UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

THAÍS DOS SANTOS RODRIGUES

EXPANSÃO DE FUNCIONALIDADES EM SISTEMA IOT DE CONTROLE DE ACESSO ATRAVÉS DE API HTTP

Recife
2024

THAÍS DOS SANTOS RODRIGUES

EXPANSÃO DE FUNCIONALIDADES EM SISTEMA IOT DE CONTROLE DE ACESSO ATRAVÉS DE API HTTP

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Orientador(a): Prof. Dr. Geraldo Leite Maia Junior

Coorientador(a): Prof. Dr. Marcio Evaristo da Cruz Brito

Recife
2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Rodrigues, Thaís dos Santos.
Expansão de funcionalidades em sistema IoT de controle de acesso através de API HTTP / Thaís dos Santos Rodrigues. - Recife, 2024.
119 p. : il., tab.

Orientador(a): Geraldo Leite Maia Junior
Cooorientador(a): Marcio Evaristo da Cruz Brito
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia de Controle e Automação - Bacharelado, 2024.
Inclui referências, apêndices, anexos.

1. API HTTP. 2. Expo. 3. IoT. 4. Javascript. 5. Node.Js. I. Maia Junior, Geraldo Leite . (Orientação). II. Brito, Marcio Evaristo da Cruz . (Coorientação). IV. Título.

620 CDD (22.ed.)

THAÍS DOS SANTOS RODRIGUES

EXPANSÃO DE FUNCIONALIDADES EM SISTEMA IOT DE CONTROLE DE ACESSO ATRAVÉS DE API HTTP

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Aprovado em: 06/08/2024.

BANCA EXAMINADORA

_____

Prof. Dr. Geraldo Leite Maia Junior (orientador)
Universidade Federal de Pernambuco

_____

Prof. Dr. Eduardo José Barbosa
Universidade Federal de Pernambuco

_____

Prof. Dr. Marcio Rodrigo Santos de Carvalho
Universidade Federal de Pernambuco

Este trabalho é dedicado a todos que de alguma forma contribuíram para esta construção. Agradeço profundamente à minha família pelo apoio constante, compreensão e amor durante toda minha trajetória acadêmica. Aos meus amigos, pela paciência e incentivo nos momentos difíceis. Aos professores, pela orientação, conhecimento e inspiração contínua. Aos colegas de curso, pelas valiosas trocas de experiências e aprendizados. E a todos que, de alguma forma, contribuíram para a realização deste trabalho, o meu mais sincero agradecimento.

AGRADECIMENTOS

Primeiramente, gostaria de expressar minha profunda gratidão à Helena Alencar, cujo apoio incondicional durante toda a graduação, especialmente no desenvolvimento deste trabalho, foi fundamental. Aos meus pais, Iranete e Márcio, que sempre acreditaram em mim e foram fontes inesgotáveis de amor e incentivo. À minha irmã mais velha, Maíra, que sempre apoiou meus sonhos e esteve ao meu lado em cada passo. Aos meus irmãos mais novos, Gabriela e Pedro, que me lembram da leveza da juventude e me inspiram a ser uma pessoa melhor a cada dia. Agradeço também a todos os meus familiares que sempre me apoiaram.

Aos meus amigos, minha gratidão eterna. Milena, por estar sempre ao meu lado, Rodrigo, por conseguir me fazer rir mesmo nos momentos mais difíceis, Cláudio e Rogério, meus parceiros, pela amizade e apoio constantes. E a todos os outros amigos que compartilharam essa jornada comigo dentro desta instituição.

Agradeço profundamente aos projetos de extensão que fizeram parte da minha formação: a equipe de robótica Maracatronics, Navicula BoatDesign, e o grupo de pesquisa Computação Afetiva. O conhecimento e as experiências adquiridos nesses projetos são inestimáveis e marcaram profundamente a minha trajetória acadêmica e pessoal.

Aos professores Alexander, Fabrício, Douglas, Márcio, e, também, ao meu gestor Diogo, minha gratidão pela paciência, suporte e orientação, especialmente nesta reta final do trabalho. Vocês foram essenciais para o meu crescimento acadêmico e profissional. Aos demais professores que tiveram a paciência e o interesse de ensinar e tirar minhas dúvidas, meu sincero agradecimento.

Um agradecimento especial à Lorena e Eduardo por terem aberto as portas de sua casa para que eu tivesse estrutura e pudesse estar próximo da universidade para finalizar este trabalho. A todos aqueles que contribuíram de alguma forma para minha trajetória acadêmica e pessoal, o meu mais sincero agradecimento. Cada um de vocês, mesmo que não mencionado individualmente, tem um lugar especial em meu coração e fez parte dessa conquista.

> O treinamento de usuários consiste em parte do processo de educação, em base repetitiva, compreende ações e/ou estratégias para desenvolver determinadas habilidades ou habilidades específicas do usuário por desconhecer situações específicas de uso da biblioteca e seus recursos informacionais, que envolvem o conjunto de meios necessários para tal (DIAS; PIRES, 2004, p. 36).

RESUMO

O presente estudo documenta a expansão de um sistema de controle de acesso, com a criação de uma API HTTP e a continuidade do desenvolvimento do aplicativo mobile, que foi inicialmente desenvolvido por Maria Helena Rocha de Alencar Bezerra Maria Helena Rocha de Alencar Bezerra (BEZERRA, 2024). O objetivo do sistema é monitorar, gerenciar e facilitar o acesso a salas do Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, capturar dados de sensores de tensão, temperatura e umidade, e acionar equipamentos como trancas de portas, aparelhos de ar-condicionado e lâmpadas.

A proposta baseia-se na hipótese de que a utilização de uma API HTTP intermediária promoverá a troca segura de dados entre um aplicativo *mobile*, um *broker* MQTT e um banco de dados *PostgreSQL*, além de melhorar a eficiência do sistema. O primeiro *software* corresponde a um aplicativo para aparelhos celulares no qual podem ser credenciados usuários para que tenham acesso a um painel de controle onde são disponibilizados acionadores dos equipamentos e dados dos sensores. Esta aplicação foi desenvolvida utilizando a plataforma *Expo,* que faz uso da linguagem *React Native* no presente trabalho. O segundo software, escrito em *Javascript*, atua como infraestrutura do primeiro, recebendo comandos do aplicativo via HTTP, processando-os, armazenando dados em um banco *PostgreSQL* e traduzindo as requisições para um centralizador IoT, conhecido como *broker* MQTT. Como dito, trata-se de um programa intermediário, hospedado neste projeto em uma R*aspberry PI* 4. Para o desenvolvimento deste último foi utilizado o *NodeJs* com o *framework Express* e uma biblioteca *MQTT*. A validação das hipóteses foi realizada por meio de testes práticos, incluindo a verificação do uso de memória e espaço em disco, bem como testes de comunicação que confirmaram a utilização dos protocolos MQTT e HTTP. Os resultados indicam uma melhoria significativa na eficiência do sistema e validam a utilização desses protocolos, destacando a importância da implementação de soluções de IoT econômicas e eficazes em ambientes acadêmicos.

Palavras-chave: API HTTP; *Expo*; IoT; *Javascript*; *Node.Js*.

ABSTRACT

This study documents the expansion of an access control system, with the creation of an HTTP API and the continuation of the development of the mobile application, which was initially developed by Maria Helena Rocha de Alencar Bezerra (BEZERRA, 2024). The system aims to monitor, manage, and facilitate access to rooms in the Department of Electrical Engineering at the Federal University of Pernambuco, capture data from voltage, temperature, and humidity sensors, and control equipment such as door locks, air conditioners, and lights.

The proposal is based on the hypothesis that using an intermediate HTTP API will promote secure data exchange between a mobile application, an MQTT broker, and a PostgreSQL database, as well as improve system efficiency. The first software corresponds to a mobile application in which users can be authenticated to access a control panel where equipment actuators and sensor data are available. This application was developed using the Expo platform, which utilizes the React Native language in this work. The second software, written in JavaScript, serves as the infrastructure for the first, receiving commands from the application via HTTP, processing them, storing data in a PostgreSQL database, and translating requests to an IoT centralizer known as an MQTT broker. As mentioned, this is an intermediate program, hosted in this project on a Raspberry Pi 4. NodeJs with the Express framework and an MQTT library were used for its development. The validation of the hypotheses was carried out through practical tests, including memory and disk space usage checks, as well as communication tests that confirmed the use of MQTT and HTTP protocols. The results indicate a significant improvement in system efficiency and validate the use of these protocols, highlighting the importance of implementing cost-effective and efficient IoT solutions in academic environments.

Keywords: API HTTP; *Expo*; IoT; *JavaScript*; *Node.Js.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARM	Advanced RISC Machine
AVAC	Aquecimento, Ventilação e Ar-Condicionado
AWS	Amazon *Web* Services
BaaS	Backend-as-a-Service
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheet
DEE	Departamento de Engenharia Elétrica
GPIO	General Purpose Input/Output
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
iOS	iPhone Operating System
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON *Web* Token
LAN	Local Area Network
LTS	Long-term support
MB	Megabytes
NFC	Near Field Communication
NoSQL	Not Only Structured Query Language
NPM	Node Package Manager
NPX	Node Package Executor
ORM	Object-Relational Mapping
PHP	Personal Home Page
QR	Quick Response
RAM	Random Access Memory
REST	Representational State Transfer

RFID	Radio Frequency Identification
RSA	Rivest-Shamir-Adleman
SDK	Software Development Kit
SGBD	Sistemas de Gerenciamento de Banco de Dados
SOAP	Subjetivo, Objetivo, Avaliação e Plano
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UFPE	Universidade Federal de Pernambuco
URI	Uniform Resource Identifier
USB	Universal Serial Bus
UUID	Universally Unique Identifier
XML	Extensible Markup Language
WAN	Wide Area Network

SUMÁRIO

1 INTRODUÇÃO

Este trabalho é uma continuação do trabalho de conclusão de curso, intitulado "Controle de Acesso a Ambientes Físicos Utilizando Ferramentas de IoT e Computação na Borda" desenvolvido por (BEZERRA, 2024). No qual foi criado um aplicativo de controle de acesso de baixo custo, operando de forma independente da conexão à internet para o Departamento de Engenharia Elétrica (DEE) da Universidade Federal de Pernambuco (UFPE). Esse aplicativo utilizava a plataforma *Supabase* como serviço de back-end, integrando diversas tecnologias como *Raspberry Pi*, *Expo*, *Node-RED*, protocolo *WebSocket* e MQTT (BEZERRA, 2024).

O *Node-RED* foi utilizado como intermediário entre a aplicação e o *broker* MQTT, facilitando a comunicação entre os diferentes componentes do sistema. No entanto, com o objetivo de simplificar a arquitetura e aumentar a eficiência da comunicação, será desenvolvida neste trabalho uma Interface de Programação de Aplicação com Protocolo de Transferência de Hipertexto, do inglês Application Programming Interface (API) e Hypertext Transfer Protocol (HTTP), que será capaz de se comunicar diretamente com o *broker* MQTT e com um banco de dados *PostgreSQL*. Essa nova solução eliminará a necessidade de usar o Node-RED como intermediário, reduzindo a complexidade do sistema. Além disso, a substituição do *Supabase*, que consumia 7 *GigaBytes* (GB) dos 8GB de *Random Access Memory* (RAM) disponíveis no *Raspberry Pi*, pela nova API resultará em uma significativa redução no uso de RAM, liberando recursos para processamento e permitindo que o sistema suporte um maior número de requisições e acessos simultâneos.

Novas funcionalidades foram adicionadas à aplicação, como ligar e desligar lâmpadas e ar-condicionado, e o monitoramento de tensão, umidade e temperatura. Esses dados serão publicados no *broker* MQTT, coletados pela API e armazenados no banco de dados *PostgreSQL*. Posteriormente, serão consumidos pela aplicação para fornecer informações atualizadas e precisas aos usuários.

Este trabalho parte das hipóteses de que a comunicação com o *broker* MQTT, sem utilizar o protocolo *WebSocket*, aumentará a segurança do sistema, e que a substituição do *Supabase* pelo *PostgreSQL* e uma solução personalizada de API resultará em um melhor uso dos recursos do dispositivo. Os métodos utilizados

incluirão o desenvolvimento e implementação de uma API HTTP utilizando o *framework Express*, a integração dessa API com comunicação direta com o *broker* MQTT e o banco de dados *PostgreSQL*. Além da adição de novas funcionalidades ao aplicativo móvel e a avaliação do desempenho do novo sistema em comparação com o apresentado em (BEZERRA, 2024), focando em métricas de uso de recursos e capacidade de processamento.

Dessa forma, este trabalho visa não apenas melhorar a capacidade do sistema original, mas também expandir suas funcionalidades e aplicações práticas, proporcionando um sistema de controle de acesso mais robusto e eficiente.

1.1 Objetivos

1.1.1 Geral

Expandir as funcionalidades de um "Aplicativo de Controle de Acesso" para integrar outros dispositivos IoT, desenvolver e utilizar uma API HTTP centralizadora das requisições para intercomunicação com um broker MQTT e um banco de dados, e otimizar a eficiência para que seja compatível com minicomputadores que dispõem de recursos reduzidos.

1.1.2 Específicos

- Identificar e selecionar novas aplicações de IoT que possam ser integradas ao sistema de controle de acesso para aprimorar suas funcionalidades.
- Desenvolver e implementar uma API HTTP com Express para facilitar a comunicação entre o broker MQTT, o banco de dados PostgreSQL e o aplicativo.
- Integrar a API HTTP com o broker MQTT e o banco de dados PostgreSQL, garantindo a segurança e a eficiência na troca de informações.
- Expandir as funcionalidades do aplicativo de controle de acesso para incluir novas aplicações de IoT, como monitoramento ambiental e controle de energia.

- Realizar testes de desempenho e eficiência para avaliar o uso de memória RAM e espaço em disco no servidor, visando otimizar a aplicação.
- Implementar medidas de segurança para proteger os dados transmitidos entre o aplicativo, a API, o broker MQTT e o banco de dados.
- Configurar e testar uma rede local para simular condições práticas de uso, avaliando a robustez e a confiabilidade do sistema expandido.
- Avaliar a eficácia da solução proposta em termos de desempenho, segurança, usabilidade e integração das novas ferramentas de IoT.
- Propor melhorias e ajustes na implementação com base nos resultados obtidos durante os testes, visando a otimização contínua do sistema.
- Documentar e apresentar de maneira clara e abrangente os resultados e conclusões alcançadas, destacando as melhorias e a expansão das funcionalidades proporcionadas pelas novas aplicações de IoT e pela API HTTP.

1.2 Organização do Trabalho

Este trabalho de conclusão de curso está estruturado em várias seções que cobrem tanto a teoria quanto o desenvolvimento prático do sistema proposto. A "Fundamentação Teórica" introduz conceitos chave como IoT e suas aplicações em ambientes educacionais, abordando também temas como gerenciamento de temperatura e umidade, iluminação inteligente, e o uso da Raspberry Pi em projetos IoT. Em seguida, explora-se a evolução das linguagens de programação, com foco em desenvolvimento para mobile e web, além de discutir plataformas e ferramentas utilizadas, como Node-RED e Visual Studio Code. A segurança da informação, incluindo criptografia e JWT, e os protocolos de comunicação HTTP e MQTT também são analisados.

Na seção "Desenvolvimento do Trabalho", detalha-se o processo prático de construção do sistema, cobrindo desde o desenvolvimento da arquitetura geral, a modelagem do banco de dados, até o desenvolvimento da API HTTP e a comunicação com MQTT. A segurança do sistema e o design do aplicativo móvel são abordados com foco na experiência do usuário e na integração de diversas plataformas.

A seção "Resultados" apresenta os testes de comunicação entre o aplicativo, a API e o banco de dados, além de explorar a eficácia do protocolo MQTT e a avaliação de desempenho em termos de uso de memória RAM e espaço em disco.

Por fim, a seção “Conclusões e Propostas de Continuidade” revisita os resultados dos testes, oferecendo uma avaliação abrangente do desempenho do sistema, e propõe futuras melhorias com base nas descobertas e tendências tecnológicas emergentes.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Internet das Coisas

A Internet das Coisas, ou *Internet of Things* (IoT) em inglês, refere-se a uma rede interconectada de dispositivos. O termo surgiu da integração de “coisas” à internet. Sua utilização concentra-se na comunicação entre dispositivos e a nuvem, bem como na comunicação entre os próprios dispositivos, como mostrado na Figura 1. Em 2022 havia mais de 27 bilhões de dispositivos conectados (FORBES TECH, 2022). A projeção para 2025 é de aproximadamente 75 bilhões (SHARIF, JUNG, *et al.*, 2022).

Figura 1 – Internet das Coisas.

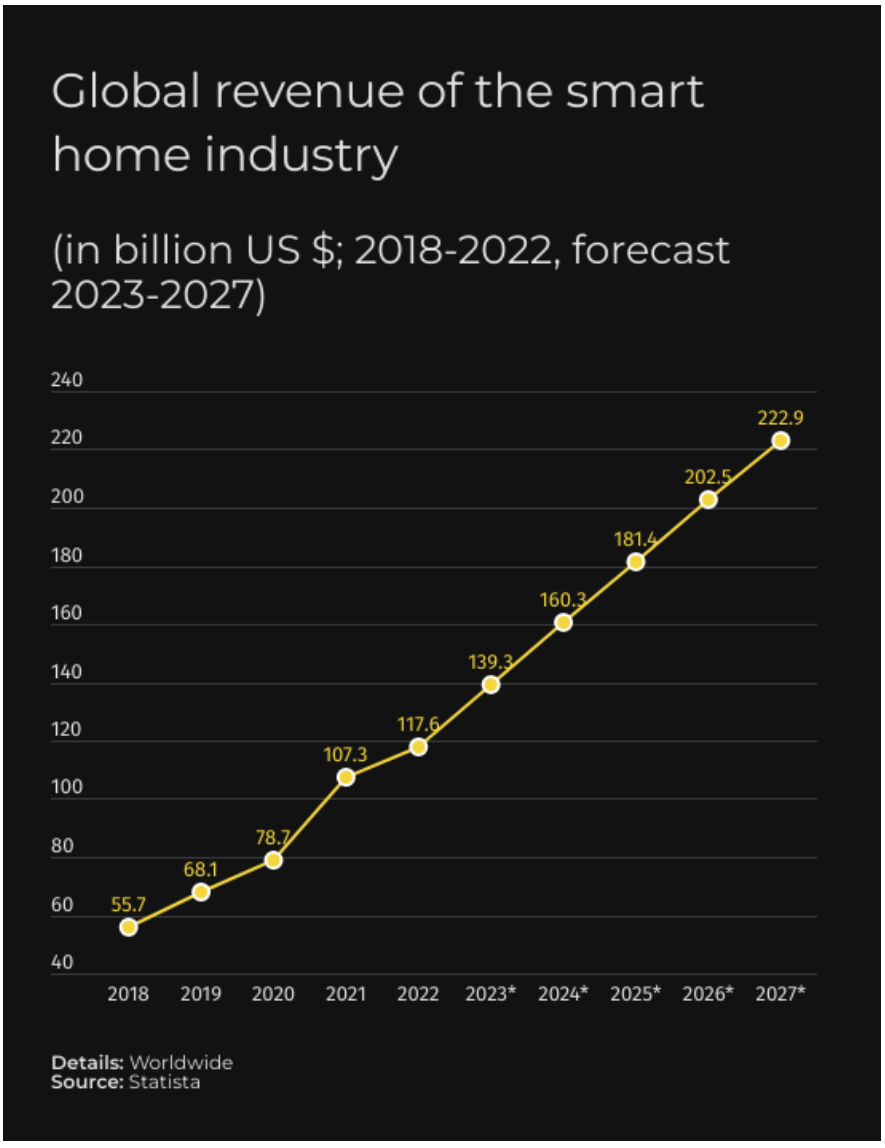
Fonte: (CARVALHO, 2021).

Desde a década de 1990, engenheiros têm integrado sensores e processadores à internet. No entanto, o progresso inicial foi lento devido ao tamanho considerável e volume dos chips. Uma solução inicial envolveu o uso de chips de computador de baixa potência chamados etiquetas de Identificação por Radiofrequência, do inglês

Radio Frequency Identification (RFID), que foram utilizados para rastrear equipamentos caros (AMAZON WEB SERVICES, 2023).

Atualmente, é possível integrar recursos de serviços de voz, como *Alexa*, a dispositivos como interruptores de luz e trancas eletrônicas. Avanços em smartphones e plataformas de *hardware* de código aberto acessíveis permitiram o desenvolvimento de sistemas de automação residencial de baixo custo, utilizando IoT (SHARIF, JUNG, *et al.*, 2022). Na Figura 2 é possível observar a previsão da receita da indústria de casa inteligente em todo mundo.

Figura 2 - Receita da indústria de casa inteligente em todo o mundo.

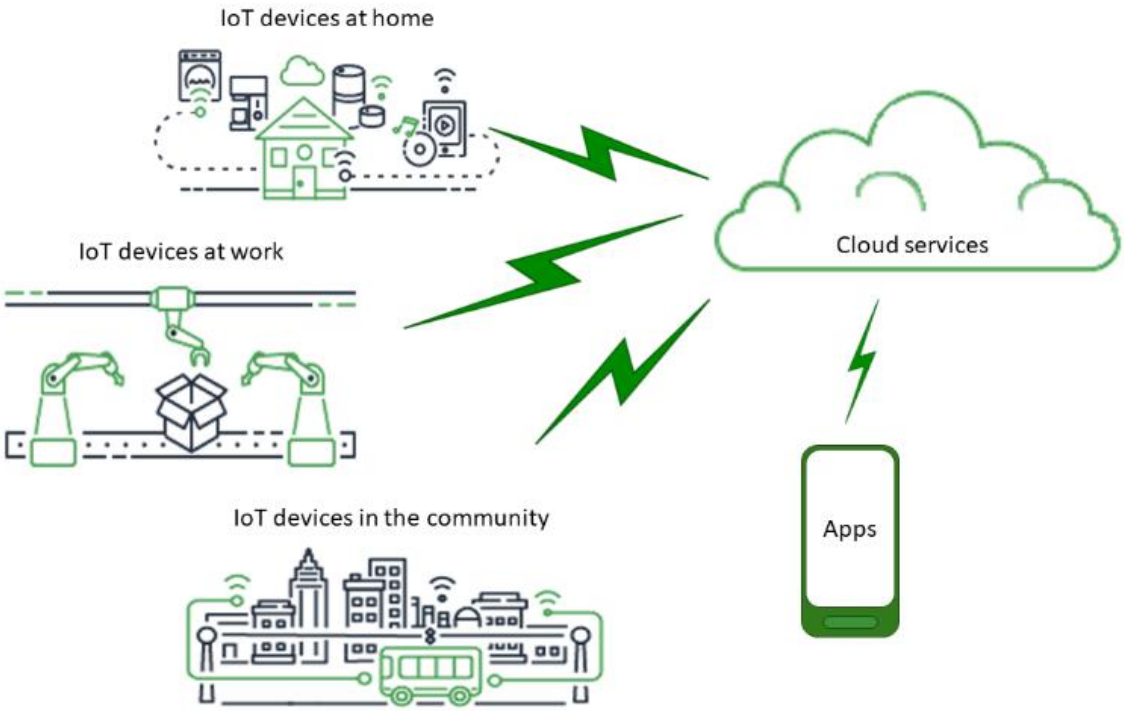


Fonte: (CARVALHO, 2023).

Por exemplo, é viável incorporar funcionalidades como leitura de tensão, temperatura e umidade. A abertura de portas pode ser associada a relés, dispositivos elétricos que realizam modificações súbitas em circuitos elétricos.

Em um projeto, uma aplicação IoT pode ser dividida em três partes distintas. A primeira envolve dispositivos inteligentes que coletam dados ambientais e padrões de uso, transmitindo essas informações para a aplicação. A segunda parte é a própria aplicação IoT, que incorpora serviços e *softwares* para integrar os dados provenientes de diversos dispositivos. Por fim, a terceira parte consiste na interface gráfica do usuário, geralmente implementada por meio de aplicativos móveis ou *web*, responsável por gerenciar usuários, registrar dados e controlar os dispositivos inteligentes (AMAZON WEB SERVICES, 2023). Essas divisões são ilustradas na Figura 3.

Figura 3 – Arquitetura de aplicações IoT.



Fonte: (AMAZON WEB SERVICES, 2023)

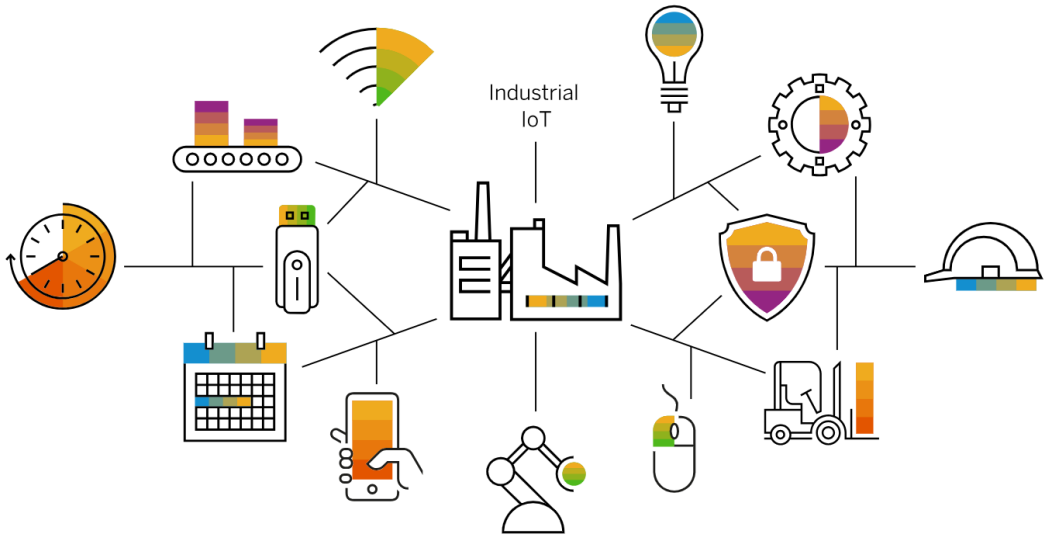
É fundamental compreender que avanços tecnológicos recentes viabilizaram a implementação de IoT. Entre esses avanços, destacam-se a facilidade do acesso à

tecnologia de sensores de baixo custo e baixo consumo de energia, a ampla variedade de protocolos de redes que facilitaram a conectividade local e remota com acesso à nuvem, o constante crescimento de plataformas na nuvem gerenciadas por empresas, o progresso do *machine learning*, a disponibilidade de grandes volumes de dados e capacidade de armazenamento, e o avanço da inteligência artificial conversacional por meio de redes neurais incorporadas em assistentes domésticos (ORACLE, 2024).

Atualmente, outro conceito amplamente empregado é o da *Internet Industrial os Things* (IIoT), que incorpora a IoT em ambientes industriais. Isso é realizado por meio de máquinas conectadas, dispositivos e sensores que se comunicam com a tecnologia em nuvem. Essa abordagem é aplicada para otimizar operações empresariais, uma vez que os dados e informações podem ser monitorados em tempo real, contribuindo para a manutenção preditiva por meio de registros operacionais, dados de desempenho, inteligência artificial e aprendizado de máquina. Os benefícios incluem maior eficiência, gestão de estoque inteligente, entre outros. (SAP, 2024).

A IIoT, enquanto subconjunto da IoT, compartilha as mesmas tecnologias fundamentais, mas seu propósito principal concentra-se na automação e eficiência de um ecossistema organizacional interconectado, diferenciando-se assim do enfoque voltado para usuários isolados, como observado na Figura 4.

Figura 4 – Internet das coisas industrial.



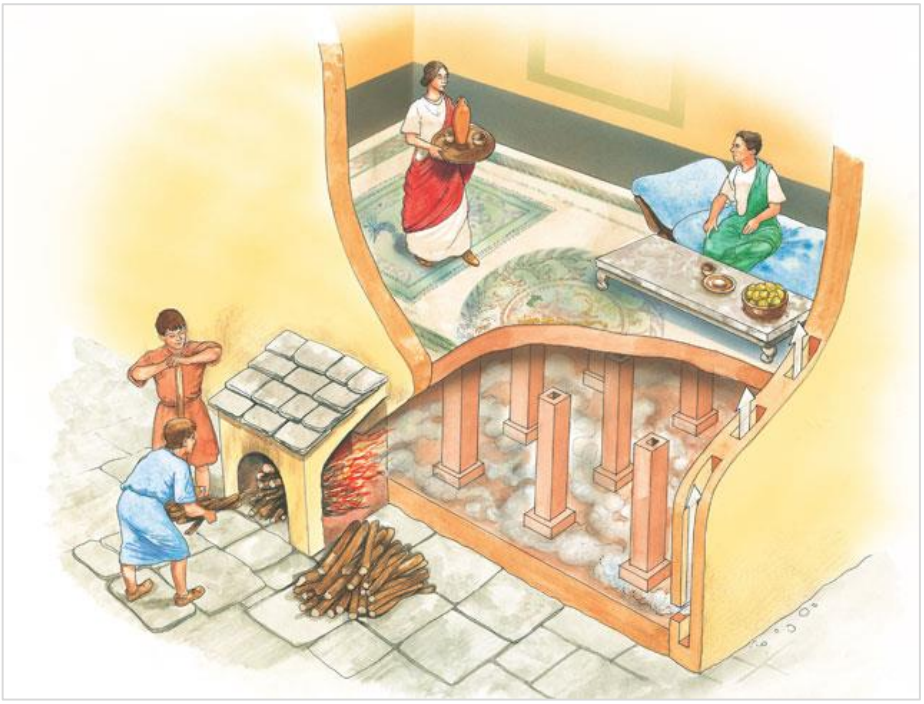
Fonte: (SAP, 2024)

2.1.1 Gerenciamento de Temperatura: uma jornada histórica, desafios atuais e implicações educacionais

O gerenciamento de temperatura, ao longo da história, passou por uma fascinante evolução, moldando não apenas o conforto em ambientes, mas também desempenhando papéis cruciais em avanços médicos, tecnológicos e, mais recentemente, no contexto educacional.

