

Sistema de Monitoramento da Orientação do Celular com APP em Flutter e ESP32

André G. G. Pinto

Abstract—O artigo descreve um projeto de um sistema de monitoramento simples utilizando o aplicativo em Flutter, ESP 32 e uma API *RESTful* feita em Express.js. No final é apresentado o resultado do sistema com figuras e discussão do resultado.

Index Terms—Dart, C, HTTP, API RESTful, Node.js .

I. INTRODUÇÃO

OS Sistemas de monitoramento contribuem significativamente para se ter um melhor ciência do que está acontecendo em um determinado lugar. Tudo isso é possível graças a um sistema de monitoramento que capta essas informações através de sensores e em tempo real mostra os dados para que possam ser analisados à longas distâncias.

Apesar dos benefícios desses sistemas de monitoramento, é preciso se atentar na organização no desenvolvimento. A construção destes sistemas deve ser feita de maneira a possibilitar a manutibilidade, escalabilidade e estabilidade.

Portanto, o objetivo deste projeto é desenvolver um sistema de monitoramento da orientação do *smartphone*, com o aplicativo em Flutter e ESP 32, e assim, exibir a orientação no *display LCD*.

As seguintes seções serão estruturadas como segue: 2 descreve os conceitos utilizados, já na seção 3 serão mostrados os materiais e métodos. Na seção 4 as discussões e análises do que foi feito, finalizando com a 5, resultados e 6 conclusão do trabalho.

II. A TEORIA

Esta seção tem por objetivo apresentar os conceitos basilares para realizar o sistema de monitoramento.

A. API RESTful

O REST (REpresentational State Transfer) define um conjunto de princípios de arquitetura para o desenvolvimento de *Web services*. Este modelo segue alguns princípios básicos: i) ser *stateless*; ii) uso explícito de métodos HTTP; iii) expor URIs(Uniform Resource Identifier) semelhantes à estrutura de diretório, entre outros [1].

1) *Ser stateless*: significa que cada requisição é completa e independente e não necessita de um servidor enquanto a requisição é processada. No *header* do HTTP e corpo de cada requisição contém todas os parâmetros, contextos e dados necessários para gerar uma resposta de maneira independente [1].

2) *Uso explícito de Métodos HTTP*: na prática com o uso de métodos HTTP cria-se um mapeamento entre operações CRUD (*create, read, update e delete*) e permite o compartilhamento de recursos. Recursos são a abstração da informação, ou seja, qualquer informação que pode ser nomeada, como um documento ou imagem [1].

3) *Expor URIs*: O identificador uniforme de recurso é uma cadeia de caracteres usada para identificar um recurso na internet. Também pode ser conhecido como uma rota, um caminho que pode ser composto de forma hierárquica, assim permitir o envio e o consumo de recursos de uma API (Application Programming Interface) [1].

B. Framework Flutter

O Flutter é um SDK(Software Development Kit) *open source* para desenvolvimento de aplicativos móveis para Android e IOS feito pelo Google [2].

Em Flutter tudo pode ser um *Widget*. O *Widget* serve para o controle da UI(User Interface), responsável por definir o Design Visual e o Design de Interação no aplicativo. O aplicativo é todo construído de um conjunto de *Widgets* do tipo *Stateful* (Com estado) e *Stateless* (Sem estado) [3].

C. ESP 32

O ESP 32 é um microcontrolador de baixo custo extremamente rápido e com muitos componentes integrados. Existem variações do ESP 32 com processador Dual Core de 32 bits e 240 MHz de frequência, com o WI-FI e *Bluetooth* integrado. Entre os componentes inclusos, os principais são: Amplificador de Potência, Amplificador de recepção de baixo ruído, Interruptor de antena e filtros, entre outros [4].

III. MATERIAS E MÉTODOS

Esta seção apresenta os componentes e a metodologia utilizada para o desenvolvimento deste artigo como descrito a seguir.

A. Materiais Utilizados

Nome	Tipo	Quantidade
ESP32	Microcontrolador	1
NodeMCU		
Cabo Jumper	Macho-Fêmea	4
Protoboard	400 Pontos	1
Display LCD 16x2	Backlight Azul	1
Modulo para LCD	Serial I2C	1
Protoboard (opcional)	760 Pontos	1

B. Arquitetura do sistema

O sistema proposto é baseado na integração do Aplicativo desenvolvido em Flutter e o microcontrolador ESP32, por meio de uma *API RESTful* em *Nodejs*. O foco principal é apresentar as informações de orientação no *display LCD* de forma automatizada.

