada, enquanto scripts auxiliares como populateReadings.ts permitem a simulação de leituras de sensores para testes e validação.

5.2. API REST

A aplicação expõe uma API RESTful estruturada, com endpoints organizados por funcionalidade. As rotas seguem as convenções de boas práticas REST e utilizam autenticação baseada em JWT, retornando sempre respostas JSON padronizadas com códigos HTTP adequados.

As rotas de autenticação estão disponíveis em /auth. O endpoint POST /auth/login permite autenticar o usuário e retornar um token JWT armazenado em cookie seguro. A rota POST /auth/logout remove o token da sessão, e GET /auth/me retorna os dados do usuário autenticado.

Na rota /users, GET /users lista os usuários cadastrados, POST /users cria um novo usuário, e GET /users/:id retorna os dados de um usuário específico.

As estações são gerenciadas em /stations. A rota GET /stations lista todas as estações visíveis ao usuário. A rota POST /stations permite a criação de uma nova estação por administradores ou gestores. A rota GET /stations/:identifier

retorna os dados de uma estação específica, `PUT /stations/:identifier` permite a edição dos dados, e `DELETE /stations/:identifier` permite a exclusão de uma estação.

Os sensores são manipulados em `/sensors`, com rotas para listagem (`GET /sensors`) e criação de novos sensores (`POST /sensors`).

Sensores associados a estações são tratados em `/station-sensors`, onde `GET /station-sensors/station/:stationId` lista os sensores de uma estação, e `POST /station-sensors` permite associar um novo sensor à estação.

As leituras de sensores estão disponíveis em `/readings`. A rota `GET /readings/station/:stationId` retorna todas as leituras sensoriais de uma estação específica, enquanto `POST /readings` permite o envio de novas leituras por operadores autorizados.

Características da API

A API segue rigorosamente o estilo RESTful, com uso consistente dos métodos HTTP `GET`, `POST`, `PUT` e `DELETE`. A autenticação é baseada em JWT e os níveis de acesso são gerenciados por papéis. Todos os dados trafegam por validação com `Zod`, tanto na entrada quanto na saída. As mensagens de erro são claras e padronizadas, com o uso adequado dos códigos HTTP como `400 Bad Request`, `403 Forbidden` e `404 Not Found`. A documentação OpenAPI da API é gerada automaticamente com base nos schemas do Zod, garantindo sincronia entre código e documentação.

6. Documentação da API RESTful

A documentação de uma API é um componente essencial em qualquer sistema de software, especialmente em aplicações com arquitetura cliente-servidor. Durante o desenvolvimento deste projeto, o processo de documentação não apenas serviu como referência para a equipe, mas também reforçou o aprendizado de boas práticas em engenharia de software adquiridas ao longo da graduação em Ciência da Computação. O contato com ferramentas de geração automática de documentação e a integração entre validação de dados e contratos de API refletiram a aplicação prática de conceitos teóricos como especificação formal, modularização e interoperabilidade entre sistemas. Este processo fortaleceu também a capacidade de planejar sistemas extensíveis e autodescritivos, características valorizadas tanto em ambientes acadêmicos quanto profissionais.

A API do servidor foi construída com base no padrão REST, com foco em manter contratos claros e previsíveis para os clientes consumidores, como o dashboard web e o aplicativo móvel. A documentação da API é gerada automaticamente a partir dos schemas de validação construídos com a biblioteca `Zod`[McDonnell 2024]. A ferramenta utilizada para essa geração foi o pacote `zod-to-openapi`, que converte os esquemas definidos em Zod para o padrão `OpenAPI 3.0`, possibilitando assim uma integração fluida com visualizadores e ferramentas de testes.

O processo de geração da documentação é executado por meio do script localizado em `scripts/generateDocs.ts`. Esse script percorre os arquivos localizados em `src/app/schemas/API/input`, onde estão definidos os esquemas de entrada para

cada controlador da API, e em `src/app/schemas/API/output`, que define os formatos das respostas esperadas. Além disso, os modelos de entidades reutilizáveis utilizados em ambos os tipos de esquema são descritos na pasta `src/app/schemas/models`. Após a execução do script, a documentação completa da API é armazenada no arquivo `openapi.json`, gerado automaticamente na raiz do projeto.

A visualização do conteúdo gerado pode ser feita por meio da ferramenta `Zudoku`, que interpreta o arquivo `openapi.json` e apresenta a documentação em uma interface interativa e acessível. A Figura **??** ilustra a visualização da rota `POST /stations` na interface da Zudoku, evidenciando a clareza das entradas esperadas, validações aplicadas e o formato das respostas retornadas.

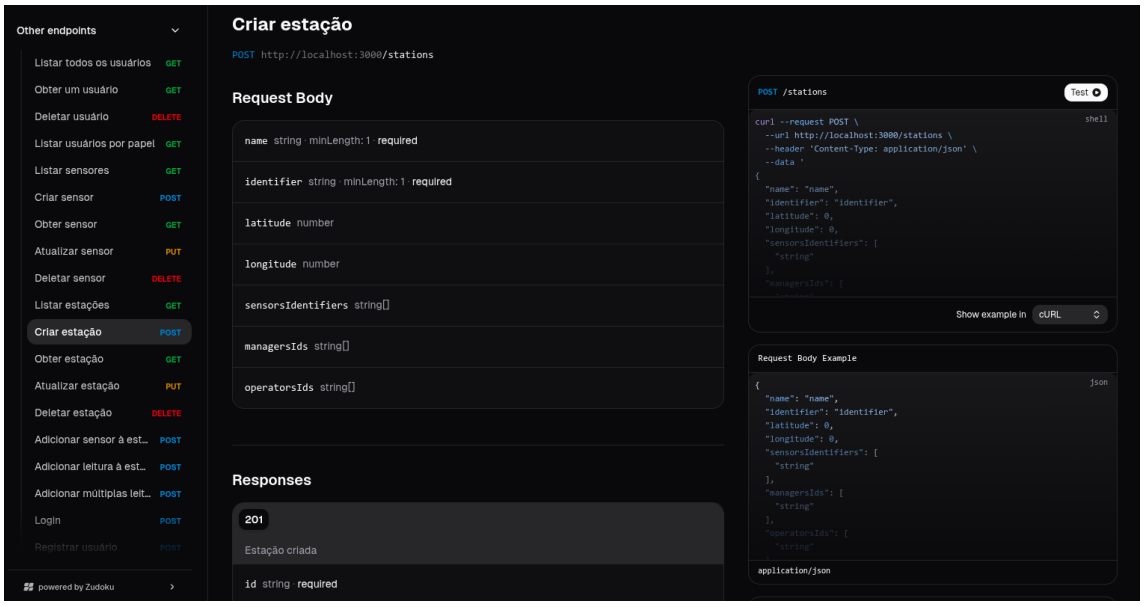

Figura 1. Documentação da rota `POST /stations`

A título de exemplo, a seguir é apresentado um retorno típico da API para a rota `GET /stations/{identifier}`. Esse retorno apresenta uma estrutura completa contendo dados da estação, sensores associados, operadores e gestores vinculados, bem como as leituras mais recentes registradas.

```
{
  "id": "1",
  "name": "Estacao Modelo",
  "identifier": "estacao-001",
  "latitude": -7.1234567,
  "longitude": -35.7654321,
  "sensors": [
    {
      "id": "s1",
      "identifier": "sensor-tmp-001",
      "name": "Temperatura",
      "measureUnit": "C"
    }
  ],
```

```
  "managers": [
    {
      "id": "u1",
      "username": "gestor1",
      "roles": [{ "id": "r1", "name": "GESTOR" }]
    }
  ],
  "operators": [
    {
      "id": "u2",
      "username": "operador1",
      "roles": [{ "id": "r2", "name": "OPERATOR" }]
    }
  ],
  "readings": [
    {
      "id": "r1",
      "timestamp": "2025-06-18T12:00:00Z",
      "value": 28.5,
      "sensor": {
        "id": "s1",
        "identifier": "sensor-tmp-001",
        "name": "Temperatura",
        "measureUnit": "C"
      }
    }
  ]
}
```

A validação de dados desempenha um papel crítico na segurança e integridade da aplicação. Com o uso do `Zod`, todas as entradas da API são verificadas antes de serem processadas pelos controladores. Isso garante que os dados estejam de acordo com o formato esperado, evitando falhas inesperadas ou entradas malformadas. A abordagem adotada também garante mensagens de erro consistentes e descritivas, o que contribui significativamente para a experiência de desenvolvimento e para a manutenção do sistema. Além disso, como a documentação é derivada diretamente dos schemas de validação, ela reflete fielmente a estrutura real da API, reduzindo a probabilidade de discrepâncias entre o comportamento esperado e o comportamento real.

