UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Felipe Pimenta Bernardo

Análise e desenvolvimento de um Sistema Web voltado para Aplicações IoT

São Carlos

2023

Felipe Pimenta Bernardo

Análise e desenvolvimento de um Sistema Web voltado para Aplicações IoT

Monografia apresentada ao Curso de Engenharia Elétrica com Ênfase em Eletrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. José Roberto Boffino de Almeida Monteiro

São Carlos

2023

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Bernardo, Felipe Pimenta
B518a Análise e desenvolvimento de um sistema Web voltado para aplicações IoT / Felipe Pimenta Bernardo; orientador José Roberto Boffino de Almeida Monteiro. São Carlos, 2023.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2023.

1. Internet da Coisas. 2. Sistemas Web. 3. MQTT. 4. Redes de computadores. 5. Protocolos de Internet. 6. Interface de usuário. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Felipe Pimenta Bernardo

Título: “Análise e desenvolvimento de um Sistema Web voltado para Aplicações IoT”

Trabalho de Conclusão de Curso defendido e aprovado em 18 / 12 / 2023 ,

com NOTA 10,0 (dez , zero) , pela Comissão Julgadora:

Prof. Associado José Roberto Boffino de Almeida Monteiro - Orientador - SEL/EESC/USP

Prof. Dr. Pedro de Oliveira Conceição Junior - SEL/EESC/USP

Prof. Associado Evandro Luis Linhari Rodrigues - SEL/EESC/ USP (docente aposentado)

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Professor Associado José Carlos de Melo Vieira Júnior

"As coisas que temos de aprender antes de poder fazê-las, aprendemos fazendo-as."
Aristóteles

RESUMO

BERNARDO, F. **Análise e desenvolvimento de um Sistema Web voltado para Aplicações IoT**. 2023. 70p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2023.

Sistemas Web voltados para aplicações IoT constituem um componente vital no cenário tecnológico, servindo como ponte entre os dispositivos IoT e as interfaces de usuário, facilitando a troca e o processamento de dados, e permitindo um melhor aproveitamento do potencial liberado pelo constante avanço dos dispositivos IoT. Nesse contexto, este trabalho aborda o desenvolvimento e a análise de um sistema Web voltado para aplicações IoT, destacando a estrutura geral do sistema, os componentes individuais que o compõem e os métodos de comunicação utilizados entre eles. Mais especificamente, o sistema proposto visa possibilitar o monitoramento e operação remota de uma máquina elétrica através de uma interface (*dashboard*) interativa, possibilitando também o armazenamento e a análise dos dados recebidos pelos dispositivos IoT conectados à máquina. A fase de desenvolvimento é dedicada ao projeto e à implementação dos componentes essenciais para a composição de um sistema Web robusto e funcional projetado para enfrentar os desafios e atender às demandas específicas descritas, abordando não só os componentes como também como estes interagem entre si para cumprir as funcionalidades esperadas para o sistema. Os resultados mostram que a integração desses componentes, sendo eles, um *Broker* MQTT na AWS, uma interface interativa hospedada na Vercel, uma API sem servidor na Vercel e um banco de dados MongoDB no Atlas, exemplificam o potencial de se combinar diversas tecnologias para comunicação, processamento de dados e interação do usuário. Através de uma análise abrangente, feita por meio de testes e auditorias, este trabalho discorre não somente sobre os resultados obtidos com o desenvolvimento dos componentes citados, como também sobre as dificuldades e limitações inerentes a esse processo encontradas durante o desenvolvimento, concluindo que, apesar das limitações mencionadas, o sistema desenvolvido cumpre os objetivos propostos, servindo não somente como uma prova de conceito como também uma base para desenvolvimento e aprimoramentos futuros.

Palavras-chave: Aplicações IoT. Sistemas Web. MQTT. Redes de Computadores. Protocolos de Internet. Interface de usuário.

ABSTRACT

BERNARDO, F. **Analysis and development of a Web System aimed at IoT Applications**. 2023. 70p. Monograph (Conclusion Course Paper) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2023.

Web systems aimed at IoT applications constitute a vital component in the technological scenario, serving as a bridge between IoT devices and user interfaces, facilitating the exchange and processing of data, and allowing better use of the potential released by the constant advancement of IoT devices. In this context, this work addresses the development and analysis of a Web system aimed at IoT applications, highlighting the general structure of the system, the individual components that make it up and the communication methods used between them. More specifically, the proposed system aims to enable the remote monitoring and operation of an electrical machine through an interactive *dashboard*, also enabling the storage and analysis of data received by IoT devices connected to the machine. The development phase is dedicated to the design and implementation of the essential components for the composition of a robust and functional Web system designed to face the challenges and meet the specific demands described, addressing not only the components but also how they interact with each other to fulfill the expected functionalities of the system. The results show that the integration of these components, namely an MQTT *Broker* on AWS, an interactive *dashboard* hosted on Vercel, a serverless API on Vercel and a MongoDB database on Atlas, exemplify the potential to combine different technologies for communication, data processing and user interaction. Through a comprehensive analysis, carried out through tests and audits, this work discusses not only the results obtained with the development of the aforementioned components, but also the difficulties and limitations inherent to this process encountered during development. Through a comprehensive analysis, carried out through tests and audits, this work discusses not only the results obtained with the development of the aforementioned components, but also the difficulties and limitations inherent to this process encountered during development, concluding that, despite the aforementioned limitations, the developed system meets the proposed objectives, serving not only as a proof of concept but also a basis for future development and improvements.

Keywords: IoT applications. Web system. MQTT. Computer Networks. Internet protocols. User interface.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ARIA	*Accessible Rich Internet Applications*
AWS	*Amazon Web Services*
CDN	*Content Delivery Network*
CLI	*Command Line Interface*
CSS	*Cascading Style Sheets*
DB	*Database*
DBMS	*Database Management System*
DOM	*Document Object Model*
DNS	*Domain Name System*
EC2	*Elastic Compute Cloud*
FaaS	*Function as a Service*
FTP	*File Transfer Protocol*
HTML	*Hypertext Markup Language*
HTTP	*Hyper Text Transfer Protocol*
HTTPS	*Hyper Text Transfer Protocol Secure*
IETF	*Internet Engineering Task Force*
IoT	*Internet of Things* (Internet das Coisas)
IP	*Internet Protocol*
ISP	*Internet Service Provider*
JS	JavaScript
JSON	*JavaScript Object Notation*
LWT	*Last Will and Testament*
MQTT	*Message Queuing Telemetry Transport*

NPM	*Node Package Manager*
OSI	*Open Systems Interconnection*
PC	*Personal Computer*
PM2	*Process Manager 2*
POO	Programação Orientada a Objetos
PWM	*Pulse Width Modulation*
QoS	*Quality of Service*
RFC	*Request For Comments*
SEO	*Search Engine Optimization*
SQL	*Structured Query Language*
SSH	*Secure Shell*
SPA	*Single Page Application*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
UI	*User Interface*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
UX	*User Experience*
WS	WebSocket
WSS	WebSocket Secure

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

A Internet das Coisas (IoT) tornou-se uma força motriz em diversos setores, com implicações significativas para os setores elétrico e industrial em geral, oferecendo um vasto potencial para monitoramento, controle e otimização aprimorados de sistemas elétricos, conectando dispositivos, sensores e equipamentos à Internet (BAYASGALAN; BYAMBASUREN; TUDEVDAGVA, 2023).

A integração de aplicações IoT em sistemas elétricos tem implicações de longo alcance, permitindo o monitoramento em tempo real, manutenção preditiva e economias substanciais de custos. Esta abordagem transformadora não só revoluciona a gestão energética, mas também aumenta as operações industriais, inaugurando uma era de sistemas inteligentes e interligados que redefinem as práticas tradicionais e estabelecem as bases para o crescimento sustentável e a competitividade (BAYASGALAN; BYAMBASUREN; TUDEVDAGVA, 2023).

Nesse contexto, os sistemas Web desempenham um papel fundamental na implementação e operação das aplicações IoT, atuando como a interface entre os dispositivos IoT e os usuários, facilitando a comunicação eficiente, a troca de dados e a interação em tempo real.

A importância dos sistemas Web voltados para aplicações IoT reside na sua capacidade de fornecer uma plataforma coesa para monitorar, controlar e analisar dispositivos conectados remotamente, fornecendo aos usuários *dashboards* acessíveis e fáceis de usar, melhorando a experiência de usuário no geral. Além disso, os sistemas Web permitem o armazenamento e o processamento de dados gerados por dispositivos IoT, contribuindo para a tomada de decisões informadas através de análises detalhadas desses dados.

Os sistemas Web são capazes de garantir a colaboração eficaz entre vários componentes, como *Brokers* MQTT, *dashboards* interativos, APIs e bancos de dados, possuindo um potencial de amalgamar diversas tecnologias para comunicação, processamento de dados e interação com o usuário.

À medida que as aplicações IoT continuam a evoluir, os sistemas Web voltados para aplicações IoT se fazem cada vez mais relevantes, promovendo um ecossistema inteligente e interconectado que redefine as práticas tradicionais e melhora a eficiência geral das aplicações IoT.

Em resumo, a integração de aplicações IoT com sistemas Web tem o potencial de redefinir os setores elétrico e industrial, permitindo monitoramento em tempo real,

manutenção preditiva e eficiência operacional. Os sistemas Web desempenham um papel vital, oferecendo interfaces acessíveis, *dashboards* amigáveis e facilitando o armazenamento e processamento eficaz de dados. Essa colaboração entre IoT e sistemas Web cria um ecossistema inteligente, interconectado e eficiente, moldando o futuro das aplicações IoT.

1.2 Objetivos

Nesse contexto, entende-se a importância do desenvolvimento de sistemas Web projetados para atender às necessidades específicas de aplicações IoT, como forma de aproveitar o potencial para comunicação e processamento de dados liberado por estas aplicações.

Assim, este trabalho tem como principais objetivos:

- Projetar e desenvolver um sistema Web de baixo custo e complexidade voltado para o monitoramento e operação remota de uma máquina elétrica;
- Possibilitar o armazenamento e análise dos dados coletados pelos sensores ligados à máquina;
- Analisar os resultados obtidos com o sistema Web desenvolvido, considerando o contexto em que se propôs sua implementação;
- Discutir as dificuldades de se desenvolver tal sistema, abordando as tecnologias e suas respectivas complexidades e custos;
- Refletir acerca das limitações do sistema apresentado, considerando as dificuldades de se expandir o sistema para escalas mais próximas de aplicações reais.

Espera-se que, com isso, este trabalho contribua para uma compreensão mais ampla de como os sistemas Web podem ser estrategicamente projetados para atender aos requisitos das aplicações IoT, estabelecendo uma base valiosa para o desenvolvimento futuro desse tipo de aplicação, preparando o terreno para inovações e melhorias.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão revisados alguns dos conceitos e tecnologias fundamentais para o desenvolvimento deste projeto. Nas sessões a seguir será feita uma contextualização acerca de Internet das Coisas e Redes de Computadores, seguida por um apanhado geral dos principais protocolos de rede necessários para a transmissão de informações dentro do sistema proposto — HTTP, MQTT e WebSocket. Por fim, o capítulo discorre sobre os dois "lados" do desenvolvimento de Sistemas Web — o Front-End e o Back-End — destacando onde se encaixam os elementos de um sistema nessa divisão.

2.1 Internet das Coisas

Internet das Coisas (IoT) é um termo utilizado para descrever um ramo de aplicações tecnológicas baseadas na interconexão de dispositivos eletrônicos ou "coisas" através da internet. Esses dispositivos podem variar desde itens comuns de uso diário, como eletrodomésticos, até sistemas complexos, como máquinas industriais, veículos, etc, possibilitando uma gama de aplicações muito ampla.

Os sistemas IoT facilitam a aquisição em tempo real de dados destes objetos, possibilitando monitora-los e controla-los remotamente. Os dados gerados pelos dispositivos IoT podem ser processados, analisados e usados para desencadear ações automatizadas, permitindo a tomada de decisões informadas, a manutenção preditiva e uma maior eficiência operacional em vários domínios.

2.2 Redes de computadores

A Internet é uma rede global de computadores que interconecta diversos dispositivos, desde sistemas tradicionais, como PCs, até uma variedade crescente de dispositivos modernos, como TVs, smartphones, carros, etc. Esses dispositivos são chamados de sistemas finais (Figura 1). A comunicação entre sistemas finais ocorre por meio de enlaces (*links*) e comutadores (*switches*) de pacotes, como roteadores (KUROSE; ROSS, 2014).

Os sistemas finais acessam a Internet por meio de Provedores de Serviços de Internet, ou ISPs (*Internet Service Providers*), que oferecem diferentes tipos de acesso, como banda larga, sem fio, etc. Os ISPs interconectam-se formando uma infraestrutura complexa. Para garantir que a informação seja transmitida de forma eficiente e confiável entre usuários finais, bem como a interoperabilidade de dispositivos e sistemas por esta infraestrutura tão complexa é necessária a adoção de padrões e protocolos.

Tais protocolos, como TCP/IP, controlam a transmissão de informações na Internet, e os padrões são estabelecidos pela IETF (*Internet Engineering Task Force* — Força de

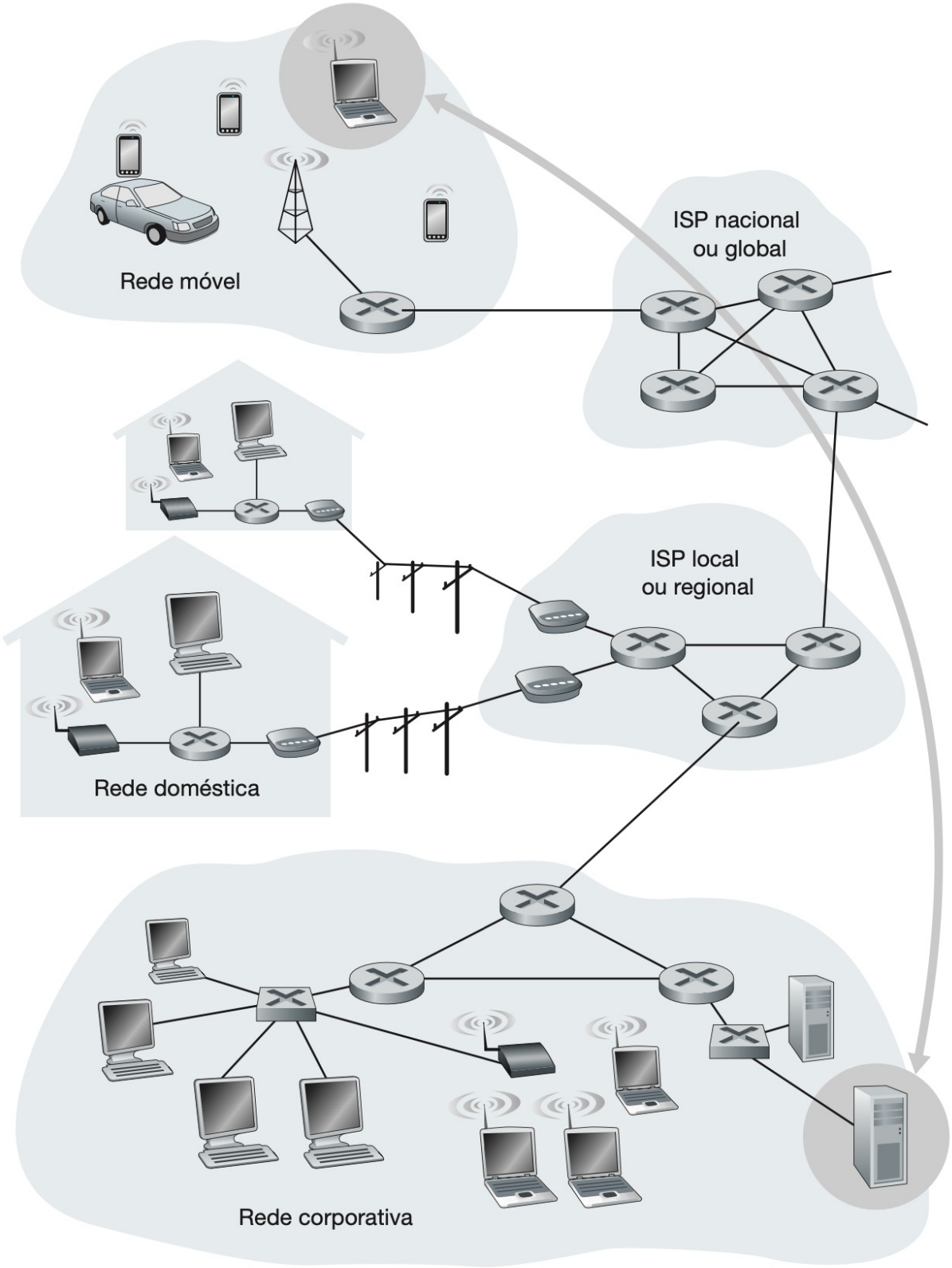


Figura 1 – Interação entre sistemas finais
Fonte: (KUROSE; ROSS, 2014).

Trabalho de Engenharia da Internet) por meio de documentos chamados RFCs (*Request For Comments* — pedido de comentários) (KUROSE; ROSS, 2014). Todas as interações na Internet, entre duas ou mais entidades remotas, são regidas por um protocolo. Por exemplo, protocolos em hardware de dois computadores conectados fisicamente controlam o fluxo de bits no cabo entre eles; protocolos de controle de congestionamento em sistemas finais gerenciam a taxa de transmissão de pacotes; enquanto os protocolos em roteadores determinam o caminho de um pacote.

Um protocolo de rede é comparável a um protocolo humano, com a diferença de que as entidades envolvidas são componentes de hardware ou software de dispositivos habilitados para rede. Um exemplo familiar de protocolo de rede é o processo de solicitação a um servidor web, onde o formato e a ordem das mensagens, como requisições e respostas, são definidos por um protocolo.

A Internet e as redes de computadores usam vários tipos de protocolos para realizar diferentes tarefas de comunicação, variando em complexidade. Sua estrutura é construída a partir de dois modelos principais: o modelo OSI (Open Systems Interconnection) e o modelo TCP/IP.

O modelo OSI consiste em sete camadas, cada uma com responsabilidades específicas, desde a Camada Física, que trata das conexões de hardware, até a Camada de Aplicação que fornece serviços aos usuários finais (FROEHLICH, 2023). O modelo TCP/IP se alinha estreitamente com o modelo OSI, compreendendo a camada de enlace, a camada de Internet, a camada de transporte e a camada de aplicação. Ambos os modelos oferecem uma estrutura conceitual para a compreensão de sistemas de rede, detalhando funções desde conexões físicas até serviços em nível de aplicação cruciais para a comunicação do usuário final (HAMMOUDI; KHANFAR, 2022).

Aprofundar-se nestes modelos foge ao escopo deste trabalho, bastando destacar que ambos propõem a estruturação da Internet em camadas, onde as camadas mais baixas estão mais relacionadas aos componentes físicos, enquanto as camadas mais altas, como a Camada de Aplicação, representam um nível mais "abstrato", interagindo diretamente com as aplicações do usuário final, serviços e interface de usuário.

