Universidade Federal de Pernambuco

Centro Informática

Curso de Bacharelado em Engenharia da Computação

Data2Board - Uma Plataforma de Visualização de Dados para Dispositivos IoT

Matheus Deodato de Oliveira

Recife, 2023

Matheus Deodato de Oliveira

Data2Board - Uma Plataforma de Visualização de Dados para Dispositivos IoT

Monografia apresentada no Curso de Bacharelado em Engenharia da Computação, como um requisito parcial para obter o Título de Bacharel em Engenharia da Computação, no Centro Informática da Universidade Federal de Pernambuco.

Área de concentração : IoT, Sistemas Embarcados e Visualização de dados

Orientadora: Edna Natividade da Silva Barros

Recife, 2023

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

Oliveira, Matheus Deodato de.
Data2Board - Uma Plataforma de Visualização de Dados para Dispositivos IoT / Matheus Deodato de Oliveira. - Recife, 2023.
36 f : il.

Orientador(a): Edna Natividade da Silva Barros
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.

1. Dispositivos IoT. 2. Simplificação do desenvolvimento. 3. Protocolo MQTT,. 4. Prototipagem rápida. I. Barros, Edna Natividade da Silva. (Orientação). II. Título.

000 CDD (22.ed.)

Creativity is contagious. Pass it on.

Albert Einstein

RESUMO

A plataforma Data2Board foi desenvolvida para acelerar a criação de protótipos de dispositivos IoT, simplificando a implementação da conexão de dados entre o dispositivo IoT e a dashboard. Com essa abordagem, o desenvolvedor pode concentrar seus esforços no desenvolvimento do hardware e na coleta de dados, o que reduz significativamente o tempo de desenvolvimento de protótipos de dispositivos IoT e aumenta a eficiência do processo de criação. Além disso, a plataforma Data2Board permite que o desenvolvedor tenha mais liberdade para explorar novas ideias e soluções para dispositivos IoT, sem ficar limitado pela complexidade do processo de implementação da conexão de dados.

Para enviar os dados coletados pelo dispositivo IoT diretamente para a dashboard, o sistema Data2Board conta com uma biblioteca para ESP32, a qual permite que o usuário crie um grid para uma dashboard web e envie os dados sem a necessidade de implementar todo o código do backend e frontend. Essa abordagem simplifica a implementação da interface gráfica, especialmente para usuários com pouco conhecimento em programação.

Para garantir uma comunicação eficiente e de alta velocidade entre o dispositivo IoT e a dashboard, o sistema utiliza o protocolo MQTT como mecanismo de transporte de dados e a comunicação entre backend e frontend via websocket. Estas tecnologias permitem uma transmissão rápida e confiável de dados, o que garante que as informações coletadas pelo dispositivo IoT sejam exibidas na dashboard em tempo hábil. Com essa abordagem, os usuários podem ter acesso a uma visualização clara e intuitiva dos dados coletados pelo dispositivo IoT, permitindo uma análise precisa e em tempo real..

Palavras-chave: dispositivos IoT, simplificação do desenvolvimento, protocolo MQTT, Prototipagem rápida.

ABSTRACT

The Data2Board platform was developed to accelerate the creation of IoT device prototypes, simplifying the implementation of data connection between the IoT device and the dashboard. With this approach, developers can focus their efforts on hardware development and data collection, significantly reducing the development time of IoT device prototypes and increasing the efficiency of the creation process. In addition, the Data2Board platform allows developers more freedom to explore new ideas and solutions for IoT devices without being limited by the complexity of the data connection implementation process.

To send the collected data from the IoT device directly to the dashboard, the Data2Board system relies on a library for ESP32, which enables users to create a grid for a web dashboard and send data without the need to implement all backend and frontend code. This approach simplifies the implementation of the graphical interface, especially for users with little programming knowledge.

To ensure efficient and high-speed communication between the IoT device and the dashboard, the system uses the MQTT protocol as a data transport mechanism and communication between backend and frontend via websocket. These technologies allow for fast and reliable data transmission, ensuring that the information collected by the IoT device is displayed on the dashboard in a timely manner. With this approach, users can have access to a clear and intuitive visualization of the data collected by the IoT device, allowing for precise and real-time analysis.

Keywords: IoT devices, development simplification, MQTT protocol, Rapid prototyping.

Sumário

1 Introdução

Integrar um dispositivo de sistemas embarcados a um backend e frontend pode ser um desafio significativo para um engenheiro de sistemas embarcados, pois para tal tarefa é necessário conhecimento em diversas áreas de desenvolvimento de software com o qual esse profissional frequentemente não está familiarizado.

O primeiro desafio é garantir que o dispositivo embarcado possa se comunicar com o backend. Isso pode exigir conhecimento em protocolos de comunicação, como MQTT ou HTTP, e habilidades de programação para implementar a comunicação em um ambiente de rede. Além disso, o engenheiro pode precisar lidar com problemas de segurança para proteger a comunicação de dados sensíveis.

O segundo desafio é garantir que os dados coletados pelo dispositivo possam ser armazenados e acessados pelo backend. Isso pode exigir habilidades em bancos de dados e arquitetura de software, como bancos de dados SQL ou NoSQL, e linguagens de programação utilizadas em backend, como Node.js, Ruby on Rails, Java ou Django.

O terceiro desafio é garantir que os dados coletados possam ser exibidos em um frontend, como uma dashboard. Isso pode exigir conhecimento em tecnologias como JavaScript, HTML e CSS, e habilidades em frameworks como React, Angular ou Vue.

Para superar esses desafios, os engenheiros de sistemas embarcados podem precisar trabalhar em colaboração com outros especialistas em desenvolvimento de software e hardware, como desenvolvedores de frontend, backend e segurança cibernética.

O objetivo do trabalho é criar uma plataforma integrada de desenvolvimento para sistemas embarcados com foco principal em ajudar iniciantes e entusiastas a superar os desafios de integração com o backend e frontend, permitindo que criem soluções completas com maior facilidade e rapidez. Com a plataforma, os usuários podem se concentrar em desenvolver suas soluções embarcadas sem ter de se preocupar com a complexidade do backend e frontend, o que tende a acelerar o processo de prototipagem e ajudar a trazer novas soluções para o mercado.

1.1 Trabalhos Relacionados

Existem uma série de plataformas com trabalhos similares, dentre elas destacam-se o Ubidots [1], Blynk [2] e Adafruit IO [3]. A seguir temos uma comparação entre as plataformas a nível de funcionalidade, suporte, e custo.

1.1.1 Funcionalidades

Ubidots: Oferece uma ampla gama de recursos, desde visualização de dados em tempo real até alertas personalizados. A plataforma tem integração com APIs de terceiros e a capacidade de controle de dispositivos remotamente, tornando-a uma escolha versátil para usuários que precisam de um alto nível de personalização.

Blynk: Permite conectar dispositivos inteligentes e controlá-los através de um aplicativo móvel ou da web. Foca em automação residencial e tem recursos específicos, como detecção de movimento, rastreamento de temperatura e controle de iluminação.

Adafruit IO: Possui uma interface simples de arrastar e soltar, tornando fácil a criação de painéis de controle de dispositivos IoT. A plataforma é altamente personalizável e pode ser integrada com outros serviços da Adafruit, como CircuitPython e seus kits de desenvolvimento, tornando-a uma opção popular para usuários que procuram uma solução de IoT personalizada.

1.1.2 Suporte

Ubidots: Fornece suporte via e-mail e chat ao vivo em horário comercial. Além disso, tem uma base de conhecimento bem organizada e uma comunidade de usuários ativos para ajudar na resolução de problemas.

Blynk: Oferece suporte via e-mail e chat ao vivo, mas não tem um horário comercial definido. A plataforma tem uma base de conhecimento extensa, mas não possui uma comunidade de usuários tão ativa quanto a da Ubidots.

Adafruit IO: Oferece suporte via e-mail e um fórum de usuários. A plataforma tem uma base de conhecimento abrangente que cobre a maioria das dúvidas comuns. No entanto, pode ser um pouco mais difícil obter ajuda em tempo real

1.1.3 Custo

Ubidots: Oferece um plano gratuito com limitações no número de dispositivos e dados armazenados. Os planos pagos começam em US $ 49 por mês.