Na Roma Antiga, os romanos já exploravam sistemas de aquecimento chamados *hypocausts*, representado na Figura 5. Esses sistemas consistiam em câmaras subterrâneas que aqueciam o ar sob o piso, proporcionando calor às salas acima. Esse método rudimentar permitia algum controle sobre a temperatura ambiente (WEBARCONDICIONADO, 2022).

Figura 5 - Ilustração do *hypocaust*.



Fonte: (CURRICULUM VISIONS, 2022)

A invenção do termostato durante a Revolução Industrial, no século XIX, forneceu um controle mais preciso da temperatura em ambientes fechados. Desde

então, o controle térmico tornou-se uma peça essencial em diversas esferas da vida moderna (SUGARMAN, 2015).

Hoje, o gerenciamento de temperatura vai além do conforto pessoal. Em residências, escritórios, escolas e outros ambientes, ele desempenha um papel crucial. Nos setores alimentício e industrial, a temperatura é vital para a preservação de alimentos e a qualidade de produtos diversos (SIMONI, 2024).

No contexto educacional, existe uma perspectiva adicional. Estudos indicam que temperaturas inadequadas nas salas de aula podem impactar o desempenho cognitivo dos alunos. A falta de climatização adequada pode influenciar negativamente a concentração e o aprendizado, ressaltando a importância do gerenciamento de temperatura não apenas no conforto, mas também na eficácia do ambiente educacional (TORRES e OLIVEIRA, 2016).

É interessante perceber que, historicamente, a busca por temperaturas ideais acompanha a evolução humana. Desde os *hypocausts* dos romanos até os modernos sistemas de Aquecimento, Ventilação e Ar-Condicionado (AVAC), a humanidade sempre buscou um controle mais refinado sobre seu ambiente térmico. Em busca dessa “temperatura ideal” para ambiente, a Resolução 09 da ANVISA, no Brasil, orienta que a faixa recomendável de temperatura para ambientes internos no verão, deve variar de 23°C a 26°C. Para o inverno, a faixa recomendável deve variar de 20°C a 22°C. Sendo a faixa máxima variando de 26,5°C a 27°C, (ANVISA, 2003).

Portanto, a aplicação de IoT para monitorar e regular a temperatura de ambientes internos, como salas de aula, é fundamental. Essa tecnologia permite atender às recomendações específicas, como as da Resolução 09 da ANVISA no Brasil, que define faixas ideais de temperatura para diferentes estações do ano, assegurando condições ideais de forma contínua e precisa.

2.1.2 Iluminação Inteligente e IoT em Ambientes Educacionais: Vantagens e Aplicações

A utilização da iluminação inteligente integrada à IoT em ambientes educacionais traz inúmeras vantagens, transformando o cotidiano acadêmico. A praticidade é

evidente, pois a iluminação pode ser controlada por aplicativos ou comandos de voz, permitindo ajustes rápidos e precisos conforme a necessidade. Por exemplo, é possível programar a intensidade da luz para se adequar a diferentes atividades, como palestras, estudo individual ou eventos, otimizando a experiência dos alunos e professores sem a necessidade de intervenções manuais constantes (Boa iluminação nas escolas influencia no desempenho dos alunos, 2024).

A economia de energia é outra vantagem significativa. Sistemas de iluminação inteligente permitem o gerenciamento eficiente das luzes, evitando desperdícios e reduzindo custos operacionais. Sensores de movimento e horários programados podem garantir que as luzes sejam utilizadas apenas quando necessário, contribuindo para uma gestão sustentável dos recursos energéticos da instituição. Além disso, a possibilidade de ajustar a iluminação conforme a presença de luz natural ajuda a maximizar o uso de recursos gratuitos, melhorando ainda mais a eficiência energética (Iluminação inteligente: o que compõe o sistema?, 2022).

Por fim, a iluminação inteligente aumenta a segurança e o conforto nos ambientes educacionais. A programação para acender e apagar as luzes em horários específicos pode simular a presença de pessoas, inibindo ações criminosas em períodos de inatividade. A capacidade de adaptar a tonalidade e a intensidade das luzes conforme a necessidade cria um ambiente mais confortável e adequado para diferentes atividades, melhorando a concentração e o bem-estar dos alunos. Em resumo, a integração da iluminação inteligente e IoT em escolas e universidades representa um avanço significativo em termos de eficiência, segurança e qualidade do ambiente de aprendizado.

2.1.3 Controle de Umidade: História, Saúde e Conforto Integrados com IoT

A história do controle de umidade remonta a séculos atrás, quando o foco inicial estava principalmente na regulação da temperatura para o conforto humano. No entanto, foi somente no início do século XX que a umidade se tornou uma preocupação significativa. *Willis Carrier*, renomado engenheiro, enfrentou o desafio de controlar a umidade em ambientes de impressão da *Sacket and Wilhelms Lithographing*, durante os quentes verões, quando altos níveis de umidade

prejudicavam a qualidade das impressões coloridas. A solução inovadora proposta por *Carrier*, um sistema que resfriava e desumidificava o ar através de serpentinas com água fria, foi o precursor dos modernos sistemas de ar-condicionado, destacando a importância do controle preciso da umidade desde então (MARIANI, 2019).

Os métodos para parametrizar a umidade do ar envolvem conceitos fundamentais como umidade absoluta e umidade relativa. A umidade absoluta mede a quantidade de vapor d'água presente em relação à quantidade de ar seco, enquanto a umidade relativa expressa essa relação em termos percentuais, comparando a quantidade atual de vapor d'água com a máxima que o ar pode conter a uma determinada temperatura (PENA, 2024). A psicometria, ramo da termodinâmica que estuda o comportamento do ar úmido, utiliza equações e diagramas psicrométricos para calcular essas grandezas, permitindo uma compreensão detalhada das condições de umidade em ambientes controlados (MARIANI, 2019).

No contexto atual, o controle eficaz da umidade não se limita apenas ao conforto pessoal, mas abrange a preservação de materiais e o desempenho de equipamentos. Em ambientes industriais e de tecnologia da informação, por exemplo, variações extremas de umidade podem comprometer a funcionalidade de equipamentos eletrônicos sensíveis. Para mitigar esses riscos, é essencial implementar sistemas de monitoramento e controle de umidade. A tecnologia IoT desempenha um papel crucial nesse sentido, permitindo o monitoramento em tempo real e ajustes automáticos com base nas condições ambientais detectadas por sensores distribuídos estrategicamente.

Normas e recomendações estabelecem diretrizes para manter níveis ideais de umidade em diferentes tipos de ambientes. A Organização Mundial da Saúde (OMS), por exemplo, recomenda manter a umidade relativa entre 30% e 60% para promover um ambiente saudável, minimizando riscos de desconforto respiratório e proliferação de microrganismos prejudiciais à saúde. Essas diretrizes são fundamentais não apenas para o conforto humano, mas também para a conservação de materiais, a eficiência operacional de equipamentos e a compreensão detalhada das condições de umidade em ambientes controlados (MARIANI, 2019).

2.1.4 Raspberry Pi aplicado em IoT

A *Raspberry Pi* tem sido a escolha preferida de entusiastas, educadores e engenheiros desde 2012, devido à sua notável performance a um preço acessível. Desenvolvida pela *Raspberry Pi Foundation*, esses computadores modulares de placa única possuem processadores baseados na arquitetura *Acorn RISC Machine* (ARM), tornando-os ideais para projetos de hardware inteligentes (FUNDAÇÃO RASPBERRY PI).

A *Raspberry Pi* tornou-se extremamente popular em projetos de IoT devido à sua acessibilidade, versatilidade e poder de computação. Com uma variedade de modelos disponíveis, desde o *Raspberry Pi Zero* até o *Raspberry Pi 5*, os desenvolvedores podem escolher o dispositivo que melhor se adapta às necessidades de seus projetos a um custo acessível. Além disso, esse minicomputador completo é capaz de executar uma variedade de sistemas operacionais, como *Raspbian,* agora *Raspberry Pi OS*, *Ubuntu* e *Windows 10 IoT Core*, tornando-o flexível para diferentes aplicações de IoT (DEEPSEA DEVELOPMENTS, 2023).

A *Raspberry Pi* também é reconhecida por sua compatibilidade com uma extensa variedade de sensores, atuadores e acessórios IoT. Essa versatilidade é crucial para o sucesso de projetos relacionados à Internet das Coisas, proporcionando aos desenvolvedores a escolha entre uma ampla gama de componentes para atender às necessidades específicas de seus projetos. Essa flexibilidade simplifica significativamente o processo de integração de hardware, reduzindo tanto o tempo quanto os custos associados ao desenvolvimento de soluções de IoT. Ao invés de lidar com componentes que exigem adaptações complexas ou interfaces personalizadas, os usuários da *Raspberry Pi* podem aproveitar os pinos de entrada/saída de uso geral (GPIO), que permitem a interação com uma ampla variedade de periféricos, sensores e outros componentes disponíveis no mercado, muitos dos quais são *plug-and-play* com a placa (FASTER CAPITAL, 2023).

Além disso, a comunidade ativa e engajada da *Raspberry Pi* contribui para essa compatibilidade, oferecendo suporte e compartilhando informações sobre como integrar diferentes componentes em projetos de IoT. Criando um ambiente propício para a inovação, permitindo que desenvolvedores de todos os níveis de habilidade

criem soluções de IoT personalizadas e eficientes (DEEPSEA DEVELOPMENTS, 2023).

A *Raspberry Pi 4B*, utilizada neste trabalho, representa um avanço significativo em termos de velocidade e desempenho. Com *Ethernet Gigabit*, conectividade sem fio e *Bluetooth* integrados, além de portas *Universal Serial Bus* (USB) 3.0 para transferência de dados mais rápida, está disponível em diferentes variantes de RAM, memória volátil, incluindo 1GB, 2GB, 4GB ou 8GB, para atender as necessidades específicas de cada projeto (FUNDAÇÃO RASPBERRY PI). Sua capacidade de processamento considerável equipado com unidade central de processamento (*CPU) ARM Cortex-A72 quad-core* e 8 GB de RAM permitem lidar com tarefas complexas e processamento de dados em tempo real ideal para o gerenciamento e monitoramento de temperatura e tensão (FASTER CAPITAL, 2023).

O sistema operacional normalmente utilizado na *Raspberry Pi*, anteriormente conhecido como *Raspbian* e agora chamado de *Raspberry Pi OS*, é baseado no *Debian* (RASPBERRY PI FOUNDATION, 2024). O *Debian* é uma distribuição *Linux* renomada por sua estabilidade, vasto repositório de software e compromisso com o *software* livre (DEBIAN, 2023,). No entanto, o *Raspberry Pi O*S muitas vezes apresenta versões mais antigas de pacotes de instalação de programas como *Node.js*, *npm*, *Node-RED*, entre outros.

Além do *Raspberry Pi OS*, existem outros sistemas operacionais compatíveis com a *Raspberry Pi*, como Fedora e Ubuntu (SANTANA, 2023). Para este trabalho, optou-se pelo *Raspberry Pi OS* devido à sua otimização específica para o hardware da *Raspberry Pi*, garantindo um desempenho mais eficiente e uma melhor compatibilidade com os periféricos da plataforma. Além disso, o *Raspberry Pi OS* oferece uma extensa documentação e uma comunidade ativa de suporte, o que facilita a resolução de problemas e o desenvolvimento de projetos. Essa escolha também é coerente com a continuidade do trabalho desenvolvido por (BEZERRA, 2024), que utilizou o mesmo sistema operacional.

2.2 A História e a Evolução do Código e da Programação

Antes de explorar as linguagens de programação, é crucial compreender o conceito de código. Código é um sistema de sinais utilizado para transmitir mensagens, onde esses sinais podem ser representados por letras ou números. A criação de códigos serve a diversos propósitos, desde comunicações secretas entre amigos até a transmissão de mensagens em contextos de guerra. Um exemplo conhecido é o código *Morse*, criado por *Samuel Morse* e *Alfred Vail* na década de 1830 e 1840, que utiliza combinações de pulsos curtos e longos, representados na escrita por pontos e traços (PETZOLD, 1999).

Ao longo da história documentada, várias máquinas foram criadas para facilitar cálculos matemáticos. Na Europa, surgiram tábuas de contagem, enquanto no Oriente Médio, o ábaco era comum. A aversão à multiplicação e divisão levou *John Napier* a inventar logaritmos para simplificar essas operações. Uma invenção notável foi o tear automatizado de *Joseph Marie Jacquard*, que usava cartões perfurados para controlar padrões em tecidos (PETZOLD, 1999).

No século XVIII, um "computador" era uma pessoa que calculava números para alugar, especialmente na criação de tabelas matemáticas. Motivado pela necessidade de eliminar erros nessas tabelas, *Charles Babbage* propôs a Máquina Diferencial por volta de 1820, uma máquina de adição mecânica que representava números decimais usando rodas dentadas. Apesar de alguns modelos iniciais e apoio do governo britânico, a Máquina Diferencial nunca foi concluída. *Babbage* então concebeu a Máquina Analítica, mais avançada, com uma memória e unidade de cálculo, projetada para ser programada usando cartões adaptados do tear *Jacquard*. Essas inovações, embora não realizadas em sua época, foram redescobertas no século XX como precursores da computação moderna.

Augusta Ada Byron, Condessa de *Lovelace*, que trabalhou com *Babbage*, é considerada a primeira programadora, dizia que a Máquina Analítica "tece padrões algébricos assim como o tear *Jacquard* tece flores e folhas." Essa conexão entre a máquina de *Babbage* e o tear reflete uma visão precursora da automação e programação na história da computação. *Ada* contribuiu significativamente para a programação, ela percebeu que essa máquina poderia ir além dos cálculos numéricos

e imaginou a criação de instruções para realizar diversos tipos de trabalho. Também elaborou o primeiro algoritmo destinado a ser processado por uma máquina, estabelecendo as bases para a programação de computadores (HICKS, 2017).

Alan Turing, considerado o pai da ciência da computação, desempenhou um papel crucial no desenvolvimento das linguagens de programação, sendo reconhecido por suas contribuições à teoria da computação, concepção de máquinas universais e quebra do código Enigma durante a Segunda Guerra Mundial. Esses marcos históricos não apenas impulsionaram a tecnologia da época, mas também semearam a compreensão e expansão contínua das linguagens de programação, fundamentais para o avanço na computação (ISAACSON, 2014).

Com o constante avanço de *hardwares* e *softwares*, é possível observar uma diversidade crescente de linguagens de programação. No âmbito tecnológico, essas linguagens são geralmente classificadas como "alto nível" e "baixo nível". Quanto mais baixo o nível, mais próxima a linguagem de programação está da linguagem de máquina. Exemplos de linguagens de baixo nível incluem *Assembly* e *C*, ao passo que linguagens de alto nível abrangem *JavaScript*, *Java* e *Python*.

No contexto da programação, é comum empregar editores de código fonte e Ambientes de Desenvolvimento Integrado, do inglês *Integrated Development Environment* (IDE). Essas ferramentas são essenciais para desenvolver códigos de maneira eficiente, oferecendo recursos abrangentes de edição, compilação, teste e empacotamento de software (AMAZON WEB SERVICES, 2023). Nos editores de código fonte, é comum encontrar suporte para o sistema de controle de versão *Git*, destaque de sintaxe e complementação inteligente do código, entre outros recursos facilitadores (MICROSOFT, 2024).

2.2.1 Linguagem de Programação para Aplicações Mobile e web

Dentre as linguagens mais empregadas para aplicações mobile, destacam-se *Java* e *Kotlin*, comumente utilizadas na programação de aplicativos *Android*. O *Kotlin*, reconhecido oficialmente pela *Google*, tem ganhado crescente popularidade. Para

dispositivos iOS, *Swift* e *Objective-C* são as escolhas predominantes (COOPERSYSTEM, 2023).

Para o ambiente *web* o *JavaScript* é uma linguagem leve, interpretada e baseada em objetos, é utilizada principalmente em navegadores *web*, mas também pode ser encontrada em outros ambientes como *Node.js, Apache CouchDB* e *Adobe Acrobat*. As funções dessa linguagem são tratadas como objetos de primeira classe onde estas são atribuídas a variáveis e retornadas por outras funções. Também apresenta vários paradigmas de programação como imperativo, comandos para controle de fluxo e suporte à programação funcional, e orientação a objetos que utiliza protótipos para herança (MOZILLA CORPORATION, 2024).

No âmbito do desenvolvimento *web*, a combinação *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e *JavaScript* desempenha papéis distintos na estruturação, estilização e interatividade do lado do cliente. *JavaScript*, quando utilizado com *Node.js* no lado do servidor, viabiliza aplicações em tempo real e escaláveis (FLANAGAN, 2020).

Python é outra linguagem amplamente utilizada no desenvolvimento *web*, oferece *framework*s como *Django* e *Flask*. *Django* é conhecido por seu foco na simplicidade e na reutilização de código, facilitando o desenvolvimento rápido de aplicações *web* robustas e seguras (HOLOVATY e KAPLAN-MOSS, 2009). *Flask*, por outro lado, é um *microframework* que proporciona maior flexibilidade e controle sobre os componentes da aplicação, sendo ideal para projetos menores ou que requerem uma abordagem mais personalizada (GRINBERG, 2018).

Hypertext Preprocessor, originalmente chamado de *Personal Home Page* (PHP), continua a ser uma das linguagens mais populares para o desenvolvimento *web*, especialmente para a criação de sites dinâmicos e aplicações baseadas em servidor. PHP é amplamente utilizado em conjunto com sistemas de gerenciamento de conteúdo como *WordPress*, *Joomla* e *Drupal*, além de ser a base de grandes plataformas de e-commerce e redes sociais (WELLING e THOMSON, 2016).

Para utilizar o *JavaScript* no desenvolvimento *mobile*, o *Facebook* criou, em 2015, o *framework React Native*. Um *framework* é um conjunto de ferramentas e bibliotecas que fornece uma estrutura padrão para o desenvolvimento de *software*,

facilitando o trabalho dos desenvolvedores. O *React Native* é baseado em *JavaScript* e é capaz de renderizar aplicativos móveis de forma nativa, de uma maneira que o aplicativo se comporta e tem a aparência de um aplicativo escrito na linguagem de programação específica da plataforma, como *Swift* para *iOS* e *Java*/*Kotlin* para *Android*. Por ser de código aberto, o *React Native* permite que várias empresas utilizem e contribuam para o seu desenvolvimento (CUNHA, 2023).

Assim, a escolha das tecnologias, tanto para desenvolvimento *web* quanto *mobile*, depende de fatores como a necessidade de escalabilidade, a complexidade da aplicação e a expertise da equipe de desenvolvimento. Com o avanço contínuo das tecnologias, os desenvolvedores têm à disposição uma variedade de ferramentas poderosas para criar soluções inovadoras e eficientes.

No presente trabalho, optou-se por utilizar *React Native* e *JavaScript* para dar continuidade ao trabalho desenvolvido por (BEZERRA, 2024), proporcionando também, flexibilidade e potencial de crescimento tanto para aplicações *web* quanto *mobile*. Essa escolha foi motivada não apenas pela compatibilidade com ambas as plataformas, *iOS* e *Android*, mas também pelo expressivo crescimento do *JavaScript* no cenário atual.

2.2.2 Plataformas de Desenvolvimento

Plataformas de desenvolvimento são *softwares* projetados para otimizar e agilizar o processo de criação de aplicativos, tanto para ambientes móveis quanto para *web*. Elas simplificam o desenvolvimento por meio de ferramentas, bibliotecas e *framework*s, que consistem em estruturas pré-escritas reutilizáveis. Essas plataformas também oferecem ambientes de execução, suporte a múltiplas plataformas, comunidade, além de documentação e tutoriais, todos essenciais para facilitar o desenvolvimento (SANTANA, 2024).

Dentre as diversas plataformas disponíveis, destacam-se a *Expo*, *Flutter*, *Ignite CLI*, entre outras, cada uma oferecendo suas características específicas para atender as necessidades variadas dos desenvolvedores. A *Expo*, por exemplo, é uma plataforma baseada no *framework React Native*, que permite a criação de aplicativos

nativos para iOS e *Android* usando *JavaScript* e *React*. Ela proporciona um ambiente de desenvolvimento rápido e eficiente, com ferramentas integradas para depuração, teste e publicação de aplicativos. O *Expo Go* permite que desenvolvedores testem suas aplicações em dispositivos móveis em tempo real, sem a necessidade de recompilação (EXPO, 2024).

Flutter, outra plataforma popular, é um *framework* de código aberto desenvolvido pelo *Google* para a criação de aplicativos nativos para *Android* e iOS, bem como para a *web* e *desktops*. Ele utiliza a linguagem *Dart* e se destaca por seu desempenho e por permitir a criação de interfaces de usuário ricas e responsivas. *Flutter* inclui um rico conjunto de *widgets* pré-construídos, que facilitam a criação de interfaces personalizadas e a integração com APIs nativas (GOOGLE , 2024).

Ignite CLI é uma ferramenta de desenvolvimento para *React Native* que oferece uma estrutura robusta para iniciar novos projetos. Desenvolvida pela *Infinite Red*, o *Ignite CLI* fornece uma configuração inicial otimizada, com várias opções de *plugins* e *templates* que agilizam o desenvolvimento. É particularmente útil para desenvolvedores que buscam uma abordagem estruturada e escalável para projetos *React Native* (INFINITE RED , 2024).

Neste trabalho, a escolha de Expo foi devido ao início prévio do desenvolvimento do aplicativo, aproveitando as funcionalidades e a praticidade oferecidas pela plataforma: a compatibilidade com iOS e Android e o ambiente de execução integrado proporcionado pelo Expo Go que facilitarão o progresso contínuo do aplicativo.

2.2.3 Ferramentas de Desenvolvimento e Monitoramento

Ferramentas de desenvolvimento e monitoramento são essenciais para testes, depuração, monitoramento e manutenção de aplicações. Elas oferecem funcionalidades específicas que ajudam a garantir a qualidade e a eficiência dos processos de desenvolvimento e operação.

2.2.3.1 Node-RED

Node-RED é uma ferramenta de desenvolvimento baseada em fluxo que permite a criação de aplicações de IoT por meio da programação visual. Com o *Node-RED*, é possível integrar dispositivos, APIs e serviços online de maneira simples e intuitiva. (NODE-RED, 2024).

Ferramentas semelhantes ao *Node-RED* incluem:

- *Microsoft Power Automate*: Uma plataforma de automação de processos que permite a criação de fluxos de trabalho automatizados entre aplicativos e serviços para sincronizar arquivos, obter notificações, coletar dados, entre outras funcionalidades (MICROSOFT, 2024).
- *Zapier*: Um serviço de automação que conecta diferentes aplicativos e serviços, permitindo a automação de tarefas repetitivas sem a necessidade de programação (ZAPIER, 2024).

2.2.3.2 Wireshark

Wireshark é uma ferramenta de análise de protocolo de rede que permite capturar e examinar os pacotes de dados trafegando em uma rede em tempo real. Utilizado amplamente para solução de problemas e análise de desempenho, o *Wireshark* oferece uma visão detalhada dos dados de comunicação, facilitando a identificação de problemas e a otimização da rede (WIRESHARK, 2024).

Ferramentas semelhantes ao *Wireshark* incluem:

- *tcpdump*: Uma ferramenta de linha de comando que permite capturar e analisar pacotes de rede. É amplamente utilizada para a solução de problemas de rede e análise de tráfego (TCPDUMP, 2024).
- *Microsoft Network Monitor*: Uma ferramenta de análise de rede que permite a captura e visualização de pacotes de rede em uma interface gráfica (MICROSOFT, 2024).

2.2.3.3 Visual Studio Code

Visual Studio Code é um editor de código aberto que roda no *Windows*, *Linux* e *macOS*. Ele traz recursos nativos como realce de sintaxe, sugestões para preenchimento automático e comandos *Git* integrados. *Visual Studio Code* é amplamente reconhecido por sua extensibilidade, permitindo que desenvolvedores adicionem funcionalidades adicionais por meio de extensões (MICROSOFT, 2024).

Ferramentas semelhantes ao *Visual Studio Code* incluem:

- *Atom*: Um editor de texto de código aberto, desenvolvido pelo *GitHub*, que oferece uma interface personalizável e uma vasta gama de pacotes e temas (GITHUB, 2024).
- *Sublime Text*: Um editor de texto sofisticado para código, marcação e prosa, conhecido por sua velocidade e interface de usuário minimalista (SUBLIME TEXT, 2024).

O *Node-RED* foi selecionado pela sua interface intuitiva e capacidade de integrar rapidamente diferentes dispositivos e serviços, essencial para a criação do dashboard de testes. O *Wireshark* foi escolhido pela sua eficácia em monitorar a comunicação de rede, permitindo uma análise detalhada do tráfego de dados durante os testes. Por fim, o *Visual Studio Code* foi a ferramenta de edição de código escolhida devido a sua versatilidade, extensibilidade e recursos integrados, que facilitam o desenvolvimento de código eficiente e organizado. A escolha dessas ferramentas também foi guiada pela afinidade e habilidade desenvolvidas ao longo da graduação, graças ao uso contínuo das mesmas.

2.3 Introdução ao Banco de Dados: Um Olhar Histórico e Conceitual

Os bancos de dados são amplamente adotados nos negócios contemporâneos, servindo como repositórios cruciais para dados em várias organizações. Grandes empresas, incluindo gigantes como *Amazon* e *Google*, fazem uso de bancos de dados para fornecer informações quando requisitadas. No ambiente acadêmico, esses bancos de dados desempenham um papel vital em pesquisas que abrangem áreas

como astronomia, bioquímica, medicina, entre outras (GARCIA-MOLINA, ULLMAN e WIDOM, 2009).

Para simplificar o uso e a administração segura desses dados, foram desenvolvidos softwares especializados e altamente complexos conhecidos como Sistemas de Gerenciamento de Banco de Dados (SGBD). Esses sistemas são capazes de realizar operações fundamentais, como inserção, consulta, atualização e exclusão de dados. Para otimizar a eficiência desses SGBDs, foram criadas ferramentas que oferecem suporte ao seu gerenciamento. Exemplos proeminentes incluem o *pgAdmin*, *DBeaver*, *MySQL Workbench*, entre outros. Essas ferramentas facilitam a interação com os bancos de dados, tornando o processo de manipulação de dados mais intuitivo e eficaz. Elas fornecem interfaces gráficas e funcionalidades avançadas para administrar, desenvolver e monitorar bancos de dados de maneira eficiente (CORONEL e MORRIS, 2015).

Neste trabalho, será empregado o *DBeaver*, uma ferramenta versátil de gerenciamento de sistemas de banco de dados com código aberto, oferecendo suporte para até 80 bancos de dados. Esta ferramenta tem a capacidade de trabalhar com extensões como *Git* e *Excel*. Os requisitos mínimos do sistema incluem 1 GB de RAM e 200 *Megabytes* (MB) de espaço em disco (DBEAVER COMMUNITY EDITION). Esses requisitos demonstram uma baixa demanda de recursos de armazenamento e processamento.

A definição de bancos de dados é uma coleção de informações que perdura ao longo do tempo. Para discutir sobre bancos de dados, é essencial compreender o funcionamento dos SGBDs. Esses sistemas possibilitam a criação de usuários e bancos de dados, o uso de uma linguagem específica, a elaboração de consultas, a modificação de dados, o controle de acesso de usuários (isolamento), além de assegurar que as operações sejam realizadas com sucesso como uma única operação indivisível (atomicidade) (CONNOLLY e BEGG, 2014)

Os primeiros bancos de dados surgiram na década de 1960 como uma resposta aos sistemas de arquivos que armazenavam dados por períodos de tempo muito longos. Esses sistemas de arquivos não ofereciam garantias contra perda ou vazamento de dados, e tampouco proporcionavam informações precisas sobre a

localização exata dos arquivos. As características mencionadas anteriormente eram praticamente inexistentes nesses sistemas (STONEBRAKER, 2012).

2.3.1 Sistemas de Bancos De Dados Relacionais

Em 1970, *Ted Codd* propôs uma reorganização dos sistemas de bancos de dados, sugerindo que eles deveriam ser estruturados visualmente por meio de tabelas, agora conhecidas como entidades. Essas tabelas estabeleciam relações entre si, visando proporcionar consultas rápidas e simplificar a preocupação do programador com a estrutura de armazenamento (GARCIA-MOLINA, ULLMAN e WIDOM, 2009).

Os sistemas de bancos de dados relacionais (SGBDR) são baseados no modelo relacional de dados proposto por *Codd*, onde os dados são organizados em tabelas, ou relações, que podem ser manipuladas usando operações matemáticas. As tabelas são compostas por linhas ou tuplas e colunas ou atributos, onde cada linha representa um registro único e cada coluna, uma propriedade do registro. As tabelas são interligadas por meio de chaves primárias e estrangeiras, permitindo que dados em diferentes tabelas sejam combinados e consultados de maneira eficiente (CODD, 1990).