2.2.1 HTTP

No contexto da Camada de Aplicação, um dos protocolos mais essenciais para a comunicação na Internet é o HTTP (*Hypertext Transfer Protocol*), utilizado, por exemplo, para carregar as informações de uma página Web acessada pelo usuário. Trata-se de um protocolo para transferência de hipertexto, isto é, texto que pode incluir, além de blocos de texto, informações como imagens, vídeos, entre outras mídias. A seguir serão apresentadas algumas das principais características deste protocolo, de acordo com a documentação oficial da Mozilla — MDN Web Docs (MOZILLA, 2023a).

Uma das caraterísticas principais do HTTP é que ele segue um modelo de **requisição** e **resposta**, onde um cliente (normalmente um navegador da Web) envia uma **requisição** a um servidor da Web que, por sua vez, processa essa **requisição** e retorna uma **resposta** com os dados ou recursos solicitados. Isso significa que o cliente inicia a comunicação abrindo uma conexão — neste caso, uma conexão TCP (*Transmission Control Protocol*) — com o servidor e essa conexão é mantida aberta até que a **resposta** seja recebida pelo cliente.

Para que a requisição seja enviada o primeiro passo é fornecer um endereço na forma de um *Uniform Resource Identifier* (URI) ou *Uniform Resource Locator* (URL). O URI/URL identifica o recurso que está sendo solicitado e a sua localização. Normalmente, consiste em um esquema — nesse caso, "`http`" ou "`https`" —, nome de domínio — por exemplo, "`dominio.com.br`" — e caminho para o recurso — por exemplo, "`/index.html`" — entre outras possibilidades, como especificado em (MOZILLA, 2023b).

As requisições HTTP podem ser feitas através de diferentes métodos que indicam a ação que se deseja executar. Existem diversos métodos HTTP, como visto em (MOZILLA, 2023c). Dentre eles, alguns dos mais utilizados são:

- **GET**: Solicita o retorno de um recurso sem enviar ou modificar nada no servidor.
- **POST**: Envia dados a serem processados pelo servidor, como um formulário.
- **PUT**: Atualiza o recurso ou cria um novo.
- **DELETE**: Remove o recurso.
- **PATCH**: Aplica modificações parciais a um recurso. Normalmente é usado para atualizar um recurso com novas informações.
- **HEAD**: Recupera os cabeçalhos de um recurso, semelhante ao método GET, mas sem o corpo da resposta.
- **OPTIONS**: Recupera informações sobre as opções de comunicação para o recurso de destino. Geralmente é usado para verificar quais métodos HTTP são suportados por um servidor web.

As requisições HTTP devem seguir uma estrutura definida para que seja possível garantir que a informação transmitida seja corretamente interpretada por ambos cliente e servidor, como visto em (MOZILLA, 2023d). Por isso, tanto a requisição quanto a resposta incluem *headers*, ou cabeçalhos (MOZILLA, 2023e). Esses cabeçalhos contém metadados sobre a requisição ou resposta, como tipo de conteúdo, comprimento do conteúdo, *tokens* para autorização e muito mais. Algumas solicitações e respostas HTTP podem incluir um *body*, ou corpo de mensagem, que contém dados associados à solicitação ou resposta. Por

exemplo, quando um cliente envia um formulário, é feita uma requisição com o método POST em que os dados do formulário são passados no corpo da requisição.

Após processarem as requisições HTTP recebidas, os servidores enviam uma resposta ao cliente. As respostas HTTP incluem um código de status de três dígitos que indica como a requisição foi processada pelo servidor, indicando, por exemplo, se tudo ocorreu como esperado ou se houve algum erro. Existem diversos códigos de status HTTP, como pode ser visto em (MOZILLA, 2023f). Dentre eles, alguns dos mais comuns estão:

- **200 — OK**: A solicitação foi bem-sucedida e a resposta contém os dados solicitados.
- **404 — Não encontrado**: O recurso solicitado não existe no servidor.
- **500 — Erro interno do servidor**: indica um erro do servidor que o impediu de processar a solicitação.

Para aumentar a segurança, requisições HTTP podem ser protegidas usando HTTPS (HTTP *Secure*). O protocolo HTTPS adiciona uma camada de criptografia (SSL/TLS) à comunicação, garantindo que os dados transmitidos entre o cliente e o servidor sejam seguros e não possam ser facilmente interceptados (MOZILLA, 2023g). Para que uma requisição seja feita utilizando este protocolo, é necessário que o servidor que está recebendo esta requisição possua um certificado SSL válido, dentre outros requisitos (MOZILLA, 2023h).

Por fim, vale mencionar que o método HTTP também oferece suporte à persistência de conexão, onde diversas requisições e respostas HTTP podem ser trocadas em uma única conexão TCP. Isso é conhecido como "*keep-alive*", ou "manter vivo" (MOZILLA, 2023i).

2.2.2 MQTT

Apesar de ser um dos protocolos mais utilizados na internet, o HTTP nem sempre é o mais adequado para determinadas aplicações tendo em vista algumas de suas características e limitações. Dentre os outros protocolos existentes, um que se apresenta como uma excelente alternativa no contexto de aplicações IoT, sendo considerado o padrão da indústria nesse ramo, é o protocolo MQTT (*Message Queuing Telemetry Transport*). Algumas de suas principais características serão apresentadas a seguir.

O MQTT é um protocolo de mensagens que, assim como o HTTP, opera sobre o protocolo TCP/IP, ou seja, no nível da camada de aplicação (HIVEMQ, 2020). Entretanto, diferentemente do HTTP, onde a comunicação é feita entre 2 partes apenas — cliente e servidor (Figura 2) — no MQTT a comunicação se baseia no esquema *publish/subscribe* (publicação e inscrição), em que os clientes se comunicam entre si através de um intermediador, chamado *Broker* (Figura 3).

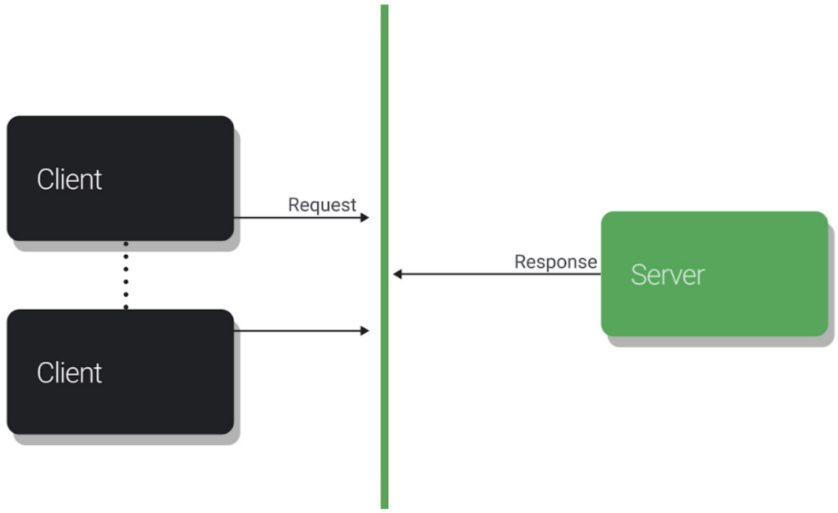


Figura 2 – Esquemático da comunicação usando o protocolo HTTP.
Fonte: (BENSAKHRIA, 2020).

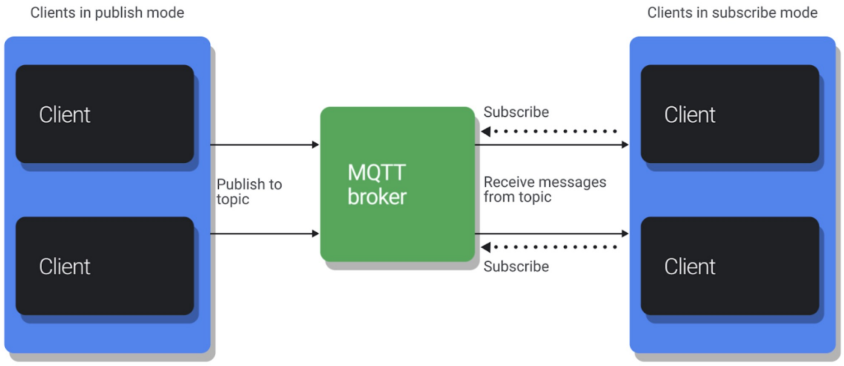


Figura 3 – Esquemático da comunicação usando o protocolo MQTT.
Fonte: (BENSAKHRIA, 2020).

O *Broker* MQTT é um servidor que atua como intermediário, sendo responsável por rotear e entregar mensagens entre os clientes conectados a ele. Vários clientes podem se conectar ao mesmo *Broker*, permitindo que uma grande quantidade deles se comuniquem entre si. Ele monitora quais estão inscritos em quais tópicos e garante que as mensagens publicadas por eles sejam entregues aos clientes apropriados com base nos tópicos em que estão inscritos (HIVEMQ, 2020).

Essa comunicação é feita utilizando um modelo de **publicações** e **inscrições** baseado em tópicos, que funciona da seguinte forma: os clientes podem se inscrever em tópicos específicos dos quais desejam receber mensagens, de forma que as mensagens publicadas em um determinado tópico serão recebidas por todos os clientes inscritos naquele tópico. Os tópicos, por sua vez, têm estrutura hierárquica, com níveis separados por barras (“`/`”). Por exemplo, “`sensores/temperatura`” e “`sensores/umidade`” são dois tópicos diferentes. Os clientes podem se inscrever em tópicos individuais, vários tópicos ou usar caracteres curinga para se inscrever em grupos de tópicos (HIVEMQ, 2020).

Os clientes MQTT também são responsáveis por fazer as publicações: eles enviam mensagens ao *Broker*, especificando o tópico no qual a mensagem deve ser publicada. Por padrão, as mensagens contêm um *payload*, um ID de mensagem e o nível de QoS desejado. O *Broker* então recebe a mensagem e fica encarregado de colocá-la na fila para entrega aos clientes inscritos no tópico especificado. As mensagens podem ser publicadas nos tópicos utilizando um de três níveis diferentes de Qualidade de Serviço, ou QoS (*Quality of Service*). A existência de diferentes tipos de QoS no protocolo MQTT o tornam extremamente flexível e adaptável (HIVEMQ, 2020).

- **QoS 0:** No máximo uma vez — As mensagens são entregues ao destinatário no máximo uma vez, sem confirmação.
- **QoS 1:** Pelo menos uma vez — É garantido que as mensagens serão entregues pelo menos uma vez, mas podem ser entregues várias vezes.
- **QoS 2:** Exatamente uma vez — É garantido que as mensagens serão entregues exatamente uma vez.

O protocolo MQTT suporta retenção de mensagens, onde o *Broker* armazena a última mensagem publicada em um tópico. Isso permite que, quando um novo cliente se inscrever neste tópico, o *Broker* possa enviar a mensagem retida imediatamente ao novo cliente inscrito (HIVEMQ, 2020). Além disso, os clientes podem fornecer mensagens específicas relacionadas ao "ciclo de vida" de sua conexão com o *Broker*. A primeira, "*Birth Message*" ("Mensagem de Nascimento"), é publicada assim que o cliente se conecta ao *Broker*. Analogamente, ao se desconectar, o cliente publica a "*Close Message*" ("Mensagem de Desconexão"). Entretanto, caso essa conexão seja interrompida de forma inesperada a mensagem enviada é a chamada de "*Last Will and Testament (LWT)*" ("Última Vontade e Testamento"). Todas as mensagens do ciclo de vida de um cliente mencionadas acima, assim como as demais mensagens publicadas, devem indicar em qual tópico e com qual QoS serão publicadas, podendo também ser retidas (HIVEMQ, 2020).

Os clientes estabelecem uma conexão com o *Broker* MQTT por TCP/IP. O *Broker* escuta em uma porta específica (geralmente 1883 para conexões não criptografadas e 8883 para conexões criptografadas). Os clientes podem se conectar com um nome de usuário e senha, e o *Broker* pode ser configurado para oferecer suporte a conexões seguras usando TLS/SSL.

2.2.3 WebSocket

WebSocket é um protocolo de comunicação que permite a criação de um canal de comunicação bidirecional, baseado em enventos, em tempo real e *full-duplex* entre um cliente (normalmente um navegador da Web) e um servidor. Ele fornece um método

eficiente e de baixa latência para transmissão de dados e permite que o cliente e o servidor iniciem a comunicação sem a necessidade de ciclos frequentes de requisição-resposta como no caso do HTTP (LOMBARDI, 2015).

O processo começa com um *handshake* (aperto de mão) WebSocket. O cliente inicia o *handshake* enviando uma solicitação HTTP ao servidor, incluindo o cabeçalho "`Upgrade: websocket`", entre outros cabeçalhos específicos do WebSocket. O servidor, ao receber a solicitação de *handshake*, verifica a compatibilidade e, se aceito, responde com um código de status HTTP 101, indicando que será feita a troca de protocolos (MELNIKOV; FETTE, 2011). Assim que o *handshake* for concluído, a conexão faz a transição do protocolo HTTP para o protocolo WebSocket. Vale ressaltar que a conexão permanece aberta durante a comunicação, permitindo uma troca eficiente de dados.

As conexões WebSocket podem usar subprotocolos para adicionar funcionalidades adicionais e modificar seu comportamento, permitindo uma gama ainda maior de aplicações. Os subprotocolos são acordados durante o *handshake* e especificam como os dados são interpretados. Um exemplo de subprotocolo é o "*MQTT over WebSockets*" ("MQTT via WebSockets"). Esse subprotocolo permite que o MQTT seja empregado em aplicações web, proporcionando uma comunicação bidirecional em tempo real entre clientes web e *Broker* MQTT, por meio de uma conexão única e de longa duração. Essa abordagem é particularmente útil quando as conexões MQTT tradicionais são restritas e as conexões WebSocket são preferidas ou obrigatórias, como em navegadores da web (MELNIKOV; FETTE, 2011).

As conexões WebSocket podem ser protegidas usando o protocolo Transport Layer Security (TLS), criando uma conexão WebSocket Secure (WSS). Isso garante a confidencialidade e integridade dos dados transmitidos pela conexão.

2.3 Sistemas Web

2.3.1 Interfaces de usuário e *Front-End*

O termo Front-End é usado para se referir à parte de um sistema Web que é executada do lado do cliente (*client-side*), composta basicamente por interfaces (páginas) Web. O desenvolvimento dessas interfaces Web (Front-End) envolve a criação de componentes visuais e interativos para aplicações Web e sites com os quais os usuários interagem por meio de seus navegadores Web. Essas interfaces são construídas basicamente a partir de 3 linguagens: HTML, CSS e JavaScript.

HTML *Hypertext Markup Language* (Linguagem de Marcação de Hipertexto) — ou HTML — é a principal linguagem utilizada para estruturar conteúdo em páginas da web. Ela se baseia na utilização de *tags* para criação de elementos como títulos, parágrafos, links, listas, imagens, formulários, dentre outros elementos. As *tags* seguem um padrão

que consiste no nome do elemento envolto por "<" e ">", como, por exemplo, `<h1>` para cabeçalhos, `<p>` para parágrafos e `<img>` para imagens. Algumas *tags* permitem a inserção de conteúdo dentro de si. Neste caso, elas devem ser abertas e fechadas após o conteúdo, como no exemplo a seguir: `<h1>Um cabeçalho</h1>`. (POWELL, 2010)

CSS Enquanto o HTML é responsável pela estrutura do conteúdo de uma interface Web, a sua estilização é feita utilizando *Cascading Style Sheets* (Folhas de Estilo em Cascata) — ou CSS — uma linguagem que se baseia na listagem de regras que serão aplicadas em determinados elementos de acordo com os seletores especificados. As regras CSS definem como os elementos devem ser exibidos, definindo o posicionamento, as cores, as fontes e outros tantos aspectos visuais da interface. Já os seletores são responsáveis por indicar em quais elementos HTML específicos serão aplicados determinados conjuntos de regras. (POWELL, 2010)

JavaScript Enquanto apenas HTML e CSS são suficientes para a construção da estrutura visual de uma página e até mesmo garantir um pequeno nível de interatividade, qualquer funcionalidade ligeiramente mais complexa que se deseje implementar será feita utilizando JavaScript. Trata-se de uma linguagem leve, baseada em objetos, dinâmica, multi-paradigma, com suporte para POO (Programação Orientada a Objetos), que é interpretada nativamente por todos os navegadores ou por um ambiente de tempo de execução (*runtime*) como Node.js (POLLOCK, 2013). Sua interação com os elementos de uma página Web é feita através do *Document Object Model* (Modelo de Objeto de Documento) — ou DOM — que é uma representação da estrutura HTML, podendo manipular e alterar o conteúdo e o comportamento dessas páginas.

Apesar da grande gama de aplicações já possibilitadas por essas linguagens "puras", a demanda por aplicações Web cada vez mais complexas em prazos cada vez menores fomentou a criação de ferramentas que auxiliam nesse processo. Estruturas e bibliotecas — os chamados *frameworks* de desenvolvimento Web — como React, Angular, Vue.js e jQuery, agilizam o desenvolvimento de interfaces Web. Eles fornecem componentes e padrões pré-construídos para acelerar o desenvolvimento, facilitar o reaproveitamento de código e garantir consistência, além de fornecer ferramentas que auxiliam a gerenciar estados, lidar com o roteamento entre as páginas do projeto, simplificar o tratamento de requisições e muito mais. (MOZILLA, 2023j)

O processo de desenvolvimento dessas interfaces requer não somente conhecimento técnico, como também conhecimento de princípios de design e uma compreensão do comportamento e das expectativas do usuário. Esse processo muitas vezes requer a criação de designs que se adaptem a diferentes tamanhos de tela e dispositivos. Para isso, o design Web responsivo envolve o uso de *media queries* CSS e layouts flexíveis para garantir que a interface tenha uma boa aparência e funcione bem em dispositivos de vários tamanhos,

desde *desktops* a *smartphones*. A colaboração com designers de UX (*User Experience*) e UI (*User Interface*) é fundamental para criar uma interface que seja visualmente atraente, fácil de usar e alinhada às expectativas do usuário.

Além disso, as interfaces da Web devem ser projetadas e desenvolvidas tendo em mente a acessibilidade para garantir que possam ser usadas por todos os tipos de usuários, incluindo pessoas com deficiência. Tamanhos de fonte e níveis mínimos de contraste auxiliam usuários com dificuldades para enxergar, por exemplo. Técnicas como HTML semântico, funções ARIA (*Accessible Rich Internet Applications*) e navegação por teclado são empregadas para facilitar a interação de usuários que necessitam de *softwares* leitores de tela ou que possuam algum outro tipo de limitação (CUNNINGHAM, 2012).

2.3.2 Bancos de dados, APIs e *Back-End*

A interface de usuário é apenas uma das várias partes que compõem um sistema Web. Sistemas complexos que apresentam funcionalidades mais avançadas — como armazenamento de informações, interação com outros sistemas, validações e autenticações — requerem a implementação de outros componentes que atuam "por trás" da interface de usuário, isto é, do lado do servidor (*server-side*). Esse conjunto de componentes é entendido como o Back-End de um sistema, e engloba componentes como bancos de dados, APIs (*Application Programming Interface*), sistemas de autenticação, algoritmos de validação e tratamento de dados, entre muitos outros. Tendo em mente o escopo deste trabalho, optou-se por abordar apenas os seguintes componentes de um Back-End: banco de dados e API.