Blynk: Oferece um plano gratuito com limitações no número de dispositivos. Os planos pagos começam em US $ 6.99 por mês.

Adafruit IO: Oferece um plano gratuito com limitações no número de feeds e dados. Os planos pagos começam em US $ 10 por mês.

1.2 Objetivos

A criação de protótipos de dispositivos IoT pode ser um processo complexo e demorado, exigindo conhecimentos técnicos especializados e muitas horas de desenvolvimento. Diante desse cenário, o objetivo do presente trabalho foi desenvolver uma plataforma que permita acelerar esse processo, simplificando a implementação da conexão de dados entre o dispositivo IoT e uma interface de visualização.

A plataforma desenvolvida irá permitir que os usuários criem protótipos de dispositivos IoT de forma mais ágil e eficiente, reduzindo o tempo necessário para a implementação da comunicação entre o dispositivo e a interface de visualização, gerenciamento de dispositivos, de usuários e do banco de dados. Com isso, espera-se que os usuários tenham condições de concentrar-se na parte criativa do processo.

2 Sistema Proposto - Data2Board

2.1 Visão geral

O sistema desenvolvido pode ser visualizado na Figura 1. Nele temos alguns componentes que serão explicados brevemente no decorrer desta seção, e com mais detalhes nas seções seguintes.

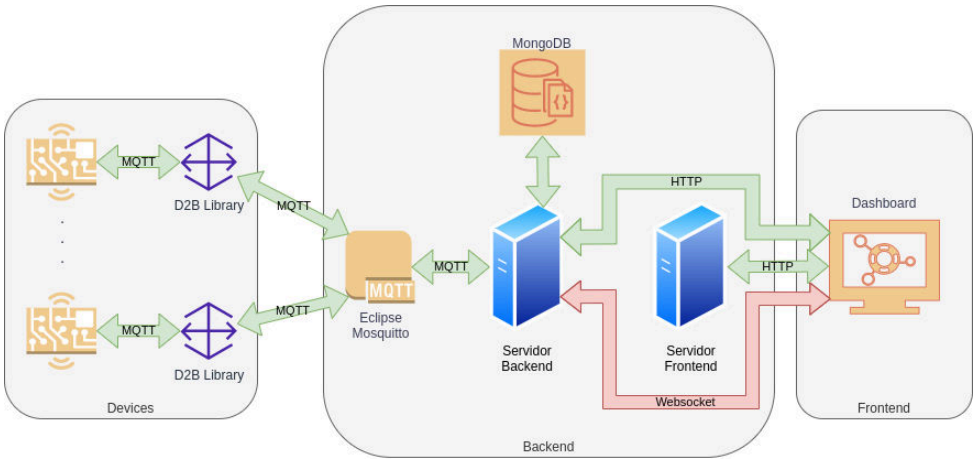

Figura 1 – Diagrama de Visão Geral do Sistema

A primeira etapa do sistema diz respeito aos dispositivos, o trabalho foi desenvolvido para dispositivos que utilizam o microcontrolador ESP32 com códigos produzidos com possibilidade de uso no framework do Arduíno. Para criar a integração com o sistema foi criada uma biblioteca que permite a comunicação do dispositivo com o backend. Para a biblioteca funcionar corretamente o dispositivo precisa ter acesso a uma rede de internet e se conectar via Wi-Fi, a partir daí é estabelecida uma conexão MQTT entre os dispositivos e o backend.

A segunda etapa do projeto, o backend, possui por alguns componentes: um broker MQTT (o Eclipse Mosquitto), um servidor backend escrito no framework NodeJS (responsável pela API) e que é ligado a um banco de dados MongoDB, por fim o servidor de páginas do frontend, que é o responsável por manter o conteúdo estático da plataforma. A figura também exibe os tipos de conexões entre cada módulo do backend. O servidor se comunica com o broker via MQTT, coletando todos os dados recebidos por ele, os processa e salva no banco de dados. Caso o usuário esteja acessando algum dispositivo

que recebeu dados, então esses dados são encaminhados para a dashboard via websocket ou HTTP (mais detalhes nas proximas seções). Por fim o servidor estático das páginas web que se conecta com a dashboard via HTTP.

A ultima etapa é o dashboard de visualização dos dados que é composto por um grid que pode ser subdividido em linhas e colunas, onde cada posição do grid (que será chamado de *campo* no trabalho) pode ter seu comportamento definido diretamente pelo código embarcado nos dispositivos. Cada linha possui três campos, para configurar o lugar correto de um campo no grid o usuário precisa definir a linha e a coluna onde o campo será definido. Existem três tipos de campos: *card* para informações instantâneas do dispositivo, *chart* para criação de gráficos e *switch* para enviar informações de *on/off* para o dispositivo. Vale salientar que a contagem das linhas e colunas é a partir do 0, ou seja, o primeiro campo deve ser declarado como linha 0 e coluna 0. A Figura 2 demonstra um grid com seis campos do tipo *card* que informam a linha e coluna ao qual foram cadastrados no dispositivo.

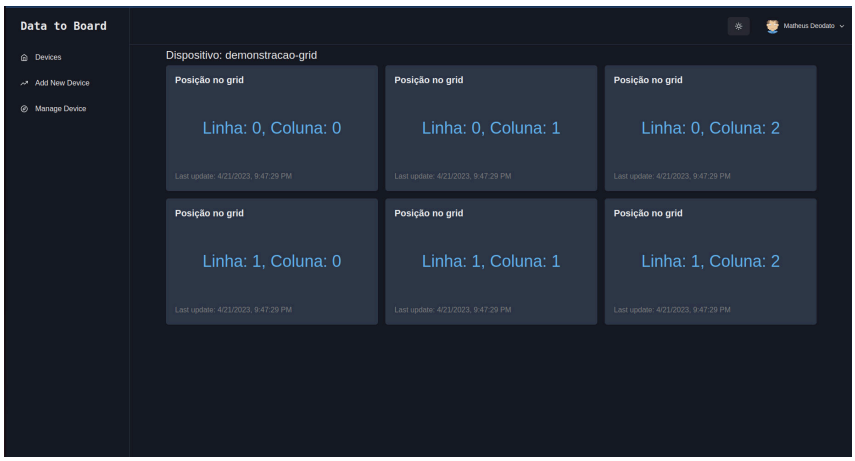

Figura 2 – Diagrama de Visão Geral do Sistema

2.2 Implementação do backend

2.2.1 Broker MQTT - Mosquitto

O Eclipse Mosquitto [4] foi escolhido como broker MQTT para o projeto devido à sua natureza open source e a capacidade de personalização. Como uma plataforma de código aberto, o Mosquitto oferece uma ampla gama de recursos de suporte e atualiza-

ções contínuas da comunidade de desenvolvedores. Além disso, sua facilidade de uso e configuração tornaram a implementação do broker MQTT mais simples e rápida.

Para facilitar ainda mais a utilização do Mosquitto, o broker foi encapsulado em um container Docker. Essa escolha permitiu a criação de um ambiente de desenvolvimento de rápida instalação. Além disso, a utilização de containers aumentou a portabilidade do projeto, permitindo que o mesmo seja executado em diferentes sistemas operacionais e infraestruturas em nuvem.

2.2.2 Backend

O Backend do sistema foi escrito em NodeJS [5]. As principais funções que o script responde são o sistema de recepção e filtragem de mensagens vindas do Mosquitto, os processos de autenticação de usuários, cadastro de usuários, cadastro de dispositivos, gerenciamento dos websockets, e envio de mensagens para os dispositivos.

O código do servidor backend possui uma API que é acessada via rotas HTTP. Uma *rota* é um nome para um endereço de chamada a uma API via HTTP (por exemplo o endereço http://www.dominio.com/listar_dominios, está fazendo uma chamada HTTP do tipo GET a API respondida pelo servidor hospedado em *www.dominio.com* com a rota */listar_dominios*). As características do sistema podem ser analisadas por meio das rotas que a API do sistema respodem, que são explicadas nas próximas seções:

2.2.2.1 Rota */login*

Quando essa rota é acionada pelo frontend, são passados dois argumentos, e-mail e senha. Esses dados são validados, e caso estejam cadastrados então um token de autenticação JWT (JSON Web Token) [6] é retornado para os solicitante. Com esse token, o usuário (páginas do frontend) consegue acessar informações privadas do usuário.