Uma das principais características dos SGBDR é o uso da linguagem de consulta estruturada, conhecida como *Structured Query Language* (SQL). O SQL permite que os usuários interajam com o banco de dados para realizar diversas operações, como inserção, atualização, exclusão e recuperação de dados. A SQL é uma linguagem declarativa, onde o usuário especifica o que deseja obter sem precisar detalhar o procedimento para alcançar esse resultado, tornando o processo de manipulação de dados mais intuitivo e eficiente (MELTON e SIMON, 1993).

Os SGBDR oferecem várias vantagens, como:

- Integridade dos Dados: Garantem a precisão e consistência dos dados por meio de restrições e regras de integridade.
- Segurança: Proporcionam mecanismos de controle de acesso para proteger os dados contra acessos não autorizados.

- Transações: Suportam transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), que garantem que todas as operações de uma transação sejam concluídas com sucesso ou revertidas em caso de falha, mantendo a integridade dos dados.
- Escalabilidade: Podem ser dimensionados para lidar com grandes volumes de dados e acessos simultâneos.

Atualmente, existem diversos sistemas de gerenciamento de bancos de dados relacionais amplamente utilizados, cada um com suas características e funcionalidades específicas (KROENKE e AUER, 2013). Alguns dos mais populares incluem:

- *PostgreSQL*: Um SGBDR avançado e de código aberto, conhecido por sua robustez, extensibilidade e conformidade com padrões SQL.
- *MariaDB*: Um *fork* do *MySQL*, também de código aberto, que oferece desempenho e escalabilidade aprimorados, além de compatibilidade com o *MySQL*.
- *MySQL*: Um dos SGBDR mais populares e amplamente utilizado no desenvolvimento de aplicações *web*, conhecido por sua facilidade de uso e confiabilidade.

No contexto dos SGBDR, as chaves primárias, do inglês *Primary Keys* (PK), e chaves estrangeiras, do inglês *Foreign Keys* (FK), desempenham um papel crucial na organização, integridade e eficiência do armazenamento e recuperação de dados.

A chave primária é um campo em uma tabela de banco de dados relacional que identifica de forma única cada registro dessa tabela. As chaves primárias são essenciais para a criação de índices, o que melhora a velocidade e a eficiência das operações de consulta (HERNANDEZ, 2013). Suas principais características incluem:

- Unicidade: Cada valor da chave primária deve ser único dentro da tabela. Isso assegura que cada registro possa ser identificado de forma distinta.

- Não Nulo: Os campos definidos como chave primária não podem conter valores nulos. Isto é essencial para garantir que cada registro seja identificável.
- Imutabilidade: Idealmente, os valores das chaves primárias não devem mudar ao longo do tempo. Isso porque a mudança de uma chave primária pode afetar a integridade referencial do banco de dados.

Por outro lado, a chave estrangeira é um campo em uma tabela que cria um vínculo entre essa tabela e a chave primária de outra tabela. A chave estrangeira assegura a integridade referencial, garantindo que os dados sejam consistentes e que as relações entre tabelas sejam válidas (HERNANDEZ, 2013). As principais características de uma chave estrangeira incluem:

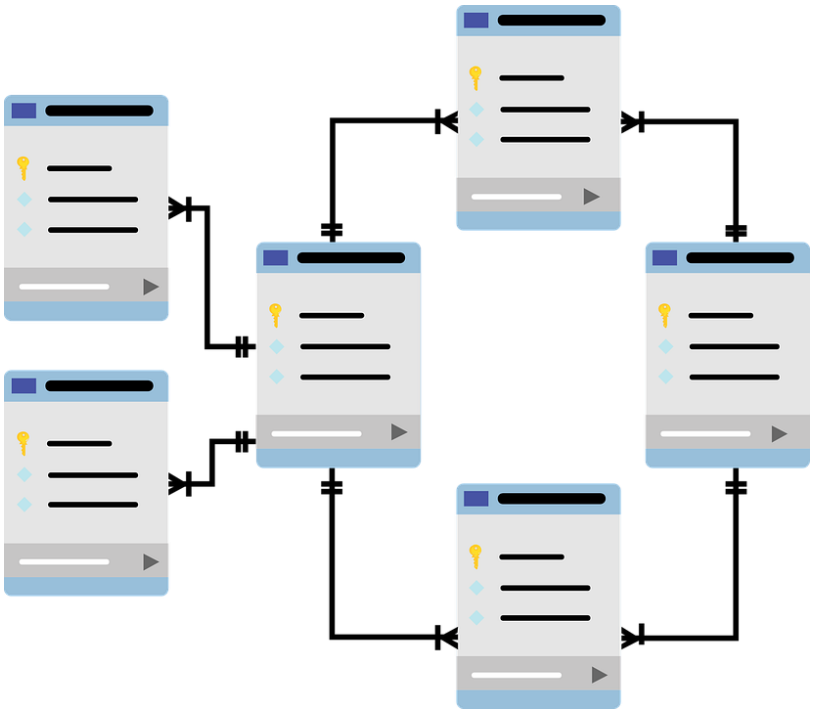
- Relacionamento: A chave estrangeira estabelece um relacionamento entre duas tabelas. Ela faz referência à chave primária de outra tabela, indicando que um valor em uma tabela deve corresponder a um valor na chave primária da outra tabela.
- Integridade Referencial: A utilização de chaves estrangeiras assegura que as relações entre tabelas permaneçam consistentes. Por exemplo, uma chave estrangeira em uma tabela de pedidos que se refere a uma chave primária em uma tabela de clientes garante que cada pedido esteja associado a um cliente válido.
- Propagação de Exclusão e Atualização: Em muitos sistemas de banco de dados, é possível configurar regras de exclusão e atualização em cascata. Isso significa que se um registro referenciado por uma chave estrangeira for excluído ou atualizado, as mudanças serão automaticamente refletidas nas tabelas relacionadas.

A utilização de chaves primárias e estrangeiras é crucial para a normalização dos dados, um processo que organiza os dados em tabelas de forma a reduzir a redundância e melhorar a integridade dos dados. Além disso, essas chaves são

fundamentais para a implementação de restrições de integridade, assegurando que os dados inseridos no banco de dados sejam válidos e coerentes.

Em resumo, as chaves primárias e estrangeiras são componentes essenciais de qualquer banco de dados relacional bem projetado. Elas garantem a unicidade dos registros, mantêm a integridade referencial e facilitam a construção de relacionamentos entre diferentes conjuntos de dados, assegurando a coerência e a integridade dos dados armazenados. A Figura 6 apresenta uma representação da estrutura de um sistema de banco de dados relacional.

Figura 6 - Representação da estrutura de um sistema de banco de dados relacional.



Fonte: (FONSECA, 2021).

2.3.2 Sistemas De Bancos De Dados Não Relacionais

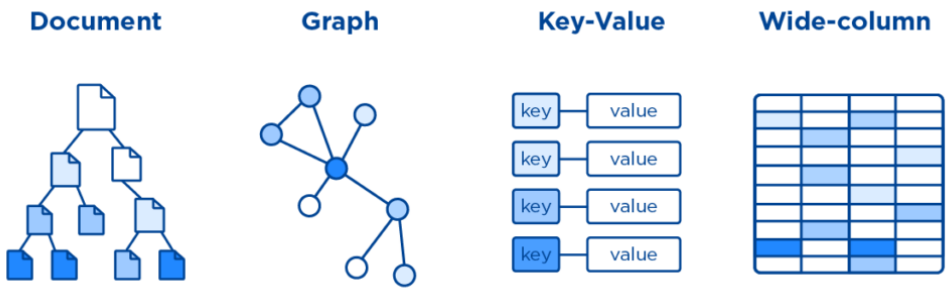
Antigamente, eram necessários computadores grandes para armazenar dados. Com o passar do tempo, surgiu a capacidade de armazenar centenas de gigabytes em um único disco, permitindo que SGBDRs pudessem ser executados em máquinas menores. No entanto, os dados corporativos cresceram exponencialmente, conforme

discutido no (GARCIA-MOLINA, ULLMAN e WIDOM, 2009). Esse cenário impulsionou o surgimento dos bancos de dados não relacionais.

Também conhecidos como bancos de dados *Not Only SQL* (NoSQL), os bancos de dados não relacionais diferem dos relacionais, principalmente na abordagem de hardware. Nos bancos de dados NoSQL, é comum o uso de *clusters*, onde servidores tradicionais são agrupados para lidar com volumes de dados distribuídos e demandas escaláveis. Essa arquitetura em clusters destaca a capacidade desses sistemas de se adaptarem a ambientes distribuídos, permitindo uma escalabilidade horizontal eficiente (SADALAGE e FOWLER, 2013).

Os bancos de dados NoSQL tendem a lidar com grandes volumes de dados por meio da escalabilidade horizontal, podendo ser definida como distribuição de processamento em vários servidores, apresentando flexibilidade no esquema, alto desempenho e baixa latência. Esses bancos de dados podem ser classificados em diferentes modelos, como Documentos (*MongoDB*, *CouchDB*), pares de chave-valor (*Redis*, *DynamoDB*), famílias de colunas (*Apache Cassandra*, *HBase*) e grafos (*Neo4j*, *Amazon Neptune*) (SADALAGE e FOWLER, 2013). A Figura 7 mostra uma representação lúdica da estrutura dos diferentes modelos de bancos de dados NoSQL.

Figura 7 - Representação lúdica da estrutura dos diferentes modelos de bancos de dados NoSQL



Fonte: (LOGAP, 2022).

Não é possível determinar qual é o melhor banco de dados ou modelo; tudo depende das especificações da sua aplicação. No contexto deste trabalho, a

armazenagem e consulta dos valores de temperatura, tensão e outros dados mostraram um desempenho superior ao utilizar banco de dado relacional, *PostgreSQL* (GRUPO DE DESENVOLVIMENTO GLOBAL POSTGRESQL, 2024).

O projeto *POSTGRES*, liderado pelo professor Michael Stonebraker no Departamento de Ciência da Computação da Universidade da Califórnia, teve início em 1986 e foi patrocinado por várias organizações. Ao longo de várias versões, introduziu conceitos inovadores e passou por melhorias significativas, sendo utilizado em diversas aplicações, desde análise financeira até sistemas de informações geográficas (GRUPO DE DESENVOLVIMENTO GLOBAL POSTGRESQL, 2024).

Em 1994, o *Postgres95*, uma versão com interpretador SQL, foi lançado como descendente do código aberto *POSTGRES* original. O *Postgres95* apresentou melhorias de desempenho e funcionalidades.

Em 1996, ficou claro que o nome *Postgres95* não seria duradouro. O novo nome escolhido foi *PostgreSQL*, mantendo assim a relação com o original e as versões mais recentes com SQL. Ainda hoje, é comum utilizar "*Postgres*" para se referir ao *PostgreSQL*, seja por tradição ou facilidade na pronúncia, como se fosse um apelido ou "*alias*”.

Como mencionado, essa plataforma de código aberto possui uma ampla gama de funcionalidades, incluindo SQL completo, Transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), extensibilidade, *triggers*, *stored procedures*, integridade referencial, índices avançados, suporte a *JavaScript Object Notation* (JSON), tipos de dados complexos, segurança avançada, entre outros (GRUPO DE DESENVOLVIMENTO GLOBAL POSTGRESQL, 2024).

Neste trabalho, será utilizado o PostgreSQL, que, além de apresentar um modelo relacional, foi escolhido por atender a todos os requisitos do sistema. A continuidade com o banco previamente construído facilita o desenvolvimento desta aplicação, beneficiando-se das características de robustez, escalabilidade e suporte a operações complexas que o PostgreSQL oferece. Além disso, trata-se de uma solução de código aberto amplamente utilizada e confiável.

2.4 Segurança da Informação

2.4.1 Introdução à Segurança da Informação

A segurança da informação é a prática de proteger informações valiosas contra ameaças e riscos. Envolve a implementação de medidas e controles para garantir a confidencialidade, integridade e disponibilidade dos dados. A importância da segurança da informação tem crescido exponencialmente com o aumento da dependência de sistemas digitais para armazenar e processar informações sensíveis (ANDERSON, 2001).

As características de cada medida de controle são:

- Confidencialidade: assegura que as informações sejam acessíveis apenas por aqueles autorizados, prevenindo o acesso não autorizado e protegendo a privacidade dos dados. Medidas comuns para garantir a confidencialidade incluem criptografia, controles de acesso e autenticação forte.
- Integridade: assegura que a informação seja confiável e precisa, prevenindo alterações não autorizadas, garantindo que os dados não sejam manipulados ou corrompidos. Medidas comuns para manter a integridade incluem *hashing*, *checksums* e controle de versão.
- Disponibilidade: garante que a informação e os sistemas estejam acessíveis e operacionais quando necessário, protegendo contra interrupções de serviço. Medidas comuns para assegurar a disponibilidade incluem *backup* e recuperação, redundância e mitigação de ataques de negação de serviço distribuído.

As ameaças e vulnerabilidades à segurança da informação incluem ataques cibernéticos, vulnerabilidades de *softwares* e erros humanos. Ataques cibernéticos, como *phishing*, *malware* e ataques de força bruta representam uma ameaça significativa à segurança da informação, podendo comprometer dados sensíveis, causar danos financeiros e prejudicar a reputação. Vulnerabilidades em *software*

podem ser exploradas por atacantes para ganhar acesso não autorizado a sistemas, incluindo falhas de segurança, *bugs* e problemas de configuração. Erros humanos, como a má configuração de sistemas, uso de senhas fracas e *phishing*, são uma causa comum de violações de segurança (NIELES, DEMPSEY e PILLITTERI, 2017).

2.4.2 Criptografia

A criptografia é uma técnica utilizada para proteger informações, transformando-as em um formato ilegível para qualquer pessoa que não tenha a chave necessária para decifrá-las. Historicamente, a criptografia tem sido usada para proteger informações confidenciais de olhares curiosos, começando com métodos simples, como a cifra de César, e evoluindo para os algoritmos complexos utilizados atualmente. O objetivo principal da criptografia é garantir que as informações sejam mantidas em segredo e acessíveis apenas a quem possui a chave correta. Além disso, a criptografia ajuda a assegurar a integridade dos dados, garantindo que a informação não foi alterada durante o trânsito (STALLINGS, 2014).

Existem diferentes abordagens para a criptografia, cada uma com suas vantagens e desafios. Na criptografia simétrica, a mesma chave é usada tanto para cifrar quanto para decifrar os dados. Este método é conhecido por sua eficiência e velocidade, sendo ideal para criptografar grandes volumes de dados. No entanto, o principal desafio é a distribuição segura da chave entre as partes que se comunicam. Um exemplo comum de criptografia simétrica é o algoritmo Blowfish, que é reconhecido por sua flexibilidade e segurança, sendo utilizado em várias aplicações, como a proteção de senhas armazenadas (SCHNEIER, 1996).

Por outro lado, a criptografia assimétrica utiliza um par de chaves: uma chave pública e uma chave privada. A chave pública é usada para cifrar os dados, enquanto a chave privada é usada para decifrá-los. Este método resolve o problema da distribuição de chaves encontrado na criptografia simétrica, pois a chave pública pode ser distribuída livremente, enquanto a chave privada é mantida em segredo. Um exemplo de criptografia assimétrica é o *Rivest-Shamir-Adleman* (RSA), amplamente utilizado em certificados digitais e assinaturas digitais para assegurar a autenticidade e integridade das comunicações. É comumente implementado em protocolos de

segurança como *Secure Sockets Layer* (SSL)/*Transport Layer Security* (TLS), que protegem as comunicações na internet (MENEZES, OORSCHOT e VANSTONE, 1996).

Além da criptografia, outra técnica essencial para a segurança da informação é o *hashing*. *Hashing* é o processo de transformar dados de entrada em uma sequência de caracteres de comprimento fixo, que parece aleatória. As funções de *hash* são unidirecionais, o que significa que não é possível reverter o *hash* para obter os dados originais. *Hashing* é utilizado principalmente para verificar a integridade dos dados e autenticar informações, sem a necessidade de armazenar dados sensíveis (FERGUSON, SCHNEIER e KOHNO, 2010).

Embora *hashing* não seja uma forma de criptografia, ele é um processo relacionado que serve a um propósito diferente. Ele é usado para garantir a integridade dos dados, enquanto a criptografia é usada para proteger a confidencialidade dos dados. Portanto, *hashing* é um processo distinto que complementa as técnicas criptográficas em muitas aplicações de segurança da informação (FERGUSON, SCHNEIER e KOHNO, 2010).

Uma biblioteca amplamente utilizada para *hashing* seguro de senhas é o *bcrypt*. O *bcrypt* foi projetado para ser computacionalmente difícil de quebrar, tornando-o uma escolha popular para proteger senhas. Ele adiciona um *salt* único a cada senha antes de realizar o *hashing*. Além disso, o *bcrypt* permite ajustar o fator de custo, aumentando o tempo necessário para calcular o *hash* conforme a capacidade de processamento aumenta, proporcionando uma camada adicional de segurança ao longo do tempo (PROVOS e MAZIERES, 1999).

As principais vantagens do *bcrypt* são:

- *Salting* automático: técnica de segurança usada durante o processo de *hashing* de senhas, onde um valor aleatório, chamado de "*salt*", é gerado e adicionado automaticamente à senha antes de ser criado o *hash*. Isso aumenta a segurança das senhas de várias maneiras:
- *Uniqueness*: mesmo que dois usuários tenham a mesma senha, a adição de um *salt* diferente para cada um garantirá que os *hashes* resultantes sejam diferentes.

- Resistência a ataques de *Rainbow Tables*: *rainbow tables* são pré-computações de *hashes* para senhas comuns. *Salting* torna essas tabelas ineficazes porque a adição de um valor aleatório cria *hashes* únicos que não correspondem aos pré-computados.
- Proteção contra ataques de força bruta: O uso de *salts* aumenta a complexidade e o tempo necessário para realizar ataques de força bruta, pois cada tentativa de senha exigiria um novo cálculo com um *salt* diferente.

Outro método de *hash* amplamente utilizada é o SHA-256. SHA-256 produz um *hash* de 256 bits e é amplamente utilizado devido à sua segurança e desempenho rápido. Embora seja menos adequado para *hashing* de senhas devido à sua velocidade, que pode facilitar ataques de força bruta, é excelente para verificar a integridade dos dados (VENESS, 2019). SHA-256 é usado em várias aplicações, incluindo a *blockchain* do *Bitcoin*, para garantir a integridade das transações (NELOGICA, 2023).

O PBKDF2 é outra opção popular para proteger senhas. Ele aplica uma função de *hash* múltiplas vezes para aumentar a segurança da senha. PBKDF2 é flexível e suporta *salting*, tornando-se uma escolha segura para proteger senhas. O uso de múltiplas iterações aumenta o tempo necessário para realizar ataques de força bruta, tornando-os menos viáveis (HACKERONE, 2024).

2.4.3 JWT (JSON Web Token)

O *JSON Web Token* (JWT) é uma forma compacta e autossuficiente de representar informações de forma segura entre partes como um objeto JSON. Ele é amplamente utilizado para autenticação e autorização em aplicações *web*, especialmente em APIs (JWT.IO, 2024).

Um JWT é composto por três partes principais: *header*, *payload* e *signature.*

- *Header*: Contém o tipo do *token* e o algoritmo de criptografia utilizado.

- *Payload*: Contém as reivindicações, *claims,* que são declarações sobre uma entidade, normalmente o usuário, e dados adicionais. Exemplos de reivindicações são *iss* (*issuer*), *sub* (*subject*) e *exp* (*expiration*).
- *Signature*: É criada combinando o *header* codificado em *base64*, um método utilizado na Internet para codificar de dados transferidos, o payload codificado também em *base64* e uma chave secreta, usando o algoritmo especificado no *header*. A assinatura é utilizada para verificar se o token não foi alterado.

As principais vantagens do JWT são:

- Autossuficiente: Contém todas as informações necessárias sobre a entidade, eliminando a necessidade de acessar um banco de dados para verificar o token.
- Compacto: Devido à sua estrutura baseada em JSON e codificação em *base64*, é eficiente para ser transmitido via HTTP por exemplo, em *headers* ou URLs.
- Seguro: Utiliza algoritmos criptográficos para garantir a integridade e a autenticidade do token.

Para a implementação de criptografia na API HTTP foi escolhido utilizar a biblioteca *bcrypt*, pela facilidade e praticidade de sua implementação, além da vasta gama de documentação disponível. Pois a adição de um *salt* automático e o uso de algoritmos robustos de *hashing* garantem que as senhas dos usuários sejam protegidas de maneira eficaz, contribuindo significativamente para a segurança geral da aplicação. Além disso, a utilização de JWT para autenticação e autorização fortalece ainda mais a segurança, garantindo que apenas usuários autenticados possam acessar recursos protegidos na API.

2.5 Protocolos de comunicação

Antes de entender o que são protocolos de comunicação é importante compreender o conceito de arquitetura de rede. A arquitetura de rede compreende um conjunto abrangente de diretrizes e estruturas para organizar, projetar e implementar uma rede de computadores, integrando conceitos físicos, como dispositivos e cabos, com elementos conceituais, como protocolos e padrões. Esse enfoque visa estabelecer uma infraestrutura de comunicação eficaz e segura. Essa arquitetura engloba diversos elementos essenciais, incluindo *hardware*, *software*, protocolos e a interconexão de dispositivos, considerando tanto os aspectos físicos quanto os conceituais (SOUZA, 2012).

No âmbito físico, a arquitetura de rede especifica os dispositivos que compõem a infraestrutura, como roteadores, *hubs*, *switches*, cabos e servidores. Esses componentes físicos desempenham papéis distintos, como roteamento de dados, concentração de conexões e fornecimento de recursos de processamento, armazenamento e serviços. A arquitetura também aborda questões relacionadas à transmissão de dados, determinando como os dados serão transmitidos entre os dispositivos conectados. Isso envolve a escolha de protocolos de comunicação, que definem como os dados são fragmentados, transmitidos e reagrupados (DAVIE, 2013).

Além disso, é crucial compreender os paradigmas de rede, que representam formas distintas de organizar a comunicação e colaboração entre os elementos em uma rede. Alguns exemplos são Cliente-Servidor, Produtor-Consumidor e Mestre-Escravo. A escolha da arquitetura e do paradigma depende da avaliação dos requisitos da aplicação e da consideração dos prós e contras de cada opção. (TANENBAUM, 2003).

Cliente-Servidor: Um cliente faz requisições para um servidor, que processa e retorna as respostas correspondentes. Permite a separação de responsabilidades entre o cliente, que solicita serviços, e o servidor, que fornece esses serviços.

Publicador-Assinante: Os sistemas podem publicar mensagens sem saber quem ou se alguém as receberá, enquanto outros sistemas podem assinar para receber

mensagens específicas. É comumente utilizado em sistemas de mensageria e em ambientes distribuídos para comunicação assíncrona.

Mestre-Escravo: Este modelo envolve um mestre que coordena e controla os escravos. O mestre delega tarefas aos escravos, que executam as tarefas atribuídas e reportam ao mestre.

Produtor-Consumidor: Os produtores geram dados ou mensagens e os colocam em um espaço compartilhado (como uma fila), enquanto os consumidores retiram esses dados do espaço compartilhado e os processam.

Para compreender o conceito deste tópico, é fundamental elucidar o termo "protocolo", que se refere a um conjunto de normas e padrões. Protocolos são estabelecidos em diversas situações, incluindo interações entre pessoas, onde uma se expressa e outra escuta, refletindo uma troca que ocorre durante a conversa. Analogamente, um protocolo de comunicação é uma maneira padronizada de transmitir, formatar e receber dados entre dispositivos.

A eficácia de um protocolo de comunicação está intrinsecamente ligada à sua capacidade de assegurar confiabilidade e segurança nas transmissões de dados, aderindo a regras específicas. A garantia da correção de erros e da integridade dos dados é de suma importância. Uma variedade de protocolos é amplamente reconhecida, incluindo *Transmission Control Protocol* (TCP)*/Internet Protocol* (IP), HTTP, SMTP, FTP, MQTT, entre outros, desempenhando papéis cruciais na conexão e comunicação entre dispositivos em redes.

2.5.1 HTTP

As sementes das redes de computadores foram plantadas na década de 1960 através de projetos de pesquisa no Departamento de Defesa dos Estados Unidos. Ao longo das décadas seguintes, esses projetos se expandiram para incluir universidades e laboratórios nos Estados Unidos e na Europa, pavimentando o caminho para a interconexão global de sistemas e dispositivos (SANTANA, 2024).

No início dos anos 1990, a *World Wide Web*, conhecida como a Internet moderna, começou a se materializar. Um desafio crucial era a falta de um protocolo

padronizado que permitisse a conexão eficiente entre clientes e servidores em escala global. Foi nesse contexto que surgiu o HTTP, desenvolvido por Tim Berners-Lee e sua equipe no Conselho Europeu para Pesquisa Nuclear (CERN) entre 1989 e 1991. Este protocolo foi concebido para ser leve e fácil de operar, funcionando na camada de aplicação e desempenha um papel fundamental na facilitação da comunicação e troca de informações entre computadores ao redor do mundo (WORLD WIDE WEB FOUNDATION., 2022).

O HTTP é empregado na comunicação entre navegadores *web* e servidores, seguindo o modelo cliente-servidor. No entanto, apresenta limitações, especialmente em segurança, devido à transferência de informações em texto simples. A complexidade na comunicação baseada em texto pode afetar o desempenho, principalmente em ambientes com largura de banda limitada. O HTTPS surge como uma extensão segura do HTTP, incorporando uma camada adicional de proteção com SSL/TLS (CLOUDFLARE, 2024).

O HTTP, Protocolo de Transferência de Hipertexto, opera com base em um modelo de requisição e resposta entre clientes e servidores. Quando um cliente, como um navegador *web*, envia uma requisição HTTP para um servidor, ele especifica um método HTTP para indicar a ação desejada sobre um recurso específico (GOURLEY e TOTTY, 2002). Os principais métodos HTTP incluem:

- *GET*: Utilizado para solicitar dados de um recurso específico no servidor.
- *POST*: Envia dados do cliente para o servidor.
- *PUT*: Atualiza um recurso existente no servidor com os dados fornecidos na requisição.
- *DELETE*: Remove um recurso específico do servidor conforme indicado na requisição.
- *PATCH*: Similar ao *PUT*, mas atualiza parcialmente um recurso com os dados especificados na requisição.
- *HEAD*: Similar ao GET, obtém apenas os cabeçalhos de resposta sem o corpo da mensagem, útil para verificar metadados ou a validade de um recurso.

- *OPTIONS*: Obtém as opções de comunicação disponíveis para um recurso ou servidor antes de realizar a requisição propriamente dita.

O HTTP é arquitetado com base em mensagens de requisição e resposta, onde tanto a requisição quanto a resposta consistem em três partes principais: linha inicial, cabeçalhos (*headers*) e opcionalmente um corpo (*body*).

Linha Inicial:

- Requisição: Contém o método HTTP (*GET*, *POST*, *PUT*, *DELETE*, entre outros.), o URL do recurso solicitado e a versão do protocolo HTTP sendo utilizada.
- Resposta: Inclui a versão do protocolo HTTP, um código de *status* (indicando o resultado da operação), e uma mensagem de *status* textual.

Cabeçalhos (*Headers*):

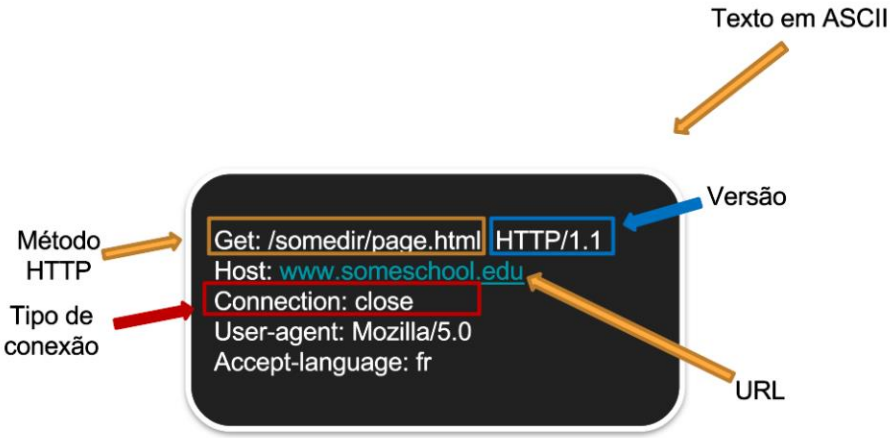
- Requisição: Contêm informações adicionais sobre a requisição, como tipo de mídia aceito pelo cliente, *cookies*, tipo de autenticação, entre outros.
- Resposta: Fornecem informações sobre o servidor, como tipo de servidor, data da resposta, tamanho do conteúdo, entre outros metadados.

Corpo (*Body*):

- Requisição: Opcionalmente, contém dados enviados pelo cliente ao servidor. Por exemplo, no método *POST*, o corpo pode conter dados de formulário ou de *upload* de arquivos.
- Resposta: Contém os dados solicitados pelo cliente, como HTML de uma página *web*, arquivos de mídia, ou qualquer outro conteúdo relevante.

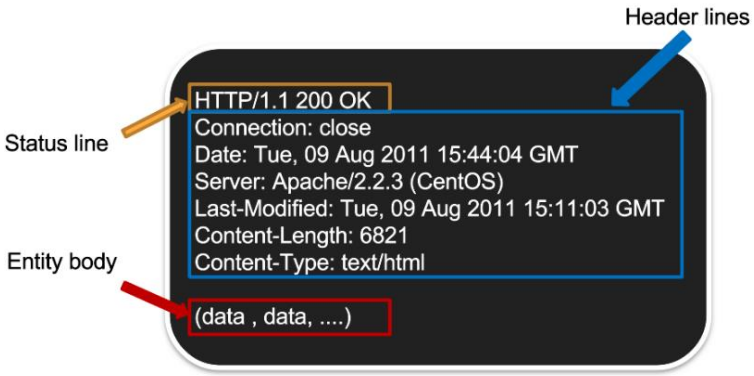
A Figura 8 apresenta um exemplo de estrutura da mensagem de requisição HTTP. Já a Figura 9 mostra um exemplo de estrutura de mensagem de resposta HTTP.