Banco de Dados Um banco de dados é uma coleção estruturada de informações gerenciada por um Sistema de Gerenciamento de Banco de Dados — ou *Database Management System* (DBMS). O DBMS é responsável por permitir aos usuários criar e definir bancos de dados, consultar e modificar dados usando uma linguagem especializada, suportar grandes armazenamentos de dados, garantir durabilidade diante de falhas, erros ou uso indevido e controlar o acesso aos dados, para evitar interações inesperadas e garantir a atomicidade. Existem diferentes tipos de bancos de dados para se adequarem aos diferentes tipos de aplicações. Dois dos principais tipos são bancos de dados relacionais e não relacionais. Os **bancos de dados relacionais** estruturam os dados em tabelas utilizando SQL para manipulá-las. Por usa vez, os **bancos de dados não relacionais** — também chamados de NoSQL — lidam com dados não estruturados ou semiestruturados. Alguns exemplos são bancos orientados a documentos, como MongoDB, bancos de pares *key-value* (valor-chave), como Redis, e bancos de dados gráficos, como Neo4j. (*Structured Query Language*). (GARCIA-MOLINA; ULLMAN; WIDOM, 2008).

API Uma API (*Application Programming Interface*) é um conjunto de regras, protocolos e ferramentas que permite que diferentes aplicações de software se comuniquem entre si, definindo como os componentes de software devem interagir e especificando os métodos e formatos de dados que as aplicações podem usar para solicitar e trocar informações. No caso de APIs Web, esse conjunto de regras se dá geralmente como um padrão para requisições HTTP entre o cliente e o servidor. Existe uma grande variedade de linguagens e *frameworks* para a criação de APIs Web, como ExpressJS para JavaScript, Laravel para PHP, Spring para Java, entre outros (RICHARDSON; AMUNDSEN, 2013).

Tanto o banco de dados quando a API são executados no lado do servidor, ou seja, precisam de servidores para serem hospedados. Entretanto, o gerenciamento de servidores envolve tempo e recursos consideráveis, desde a manutenção do hardware até o tratamento de atualizações de software e medidas de segurança. Nesse contexto hospedagens com arquitetura *serverless* ("sem servidor"), como *Function as a Service* (Função como Serviço), ou FaaS, aliviam esses encargos à medida em que terceirizam essas responsabilidades atribuindo-as a um provedor terceirizado.

No contexto de um sistema Web, os componentes individuais não só precisam funcionar de forma autônoma, mas também colaborar entre si. Cada componente, desde a interface de usuário, até a API e o banco de dados, desempenha uma função única. No entanto, é a colaboração entre eles que possibilita funcionalidades que vão além das suas capacidades isoladas, permitindo que o sistema web atinja um nível de funcionalidade que transcende a soma de suas partes.

3 DESENVOLVIMENTO

O presente capítulo se encabe de relatar o processo de desenvolvimento do sistema Web especificado nos objetivos deste trabalho. Isso será feito apresentando a estrutura geral escolhida para o projeto e, em seguida, relatando o processo de desenvolvimento de cada elemento apresentado na estrutura geral, destacando as tecnologias ou serviços utilizados e buscando sempre salientar os motivos que os levaram a serem escolhidos.

3.1 Arquitetura do sistema

O sistema Web deste trabalho foi projetado considerando o objetivo de monitorar e controlar remotamente uma máquina elétrica (um motor *brushless* DC). A máquina é controlada via Modulação de Largura de Pulso — ou PWM (*Pulse Width Modulation*) — por um microcontrolador da família STM32 que, por sua vez, recebe as informações captadas pelos sensores no motor após estas terem sido convertidas de sinais analógicos para digitais através de um Conversor Analógico-Digital — ou ADC (*Analog-to-Digital Converter*). O microcontrolador também é responsável por estabelecer uma conexão com a Internet e trocar informações com um *Broker* MQTT hospedado em nuvem.

Para a interação com o usuário, será desenvolvido um *dashboard* na forma de uma aplicação Web, também hospedado em nuvem para que possa ser acessado de qualquer lugar, que se conectará ao *Broker* através do protocolo MQTT via WebSocket. Devido à natureza das comunicações através do protocolo MQTT e do Broker utilizado, não há neste a possibilidade de um armazenamento duradouro de informações. Sendo assim, visando fornecer informações mais completas e detalhadas a respeito do histórico de funcionamento da máquina elétrica para a interface do *dashboard*, optou-se pela implementação de um banco de dados (DB) não relacional, intermediado por uma API separada.

O monitoramento e a análise dos dados coletados na máquina elétrica desempenharão um papel fundamental na melhoria do desempenho e na prevenção de erros. Ao avaliar regularmente os dados operacionais, potenciais problemas poderão ser identificados precocemente, permitindo medidas proativas e manutenção.

O escopo deste trabalho abrange apenas a parte dos componentes Web. O acionamento do motor, feito pelo microcontrolador, foi definido em um trabalho adjacente a este, com o qual se acordou o funcionamento do sistema completo. Vale ressaltar que como a comunicação entre o microcontrolador e o sistema Web acontece inteiramente por meio de mensagens MQTT, qualquer dispositivo com conexão à Internet e suporte para o protocolo MQTT poderia se conectar ao *Broker*, fazendo com que o sistema Web seja compatível e adaptável para diferentes dispositivos em diferentes aplicações.

Sendo assim, o sistema Web será, então, composto por 4 componentes:

- ***Broker* MQTT**: responsável pelo roteamento das mensagens MQTT compartilhadas pelo sistema;
- **Interface Web (*dashboard*)**: responsável pela interação com o usuário e apresentação das informações;
- **Banco de dados (DB)**: Responsável por organizar e armazenar as informações a longo prazo; e
- **API**: Responsável por intermediar a comunicação com o banco de dados.

Na Figura 4 seguir é apresentado um esquemático para ilustrar o fluxo de informações entre os componentes mencionados na arquitetura e os protocolos utilizados por eles — exceto no que se refere à comunicação entre o motor e o microcontrolador, onde o uso do termo "protocolo" não se aplica. Cada um destes componentes e protocolos de comunicação do sistema Web serão melhor desenvolvidos nas próximas seções.

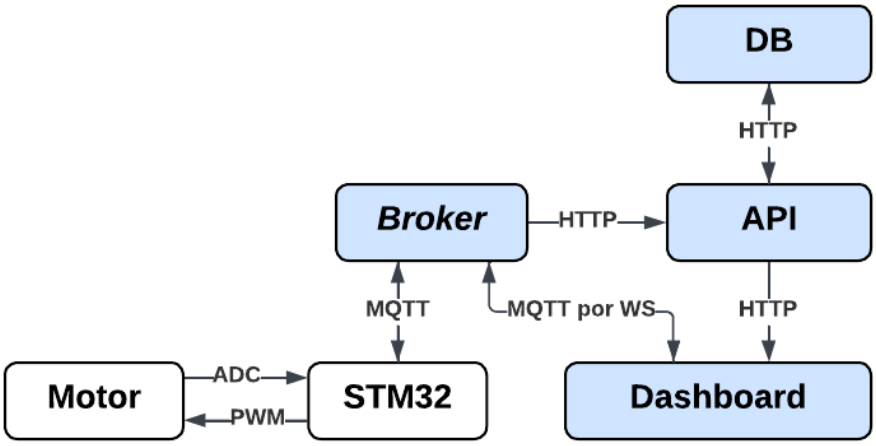


Figura 4 – Esquemático ilustrativo da arquitetura inicial do projeto.
Fonte: Autoria própria.

3.1.1 Broker MQTT

3.1.1.1 Node RED

O primeiro passo para a utilização do MQTT é a implementação de um *Broker* que será responsável por intermediar todas as mensagens compartilhadas entre os clientes. Para isso, optou-se por utilizar o Node-RED, uma aplicação Node.js rápida e leve com foco em sistemas IoT que oferece uma interface gráfica para fácil interação com o usuário e uma extensa biblioteca de módulos, que podem ser adicionados ao projeto para expandir suas funcionalidades.

Por se tratar de uma aplicação Node, o Node-RED pode ser instalado pelo gerenciador de pacotes oficial do Node (instalado automaticamente junto com o Node), o NPM (*Node Package Manager*), em qualquer dispositivo em que se possa instalar o Node, como computadores, Raspberrys Pi, servidores na nuvem, etc.

O desenvolvimento do *Broker* foi, em um primeiro momento, feito localmente em um computador pessoal. Para isso, o primeiro passo foi instalar o Node e o NPM neste computador. Com ambos instalados, o Node-RED pôde ser instalado globalmente executando o comando a seguir no terminal do computador.

```
sudo npm install -g --unsafe-perm node-red
```

Uma vez instalado, pôde ser executado com o comando `node-red` também no terminal, como visto na Figura 5, iniciando um servidor na porta 1880.

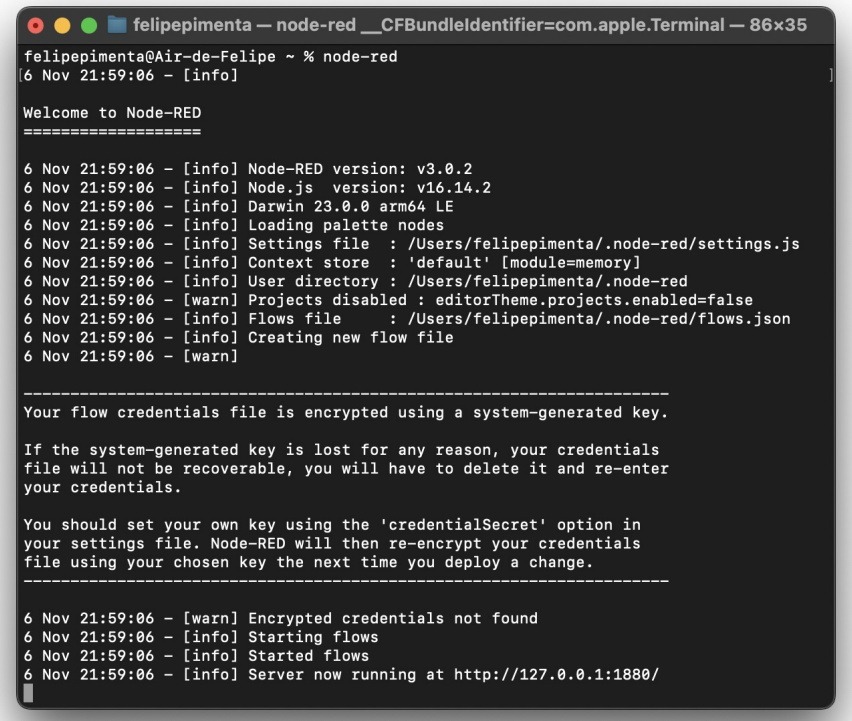


Figura 5 – Terminal após a execução do comando `node-red`.
Fonte: Autoria própria.

Acessando o endereço local indicado pelo terminal (Figura 5) teve-se acesso à interface gráfica do Node-RED, como visto na Figura 6. O funcionamento do Node-RED se baseia em nós interligados por fios para indicar o fluxo de informações. Por ser construído em cima do Node.js, o Node-RED é baseado em eventos, ou seja, todos os comportamentos dos nós são executados a partir de recebimento de algum evento, tanto em sua entrada

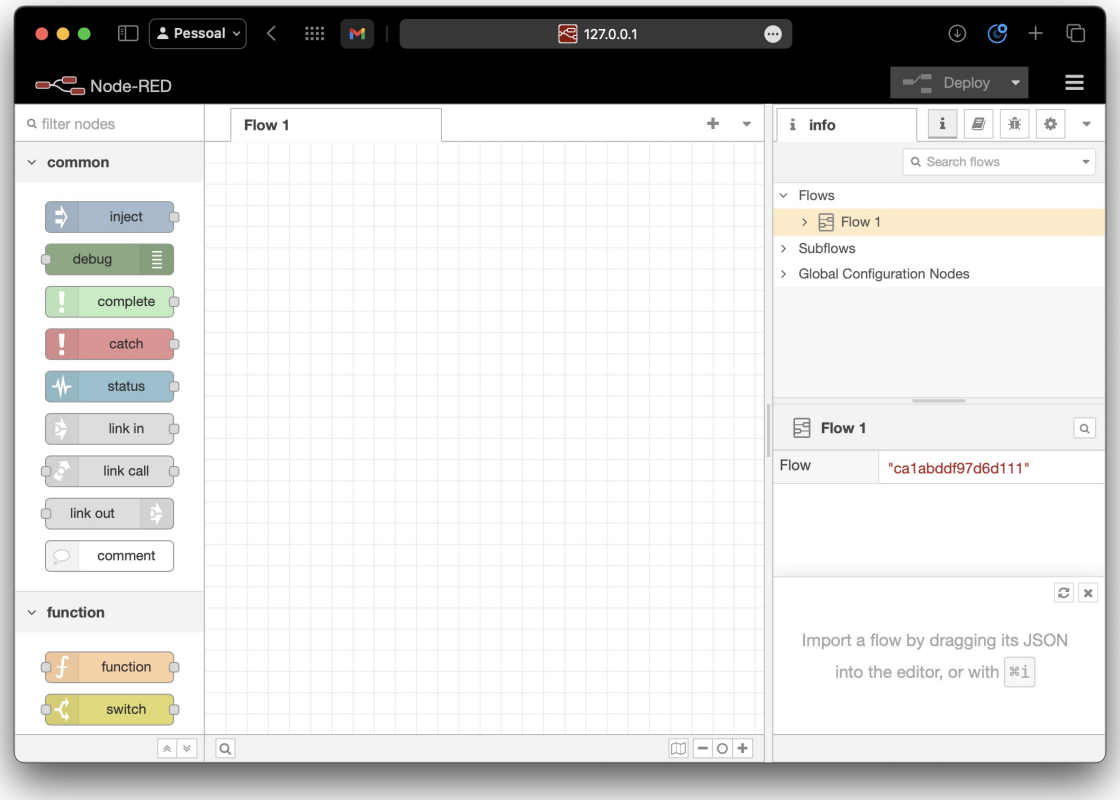


Figura 6 – Interface em branco do Node RED.
Fonte: Autoria própria.

como remotamente — como no caso do recebimento de uma publicação em algum tópico, por exemplo. Além disso, todas as informações são passadas no formato de um objeto JavaScript — ou JSON (*JavaScript Object Notation*) — facilitando sua compatibilidade com outras formas de comunicação pela internet.

O Node-RED permite a inscrição e publicação em tópicos MQTT através dos nós "MQTT in" e "MQTT out", respectivamente. Entretanto, para que o Node-RED se comporte como um *Broker*, é necessário a utilização de uma biblioteca aberta chamada `node-red-contrib-aedes`. Essa biblioteca adiciona ao projeto um nó que, ao ser inserido no fluxo e devidamente configurado, inicia uma instância de um *Broker* MQTT no projeto, permitindo que outros clientes MQTT se conectem a ele. Outros nós dentro do mesmo projeto podem se conectar a esse *Broker*, se inscrevendo e publicando em seus tópicos. Essa funcionalidade pode ser aproveitada para que se façam validações nas mensagens recebidas nos tópicos, possibilitando detectar valores fora dos limites permitidos e disparando avisos quando necessário, por exemplo.

Fazendo proveito das funcionalidade descritas acima e considerando as funcionalidades desejadas pelo sistema, foi desenvolvido um fluxo para o *Broker* do sistema, apelidado de "Pernilongo". O fluxo completo com todos os nós pode ser visto na Figura 7, suas

funcionalidades serão listadas e detalhadas nos próximos parágrafos.

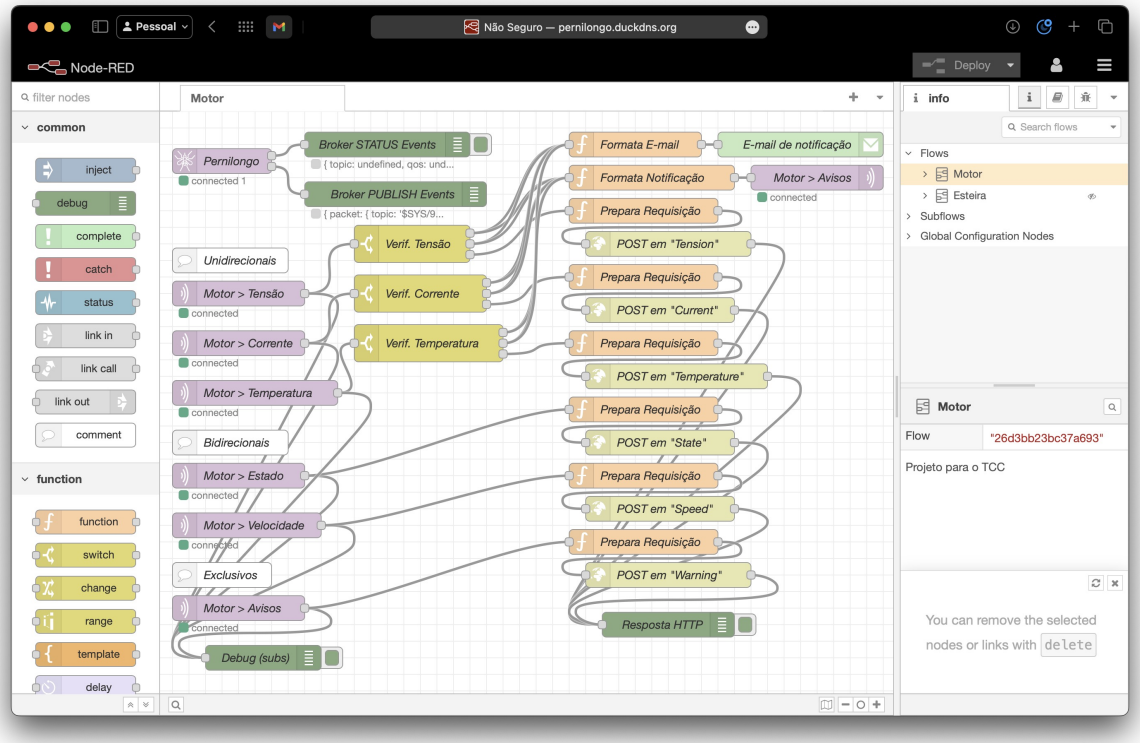


Figura 7 – Esquemático do Broker MQTT hospedado na AWS.
Fonte: Autoria própria.

Vale ressaltar que como o fluxo deste projeto possui tanto o nó que adiciona a funcionalidade de *Broker* quanto nós para se inscrever nos tópicos, ele pode ser considerado ao mesmo tempo *Broker* e cliente. O nó responsável por fazer com que essa aplicação Node se comporte como um *Broker* MQTT é o nó "Pernilongo", localizado no canto superior esquerdo da Figura 7. Os nós verde escuro, ligados em suas saídas, são nós de *debug*, utilizados para visualizar os *logs* do *Broker* no console.