O aplicativo em flutter é responsável pela coleta de informações dos sensores, esses sendo o acelerômetro e o giroscópio. A partir delas, a biblioteca *Native Device Orientation* é encarregada de determinar a orientação do dispositivo [5].

Com os dados dos sensores, o aplicativo determina orientação do celular e envia-a para a API a cada 1 segundo. A API recebe a orientação e gravá-a em uma variável.

O microcontrolador ESP 32, por sua vez, é encarregado pela obtenção da orientação do *smartphone* registrada na API. A busca é realizada a cada 4 segundos, e com a resposta recebida é então exibida na tela LCD. O diagrama do processamento realizado por todas as partes do sistema pode ser observado na Fig. 1.



Fig. 1. Arquitetura do sistema de monitoramento da orientação do celular.

1) *Microcontrolador ESP32*: O microcontrolador se comunica com o *Display LCD* por meio do módulo I2C com CI PCF8574. Este módulo faz com que a comunicação aconteça com apenas dois pinos, o pino SDA e o SCL no módulo I2C, conectados, respectivamente, nos pinos GPIO21 e GPIO22 no ESP32. Os outros pinos são o GND (acima do GPIO23) e VCC (a baixo do GPIO13). Os pinos podem ser observados na Fig. 2 marcados por uma borda de cor amarela.

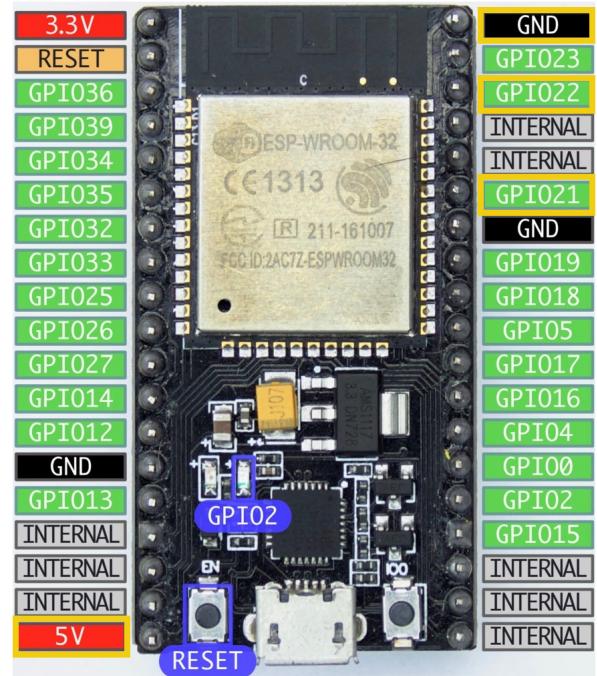


Fig. 2. Pinos do microcontrolador ESP32 NodeMCU. Imagem obtida do site Circuit Setup [6]

No inicio do código possui a inclusão das bibliotecas de WiFi, HTTPClient, ArduinoJson e LiquidCrystal_I2c antes da função Setup. O endereço do *Display LCD* pode variar em alguns modelos, sendo normalmente 0x27 ou 0x3F, e pode ser encontrado caso uma seja feita um *scanner* do módulo I2C. Essa parte do código pode ser encontrada na Listing 1.

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
//LDC
#include <LiquidCrystal_I2C.h>
// Seta o numero de colunas e linhas do LCD
int lcdColumns = 16;
int lcdRows = 2;
// Seta o endereco, numero de colunas e linhas
LiquidCrystal_I2C lcd(0x3F, lcdColumns, lcdRows);
// Fornece as credenciais da rede WIFI
const char* ssid = "WIFI NAME";
const char* password = "PASSWORD";
// String para guardar a resposta da API
String response = "";
// Documento JSON que aloca espaco dinamicamente
DynamicJsonDocument doc(2048);
```

Listing 1. Código inicial. Contém as inclusões das bibliotecas necessárias para o funcionamento do programa.

Na rotina setup o LCD é inicializado e o *backlight* ligado. Além disso é feita a conexão com a rede WIFI, cuja as informações foram dadas anteriormente. Em caso de sucesso o endereço IP do ESP32 é mostrado na IDE. O código pode ser visto no Listing 2.