Figura 8 - Exemplo de estrutura da mensagem de requisição HTTP.



Fonte: (SIMPLIFICANDOREDE, 2021).

Figura 9 - Exemplo de estrutura da mensagem de resposta HTTP.



Fonte: (SIMPLIFICANDOREDE, 2021).

2.5.2 MQTT

O MQTT, criado pela IBM e *Eurotech* na década de 90, destaca-se por sua simplicidade e segurança na comunicação. Projetado para dispositivos com recursos limitados, opera como um protocolo de mensagens de publicação/assinatura, garantindo confiabilidade na entrega. Sua eficiência em largura de banda e processamento é notável, sendo amplamente utilizado em IoT, automação industrial e mensagens em tempo real. O MQTT proporciona uma comunicação eficiente entre

dispositivos e servidores, tornando-se essencial em diversos setores (AMAZON WEB SERVICES, 2023).

Principais usos do MQTT incluem:

- Comunicação IoT: Aplicado em dispositivos IoT para comunicação eficiente entre dispositivos e servidores back-end, como em sistemas domésticos inteligentes.
- Automação industrial: Eficiente troca de dados em sistemas de automação industrial, controlando e monitorando máquinas.
- Mensagens em tempo real: Capacidade de enviar mensagens em tempo real, sendo utilizado em chats e serviços de mensagens instantâneas.

O MQTT é implementado em diversos contextos, como no *Eclipse Mosquitto* (*broker* MQTT de código aberto), *Amazon Web Services IoT Core* (serviço de nuvem para comunicação MQTT em IoT) e *HiveMQ* (*broker* MQTT comercial com recursos avançados). O protocolo desempenha um papel crucial em setores como automotivo, logística, manufatura, automação residencial, produtos de consumo e transporte.

Roisenberg (2023) cita que os principais componentes e conceitos da arquitetura MQTT, baseiam-se no paradigma publicador-assinante, são eles:

- Cliente MQTT: Existem dois tipos principais de clientes MQTT: Os clientes publicadores são responsáveis por enviar mensagens para o servidor MQTT (*broker*), já os clientes assinantes recebem mensagens de tópicos específicos nos quais estão inscritos.
- *Broker* MQTT: O *broker* MQTT é um servidor intermediário que recebe todas as mensagens publicadas pelos clientes e, em seguida, encaminha essas mensagens aos clientes assinantes que estão inscritos nos tópicos correspondentes. O *broker* gerencia a distribuição das mensagens de forma eficiente.
- Tópicos: Os tópicos são *strings* usadas pelos clientes para categorizar as mensagens que desejam publicar ou assinar. Os tópicos são organizados

em uma estrutura de árvore hierárquica, permitindo que os clientes assinantes recebam mensagens de forma granular.

- *Quality of Service* (QoS): O MQTT suporta três níveis de QoS para garantir a entrega das mensagens conforme a necessidade do aplicativo:
    - QoS 0: Entrega no máximo uma vez onde não há garantia de entrega.
    - QoS 1: Pelo menos uma entrega, garante que a mensagem é recebida pelo broker.
    - QoS 2: Exatamente uma entrega, garante que a mensagem é recebida pelo broker e pelo assinante apenas uma vez.

O protocolo MQTT define vários tipos de mensagens, cada uma com uma função específica. Abaixo, detalhamos os três principais tipos de mensagens:

- *Connect*: Inicia uma conexão com o broker MQTT, aguardando até que a conexão seja estabelecida. Uma vez conectada, a aplicação começa a escutar as mensagens publicadas.
- *Disconnect*: Aguarda a finalização de qualquer ação em curso pelo cliente e encerra a conexão TCP/IP, interrompendo a recepção de novas mensagens.
- *Publish*: Envia a informação do cliente MQTT para o broker, que a repassa para todos os assinantes interessados.

Além desses, o MQTT inclui outros tipos de mensagens, como:

- *Subscribe*: Inscreve o cliente em um ou mais tópicos.
- *Unsubscribe*: Cancela a inscrição do cliente em um ou mais tópicos.
- *Pingreq* e *Pingresp*: Mantêm a conexão ativa e verificam se o broker está acessível.

A Tabela 1 apresenta as mensagens MQTT com suas respectivas direções e descrição.

Tabela 1 - Tabela de Mensagem do protocolo MQTT.

Mensagem MQTT	Direção	Descrição
CONNECT	Cliente para Servidor	Requisição do cliente para conectar ao servidor
CONNACK	Servidor para Cliente	Reconhecimento da conexão
PUBLISH	Cliente para Servidor ou Servidor para cliente	Publicar mensagem
PUBACK	Cliente para Servidor ou Servidor para cliente	Reconhecimento da publicação
PUBREC	Cliente para servidor ou servidor para cliente	Publicação recebida
PUBREL	Cliente para servidor ou servidor para cliente	Publicar lançamento
PUBCOMP	Cliente para Servidor ou Servidor para cliente	Publicação completa
SUBSCRIBE	Cliente para Servidor	Pedido de inscrição
SUBACK	Servidor para cliente	Reconhecimento de inscrição
UNSUBSCRIBE	Cliente para Servidor	Pedido de cancelar subscrição
UNSUBACK	Servidor para cliente	Confirmação de cancelamento de inscrição
PINGREQ	Cliente para Servidor	Requisição PING
PINGRESP	Servidor para cliente	Resposta PING
DISCONNECT	Cliente para Servidor	Cliente está desconectado

Fonte: (RF WIRELESS WORLD, 2022).

2.6 Integração e Aplicação de APIs: Desenvolvimento e Implementação no Contexto de Sistemas Modernos

Segundo a *Mozilla Developer Network* (2024), "uma API (*Application Programming Interface*) é um conjunto de características e regras existentes em uma aplicação que possibilitam interações com esta através de um *software*". Essa Interface de Programação de Aplicações possui um conjunto de regras que tem o

objetivo de facilitar a troca de dados, recursos e funcionalidades entre aplicações (GOODWIN, 2024).

Entre as importâncias das APIs no desenvolvimento de *software* moderno estão a simplificação do desenvolvimento de aplicativos e serviços, e a segurança nos sistemas, já que a infraestrutura de serviços e respostas está separada do aplicativo em si e, em geral, são utilizadas credenciais de autenticação nas comunicações. A segurança e privacidade do usuário também são garantidas, uma vez que essas solicitações são permitidas ou não pelo usuário. Nos navegadores *web* e sistemas operacionais, em geral, as permissões já vêm integradas no ambiente.

Os benefícios de usar APIs incluem a reutilização de código, o desenvolvimento rápido, a interoperabilidade, a escalabilidade e a flexibilidade. Para compreender o funcionamento de uma API, é possível compará-la a um contrato. Se temos um documento que representa um acordo entre duas partes, a primeira envia uma solicitação com uma estrutura específica, e assim a segunda parte responderá (RED HAT, 2022).

Assim como um documento, a Interface de Programação de Aplicações possui uma estrutura. Em sua composição estão o endpoint (ponto de acesso ao serviço da API), os métodos HTTP (*GET*, *POST*, *PUT*, *DELETE*), parâmetros (rota, *query*, cabeçalho e corpo) e a resposta, que define o formato de retorno geralmente JSON ou *Extensible Markup Language* (XML).

2.6.1 Tipos de APIs e Arquitetura RESTful

Segundo Goodwin (2024), existem quatro tipos de interfaces de programação de aplicativos. As APIs de banco de dados são utilizadas para conectar sistemas de gerenciamento de banco de dados a aplicativos. As APIs de sistema operacional local definem como os aplicativos utilizam recursos e serviços do sistema. As APIs remotas permitem a troca de serviços e dados entre dispositivos diferentes. E as APIs *web* transferem dados e funcionalidades através do protocolo HTTP.

Dentro das APIs *web*, existem quatro tipos principais: *RESTful*, *Simple Object Access Protocol* (SOAP), *GraphQL* e *WebSockets*.

As interfaces de programação de aplicativos do tipo *WebSocket* são ideais para comunicação em tempo real, visto que permitem uma comunicação bidirecional entre o cliente e o servidor. Uma vez que o canal de comunicação é aberto, ele permanece aberto, permitindo uma troca contínua de dados (PEREIRA, 2023). O método HTTP utilizado é o *GET*, que é utilizado para estabelecer a conexão antes do *handshake* inicial.

O *GraphQL* oferece uma abordagem simples e flexível, pois os clientes podem solicitar exatamente os dados de que precisam em apenas uma requisição, especificando a estrutura desejada. Geralmente, utiliza requisições do tipo *POST*, mas também suporta requisições *GET* para consultas simples (THE GRAPHQL FOUNDATION, 2024).

O SOAP é uma especificação de protocolo baseada em XML que permite o envio e recebimento de dados através de diversos protocolos, como *Simple Mail Transfer Protocol* SMTP e HTTP (EXPRESSVPS, 2024).

A API *RESTful*, diferente do protocolo SOAP, é um estilo arquitetural. A palavra "*REST*" vem de Transferência de Estado, do inglês *Representational State Transfer* (REST) e existem seis restrições fundamentais: Arquitetura Cliente-Servidor, Ausência de Estado, Capacidade de Armazenamento em Cache, Sistema em Camadas, Código Sob Demanda e Interface Uniforme (FIELDING, 2000).

- Arquitetura Cliente-Servidor: É baseada em uma separação clara entre clientes, servidores e recursos, com comunicação ocorrendo via HTTP.
- Ausência de Estado: O servidor não mantém informações de estado da sessão do cliente entre as solicitações. O cliente gera o estado de sessão.
- Capacidade de Armazenamento em *Cache*: Respostas são armazenadas em cache para melhorar a eficiência e reduzir a latência.
- Sistema em Camadas: Suporta utilização de sistemas intermediários entre o cliente e o servidor, proporcionando escalabilidade, segurança e melhor desempenho.
- Código Sob Demanda: Os servidores podem enviar código executável ao cliente para aprimorar suas funcionalidades.

- Interface Uniforme: Abrange quatro aspectos fundamentais: a identificação de recursos através de *Uniform Resource Identifier* (URIs) em solicitações; a manipulação desses recursos por meio de representações como JSON ou XML, permitindo operações CRUD (Criar, Ler, Atualizar e Deletar); mensagens auto descritivas que garantem que cada resposta do servidor contenha todas as informações necessárias para o cliente processá-la, incluindo ações permitidas e formatos de dados aceitos; e o uso de hipermídia como mecanismo para navegação dinâmica na aplicação, possibilitando que o cliente explore a aplicação de forma dinâmica através de links hipermídia retornados nas respostas da API.

Outro tipo menos comum de API é a API HTTP, que utiliza o protocolo HTTP para comunicação entre clientes e servidores. Ela funciona como socket aberto no servidor *web*, aguardando requisições específicas. Esse protocolo opera com conceitos essenciais: *request*, que representa um pedido; *response*, a resposta ao pedido; *resource*, o recurso solicitado; e *verb/method*, que são ações aplicáveis a um recurso. No contexto de aplicações *web*, as APIs HTTP permitem a interação com recursos de um servidor através dos métodos HTTP. Esses recursos são gerenciados e manipulados usando métodos como GET, POST, PUT, PATCH e DELETE. Essa abordagem facilita a gestão eficiente dos recursos, possibilitando operações de criação, leitura, atualização e exclusão (ANSELME, 2023).

Uma das principais vantagens da APIs HTTP é o canal de comunicação eficiente entre o cliente e o servidor. Os clientes enviam requisições HTTP, e os servidores respondem com os dados solicitados. Esse modelo é essencial para a interação dinâmica em aplicativos *web* modernos, permitindo uma troca de informações fluida e em tempo real. Como são extremamente versáteis podem ser usadas em uma variedade de aplicações. Elas permitem a criação de serviços acessíveis por diferentes tipos de clientes, como navegadores, dispositivos móveis e outros servidores, facilitando a interoperabilidade entre diversos sistemas (BOLTIC, 2024).

Também suportam autenticação, garantindo que apenas usuários autorizados possam acessar informações sensíveis. Isso é crucial para a segurança em

aplicações que lidam com dados confidenciais ou transações seguras, protegendo contra acessos não autorizados e ataques maliciosos. Além de facilitar a comunicação entre aplicações, as APIs HTTP são úteis para tarefas de manutenção e monitoramento. Elas podem ser utilizadas para coletar métricas de desempenho, gerenciar configurações de sistemas remotamente e realizar diagnósticos, auxiliando na administração eficiente de sistemas complexos (BOLTIC, 2024).

É comum confundir API HTTP e API *RESTful*. Embora ambas utilizem o protocolo HTTP, há diferenças importantes entre APIs HTTP e APIs *RESTful*:

No contexto da arquitetura:

- API HTTP: É uma implementação genérica que utiliza HTTP para a comunicação, sem seguir necessariamente um conjunto rigoroso de princípios arquiteturais.
- API *RESTful*: Adere aos princípios arquiteturais do REST, sendo projetada para ser escalável, sem estado e baseada em recursos.

No contexto dos princípios arquiteturais:

- API HTTP: Pode não seguir princípios como *statelessness*, *cacheability* e a separação entre cliente e servidor.
- API *RESTful*: Deve seguir os seis princípios arquiteturais do REST, que incluem uniformidade da interface, *statelessness*, capacidade de cache, sistema em camadas, código sob demanda (opcional) e a separação entre cliente e servidor.

No contexto de design e usabilidade:

- API HTTP: Pode ter um design menos uniforme e mais específico para a aplicação, sem necessariamente expor recursos de maneira consistente.
- - **API *RESTful***: Foca na exposição de recursos e na manipulação desses recursos usando uma interface uniforme e consistente.

No contexto de flexibilidade:

- API HTTP: Oferece mais flexibilidade para implementar qualquer tipo de comunicação baseada em HTTP, sem restrições arquiteturais.

- API *RESTful*: É mais restrita em termos de design, pois segue os princípios do REST, proporcionando maior consistência e previsibilidade.

APIs HTTP são uma peça fundamental na construção de aplicações *web* modernas, oferecendo uma maneira eficiente e versátil de comunicação entre clientes e servidores. Com suporte para autenticação, versatilidade no uso e funcionalidades que facilitam a manutenção e o monitoramento de sistemas, elas se destacam como uma ferramenta poderosa para desenvolvedores. Ao entender as diferenças entre APIs HTTP e *RESTful*, é possível escolher a abordagem mais adequada às necessidades específicas de cada projeto, garantindo eficiência e escalabilidade.

Para este trabalho foi escolhido utilizar API HTTP, pois possui uma implementação mais simples e engloba todos os requisitos determinados no escopo da pesquisa.

2.6.2 Aplicações de APIs em Diferentes Setores

Existem inúmeras aplicações de APIs em diversos setores e contextos, como Redes Sociais e Mídias Digitais, Pagamentos e Finanças, *E-commerce*, Saúde e Bem-estar, e Educação. Algumas das APIs mais utilizadas incluem o *Facebook Graph* API, que permite aos desenvolvedores acessarem e interagir com dados do *Facebook*, como perfis de usuário, postagens, fotos e eventos. A *Google Maps* API facilita a integração de mapas interativos, direções e serviços de localização em aplicativos *web* e móveis. A *PayPal* API possibilita realizar pagamentos online, enviar dinheiro e gerenciar transações financeiras. A *Amazon Product Advertising* API fornece informações detalhadas sobre produtos disponíveis na *Amazon*, como preços, descrições e avaliações. A *Apple HealthKit* API permite a integração de dados de saúde de dispositivos *Apple*, como frequência cardíaca e passos. E a *Google Classroom* API oferece acesso a informações sobre cursos, tarefas e usuários na plataforma *Google Classroom*, facilitando a integração com sistemas educacionais (MOZILLA DEVELOPER NETWORK, 2024).

2.7 *Node.js* e *Express* para Criação de APIs

Os sistemas *web* desenvolvidos em plataformas como .NET, *Java*, PHP, *Ruby* ou *Python* frequentemente utilizam um modelo bloqueante, conhecido como *Blocking-Thread*, onde cada requisição de usuário paralisa o processamento enquanto realiza operações de entrada e saída no servidor. Isso resulta em uma fila de requisições, onde cada uma é processada sequencialmente, causando espera para as demais. Este modelo clássico é ineficiente, já que tarefas como consultas a banco de dados ou leituras em disco bloqueiam o sistema inteiro, resultando em tempo ocioso e necessidade de atualizações de *hardware* para lidar com picos de tráfego. A *Node.js* surge como uma alternativa eficiente, maximizando o uso do processador atual ao lidar de forma assíncrona com tarefas bloqueantes, reduzindo a espera e melhorando o desempenho geral do sistema (PEREIRA, 2013).

A *Node.js* se diferencia de outros sistemas por sua abordagem no ciclo de eventos, onde não há uma chamada bloqueante para iniciar o ciclo. Em vez disso, a execução entra diretamente no ciclo de eventos após a conclusão do programa de entrada. Esse comportamento é semelhante ao *JavaScript* no navegador, onde o ciclo de eventos é gerenciado internamente e não é visível para o usuário. Além disso, a *Node.js* valoriza o protocolo de transferência de hipertexto (HTTP) como fundamental, otimizado para fluxo contínuo e baixa latência. Isso a torna adequada como base para bibliotecas e abstrações *web* (OPENJS FOUNDATION, 2024).

Ao falar do *Node.js* é importante mencionar o *Node Package Manager* (NPM), o gerenciador de pacotes padrão para o *Node.js*. Ele facilita a instalação, atualização, remoção e gerenciamento de pacotes e bibliotecas de *software JavaScript* que podem ser utilizados em projetos *Node.js.*

2.7.1 Frameworks do Node.js

Segundo *Amazon Web Services* (2023) “um *framework* é uma coleção de componentes de *software* reutilizáveis que tornam mais eficiente o desenvolvimento de novas aplicações.”.

Para facilitar a criação de aplicações *web* e APIs utilizando o *Node.js* podem ser utilizados *frameworks*. O mais popular é o *Express.js* por ser mínimo e flexível fornecendo um conjunto robusto de recursos, possui também uma infinidade de métodos HTTP e *midleware*, serviço intermediário entre aplicações, para criação de APIs robustas de forma rápida, (NODE.JS FOUNDATION, 2024).

Outro *framework* utilizado no *Node.js* é o *Koa*, criado pela equipe do *Express*, é um *framework web* mais compacto, expressivo e robusto para aplicações *web* e APIs. Ele utiliza funções assíncronas para substituir *callbacks*, melhorando o tratamento de erros. Ao contrário do *Express*, não inclui *middleware* embutido, mas fornece métodos elegantes que simplificam a escrita eficiente de servidores (Koa, 2024).

O *Adonis.js* fornece uma estrutura clara e poderosa para desenvolver aplicações back-end, com suporte integrado para autenticação, *Object-Relational Mapping* (ORM) e outras funcionalidades comuns (adonis, 2024).

Para a construção da API no presente trabalho, foi utilizado o *Node.js* com o *framework Express.js*, visando facilitar e acelerar o desenvolvimento do sistema.

3 DESENVOLVIMENTO DO TRABALHO

3.1 Desenvolvimento do Sistema

No desenvolvimento do sistema, serão abordadas diversas etapas cruciais para o controle e monitoramento eficiente de dispositivos IoT. A fase inicial concentra-se na reconstrução do banco de dados PostgreSQL, utilizado para armazenar e gerenciar informações críticas como configurações de dispositivos, registros de atividades e histórico de operações. A integração contínua com a API HTTP assegura a consistência e disponibilidade dos dados fundamentais para o funcionamento do sistema.

Em seguida, a API HTTP assume um papel central ao facilitar a comunicação entre o aplicativo mobile e os demais componentes do sistema. Esta camada possibilita operações como o gerenciamento de usuários, a manipulação de dados no banco de dados e o envio de comandos para dispositivos IoT através do *broker* MQTT. Além disso, inclui a implementação de um sistema de autenticação via *tokens* JWT para garantir a segurança das interações.

O *broker* MQTT é configurado para viabilizar a troca de mensagens em tempo real entre os dispositivos IoT e o sistema central. Isso envolve a definição de tópicos, assinaturas e publicações de mensagens, assegurando a eficiência na transmissão de dados.

Finalmente, a expansão da interface do usuário, utilizando o *Expo*, proporciona uma experiência intuitiva de controle e monitoramento dos dispositivos IoT. Esta etapa é crucial para garantir que os usuários possam interagir de maneira eficiente e amigável com o sistema.

Para este projeto as ferramentas usadas foram: o *Node-RED* foi utilizado apenas para desenvolver uma *dashboard* que auxilia nos testes de acionamento das portas, ar-condicionado e lâmpadas, além de realizar a leitura dos sensores de umidade, temperatura e tensão; o *Wireshark* também foi empregado nas realizações dos testes para monitoramento dos pacotes dos protocolos de comunicação; o *Visual Studio Code* foi utilizado para programar todo o sistema.

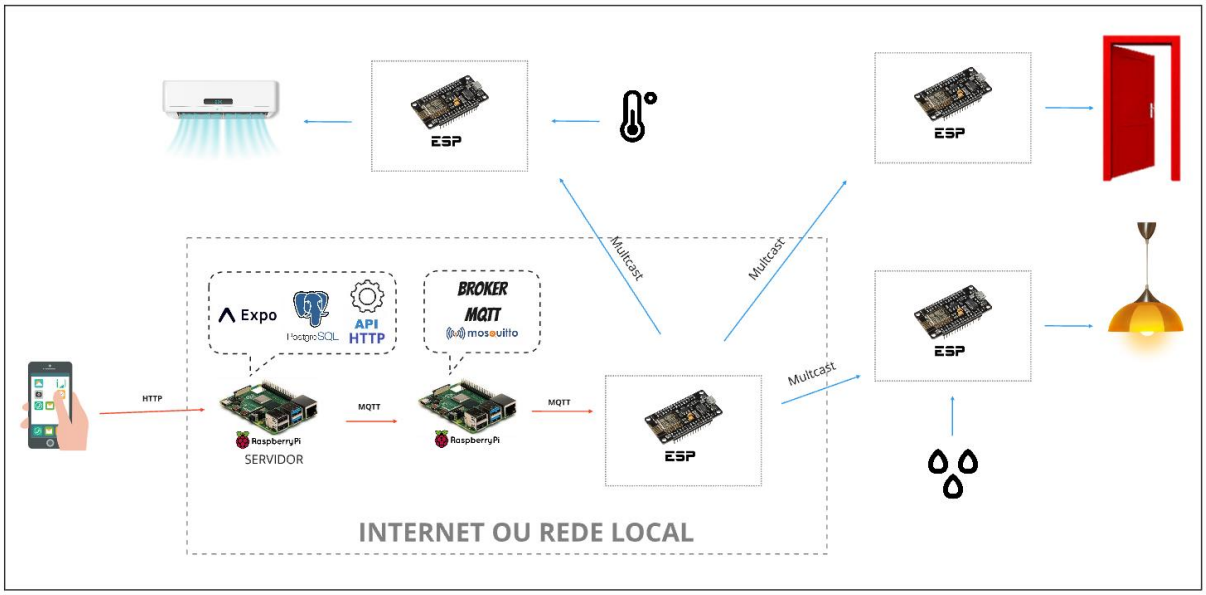
3.2 Arquitetura Geral do Sistema

A arquitetura do sistema foi projetada para permitir a comunicação eficiente e segura entre um aplicativo *mobile*, a API HTTP, dispositivos IoT e um banco de dados relacional. O objetivo é criar uma solução robusta para o controle e monitoramento de dispositivos IoT em tempo real. É relevante apontar que o termo 'monitoramento em tempo real' pode ser aplicado ao monitoramento de temperatura e umidade, pois a dinâmica de variação dessas variáveis é lenta. No caso da tensão, cuja dinâmica de variação é mais rápida, o monitoramento contínuo é limitado pelos códigos embarcados nos *hardwares* de medição, o que não permite o uso pleno do termo 'monitoramento em tempo real', tendo em vista que a coleta é realizada a cada 30 segundos.

A Figura 10 apresenta arquitetura do sistema com seus componentes principais de *hardware*: a *Raspberry* Pi, os aparelhos móveis e os dispositivos IoT. E os de *software*: API HTTP, aplicativo *mobile*, *broker* MQTT e o banco de dados *PostgreSQL*. Ressaltando que o *broker* MQTT está em outra *Raspberry Pi*, mas presente na mesma rede. O aplicativo mobile envia credenciais para a API, que usa JWT para autenticar o usuário e gerar um *token* de acesso. Quando feita uma requisição pela aplicação para se comunicar com os dispositivos IoT do ambiente o aplicativo envia via HTTP dados como tópico e mensagem para a API, que por sua vez envia a mensagem para o tópico requisitado no *broker* MQTT, que repassa para os dispositivos IoT. Os dispositivos respondem com dados de *status* que são encaminhados de volta para a API e armazenados no banco de dados.

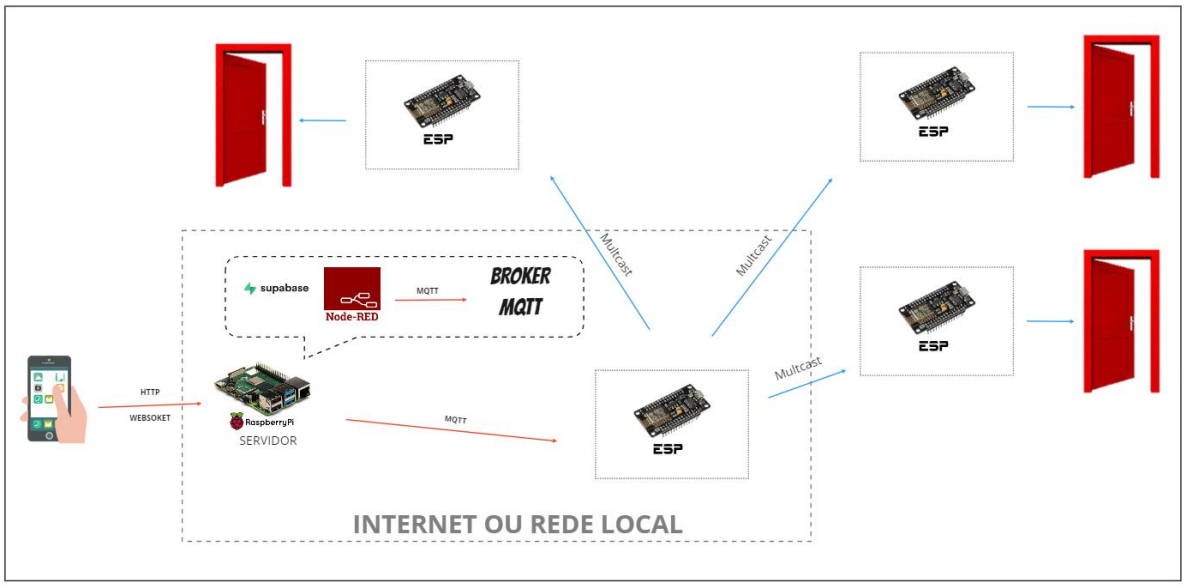
A Figura 11 apresenta a arquitetura do sistema apresentado em (BEZERRA, 2024), que era composta pelo *Supabase* e pelo Node-RED. O *Supabase* atuava como uma plataforma de back-end que facilitava o armazenamento de dados e a autenticação, enquanto o Node-RED funcionava como um intermediário entre a aplicação e o *broker* para realizar a tradução do protocolo *WebSocket* para MQTT.

Figura 10 - Arquitetura geral do sistema de controle e monitoramento de dispositivos IoT.



Fonte: Adaptado de (BEZERRA, 2024).

Figura 11 - Arquitetura geral da aplicação de controle de acesso apresentado em (BEZERRA, 2024).



Fonte: (BEZERRA, 2024).

3.3 Reconstrução e Modelagem do Banco de Dados

O banco de dados do *Supabase* foi reconstruído no PostgreSQL e instalado na *Raspberry Pi 4*. Este novo banco de dados foi desenvolvido para melhorar a eficiência e a capacidade de gerenciamento de dados do sistema. No processo de reconstrução, foram criadas duas novas tabelas no modelo entidade-relacionamento: “*devices*” e “*mqtt_device_history*”. As tabelas antigas “*users*”, “*accessValidation*” e “*environments*” foram mantidas para garantir a continuidade dos dados existentes e a integridade do sistema.

A tabela “*users*” armazena informações sobre os usuários cadastrados no sistema, incluindo identificadores únicos (*uuid*), nomes de usuário, *hashes* de senha, CPF, celular, e-mail, tipo de usuário e a data de criação.

- *uuid*: Identificador único do usuário.
- *username*: Nome de usuário.
- *passhash*: *Hash* da senha do usuário.
- cpf: Cadastro de Pessoa Física.
- *cellphone*: Número de telefone celular.
- *email*: Endereço de e-mail.
- *user_type*: Tipo de usuário (por exemplo, administrador, usuário comum).
- *created_at*: Data de criação do registro do usuário.

A tabela “*accessValidation*” registra as solicitações de acesso feitas pelos usuários, bem como o status atual de aprovação ou não do acesso ao ambiente.