Os demais nós roxos à esquerda do fluxo representam inscrições em tópicos MQTT. Estes nós foram nomeados de forma que refletissem o tópico em que estão inscritos, ou seja, o nó "Motor > Tensão" representa uma inscrição no tópico "`/motor/tension`", e assim também com os demais nós. Assim como o nó "Pernilongo", estes nós de inscrição também estão ligados a um nó de *debug*, para auxiliar durante o desenvolvimento e teste do fluxo.

Como é possível observar na Figura 7, os tópicos estão divididos em **unidirecionais**, **bidirecionais** e **exclusivos**. Os tópicos **unidirecionais** representam canais que apenas recebem informações do motor, sem que seja possível enviar qualquer espécie de controle ou instrução para este. Já os tópicos **bidirecionais** representam canais em que, além de se receber informações do motor, é também possível enviar instruções a este. Por último,

o tópico exclusivo representa um canal para transmissão de informações não vindas do motor, como o envio de avisos após a validação.

A validação, por sua vez, é feita nos nós verde-amarelado conectados às saídas do nós de inscrição nos tópicos **unidirecionais**, verificando se os valores recebidos estão dentro de limites pré estabelecidos. Estes nós funcionam como condicionais, fornecendo saídas com "caminhos" distintos para o fluxo de acordo com o resultado da validação dos valores de cada um dos nós.

Nos casos em que o valor recebido está acima ou abaixo dos limites pré estabelecidos o fluxo segue para os nós que farão o envio de um e-mail — utilizando um nó especial adicionado por uma biblioteca — e uma publicação no tópico de avisos, ambos informando qual valor que extrapolou o limite. Quanto ao envio de e-mail, foi criada uma nova conta Gmail (`pernilongo.broker@gmail.com`), da qual se concedeu permissão ao *Broker* para que fosse feito o envio de e-mails a partir dela.

Em suma, valores recebidos fora dos limites pré estabelecidos fazem com que o *Broker* dispare dois avisos: um na forma de uma publicação no tópico de avisos, para ser exibido em tempo real no *dashboard*; e outro na forma de um e-mail, para que seja visto mesmo quando não se estiver observando o *dashboard*.

Se o valor recebido estiver dentro dos limites pré estabelecidos o fluxo segue para o mesmo caminho que os valores recebidos pelos tópicos **bidirecionais** — segue para ser enviado à API via requisição HTTP, para ser salvo no banco de dados. Essa requisição é feita com o método POST e contém em seu *body* as informações a respeito daquela publicação.

Ao longo dos fluxos foram adicionados diversos nós laranjas — nós em que se pode escrever a função JavaScript que se desejar — indicados como nós para Preparar ou Formatar. Isso foi necessário para garantir que a mensagem transmitida estivesse sempre no formato correto para ser publicada ou adicionada ao corpo da requisição POST, uma vez que as mensagens dentro de um fluxo do Node-RED seguem um formato específico: um objeto JavaScript chamado `msg` com a informação transmitida na chave `payload`.

Esse formato específico das mensagens dentro do Node-RED está diretamente ligado ao padrão das mensagens publicadas em tópicos MQTT. As mensagens publicadas são sempre JSONs transformados em *string*, contendo as chaves `payload`, `topic`, `qos` e `_id`. Essas chaves representam: a informação que será publicada; o tópico em que ela será publicada; a qualidade de serviço (QoS) com que será publicada; e um ID único.

Considerando o objetivo de se armazenar um histórico das publicações ao longo do tempo, fez sentido que se enviasse também nas publicações uma *timestamp* indicando o horário exato em que a mensagem foi publicada. Dessa forma, definiu-se que a chave `payload` mencionada anteriormente seria também um objeto transformado em *string*,

contendo dentro de si outras duas chaves: `message`, com a mensagem a ser publicada, e `time`, com a *timestamp*, indicando o horário extado da publicação.

3.1.1.2 AWS

Tendo em mente o objetivo de se desenvolver um sistema que pudesse ser acessado remotamente de qualquer lugar foi fundamental que o *Broker* estivesse hospedado em um servidor com um endereço público. Para isso, o serviço de hospedagem escolhido foi o da AWS (*Amazon Web Services*), devido à grande quantidade de informações disponíveis a respeito da utilização de seus serviços e pela disponibilidade de um nível gratuito generoso capaz de atendender confortavelmente às demandas deste projeto. Foi, então, criada uma instância EC2 (*Elastic Compute Cloud*) com Ubuntu, como visto na Figura 8.

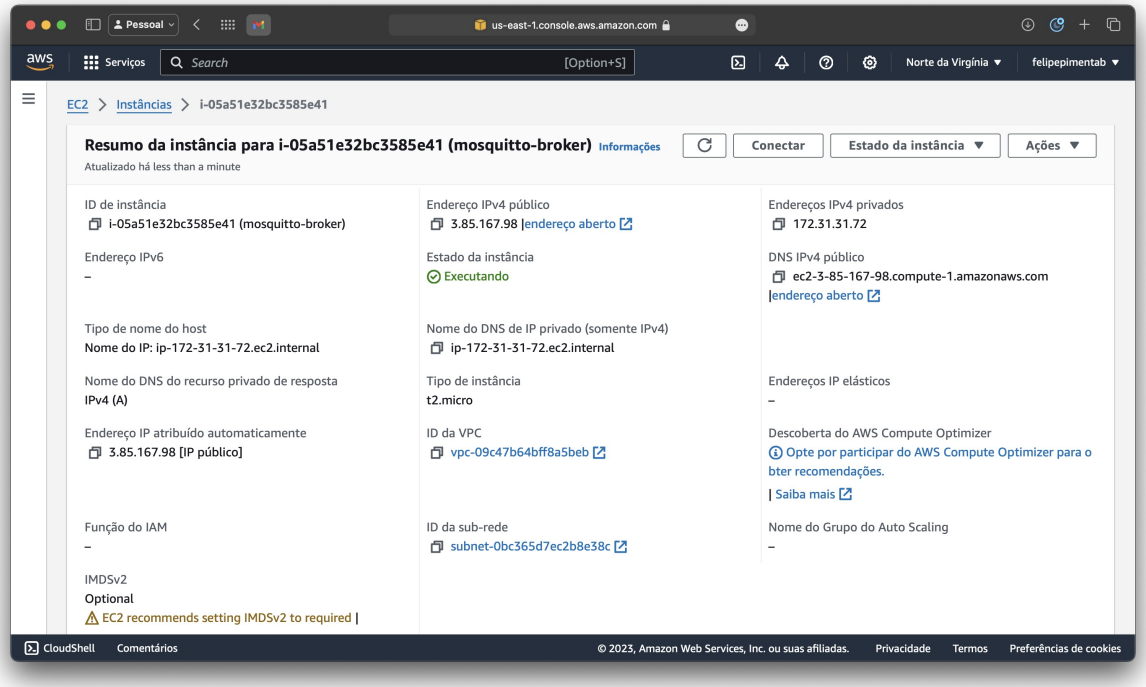


Figura 8 – Resumo da instância EC2 hospedada gratuitamente pela AWS.
Fonte: Autoria própria.

Uma vez criada a instância EC2, esta pôde ser acessada via SSH, como instruído na Figura 9. Para isso, primeiro foi necessária a criação de uma chave privada para ser utilizada para autenticação. Esta chave, chamada de `pipopb.pem`, foi então salva no diretório raiz do computador usado para se conectar à instância, para que pudesse ser facilmente referenciada no terminal. O acesso à instância via SSH, portanto, se deu através da execução do comando a seguir no terminal, como mostrado na Figura 10.

```
ssh -i "pipopb.pem" ubuntu@ec2-3-85-167-98.compute-1.amazonaws.com
```

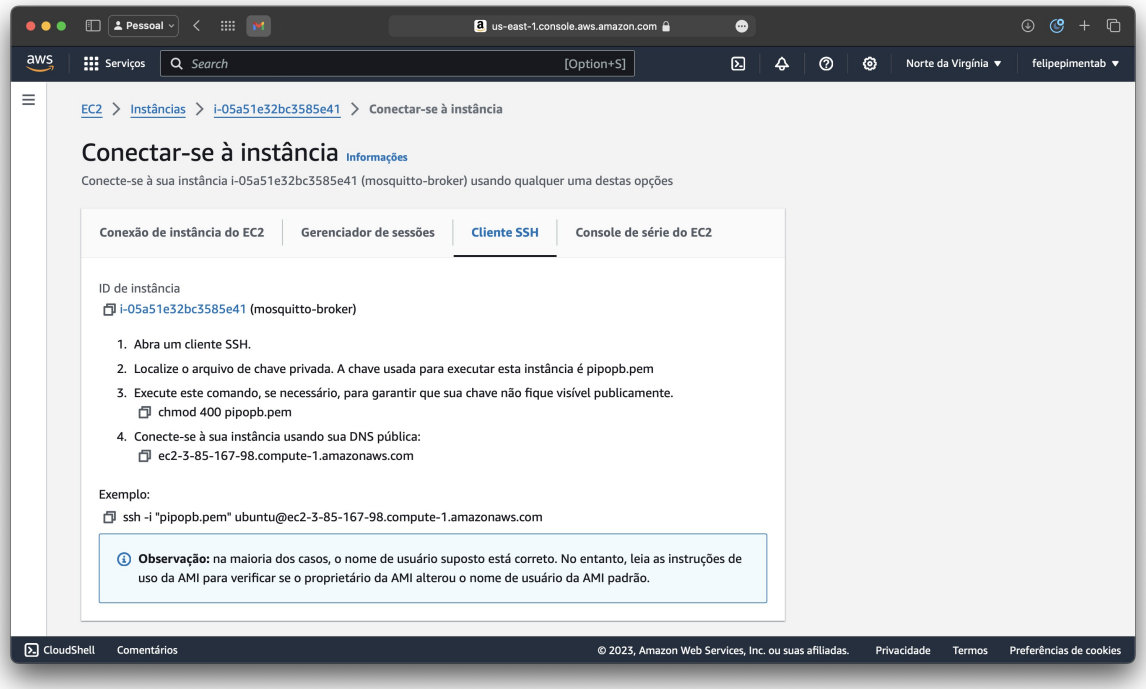


Figura 9 – Instruções para se conectar à instância via SSH.
Fonte: Autoria própria.

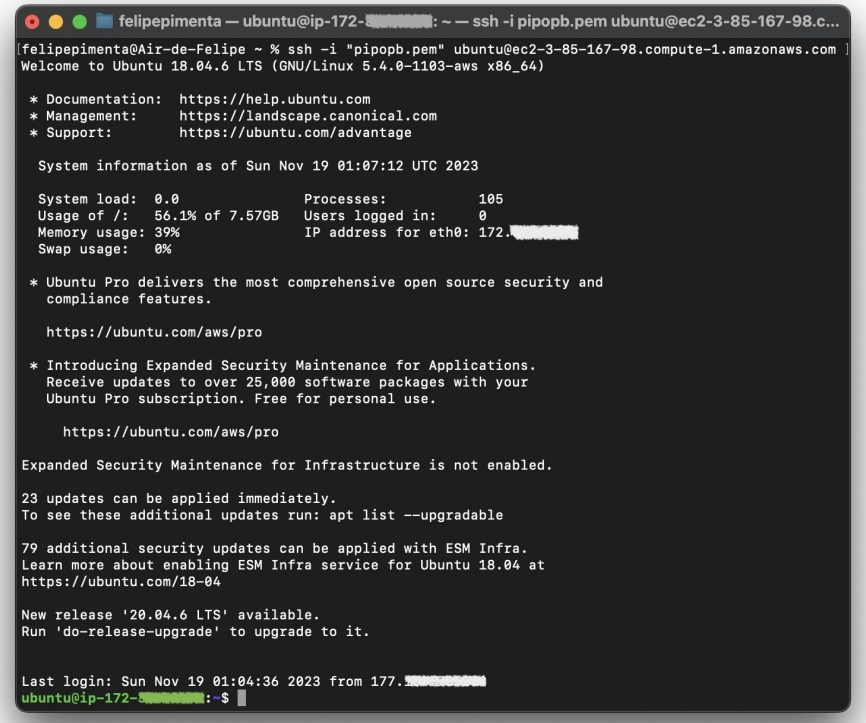


Figura 10 – Terminal conectado à instância via SSH.
Fonte: Autoria própria.

Uma vez executado este comando com êxito, teve-se acesso ao terminal da instância EC2, por onde puderam ser instalados o Node.js e o Node-RED com os comandos:

```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
sudo apt-get install -y nodejs build-essential
sudo npm install -g --unsafe-perm node-red
```

Uma vez instalados, foi possível executar o Node-RED na instância com o comando `node-red`, assim como mostrado na Figura 5. Entretanto, para que fosse possível acessar a interface gráfica do Node-RED para configuração dos nós foi necessário alterar as regras de segurança da instância EC2, para que esta permitisse o acesso às portas que seriam utilizadas. Isso pôde ser feito através do próprio *dashboard* da AWS, como pode ser visto na Figura 11. Nessa figura podemos observar as portas que foram liberadas, as origens permitidas em cada, os protocolos permitidos em cada e um nome descritivo de qual funcionalidade será atribuída a cada porta — a origem `0.0.0.0/0` indica que todas as origem são permitidas.

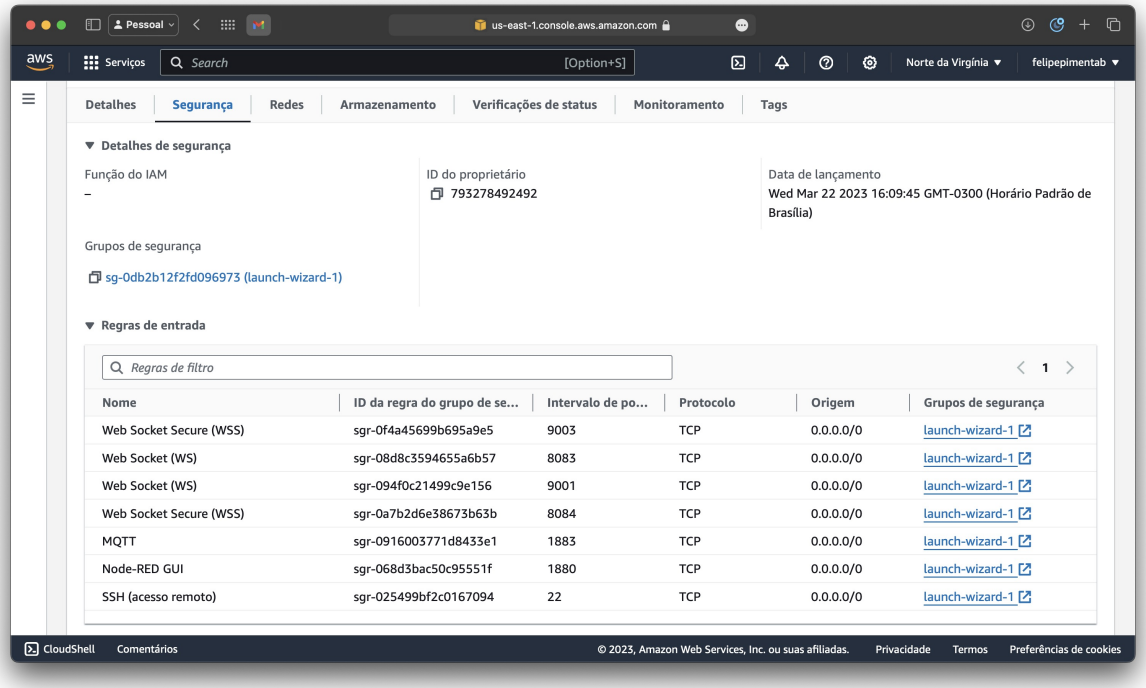


Figura 11 – Configurações de segurança da instância EC2.
Fonte: Autoria própria.

Dessa forma, ao navegar para o endereço do servidor na porta `1880`, é possível acessar a interface gráfica do Node-RED, assim como mostrado na Figura 6. Apesar da interface gráfica do Node-RED ter ficado aberta para qualquer endereço, o Node-RED permite a restrição do acesso à ela através da configuração de usuários com diferentes níveis de permissão, como “edição” ou “apenas leitura”, por exemplo. Para isso, foi necessário

se conectar novamente à instância via SSH, como mostrado na Figura 10, por onde se pôde acessar o arquivo de configurações, `settings.js`, localizado dentro da pasta oculta `/.node-red`. Isso foi feito utilizando os comandos `ls -a`, para listar todos os diretórios, incluindo os ocultos; `cd .node-red`, para navegar até o diretório especificado; e `nano settings.js`, para abrir o arquivo com o editor de texto do terminal, como mostrado na Figura 12.

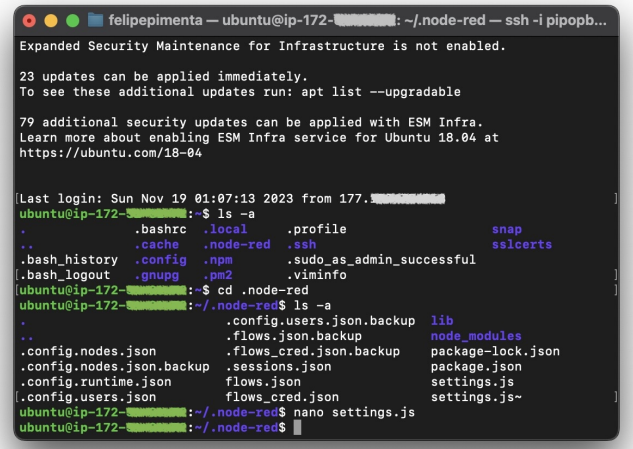


Figura 12 – Acesso ao arquivo de configuração `settings.js` na instância via SSH.
Fonte: Autoria própria.

Foram então criados 2 usuários: um administrador, com todas as permissões, para edição dos fluxos; e um usuário comum, com permissão apenas de leitura, para que outras pessoas pudessem acessar e visualizar o fluxo sem que houvesse o risco de ser feita alguma alteração indesejada. Esses usuários são apresentados na Figura 13, onde é possível notar também que as senhas são armazenadas após o *hashing*, por segurança. O *hashing* das senhas, por sua vez, foi gerado com o comando a seguir.

```
node-red admin hash-pw
```

A execução do comando acima em um terminal separado abriu um *prompt* pra inserção da senha. Uma vez inserida, foi gerado o *hash* da senha, que pôde ser copiado e inserido no campo adequado do arquivo de configuração `settings.js` (Figura 13).

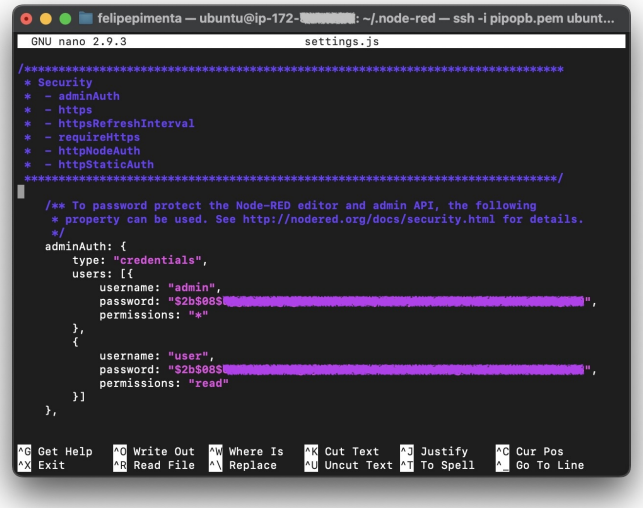


Figura 13 – Configuração de acesso à interface gráfica do Node-RED na instância EC2 via SSH.
Fonte: Autoria própria.