2.2.2.2 Rota */register*

O objetivo dessa rota é registrar novos usuários. Quando a rota é acionada, então é esperado o envio de três dados: nome do usuário, e-mail e senha. Com esses dados o cadastro na plataforma é realizado (desde que não exista outro usuário com o mesmo e-mail).

2.2.2.3 Rota */register_device*

Essa rota é responsável por cadastrar um dispositivo. A rota precisa de um argumento, o nome do dispositivo. Caso não exista outro dispositivo com o mesmo nome, então o dispositivo é criado e retornado o Token do dispositivo para o frontend. Vale salientar que esta rota é privada (apenas usuários autenticados conseguem acessar).

2.2.2.4 Rota */get_devices*

A rota tem por finalidade retornar todos os dispositivos cadastrados (no usuário que solicitou). Vale salientar que esta rota é privada (apenas usuários autenticados conseguem acessar).

2.2.2.5 Rota */devices/:deviceId*

Acionando essa rota o valor passado em 'deviceId' é o nome de um dispositivo, com esse valor os dados de grid, e os últimos valores recebidos do dispositivo são enviados para o frontend (para preencher o grid). Vale salientar que esta rota é privada (apenas usuários autenticados conseguem acessar).

2.2.3 Sistema de verificação de autenticação

O sistema de verificação de autenticação dos usuários foi escrito como um middleware provido pelo modulo Express [7] do NodeJS. Com ele é possivel interceptar as chamadas feitas as suas rotas definidas e verificar se a requisição enviou um token JWT válido. Caso o token seja válido, então a requisição é processada, caso contrário a requisição retorna acesso negado ao solicitante. O mesmo tipo de verificação também é realizada para as conexões via websocket.

2.2.4 Esquema do banco de dados

O MongoDB [8] foi adotado como o banco de dados para o projeto devido à sua excelente integração com backends em NodeJS e sua flexibilidade no armazenamento de dados em formato de documentos. Este banco de dados NoSQL utiliza documentos JSON para armazenar os dados, o que o torna uma opção altamente popular para projetos em Node.js. Isso se deve ao fato de que o Node.js é um framework JavaScript que possui

suporte nativo a JSON (JavaScript Object Notation). Em conjunto, o uso do MongoDB e do Node.js proporcionam uma abordagem ágil e escalável para o desenvolvimento de aplicativos.

No MongoDB, um documento é a unidade básica de armazenamento de dados. Ele é representado em formato BSON (Binary JSON) e é composto por um conjunto de campos com seus respectivos valores.

Cada documento é armazenado em uma coleção, que é semelhante a uma tabela em um banco de dados relacional. No entanto, ao contrário de um esquema rígido em um banco de dados relacional, as coleções do MongoDB podem conter documentos com campos diferentes e tipos de dados variados.

No projeto, foi adotado o Mongoose Schema [9], uma ferramenta de gerenciamento do MongoDB para NodeJS que permite a criação de modelos para garantir a consistência dos dados inseridos. O modelo define a estrutura dos dados, incluindo os tipos de dados permitidos para cada campo, e é utilizado para validar os dados antes de serem inseridos no banco de dados.

Com o Mongoose Schema, é possível estabelecer restrições e regras para cada campo, como valores mínimos e máximos, valores padrão, e até mesmo validação personalizadas baseada em funções. Isso garante que as informações armazenadas no banco de dados estejam sempre no formato esperado, evitando erros e inconsistências de dados.

Dessa forma, cada vez que um novo dado é adicionado ao banco de dados, o Mongoose Schema verifica se o valor a ser inserido está de acordo com o modelo definido. Se o valor não atender aos critérios estabelecidos, ele é rejeitado e um erro é gerado, impedindo a inserção de dados inválidos. Essa funcionalidade oferece segurança e confiabilidade aos dados armazenados, além de facilitar a manutenção do sistema ao longo do tempo. O código abaixo descreve o Mongoose Schema utilizado no projeto. O documento descreve todos os campos de dados e seus tipos.

```
1 user: {
2     name: {
3       type: String,
4       required: true,
5     },
6     email: {
7       type: String,
```

```
    unique: true,
    required: true,
    match: /.+\@.+\..+/,
  },
  password: {
    type: String,
    required: true,
  },
  devices: [
    {
      deviceName: String,
      userName: String,
      deviceToken: String,
      serverIP: String,
      grid: {
        positions: [
          {
            field_type: String,
            x_pos: Number,
            y_pos: Number,
            field_id: Number,
            title: String,
            description: String,
            x_axis_unity: String,
            y_axis_unity: String,
            line_color: String,
            unity: String,
          },
        ],
      },
      data: [
        {
          field_id: Number,
          values: [Mixed],
        },
      ],
    },
  ],
},
```

```
47   {
48     timestamps: true,
49   }
50 );
```

No código conseguimos ver a declaração das restrições para cada campo do documento. Por exemplo, o campo "nome" é do tipo String e deve ser preenchido obrigatoriamente. Da mesma forma, o campo "email" deve ser do tipo String, ser único e ter um formato específico definido pelo campo *match*, além de também ser obrigatório.

2.2.5 Fluxo de dados do Mosquitto

O backend recebe todos os dados do Mosquitto por meio de um módulo chamado mqtt [10] . Com ele é possível criar um client que ouve todos os eventos que acontecem no Broker. Quando uma mensagem é recebida o client a redireciona para um decodificador de mensagens que a destina para alguns caminhos (mensagem de dados, de definição de grid, etc).

Caso seja uma mensagem de publicação de grid (*grid_publish*) então o grid é inserido ou atualizado no dispositivo (caso seja um dado válido). Caso seja uma mensagem de envio de dados, então o dado é inserido no bando de dados e também enviado via websocket caso o usuário esteja com a dashboard do dispositivo aberta.

2.2.6 WebSocket

O websoket foi implementado utilizando o módulo Socket.IO [11]. Com esse módulo conseguimos estabelecer uma conexão duradoura entre o frontend e o backend, com uma baixa latência. Por meio desse canal são trocadas informações sobre o estado dos campos do grid. Caso um campo da dashboard (um switch, por exemplo) seja modificado, então essa mensagem é enviada para o backend via websocket, e por fim o valor é enviado para o dispositivo via MQTT.

2.3 Implementação do frontend

O frontend do projeto foi desenvolvido em React [12]. O React é uma framework frontend de código aberto desenvolvido em JavaScript e foi desenvolvido para criar inter-

faces de usuário em páginas web. Para melhorar esteticamente a interface foi utilizada uma biblioteca de componentes, a Chakra UI [13], com ela muitos componentes já possuem um apelo visual melhorado. Além desses, outros módulos foram utilizados, como o ChartJS [14] para a criação de gráficos, o SocketIO para a comunicação via websocket, também o Axios [15] que permite fazer requisições HTTP ao backend.

As próximas seções descrevem as principais páginas do projeto, e seu funcionamento e comunicação com o backend. Nas seções não são mostradas as imagens, pois temos as mesmas na seção 3.1.

2.3.1 Telas de Login e Cadastro

A tela de cadastro de usuários é composta por um formulário onde o usuário deve preencher os campos de nome de usuário, e-mail e senha. Os dados preenchidos são enviados ao backend para validações (se já existe algum usuário com o mesmo e-mail, por exemplo), caso algum dado seja inválido então uma mensagem é exibida ao usuário, em caso de sucesso o usuário é redirecionado para a página de login para que entre na plataforma.

A tela de login é intuitiva, basta inserir os dados de cadastro e clicar no botão de login, em caso de ocorrer uma falha então uma mensagem será exibida com o motivo. O processo de autenticação é realizado no processo de login onde a seção do usuário recebe um token JWT que deve ser enviado nas requisições posteriores para o acesso a áreas privadas. As Figuras 4 e 3 exibem a tela de login e cadastro respectivamente.