- *uuid*: Identificador único da solicitação de acesso.
- *validation_flag*: Indicador de validação (aprovado ou não).
- *entry_time*: Hora de entrada do usuário.
- *exit_time*: Hora de saída do usuário.
- *environments_id*: Identificador do ambiente ao qual o acesso foi solicitado.

- *users_uuid*: Identificador do usuário que solicitou o acesso.
- *access_type*: Tipo de acesso (por exemplo, entrada, saída).
- *created_at*: Data de criação do registro de solicitação.

A tabela “*environments*” armazena dados sobre os diferentes ambientes monitorados. No estudo apresentado em (BEZERRA, 2024), o tópico MQTT da tranca eletrônica da porta era registrado nesta tabela. No entanto, neste trabalho, os tópicos foram transferidos para a tabela devices para melhor organização e flexibilidade.

- *id*: Identificador único do ambiente.
- *name*: Nome do ambiente.
- *mqtt_topic*: Tópico MQTT associado a porta do ambiente.
- *message*: Mensagem associada a porta do ambiente.
- *isActive_flag*: Indicador se o ambiente está ativo.
- *created_at*: Data de criação do registro do ambiente.

A nova tabela “*devices*” armazena tópicos de todos os dispositivos IoT, incluindo trancas eletrônicas de portas, lâmpadas, ar-condicionado, sensores de temperatura, umidade e tensão. A separação dos tópicos nesta tabela facilita a adição de novos dispositivos aos ambientes.

- *device_topic*: Tópico MQTT do dispositivo.
- *environments_id*: Identificador do ambiente associado ao dispositivo.
- *enable*: Indicador se o dispositivo está habilitado.
- *uuid*: Identificador único do dispositivo.
- *created_at*: Data de criação do registro do dispositivo.

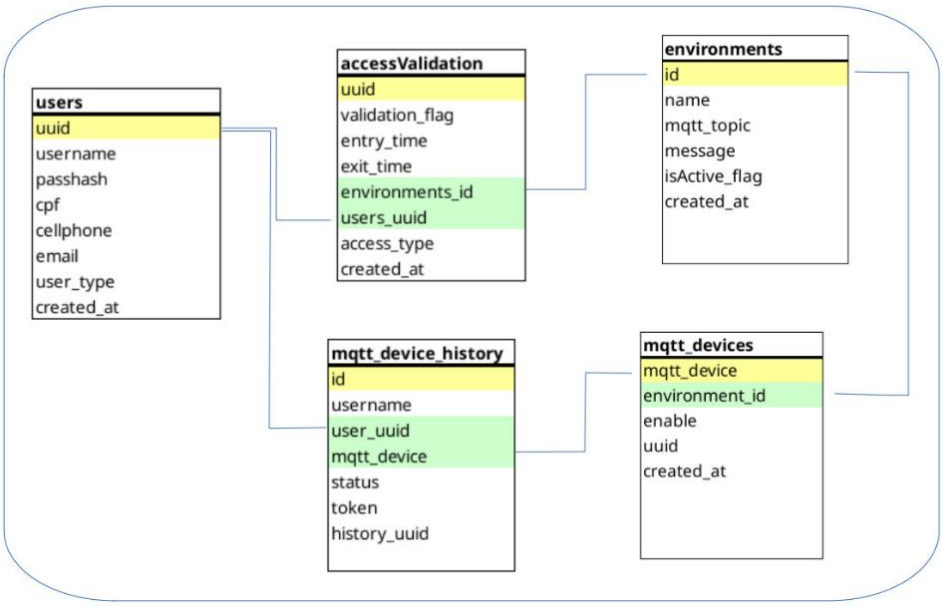
A nova tabela “*mqtt_device_history*” é responsável por registrar todas as requisições feitas pelos usuários através do aplicativo mobile. Esta tabela armazena

identificadores únicos, nomes de usuário, *uuids* de usuários, dispositivos MQTT associados, *status*, *tokens* e identificadores de histórico.

- *id*: Identificador único do histórico de dispositivo MQTT.
- *username*: Nome de usuário que fez a requisição.
- *user_uuid*: Identificador único do usuário que fez a requisição.
- *mqtt_device*: Dispositivo MQTT associado.
- *status*: Status da requisição (*on/off)*.
- *token*: Token de autenticação utilizado.
- *history_uuid*: Identificador único do histórico de requisição.

A Figura 12 ilustra o modelo entidade-relacionamento do banco de dados reconstruído. Ela mostra as relações entre as tabelas, destacando como “*users*” se relaciona com “*accessValidation*”, que por sua vez está vinculada a “*environments*”. As novas tabelas, “*devices*” e “*mqtt_device_history*”, também estão interligadas às tabelas existentes, demonstrando a integração contínua e a expansão do banco de dados para acomodar novos requisitos funcionais. Os atributos grifados em amarelo representam as chaves primárias (PK) e os em verde as chaves estrangeiras (FK) dos relacionamentos. Os SQL gerados ao criar as tabelas encontram-se no apêndice A.

Figura 12 - Relacionamento de entidades.



Fonte: O'Autor.

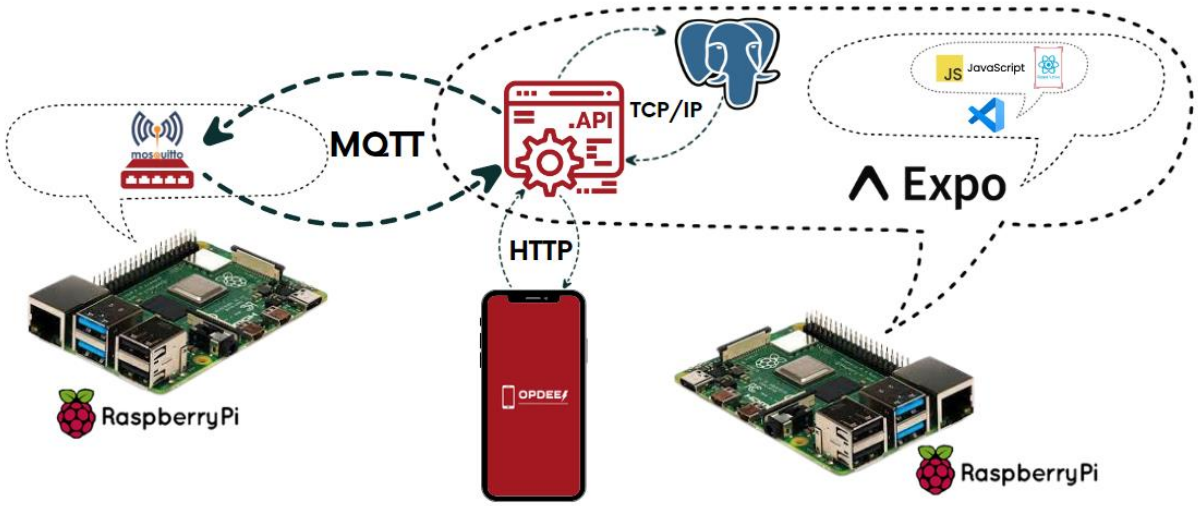
Como citado anteriormente, os atributos "*mqtt_topic*" e "*message*" da entidade "*environments*" deixaram de ser utilizados conforme o desenvolvimento do projeto, para possibilitar a escalabilidade de dispositivos IoT por ambiente. Os atributos "*entry_time*"e "e*xit_time*" armazenam a faixa de horário que o usuário pode acessar o ambiente, porém, como apresentado em (BEZERRA, 2024), neste trabalho também não houve a oportunidade de implementação desta restrição devido ao curto tempo de desenvolvimento e o foco no elemento principal do trabalho estar atrelado ao desenvolvimento da API HTTP priorizando a comunicação efetiva e eficiente com o *broker* MQTT e a otimização dos recursos da *Raspberry Pi*. O mesmo para o atributo "*user_type*" para sinalizar e diferenciar administrados do sistema, que devem ter acesso como coordenador em todos os ambientes e poder alterar configurações, de usuários comuns. Ambas as funcionalidades e utilização dos atributos ficam como sugestão para proposta de continuidade.

3.4 Desenvolvimento da API HTTP e Comunicação com HTTP e MQTT

Após reestruturar o banco de dados, foi iniciado o desenvolvimento da API HTTP com método *POST* e *endpoints*. Foi criado um módulo "*settings.js*" para armazenar as configurações do software, incluindo informações sobre o autor, além de descrever sua função. As configurações compõem detalhes de conexão para o banco de dados *PostgreSQL*, como usuário, senha, *host*, nome do banco de dados e porta, como também para o *broker* MQTT, incluindo endereço IP e porta. Foi a maneira mais organizada de definir e exportar configurações importantes, a centralização facilita a modificação dessas configurações em um único lugar, o que é útil para o este projeto visando a complexidade por várias partes do sistema precisarem acessar essas informações.

A conexão com o *broker* MQTT é realizada no arquivo principal da API o "*app.js*" utilizando as configurações armazenas no arquivo "*settings.js*" através da biblioteca "*mqtt.js*". Após iniciada a conexão para escutar o *broker* na porta 1883 a API se inscreve em todos os tópicos cadastrados no banco de dados. O arquivo principal contém as rotas, *endpoints*, criadas para o funcionamento da API para realizar as consultas no banco de dados *PostgreSQL* e publicar no *broker*, também contém a variável global que foi desenvolvida para armazenar a última mensagem publicada em cada tópico do *broker*. A Figura 13 - Comunicação entre API HTTP, banco de dados *PostgreSQL* e *broker* MQTT.Figura 13 exemplifica como e quais protocolos são utilizados para a comunicação entre a API, o banco de dados e o *broker* MQTT.

Figura 13 - Comunicação entre API HTTP, banco de dados *PostgreSQL* e *broker* MQTT.



Fonte: O'Autor.

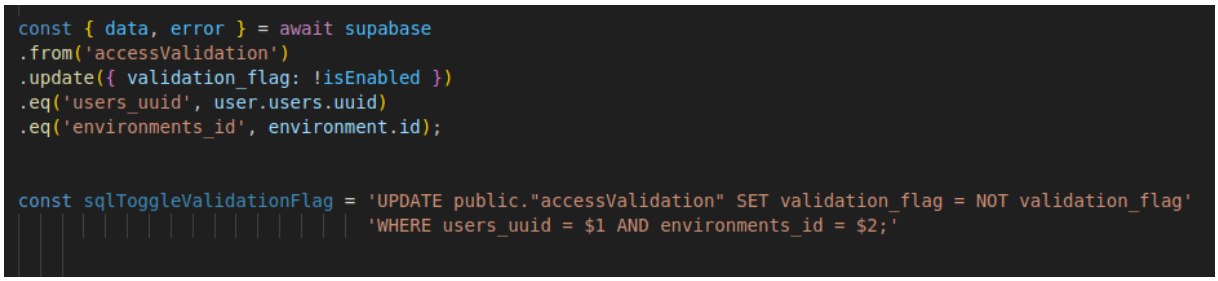
Foi criada uma função auxiliar "*doQuery"*, disponível no módulo "*sqlQuery.js*", com os parâmetros "*sqlComand*" e "*arrArguments*", respectivamente, um para armazenar a consulta SQL que deseja ser feita no banco de dados *PostgreSQL* e outro com o *array* de argumentos que deseja inserir nesta consulta. A função importa na constante "*db*" as configurações do banco de dados utilizando, também, o arquivo "*settings.js*". O método "query" que recebe dois parâmetros citados anteriormente. Essa função tem extrema importância no funcionamento do sistema como um todo, pois ela configura a consulta a partir dos argumentos fornecidos, possibilitando sua reutilização em todas as requisições destinadas ao banco de dados. A Figura 14 exibe o código desenvolvido no módulo "*sqlQuery.js*" explicado neste parágrafo.

Figura 14 - Código desenvolvido no módulo *“sqlQuery.js”*.

Fonte: O’Autor.

Em seguida, iniciou-se a substituição das consultas ao banco de dados desenvolvidas em (BEZERRA, 2024) pelo *Supabase*. A estrutura das consultas foi mantida, sendo convertido somente a escrita da consulta com as funções definidas pelo *Supabase* para consultas na linguagem SQL. Na Figura 15 apresenta um exemplo desta conversão.

Figura 15 - Comparação conversão da consulta pelo *Supabase* e pela API.



Fonte: O’Autor.

Após converter a consulta foi criado os *endpoints* e os módulos para ação, podendo conter apenas uma ou mais consultas. Foi definido quais argumentos era necessário individualmente para cada consulta para serem inseridos no SQL substituindo os cifrões ($). Vale salientar que cada cifrão acompanha um número que representa a posição do dado no *array* do argumento.

Cada módulo contém sua função para receber os argumentos e executar a função “*doQuery*” para compor a consulta SQL, comunicar com o banco de dados e retorna a resposta da consulta realizada. Como demonstrado na Figura 16, observa-se que a função “*byUserandEnvironment*” possui dois parâmetros o "*req*” que consiste no *array* de argumentos enviado na requisição feita pelo aplicativo *mobile* e o “*res*” que armazena a resposta da consulta e devolve para a aplicação executar suas rotinas. Todos os módulos seguem o mesmo padrão de funcionamento e possuem tratamento de erro.

Figura 16 - Exemplo de estrutura da requisição de consulta da API ao banco de dados.

```
> controllers > JS toggleValidationFlag.js > byUserandEnvironment > byUserandEnvironment
const successStatus = 200

const failStatus = 401

const failMessage = 'Falha ao modificar o acesso do usuário.'

const { doQuery } = require('../helpers/sqlQuery')

const sqlToggleValidationFlag = 'UPDATE public."accessValidation" SET validation_flag = NOT validation_flag WHERE users_uuid = $1 AND environments_id = $2;'


exports.byUserandEnvironment = async (req, res) => {

    const { user_uuid, environments_id } = req.body

    params = [user_uuid, environments_id]

    const toggleSwitch = await doQuery(sqlToggleValidationFlag, params)

    const error = (toggleSwitch == "Erro ao consultar o banco de dados")

    if (error) res.status(failStatus).send({ message: failMessage })

    const response = { toggleSwitch }

    res.status(successStatus).send(response)

console.log("response", response)

}
```

Fonte: O’Autor.

Em seguida, foram desenvolvidos outros módulos e consultas, como verificar se o usuário existe, listar os ambientes controlados pela API, checar os ambientes em que o usuário tem acesso, entre outros.

Para publicar em um tópico no *broker* MQTT foi desenvolvido um módulo específico, que apesar de algumas alterações segue o mesmo padrão dos módulos direcionados a comunicar-se com o banco de dados. O módulo "*mqttPublish*" é utilizado apenas quando o usuário desejar acionar as trancas eletrônicas das portas, lâmpadas e ar-condicionados dos ambientes.

Vale ressaltar o funcionamento de tal dispositivos. Para as trancas eletrônicas foi configurado que ao receber a mensagem "*on*" em seus tópicos no *broker* elas seriam acionadas ativando a tranca para abrir e após 5 segundos fechar, portanto esses dispositivos possuem apenas uma mensagem para ser publicada, não havendo a necessidade de "desligá-los". Para as lâmpadas e ar-condicionados é necessário ligar e desligar os dispositivos, a solução encontrada para que apenas um botão executasse as duas funções foi a implementação de uma sub-rotina dentro do módulo "*mqttPublish*".

Essa sub-rotina recebe o tópico e a mensagem enviadas pela aplicação. É enviado a mensagem "*on*" quando se trata de trancas eletrônicas e "*change*" quando são acionadas lâmpadas ou ar-condicionados. Se a sub-rotina receber a mensagem "*change*" ela consulta a última mensagem do estado do dispositivo publicado em seu tópico, que está armazenada na variável global e publica no *broker* a mensagem inversa, utilizando o *QoS 2* para garantir que a mensagem seja entregue apenas uma vez, portanto se a última mensagem publicada foi "*on*" (ligar) ela publicará "*off*" (desligar) atribuindo o estado inverso ao dispositivo.

Como já mencionado a API está configurada para escutar o *broker* MQTT na porta 1883 e a cada 30 segundos os dados dos sensores são coletados e publicados no *broker*. A API, ao receber essas publicações, armazena os valores no tópico correspondente na variável global.

Para simular os dados dos sensores de tensão temperatura e umidade foi projetada a função "*getRandomIntegerInclusive*" em um módulo a parte e executada dentro da função "*getGlobal*" que é acionada no módulo principal da API. Seu objetivo é gerar um número inteiro aleatório dentro de um intervalo específico, determinado pelos parâmetros mínimo, "*min*" e máximo, "*max*". Para auxiliar no desenvolvimento foi utilizada a biblioteca "*Math*", mais precisamente as funções "*random*" e "*floor".* A função é apresentada na Figura 17 junto com exemplos utilizados durante os testes.

Figura 17 - Função getRandomIntegerInclusive e exemplo de uso.

```
const getRandomIntegerInclusive = (min, max) =>
Math.floor(Math.random() * (max-min+1)) + min

exports.getGlobal = async (req, res) => {

    const response = { ...global.data }

    response['CLP1/humidade'] = getRandomIntegerInclusive(750, 840)/10

    response['CLP1/temperatura']= getRandomIntegerInclusive(260, 320)/10

    response['CLP1/tensao']= getRandomIntegerInclusive(2180, 2220)/10

    res.status(successStatus).send(response)
}
```

Fonte: O'Autor.

A função *"Math.random()"* é uma função nativa do JavaScript que retorna um número decimal pseudoaleatório no intervalo [0, 1). Para ajustar o intervalo para incluir números inteiros de "*min*" até "*max*", foi utilizada a expressão:

$$Math.random() * (\max - \min + 1) \tag{3.1}$$

Isso multiplica o número aleatório gerado pelo comprimento do intervalo desejado, mais 1. Já a função "*Math.floor()*" foi usada para arredondar para baixo o número resultante da multiplicação. Isso garante que o resultado seja um número inteiro e por fim e adicionar do valor "*min*" para que o número aleatório esteja dentro dos valores de máximo e mínimo definidos. Nos exemplos os números inteiros escolhidos para máximo e mínimo foram multiplicados por 10 antes de serem enviados para a função e divididos por 10 após o chamar a função para obter uma casa decimal.

É relevante destacar que toda a API está modularizada, o que é crucial para a organização e reutilização dos componentes desenvolvidos. Seguindo essa lógica de estruturação, os módulos foram distribuídos em pastas específicas. A pasta "*helpers*"

contém módulos auxiliares, como o que inclui a função "*doQuery*". A pasta "*controllers*" reúne os módulos responsáveis pelas requisições e consultas ao *broker* MQTT e ao banco de dados *PostgreSQL*. Por fim, a pasta "*config*" inclui o módulo "*settings.js*", que armazena as configurações gerais da API.

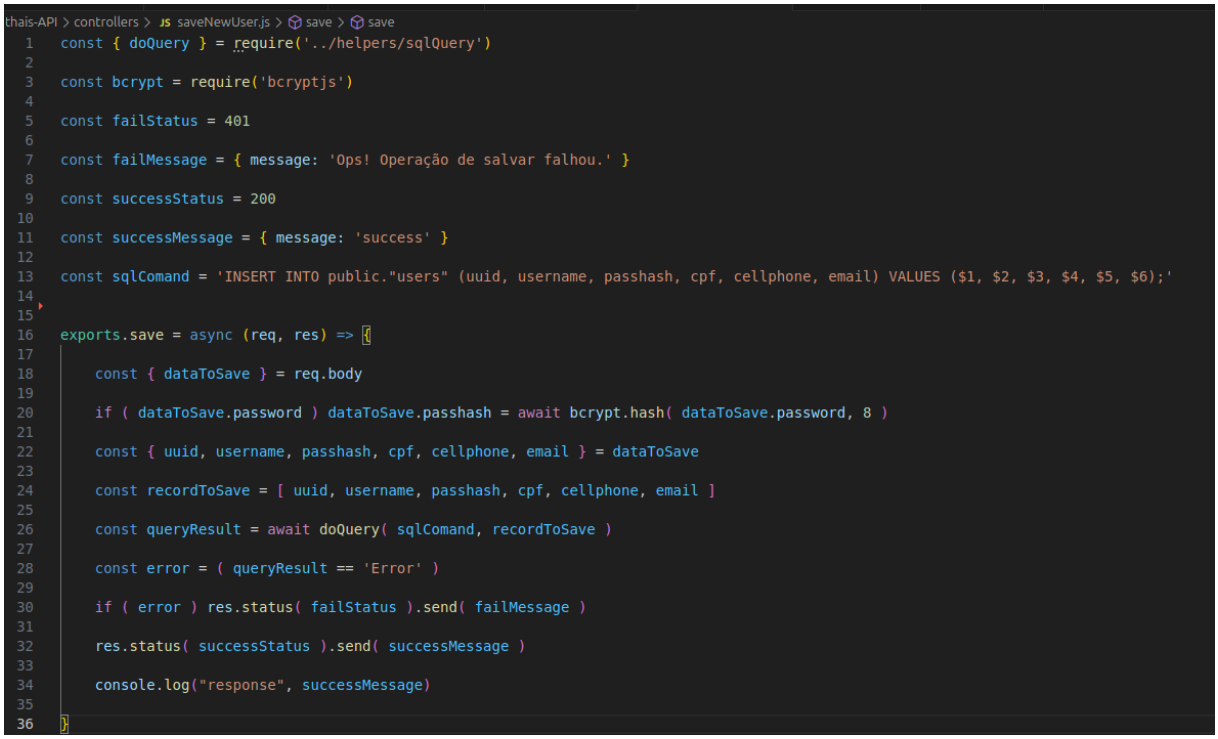
3.5 Arquitetura e Desenvolvimento da Segurança do Sistema

O esquema de segurança construído está focado no elemento central ao sistema desenvolvido, sendo a API HTTP que ocupa esta posição, bem como o equipamento que o hospeda, a *Raspberry PI 4*. Por sua característica central, o controle de acesso foi implementado na API como também as comunicações de autenticação e uso do banco de dados e do *broker* MQTT. Desta forma, todas as requisições de operações feitas através do *software* dos aparelhos móveis são encaminhas através de pacote HTTP do tipo *POST*, sendo estas submetidas a uma rotina de autenticação para autorização do acesso.

Inicialmente, o cadastro de usuários se deu através de uma requisição vinda do aparelho móvel do usuário para a API, contendo os dados de registro inicial do usuário como também o “*username*”, a “password” e o identificador único do celular “*uuid*”.

Na API, o processamento da inscrição do usuário em um novo ambiente se dá através da rotina de salvamento de um novo usuário no banco de dados, acessível pelo *endpoint* “*/saveNewUser*”. Durante essa operação, é verificado se já há um registro do celular e do usuário no banco de dados. Se não houver, inicia-se o processo de cadastro do novo usuário. Nesse processo, a senha fornecida é convertida em um *hash* através da biblioteca de criptografia *Bcrypt* utilizando a função “*bcrypt.hash*” antes de ser armazenada no registro. É importante destacar que a senha original não é armazenada no banco de dados, garantindo segurança. A Figura 18 apresenta a execução da função para cryptografar a senha e salvar os demais dados do usuário.

Figura 18 - Módulo "*SaveNewUser*" que utiliza a biblioteca "*bcrypt*" para gerar o hash da senha.



```
thais-API > controllers > saveNewUser.js > save > save
1   const { doQuery } = require('../helpers/sqlQuery')
2
3   const bcrypt = require('bcryptjs')
4
5   const failStatus = 401
6
7   const failMessage = { message: 'Ops! Operação de salvar falhou.' }
8
9   const successStatus = 200
10
11  const successMessage = { message: 'success' }
12
13  const sqlComand = 'INSERT INTO public."users" (uuid, username, passhash, cpf, cellphone, email) VALUES ($1, $2, $3, $4, $5, $6);'
14
15
16  exports.save = async (req, res) => {
17
18      const { dataToSave } = req.body
19
20      if ( dataToSave.password ) dataToSave.passhash = await bcrypt.hash( dataToSave.password, 8 )
21
22      const { uuid, username, passhash, cpf, cellphone, email } = dataToSave
23
24      const recordToSave = [ uuid, username, passhash, cpf, cellphone, email ]
25
26      const queryResult = await doQuery( sqlComand, recordToSave )
27
28      const error = ( queryResult == 'Error' )
29
30      if ( error ) res.status( failStatus ).send( failMessage )
31
32      res.status( successStatus ).send( successMessage )
33
34      console.log("response", successMessage)
35
36  }
```

Fonte: O'Autor.

Após o registro do novo usuário, é necessário que um usuário habilitado, como um coordenador do ambiente solicitado, conceda a autorização. Esse procedimento adiciona um elemento de segurança, uma vez que a supervisão da operação é realizada por um componente humano.

A requisição de acesso realizada através do *endpoint* "*/login*" na API é composta do "*username*" do usuário, a senha de acesso, "*password*", e do número identificador único do aparelho móvel, "*uuid*". O processamento da requisição se dá com a consulta do banco de dados de um registro que tenha o "*uuid*" do aparelho. Em seguida compara o "*username*", se esses dois primeiros passos forem atendidos a senha é fornecida para a função "*bcrypt.compare*", da biblioteca de criptografia utilizada *Bcrypt*, que verifica se a senha tem correspondência com o *hash* armazenado no registro do usuário no banco de dados. Se a operação for bem sucedida, com o uso da função "*jwt.sign*", com o parâmetro *claim exp,* da biblioteca *JsonWebToken* é gerado e devolvido como resposta um *token* JWT assinado com os dados do usuário e uma senha secreta utilizada pela API que foi definida no início do desenvolvimento. O *token*

e o “*username*” do usuário são armazenados no aparelho móvel através da biblioteca “*expo-secure-store*”, ficando disponível para ser acessado em qualquer tela da aplicação e sendo deletados do aparelho ao realizar *logout* ou fechar o aplicativo.

É importante destacar que foi configurado um tempo para que o token seja expirado. Passado esse tempo o usuário necessitará realizar o login novamente na aplicação. O uso das funções “*bcrypt.compare*” e “*jwt.*sign” são apresentados na Figura 19 por motivos de segurança algumas informações foram retiradas da imagem.

O *token* JWT gerado será o autenticador das operações daí em diante, conferindo a segurança necessária por sua composição, visto que vincula senha criada para a API ao aparelho celular pelo uso de seu “*uuid*” e o usuário através do seu “*username*”, “*password*” e também ao “*uuid*” do aparelho. Assim, o acesso às operações precisa deste token que é gerado no processo de login.

Figura 19 - Módulo de autenticação que gera o *token* JWT e compara o *passhash* com a senha inserida

```
const jwt = require('jsonwebtoken')

const bcrypt = require('bcryptjs')

const { doQuery } = require('../helpers/sqlQuery')

const sqlGetUserRecord = 'SELECT * FROM public."users" WHERE uuid=$1 and username=$2 limit 1;'

const sqlGetUserAccessValidation = 'SELECT * FROM public."accessValidation" WHERE users_uuid = $1 AND validation_flag = true;'

//~ ..............................""

exports.login = async (req, res) => {

    const { username, password, deviceId } = req.body

    const passwordFail = ! ( await (bcrypt.compare(password, userRecord.passhash)) )

    if (passwordFail) return res.status(failStatus).send({ message: failMessage })

    const token = jwt.sign({ id: userRecord.uuid }, systemSecretKey, { expiresIn: expirationTime })

    const accessValidations = await doQuery(sqlGetUserAccessValidation, [deviceId])

    const response = { token, userRecord, accessValidations }

    res.status(successStatus).send(response)

    logger('response', response)

}
```

Fonte: O’Autor.

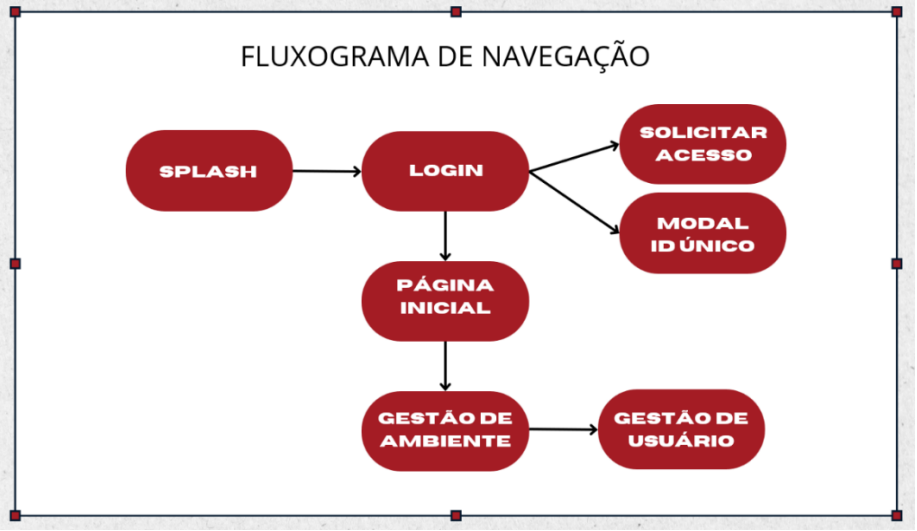
3.6 O Aplicativo Móvel Sob a Ótica do Usuário

Apesar das modificações e implementação de novas funcionalidade no aplicativo *mobile*, seu fluxo de navegação principal foi mantido semelhante ao apresentado em (BEZERRA, 2024). Foi retirado apenas a tela modal. O fluxograma do trabalho anterior é apresentado na Figura 20 e o fluxograma deste trabalho atual na Figura 21.

Em (BEZERRA, 2024), o cadastramento foi pensado para não utilizar usuário e senha, pois foi verificado que podia ser gerado um número identificador do aparelho celular. Como esse dado é único e vinculado ao aparelho e a aplicação, poderia ser utilizado como elemento chave, identificador do celular e, por conseguinte, do usuário.