Assim, a instância EC2 hospedada pela AWS pôde ser configurada com o fluxo de nós apresentado anteriormente na Figura 7, permitindo que todas as funcionalidades descritas fossem executadas, incluindo permitir que outros clientes MQTT se conectem ao *Broker* na porta `1883`.

Por fim, dois pequenos ajustes foram feitos para facilitar a execução do Node-RED na instância EC2 e o acesso ao *Broker*. O primeiro deles foi utilizar um gestor de processos Node, o PM2 (Process Manager 2), para executar o Node-RED automaticamente na instância sempre que esta fosse iniciada. Isso foi feito executando, no terminal da instância, os comandos a seguir:

```
sudo npm install -g --unsafe-perm pm2
pm2 start 'which node-red' -- -v
pm2 save
pm2 startup
```

O segundo ajuste foi adicionar um domínio que redirecione o tráfego para o endereço da instância, através de um serviço de DNS (*Domain Name System*). Para isso, foi utilizado um serviço gratuito chamado Duck DNS, como visto na Figura 14. Isso permite que a interface gráfica do Node-RED da instância seja acessada pelo endereço `http://pernilongo.duckdns.org:1880`, por exemplo, facilitando o acesso e evitando a necessidade de se memorizar o endereço da instância.

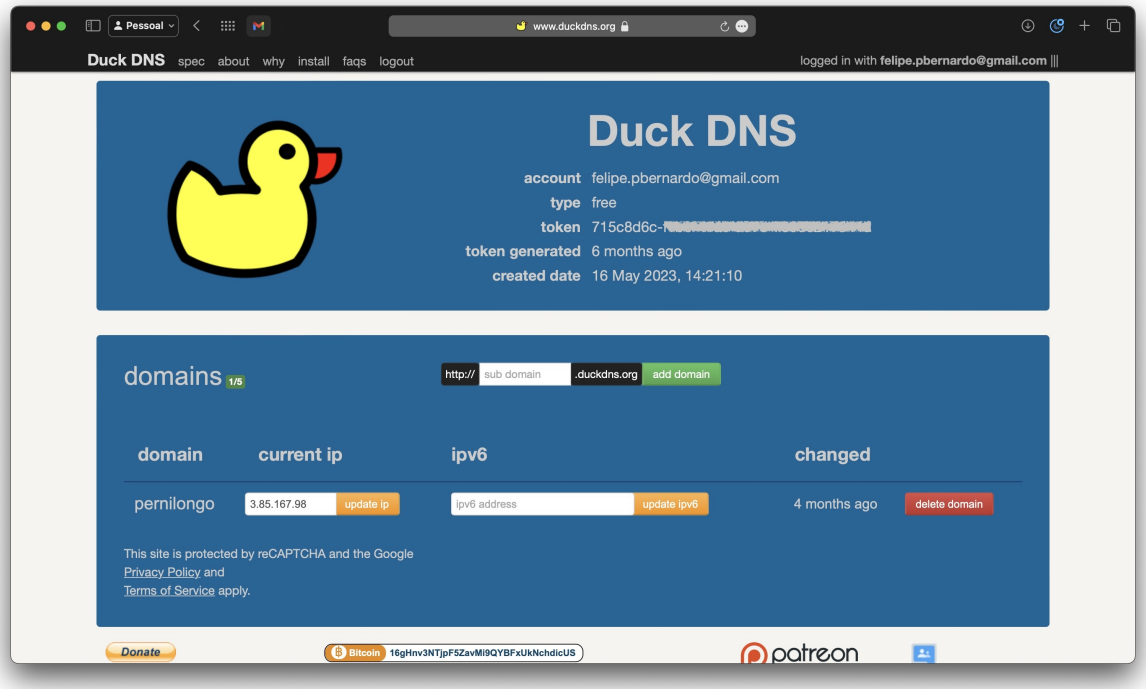


Figura 14 – Configuração de um DNS gratuito para a instância EC2.
Fonte: Autoria própria.

3.1.2 *Dashboard*

3.1.2.1 Vue.js

Como foi dito na revisão bibliográfica (2.3.1), o uso de *frameworks* Front-End permite a criação de interfaces de usuário mais elaboradas com muito menos esforço. Sendo assim, o *framework* escolhido para a implementação do *dashboard* foi o Vue.js — mais especificamente, a versão mais recente dele: **Vue 3**. Este *framework* possui uma CLI (*Command Line Interface*) que permite a criação rápida de um novo projeto Vue apenas executando o comando `npm create vue@latest` e selecionando dentre as opções dadas no terminal.

A aplicação criada por este *framework* é chamada de SPA (*Single Page Application* - Aplicação de uma única página), onde é gerado um único arquivo HTML, isto é, uma única página "verdadeira", na qual todo o seu conteúdo (incluindo trechos de HTML e CSS) será carregado e modificado dinamicamente usando JavaScript, de tal forma que a troca de conteúdo dentro dessa página aparenta, para o usuário, ser uma troca de página. Isso é vantajoso ao se considerar que, em uma aplicação tradicional, as páginas HTML não conseguem acessar as informações e estados armazenadas nas demais páginas, ou seja, o uso de SPAs possibilita o compartilhamento de estados entre todas as "páginas" da aplicação.

Além disso, existem pacotes NPM com *state managers* gerenciadores de estados)

prontos, feitos pela comunidade e suportados oficialmente pelo *framework*, facilitando ainda mais o gerenciamento de estados em toda a aplicação, como VueX e Pinia, por exemplo.

O *dashboard* foi, então, criado utilizando a CLI para configuração rápida. Além desse *framework*, foram utilizados alguns pacotes e bibliotecas NPM para facilitar a implementação de certas funcionalidades, dentre eles:

- `mqtt` para facilitar e simplificar a criação de uma conexão MQTT no lado do cliente via WebSocket, permitindo o recebimento das mensagens publicadas nos tópicos para que fossem exibidas na página inicial do *dashboard* em tempo real;
- `axios` para facilitar e simplificar o processo de fazer requisições HTTP, possibilitando fazer requisições a API, a fim de se obter o histórico de publicações para ser exibido em gráficos e em lista;
- `chart.js` para possibilitar a plotagem de gráficos com os dados recebidos da API.
- `pinia` para auxiliar no gerenciamento de estados na aplicação.

Além disso, para se ter um melhor controle do código, foi usado o **Git** como ferramenta de versionamento de código. O repositório, chamado de `pernilongo-dashboard`, foi publicado no GitHub (Figura 15) e pode ser acessado no endereço https://github.com/felipepimentab/pernilongo-dashboard/tree/main.

Além dos aspectos técnicos do projeto, é fundamental que a interface de usuário possua um *design* agradável para que a informação possa ser interpretada de forma fácil e rápida pelo usuário e para que este consiga navegar pela aplicação sem dificuldades, tendo acesso a todas as suas funcionalidades. Propriedades como tamanho do texto, fonte, contraste com o fundo, cores, uso de ícones e outros elementos visuais, além de facilitar a visualização da informação, servem também para definir a hierarquia entre os elementos visuais e dão dicas ao usuário sobre como interagir com a aplicação. Por isso, durante todo o desenvolvimento das interfaces de usuário teve-se como base as Diretrizes de Interface Humana disponibilizada pela Apple (Apple Inc., 2023).

3.1.2.2 Vercel

Vercel é uma plataforma que simplifica e acelera a implementação de aplicações Web, oferecendo suporte a vários *frameworks*, *deploys* automáticos, *serverless functions*, certificados SSL automáticos, entre outros serviços. O foco da Vercel na simplicidade a torna uma escolha popular para desenvolvedores, e sua faixa de serviços gratuitos atende às demandas deste projeto. Por estes motivos, esta foi a plataforma escolhida para hospedar a aplicação da *dashboard* (Figura 16).

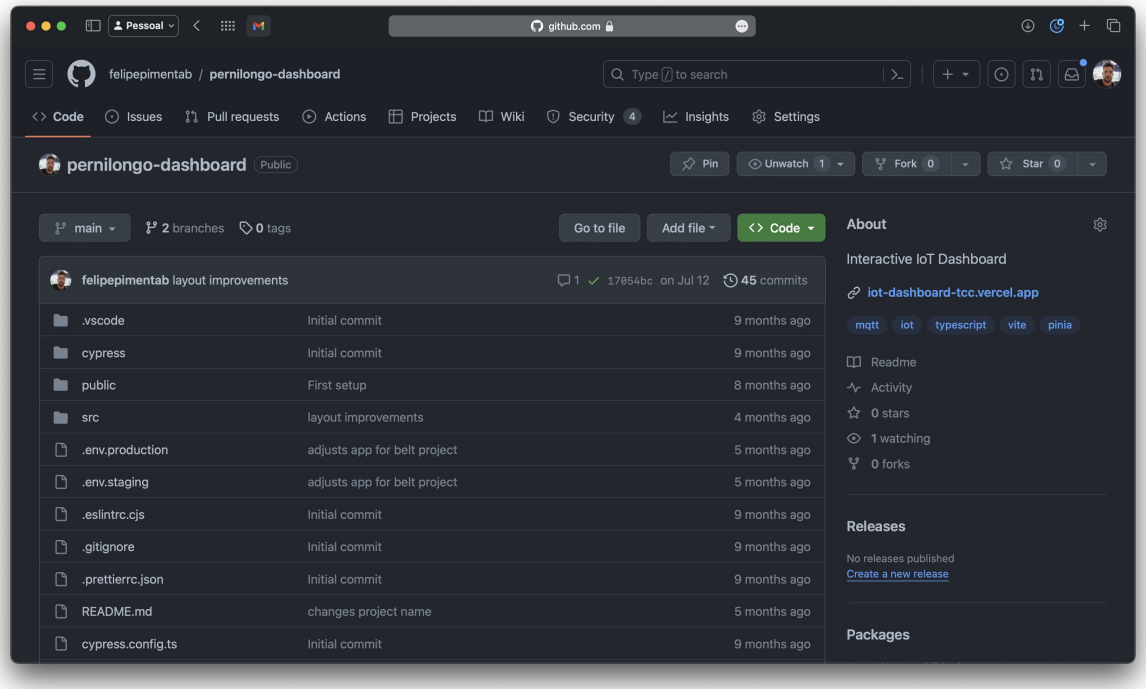


Figura 15 – Repositório do *dashboard* publicado no GitHub.
Fonte: Autoria própria.

Esse processo pôde ser feito de maneira muito simples, diretamente do site da Vercel. O primeiro passo foi a criação de uma conta gratuita na plataforma. Em seguida, a implementação da aplicação do *dashboard* foi feita conectando a conta Vercel à conta do GitHub, de forma que a plataforma tivesse acesso ao repositório mostrado na Figura 15. A necessidade de configuração do projeto foi mínima, uma vez que a plataforma já possui um modelo pronto para o *framework* utilizado. Além disso, a aplicação hospedada recebe um certificado SSL automaticamente, de forma que esta seja acessada utilizando o protocolo HTTPS.

Apesar da Vercel disponibilizar um domínio gratuito para cada projeto, é possível optar também por utilizar um domínio próprio. Uma vez que o domínio `pernilongo.cc` foi comprado pelo período de 1 ano, optou-se por utilizá-lo para esta aplicação. Mais especificamente, a *dashboard* teve então o domínio https://dashboard.pernilongo.cc associado a ela. Assim como a implementação, a atribuição do domínio também foi feita com uma necessidade mínima de configuração e com muita facilidade, apenas seguindo as orientações da própria plataforma.

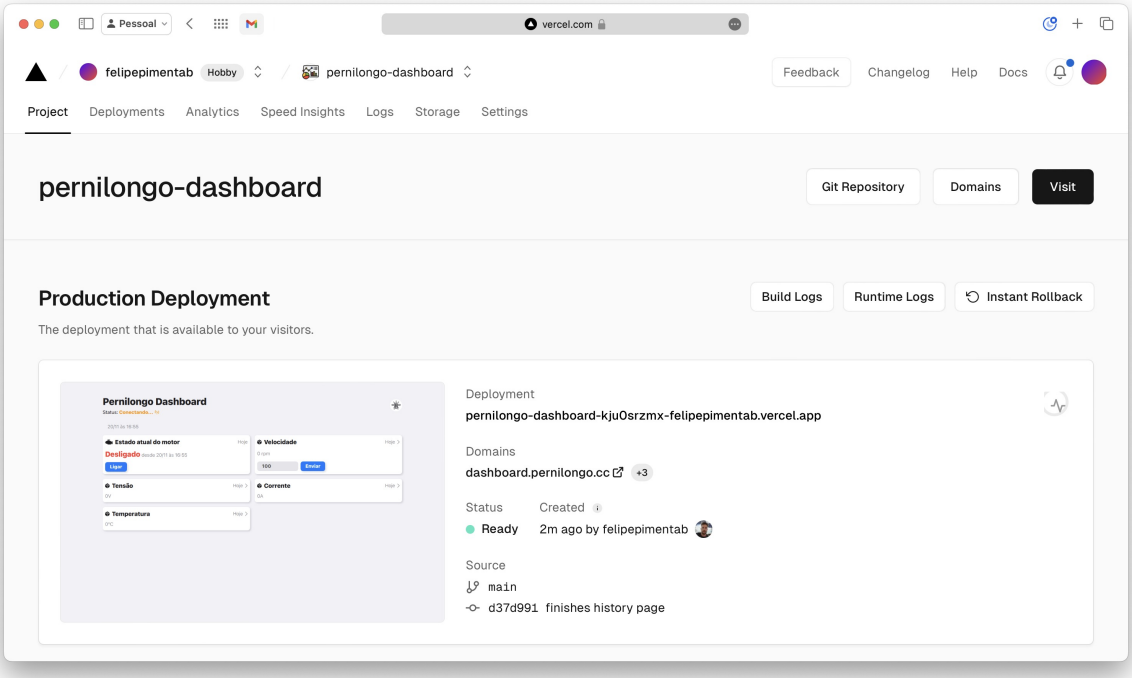


Figura 16 – Projeto do *dashboard* hospedado na Vercel.
Fonte: Autoria própria

3.1.3 Banco de dados

Devido à natureza da informação que se pretendia armazenar — já em JSON e sem a necessidade de ser relacional — optou-se por usar o MongoDB como banco de dados. MongoDB é um banco de documentos não relacional, que possuí um nível gratuito generoso para hospedagem na Atlas e uma interface gráfica amigável para fácil visualização dos documentos armazenados, como mostrado pela Figura 17. Os documentos em um banco MongoDB ficam organizados em *Collections* que, por sua vez, ficam dentro de *Databases*. Na Figura 17 é possível observar que se optou por utilizar apenas uma única *Database*, chamada “Publications”, com uma *Collection* para cada tópico.

Desta vez, ao invés de criar o projeto localmente e depois implementá-lo em alguma plataforma, optou-se por criá-lo diretamente na plataforma da Atlas — a plataforma oficial para hospedagem de bancos de dados MongoDB. Isso foi feito devido ao fato de que, neste caso, a complexidade de criar o banco diretamente nessa plataforma é menor do que de criar o banco localmente, além de que o banco ainda precisaria, necessariamente, ser implementado em alguma plataforma para que pudesse ser acessado pela API posteriormente. O processo de criação do banco foi simples, bastou criar uma conta e seguir as instruções da plataforma.

Com o objetivo de adicionar uma camada de “proteção” e validação ao banco de dados, decidiu-se limitar apenas à API o acesso às informações armazenadas no banco.

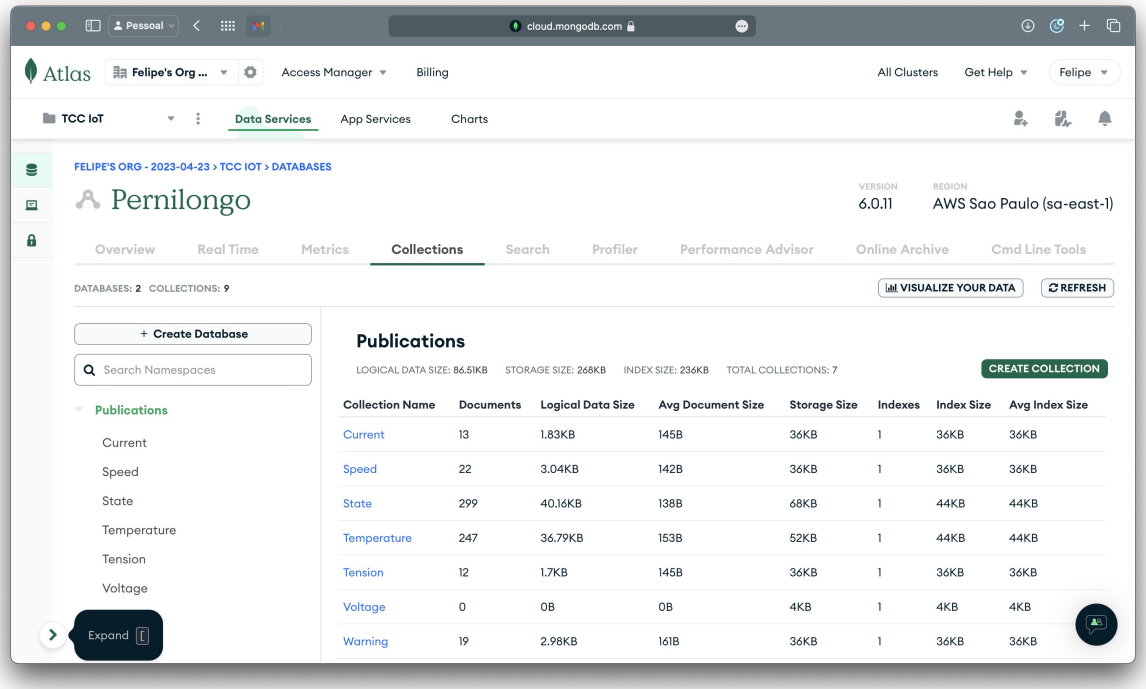


Figura 17 – *Dashboard* da Atlas para visualização dos documentos do banco de dados MongoDB.

Fonte: Autoria própria.

3.1.4 API

3.1.4.1 ExpressJS

A fim de se garantir um melhor controle dos dados lidos e escritos no banco de dados, optou-se por desenvolver uma interface isolada para ele — uma API. Assim, nenhum cliente se conecta diretamente ao banco de dados. Todas as requisições de leitura ou publicação devem ser feitas à API, que ficará responsável por garantir que todos os dados lidos e escritos no banco sejam validados e estejam de acordo com o padrão definido.

Para o desenvolvimento dessa API, utilizou-se o *framework* **Express.js**, construído em cima de Node.js, simplificando, assim, a troca de dados entre todos os componentes do sistema, uma vez que todos compartilham JSON. A conexão com o banco de dados MongoDB, por sua vez, foi feita utilizando um adaptador comum, chamado **Mongoose**, que simplifica esse processo. Há a necessidade de se fornecer uma senha junto à URL usada para se conectar ao banco. Por segurança, essa senha é importada como uma variável de ambiente no projeto, de forma que ela não fique disponível no repositório, uma vez que este é público.