```
void setup(void) {
  // Inicializa o LCD
  lcd.init();
  // Liga o a luz de fundo
  lcd.backlight();
  //Comeca uma comunicacao serial no monitor(IDE)
  Serial.begin(115200);
```

```

9 // Inicializa a conexao WIFI
10 WiFi.mode(WIFI_STA);
11 // Passa o nome e a senha do WIFI
12 WiFi.begin(ssid, password);
13 Serial.println(";");
14 // Espera pela conexao
15 while (WiFi.status() != WL_CONNECTED) {
16     delay(500);
17     Serial.print(".");
18 }
19 Serial.print("WiFi connected with IP: ");
20 Serial.println(WiFi.localIP());
21 }
```

Listing 2. Código de inicialização no ESP32.

O loop no Listing 3 é uma rotina que se repete até que haja alguma interrupção externa ou alguma condição de saída. O começo da conexão começa na linha 8 em seguida os headers adicionados permitem que a API *RESTfull* tenha informações necessárias para permitir o acesso do ESP 32. Da linha 13 à 24 é realizado a requisição GET para obter o recurso da API, em caso houver algum erro, este é exibido no monitor serial. Finalmente, o restante do código se remete a impressão das informações no *Display LCD* e no monitor serial.

```

1 void loop(void) {
2     // Cria a variavel para conexao HTTP
3     HTTPClient http;
4     // A URL da API
5     String request = "http://192.168.1.4:8085/accmeter";
6     // Inicia a requisicao para a API
7     http.begin(request);
8     // Adiciona Headers com informacoes necessarias
9     // para a conexao
10    http.addHeader("Content-Type", "application/json");
11    ;
12    http.addHeader("Accept", "application/json");
13    // Escolhe o metodo GET para fazer a requisicao
14    http.GET();
15    // Recebe a resposta da requisicao GET
16    response = http.getString();
17    // Imprime a resposta no monitor serial
18    Serial.println(response);
19    // Separa as variaveis do JSON, caso nao houver
20    // erros
21    DeserializationError error = deserializeJson(doc,
22        response);
23    if(error) {
24        Serial.print(F("deserializeJson() failed: "));
25        Serial.println(error.f_str());
26        return;
27    }
28    // Seta o cursor para a primeira posicao
29    // Primeira linha
30    lcd.setCursor(0, 0);
31    // Imprime a orientacao na tela LCD
32    lcd.print(doc["title"].as<char*>());
33    delay(1000);
34    // Imprime a orientacao no monitor serial
35    Serial.println(doc["title"].as<char*>());
36    // Termina a conexao HTTP
37    http.end();
38 }
```

Listing 3. Código principal que permite a requisição GET e a apresentação da orientação do celular na tela LCD

2) *Aplicativo em Flutter*: O aplicativo utiliza dos sensores de acelerômetro e giroscópio. As bibliotecas *Native Device Orientation* e *HTTP* são utilizadas para o desenvolvimento

do projeto [7]. O aplicativo se comunica com a API através da comunicação HTTP em função de algum movimento no *smartphone*. A lista de dependências do projeto está na Fig. 3.

```

dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.2
  native_device_orientation: ^1.0.0
  http: ^0.12.0+2
```

Fig. 3. Dependências do aplicativo em Flutter. As dependências são colocadas no arquivo "pubspec.yaml".

Funcionalmente o Flutter lida com estados, que são mudanças que podem ocorrer nos objetos ou *widgets* durante a execução da aplicação. Com isso, para lidar com mudanças de estados nos objetos é necessário criar um *StatefulWidget*, que significa **Widget com Estado**. Este *widget* lida com as classes de estado. O código referente esta classe está na Fig. 4.

```

class HomePage extends StatefulWidget {
  @override
  State<HomePage> createState() {
    // O StatefulWidget pede um estado (State).
    // Então, é necessário criar uma outra classe
    // para que possa ser retornada e assim
    // Funcionar como um estado que vai ser sempre
    // modificado.