A escolha dessa abordagem trouxe múltiplos benefícios. A geração automatizada da documentação garante que a documentação evolua junto com o código, eliminando a necessidade de atualizações manuais. Essa sincronia contínua contribui para a confiabilidade da API e reduz o risco de inconsistências entre o frontend e o backend. Além disso, a clareza dos contratos gerados a partir dos schemas fortalece a colaboração entre equipes, promove melhores práticas de versionamento e acelera o processo de integração com novos consumidores da API.

7. Processo de Desenvolvimento

O desenvolvimento do sistema servidor e do dashboard foi conduzido de maneira alinhada aos princípios da Engenharia de Software estudados ao longo da graduação em Ciência da Computação. Essa etapa prática proporcionou a oportunidade de aplicar conceitos como modularidade, versionamento, controle de requisitos e boas práticas de

codificação em um projeto real, com múltiplos atores e interfaces. A experiência permitiu consolidar habilidades técnicas relacionadas à organização de tarefas, implementação orientada a requisitos e uso de ferramentas colaborativas, além de reforçar a importância da rastreabilidade e da documentação contínua no ciclo de vida de sistemas.

Durante o projeto, optou-se por uma abordagem iterativa e incremental, com foco na entrega contínua de valor e no ajuste das funcionalidades ao longo do tempo. Essa estratégia facilitou a adaptação às necessidades técnicas identificadas ao longo do processo, assegurando que as decisões de arquitetura e implementação fossem tomadas com base em evidências práticas e feedback recorrente. A separação de responsabilidades no código, bem como o uso de ferramentas para controle de versão e organização de tarefas, foram fundamentais para manter a coesão e a escalabilidade do sistema.

O gerenciamento das atividades foi realizado por meio do recurso *GitHub Projects*, onde foram organizadas todas as tarefas relacionadas ao desenvolvimento do backend e do dashboard web. Utilizou-se a estrutura de *user stories* da metodologia ágil, permitindo que cada funcionalidade fosse descrita a partir da perspectiva do usuário. Cada história de usuário foi registrada como um cartão com título, descrição e critérios de aceitação bem definidos. Essa prática não só facilitou a visualização do progresso, como também auxiliou na priorização de tarefas e na rastreabilidade das decisões de implementação.

O controle de versão foi realizado com a ferramenta Git, e o repositório foi hospedado na plataforma GitHub.

Ao longo do projeto, foram seguidas boas práticas recomendadas na literatura e na disciplina de desenvolvimento de sistemas. A lógica de negócio foi separada da lógica de apresentação, promovendo a coesão dos módulos e facilitando o reaproveitamento e teste das unidades. Foram utilizadas variáveis de ambiente para parametrização de dados sensíveis e específicos de ambiente, como credenciais e configurações de banco de dados, promovendo segurança e flexibilidade entre ambientes de desenvolvimento e produção.

A segurança da API foi tratada com rigor: as requisições foram protegidas por autenticação via tokens JWT, e a comunicação entre cliente e servidor utilizou cookies com atributo `httpOnly` e criptografia de senhas, conforme diretrizes da documentação da MDN[MDN Web Docs Contributors 2024]. Além disso, o projeto foi estruturado de forma modular, com divisão clara entre controladores, repositórios, entidades, middlewares e serviços, o que facilitou tanto a manutenção quanto a inclusão de novos recursos ao longo do tempo.

Esse conjunto de práticas e ferramentas permitiu construir uma base de código robusta, segura e de fácil manutenção, favorecendo a escalabilidade do sistema e sua evolução contínua. O processo, além de cumprir os objetivos técnicos do projeto, contribuiu significativamente para o amadurecimento profissional e acadêmico ao aplicar, na prática, conhecimentos essenciais adquiridos ao longo da formação em Ciência da Computação.

8. Arquitetura em Camadas e Componentes de Suporte

A organização interna de um sistema de software afeta diretamente sua manutenibilidade, escalabilidade e clareza. Ao longo do curso de Ciência da Computação, diversos padrões arquiteturais foram estudados, e sua aplicação prática se tornou essencial no desenvolvimento deste projeto. Ao adotar uma arquitetura em camadas, foi possível aplicar conceitos fundamentais da engenharia de software, como separação de responsabilidades, modularidade e baixo acoplamento. Este tipo de organização permite que as camadas evoluam independentemente, com impactos controlados nas demais partes do sistema.

O servidor web construído para este projeto foi projetado com base na arquitetura em camadas[Rocha 2018]. Essa abordagem visa garantir que as operações da aplicação ocorram de forma estruturada, desde a recepção da requisição HTTP até a execução da lógica de negócio e persistência dos dados. Essa estrutura modular também favorece a testabilidade do código e a organização colaborativa do projeto.

A camada de apresentação, localizada no diretório `routes/`, é responsável por definir os endpoints da API RESTful e associá-los às funções correspondentes nos controladores. Cada rota especifica o método HTTP utilizado e os middlewares que devem ser aplicados antes do redirecionamento para o controlador.

A camada de aplicação está implementada nos arquivos do diretório `controllers/`. Essa camada recebe as requisições processadas pelas rotas, interpreta os dados, aplica regras específicas de lógica de negócio e aciona os repositórios ou serviços necessários para atender à solicitação. As respostas são padronizadas em formato JSON, com os códigos HTTP apropriados.

A camada de domínio é composta pelas entidades do projeto, localizadas no diretório `entities/`. Cada entidade representa uma tabela no banco de dados relacional, com seus atributos e relacionamentos devidamente mapeados através de decorators do TypeORM. Essa camada encapsula as estruturas centrais do sistema, como Usuário, Estação, Sensor e Leitura.

A camada de persistência é responsável por encapsular o acesso ao banco de dados e está implementada nos arquivos do diretório `repositories/`. Os repositórios fornecem métodos reutilizáveis e específicos para manipular as entidades. Eles representam a única interface entre o restante da aplicação e a camada de armazenamento, promovendo independência e desacoplamento.

Complementando essas camadas, o sistema utiliza diversos componentes auxiliares. Os middlewares, localizados no diretório `middlewares/`, são funções executadas entre a recepção da requisição e sua entrega ao controlador. Entre eles estão o `AuthMiddleware` para verificação de tokens JWT, o `RequireRoleMiddleware` para controle de acesso baseado em papéis, o `ValidationMiddleware` para validação dos dados com `Zod`, e o `ErrorMiddleware` para tratamento global de erros.

Os schemas de validação e tipagem, organizados no diretório `schemas/`, foram implementados utilizando a biblioteca `Zod`[McDonnell 2024]. Esses schemas asseguram que os dados recebidos e enviados pela API estejam no formato esperado.

Além disso, eles permitem a geração automática de documentação no padrão OpenAPI, integrada com a ferramenta Zudoku.

Os serviços, localizados no diretório `services/`, encapsulam regras de negócio reutilizáveis, como a autenticação de usuários no `AuthService` ou a verificação de permissões no `AccessControlService`. Esses componentes foram fundamentais para manter os controladores enxutos e coesos.