Quanto às funcionalidades da API, estas ficaram divididas basicamente em duas categorias: ler e escrever informações no banco de dados. Em ambos os casos, o fluxo se inicia com o recebimento de uma requisição HTTP, vinda de um cliente, em uma de suas rotas. A requisição é processada e, de acordo com a rota e com o método da requisição, é

feito o que foi pedido.

Para requisições GET, a API acessa o banco, busca as informações solicitadas — como publicações em um tópico específico, por exemplo — e retorna uma resposta com código 200 (OK) e as informações solicitadas no corpo da resposta. Já para requisições POST, a API valida se o objeto vindo no corpo da requisição está de acordo com os padrões definidos para o tópico em que se deseja salvar a publicação. Se sim, é criado um documento a partir da informação recebida, que é então salvo no banco. Uma resposta com código 201 (Criado) e um cópia do documento salvo são retornados ao cliente como forma de confirmação que a publicação foi salva. Se não, é retornada uma resposta com código 400 (*bad request*), indicando que há um erro na requisição. Além disso, se alguma requisição for recebida em alguma rota não mapeada a API retorna uma resposta com código 404 (*not found*), indicando que a rota não foi encontrada.

Se uma requisição GET for feita à rota base ("/"), especificamente, a respostá será um documento HTML, ou seja, uma página Web, que contém uma documentação detalhada de como interagir com a API. Essa documentação pode ser visualizada na página da rota https://api.pernilongo.cc ou na transcrição da documentação no Apêndice B.

Este projeto também foi versionado com **Git** e publicado no GitHub como visto na Figura 18. O repositório é publico e pode ser acessado através do endereço https://github.com/felipepimentab/pernilongo-api.

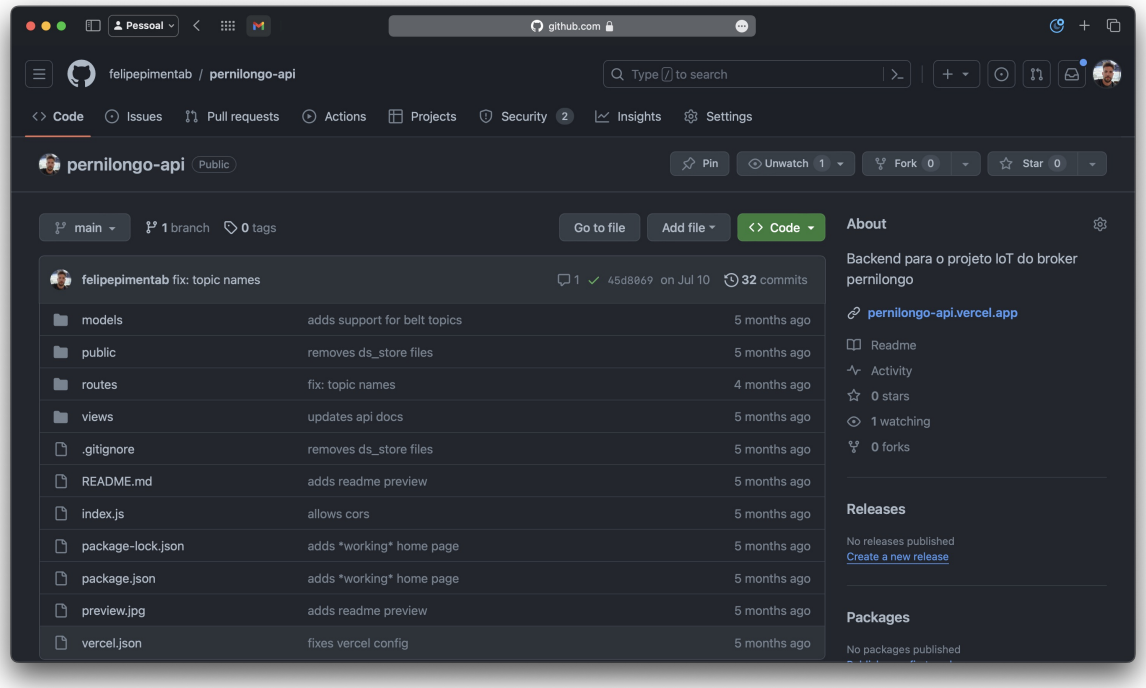


Figura 18 – Repositório da API publicado no GitHub.
Fonte: Autoria própria.

3.1.4.2 Serverless function

A API também foi hospedada pela Vercel, assim como o *dashboard*, como visto na Figura 19. Diferentemente do *dashboard*, porém, a API foi hospedada como uma *serverless function*, ou seja, uma função que é executada sempre que feita uma requisição a uma de suas rotas. Assim como no projeto do *dashboard*, no projeto da API também se optou por utilizar um domínio próprio, sendo ele https://api.pernilongo.cc.

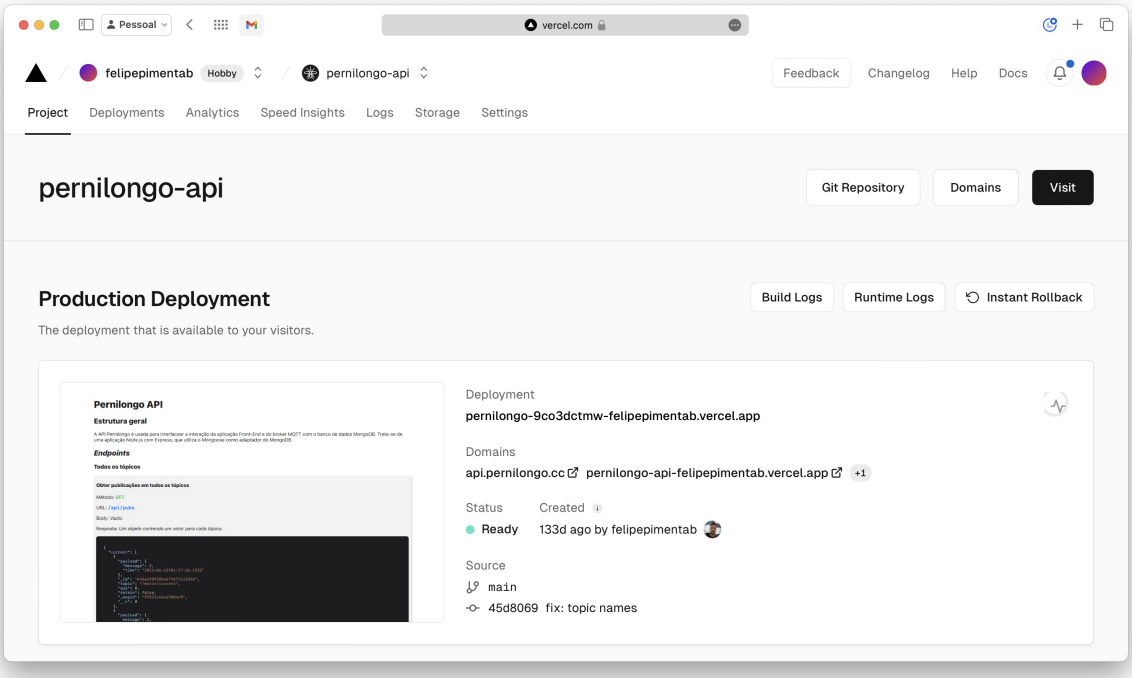


Figura 19 – Projeto da API hospedado na Vercel como uma *serverless function*.
Fonte: Autoria própria.

4 RESULTADOS

Neste capítulo será feito um apanhado dos resultados obtidos em cada um dos componentes do sistema Web desenvolvido. Ao final de cada apanhado será feita uma discussão a respeito das principais dificuldades e limitações encontradas, e como elas afetaram o resultado obtido. Serão discutidas também as possíveis soluções para estes problemas, melhorias e novas funcionalidades que poderiam ser implementadas.

4.1 *Broker*

Como foi visto na sessão anterior, o *Broker* desenvolvido não somente atendeu à demanda proposta, isto é, fazer o roteamento das mensagens publicadas pelos clientes MQTT, como também foi capaz de desempenhar outras funções, como validações nos valores recebidos e o envio de notificações por e-mail. Sendo assim, considera-se que a escolha do Node-RED se mostrou muito vantajosa, uma vez que ele possibilitou o desenvolvimento do fluxo interno do *Broker* e a expansão de suas funcionalidades através de bibliotecas externas com muita facilidade.

A Figura 7 mostra todo o fluxo de nós implementado no *Broker* MQTT. Alternativamente, foi criado também um usuário com permissões de leitura para que se pudesse visualizar o fluxo completo com os nós e suas configurações internas diretamente da interface gráfica do Node-RED. Para isso, basta acessar o endereço http://pernilongo.duckdns.org:1880, preenchendo os campos de usuário com "`user`" e de senha com "`user12345`" (Figura 20).

Os clientes podem se conectar ao *Broker* diretamente por MQTT pelo endereço http://pernilongo.duckdns.org:1883 ou por MQTT via WebSocket, no caso de interfaces Web, pelo endereço http://pernilongo.duckdns.org:8083. Em ambos os casos é necessário fornecer um par de usuário e senha que foi definido nas configurações do nó "Pernilongo", também mostrado na Figura 7.

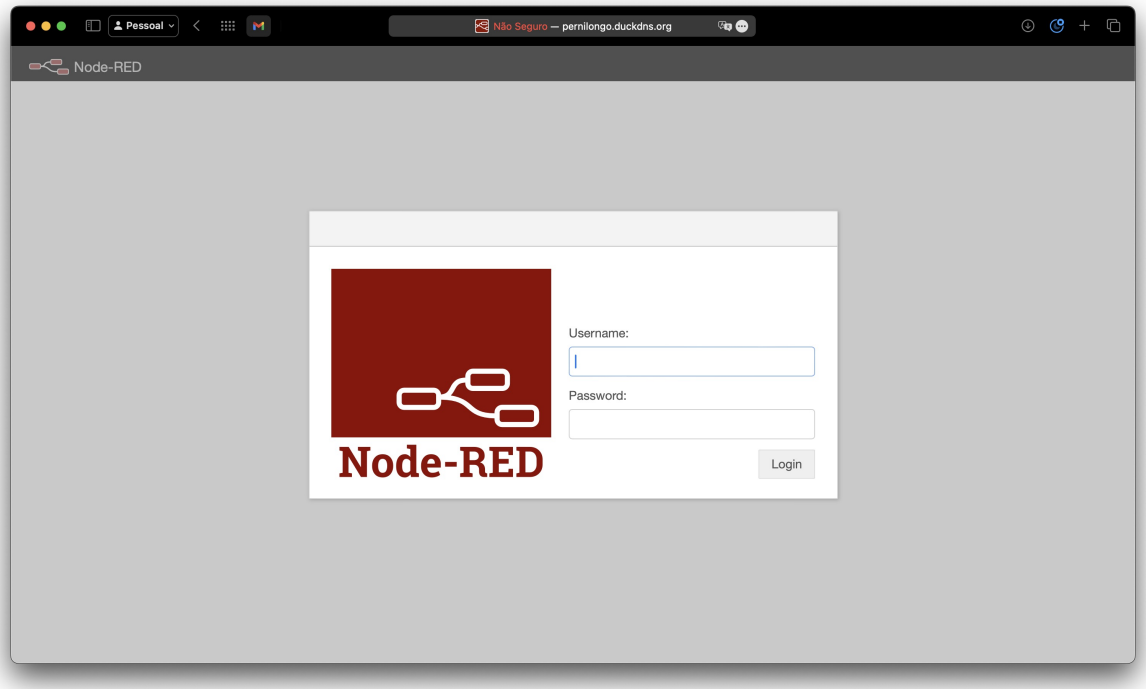


Figura 20 – Tela de login para acessar a interface gráfica de usuário do Node-RED.
Fonte: Autoria própria.

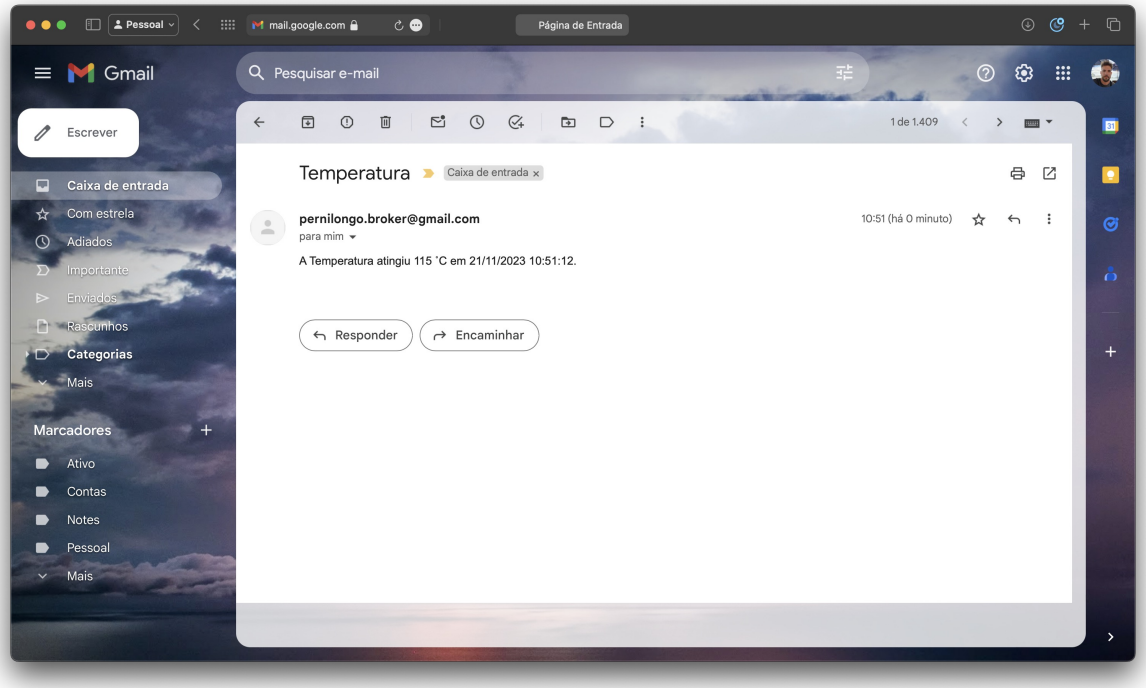


Figura 21 – Notificação via e-mail enviada pelo *Broker* MQTT.
Fonte: Autoria própria.

Diferentemente dos demais componentes desse sistema, não foi possível alcançar o objetivo de adicionar um certificado SSL ao *Broker* para que esse pudesse ser acessado

por protocolos seguros, como HTTPS (HTTP *Secure*) e WSS (WebSocket Secure). Além dos riscos inerentes ao uso de protocolos não seguros por si só, o uso destes protocolos desencadeou outros problemas, que serão mencionados na sessão seguinte.

4.2 *Dashboard*

A estrutura visual do *dashboard* foi pensada para possibilitar a visualização das informações com clareza e eficiência, entregando um *design* mínimo mas ao mesmo tempo moderno e completo com todas as informações disponíveis. Além disso, a aplicação foi desenvolvida com *layout* responsivo, se adaptando a telas menores, como as de dispositivos móveis. No Apêndice A é apresentado o *layout* do *dashboard* simulando como seria sua apresentação em uma tela de dispositivo móvel.

O *layout* final do *dashboard* tem como base duas páginas. A primeira, chamada de *home*, apresenta informações em tempo real recebidas do *Broker* por MQTT via WebSockets. Nesta página, vista na Figura 22, é possível observar:

- O *status* da conexão do *dashboard* com o *Broker*;
- O estado atual do motor;
- A última medida velocidade publicada pelo motor, em rpm;
- A última medida de tensão publicada pelo motor, em Volts (V);
- A última medida de corrente publicada pelo motor, em Amperes (A);
- A última medida de temperatura publicada pelo motor, em graus célsius (ºC);
- O último aviso emitido pelo *Broker*.

A página está organizada de forma que as informações de cada tópico estejam contidas em um *card*, onde é possível observar não somente o último valor recebido naquele tópico como também a data em que foi recebido. Cada *card* (com exeção do *card* para o estado do motor) é também um *link* para navegar para a segunda página, mostrada na Figura 23.

A segunda é a página de informações específicas de um determinado tópico, retornadas pela API via requisição HTTP. Nela é possível visualizar um gráfico com as 10 últimas publicações feitas naquele tópico, bem como uma lista com todas as publicações recebidas por ele.

Com o intuito de obter dados mensuráveis a respeito do desempenho da aplicação, foi feita uma auditoria da página utilizando o Google *Lighthouse*, uma ferramenta disponível no navegador Google Chrome, considerada a principal referência para auditoria de interfaces

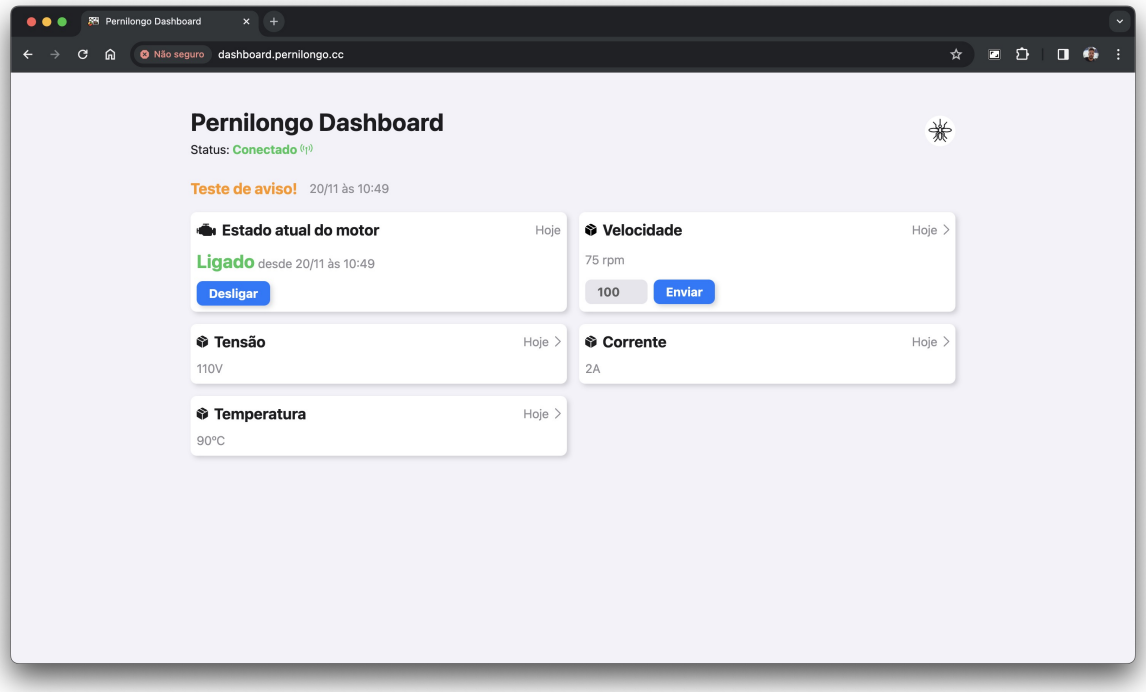


Figura 22 – Página *home* do *dashboard*.
Fonte: Autoria própria.