    // Funciona somente para esta classe
    return HomePageState();
  }
}
```

Fig. 4. Função para criar estados. No retorno espera-se um estado do tipo *HomePage*. No final da função possui o retorno de um estado que é criado na próxima classe.

O parte principal da aplicação é na coleta e determinação da orientação. Esta é realizada por meio do *widget NativeDeviceOrientationReader* que vai ler os sensores e passar para o **builder** o **content** que contém informações do contexto. Então, usando o método **orientation** e o **content** obtém-se um objeto contendo informações de orientação.

A função **createCoord** é responsável por enviar esses dados à API *RESTful*. Essa seção é crucial, o código inteiro é síncrono, mas nessa parte espera-se um comportamento assíncrono. Com isso, deve-se realizar a chamada desta função e deixar ela rodando em segundo plano enquanto outros processos continuam. Isso é possível usando o método **then** que trata o retorno da função no retorno da requisição. Este código pode ser visto na Fig. 5.

```

body: NativeDeviceOrientationReader(
  builder: (content) {
    //Permite controlar quando o dispositivo começa a ouvir
    //por mudanças na orientação
    final orientation =
      NativeDeviceOrientationReader.orientation(content);
    //Tradução, abstração da biblioteca
    String oriStr = orientation.toString();
    oriStr = orientationPt[oriStr].toString();
    createCoord(oriStr, '0.0', '0.0', '0.0')
      .then(
        (value) => print('Orientação Enviada (API): ${value.title}'))
      .catchError((err) => print('Caught error: $err'));
  }
)

```

Fig. 5. A função `createCoord` retorna o response da API de maneira assíncrona. O uso do `then` na função permite que a aplicação rode sem que haja travamentos no processamento.

A criação de uma função assíncrona na Fig. 6 é necessária para a comunicação com a API RESTful. A função assíncrona permite que a requisição do método POST com o body e headers seja feita sem nenhum problema. A palavra chave `await` faz com que o processamento dentro da função pause até que a requisição seja concluída. O retorno esperado é um objeto *Future* do tipo **AccelerometerModel**.

```

import 'package:http/http.dart' as http;
import 'dart:convert';
import 'accelerometer_model.dart';

Future<AccelerometerModel> createCoord(
  String title, String x, String y, String z) async {
  const String apiURL = 'http://192.168.1.4:8085/accmeter';
  final response = await http.post(apiURL,
    body: json.encode({'x': x, 'y': y, 'z': z, 'title': title}),
    headers: {
      'Content-type': 'application/json',
      'Accept': 'application/json',
    });
  if (response.statusCode == 201 || response.statusCode == 200) {
    final String responseString = response.body;
    return accelerometerModelFromJson(responseString);
  } else {
    throw Exception('Failed to load Accelerometer values');
  }
}

```

Fig. 6. Função assíncrona para enviar a informação de orientação para a API RESTful. Os dados são convertidos para JSON e headers são adicionados antes do envio. Caso tenha sido bem sucedido a resposta é recebida com os dados enviados em formato JSON.

Finalmente, quando a resposta é retornada ela é exibida na tela do aplicativo com a orientação atual do *smartphone*.

3) *API RESTful*: A API desenvolvida em Express.js encarrega-se de escutar as requisições e tomar certas ações baseado na rota e métodos. A rota é apenas um endereço e os métodos determinam se haverá inclusão, edição, exclusão ou obtenção dos recursos da API.

O Express.js é iniciado e o CORS é incluído para permitir a conexão de qualquer dispositivo na API. A porta utilizada para a aplicação é 8085 e as rotas são incluídas no arquivo `messageRouter`. Esse código pode ser observado na Fig. 7.

```

const messageRouter = require('./routes/messageRoutes');
const app = express();
app.use(cors());
app.use(express.json());

const port = process.env.PORT || 8085;

app.use(messageRouter);

app.listen(port);
console.log("Server is running in port " + port);

```

Fig. 7. Arquivo `server.js` representa a parte principal da API responsável por ouvir pelas requisições.

A rota permite o acesso a alguma função, esta executa determinada ação programada caso seja chamada. No arquivo **messageRoutes.js** possui apenas a rota `/accmeter` com métodos GET e POST. A função com método GET retorna o objeto *coord* que possui a orientação. A função com o método POST exige o envio de informações e neste caso ela espera um objeto do tipo **coord** para salvá-lo. As rotas e métodos podem ser observadas na Fig. 8.