2.3.2 Tela de cadastro de dispositivos

Após fazer login, o usuário terá acesso à plataforma. Na tela inicial, ele visualizará os dispositivos que o pertence. Se esta for a primeira vez que o usuário está usando a plataforma a lista estará vazia. Para cadastrar um dispositivo o usuário precisará clicar na barra lateral para adicionar um novo dispositivo (opção *Add New Device*). Na tela de cadastro de dispositivos ele deverá inserir um nome para o dispositivo e aguardar as informações necessárias para concluir o registro do dispositivo. Caso o usuário insira um nome que já foi utilizado, o sistema retornará uma mensagem de erro informando que já existe um dispositivo com o mesmo nome. As Figuras 5 6 exibem o fluxo de cadastro de dispositivos.

2.3.3 Tela de dispositivos e dashboard

Para acessar a lista de dispositivos, o usuário deve clicar na opção *Devices* no menu lateral. A partir dessa ação uma lista com todos os dispositivos cadastrados pelo usuário serão exibidos na tela. Para visualizar a dashboard de um determinado dispositivo, basta clicar no dispositivo na lista de dispositivos. A Figura 9 exibe a tela da lista de dispositivos.

A tela de dashboard é a mais complexa do projeto, pois envolve várias requisições. Inicialmente, a plataforma requisita informações sobre o dispositivo ao backend. Em seguida, o backend responde com as configurações de grid cadastradas e os últimos dados cadastrados para cada campo do grid. Ao mesmo tempo é estabelecida uma conexão websocket com o backend para que os novos dados dos campos do grid possam fluir sem a necessidade de novas requisições.

Para realizar a renderização do grid, é realizado um mapeamento dos dados do dispositivo (requerida do backend), onde cada componente do grid é vinculado a um identificador para uma atualização conveniente. A Figura 10 exibe um exemplo de dashboard.

2.4 Implementação do código embarcado

2.4.1 Biblioteca data2board.h

A biblioteca data2board.h foi criada para que os dispositivos cadastrados no sistema se comunicam com ele por meio de uma API. A API criada nesse trabalho foi desenvolvida em C++ para o microcontrolador ESP32 da Espressif [16]. Para utiliza-la é necessário ter a disposição uma conexão WiFi com internet (ou acesso ao servidor, em casos de *localhost*). A API foi criada com o intuito de ser computacionalmente leve e transparente ao programador. As seções a seguir descrevem as funções disponíveis na API, e um exemplo de como utilizá-las em conjunto.

2.4.1.1 Função *setup*

A função de *setup* é responsável pela conexão Wi-Fi e pela criação da conexão com o servidor MQTT. No momento em que é realizada a chamada da função, os argumentos de nome da rede Wi-Fi e senha, endereço do servidor MQTT, nome do usuário e o token

do dispositivo (esses dois últimos são referentes ao que está cadastrado na plataforma online) são necessários para se conectar a plataforma.

```
// Funcao que configura o acesso ao servidor
void setup(
               const char *SSID, const char *PASSWORD,
               const char *MQTT_SERVER, String d2b_user,
               String d2b_device_token
            );
```

2.4.1.2 Função *connected*

A função *connected* foi dedicada a verificar o estado de conexão com a rede Wi-Fi, e o estado de conexão com o servidor MQTT. Caso o sistema esteja conectado à rede e com acesso ao servidor, a função retornará *true*, caso contrário retornará *false*. O código abaixo descreve o protótipo da função *connected*.

```
// Funcao de connected que retorna se o sistema esta conectado
boolean D2B::connected()
```

2.4.1.3 Função *reconnect*

A utilização desta função é dedicada a realizar a conexão e reconexão ao servidor MQTT. Abaixo temos a definição do protótipo da função *reconnect*.

```
// Funcao de reconnect que conecta e reconecta o sistema ao broker
void D2B::reconnect()
```

2.4.1.4 Função *loop_d2b*

A função *loop_d2b* realiza a execução das rotinas do protocolo MQTT, funções relacionadas a manutenção da conexão com o MQTT como gerenciamento de filas, etc. A função retorna *true* caso os procedimentos sejam bem sucedidos, e *false* caso contrário.

```
//Funcao de manutencao da conexao com o MQTT
boolean D2B::loop_d2b()
```

2.4.1.5 Função *insert_on_grid*

A função *insert_on_grid* é dedicada a configurar a dashboard. Para utilizá-la, basta passar como parâmetros um identificador único para o campo (esse identifica-

dor é importante para o usuário saber para que campo do grid ele está mandando o dado), a posição do campo no grid (linha e coluna), o tipo de campo que será publicado (D2B_CHART, D2B_CARD ou D2B_SWITCH) e uma descrição que aparecerá no topo do campo.

```
//Rotina de configuracao do grid no servidor MQTT
void D2B::insert_on_grid(
      int id, int line, int column,
      String type, String description
)
```

A implementação da função consiste em inserir os novos campos do grid em um objeto privado a classe do D2B, onde é implementada uma lista de campos. Caso o usuário tente criar dois campos com o mesmo ID apenas a ultima tentativa será a registrada na classe (os valores serão substituídos).

2.4.1.6 Função *grid_publish*

A função é responsável por enviar o *grid* para o servidor do Data2Board.

```
void D2B::grid_publish();
```

A ação executada na chamada da função tem como resultante no sistema a conversão da lista de campos em um objeto JSON que é publicado no tópico "*nome_de_usuário/token_do_dispositivo/grid*" do broker com o conteúdo seguindo o padrão do código abaixo. Cada campo do *grid* tem suas informações enviadas como um objeto no array *positions*.

```
{
  positions: [
    {
        description: <str>,
        field_type: <field_type>,
        field_id: <id>,
        x_pos: <int>,
        y_pos: <int>,
    }
  ]
}
```

2.4.1.7 Função *card_publish*

A função *card_publish* faz a atualização do valor de um campo do tipo *card* no *dashboard*. Para publicar, basta passar os parâmetros do identificador do *card*, o valor a ser enviado, e pode-se enviar a unidade do dado. Existem duas funções com os seguintes protótipos:

```
// Rotina que envia dados de card com unity
boolean card_publish(int IdMessage, String value, String unity);
```

```
// Rotina que envia dados de card sem unity
boolean card_publish(int IdMessage, String value);
```

A rotina retorna *true* se a informação foi enviada ao servidor, e *false* caso algum erro aconteça. O procedimento da função envia um JSON no formato descrito no código abaixo para o tópico "*nome_do_usuario/token_do_dispositivo/data/IdMessage*". O JSON é preenchido com o valor (no campo *value* e a unidade de medida (*unity*) que será enviado e atribuído ao respectivo card (definido no IdMessage do tópico de envio), e então exibido na dashboard.

```
{
  data:{
    value: <String>,
    unity: <String>
  }
}
```

2.4.1.8 Função *chart_publish*

A rotina *chart_publish* faz a publicação de um ponto num gráfico presente na *dashboard*. Para publicar, basta passar os parâmetros do identificador do gráfico onde a mensagem deve ser publicada e os valores de x e y no gráfico.

```
// Rotina que publica dados num chart
boolean chart_publish(int IdMessage, String x, String y);
```

A rotina retorna *true* se a informação foi enviada ao servidor, e *false* caso algum erro aconteça. A função envia um JSON no formato descrito no código abaixo para o tópico "*nome_do_usuario/token_do_dispositivo/data/IdMessage*". O JSON tem os valores de x e y preenchidos com os valores que foram passados para a função, e então

enviados para o respectivo gráfico definido pelo *IdMessage* (do tópico MQTT) onde por fim um ponto com coordenadas (x, y) é adicionado ao gráfico da dashboard.

```
{
  data:{
    x: <String>,
    y: <String>
  }
}
```

2.4.1.9 Função *set_callback*

A função *set_callback* é responsável por fazer o cadastro da rotina que será executada quando uma mensagem é recebida advinda da dashboard. Para cadastrar a função de *callback*, basta passar a função de que deve ser chamada como parâmetro. A Seção 2.4.1.10 dá um exemplo de como fazer uma função de *callback*.

```
// Rotina que cadastra o callback de recebimento de mensagens do MQTT
PubSubClient& D2B::setCallback(
                                  std::function<void(
                                                      char*,
                                                      uint8_t*,
                                                      unsigned int
                                                    )> callback
                                )
```