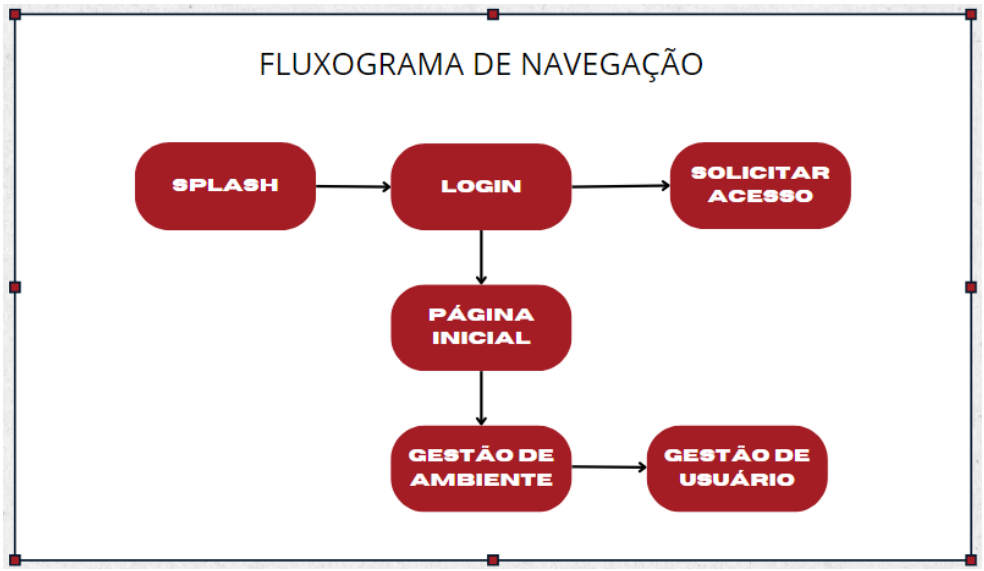
Inicialmente, podia-se pensar que a alternativa de utilizar este número identificador gerado facilitaria o processo, eliminando a necessidade de digitar senha e nome do usuário. Contudo, o uso de tal funcionalidade envolve um dado externo ao sistema, sobre o qual não se tem controle quanto à possibilidade de ser modificado ou clonado. Além disso, é considerado um dado “sensível”, pois é utilizado como medida de segurança na aplicação. Por essa razão, o botão que exibia o ID do aparelho foi retirado do aplicativo, assim como o campo que o exibia no formulário de cadastro.

Figura 20 - Fluxograma de navegação do aplicativo apresentado em (BEZERRA, 2024).



Fonte: (BEZERRA, 2024).

Figura 21 - Fluxograma de navegação do aplicativo deste trabalho.

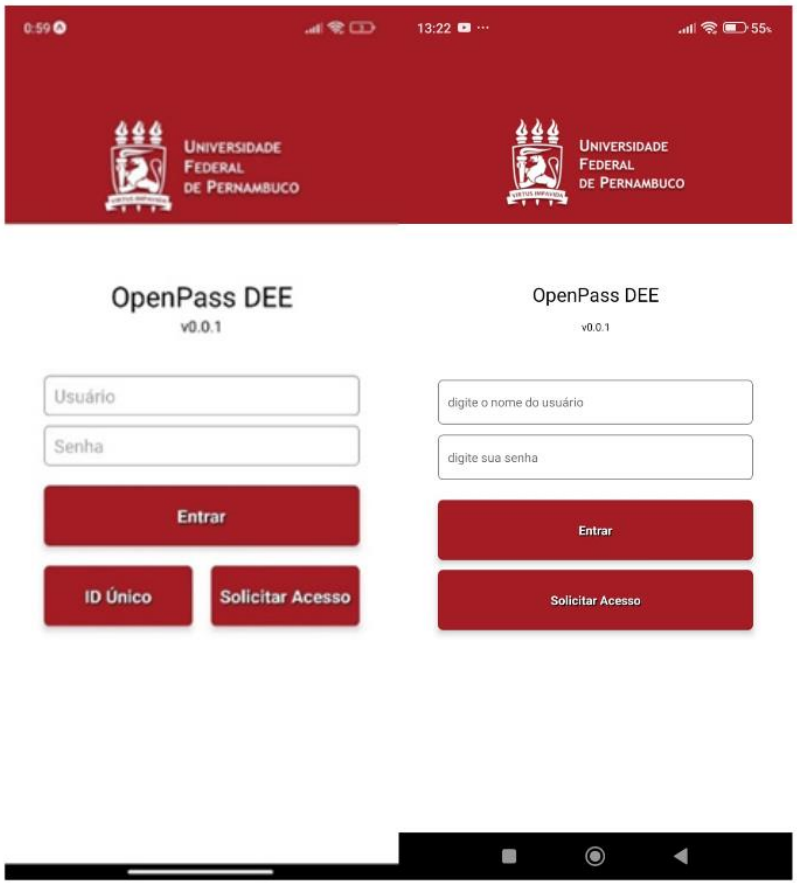


Fonte: Adaptado de (BEZERRA, 2024).

A tela de “*Splash*” que é a tela de abertura da aplicação se manteve a mesma sem alterações como mostrado na Figura 44. As telas que não foram modificadas estão exibidas no Anexo A.

Na tela de “*Login*” foi realizada a retirada do botão de ID Único por exibir uma informação sensível como justificado anteriormente e a liberação para preenchimento dos campos de usuário e senha. A comparação entre a tela antes e após a modificação podem ser observadas na Figura 22. Se o usuário ainda não possui cadastro, a alternativa de se cadastrar é através do botão “Solicitar Acesso”.

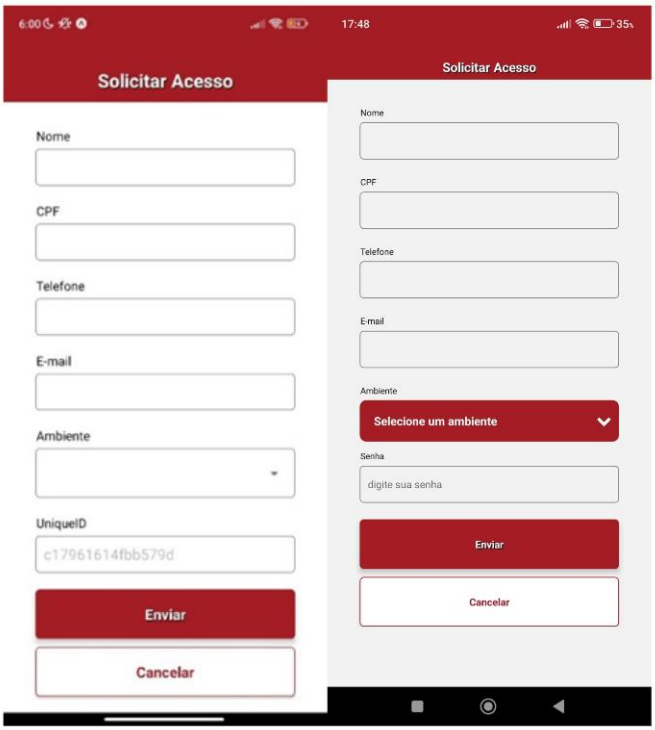
Figura 22 - Comparação entre as telas de *Login* antes e depois das modificações deste trabalho.



Fonte: Adaptado de (BEZERRA, 2024).

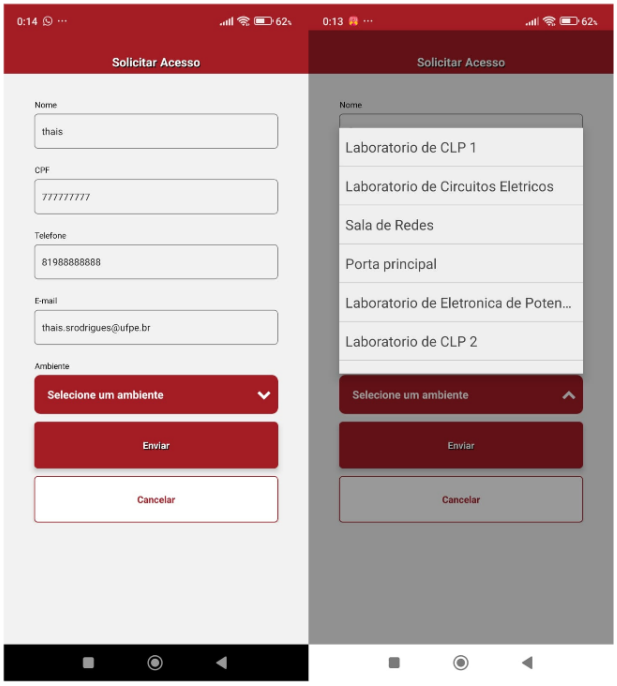
Na tela de “Solicitar Acesso”, foi retirado o campo de “*UniqueID*”, pela mesma razão da tela modal de ID Único, também foi adicionado um campo de entrada para a senha e modificando o estilo do menu *dropdown* já implementado em (BEZERRA, 2024), como mostrado na Figura 23. O funcionamento da tela segue o mesmo que o apresentado em (BEZERRA, 2024), caso o usuário já esteja cadastrado os campos apareceram preenchidos sendo necessário apenas escolher o novo ambiente que deseja ter acesso. Nota-se que o campo de senha não aparece como apresentado Figura 24, pois a senha foi previamente cadastrada.

Figura 23 - Comparação entre as telas de Solicitar Acesso.



Fonte: Adaptado de (BEZERRA, 2024).

Figura 24 - Tela de Solicitar Acesso autopreenchida para usuário previamente cadastrado e lista exibida no menu *dropdown*.

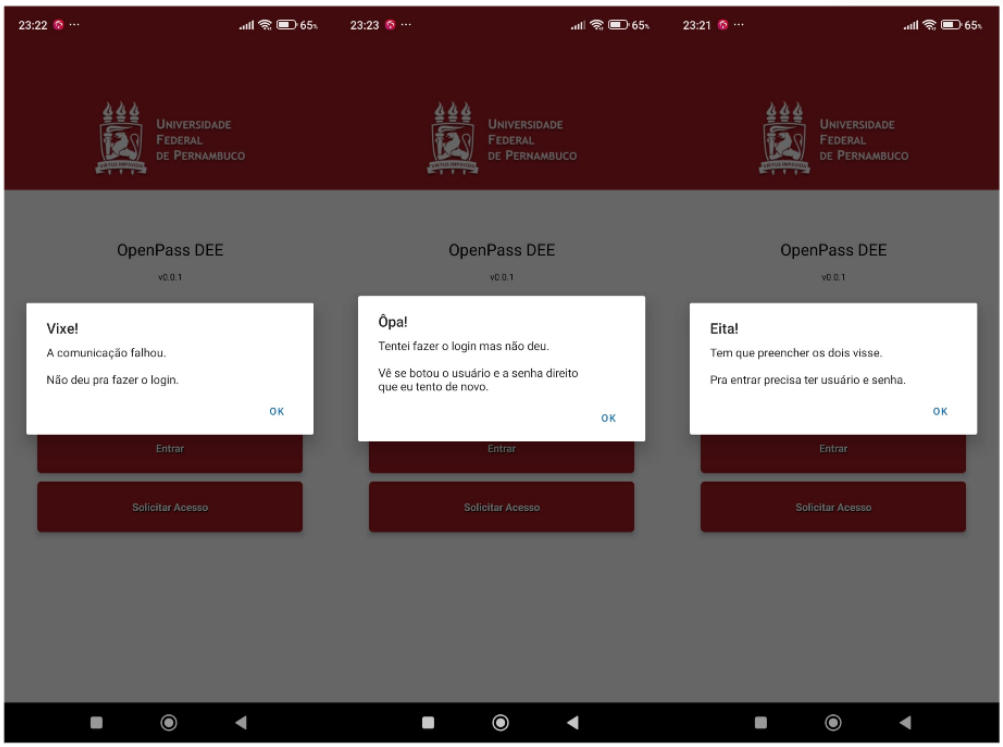


Fonte: O'Autor.

As operações de acesso e cadastro incluíram um elaborado tratamento de erros, conforme ilustrado nas Figura 25 e Figura 26. Esses tratamentos cobriram tanto os erros causados pelos usuários, como a falta de preenchimento de campos obrigatórios e senhas ou nomes de usuário incorretos, quanto falhas na infraestrutura, como problemas de comunicação na rede, com a API, com o banco de dados ou com o broker.

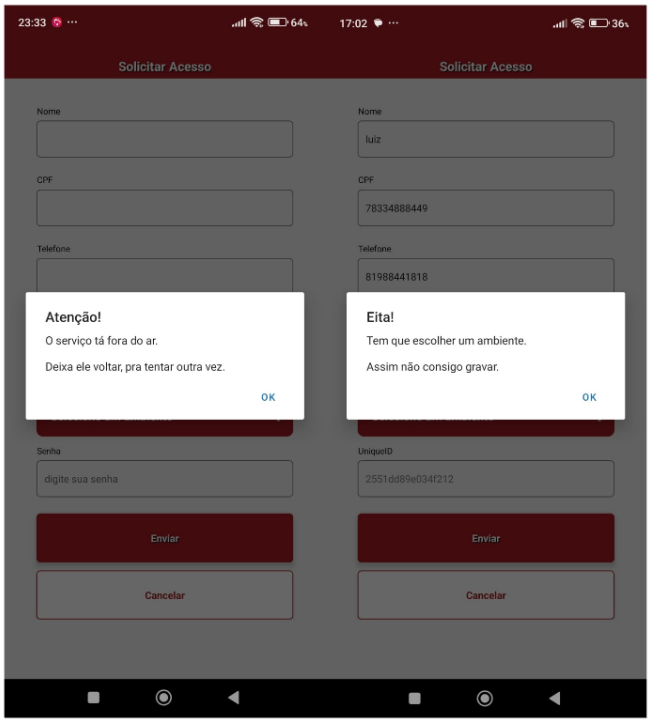
Para a comunicação de erros e falhas, bem como para os resultados de sucesso de algumas operações, foram utilizados cartões de mensagens (modal cards). Seguindo a tendência dos aplicativos móveis atuais, foi empregada uma linguagem informal com características regionais, visando proporcionar um acolhimento ao usuário que operará o sistema.

Figura 25 - Tratamento de erro nas requisições de *login*.



Fonte: O'Autor.

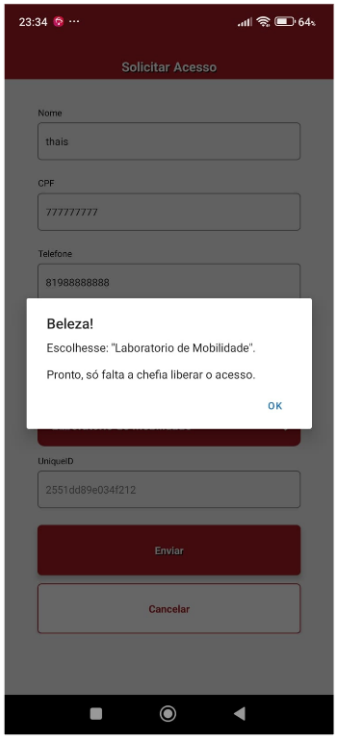
Figura 26 - Tratamento de erro nas requisições de cadastro.



Fonte: O’Autor.

Após o cadastro, é necessário que um usuário coordenador do ambiente habilite o novo usuário nas áreas de acesso que ele escolheu como mostrado na Figura 27, para que os controles disponíveis do ambiente pretendido possam ser exibidos na tela de “Ambientes”. Assim que concedida a permissão ao ambiente a tela será preenchida com os controles disponíveis e as leituras dos sensores, como apresentado na Figura 28.

Figura 27 - Comunicação de sucesso da requisição de cadastro.



Fonte: O'Autor.

Figura 28 - Tela de Ambientes com os ambientes que o usuário tem acesso.

Fonte: O'Autor.

Quando o usuário for atribuído ao ambiente como coordenador, torna-se visível o botão para acessar a tela de “Configuração de Ambientes”. O botão é representado pelo desenho de uma engrenagem como pode ser observado na Figura 29.

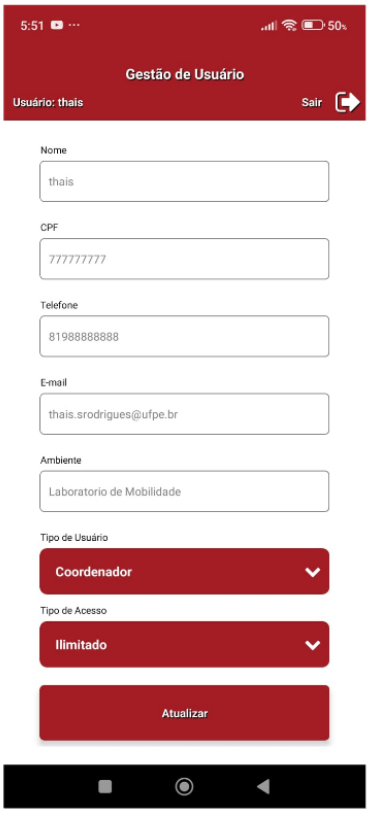
Figura 29 - Tela de Ambientes com usuário coordenador.

Fonte: O’Autor.

A tela “Configuração de Ambientes” não foi modificada mantendo o funcionamento e *layout* do trabalho apresentado em (BEZERRA, 2024) mostrada na Figura 45, localizada no anexo A.

A tela de “Gestão de Usuário” passou por uma modificação semelhante a de “Solicitação de Acesso”, houve a retirada do campo de UniqueID e a adição de um menu dropdown para modificar o tipo de acesso que o usuário tem a aquele ambiente, como usuário ou coordenador como mostrado na Figura 30.

Figura 30 - Tela de Gestão de Usuário.



Fonte: O'Autor.

3.7 Expansão e Desenvolvimento da Comunicação do Aplicativo *Mobile*

Nesta seção serão abordadas as novas funcionalidades do aplicativo *mobile*. A primeira grande mudança foi na forma de efetuar as consultas ao banco de dados deixando de utilizar a consulta pelo *Supabase* para utilizar a função "*requestToAPI*".

A função assíncrona "*requestToAPI"* utilizando o método *POST* foi desenvolvida para realizar a requisição HTTP para a API HTTP.

Dentro da função "*requestToAPI"*, foram definidos dois parâmetros:

- *API*: que representa o *endpoint* que recebe a requisição na API.
- *bodyContent*, que contém o conteúdo a ser enviado no corpo da requisição formatado como JSON.

A função cria um objeto vazio chamado "*requestContent"* para armazenar os detalhes da requisição, como método (*POST*) e cabeçalhos (*Content-Type:*

application/json). Também foi definida uma mensagem de erro padrão, "*errorMessage*", para ser exibida caso ocorra uma falha na comunicação com a API.

Posteriormente, a função verifica se há algo para ser enviado no corpo da requisição. Se houver, converte o conteúdo para JSON e o adiciona ao objeto "*requestContent"*. Então a função tenta executar a requisição usando "*fetch*", uma função nativa do *JavaScript* para realizar requisições HTTP assíncronas. Esta requisição é feita para o *endpoint* especificado da API usando os parâmetros definidos anteriormente em "*requestContent*".

Durante a tentativa de envio da requisição, o código verifica o status da resposta usando "*response.status*", uma classe nativa do *JavaScript*. Se a requisição for bem-sucedida, o código converte o conteúdo da resposta de JSON para um objeto JavaScript usando "*response.json()*". Caso ocorra algum erro durante a execução da requisição, como problemas de rede ou erros internos no servidor, o catch captura o erro e retorna como resposta da função. Esse erro foi tratado no código e é exibido para o usuário em forma de alerta.

Em resumo, a função "*requestToAPI"* foi projetada para enviar requisições *POST* para uma API, lidando com sucesso e falhas na comunicação de forma adequada, registrando informações relevantes no console para facilitar o monitoramento e diagnóstico de problemas durante o desenvolvimento e uso da API.

Ao longo do desenvolvimento, a função foi adaptada para enviar o token de segurança para todas as requisições feitas através dela, esta adaptação foi nomeada de "*requestToAPInova*". O *token* é acessado através do carregamento dos dados do armazenamento realizado no login. Ao final da adaptação, identificou-se a necessidade de criar outra função semelhante, sem o carregamento do token, para fazer a requisição dos ambientes para alimentar o menu *dropdown* de ambientes na tela de solicitação de acesso, pois o *token* é gerado somente após o login do usuário, esta foi nomeada como "*requestToAPInoToken*". Todas as versões da "*requestToAPI*" são chamadas utilizando os mesmos parâmetros como demonstrado na Figura 31.

Figura 31 - Comparação de parâmetros das funções de requisição desenvolvidas.

requestToAPI.js

```
import { log } from './log'

export const requestToAPI = async (API, bodyContent) => {

    const requestContent = {}

    requestContent.method = 'POST'

    requestContent.headers = { 'Content-Type': 'application/json' }

    errorMessage = 'Ops! Houve um falha na comunicação.'
```

requestToAPInova.js

```
import { log } from './log'
import { psiu } from './psiu'
import { getDeviceId } from './getDeviceId'
import storage from '../storage/expoSecStore'

export const requestToAPInova = async (API, bodyContent) => {

    const requestContent = {}

    requestContent.method = 'POST'

    requestContent.headers = { 'Content-Type': 'application/json' }

    errorMessage = 'novaOps! Houve um falha na comunicação.'
```

requestToAPInoToken.js

```
import { log } from './log'


export const requestToAPInoToken = async (API, bodyContent) => {

    const requestContent = {}

    requestContent.method = 'POST'

    requestContent.headers = { 'Content-Type': 'application/json' }

    errorMessage = 'novaOps! Houve um falha na comunicação.'
```

Fonte: O’Autor.

As mencionadas funções de requisição de dados à API seguem um mesmo padrão de comunicação, utilizando o método POST que embute a remessa de dados no cabeçalho do pacote HTTP. Simplificando também o desempacotamento para o processamento dos dados, cumprindo as máximas da programação de reusabilidade do código e padronização para facilitar a manutenção e expansão quando necessário. Um exemplo do uso da nova função pode ser observado na Figura 32 que

complementa o exemplo mostrado na Figura 16 do recebimento da requisição pela API.

Figura 32 - Exemplo de utilização da função "*requestToAPInova*"

```
const toggleSwitch = async(user_uuid, environments_id) => {

  const result = await requestToAPInova(toggleSwitchEndPoint, {user_uuid , environments_id})

  logger('TOGGLE REQUEST', result)

  if (result.status == 200 && result.toggleSwitch != 'Error'){

    setIsEnabled((previousState) => !previousState);

  }

}
```

Fonte: O'Autor.

Todas as consultas construídas em (BEZERRA, 2024) passaram por essa adaptação a nova forma de realizar a consulta no banco de dados.

A expansão, de fato, das funcionalidades do aplicativo foi feita no *login* que foi modificado para solicitar usuário e senha para ser enviados a API e executar as rotinas de comparação da senha com o *hashing* armazenado e gerar o *token* JWT. E principalmente na tela de "Ambientes" (Figura 28), onde foi adicionado novos dois botões para efetuar o acionamento das lâmpadas e ar-condicionados dos ambientes. Funcionando de forma análoga a tranca eletrônica, os outros dois dispositivos quando acionados enviam o tópico e a mensagem "*change*" para a API que verifica o estado do dispositivo e configura o estado inverso e publica no *broker.* Além desses dois botões foi adicionado três campos de leitura dos sensores de temperatura, tensão e umidade. Uma rotina foi implementada e a cada 30 segundos a aplicação envia uma requisição à API para atualizar o dado exibido, essa requisição como a maioria das outras foi realizada através da função "*requestToAPInova*".

É interessante comentar que o método de geração das mensagens de retorno na tela para o usuário, também foi elaborado com foco na reutilização da programação e padronização, facilitando o uso e a manutenção do aplicativo. A função chamada

"*psiu*", gravada na pasta "*helpers*" no módulo "*psiu.js*" mostrada na Figura 33, necessita receber um objeto *JavaScript* contendo o título do cartão, a mensagem que se deseja exibir e, opcionalmente, um comando a ser executado quando o cartão é fechado.

Figura 33 - Módulo "*psiu.js*"

Fonte: O'Autor.

3.8 Configuração da *Raspberry Pi*, *Expo* e Banco de Dados *PostgreSQL*

O trabalho apresentado em (BEZERRA, 2024), foi configurado inicialmente com uma *Raspberry Pi 4B* de 8GB de RAM, foi constatado que o espaço utilizado superou 30GB (BEZERRA, 2024). No presente estudo, utilizou-se uma *Raspberry Pi 4B* com 4GB de RAM e um cartão de memória com capacidade máxima de 16GB, devido ao fato de o equipamento ter sido emprestado para a realização dos testes. Nesse *hardware*, foram instalados o *Expo*, o *Node.js* e o banco de dados *PostgreSQL*.

Seguindo a metodologia adotada em (BEZERRA, 2024), o sistema operacional selecionado foi o *Raspbian*, devido ao suporte oficial da *Raspberry Pi*. Para a instalação, utilizou-se o "*Raspberry Pi Imager*". Este programa, ao ser instalado em um computador, permitiu selecionar o sistema operacional e realizar a gravação através de um leitor de cartão de memória. Após a conclusão do processo, o cartão de memória foi inserido no *Raspberry Pi*, que em seguida foi conectado a um teclado, mouse e monitor, para ser usado de forma convencional.

Após configurar o sistema operacional, iniciou-se a instalação do ambiente de testes. Primeiramente, foi instalado o *Node.js*, seguido pela instalação do *Expo CLI* que exigiu a instalação do *Git* e *Watchman* para sistemas operacionais baseados em *Linux*. A instalação do *PostgreSQL* foi realizada no terminal através do comando "*sudo*

apt install postgresql", as entidades, atributos e dados foram inseridos também no terminal utilizando o SQL gerado a partir do banco configurado e populado no computador. O SQL das tabelas pode ser observado no apêndice A.

Posteriormente, foi executada a instalação das ferramentas escolhidas para realização dos testes, *Wireshark* e *Node-RED*. Para o *Node-RED* foi importado o fluxo da *dashboard* utilizada nos testes.

Com todos os *softwares* devidamente instalados, foi feito o *git clone* do projeto hospedado no repositório do *github*. Portando foi baixada uma cópia do sistema contendo a API e a aplicação mobile. Finalizado todas essas etapas, o servidor de desenvolvimento foi iniciado com o comando "*npx expo start*", e o aplicativo Expo Go foi instalado em um dispositivo móvel, permitindo a abertura do aplicativo ao ler o código QR gerado no terminal.

4 RESULTADOS

4.1 Teste de Comunicação Entre Aplicativo, API e Banco de Dados

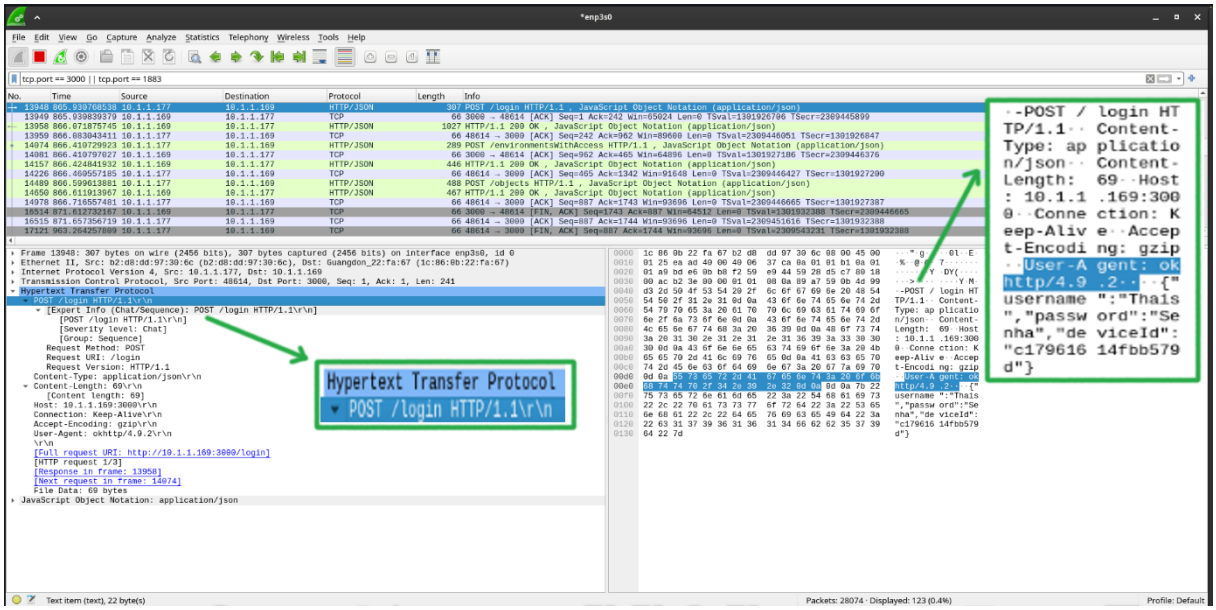
Para verificar a comunicação entre o aplicativo, a API e o banco de dados, foram realizadas requisições utilizando a API e o protocolo HTTP. O *software Wireshark* foi utilizado para monitorar essa comunicação, mapeando as requisições e respostas para demonstrar o tráfego dos dados.

As etapas do teste incluíram:

- Envio de requisições do aplicativo para a API.
- A API processando as requisições e interagindo com o banco de dados.
- Resposta da API ao aplicativo com os dados solicitados.