```

// getting information from the database
app.get('/accmeter', (req, res) => {
  console.log("Sending ... " + coord.title);
  const responseMsg = JSON.stringify(coord);
  res.send(responseMsg);
})

app.post('/accmeter', (req, res) => {
  let msg = req.body
  let {x, y, z, title} = msg;
  coord = {
    x,
    y,
    z,
    title
  }
  console.log("Receiving ... " + coord.title);
  const responseMsg = JSON.stringify(coord)
  res.send(responseMsg);
})

```

Fig. 8. Arquivo `messageRoutes.js` representa as rotas da API com seus devidos recursos.

IV. ANÁLISE E DISCUSSÕES

Esta seção apresenta os resultados do sistema de monitoramento da orientação proposto com o aplicativo em Flutter, ESP32 e o *Display LCD*. No final o texto contendo a orientação atual do dispositivo é exibida na tela periodicamente.

1) *ESP 32*: No inicio da montagem o ESP 32 apresentou alguns problemas, como o não reconhecimento da porta serial COM na IDE do Arduino e no momento da gravação do programa compilado. O reconhecimento da porta foi resolvido ao instalar o driver USB no site da Silicon Labs [8] e o

problema da gravação foi preciso reinstalar a IDE seguindo os passos do vídeo em [9].

Existem diversos *boards* de desenvolvimento com o uso do ESP 32, assim é importante identificar exatamente qual o modelo escolhido para evitar erros. O modelo utilizado neste projeto é o ESP32 NodeMCU-32S WROOM-32 e no momento da gravação no dispositivo é preciso segurar o botão **boot** para que a porta serial seja reconhecida e o programa compilado seja enviado [6].

Na conexão dos *jumpers* no ESP 32 ao *Display LCD* existia muito mal contanto. Para funcionar adequadamente foi utilizado uma *protoboard* de 400 pontos, na qual sendo mais espaçada contribuiu para um ótimo encaixe do ESP 32. Esta detalhe é importante, caso esteja mal encaixado o LCD liga de maneira intermitente, ou nem mesmo liga. A ligação do ESP 32 pode ser vista na Fig. 9 e do módulo I2C no LCD na Fig. 10.