O diretório `utils/` concentra funções auxiliares de uso geral, como extensão de erros personalizados, mensagens padronizadas ou lógica de formatação. Já a camada de configuração engloba arquivos como `database/data-source.ts`, onde está definida a conexão com o banco de dados MySQL[Corporation 2024], o carregamento de variáveis de ambiente com a biblioteca `dotenv`, a configuração de permissões de CORS e a inicialização do servidor em `index.ts`.

A estrutura resultante é altamente coesa, permitindo que cada parte do sistema seja facilmente entendida, testada e expandida. Essa organização favorece tanto a implementação incremental quanto a integração com outras ferramentas e plataformas futuras.

9. Organização da Codebase

O desenvolvimento de software envolve não apenas a implementação funcional, mas também uma estruturação cuidadosa do código para garantir manutenção, escalabilidade e legibilidade ao longo do tempo. Este capítulo apresenta a organização da codebase do projeto desenvolvido, destacando como o conhecimento adquirido na faculdade de Ciência da Computação contribuiu para a definição de uma arquitetura modular e limpa, com separação clara de responsabilidades. Compreender a estrutura do código é fundamental para facilitar o desenvolvimento colaborativo e a evolução do sistema, além de promover boas práticas profissionais.

Ao longo da graduação, conceitos como arquitetura de software, padrões de projeto e princípios de engenharia de software foram importantes para estabelecer uma base sólida para este trabalho. A divisão entre camadas de controle, serviço, persistência e validação, por exemplo, reflete esses conceitos aplicados na prática, mostrando a importância de um código organizado para garantir qualidade e eficiência no desenvolvimento.

A seguir, é apresentada uma descrição detalhada das principais pastas e arquivos que compõem a codebase, explicando suas funções e responsabilidades dentro do projeto.

A pasta `docs/` contém configurações relacionadas à geração automática de documentação da API via OpenAPI Generator, utilizando a biblioteca `@asteasolutions/zod-to-openapi`. Dentro dela, há o diretório `.openapi-generator/`, que armazena arquivos internos do gerador, e o arquivo `.openapi-generator-ignore`, responsável por excluir arquivos ou pastas durante o processo de geração da documentação.

A pasta `scripts/` inclui scripts utilitários que auxiliam no desenvolvimento, como `generateDocs.ts`, que gera documentação OpenAPI com base nos schemas definidos, e `populateReadings.ts`, que popula o banco de dados com leituras fictícias de sensores para fins de teste.

O diretório principal da aplicação está localizado em `src/`. Nele, a pasta `app/` contém as implementações centrais do sistema: `controllers/` implementam a lógica de controle para cada rota, como `AuthController` e `StationController`; `entities/` definem as entidades TypeORM que mapeiam as tabelas do banco MySQL, incluindo `User`, `Station` e `Sensor`; `middlewares/` agrupam middlewares globais e específicos por rota, como autenticação (`AuthMiddleware`), autorização (`RequireRoleMiddleware`) e tratamento de erros; `repositories/` compõem a camada de acesso a dados, com métodos para manipulação das entidades, exemplificados por `UserRepository` e `StationRepository`; `routes/` define e registra as rotas da API; `schemas/` traz a tipagem e validação de entrada e saída utilizando Zod, subdividida em `API/input/` e `API/output/` para os schemas específicos de controladores, e `models/` que contém schemas compartilhados das principais entidades; `services/` encapsula a lógica de negócios, separando-a dos controladores, como em `AuthService` e `AccessControlService`; `utils/` disponibiliza funções utilitárias, incluindo o tratamento de erros personalizados via `ErrorExtension`; e `index.ts` é o ponto de entrada da aplicação, onde o app Express é inicializado.

Ainda dentro de `src/`, a pasta `database/` guarda a configuração da conexão com o banco de dados por meio do TypeORM, através do arquivo `data-source.ts`, e scripts para popular o banco com dados iniciais localizados em `seeders/`, como o `UserSeed.ts`. O arquivo `seeds/init.ts` executa todos os seeders configurados para inicialização do banco.

Na raiz do projeto encontram-se os arquivos de configuração essenciais para o ambiente, linting e build, tais como `.gitignore`, `tsconfig.json`, `package.json`, `eslint.config.mjs` e `.prettierrc`. Também estão presentes os arquivos de definição da documentação da API, `openapi.json` e `openapitools.json`, o arquivo de orquestração dos contêineres Docker (`compose.yaml`) e o `readme.MD` que contém a documentação geral do projeto.

9.1. Resumo da Estrutura

A arquitetura adotada separa claramente os níveis de responsabilidade da aplicação. O frontend, implementado em Angular, realiza chamadas HTTP para os endpoints expostos na camada de apresentação da API. As requisições recebidas são autenticadas, validadas e direcionadas aos controladores, que são responsáveis por orquestrar as ações conforme a regra de negócio. Os controladores acessam os repositórios para persistência e consulta de dados e interagem com os serviços para executar regras de negócio específicas. A camada de entidades assegura que os dados manipulados estejam estruturados conforme as regras do domínio, enquanto os schemas garantem a validação das interações. Por fim, middlewares reforçam aspectos de segurança, integridade e controle de acesso ao longo do ciclo da requisição.

9.2. Fluxo de uma Requisição

O fluxo de processamento de uma requisição HTTP no servidor inicia com o envio da requisição pelo cliente, seja um aplicativo ou dashboard, para um endpoint da API, por exemplo, `POST /stations`. Em seguida, middlewares executam etapas importantes: o parsing do JSON transforma o corpo da requisição em um objeto JavaScript; a

autenticação via JWT valida o token no cabeçalho, retornando `401 Unauthorized` em caso de falha; e a validação Zod assegura que o corpo esteja no formato esperado, verificando campos obrigatórios e tipos válidos.

Após a validação, a requisição é roteada ao controlador correspondente, que extrai dados de `req.body`, `req.params`, entre outros, executa regras de autorização específicas, e chama métodos nos repositórios para manipular os dados. A camada de repositório utiliza o TypeORM para realizar operações no banco de dados, como consultas, inserções ou atualizações, retornando os resultados ao controlador.

O controlador, por sua vez, envia a resposta em formato JSON, com o código HTTP adequado (como `200 OK` ou `201 Created`), ou, em caso de erro, retorna códigos como `400`, `403`, `404` ou `500`, conforme a natureza da falha. Finalmente, o cliente recebe a resposta e atualiza a interface ou executa ações correspondentes.