Uma observação é que a função é ativada quando um dado é publicado no tópico "*nome_do_usuario/token_do_dispositivo/rcv*", e o valor *payload* que é passado para a função de *callback* é um ponteiro de char que corresponde ao um JSON no formato abaixo, onde o campo *id* diz respeito ao campo que teve o valor alterado na dashboard, e o *value* é o novo valor.

```
{
  id: <String>,
  value: <String>
}
```

2.4.1.10 Criação da função de *callback*

A função de *callback* é sempre chamada quando uma mensagem chega ao dispositivo (desde que seja previamente cadastrada por meio da função *set_ callback*). Existem três campos que são preenchidos: o *topic*, que retorna o tópico que foi escrito, o *payload* da mensagem onde temos o id e o valor do objeto o qual foi modificado na dashboard, e por fim *length* que é o tamanho do *payload*. O código a seguir demonstra uma declaração de função de *callback*.

```
void callback(char *topic, byte *payload, unsigned int length){
  // Transforma o conteudo da mensagem em JSON
  deserializeJson(doc_json, payload);
  JsonObject obj = doc_json.as<JsonObject>();

  // Separa os dois campos do JSON recebido
  String id = obj[String("id")];
  String value = obj[String("value")];

  // Atualiza o estado do pino do LED
  if (id == String(LED_SWITCH_ID))
  {
    led_status = std::stoi(value.c_str());
    digitalWrite(PIN_LED, led_status);
  }
}
```

O código abaixo demonstra como cadastrar uma função de *callback* (como a do código acima) que será acionada uma mensagem for publicada.

```
// Definindo o callback para a recepcao de mensagens
d2b.setCallback(callback);
```

2.4.2 Bibliotecas utilizadas na construção do data2board.h

No desenvolvimento da biblioteca foram prioritariamente utilizadas duas bibliotecas, a PubSubClient [17] e a ArduinoJson [18], além de bibliotecas padrão do ambiente de desenvolvimento de código para ESP32. A PubSubClient foi utilizada no projeto para realizar a comunicação com o broker MQTT, e a biblioteca ArduinoJson foi utilizada para facilitar o uso de estruturas JSON em código C++.

3 Resultados

Como resultado da proposta de criação da plataforma temos o exemplo a seguir que demonstra o suporte que a plataforma pode dar ao usuário.

3.1 Exemplo de uso do sistema

3.1.1 Cadastro e Login

O cadastro na plataforma é realizada na tela de cadastro disponível no site da plataforma. Basta preencher os dados e clicar em cadastrar. Logo após o cadastro clicar em logar, como nas Figuras 3 e 4

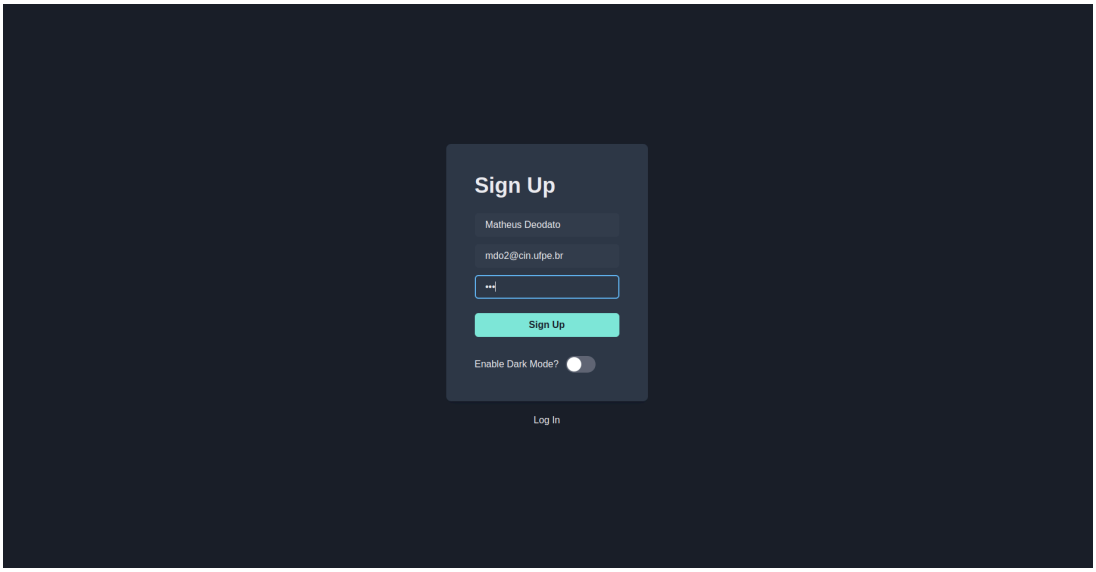

Figura 3 – Tela de cadastro

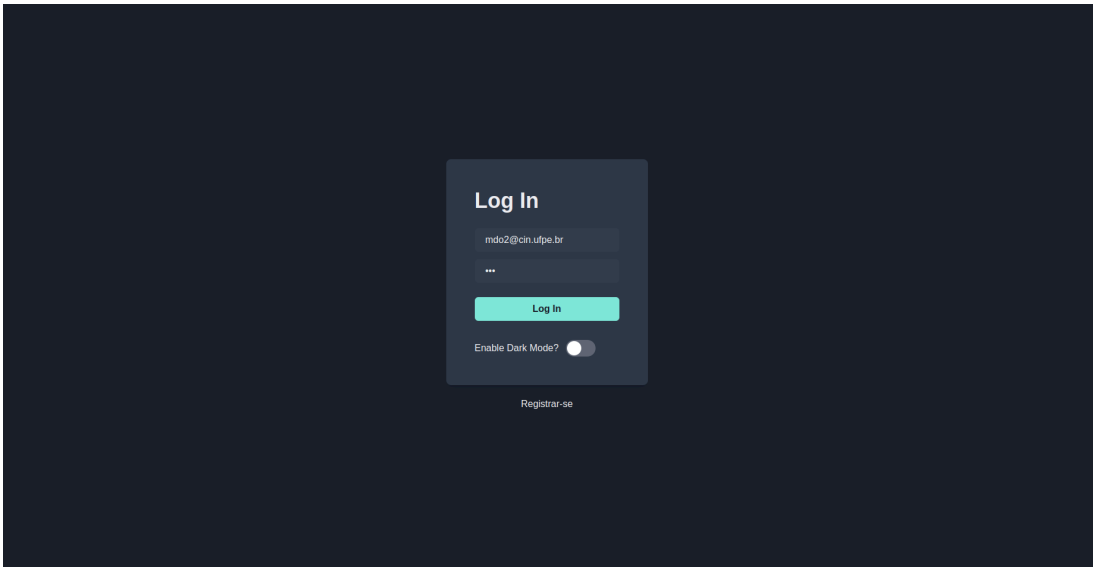

Figura 4 – Tela de login

3.1.2 Cadastro do dispositivo

Após fazer o login e acessar a tela inicial, clique em "Add New Device" na barra lateral. Em seguida, atribua um nome ao dispositivo, conforme exemplificado na Figura 5, e finalize o cadastro. Em seguida, será exibida uma tela com os campos necessários para preencher os dados do dispositivo, conforme mostrado na Figura 6.