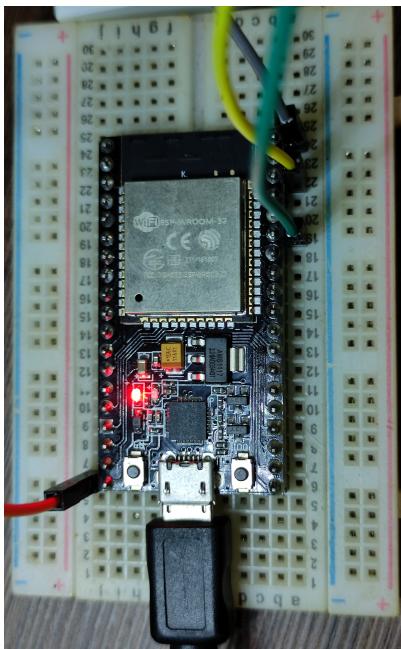


Fig. 9. Conexão do pinos do ESP 32 para o uso do LCD pelo módulo I2C. Repare com o ESP 32 está bem encaixado na *protoboard* de 400 pontos.

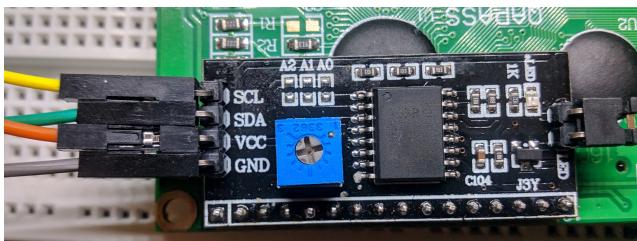


Fig. 10. Conexão dos pinos fêmea no módulo I2C. Repare nas cores dos fios que estão sendo conectados em cada pino.

A ligação completa do ESP 32 com o *display LCD* faz a busca da informação da orientação do celular na API *RESTful* e exibe na tela, como é exemplificado na Fig. 11.

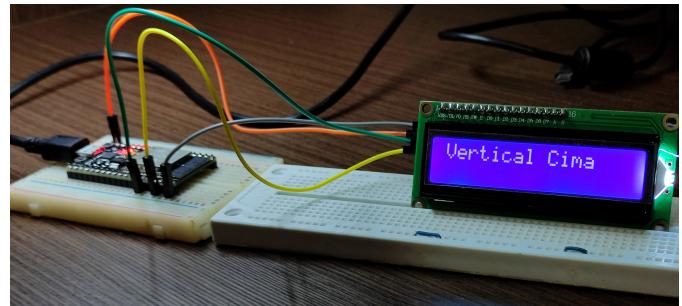


Fig. 11. Resultado da ligação do ESP 32 conectado ao *Display LCD* e exibição dos dados na tela.

2) *Aplicativo em Flutter*: Nas versões preliminares do aplicativo o principal problema era conseguir fazer a comunicação com a API *RESTFul*. Para que a API fosse acessada de qualquer origem foi fundamental o uso do CORS(Cross-Origin Resource Sharing), um mecanismo que permite que o servidor reconheça qualquer origem da requisição. Com o uso do Express.js a inclusão do CORS foi feita facilmente.

O aplicativo desenvolvido com o Flutter é apresentado na Fig. 12. No centro da tela dispõe o texto exibindo a orientação do *smartphone* e no canto superior direito existe um botão *switch* responsável por ligar os sensores. O dado recente da orientação do dispositivo é enviada para a API como previsto.

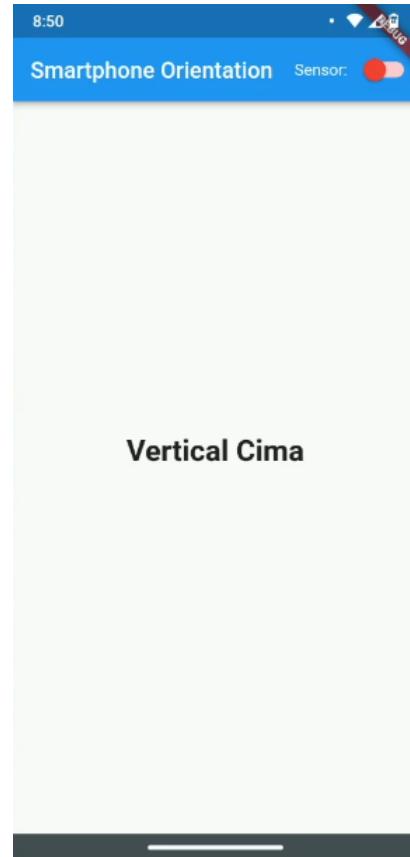


Fig. 12. Resultado do desenvolvimento do aplicativo em Flutter. O design é simples o suficiente para exibir a orientação atual do *smartphone* no centro. O botão superior a direita ativa os sensores.

3) API RESTful - Express.js: A API *RESTful* é ligada e escuta as requisições que são realizadas. Os dispositivos acessam as rotas e os recursos são coletados ou enviados como esperado. Para comprovar seu funcionamento é efetuado um *log* de cada requisição de envio ou busca do dado na Fig. 13.