No *Wireshark* em um computador, foram observadas as requisições HTTP enviadas pelo aplicativo, incluindo os métodos *GET* e *POST*, os *endpoints* acessados e os dados transmitidos, conforme mostrado na Figura 34 a requisição de login que utiliza o método *POST*. As respostas HTTP da API, contendo os dados do banco de dados, também foram visíveis, mostrando o ciclo completo de comunicação. Essa análise confirmou que as requisições estavam sendo corretamente processadas e que a comunicação entre o aplicativo, a API e o banco de dados eram funcionais.

Figura 34 - Teste utilizando *Wireshark* no computador para capturar a requisição de login.

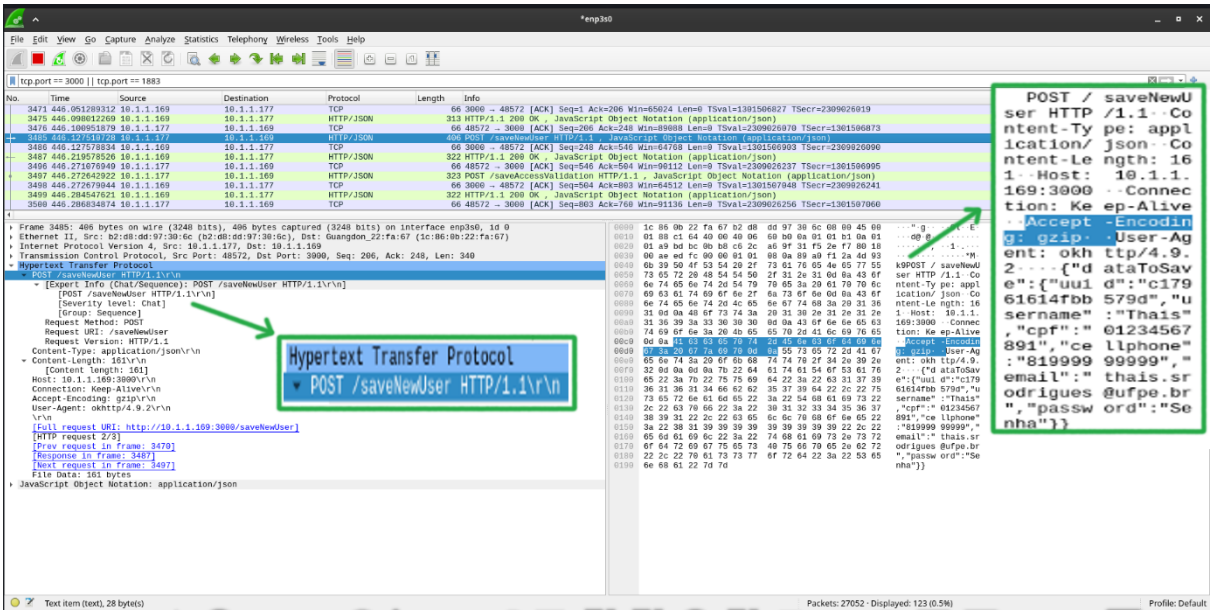


Fonte: O'Autor.

A Figura 35 apresenta a requisição de cadastro de novo usuário, também observada com o *Wireshark* em um computador, foi constatado que era possível visualizar a senha que o usuário informou.

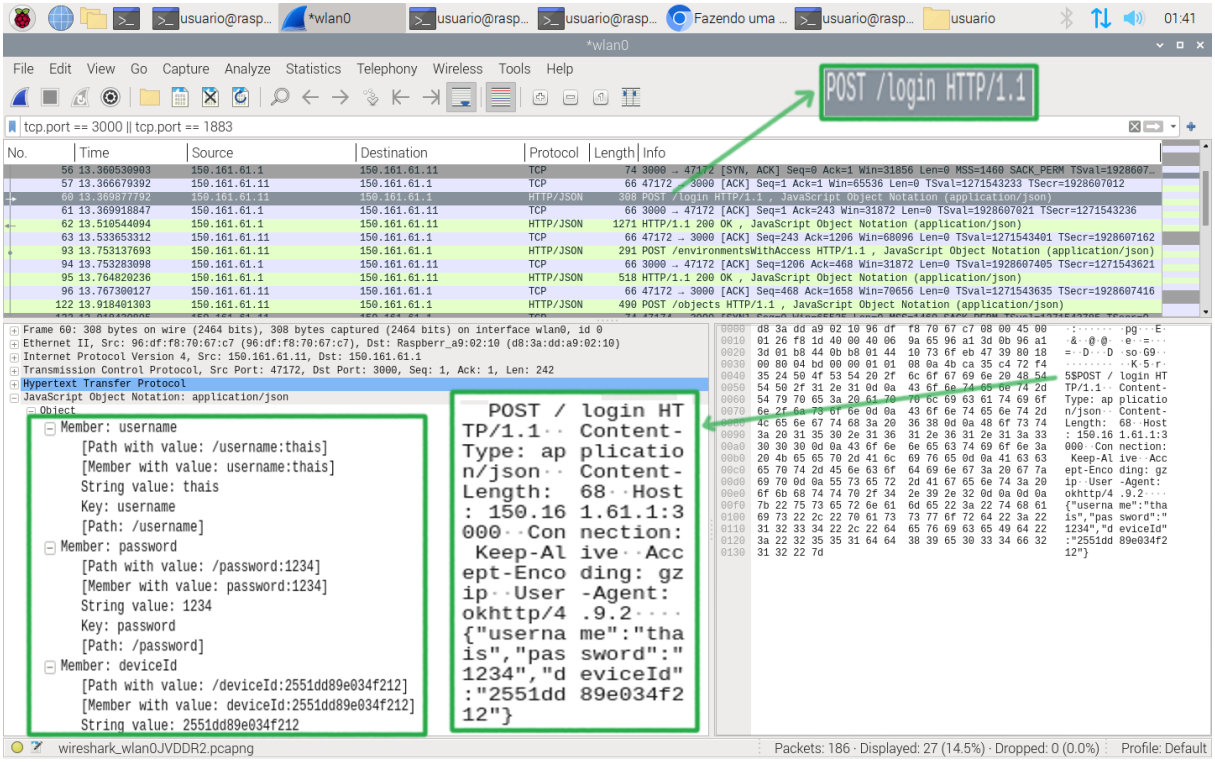
Os mesmos testes realizados no computador foram efetuados na *Raspberry Pi* como mostrado nas Figura 36 e Figura 37.

Figura 35 - Teste utilizando Wireshark no computador para capturar a requisição de salvar novo usuário



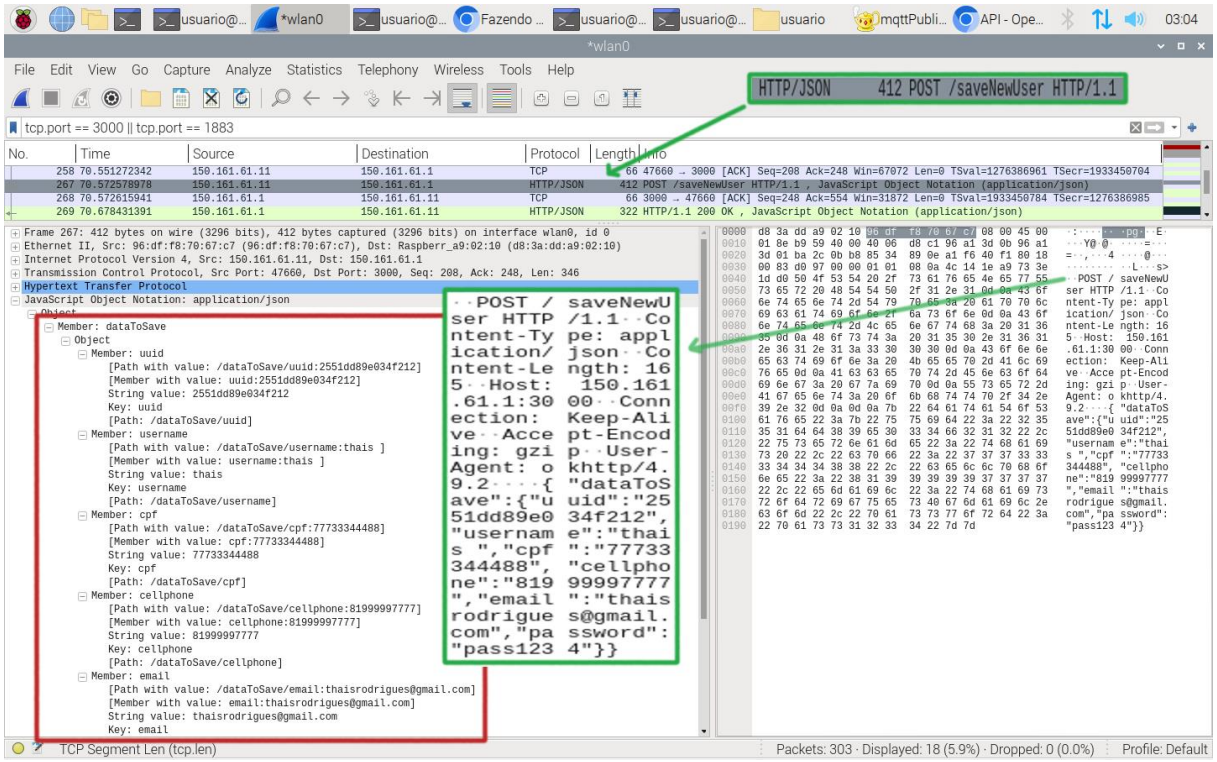
Fonte: O'Autor.

Figura 36 - Teste utilizando *Wireshark* na *Raspberry Pi* para capturar a requisição de login



Fonte: O'Autor.

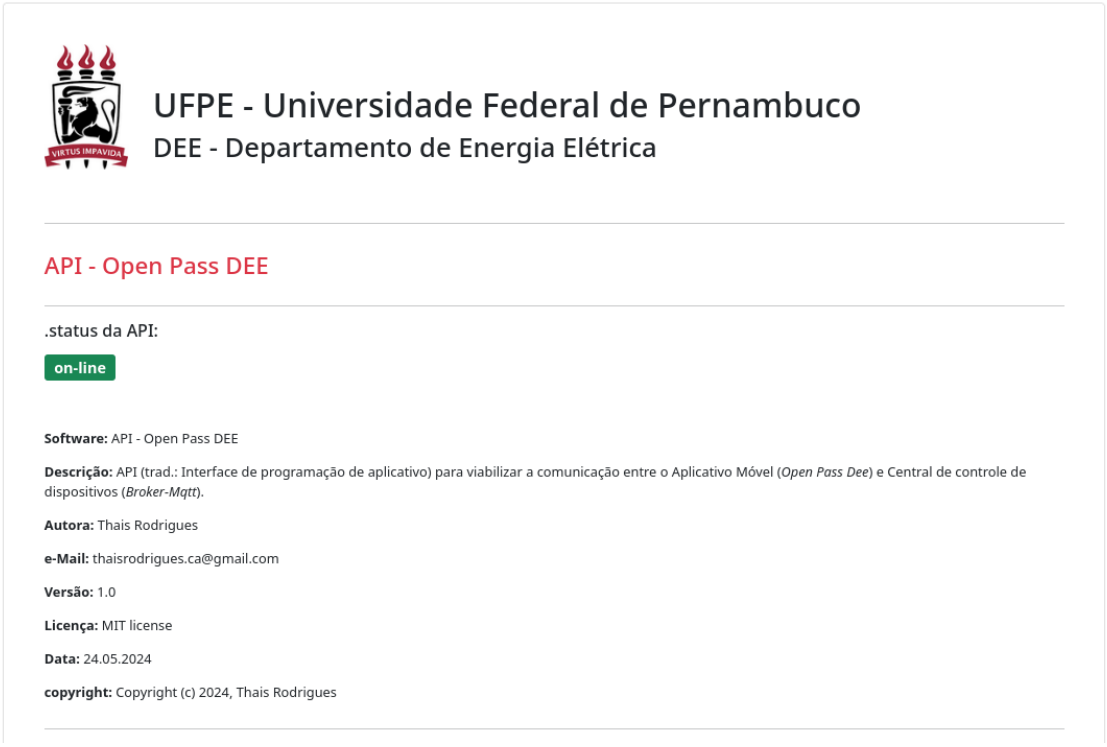
Figura 37 - Teste utilizando *Wireshark* na *Raspberry Pi* para capturar a requisição do de salvar novo usuário



Fonte: O'Autor.

A única requisição utilizando o método *GET* foi realizada para a construção de uma página apresentando as informações sobre o sistema. O objetivo dessa página é ser acessada em navegadores através do endereço de IP:porta do servidor que hospeda a API configurada. Nos testes, foi utilizado o endereço http://150.161.61.1:3000/. As Figura 38 e Figura 39 exibem a página renderizada em duas situações. A página, depois de aberta, monitora se a API responde a uma requisição feita do lado do cliente, a requisição parte do navegador para o servidor automaticamente. Caso a requisição seja bem-sucedida, a página indica que a API está on-line; caso contrário, a página muda para indicar que a API está off-line.

Figura 38 - Página de informações sobre o sistema, com o sistema *online*.



Fonte: O'Autor.

Figura 39 - Página de informações sobre o sistema, com o sistema *offline*.

Fonte: O'Autor.

4.2 Teste do Protocolo MQTT

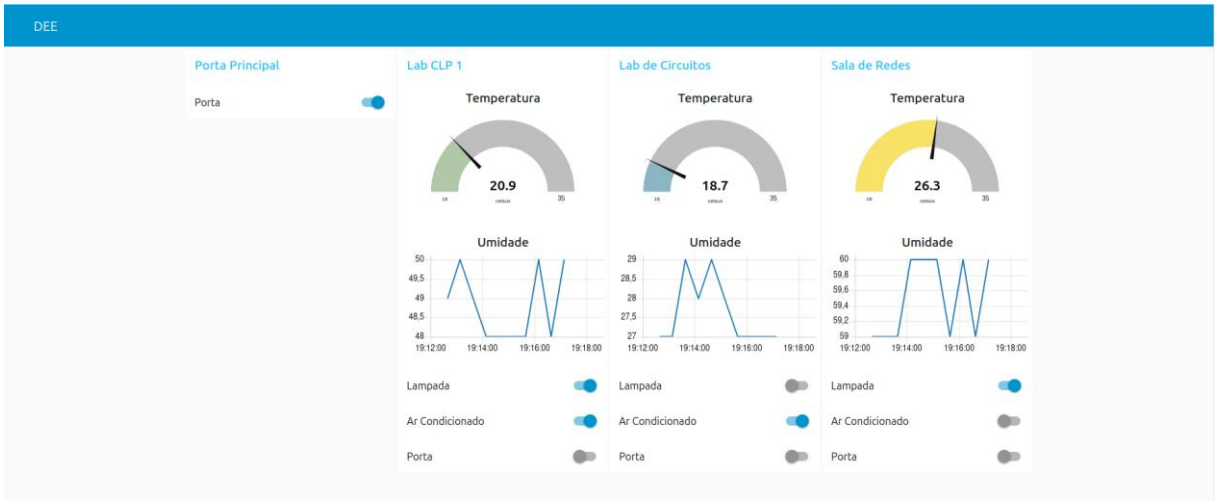
Foi montada uma rede para os testes, com características similares às da Universidade Federal De Pernambuco, utilizando a faixa de IP 150.161.61.X, máscara de rede 255.255.255.0 e *gateway* 150.161.61.254. O aplicativo *Expo Go* foi instalado em um dispositivo móvel e integrado à rede. Além disso, utilizou-se um *broker* MQTT do *Mosquitto* em outra máquina, replicando as condições presentes na universidade, conforme pode ser observado na Figura 10.

Os testes realizados incluíram:

- Abertura de porta.
- Acendimento e apagamento de luzes.
- Leitura de temperatura e umidade da sala.
- Leitura de tensão da rede.

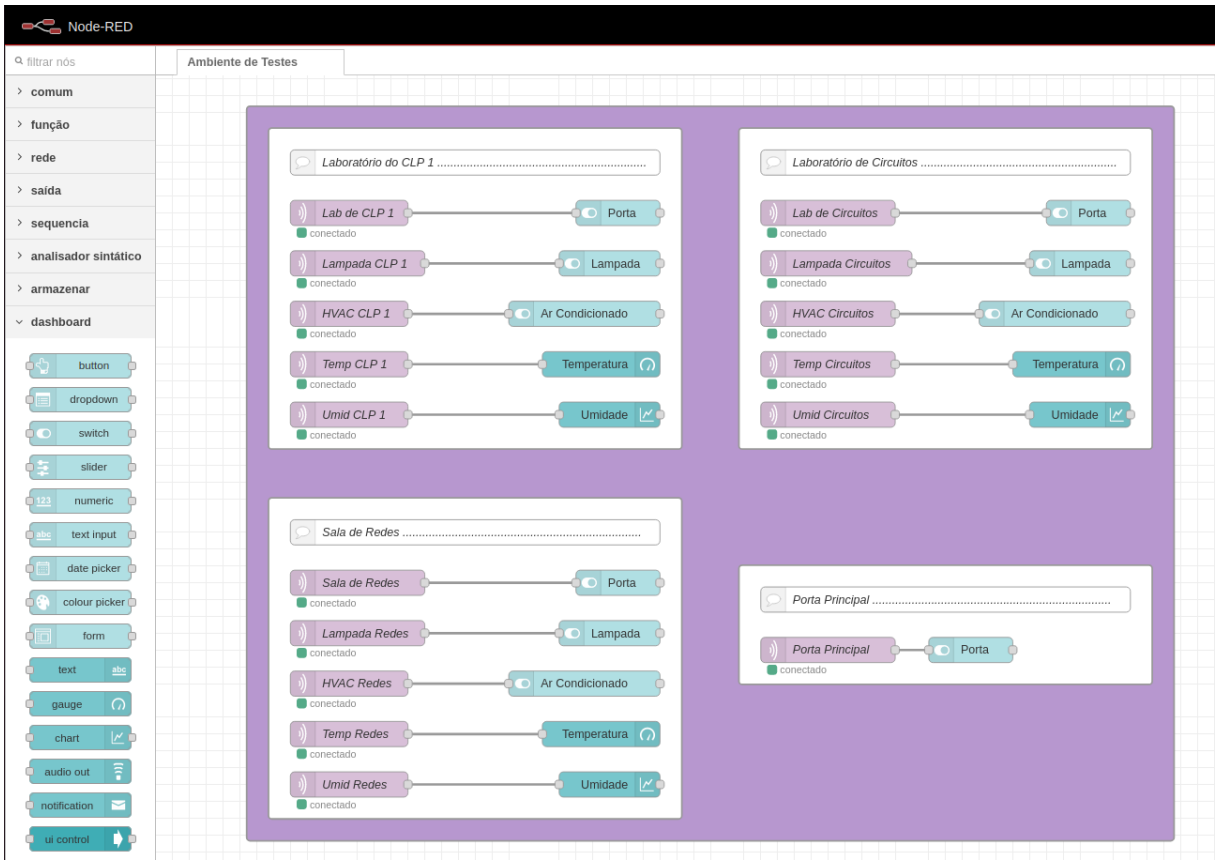
A Figura 40 apresenta a *dashboard* presente no *Node-RED* para simulação do ambiente físico. É importante destacar que, devido à ausência dos sensores, os dados foram simulados na API e enviados para o *broker*. O fluxo desenvolvido no *Node-RED* consistiu na leitura dos tópicos e no envio dos dados para os blocos do *dashboard*. A Figura 41 apresenta o fluxo completo que ilustra o processo de comunicação entre os componentes, desde a coleta dos dados simulados até a exibição na dashboard do *Node-RED*. Que consiste em um bloco assinante de um tópico do *broker* MQTT conectado ao bloco de elementos da biblioteca “*node-red-dashboard”.*

Figura 40 - Dashboard desenvolvida no Node-RED para simulação do ambiente físico



Fonte: O'Autor.

Figura 41 - O fluxo desenvolvido no Node-RED.

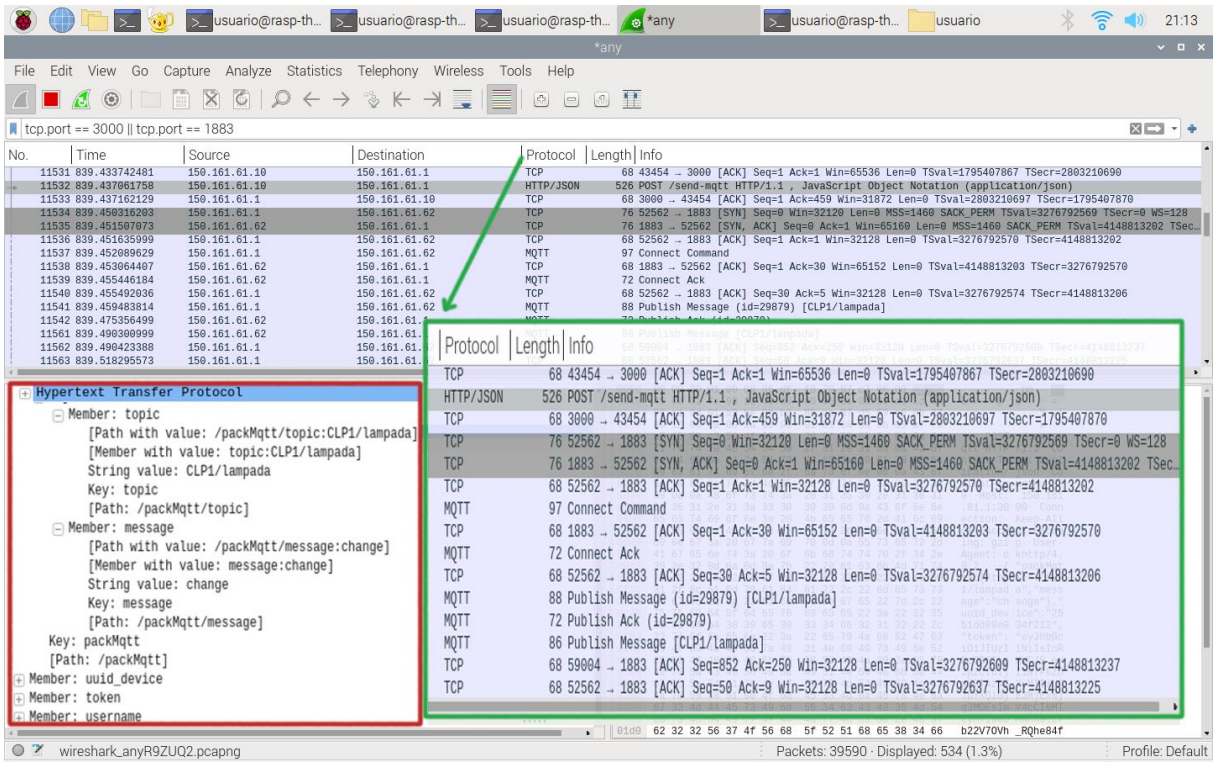


Fonte: O'Autor.

Na Figura 42, o software *Wireshark* monitorou a comunicação MQTT entre a API e o *broker* após a requisição HTTP do aplicativo à API. No *Wireshark*, foram observados pacotes MQTT sendo trocados entre os dispositivos, com mensagens publicadas e assinadas visíveis em tempo real. Isso incluiu a identificação dos tópicos MQTT usados, a qualidade do serviço (QoS) e o conteúdo das mensagens sendo trafegadas.

Posteriormente, utilizou-se a internet nessa rede local e os testes foram repetidos. Ao conectar à rede local à internet, o *Wireshark* mostrou pacotes adicionais referentes ao tráfego de internet, mas a comunicação MQTT interna entre os dispositivos foi visível da mesma forma. Os testes repetidos confirmaram a robustez da API HTTP e do sistema em condições de rede mais complexas, incluindo a presença de tráfego de internet.

Figura 42 - Teste utilizando *Wireshark* na *Raspberry Pi* para monitorar a requisição de publicação no *broker.*



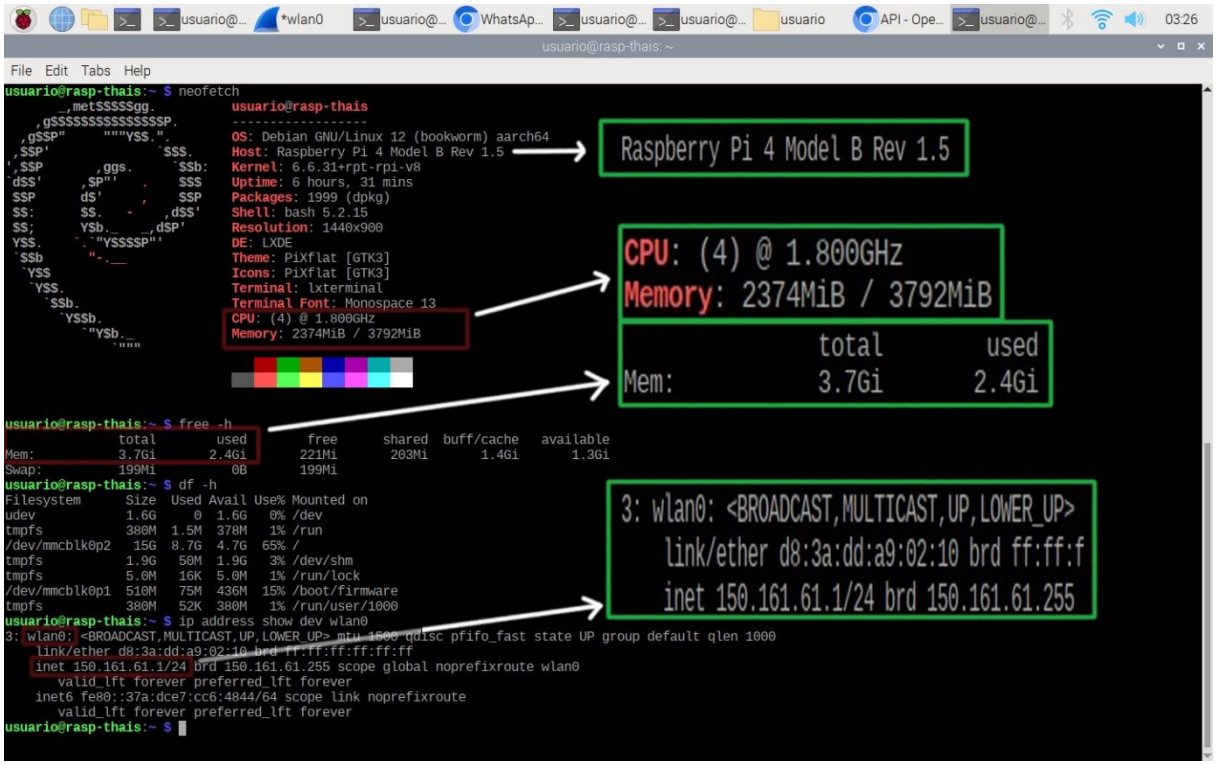
Fonte: O'Autor.

Esses testes garantiram a funcionalidade correta da API HTTP em um ambiente controlado, similar ao da universidade, utilizando as ferramentas e configurações mencionadas.

4.3 Avaliação de Desempenho: Utilização de Memória RAM e Espaço em Disco

Para execução dos testes de uso do aplicativo e comunicação com a API e o *broker* MQTT, na Figura 43 é possível observar o espaço de memória RAM utilizado de 2,4GB, armazenamento utilizados, rede e configuração do *hardware*.

Figura 43 - Relatório de Sistema: Informação de Hardware, Utilização de Memória e Rede



Fonte: O'Autor.

Para avaliar o desempenho do sistema, foram realizados testes de utilização de memória RAM com todos os serviços levantados, incluindo a API, o *Expo CLI* e o banco de dados. Observou-se o espaço de disco utilizado no cartão de memória e comparou-se com os valores antigos. No sistema antigo, a memória RAM utilizada era

de pelo menos 7GB devido ao uso do *Supabase* (BEZERRA, 2024), e o espaço em disco utilizado era de pelo menos 30GB. No sistema atual, a memória RAM utilizada foi de 2.4GB e o espaço em disco foi de aproximadamente 9GB. Possibilitando uma redução de *hardware* do sistema, para uma *Raspberry Pi* de 4GB de memória RAM.

A Tabela 2, compara os recursos utilizados neste trabalho e no trabalho desenvolvido anteriormente por (BEZERRA, 2024).

Tabela 2 – Comparação Consumo de Recursos deste Trabalho com o apresentado em (BEZERRA, 2024).

Recursos Utilizados	**Com API HTTP**	**Sem API HTTP** (BEZERRA, 2024)
Memória RAM	2,4GB	8GB
Armazenamento Interno	9GB	30GB
Expo	Utilizado	Utilizado
PostgreSQL	Utilizado	Utilizado
Supabase	Não Utilizado	Utilizado
Node-RED	Não Utilizado	Utilizado
Protocolo HTTP	Utilizado	Utilizado
Protocolo MQTT	Utilizado	Utilizado
Protocolo WebSocket	Não Utilizado	Utilizado
Criptografia de Senha	Utilizado	Não Utilizado
Token	Utilizado	Utilizado
Atualização de Token	Utilizado	Não Utilizado

Fonte: O'Autor.