Web. A Figura 24 a seguir apresenta um resumo do resultado obtido. O resultado completo pode ser visualizado no Anexo A.

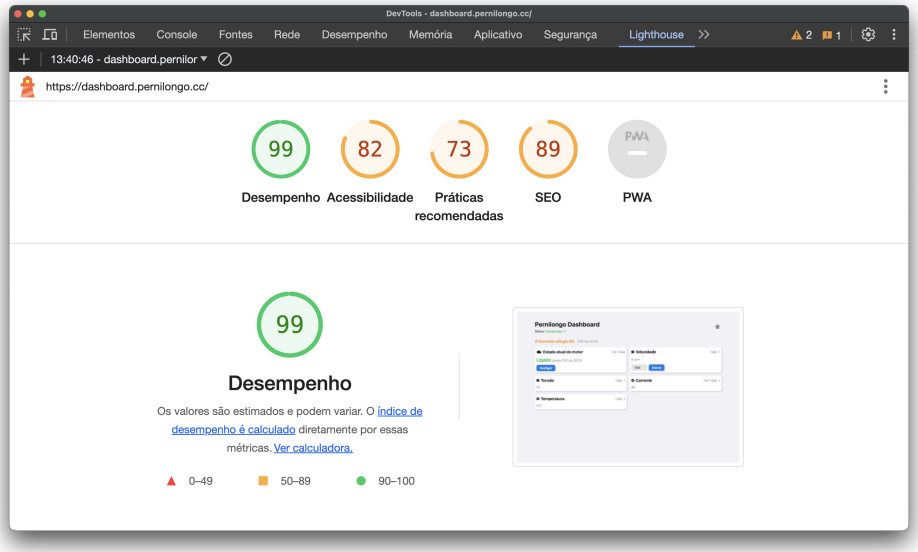


Figura 24 – Resumo dos resultados da auditoria do *dashboard* utilizando o Google *Lighthouse*.

Os resultados apresentados na auditoria trazem algumas informações relevantes a respeito da página. A primeira informação está relacionada ao alto **Desempenho** (99/100) obtido. Sabe-se que SPAs comumente possuem um desempenho mais baixo

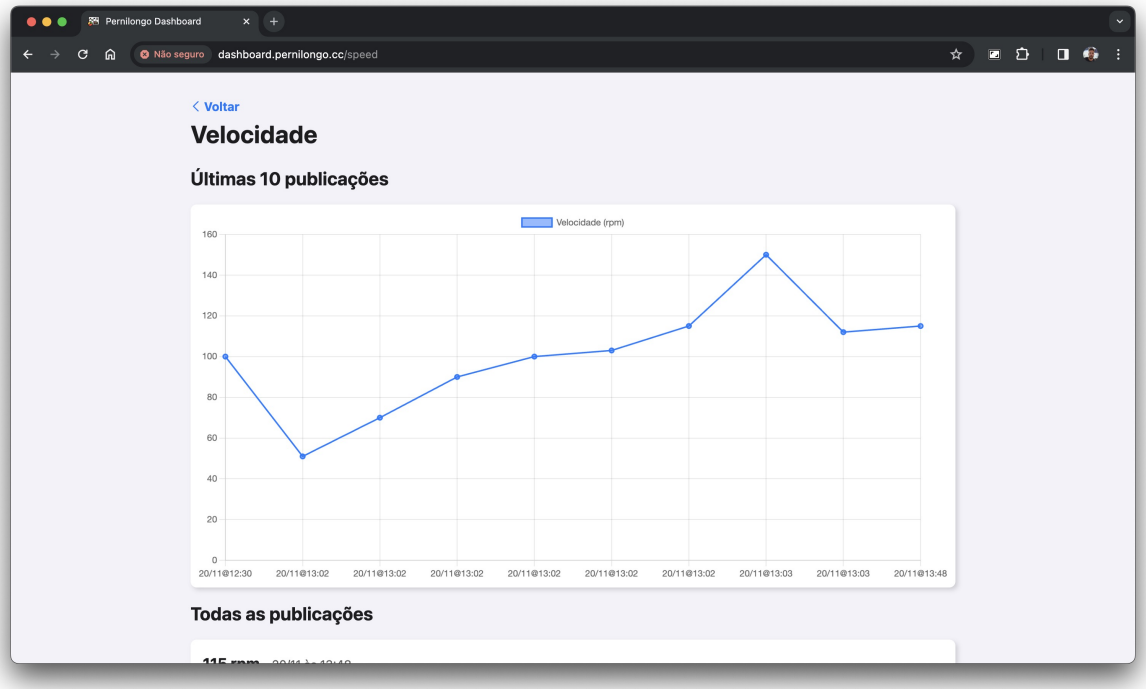


Figura 23 – Página de um tópico no *dashboard*.
Fonte: Autoria própria.

quando comparadas a sites estáticos, devido à necessidade de carregar o conteúdo da página com JavaScript, enquanto sites estáticos servem o conteúdo (HTML) pronto. O alto desempenho obtido, portanto, pode ser entendido como uma consequência da simplicidade e da baixa quantidade de conteúdo carregado dinamicamente na página, além da ausência de *assets* pesados (como imagens, vídeos ou animações) que devem ser carregados para exibir o conteúdo da página, tornando-a tão rápida quanto um site estático.

No que se refere à **Acessibilidade** da página, tem-se que, de acordo com os pontos listados pela auditoria (Anexo A), os principais quesitos que contribuíram para que a nota não fosse ótima (isto é, acima de 90/100) foram: o baixo contraste em alguns textos, que pode dificultar a leitura para alguns usuários; a ausência de elementos de *label* (rótulo) para identificar a informação a ser inserida nos campos de *input*; e a não utilização de elementos de título em uma ordem sequencial descendente, o que pode dificultar o entendimento e a navegação pelo conteúdo da página, especialmente para usuários que utilizam leitores de tela.

Já no quesito **SEO** (Serch Engine Optimization), tem-se que o principal ponto responsável por piorar a nota obtida foi a ausência de metadescrição com informações a respeito da aplicação que possam ser lidas por *crawlers* de ferramentas de busca. Entretanto, considerando o contexto dessa aplicação, entende-se que não há a necessidade de otimização para ferramentas de busca.

Por último, no quesito **Práticas recomendadas**, o resultado da auditoria destaca

o carregamento do chamado “conteúdo misto” na página como um risco de segurança. O termo “conteúdo misto” se refere ao fato de que a página, carregada por um protocolo seguro — HTTPS —, está carregando conteúdo obtido por um protocolo não seguro — WS ao invés de WSS —, ou seja, de fato misturando conteúdo seguro e não seguro, de forma que a página inteira seja considerada não segura pelo navegador. Na verdade, a maioria dos navegadores Web nem sequer permitem o carregamento de “conteúdo misto”: foi necessário alterar as configurações do navegador — neste caso, o Google Chrome — para que se permitisse esse comportamento nessa página.

4.3 API

A implementação da API possibilitou um melhor controle das informações que foram lidas e escritas no banco de dados, permitindo que tanto o *Broker* como o *dashboard* se comunicassem com a API e ao mesmo tempo garantindo uma camada de proteção ao banco. A documentação das rotas pode ser visualizada acessando a página principal da API no endereço https://api.pernilongo.cc (Figura 25). A transcrição dessa documentação está apresentada no Apêndice B.

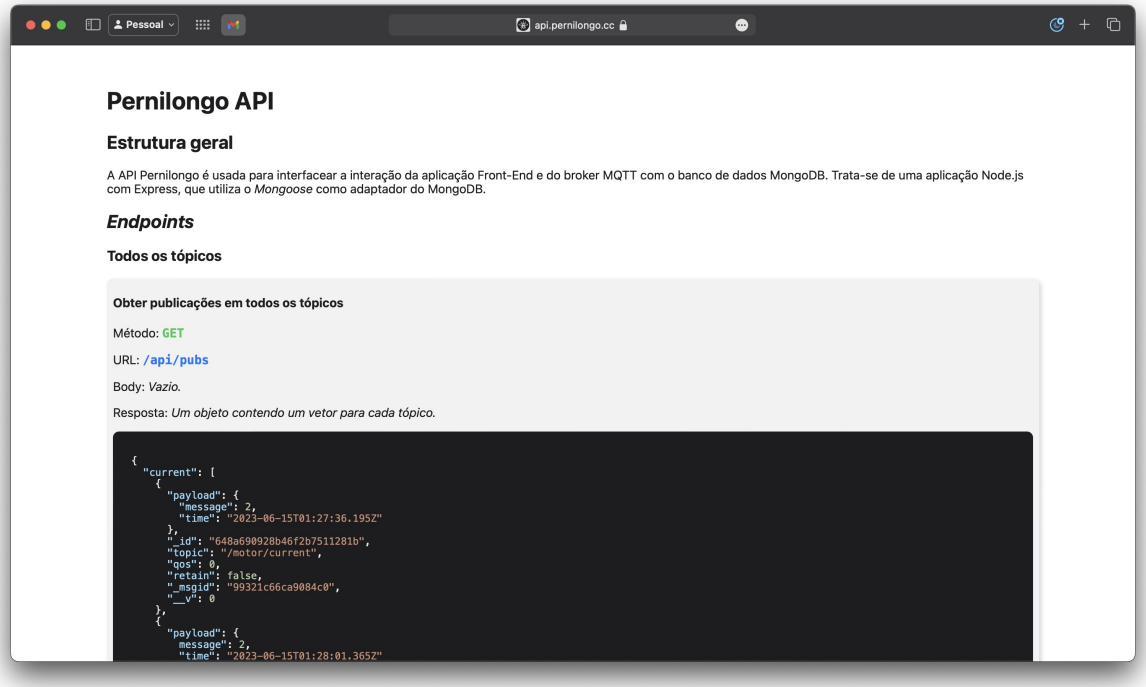


Figura 25 – Página *home* da API.
Fonte: Autoria própria.

A fim de obter informações a respeito do desempenho da API, optou-se por realizar um *stress test* (teste de estresse). Trata-se de um teste automatizado que executa repetidas requisições a todas as rotas da API durante um determinado período de tempo e analisa como esta se comporta durante esse período. Dois dos principais parâmetros para análise

de performance de uma API são a média do tempo de resposta das requisições e a média da porcentagem de erro das respostas. A ferramenta escolhida para realizar o teste foi o aplicativo Postman, que também foi utilizado durante a fase de desenvolvimento para testar o comportamento das rotas individualmente. O resultado completo desse teste é apresentado no Anexo B.

Analisando os resultados do teste é possível reparar que a API apresenta tanto a média de erro quanto a média de tempo de resposta particularmente elevados nas primeiras requisições. Esse resultado é coerente com o comportamento percebido durante a utilização da interface, onde o carregamento dos dados vindos da API, em um primeiro momento, demorava mais que o normal (na escala dos segundos) — seguido por esse tempo de resposta apresentando valores consideravelmente menores (na escala dos milissegundos) nas requisições subsequentes. Esse comportamento está associado ao fato de a API ter sido implementada na forma de uma *serverless function*, isto é, uma função sem servidor. Isso significa que, de fato, não há um servidor de origem, permitindo que a função seja executada em um servidor de borda — servidor localizado perto do usuário.

O ponto negativo desta forma de implementação é que, após certo tempo sem ser utilizada, a função precisa ser alocada e carregada novamente antes que possa ser de fato executada ao receber uma requisição, podendo ser executada diretamente para a requisições recebidas subsequentemente. Esse problema poderia, portanto, ser resolvido através da reimplementação da API com uma arquitetura tradicional, ou seja, com servidor.

Outra informação que pode ser obtida dos resultados do teste é a de que, mesmo após as requisições iniciais, isto é, após o *cold start*, a média dos tempos médios de resposta das requisições em todas as rotas ainda permaneceu relativamente elevada — próxima a 2 segundos. A análise dos tempos médios individuais de cada rota permite identificar que a rota responsável por elevar a média geral foi a rota responsável por retornar as informações de todos os tópicos — com um tempo de resposta médio superior a 6 segundos.

Esse comportamento não é surpreendente, considerando que a maneira com que essa rota foi desenvolvida faz com que as publicações sejam lidas do banco de dados de maneira sequencial, ou seja, um tópico de cada vez, de forma que o tempo para obter as publicações de todos os tópicos seja próximo da soma dos tempos para obter as publicações de cada tópico. Uma alternativa para melhorar esse tempo de resposta seria fazer com que as informações de cada tópico fossem lidas do banco em paralelo.

5 CONCLUSÃO

O sistema desenvolvido possui como elemento fundamental o *Broker* MQTT, que, aproveitando o poder e a confiabilidade da AWS (*Amazon Web Services*), estabelece um canal de comunicação eficiente para dispositivos IoT. O *Broker* MQTT serve como pilar do sistema, fazendo o roteamento das mensagens (publicações) entre os clientes MQTT, além da realização de validações e do disparo de notificações por e-mail.

Complementando o *Broker* MQTT, o *dashboard* interativo hospedado na Vercel é capaz de proporcionar aos usuários uma visualização remota e em tempo real dos dados gerados pelos dispositivos IoT, oferecendo uma interface amigável para monitorar e analisar informações cruciais.

A API desenvolvida para lidar com a lógica de Back-End e gerenciamento de dados, implementada com uma arquitetura sem servidor (*serverless function*) hospedada na Vercel, facilita a comunicação do *dashboard* e do *Broker* com o banco de dados MongoDB, hospedado no Atlas, fornecendo uma camada de proteção ao banco. O banco de dados armazena e organiza os dados coletados, garantindo sua persistência e acessibilidade para análises futuras.

Embora o sistema desenvolvido demonstre com sucesso a integração destas tecnologias, é importante reconhecer as suas limitações. O projeto, concebido como uma prova de conceito, foi projetado para demonstrar a viabilidade da arquitetura proposta, ao invés de servir como uma solução pronta para produção. Como resultado, o sistema tem certas concessões e restrições inerentes, que justificam um refinamento adicional para implantação em grande escala, como mencionado no capítulo de resultados.

Entre as limitações identificadas estão potenciais desafios de escalabilidade, considerações de segurança e a necessidade de mecanismos de tratamento de erros mais robustos. Além disso, futuras iterações do sistema poderiam se beneficiar de recursos aprimorados, como autenticação de usuário, melhorias de desempenho e opções aprimoradas de visualização no painel.

Concluindo, entende-se que o projeto atingiu com sucesso os seus objetivos principais ao ilustrar o potencial de um sistema Web voltado para aplicações IoT. Embora o sistema possa ter limitações, sua capacidade de produzir os resultados esperados serve como uma base sólida para o desenvolvimento e aperfeiçoamento futuro. A jornada de criação deste sistema forneceu informações valiosas sobre as complexidades e oportunidades no domínio do desenvolvimento de aplicações IoT, abrindo caminho para maior exploração e inovação neste campo dinâmico.

REFERÊNCIAS

Apple Inc. **Apple Human Interface Guidelines**. 2023. Disponível em: https://developer.apple.com/design/human-interface-guidelines/. Acesso em: 20 nov. 2023.

BAYASGALAN, Z.; BYAMBASUREN, B.-E.; TUDEVDAGVA, U. Advanced energy and industrial iot. **Embedded Selforganising Systems**, v. 10, p. 2–3, 06 2023. Disponível em: https://www.researchgate.net/publication/373859106_Advanced_Energy_and_Industrial_IoT. Acesso em: 6 nov. 2023.

BENSAKHRIA, A. **IoT - Communicating with devices: Introduction to MQTT and HTTP**. [*S.l.: s.n.*], 2020. Disponível em: https://www.researchgate.net/publication/344217719_IoT_-_Communicating_with_devices_Introduction_to_MQTT_and_HTTP. Acesso em: 11 nov. 2023.

CUNNINGHAM, K. **Accessibility Handbook**: Making 508 websites for everyone. O'Reilly Media, Inc., 2012. Disponível em: https://www.academia.edu/26417718/_The_Accessibility_Handbook. Acesso em: 14 nov. 2023.

FROEHLICH, A. **OSI model (Open Systems Interconnection)**: Definition. 2023. Disponível em: https://www.techtarget.com/searchnetworking/definition/OSI. Acesso em: 6 nov. 2023.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **DATABASE SYSTEMS**: The complete book. Second edition. Pearson Prentice Hall, Upper Saddle River, N.J., 2008. Disponível em: https://people.inf.elte.hu/miiqaai/elektroModulatorDva.pdf. Acesso em: 15 nov. 2023.

HAMMOUDI, A.; KHANFAR, K. Osi model. 02 2022. Disponível em: https://www.researchgate.net/publication/358768960_OSI_Model. Acesso em: 6 nov. 2023.

HIVEMQ. **MQTT & MQTT 5 Essentials**: A comprehensive overview of mqtt facts and features for beginners and experts alike. 2020. Disponível em: https://www.hivemq.com/downloads/hivemq-ebook-mqtt-essentials.pdf. Acesso em: 11 nov. 2023.

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a internet**: uma abordagem top-down. 6ª edição. ed. Pearson Education do Brasil Ltda., 2014. Disponível em: https://github.com/free-educa/books/blob/main/books/Livro%20de%20Rede%20de%20Computadores%20e%20a%20Internet.pdf. Acesso em: 15 nov. 2023.

LOMBARDI, A. **WebSocket**: Lightweight client-server communications. O'Reilly Media, Inc., 2015. Disponível em: https://pepa.holla.cz/wp-content/uploads/2016/12/WebSocket.pdf. Acesso em: 13 nov. 2023.

MELNIKOV, A.; FETTE, I. **The WebSocket Protocol**. RFC Editor, 2011. RFC 6455. (Request for Comments, 6455). Disponível em: https://www.rfc-editor.org/info/rfc6455. Acesso em: 13 nov. 2023.

MOZILLA. **MDN Web Docs**: An overview of http. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Data urls. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URLs. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Http request methods. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Http messages. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Http headers. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Http response status codes. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Https. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Glossary/HTTPS. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Secure sockets layer (ssl). 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Glossary/SSL. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Keep-alive. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Keep-Alive. Acesso em: 8 nov. 2023.

MOZILLA. **MDN Web Docs**: Understanding client-side javascript frameworks. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks. Acesso em: 14 nov. 2023.

POLLOCK, J. **JavaScript**: A beginner's guide. Fourth edition. The McGraw-Hill Companies, 2013. Disponível em: https://pepa.holla.cz/wp-content/uploads/2015/11/JavaScript-A-Beginners-Guide-Fourth-Edition1.pdf. Acesso em: 14 nov. 2023.

POWELL, T. A. **HTML & CSS**: The complete reference. Fifth edition. The McGraw-Hill Companies, 2010. Disponível em: https://www.dcpehvpm.org/E-Content/BCA/BCA-II/Web%20Technology/the-complete-reference-html-css-fifth-edition.pdf. Acesso em: 14 nov. 2023.