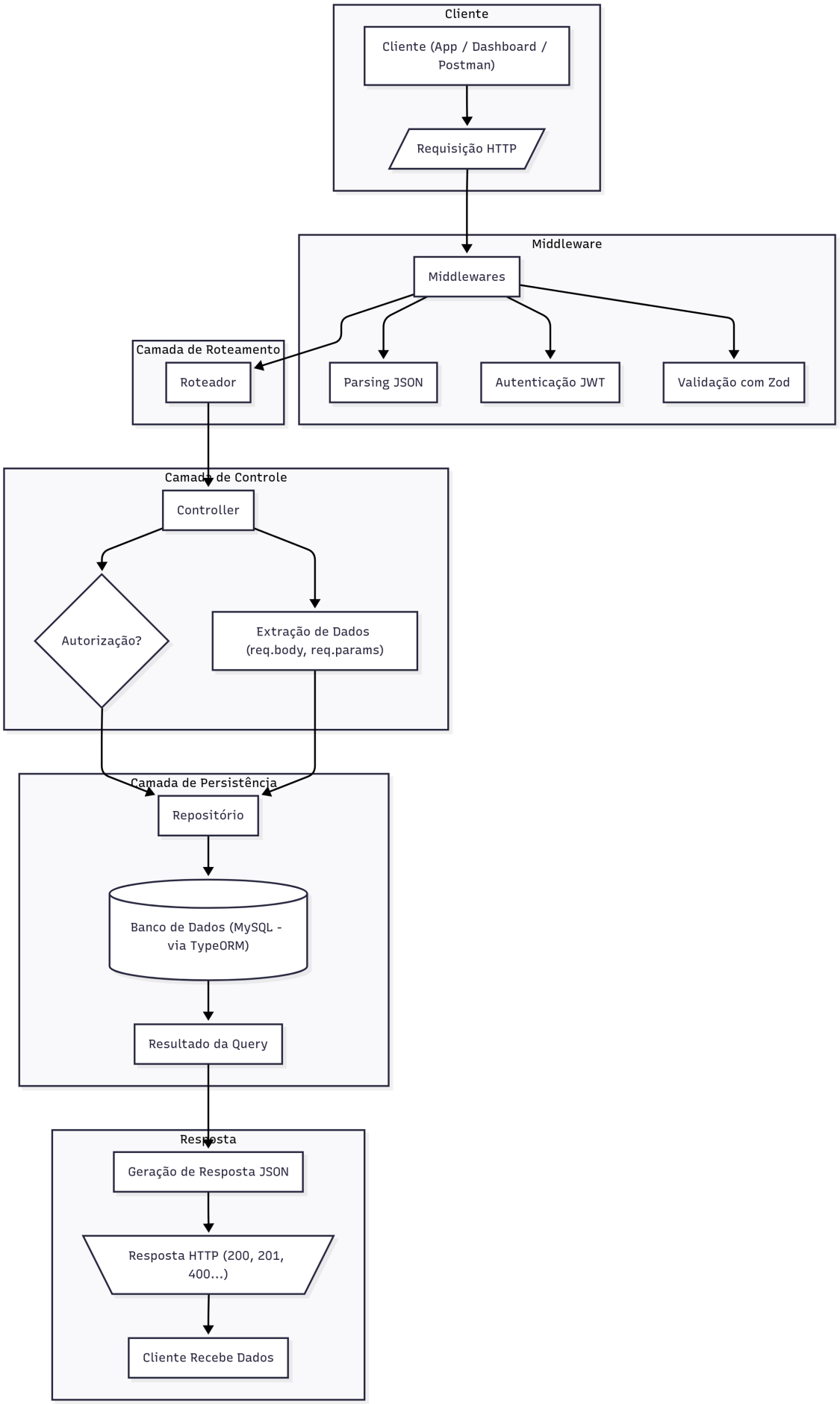

Figura 2. Diagrama do fluxo de uma requisição

9.3. Vantagens da Arquitetura Utilizada

Esta arquitetura modular proporciona manutenção facilitada, pois alterações em uma camada não impactam as demais. Além disso, promove a reutilização da lógica, visto que repositórios e middlewares podem ser compartilhados por diferentes controladores. A segurança é centralizada, com autenticação e validação tratadas de forma uniforme, o que contribui para a robustez do sistema. Finalmente, a organização modular permite escalabilidade, possibilitando a adição de novas funcionalidades sem comprometer a integridade do sistema existente.

10. Modelagem do Banco de Dados

A modelagem segue os princípios de normalização relacional e suporte a históricos de dados coletados por sensores.

Diagrama do Banco de Dados

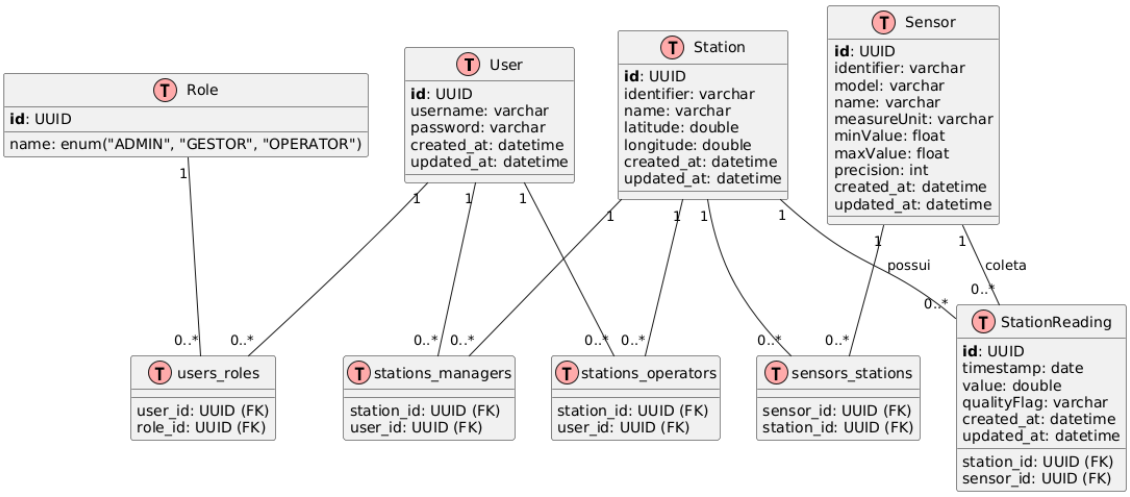

Figura 3. Estrutura Relacional do Banco de Dados

11. Dependências Principais do Projeto

Neste capítulo, são apresentadas as principais bibliotecas utilizadas no desenvolvimento do servidor web do sistema, evidenciando as escolhas tecnológicas adotadas e as razões que fundamentaram sua seleção. A escolha dessas dependências foi pautada tanto na aplicabilidade prática dos conhecimentos adquiridos durante a graduação quanto na busca por soluções que garantissem robustez, eficiência e escalabilidade frente aos desafios técnicos encontrados. Além disso, a experiência com essas ferramentas propiciou o aprimoramento de competências fundamentais à engenharia de software, como modelagem de dados relacionais, validação rigorosa de entradas e mecanismos seguros de autenticação e gerenciamento de sessões. A seguir, detalham-se as bibliotecas que compõem a stack principal do servidor, associando sua descrição às justificativas para sua adoção e apresentando exemplos extraídos diretamente do código-fonte do projeto.

11.1. Express

O `Express`[Holowaychuk and Contributors 2024] é um framework minimalista e flexível para Node.js, amplamente utilizado para a construção de APIs e servidores HTTP. Sua leveza, simplicidade na definição de rotas e compatibilidade com middlewares o tornam uma escolha adequada para aplicações que demandam rapidez no desenvolvimento e facilidade de manutenção. A popularidade do `Express` também assegura uma vasta documentação e comunidade ativa, fatores essenciais para a resolução ágil de eventuais problemas.

```
1 import express from "express";
2
3 const app = express();
4 app.use(express.json());
5
6 app.get("/stations", (req, res) => {
7   res.send("Lista de estacoes");
8 });
```

Listing 1. Inicialização do servidor com Express

11.2. TypeORM

A biblioteca `TypeORM`[Contributors 2024d] é um ORM (Object-Relational Mapper) para TypeScript e JavaScript que simplifica o mapeamento entre classes e tabelas do banco de dados. Sua utilização abstrai operações SQL complexas por meio de métodos orientados a objetos, reduzindo a probabilidade de erros e facilitando a manutenção do código. O suporte completo ao MySQL, sistema gerenciador de banco de dados escolhido para o projeto, foi determinante para a sua adoção.

```
1 @Entity("station_readings")
2 export class StationReading {
3   @PrimaryGeneratedColumn("uuid")
4   id: string;
5
6   @Column()
7   date: Date;
8
9   @Column("float")
10  temperature: number;
11
12  @ManyToOne(() => Station, (station) => station.readings)
13  station: Station;
14 }
```

Listing 2. Entidade "StationReading"