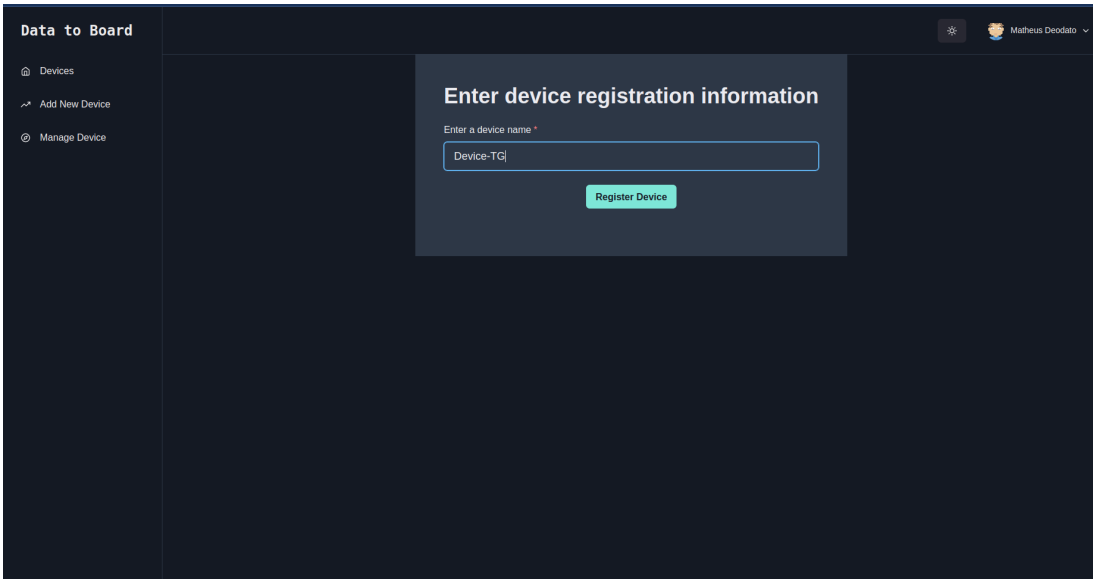

Figura 5 – Tela de cadastro de dispositivos

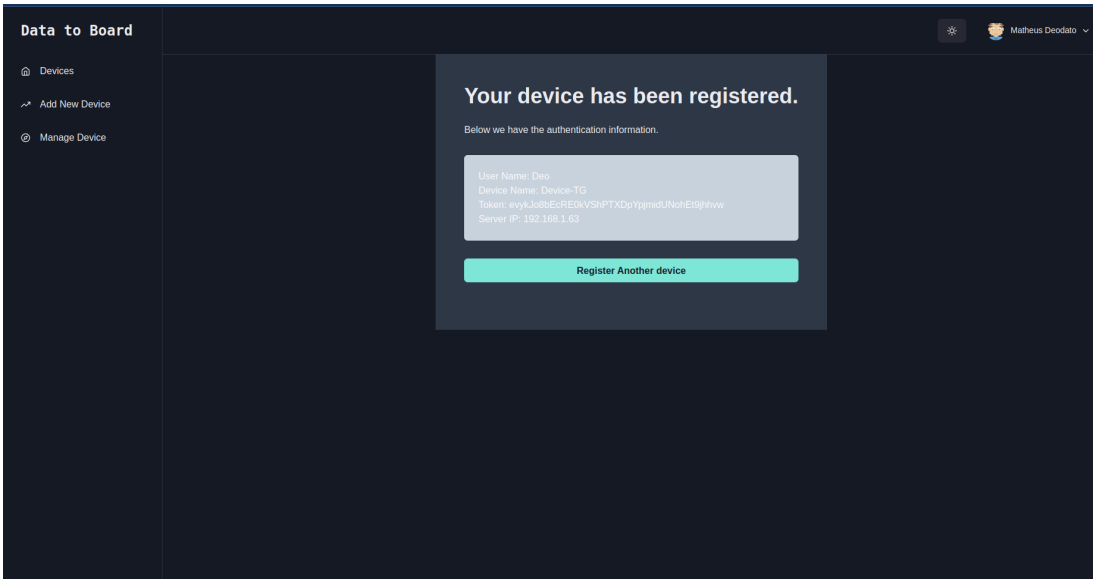

Figura 6 – Tela informações do dispositivo cadastrado

A próxima seção descreve um exemplo de código embarcado para criar uma dashboard, e na seção 3.1.4 temos o resultado da criação da dashboard.

3.1.3 Código embarcado

O objetivo desse tópico é mostrar um exemplo de código real que utiliza o sistema. Na execução foi utilizado uma ESP-WROOM-32 (que é o microcontrolador do exemplo, o qual consegue realizar as medidas necessárias no circuito e se conectar a internet), um LED, um botão, um potênciometro linear de 10KΩ, dois resistores de 1KΩ, um para pull-down do botão, e outro para limitar a corrente do LED. O diagrama elétrico do circuito do exemplo está na Figura 7, onde os valores PIN 5, 32 e 35 são pinos de GPIO da ESP-WROOM-32.

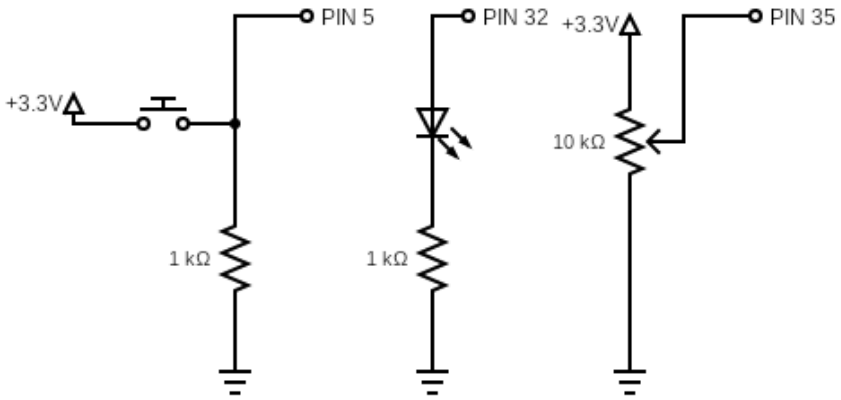

Figura 7 – Diagrama do Circuito do exemplo

O sistema foi montado conforme o diagrama, e o exemplo foi reproduzido no protótipo mostrado na Figura 8.

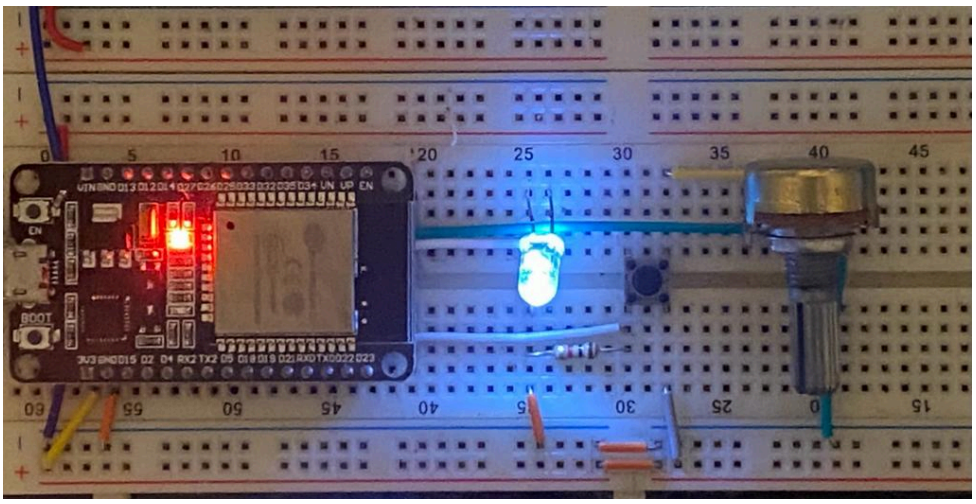

Figura 8 – Circuito do exemplo

A ferramenta utilizada para desenvolvimento do software, debug e gravação do firmware na placa foi o VSCode com a extensão do PlatformIO. O código utilizado foi subdividido em etapas para melhor entendimento:

3.1.3.1 Inclusão das bibliotecas

Como especificado no código abaixo, as bibliotecas utilizadas foram Arduino.h para manter a compatibilidade com a IDE e o framework do Arduíno, data2board.h que faz a integração com o servidor, e o ArduinoJson.h que se propõe a decodificar mensagens em formato JSON.

```
1 #include <Arduino.h>
2 #include <data2board.h>
3 #include <ArduinoJson.h>
```

3.1.3.2 Parâmetros de configuração de rede e de conexão com o servidor

Abaixo temos as configurações de conexão do dispositivo com a rede Wi-Fi e com o servidor do Data2Board (dados de conexão com o servidor que foram disponibilizados no cadastro do dispositivo).

```
1 #define SSID "NOME_DA_REDE_WIFI"
2 #define PASSWORD "SENHA_DA_REDE"
3 #define MQTT_SERVER "IP_DO_SERVIDOR"
```

```
#define D2B_USER "USUARIO"
#define D2B_DEVICE_TOKEN "TOKEN_DO_DEVICE"
```