RICHARDSON, L.; AMUNDSEN, M. **RESTful Web APIs**. First edition. O'Reilly Media, Inc., 2013. Disponível em: https://sd.blackball.lv/library/RESTful_Web_APIs_(2013).pdf. Acesso em: 15 nov. 2023.

APÊNDICES

APÊNDICE A – *DESIGN MOBILE* RESPONSIVO

Figura 26 – *Layout* do *dashboard* adaptado a telas menores (*mobile*).
Fonte: autoria própria

APÊNDICE B – TRANSCRIÇÃO DA DOCUMENTAÇÃO DA API

Pernilongo API

B.1 Estrutura geral

A API Pernilongo é usada para interfacear a interação da aplicação Front-End e do broker MQTT com o banco de dados MongoDB. Trata-se de uma aplicação Node.js com Express, que utiliza o Mongoose como adaptador do MongoDB.

B.2 Endpoints

B.2.1 Todos os tópicos

B.2.1.1 Obter publicações em todos os tópicos

Método: `GET`

URL: `/api/pubs`

Body: Vazio.

Resposta: Um objeto contendo um vetor para cada tópico.


```
{
  "current": [
    {
      "payload": {
        "message": 2,
        "time": "2023-06-15T01:27:36.195Z"
      },
      "_id": "648a690928b46f2b7511281b",
      "topic": "/motor/current",
      "qos": 0,
      "retain": false,
      "_msgid": "99321c66ca9084c0",
      "__v": 0
    },
    {
      "payload": {
        message": 2,
        "time": "2023-06-15T01:28:01.365Z"
      },
```

```
      "_ id": "648a6921de0d30cd70395159",
      "topic": "/motor/current",
      "qos": 0,
      "retain": false,
      "_msgid": "caaff3e431ad5145",
      "__v": 0
    },
    ...
  ],
  "speed": [ ... ],
  "state": [ ... ],
  "temperature": [ ... ],
  "tension": [ ... ],
  "warning": [ ... ]
}
```


B.2.2 Único Tópico

B.2.2.1 Obter publicações de um tópico específico

Método: `GET`

URL: `/api/pubs/:topic`

Body: Vazio.

Resposta: Um vetor com as publicações no tópico.


```
[
  {
    "payload": {
      "message": 2,
      "time": "2023-06-15T01:27:36.195Z"
    },
    "_id": "648a690928b46f2b7511281b",
    "topic": "/motor/current",
    "qos": 0,
    "retain": false,
    "_msgid": "99321c66ca9084c0",
    "__v": 0
  },
  {
    "payload": {
```

```
      message": 2,
      "time": "2023-06-15T01:28:01.365Z"
    },
    "_ id": "648a6921de0d30cd70395159",
    "topic": "/motor/current",
    "qos": 0,
    "retain": false,
    "_msgid": "caaff3e431ad5145",
    "__v": 0
  }
]
```


B.2.2.2 Criar nova uma publicação em um tópico específico

Método: `POST`

URL: `/api/pubs/:topic`

Body: É necessário passar um objeto (JSON) contendo a publicação como no exemplo a seguir.


```
{
  "payload": {
    "message": 2,
    "time": "2023-06-15T01:27:36.195Z"
  },
  "topic": "/motor/current",
  "qos": 0,
  "retain": false,
  "_msgid": "99321c66ca9084c0",
}
```


Resposta: Como confirmação, a API retorna o objeto salvo com adição dos campos "`_id`" e "`__v`", adicionados automaticamente pelo MongoDB.


```
{
  "payload": {
    "message": 2,
    "time": "2023-06-15T01:27:36.195Z"
  },
  "_id": "648a690928b46f2b7511281b",
  "topic": "/motor/current",
```

```
  "qos": 0,
  "retain": false,
  "_msgid": "99321c66ca9084c0",
  "__v": 0
}
```

ANEXOS

ANEXO A – RESULTADO DA AUDITORIA DO *DASHBOARD* UTILIZANDO O GOOGLE *LIGHTHOUSE*

https://dashboard.pernilongo.cc/

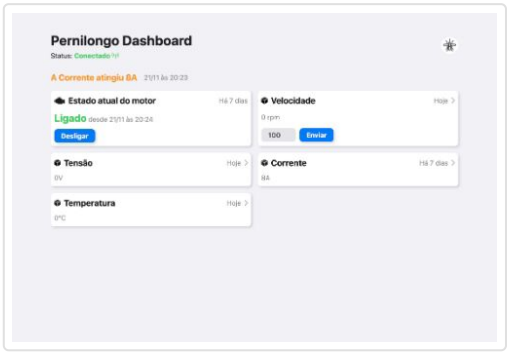
99	82	73	89	PWA
Desempenho	Acessibilidade	Práticas recomendadas	SEO	PWA

99

Desempenho

Os valores são estimados e podem variar. O índice de desempenho é calculado diretamente por essas métricas. Ver calculadora.

▲ 0–49 ■ 50–89 ● 90–100

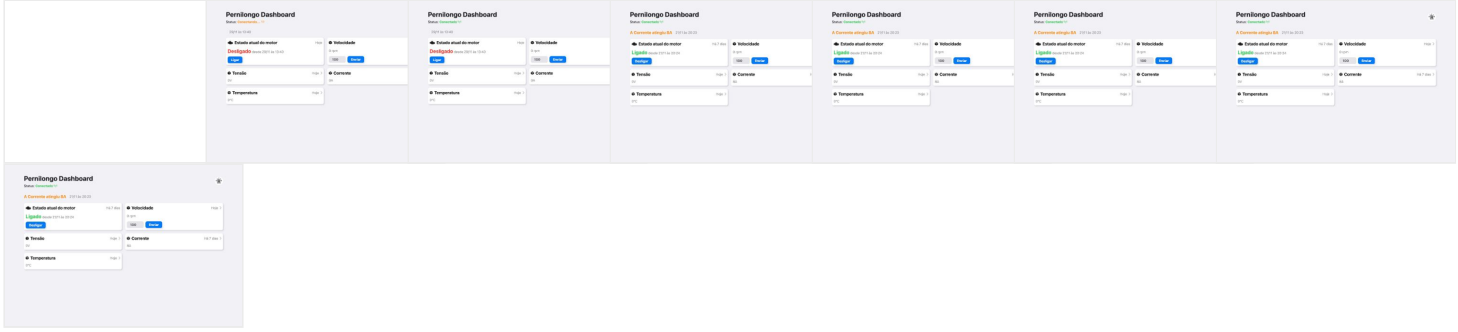


MÉTRICAS

Abrir visualização

- First Contentful Paint **0,6 s**
- Largest Contentful Paint **0,6 s**
- Total Blocking Time **0 ms**
- Cumulative Layout Shift **0,069**
- Speed Index **0,7 s**

Ver Treemap



Mostrar auditorias relevantes para: All FCP LCP TBT CLS

DIAGNÓSTICO

▲ A página impede a restauração do cache de avanço e retorno — 1 motivo para a falha

- Evitar encadeamento de solicitações críticas — 3 redes encontradas
- Elemento de Maior exibição de conteúdo — 630 ms
- Evite grandes mudanças no layout — 4 elementos encontrados

Mais informações sobre o desempenho do seu aplicativo. Esses números não afetam diretamente o índice de desempenho.

AUDITORIAS APROVADAS (35) Mostrar

82

Acessibilidade

Essas verificações destacam oportunidades para melhorar a acessibilidade do seu app da Web. A detecção automática só detecta um subconjunto de problemas e não garante a acessibilidade do seu app da Web. Portanto, também recomendamos o teste manual.

CONTRASTE

- ▲ As cores de primeiro e segundo plano não têm uma taxa de contraste suficiente.

Veja aqui oportunidades de melhorar a legibilidade do seu conteúdo.

NOMES E ETIQUETAS

- ▲ Os elementos de formulário não têm etiquetas associadas

Veja aqui oportunidades de melhorar a semântica dos controles do seu aplicativo. Isso pode melhorar a experiência de usuários de tecnologias assistivas, como leitores de tela.

NAVEGAÇÃO

- ▲ Os elementos de título não aparecem em uma ordem sequencial descendente

Veja aqui oportunidades de melhorar a navegação por teclado no seu aplicativo.

OUTROS ITENS PARA VERIFICAÇÃO MANUAL (10) Mostrar

Esses itens se referem a áreas que uma ferramenta de teste automatizada não pode cobrir. Saiba mais no nosso guia sobre como realizar uma avaliação de acessibilidade.

AUDITORIAS APROVADAS (9) Mostrar

NÃO APLICÁVEL (49) Mostrar

73

Práticas recomendadas

GARANTIA E SEGURANÇA

- ▲ Não utiliza HTTPS — 1 solicitação não segura encontrada
- ○ Conferir se a CSP é eficaz contra ataques de XSS

GERAL

- ▲ Os problemas foram registrados no painel `Issues` do Chrome Devtools

AUDITORIAS APROVADAS (11) Mostrar

NÃO APLICÁVEL (2) Mostrar

89

SEO

Essas verificações garantem que sua página siga orientações básicas para otimização de mecanismos de pesquisa. Há muitos outros fatores não avaliados pelo Lighthouse que ainda podem afetar sua classificação na pesquisa, como a performance nas Principais métricas da Web. Saiba mais sobre os Fundamentos da Pesquisa Google.

PRÁTICAS RECOMENDADAS PARA CONTEÚDO

- ▲ O documento não tem uma metadescrição

Formate seu HTML de maneira que permita que os rastreadores entendam melhor o conteúdo do seu app.

OUTROS ITENS PARA VERIFICAÇÃO MANUAL (1) Mostrar

Execute estes validadores adicionais no seu site para verificar mais práticas recomendadas de SEO.

AUDITORIAS APROVADAS (8) Mostrar

NÃO APLICÁVEL (5) Mostrar

PWA

Essas verificações validam os aspectos de um App Web Progressivo.
Aprenda a criar um bom App Web Progressivo.

PODE SER INSTALADO

- ▲ O manifesto do app da Web ou o service worker não atendem aos requisitos para instalação — 2 motivos

OTIMIZADO PARA PWA

- ▲ Não foi configurado para uma tela de apresentação personalizada Failures: No manifest was fetched.
- ▲ Não foi definida uma cor de tema para a barra de endereços.
  Failures: No manifest was fetched, No `<meta name="theme-color">` tag found.
- ○ O conteúdo está no tamanho correto para a janela de visualização
- ● Há uma tag `<meta name="viewport">` com `width` ou `initial-scale`
- ▲ O manifesto não tem um ícone mascarável No manifest was fetched

OUTROS ITENS PARA VERIFICAÇÃO MANUAL (3) Mostrar

Essas verificações são solicitadas pela Lista de verificação de PWA de referência, mas não são automaticamente realizadas pelo Lighthouse. Elas não afetam sua pontuação, mas é importante verificá-las manualmente.

Captured at 29 de nov. de 2023, 13:40 BRT
Carregamento inicial da página

Área de trabalho emulada with Lighthouse 11.1.0
Limitação personalizada

Carregamento de uma única página
Using Chromium 119.0.0.0 with devtools

ANEXO B – *STRESS TEST* REALIZADO NAS ROTAS DA API UTILIZANDO O APLICATIVO POSTMAN

Performance test report - Nov 21, 2023 (#2)

Open in Postman

Postman collection: **Pernilongo API**

Report exported on: **Nov 22, 2023, 1:15:52 (GMT-3)**

Test setup

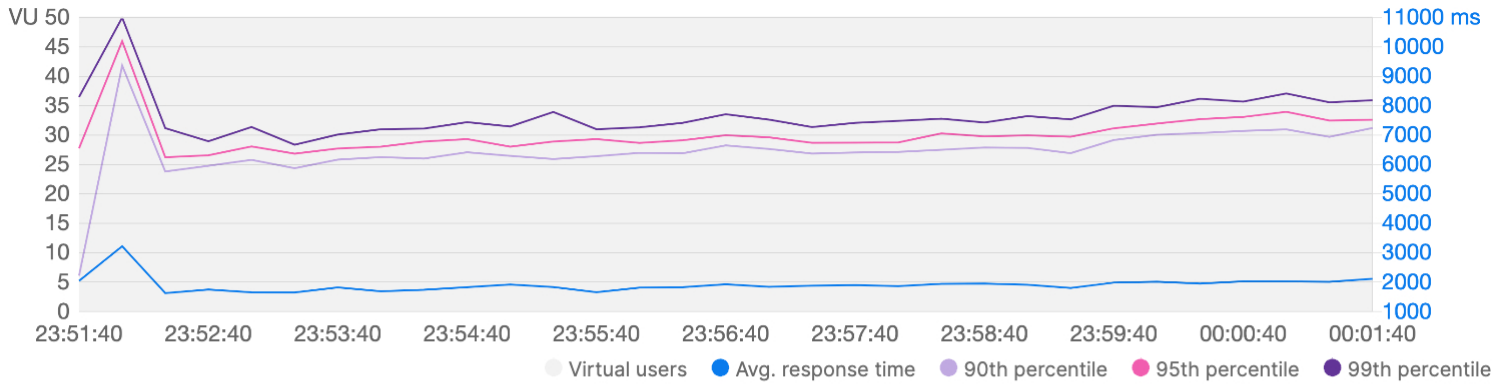
Virtual users	Start time	Load profile
50 VU	Nov 21, 23:51:46 (GMT-3)	Fixed
Duration	**End time**	**Environment**
10 minutes	Nov 22, 0:01:53 (GMT-3)	-

1. Summary

Total requests sent	Throughput	Average response time	Error rate
14,022	23.11 requests/second	1,893 ms	0.35 %

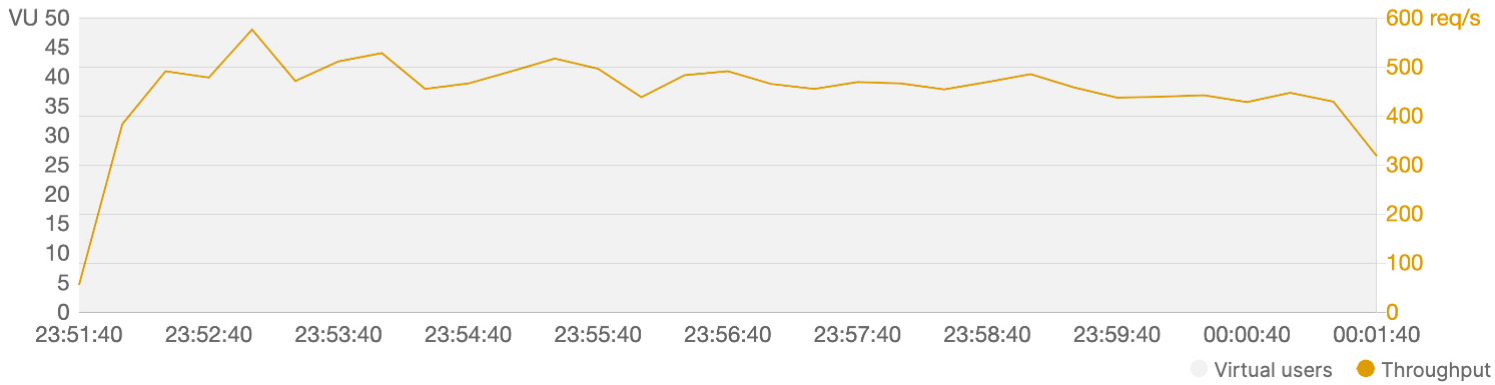
1.1 Response time

Response time trends during the test duration.



1.2 Throughput

Rate of requests sent per second during the test duration.



1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
GET Get all publications https://api.pernilongo.cc/api/pubs	6,313	7,409	7,766	9,781	2,849	11,710
GET Get Speed Pubs https://api.pernilongo.cc/api/pubs/speed	572	722	769	4,213	277	8,165
POST Add new State Pub https://api.pernilongo.cc/api/pubs/speed	447	847	981	1,023	277	5,976
GET Home Page https://api.pernilongo.cc/	249	359	389	1,136	158	2,220

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
GET Get all publications https://api.pernilongo.cc/api/pubs	49	500 Internal Server Error (18)	504 Gateway Time Out (31)	0

2. Metrics for each request

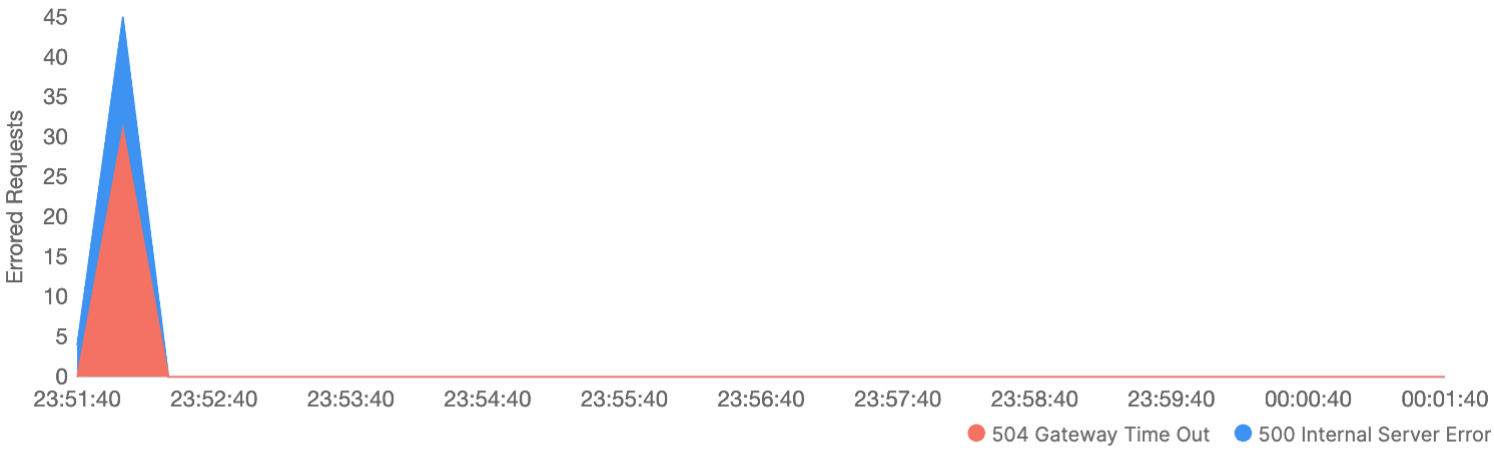
The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
GET Home Page https://api.pernilongo.cc/	3,529	5.82	158	249	359	2,220	0
GET Get all publications https://api.pernilongo.cc/api/pubs	3,501	5.77	2,849	6,313	7,409	11,710	1.4
GET Get Speed Pubs https://api.pernilongo.cc/api/pubs/speed	3,498	5.76	277	572	722	8,165	0
POST Add new State Pub https://api.pernilongo.cc/api/pubs/speed	3,494	5.76	277	447	847	5,976	0

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.



3.2 Error distribution for requests

Errored requests grouped by error class, along with the error count for each class.

Error class	Total counts
504 Gateway Time Out	31
GET Get all publications	31
500 Internal Server Error	18
GET Get all publications	18

Testing API performance on Postman

Postman enables you to simulate user traffic and observe how your API behaves under load. It also helps you identify any issues or bottlenecks that affect performance.

Learn more about testing API performance.