11.3. Zod

`Zod`[McDonnell 2024] é uma biblioteca para validação de esquemas em TypeScript que permite garantir a integridade dos dados recebidos nas requisições HTTP. Sua integração nativa com TypeScript e a capacidade de definir validações rigorosas tornam-na ideal para prevenir a propagação de dados incorretos para a camada de negócios, aumentando a segurança e confiabilidade do sistema.

```
export const createReadingSchema = z.object({
  temperature: z.number().min(0).max(100),
  tds: z.number(),
  pressure: z.number(),
  flow: z.number(),
  date: z.coerce.date()
});
```

Listing 3. Validação de leitura de sensores

11.4. jsonwebtoken

A biblioteca `jsonwebtoken`[Auth0 2024] é utilizada para criação e verificação de tokens JWT (JSON Web Token), amplamente adotados para autenticação segura em aplicações web. Sua escolha deve-se à robustez e à compatibilidade com padrões modernos de autenticação baseada em tokens, permitindo controlar o acesso à API de forma segura e eficiente.

```
const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET!, {
  expiresIn: "1h",
});

const decoded = jwt.verify(token, process.env.JWT_SECRET!);
```

Listing 4. Geração e verificação de token

11.5. bcrypt

Para garantir a segurança das senhas dos usuários, utilizou-se a biblioteca `bcrypt`, reconhecida como padrão para hashing resistente contra ataques de força bruta. Sua implementação assegura que dados sensíveis estejam protegidos, fator imprescindível em sistemas que gerenciam acesso a informações críticas.

```
const hashedPassword = await bcrypt.hash(password, 10);
const isValid = await bcrypt.compare(password, hashedPassword);
```

Listing 5. Hash e verificação de senha

11.6. dotenv

A biblioteca `dotenv` foi incorporada para o carregamento de variáveis de ambiente a partir de arquivos `.env`, facilitando a configuração do sistema em ambientes distintos (desenvolvimento e produção) e promovendo a separação segura entre credenciais e código-fonte, aspecto essencial para a segurança e portabilidade da aplicação.

```
import dotenv from "dotenv";
dotenv.config();

const PORT = process.env.PORT || 3000;
```

Listing 6. Configuração com dotenv

11.7. cors

Para permitir a comunicação entre aplicações hospedadas em domínios distintos, a biblioteca `cors`[Contributors 2024b] foi adotada, viabilizando o Cross-Origin Resource Sharing. Essa configuração é necessária especialmente para que o front-end desenvolvido em Angular realize requisições à API, tanto durante o desenvolvimento quanto em ambiente de produção.

```
1 import cors from "cors";
2
3 app.use(cors());
```

Listing 7. Uso do middleware CORS

11.8. mysql

O driver oficial `mysql` para Node.js foi utilizado para estabelecer a conexão entre o `TypeORM` e o banco de dados MySQL. A escolha pelo MySQL deve-se à sua ampla adoção, robustez, eficiência nas consultas e compatibilidade plena com o ORM empregado.

```
1 const AppDataSource = new DataSource({
2   type: "mysql",
3   host: process.env.DB_HOST,
4   port: 3306,
5   username: process.env.DB_USER,
6   password: process.env.DB_PASSWORD,
7   database: process.env.DB_NAME,
8   entities: [User, Station],
9 });
```

Listing 8. Configuração do DataSource com MySQL

11.9. uuid

A biblioteca `uuid` foi utilizada para a geração de identificadores únicos universais, que servem como chaves primárias das entidades do sistema. A adoção de UUIDs é especialmente indicada em sistemas distribuídos, pois assegura a unicidade dos registros mesmo em ambientes com múltiplas fontes de inserção de dados.

```
1 @PrimaryGeneratedColumn("uuid")
2 id: string;
```

Listing 9. Uso de UUID como Primary Key

12. Configuração do Servidor

Nesta seção, apresenta-se o processo de configuração do servidor da aplicação, que constitui uma etapa fundamental para garantir a correta interação entre o cliente e o backend. A configuração de servidores envolve conhecimentos adquiridos em diversas disciplinas da graduação em Ciência da Computação, como Redes de Computadores, Sistemas Distribuídos e Engenharia de Software, onde se aprende a importância da modularidade, segurança e tratamento adequado de erros para a construção de sistemas robustos e escaláveis.

Além disso, a prática da configuração e desenvolvimento de servidores contribui para consolidar conceitos essenciais de programação e arquitetura de software, demonstrando a aplicação concreta de padrões e boas práticas para o desenvolvimento de sistemas web modernos. O uso de ferramentas e bibliotecas específicas, como Express e dotenv, evidencia a integração entre teoria e prática, destacando a importância da gestão eficiente de variáveis de ambiente e modularização do código para facilitar a manutenção e evolução do sistema.

O servidor é configurado para fornecer uma interface robusta que lide com requisições HTTP, possibilitando a manipulação dos dados da aplicação e o acesso seguro aos seus recursos. A seguir, são apresentados os detalhes técnicos da configuração realizada, acompanhados de exemplos de código para melhor compreensão.

12.0.1. Inicialização do Servidor

Para iniciar o servidor, foi criada uma instância do Express, um framework minimalista para Node.js que facilita o desenvolvimento de aplicações web. Nessa etapa, configuram-se os principais *middlewares*, responsáveis pelo tratamento das requisições e respostas, incluindo o suporte para JSON e a configuração do CORS (Cross-Origin Resource Sharing), que permite acessos entre diferentes origens.

```
1  import express from "express";
2  import dotenv from "dotenv";
3  import cors from "cors";
4  import routes from "./routes";
5
6  dotenv.config();
7
8  const app = express();
9
10 app.use(express.json());
11 app.use(cors()); // Permitir acessos entre origens
12
13 app.use("/api", routes);
14
15 const PORT = process.env.PORT || 3000;
16 app.listen(PORT, () => {
17   console.log(`Server is running on port ${PORT}`);
18 });
```

Listing 10. Inicialização do servidor com Express

12.0.2. Configuração de Variáveis de Ambiente

As configurações sensíveis, como a porta do servidor e as credenciais de acesso ao banco de dados, são armazenadas em variáveis de ambiente para garantir segurança e flexibilidade na configuração da aplicação. Para isso, utiliza-se a biblioteca `dotenv`, que carrega essas variáveis a partir de um arquivo `.env`. A seguir, um exemplo do arquivo utilizado.

```
1 PORT=3000
```

```
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=senha
DB_NAME=projeto
JWT_SECRET=chave-secreta
```

Listing 11. Exemplo de arquivo .env

12.0.3. Definição de Rotas

As rotas da aplicação são organizadas em um arquivo separado, o que assegura a modularidade do código e facilita a manutenção. Cada rota está associada a controladores que executam as operações solicitadas. O exemplo abaixo demonstra a configuração de rotas para obtenção e criação de estações.

```
import { Router } from "express";
import { getStations, createStation } from "./controllers/
    stationController";

const router = Router();

router.get("/stations", getStations);
router.post("/stations", createStation);

export default router;
```

Listing 12. Definição de rotas em um arquivo separado

12.0.4. Tratamento de Erros

Para garantir uma resposta consistente e apropriada diante de erros, foi implementado um *middleware* global de tratamento de erros. Esse componente captura exceções que possam ocorrer durante o processamento das requisições, registra informações úteis para o diagnóstico e retorna uma resposta adequada ao cliente.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: "Internal Server Error" });
});
```

Listing 13. Middleware de tratamento de erros

12.0.5. Execução do Servidor

Após todas as configurações, o servidor é finalmente iniciado. A porta utilizada para escutar as requisições é definida pela variável de ambiente, garantindo que possa ser alterada sem necessidade de modificar o código-fonte.

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
```

```
3   console.log(`Server is running on port ${PORT}`);
4 });
```

Listing 14. Início da execução do servidor

‘‘‘

Se quiser, posso ajudar a adaptar ainda mais para a sua monografia, só avisar!