3.1.3.3 Declaração dos pinos utilizados e dos IDs dos campos do grid da dashboard

A seguir temos o código que simplemente dá nome aos nossos pinos físicos da ESP32 e os identificadores (IDs) aos nossos campos na dashboard.

```
// Mapeamento dos pinos da ESP32
#define PIN_LED 32
#define PIN_BUTTON 5
#define PIN_POTENCIOMETER 35
// Mapeamento dos IDs dos campos do grid na dashboard
#define VOLTAGE_CARD_ID 0
#define VOLTAGE_CHART_ID 1
#define BUTTON_CARD_ID 2
#define LED_CARD_ID 3
#define LED_SWITCH_ID 4
```

3.1.3.4 Variáveis auxiliares

As variáveis a seguir são componentes de controle para o funcionamento do exemplo. A variável *d2b* é utilizada para interagir com o código da biblioteca de comunicação com a plataforma, a *doc_json* foi criada para salvar e decodificar as mensagens recebidas pelo dispositivo, o *led_status* salva o estado do LED (1 é equivalente a ligado e 0 a desligado), por fim uma variável que é usada na medição do tempo de envio dos dados para o servidor (no código foi definido para 10 segundos).

```
// Instancia do Data2Board
D2B d2b;
// Variavel que decodifica o JSON retornado pelo Data2Board
DynamicJsonDocument doc_json(1024);
// Variavel de controle do status do LED
int led_status;
// Variavel de controle do tempo de envio da ultima mensagem
unsigned long time_of_last_message;
```

3.1.3.5 Função de callback

Neste ponto temos a declaração da função que será chamada quando uma mensagem for enviada do servidor para o dispositivo. A função decodifica a mensagem, verifica se o ID da mensagem (no caso esse ID se refere ao identificador do campo que gerou a mensagem) é o do LED_SWITCH_ID (o botão que aparece na dashboard), caso seja o valor que foi recebido na mensagem será atribuído a variável *led_status* e o nível lógico do pino do LED será atualizado para o valor recebido.

```
void callback(char *topic, byte *payload, unsigned int length){
  // Transforma o conteudo da mensagem em JSON
  deserializeJson(doc_json, payload);
  JsonObject obj = doc_json.as<JsonObject>();

  // Separa os dois campos do JSON recebido
  String id = obj[String("id")];
  String value = obj[String("value")];

  // Atualiza o estado do pino do LED
  if (id == String(LED_SWITCH_ID))
  {
    led_status = std::stoi(value.c_str());
    digitalWrite(PIN_LED, led_status);
  }
}
```

3.1.3.6 Função de setup

A função de setup é padrão em códigos que utilizam o framework do Arduíno. Aqui são declaradas informações de configuração do sistema. As declarações são: definição de pinos de entrada/saída, configuração da instância do d2b, configuração do grid, e cadastro da função de callback definida no passo anterior a qual será chamada quando uma mensagem for recebida.

```
void setup(){
  // Configura o tipo da porta (INPUT/OUTPUT)
  pinMode(PIN_LED, OUTPUT);
  pinMode(PIN_BUTTON, INPUT);
  pinMode(PIN_POTENCIOMETER, INPUT);
```

```
  // Configura a instancia do d2b
  d2b.setup(SSID, PASSWORD, MQTT_SERVER, D2B_USER, D2B_DEVICE_TOKEN);
  // Configura os elementos do grid
  d2b.insert_on_grid(VOLTAGE_CARD_ID, 0, 0, D2B_CARD, "Tensao");
  d2b.insert_on_grid(LED_CARD_ID, 0, 1, D2B_CARD, "LED");
  d2b.insert_on_grid(BUTTON_CARD_ID, 0, 2, D2B_CARD, "Botao");
  d2b.insert_on_grid(VOLTAGE_CHART_ID, 1, 0, D2B_CHART,
                       "Historico da tensao"
                     );
  d2b.insert_on_grid(LED_SWITCH_ID, 1, 1, D2B_SWITCH, "LED SWITCH");
  // Define a funcao de callback que sera chamada quando uma mensagem
   for recebida
  d2b.setCallback(callback);
}
```

3.1.3.7 Função de loop

Por fim, temos a função loop. Nesta função, realizamos uma série de tarefas importantes. Primeiro, verificamos se o dispositivo está conectado. Se não estiver, tentamos reconectar. Em seguida, executamos as tarefas do MQTT, incluindo a verificação de filas de mensagens e a gestão de QoS. Por último, temos um script que é executado a cada 10 segundos. Neste script, realizamos a amostragem dos dados e enviamos esses dados para o servidor e sem bloquear o código. Esta última etapa é particularmente importante para garantir o bom desempenho do sistema.

```
void loop(){
  if (!d2b.connected()){// Verifica se o dispositivo esta conectado
    d2b.reconnect();    // Se nao estiver tenta reconectar e envia o
   grid
    d2b.grid_publish();
  }
  d2b.loop_d2b(); // Executa atividades do MQTT
  // Envia as mensagens a cada 10 segundos
  unsigned long time_atual = millis();
  if (time_atual - time_of_last_message > 10000){
    time_of_last_message = time_atual;
    // Le do valor do potenciometro e transforma o valor em volts
    float voltage = (analogRead(PIN_POTENCIOMETER) / 4095.0) * 3.3;
```

```
13      // Realiza a leitura do valor do botao
14      int button_status = digitalRead(PIN_BUTTON);
15      // Publicando informacoes de tensao
16      d2b.card_publish(VOLTAGE_CARD_ID, String(voltage), "V");
17      d2b.chart_publish(VOLTAGE_CHART_ID, String(time_atual), String(
        voltage));
18      // Publicando informacoes do botao
19      d2b.card_publish(BUTTON_CARD_ID, button_status == HIGH ? "Ativado" :
        "Desativado");
20      // Publicando informacoes do LED
21      d2b.card_publish(LED_CARD_ID, led_status == HIGH ? "Ligado" : "
        Desligado");
22    }
23  }
```

3.1.4 Resultado do dashboard

Para ver o resultado do exemplo acima basta clicar na aba *Devices* como na Figura 9, onde existe uma lista com todos os dispositivos pertencentes ao usuário, e clicar no dispositivo em que se quer ver a dashboard, exemplificado na Figura 10.