A fórmula para calcular o percentual de melhoria é derivada do conceito básico de porcentagem e de diferença relativa entre dois valores. Originalmente usada em matemática financeira e estatística, essa fórmula mede a variação percentual entre um valor novo e um valor antigo, permitindo avaliar o quanto algo melhorou ou piorou em relação a uma referência anterior.

$$\text{Percentual de Melhoria} = \left(\frac{\text{Valor Antigo} - \text{Valor Novo}}{\text{Valor Antigo}}\right) \times 100 \qquad (4.1)$$

Utilizando a fórmula de percentual de melhoria:

$$\text{Percentual de Melhoria da Memória RAM} = \left(\frac{8 - 2.4}{8}\right) \times 100 \approx 70\% \quad (4.2)$$

$$\text{Percentual de Melhoria do Espaço em Disco} = \left(\frac{30 - 9}{30}\right) \times 100 \approx 70\% \quad (4.3)$$

5 CONCLUSÕES E PROPOSTAS DE CONTINUIDADE

Com base nos testes realizados e nos resultados obtidos, pode-se concluir que a API HTTP desenvolvida para integrar o aplicativo móvel de controle de acesso às salas de aula e laboratórios do Departamento de Engenharia Elétrica da UFPE atende os requisitos definidos. A utilização das tecnologias de Internet das Coisas (*IoT*), protocolo MQTT, aplicativo móvel *React Native* e banco de dados *PostgreSQL* se mostrou eficaz para criar um ambiente acadêmico mais seguro e conectado.

Os testes de comunicação entre o aplicativo, a API e o banco de dados, monitorados pelo *Wireshark*, demonstraram que as requisições HTTP (GET e POST) foram corretamente processadas e que os dados trafegaram de maneira eficiente entre o aplicativo e o banco de dados. Isso confirma a funcionalidade adequada da API no gerenciamento de dados de acesso e controle.

Apesar dessa falha de segurança constatada no teste apresentado na Figura 35, o sistema continuará seguro devido às várias camadas de proteção desenvolvidas. Mesmo a senha sendo o "*uuid*" do dispositivo e o "*username*" forem decifrados há verificações adicionais para manter a segurança do sistema. O "*uuid*" do dispositivo foi gerado a partir do valor de "*Settings.Secure.ANDROID_ID*" através da função "*Application.getAndroidId*" no Android, que consiste em uma *string* hexadecimal exclusiva para cada combinação de chave de assinatura de aplicativo, usuário e dispositivo. Para o iOS, o valor identificador para fornecedor é obtido através da função "*Application.getIosIdForVendorAsync*", um ID de *string* que identifica exclusivamente um dispositivo para o fornecedor do aplicativo.

A aplicação utilizou essas funções para assegurar que o dispositivo era o mesmo durante a navegação do usuário, essa funcionalidade foi herdada do trabalho desenvolvido por (BEZERRA, 2024). Além disso, o um token gerado com o "*uuid*" do dispositivo, foi verificado em cada tela e requisição, garantindo uma camada adicional de segurança.

Os testes do protocolo MQTT, realizados em uma rede com características similares às da universidade, mostraram que a comunicação entre o aplicativo, a API e o *broker* MQTT foi eficiente. As operações de controle, como abertura de portas,

acendimento e apagamento de luzes, leitura de temperatura e umidade, e leitura de tensão da rede, foram executadas corretamente. O monitoramento pelo *Wireshark* confirmou a troca de pacotes MQTT em tempo real, evidenciando a robustez do sistema mesmo em condições de rede mais complexas.

Além da redução no *hardware*, as mudanças efetuadas no sistema resultaram em um aumento significativo de eficiência. Os testes mostraram uma redução notável tanto no uso de memória RAM, que passou de 8GB (com o *Supabase* consumindo ao menos 7GB) para 2.4GB, quanto no espaço em disco utilizado, que diminuiu de mais de 30GB para 9GB, resultando em uma melhoria de 70% em ambos os casos. Isso valida a eficácia dos novos componentes e da arquitetura proposta, demonstrando que as otimizações implementadas compensaram a diminuição nos recursos de hardware.

A implementação da API HTTP, em conjunto com as tecnologias de *IoT*, protocolo MQTT, aplicativo *React Native* e banco de dados *Postgres*, proporcionam um controle de acesso eficiente e seguro, alinhado com o objetivo de monitorar e assegurar o ambiente acadêmico do Departamento de Engenharia Elétrica da UFPE. Os testes realizados validaram a funcionalidade e a integridade do sistema, demonstrando que a solução desenvolvida é adequada para as necessidades do departamento.

Para aprimorar ainda mais o sistema e garantir sua sustentabilidade a longo prazo, poderiam ser implementadas diversas ações contínuas. Isso inclui o estabelecimento de um plano de manutenção regular para monitorar o desempenho da API, a integridade dos dados e a segurança do sistema. Além disso, poderiam ser adotados alertas automáticos para identificar e resolver rapidamente problemas de comunicação ou falhas no sistema.

Expansão das funcionalidades do aplicativo móvel, com a adição de notificações *push* para eventos específicos, como tentativas de acesso não autorizadas ou alterações nos parâmetros ambientais das salas. A integração de sensores adicionais e dispositivos IoT poderia ser priorizada para ampliar a cobertura e o monitoramento dos ambientes.

Oferecer treinamentos regulares para o pessoal técnico e administrativo do departamento, visando melhorar o conhecimento sobre o uso e a administração do sistema. Paralelamente, desenvolver documentação detalhada e tutoriais para facilitar a adoção e operação do sistema por novos usuários.

Desenvolver uma coleta regular de *feedback* dos usuários para identificar áreas de melhoria e adaptar as funcionalidades conforme as necessidades reais. Avaliações periódicas da experiência do usuário também seriam realizadas para garantir a usabilidade e eficácia contínua do aplicativo.

Medidas rigorosas de segurança poderiam ser implementadas para proteger os dados e as comunicações do sistema. Isso incluiria a adoção de políticas de segurança robustas e a aplicação regular de atualizações de *software* e *patches* de segurança.

Adicionalmente, seria incorporada a capacidade de configurar dias e horários específicos para a permissão de acesso aos ambientes para cada usuário, visando reforçar ainda mais a segurança do sistema. Um gerenciador das configurações do *broker* poderia ser implementado para facilitar ajustes de seus parâmetros, como *host*, porta, usuário e senha. E usuários administradores que tenham acesso como coordenador em todos os ambientes automaticamente quando se tornam administradores. A implementação de uma nova tela com a inclusão de histórico de umidade, temperatura e acesso aos ambientes.

Outro ponto importante a ser observado é a possibilidade de expansão da API para ser acessível em navegadores. Conforme foi feito na página GET que exibe informações sobre a API, é possível construir um sistema de login para modificação de configurações, gestão de usuários e amplo acesso aos dados coletados com o uso do sistema.

A integração da tecnologia NFC poderia ser explorada para permitir o acesso independente de redes locais, proporcionando maior flexibilidade aos usuários.

Essas medidas e propostas visam fortalecer a segurança, expandir funcionalidades e aprimorar a usabilidade do sistema de controle de acesso, contribuindo para um ambiente acadêmico seguro e bem monitorado.

REFERÊNCIAS

ADONIS. **adonis**, 2024. Disponível em: <https://adonisjs.com/about>. Acesso em: 21 jun. 2024.

AMAZON WEB SERVICES. O que é IDE (Ambiente de desenvolvimento integrado)? **aws**, 2023. Disponível em: <https://aws.amazon.com/pt/?nc2=h_lg>. Acesso em: 03 mar. 2024.

AMAZON WEB SERVICES. O que é IoT (Internet das Coisas)? **aws**, 2023. Disponível em: <https://aws.amazon.com/pt/what-is/iot/>. Acesso em: 11 mar. 2024.

AMAZON WEB SERVICES. O que é MQTT? **aws**, 2023. Disponível em: <https://aws.amazon.com/pt/what-is/mqtt/>. Acesso em: 09 maio 2024.

ANDERSON, R. J. **Security Engineering:** A Guide to Building Dependable Distributed Systems.

ANSELME. API HTTP, REST ou RESTFul. **Anselme**, 2023. Disponível em: <https://www.anselme.com.br/2023/11/15/api-http-rest-ou-restful/#API-HTTP-8211-HyperText-Transfer-Protocol>. Acesso em: 12 jun. 2024.

ANVISA. Resolução - RE nº 9. **ANVISA**, 2003. Disponível em: <https://static.webarcondicionado.com.br/blog/uploads/2012/02/resolucao9anvisa.pdf>. Acesso em: 05 fev. 2024.

BEZERRA, Maria H. R. D. A. **Controle de Acesso a Ambientes Físicos Utilizando Ferramentas**. UFPE. [S.l.]. 2024.

BOA iluminação nas escolas influencia no desempenho dos alunos. **ultraluz**, 2024. Disponível em: <https://ultraluz.com.br/boa-iluminacao-nas-escolas-influencia-no-desempenho-dos-alunos/>. Acesso em: 04 mar. 2024.

BOLTIC. Choosing Between REST APIs and HTTP APIs - Key Differences and Advantages. **Boltic**, 2024. Disponível em: <https://www.boltic.io/blog/rest-api-http-api>. Acesso em: 12 jun. 2024.

CARVALHO, Cristiana. Internet das coisas: entenda o que é e como funciona. **tecmundo**, 2021. Disponível em: <https://www.tecmundo.com.br/internet/230884-internet-coisas-entenda-funciona.htm>. Acesso em: 11 mar. 2024.

CARVALHO, Jackeline. O futuro da internet das coisas: últimas estatísticas. **IPNews**, 2023. Disponível em: <https://ipnews.com.br/o-futuro-da-internet-das-coisas-ultimas-estatisticas-reveladas/#google_vignette>. Acesso em: 11 mar. 2024.

CLOUDFLARE. Por que o HTTP não é seguro? | HTTP x HTTPS. **Cloudflare**, 2024. Disponível em: <https://www.cloudflare.com/pt-br/learning/ssl/why-is-http-not-secure/>. Acesso em: 23 jun. 2024.

CODD, E. F. **The Relational Model for Database Management:** Version 2.

CONNOLLY, T.; BEGG, C. **Database Systems:** A Practical Approach to Design, Implementation, and Management.

COOPERSYSTEM. As 7 melhores linguagens de programação para desenvolvimento mobile. **coopersystem**, 2023. Disponível em: <https://www.coopersystem.com.br/as-7-melhores-linguagens-de-programacao-para-desenvolvimento-mobile/>. Acesso em: 03 abr. 2024.

CORONEL, C.; MORRIS, S. **Database Systems:** Design, Implementation, & Management.

CUNHA, André. React Native: o que é e tudo sobre o Framework. **alura**, 2023. Disponível em: <https://www.alura.com.br/artigos/react-native#:~:text=React%20Native%20(tamb%C3%A9m%20conhecido%20como,a%20mesma%20base%20de%20c%C3%B3digo.>. Acesso em: 15 fev. 2024.

CURRICULUM VISIONS. Hypocaust. **Curriculum Visions**, 2022. Disponível em: <https://www.curriculumvisions.com/search/H/hypocaust/hypocaust.html>. Acesso em: 11 mar. 2024.

DAVIE, Larry L. P. E. B. S. **Redes de computadores uma abordagem de sistemas**. 5ª. ed.

DBEAVER COMMUNITY EDITION. About. **DBeaver Community**. Disponível em: <https://dbeaver.io/>. Acesso em: 27 fev. 2024.

DEBIAN. Debian. **Debian**, 2023. Disponível em: <https://www.debian.org/index.pt.html>. Acesso em: 06 abr. 2024.

DEEPSEA DEVELOPMENTS. IoT com Raspberry Pi: a combinação perfeita para soluções inteligentes. **DeepSea Developments**, 2023. Disponível em: <https://www.deepseadev.com/en/blog/iot-with-raspberry-pi/#:~:text=Raspberry%20Pi%20boards%20are%20compatible,using%20IoT%20with%20Raspberry%20Pi.>. Acesso em: 08 mar. 2024.

EXPO. Expo. **Expo**, 2024. Disponível em: <https://expo.dev/>. Acesso em: 07 jan. 2024.

EXPRESSVPS. O que é: SOAP (Simple Object Access Protocol). **ExpressVPS**, 2024. Disponível em: <https://expressvps.com.br/glossario/o-que-e-soap-simple-object-access-protocol/>. Acesso em: 12 jun. 2024.

FASTER CAPITAL. Internet of Things: IoT: Connecting Devices with RPi. **Faster Capital**, 2023. Disponível em: <https://fastercapital.com/content/Internet-of-Things--IoT---Connecting-Devices-with-RPi.html#Monitoring-and-Controlling-IoT-Devices-with-Raspberry-Pi>. Acesso em: 09 mar. 2024.

FERGUSON, Niels; SCHNEIER, Bruce; KOHNO, Tadayoshi. **Cryptography Engineering:** Design Principles and Practical Applications.

FIELDING, Roy T. **Architectural Styles and the Design of Network-based Software Architectures**. UNIVERSITY OF CALIFORNIA. IRVINE, p. 76. 2000.

FLANAGAN, D. **JavaScript:** The Definitive Guide.

FONSECA, Leonardo. Fundamentos de Banco de Dados – Banco de Dados Relacional. **leonardofonseca**, 2021. Disponível em: <https://leonardofonseca.com.br/2021/02/22/fundamentos-de-banco-de-dados-banco-de-dados-relacional/>. Acesso em: 02 jul. 2024.

FORBES TECH. IoT: até 2025, mais de 27 bilhões de dispositivos estarão conectados. **Forbes**, 2022. Disponível em: <https://forbes.com.br/forbes-tech/2022/08/iot-ate-2025-mais-de-27-bilhoes-de-dispositivos-estarao-conectados/>. Acesso em: 11 mar. 2024.

FUNDAÇÃO RASPBERRY PI. Raspberry Pi. **Raspberry Pi**. Disponível em: <https://www.raspberrypi.com/>. Acesso em: 17 jan. 2024.

GARCIA-MOLINA, Hector; ULLMAN, Jeffrey D.; WIDOM, Jennifer. **DATABASE SYSTEMS:** The Complete Book. 2ª. ed.

GITHUB. Atom Documentation, 2024. Disponível em: <https://atom.io/docs>. Acesso em: 10 jun. 2024.

GOODWIN, Michael. O que é uma API. **IBM**, 2024. Disponível em: <https://www.ibm.com/br-pt/topics/api#:~:text=IBM&text=O%20que%20%C3%A9%20uma%20API%3F,trocar%20dados%2C%20recursos%20e%20funcionalidades.>. Acesso em: 24 jun. 2024.

GOOGLE. Flutter Documentation. **Flutter**, 2024. Disponível em: <https://docs.flutter.dev/>. Acesso em: 14 jun. 2024.

GOURLEY, David; TOTTY, Bryan. **HTTP The Definitive Guide**. 1. ed.

GRINBERG, M. **Flask Web Development:** Developing Web Applications with Python.

GRUPO DE DESENVOLVIMENTO GLOBAL POSTGRESQL. Sobre. **PostgreSQL**, 2024. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 07 jan. 2024.

HACKERONE. Understanding the Benefits of Key Derivation Functions: A Deep Dive into PBKDF2. **PullRequest**, 2024. Disponível em: <https://www.pullrequest.com/blog/understanding-the-benefits-of-key-derivation-functions-a-deep-dive-into-pbkdf2/>. Acesso em: 16 jun. 2024.

HERNANDEZ, M. J. **Database Design for Mere Mortals:** A Hands-On Guide to Relational Database Design. 3. ed.

HICKS, Marie. **Programming Inequality:** How British Discarded Women Technologists and Lost Its Edge in Computing. 1ª. ed.

HOLOVATY, A.; KAPLAN-MOSS, J. **The Definitive Guide to Django:** Web Development Done Right.

ILUMINAÇÃO inteligente: o que compõe o sistema? **POSITIVO Casa Inteligente**, 2022. Disponível em: <https://blog.positivocasainteligente.com.br/iluminacao-inteligente-o-que-compoe-o-sistema/>. Acesso em: 07 mar. 2024.

INFINITE RED. Ignite CLI Documentation., 2024. Disponível em: <https://github.com/infinitered/ignite>. Acesso em: 14 jun. 2024.

ISAACSON, Walter. **The Innovators**.

JWT.IO. Introduction to JSON Web Tokens. **JWT.IO**, 2024. Disponível em: <https://jwt.io/introduction>. Acesso em: 20 jun. 2024.

KOA. **Koa**, 2024. Disponível em: <https://koajs.com/#>. Acesso em: 21 jun. 2024.

KROENKE, D. M.; AUER, D. **Database Concepts**. 7. ed.

LOGAP. Banco de dados NoSQL para iniciantes: o que é, quais os tipos existentes e principais SGBDs para NoSQL. **LogAp**, 2022. Disponível em: <https://logap.com.br/blog/banco-de-dados-nosql-para-iniciantes/>. Acesso em: 02 jul. 2024.

MARIANI, Antonio L. D. C. Controle de umidade e a qualidade dos ambientes internos. **engenharia arquitetura**, 2019. Disponível em: <https://www.engenhariaearquitetura.com.br/2019/04/controle-de-umidade-e-a-qualidade-dos-ambientes-internos>. Acesso em: 06 fev. 2024.

MELTON, J.; SIMON, A. R. **SQL:** 1999 - Understanding Relational Language Components.

MENEZES, Alfred J.; OORSCHOT, Paul C. V.; VANSTONE, Scott A. **Handbook of Applied Cryptography**.

MICROSOFT. **Visual Studio**, 2024. Disponível em: <https://visualstudio.microsoft.com/pt-br/>. Acesso em: 04 mar. 2024.

MICROSOFT. Learn Microsoft, 2024. Disponível em: <https://learn.microsoft.com/pt-br/windows-hardware/drivers/portable/using-the-netmon-tool>. Acesso em: 09 jun. 2024.

MICROSOFT. Power Automate Documentation, 2024. Disponível em: <https://learn.microsoft.com/en-us/power-automate/>. Acesso em: 23 maio 2024.

MOZILLA CORPORATION. JavaScript. **mdn**, 2024. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 04 fev. 2024.

MOZILLA DEVELOPER NETWORK. API - MDN Web Docs, 2024. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Glossary/API>. Acesso em: 23 jun. 2024.

NELOGICA. Criptomoedas de A a Z. **akeloo Nelogia**, 2023. Disponível em: <https://akeloo.com.br/glossario-cripto/sha-256/#:~:text=SHA%2D256%20%C3%A9%20uma%20fun%C3%A7%C3%A3o,fluxo%20de%20informa%C3%A7%C3%B5es%20do%20sistema.>. Acesso em: 16 jun. 2024.

NIELES, M.; DEMPSEY, K.; PILLITTERI, V. **An Introduction to Information Security**. National Institute of Standards and Technology. [S.l.]. 2017.

NODE.JS FOUNDATION. Express. **Express**, 2024. Disponível em: <https://expressjs.com/pt-br/>. Acesso em: 20 jun. 2024.

NODE-RED. Node-RED. **Node-RED:** Low-code programming for event-driven applications., 2024. Disponível em: <https://nodered.org/>.

OPENJS FOUNDATION. Sobre a Node.js. **Node**, 2024. Disponível em: <https://nodejs.org/pt>. Acesso em: 20 jun. 2024.

ORACLE. O que é IoT? **Oracle**, 2024. Disponível em: <https://www.oracle.com/br/internet-of-things/what-is-iot/>. Acesso em: 11 mar. 2024.

PENA, Rodolfo F. A. Umidade atmosférica. **MUNDO EDUCAÇÃO**, 2024. Disponível em: <https://mundoeducacao.uol.com.br/geografia/umidade-atmosferica.htm#:~:text=A%20primeira%2C%20tamb%C3%A9m%20chamada%20de,antes%20que%20ela%20se%20precipite.>. Acesso em: 15 fev. 2024.

PEREIRA, Anderson L. Os Benefícios dos WebSockets. **Medium**, 2023. Disponível em: <https://andy2903-alp.medium.com/os-benef%C3%ADcios-dos-websockets-dcbf25c1e680>. Acesso em: 10 jun. 2024.

PEREIRA, Caio R. **Aplicações web real-time com Node.js**.

PETZOLD, Charles. **The Hidden Language of Computer Hardware and Software**. 1. ed.

PROVOS, Niels; MAZIERES, David. **A Future-Adaptable Password Scheme**.

RASPBERRY PI FOUNDATION. Raspberry Pi OS. **Raspberry Pi**, 2024. Disponível em: <https://www.raspberrypi.com/software/>. Acesso em: 04 mar. 2024.

RED HAT. O que é uma API? **Red Hat**, 2022. Disponível em: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>. Acesso em: 24 jun. 2024.

RF WIRELESS WORLD. MQTT Tutorial | MQTT Architecture, MQTT Protocol Use Cases. **RF Wireless World**, 2022. Disponível em: <https://www.rfwireless-world.com/Tutorials/MQTT-tutorial.html>. Acesso em: 18 jun. 2024.

SADALAGE, P. J.; FOWLER, M. **NoSQL Distilled:** A Brief Guide to the Emerging World of Polyglot Persistence.

SANTANA, Bruno. O Que é Ubuntu? Um Guia Rápido para Iniciantes. **Hostinger Tutoriais**, 2023. Disponível em: <https://www.hostinger.com.br/tutoriais/o-que-e-ubuntu-linux>. Acesso em: 03 fev. 2024.

SANTANA, Bruno. As 20 Melhores Ferramentas de Desenvolvimento Web para Melhorar Seu Trabalho. **Hostinger Tutorias**, 2024. Disponível em: <https://www.hostinger.com.br/tutoriais/ferramentas-de-desenvolvimento-web>. Acesso em: 02 maio 2024.

SANTANA, Bruno. O Que é HTTP e Como Ele Permite o Acesso ao Seu Site. **HOSTINGER TUTORIAIS**, 2024. Disponível em: <https://www.hostinger.com.br/tutoriais/http>. Acesso em: 08 maio 2024.

SAP. O que é a Internet das Coisas Industrial (IIoT)? **SAP**, 2024. Disponível em: <https://www.sap.com/brazil/products/scm/industry-4-0/what-is-iiot.html#:~:text=A%20IIoT%20%C3%A9%20um%20subconjunto,em%20um%20processo%20mais%20complexo.>. Acesso em: 11 mar. 2024.

SCHNEIER, Bruce. **Applied Cryptography:** Protocols, Algorithms, and Source Code in C. 2ª edição. ed.

SHARIF, Zubair et al. Smart Home Automation by Internet-of-Things Edge. **International Journal of Advanced Computer Science and Applications**, Malasya, 13, n. 4, 2022. 1-11.

SIMONI, Thayna. IMPACTOS DA ALTA DA TEMPERATURA NA ESTABILIDADE DOS MEDICAMENTOS. **SensorWeb**, 2024. Disponível em: <https://sensorweb.com.br/impactos-da-alta-da-temperatura-na-estabilidade-dos-medicamentos/>. Acesso em: 18 jan. 2024.

SIMPLIFICANDOREDE. Protocolo HTTP – Como funciona? **Simplificando Redes**, 2021. Disponível em: <https://simplificandoredes.com/protocolo-http-como-funciona/>. Acesso em: 05 jul. 2024.

SOUZA, Lindeberg B. D. **Protocolos e Serviços de Rede**. 1ª. ed.

STALLINGS, William. **Criptografia e Segurança de Redes**..6ª edição. ed.

STONEBRAKER, M. SQL Databases v. NoSQL Databases. **Communications of the ACM**, 2012.

SUBLIME TEXT. Sublime Text Documentation., 2024. Disponível em: <https://www.sublimetext.com/docs/>. Acesso em: 10 jun. 2024.

SUGARMAN, Samuel C. **HVAC Fundamentals**. 3. ed.

TANENBAUM, Andrew S. **Computer Networks**. 4. ed.

TCPDUMP. tcpdump Documentation, 2024. Disponível em: <https://www.tcpdump.org/>. Acesso em: 27 maio 2024.

THE GRAPHQL FOUNDATION. Introduction to GraphQL. **GraphQL**, 2024. Disponível em: <https://graphql.org/>. Acesso em: 10 jun. 2024.

TORRES, Manoel G. L.; OLIVEIRA, Paulo. **CONDIÇÕES TÉRMICAS E DESEMPENHO EM AMBIENTES DE ENSINO - NORTE DE PORTUGAL E NORDESTE DO BRASIL**. UFPB; ESTG. João Pessoa, p. 162. 2016.

VENESS, Chris. Scripts de tipo móvel, 2019. Disponível em: <https://www.movable-type.co.uk/scripts/sha256.html>. Acesso em: 16 jun. 2024.

WEBARCONDICIONADO. História dos Sistemas de Aquecimento. **WebArcondicionado**, 2022. Disponível em: <https://www.webarcondicionado.com.br/historia-dos-sistemas-de-aquecimento>. Acesso em: 12 fev. 2024.

WELLING, L.; THOMSON, L. **PHP and MySQL Web Development**.

WIRESHARK. Wireshark User's Guide, 2024. Disponível em: <https://www.wireshark.org/docs/>. Acesso em: 04 maio 2024.

WORLD WIDE WEB FOUNDATION. History of the Web. **World Wide Web Foundation**, 2022. Disponível em: <https://webfoundation.org/about/vision/history-of-the-web/>. Acesso em: 23 jun. 2024.

ZAPIER. Zapier Documentation, 2024. Disponível em: <https://zapier.com/>. Acesso em: 23 maio 2024.

APÊNDICE A – SQL para Criação das Entidades do Banco de Dados

SQL para criar a entidade “*users*”

```
-- public.users definition

-- Drop table

-- DROP TABLE public.users;

CREATE TABLE public.users (
	"uuid" varchar DEFAULT "::character varying NOT NULL,
	created_at timestamptz DEFAULT now() NOT NULL,
	username text DEFAULT "::text NOT NULL,
	passhash text DEFAULT "::text NOT NULL,
	cpf text DEFAULT "::text NOT NULL,
	cellphone numeric DEFAULT '81999999999'::numeric NOT NULL,
	email text DEFAULT 'EMPTY'::text NOT NULL,
	user_type public."users_type" DEFAULT 'coordinator'::users_type NOT NULL,
	CONSTRAINT users_pkey PRIMARY KEY (uuid)
);
```

SQL para criar a entidade “*environments*”

```
-- public.environments definition

-- Drop table

-- DROP TABLE public.environments;

CREATE TABLE public.environments (
```

```
	id int8 GENERATED BY DEFAULT AS IDENTITY( INCREMENT BY 1
MINVALUE 1 MAXVALUE 9223372036854775807 START 1 CACHE 1 NO CYCLE)
NOT NULL,
	"name" text DEFAULT ''::text NOT NULL,
	mqtt_topic text DEFAULT ''::text NOT NULL,
	created_at timestamptz DEFAULT now() NOT NULL,
	message text DEFAULT 'on'::text NULL,
	"isActive_flag" bool DEFAULT false NULL,
	CONSTRAINT environments_mqtt_topic_key UNIQUE (mqtt_topic),
	CONSTRAINT environments_name_key UNIQUE (name),
	CONSTRAINT environments_pkey PRIMARY KEY (id)
);
```

SQL para criar a entidade “*accessValidation*”

```
-- public."accessValidation" definition

-- Drop table

-- DROP TABLE public."accessValidation";

CREATE TABLE public."accessValidation" (
	"uuid" uuid DEFAULT gen_random_uuid() NOT NULL,
	validation_flag bool DEFAULT false NOT NULL,
	entry_time timetz NULL,
	exit_time timetz NULL,
	created_at timestamptz DEFAULT now() NOT NULL,
	environments_id int8 NOT NULL,
	users_uuid varchar NOT NULL,
	"access_type" public."access_type" DEFAULT 'limited'::access_type NULL,
	"limitAccess_flag" bool DEFAULT false NOT NULL,
	CONSTRAINT "accessValidantion_pkey" PRIMARY KEY (uuid),
```

```
	CONSTRAINT unique_uuid_id UNIQUE (users_uuid, environments_id)
);

-- public."accessValidation" foreign keys

ALTER TABLE public."accessValidation" ADD CONSTRAINT
"accessValidation_environments_id_fkey" FOREIGN KEY (environments_id)
REFERENCES public.environments(id) ON DELETE CASCADE ON UPDATE
CASCADE;
ALTER TABLE public."accessValidation" ADD CONSTRAINT
"accessValidation_users_uuid_fkey" FOREIGN KEY (users_uuid) REFERENCES
public.users("uuid") ON DELETE CASCADE ON UPDATE CASCADE;
```

SQL para criar a entidade “*mqtt_devices*”

```
-- public.mqtt_devices definition

-- Drop table

-- DROP TABLE public.mqtt_devices;

CREATE TABLE public.mqtt_devices (
	mqtt_device text NOT NULL,
	environments_id int8 NULL,
	enabled bool DEFAULT true NULL,
	"uuid" uuid DEFAULT gen_random_uuid() NOT NULL,
	created_at timestamptz DEFAULT now() NOT NULL,
	CONSTRAINT mqtt_devices_pkey PRIMARY KEY (mqtt_device)
);
```

SQL para criar a entidade “*mqtt_devices_history*”

```
-- public.mqtt_device_history definition

-- Drop table

-- DROP TABLE public.mqtt_device_history;

CREATE TABLE public.mqtt_device_history (
	id int8 GENERATED BY DEFAULT AS IDENTITY( INCREMENT BY 1 MINVALUE 1 MAXVALUE 9223372036854775807 START 1 CACHE 1 NO CYCLE) NOT NULL,
	username text NULL,
	user_uuid text NULL,
	mqtt_device text NULL,
	status text NULL,
	"token" text NULL,
	history_uuid uuid DEFAULT gen_random_uuid() NOT NULL,
	created_at timestamptz DEFAULT now() NOT NULL,
	CONSTRAINT mqtt_device_history_pkey PRIMARY KEY (id)
);
```

ANEXO A – Telas do Não Modificadas do Trabalho de (BEZERRA, 2024)

Figura 44 - Tela de *Splash.*

Fonte: (BEZERRA, 2024).

Figura 45 - Tela de Configuração de Ambiente.

Fonte: (BEZERRA, 2024).

Figura 46 - Tela de Gestão de Usuário.

Fonte: (BEZERRA, 2024).