13. Aplicação Web

Nesta seção, apresenta-se a implementação da aplicação web desenvolvida para gerenciar as estações de monitoramento de sensores. A elaboração desta aplicação envolveu a aplicação prática de diversos conceitos aprendidos ao longo do curso de Ciência da Computação, como arquitetura de software, desenvolvimento orientado a componentes, e padrões de projeto para construção de interfaces web. Além disso, a necessidade de garantir usabilidade, modularidade e segurança na aplicação reforçou o aprendizado em engenharia de software e tecnologias de front-end modernas.

O desenvolvimento da aplicação foi realizado utilizando o framework `Angular 19`[Team 2024a], que propicia um ambiente estruturado para criação de aplicações web robustas e escaláveis. A adoção da biblioteca `Angular Material`[Team 2024b] permitiu a construção de uma interface visual moderna, responsiva e acessível, importante para atender a diferentes perfis de usuários, como operadores, gestores e administradores. Dessa forma, foi possível integrar conhecimentos teóricos com práticas de desenvolvimento ágil e design centrado no usuário, resultando em uma solução funcional e alinhada com as demandas do projeto.

A seguir, são detalhadas a arquitetura, funcionalidades, organização da base de código, bibliotecas utilizadas, aspectos de usabilidade e segurança da aplicação web.

13.1. Arquitetura da Aplicação Web

A aplicação front-end foi estruturada com base na separação entre Modelos, Visões e Serviços, conceito inspirado no padrão *Model-View-Controller (MVC)*[de Lemos et al. 2013], porém adaptado ao estilo e às melhores práticas de desenvolvimento com Angular. Os modelos consistem em interfaces e tipagens que representam as estruturas de dados da aplicação, como estações, sensores, usuários e leituras, garantindo tipagem forte e maior segurança na manipulação desses dados graças ao uso do TypeScript. As visões correspondem aos componentes e templates responsáveis pela renderização da interface e interação com o usuário, englobando elementos visuais como formulários, gráficos, abas e mapas. Os serviços encapsulam a lógica de negócio, cuidando da comunicação com o back-end via `HttpClient` e do gerenciamento de estado, promovendo reuso e desacoplamento entre lógica e interface.

A aplicação foi modularizada em páginas e componentes reutilizáveis, com roteamento configurado para proteger o acesso conforme o perfil do usuário.

13.2. Principais Funcionalidades Implementadas

No que tange às funcionalidades, a aplicação contempla a autenticação e registro de usuários, onde o login é realizado por meio de formulário com feedback visual e

ocultação de senha, e o registro permite a escolha da função (OPERATOR, GESTOR), integrando-se à API para autenticação via JWT. O dashboard das estações apresenta cada estação em um *card* com abas interativas, exibindo gráficos implementados com `Chart.js`[Contributors 2024a] por meio do wrapper `ng2-charts`[ng2-charts Contributors 2024], além de dados da última leitura, localização geográfica utilizando Leaflet[Contributors 2024c], alertas visuais e usuários associados. O cadastro de estações oferece formulário responsivo para registro de novas estações, seleção de sensores, operadores e gestores, configuração da localização geográfica e submissão para a API REST com redirecionamento automático. O cadastro de sensores inclui formulário validado com campos para identificador, modelo, nome, unidade, precisão e limites, permitindo interação direta com a API, além de feedbacks visuais de validação. A aplicação também dispõe de uma tela dedicada para visualização de relatórios por estação e intervalo, com funcionalidades planejadas para download, visualização e exclusão de relatórios, além de filtros por período e seleção de sensores e estações. Por fim, a localização geográfica das estações é exibida em um componente baseado na biblioteca `Leaflet`, com marcadores personalizados e integração às informações de cada estação.

13.3. Organização da Codebase da Aplicação Web

A base de código da aplicação está organizada para facilitar manutenção e extensibilidade. As páginas completas, como login, registro de usuário, dashboard e listagem de estações, encontram-se em `src/app/pages/`. Componentes reutilizáveis, como formulários de sensores e estações, visualizações de localização, leituras e tabelas de usuários, estão em `src/app/components/`. Os serviços responsáveis pela comunicação com a API, incluindo autenticação, estações, sensores, usuários e dashboard, estão localizados em `src/app/services/`. Componentes globais e estilos compartilhados ficam em `src/app/shared/`, enquanto as configurações específicas para diferentes ambientes de execução estão em `src/environments/`.

13.4. Bibliotecas e Tecnologias Utilizadas

Para implementar as funcionalidades descritas, foram utilizadas as bibliotecas e tecnologias principais do ecossistema Angular, como `@angular/core`, `@angular/router` e `@angular/forms`, que formam a base do framework. A biblioteca `Angular Material` foi adotada para fornecer componentes visuais prontos e acessíveis. Para visualização gráfica de dados em tempo real, utilizou-se `Chart.js` integrado com `ng2-charts`. A biblioteca `Leaflet` foi escolhida para o mapeamento interativo das localizações das estações. Para o gerenciamento reativo de estado e observação de dados, empregou-se `RxJS` e o novo sistema de `Signals` do Angular. Para garantir a geração de identificadores únicos, foi utilizado o pacote `uuid`.

13.5. Estilo e Usabilidade

O design da aplicação segue um padrão visual consistente, com uso predominante de cores neutras e elementos destacados em azul para chamar a atenção do usuário aos pontos de foco. Os formulários implementados adotam validações reativas com mensagens claras para facilitar a correção de erros durante o preenchimento. O layout é responsivo, combinando o uso de Grid CSS e Flex Layout para garantir boa adaptação a

diferentes tamanhos de tela. Além disso, são oferecidos feedbacks visuais durante carregamentos por meio de spinners e barras de progresso, melhorando a percepção do usuário sobre o estado da aplicação.

13.6. Segurança e Acesso

No que concerne à segurança, o `AuthService` mantém o estado do usuário autenticado em um sinal reativo, o que possibilita o bloqueio ou a exibição condicional de funcionalidades conforme o perfil. Embora a proteção de rotas ainda esteja em expansão, ela poderá ser implementada utilizando `RouteGuards` no `AppRoutingModule`, garantindo controle de acesso refinado e alinhado com as necessidades futuras da aplicação.

14. Capturas de Tela da Aplicação Web

A seguir, são apresentadas capturas de tela da aplicação web que ilustram as principais funcionalidades implementadas no sistema. Cada imagem demonstra uma interface relevante no fluxo de uso da plataforma.

14.1. Tela de Login

A tela de login permite que usuários se autentiquem utilizando suas credenciais.

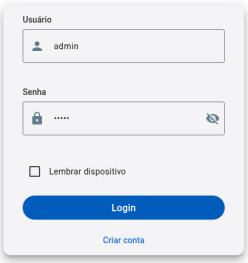

Figura 4. Tela de login com validação de campos

14.2. Tela de Registro de Usuário

O formulário de registro permite que novos usuários sejam cadastrados no sistema, com campos obrigatórios como nome de usuário, senha e tipo de perfil (gestor ou operador).