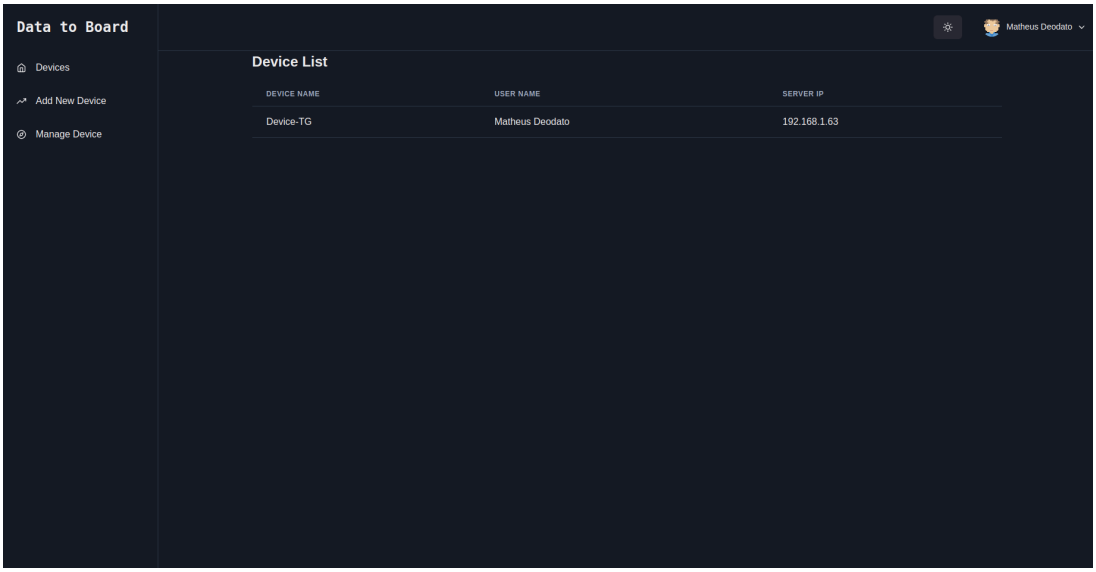

Figura 9 – Lista dos dispositivos

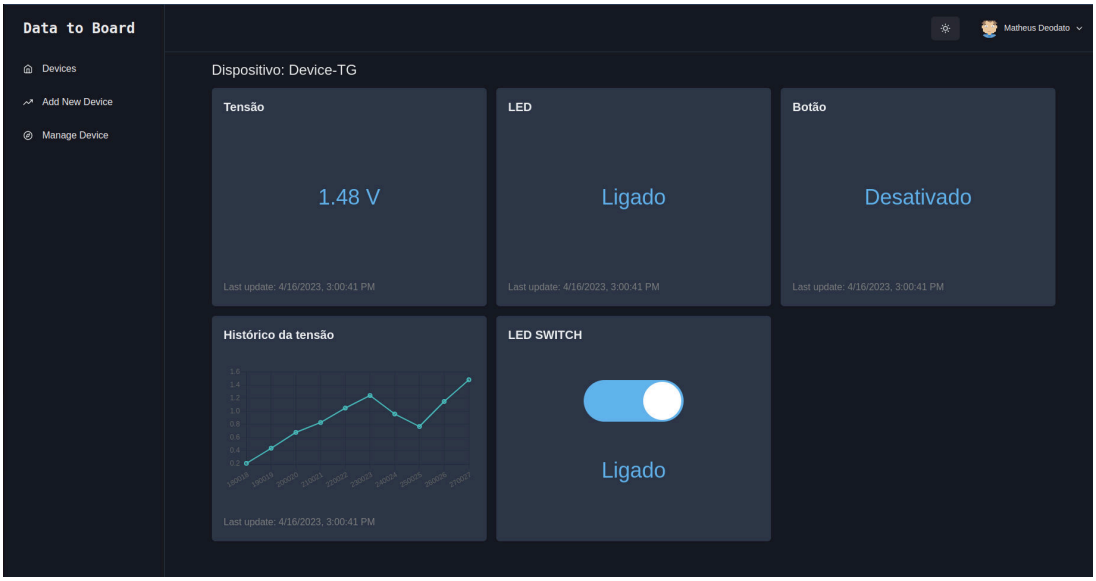

Figura 10 – Dasboard do dispositivo

Como esperado temos a exibição dos dados criados no código embarcado disponíveis na plataforma. Um link para a demonstração em vídeo do exemplo se encontra nas referências [19].

Como resultado do projeto, podemos afirmar que a quantidade de código implementado foi relativamente baixa, pouco mais de 3000 linhas. Isso só foi possível graças à utilização de várias ferramentas já disponíveis na comunidade. No entanto, a principal dificuldade do projeto consistiu em selecionar e integrar os módulos corretos, o que demandou um bom tempo de pesquisa e testes para assegurar o bom funcionamento do sistema.

Além disso, foi necessário ter conhecimento em diversos frameworks e módulos, entendendo suas particularidades e maneiras de desenvolver software com boas práticas em cada um deles. Esse processo exigiu atenção e dedicação para garantir que todas as partes do sistema estivessem funcionando em harmonia.

Dessa forma, podemos concluir que a implementação de um projeto como esse requer habilidades técnicas e um bom conhecimento das ferramentas disponíveis na comunidade. É preciso estar disposto a enfrentar desafios e a dedicar tempo para garantir que tudo funcione corretamente. Com o planejamento e a execução adequados, é possível criar um sistema eficiente e funcional com uma quantidade relativamente baixa de código.

4 Conclusões e Trabalhos Futuros

Como conclusão do trabalho, é possível afirmar que a plataforma desenvolvida atingiu plenamente os objetivos propostos, acelerando a criação de protótipos de dispositivos IoT e simplificando a implementação da conexão de dados entre esses dispositivos e a interface de visualização.

No entanto, ao longo do desenvolvimento da solução, foram identificadas algumas dificuldades em relação à integração de uma segurança mais robusta na comunicação entre os dispositivos e o broker MQTT. Em particular, foi necessário optar entre a segurança e a disponibilidade do serviço, uma vez que não foi possível inserir novas chaves SSL ao Mosquitto em tempo de execução. Essa limitação foi uma das principais motivações para o problema encontrado.

Apesar desses desafios, o projeto foi elaborado com foco em arquiteturas escaláveis e flexíveis, permitindo futuras modificações e melhorias nos módulos do sistema. Além disso, o sistema é altamente confiável e pode ser instalado em nuvem, sendo possível ainda modificar seu código fonte para implementar novas funcionalidades.

Outro ponto importante é a possibilidade de implementar um certificado HTTPS nas conexões entre backend e frontend, tornando o sistema ainda mais seguro. Com todos esses recursos, a plataforma é uma solução viável e eficaz para acelerar a criação de protótipos de dispositivos IoT, contribuindo para a popularização e democratização do uso dessa tecnologia.

Como possíveis trabalhos futuros, destacam-se as otimizações na experiência do usuário e na interface gráfica da plataforma, bem como o aprimoramento do gerenciamento e visualização dos dados dos dispositivos. É importante destacar a possibilidade de implementação de novas funcionalidades, como a capacidade de dar zoom nos gráficos e selecionar intervalos de tempo para uma visualização mais precisa dos dados, o que poderia tornar a plataforma ainda mais útil e eficiente para seus usuários.

Os códigos do projeto se encontram nos links: frontend [20], backend [21], biblioteca da ESP32 [22] e inicialização do Mosquitto [23].

Referências

[1] UBIDOTS. *Ubidots*. 2021. https://ubidots.com/.

[2] BLYNK. *Blynk*. 2021. https://blynk.io/.

[3] INDUSTRIES, A. *Adafruit IO*. 2021. https://io.adafruit.com/.

[4] ECLIPSE Mosquitto. https://mosquitto.org/. Acesso em: 16 abril 2023.

[5] NODE.JS. https://nodejs.org/. Acesso em: 16 abril 2023.

[6] JWT.IO. https://jwt.io/. Acesso em: 16 abril 2023.

[7] EXPRESS. https://expressjs.com/. Acesso em: 16 abril 2023.

[8] MONGODB. https://www.mongodb.com/. Acesso em: 16 abril 2023.

[9] TEAM, M. *Mongoose*. 2021. https://mongoosejs.com/. Acesso em: 16 abril 2023.

[10] TEAM, M. *MQTT.js*. 2021. https://github.com/mqttjs/MQTT.js. Acesso em: 16 abril 2023.

[11] SOCKET.IO. https://socket.io/. Acesso em: 16 abril 2023.

[12] REACT. https://reactjs.org/. Acesso em: 16 abril 2023.

[13] CHAKRA UI. https://chakra-ui.com/. Acesso em: 16 abril 2023.

[14] CHART.JS. https://www.chartjs.org/. Acesso em: 16 abril 2023.

[15] AXIOS. https://axios-http.com/. Acesso em: 16 abril 2023.

[16] ESPRESSIF. *ESP32*. https://www.espressif.com/en/products/modules/esp32. Acesso em: 16 abril 2023.

[17] O'LEARY, N. *PubSubClient*. https://pubsubclient.knolleary.net/. Acesso em: 16 abril 2023.

[18] BLANCHON, B. *ArduinoJson*. https://arduinojson.org/. Acesso em: 16 abril 2023.

[19] DEODATO, M. *Demonstração do Data2Board*. https://youtu.be/4OEjKWf_z8w. Acesso em: 21 abril 2023.

[20] DEODATO, M. *Data2Board-Frontend*. https://github.com/deodatomatheus/Data2Board-Frontend. Acesso em: 16 abril 2023.

[21] DEODATO, M. *Data2Board-Backend*. https://github.com/deodatomatheus/Data2Board-Backend. Acesso em: 16 abril 2023.

[22] DEODATO, M. *Data2Board-ESP32*. https://github.com/deodatomatheus/data2board-esp32. Acesso em: 16 abril 2023.

[23] DEODATO, M. *Data2Board-Mosquitto*. https://github.com/deodatomatheus/data2board-mosquitto. Acesso em: 16 abril 2023.