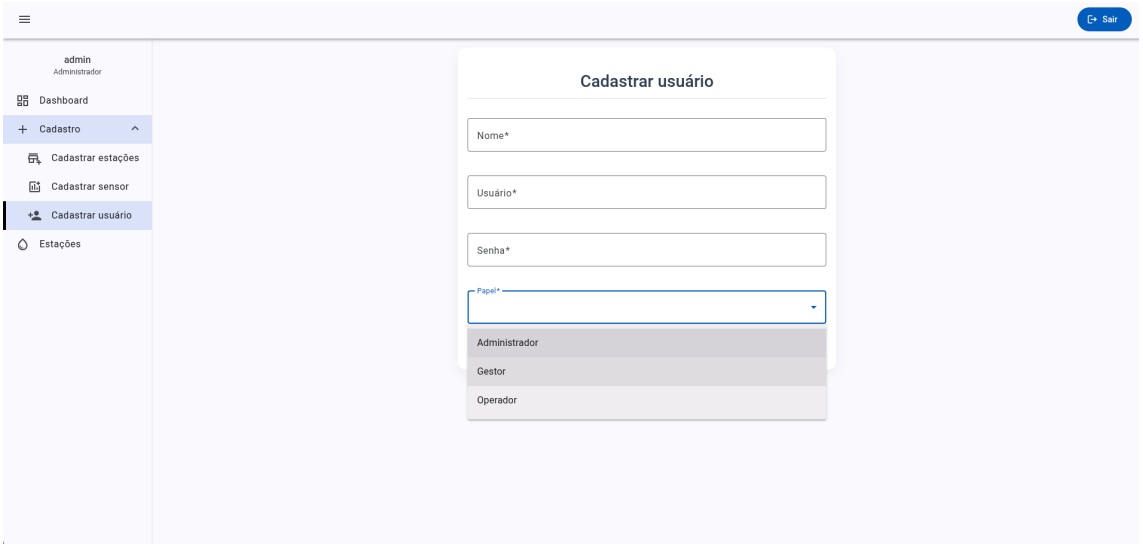

Figura 5. Formulário de registro de novos usuários

14.3. Dashboard de Estações

O dashboard apresenta uma visão interativa das estações monitoradas. É possível visualizar gráficos, alertas, localização e informações detalhadas de sensores.

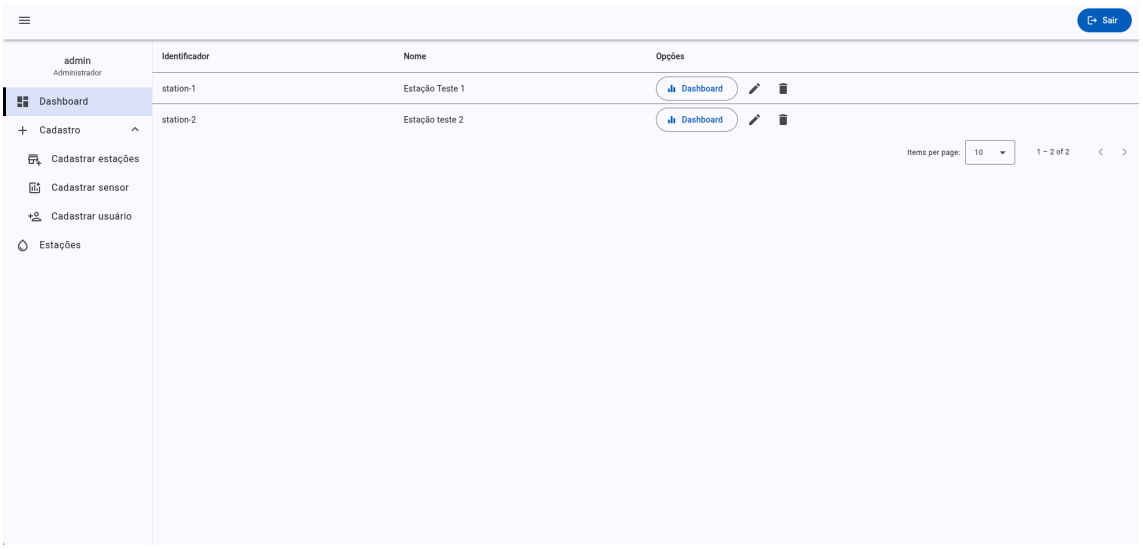

Figura 6. Dashboard interativo de uma estação de dessalinização

14.4. Formulário de Cadastro de Estações

Esta tela permite o cadastro de novas estações, incluindo seus sensores, localização geográfica e usuários responsáveis.

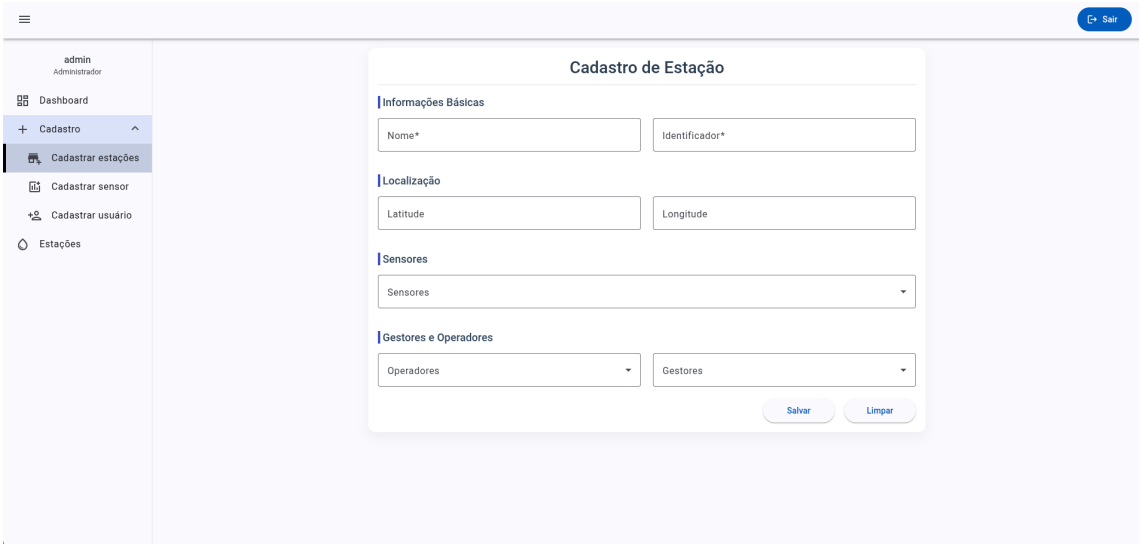

Figura 7. Formulário de cadastro de estação com validação

14.5. Formulário de Cadastro de Sensores

Componente dedicado ao registro de sensores no sistema, com campos para modelo, unidade, limites de operação e precisão.

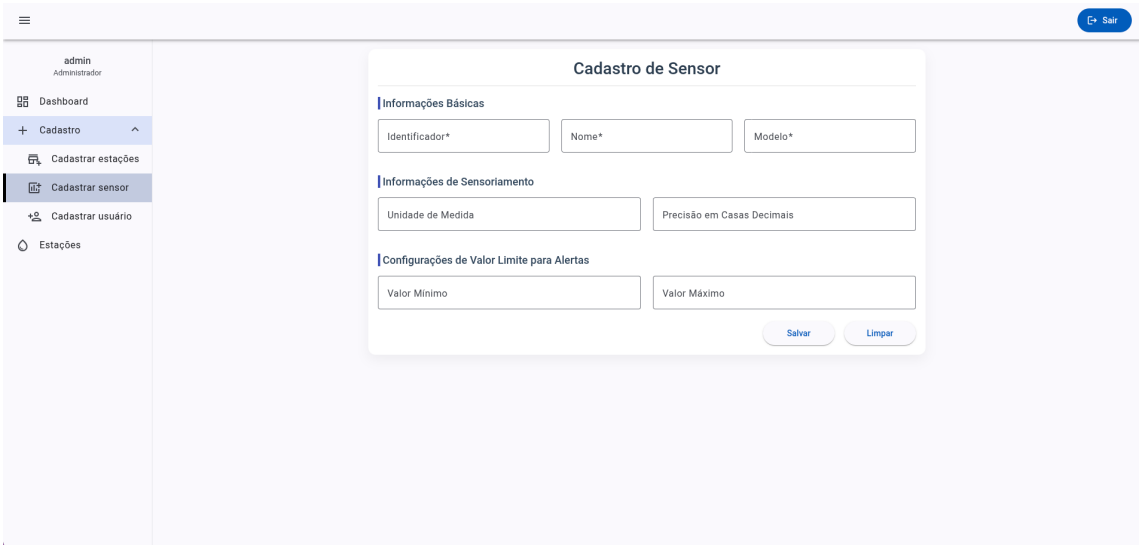

Figura 8. Cadastro de novo sensor com campos validados

15. Conclusão

O desenvolvimento deste projeto representou uma oportunidade significativa para a aplicação prática de conceitos fundamentais aprendidos ao longo da graduação em Ciência da Computação, tais como engenharia de software, arquitetura de sistemas, segurança da informação e desenvolvimento web. A experiência de construir uma solução tecnológica integrada para o monitoramento de estações de dessalinização permitiu não apenas consolidar conhecimentos teóricos, mas também desenvolver

competências técnicas essenciais para o ambiente profissional. Além disso, o projeto ressaltou a importância da adaptação de tecnologias modernas a contextos reais e desafiadores, promovendo uma visão crítica sobre o impacto social e ambiental da computação aplicada.

Esta seção apresenta uma reflexão sobre as contribuições técnicas, os desafios enfrentados e as perspectivas futuras do sistema desenvolvido. O trabalho buscou construir uma arquitetura cliente-servidor robusta, onde a divisão clara de responsabilidades e o emprego de boas práticas foram fundamentais para garantir modularidade, escalabilidade e manutenção. Foram explorados recursos avançados de tipagem estática, validação de dados e segurança, enfatizando a integração entre front-end e back-end por meio de contratos fortemente tipados, prática que reflete a maturidade alcançada no desenvolvimento de software ao longo do curso.

Por fim, destaca-se o papel social da tecnologia desenvolvida, cujo objetivo transcende o campo da computação para contribuir com a gestão sustentável dos recursos hídricos em regiões semiáridas, alinhando conhecimento acadêmico com demandas concretas da sociedade.

Síntese das Contribuições

O projeto proporcionou uma arquitetura modular e sustentável, estruturada em camadas claras, incluindo controllers, services, repositories, middlewares e entities, o que favorece escalabilidade e facilita a manutenção do sistema. A API RESTful foi construída com validação rigorosa e documentada automaticamente, seguindo boas práticas do desenvolvimento web moderno e facilitando futuras integrações. Para garantir a segurança e o controle de acesso, implementou-se uma gestão granular de permissões baseada em papéis, como `ADMIN`, `GESTOR` e `OPERATOR`, com autorização detalhada por rota. No front-end, uma interface web intuitiva foi desenvolvida, oferecendo dashboards funcionais com gráficos interativos, filtros dinâmicos e feedback visual, permitindo o monitoramento em tempo real das estações. O projeto também considerou a portabilidade e a implantação facilitada, utilizando `Docker`, variáveis de ambiente configuráveis por meio do `dotenv` e scripts de inicialização automatizados para garantir flexibilidade em diferentes ambientes.

Desafios Enfrentados

Entre os principais desafios técnicos, destacou-se a modelagem de dados capaz de refletir a hierarquia e as relações complexas entre sensores, estações e usuários com múltiplos papéis, exigindo uma estrutura flexível e consistente. A validação rigorosa dos dados de entrada da API foi essencial para evitar inconsistências e vulnerabilidades, demandando o uso cuidadoso de bibliotecas especializadas como `Zod`. Além disso, a integração harmoniosa entre o back-end e o front-end representou um desafio significativo, que foi superado com a adoção de contratos fortemente tipados e padronizados, garantindo a consistência e facilitando a manutenção e evolução do sistema. Esses obstáculos foram enfrentados por meio de uma abordagem iterativa, validação constante junto aos orientadores e aplicação de metodologias ágeis.

Impacto Técnico, Acadêmico e Social

O impacto do projeto é multifacetado. Tecnicamente, foi criada uma base tecnológica modular, segura e escalável, que pode ser reutilizada em outras aplicações de

sensoriamento remoto e monitoramento ambiental. Acadêmicamente, o projeto serviu como um importante elo entre o conhecimento teórico da graduação e sua aplicação prática, englobando disciplinas como bancos de dados, redes de computadores, engenharia de software e programação web. No aspecto social, o sistema contribui diretamente para comunidades em regiões semiáridas, promovendo o uso racional da água dessalinizada e aumentando a transparência na gestão dos sistemas, o que representa um benefício real para a qualidade de vida das populações atendidas.

Perspectivas Futuras

Quanto à evolução do projeto, destacam-se algumas frentes importantes para continuidade. A finalização da integração com o aplicativo móvel e o firmware do ESP32 permitirá a coleta e o envio de dados em campo, consolidando o ciclo completo de monitoramento. A implantação prática com estações reais possibilitará o recebimento de feedback dos usuários finais e ajustes necessários. Futuramente, pretende-se incluir funcionalidades avançadas como exportação de dados, auditoria detalhada e logs de uso para maior transparência e segurança. O desenvolvimento de pipelines de integração e entrega contínua (`CI/CD`) será essencial para manter a qualidade e a agilidade no desenvolvimento. Além disso, otimizações de performance e segurança deverão ser implementadas para suportar escalas maiores de dados e usuários, preparando o sistema para futuras expansões.

Considerações Finais

A realização deste projeto proporcionou uma experiência valiosa, integrando teoria e prática em um contexto real de impacto social. A maturidade técnica alcançada, aliada ao aprendizado em metodologias ágeis e trabalho colaborativo, foi fundamental para compreender o ciclo de vida completo de sistemas complexos. Acredita-se que o sistema desenvolvido está pronto para ser ampliado, evoluído e replicado, contribuindo de forma relevante para a área de tecnologias aplicadas ao saneamento e à sustentabilidade ambiental.

Referências

Auth0 (2024). jsonwebtoken - json web token implementation for node.js. https://github.com/auth0/node-jsonwebtoken. Versão 9.x.

Contributors, C. (2024a). Chart.js - simple yet flexible javascript charting for designers & developers. https://www.chartjs.org/. Versão 4.x.

Contributors, E. (2024b). cors - node.js cors middleware. https://github.com/expressjs/cors. Versão 2.x.

Contributors, L. (2024c). Leaflet - javascript library for interactive maps. https://leafletjs.com/. Versão 1.9.x.

Contributors, T. (2024d). Typeorm - orm para typescript e javascript (es7, es6, es5). https://typeorm.io. Versão 0.3.x.

Corporation, O. (2024). Mysql - the world's most popular open-source database. https://www.mysql.com. Versão 8.x.

de Lemos, M. F., Oliveira, P. C., Ruela, L. C., da Silva Santos, M., Slveira, T. C., and de Sousa Reis, J. C. (2013). Aplicabilidade da arquitetura MVC em uma aplicação web (WebApps). *RE3C-Revista Eletrônica Científica de Ciência da Computação*, 8(1).

Holowaychuk, T. and Contributors, E. (2024). Express - fast, unopinionated, minimalist web framework for node.js. https://expressjs.com. Versão 4.x.

McDonnell, C. (2024). Zod - typescript-first schema declaration and validation library. https://zod.dev. Versão 3.x.

MDN Web Docs Contributors (2024). Using http cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies. Acessado em 21 de junho de 2025.

ng2-charts Contributors (2024). ng2-charts - angular wrapper for chart.js. https://github.com/valor-software/ng2-charts. Versão 4.x.

Node.js contributors (2024). Node.js JavaScript runtime. https://nodejs.org/. Versão 20.x, disponível em: https://nodejs.org.

Raharjana, I. K., Siahaan, D., and Fatichah, C. (2021). User stories and natural language processing: A systematic literature review. *IEEE access*, 9:53811–53826.

Rocha, J. G. (2018). Arquitetura em camadas com uso do paradigma mvc e processo unificado na programação de software orientado a objetos. *Tecnologias em Projeção*, 9(1):31–49.

Team, A. (2024a). Angular - web framework. https://angular.io/. Versão 17.x.

Team, A. (2024b). Angular material - material design components for angular. https://material.angular.io/. Versão 17.x.

Valente, M. T. (2020). Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, 1(24).