```
Receiving...Horizontal Direita
Sending...Horizontal Direita
Receiving...Vertical Cima
Receiving...Horizontal Esquerda
Sending...Horizontal Esquerda
Receiving...Vertical Cima
Receiving...Horizontal Direita
Receiving...Vertical Baixo
Sending...Vertical Baixo
Receiving...Horizontal Direita
Receiving...Vertical Cima
Receiving...Horizontal Direita
```

Fig. 13. Resultado do desenvolvimento da API *RESTful* usando o Express.js. A figura exibe o *log* das requisições recebidas na API durante o seu funcionamento. Como é possível observar, a orientação é gravada como esperado.

4) O sistema de monitoramento completo: O sistema de monitoramento de forma completa demonstra a captação da orientação do *smartphone* no *display LCD*. O resultado são as seguintes possibilidades: vertical para cima, vertical para baixo, horizontal para direita e horizontal para esquerda. Como pode ser visto, respectivamente, na Fig. 14, Fig. 15, Fig. 16 e Fig. 17.

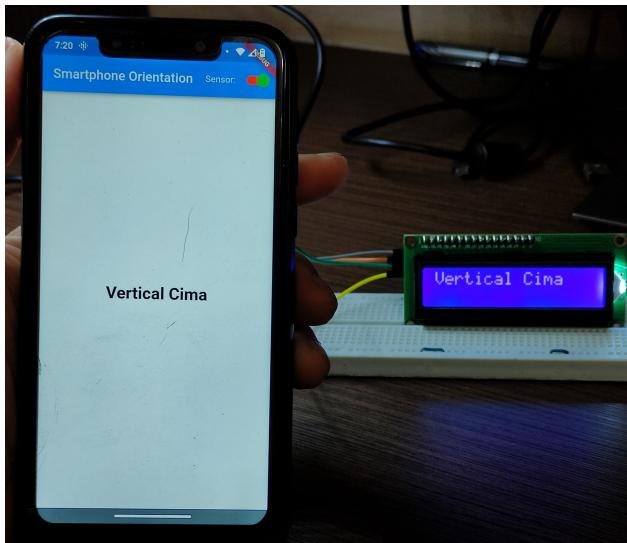


Fig. 14. Resultado do sistema. Tanto a aplicação e o aplicativo mostram a Orientação "Vertical Cima".

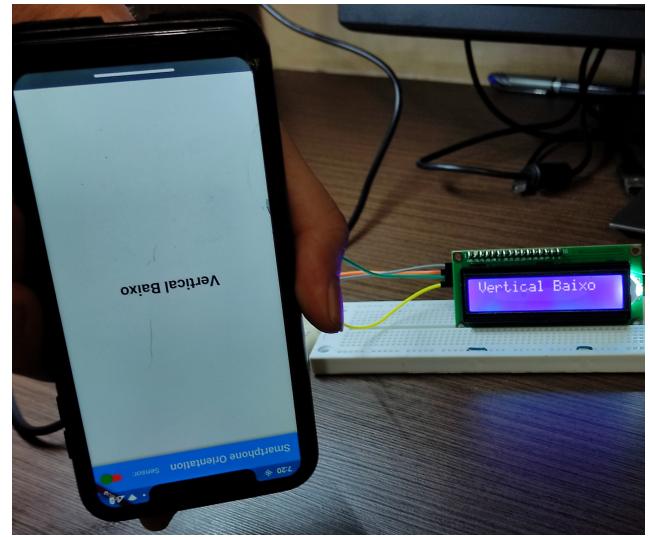


Fig. 15. Resultado do sistema. Tanto a aplicação e o aplicativo mostram a Orientação "Vertical Baixo".

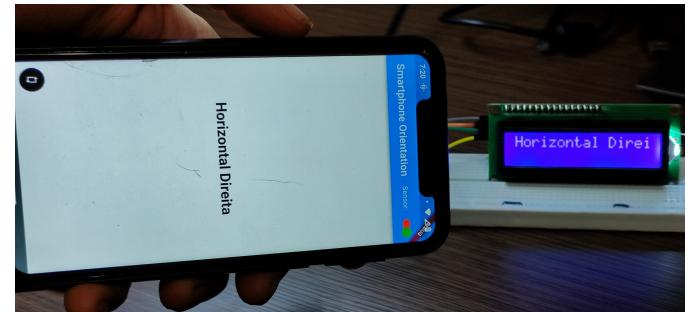


Fig. 16. Resultado do sistema. Tanto a aplicação e o aplicativo mostram a Orientação "Horizontal Direita".

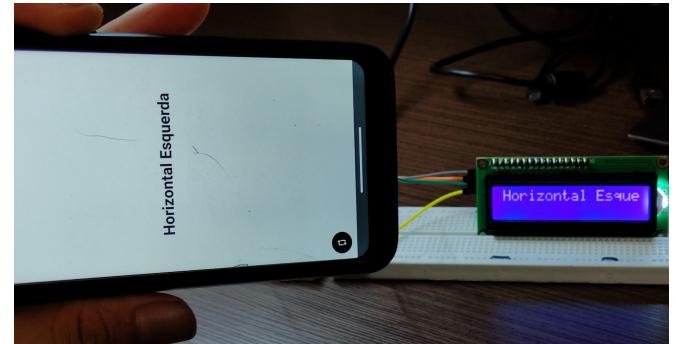


Fig. 17. Resultado do sistema. Tanto a aplicação e o aplicativo mostram a Orientação "Horizontal Esquerda".

V. CONCLUSÃO

Os sistemas de monitoramento remoto são importantes devido a facilidade ao acesso a informações em tempo real e permitir a análise sem estar em campo. O sistema apresentado cumpriu seu propósito de exemplificar um simples sistema de monitoramento remoto.

A metodologia utilizada possibilita a utilização de inúmeros dispositivos e aplicação do mesmo sistema para diversos casos

de uso. Além disso, tem-se facilidade no desenvolvimento e o baixo custo de recursos. Então, o sistema permite a escalabilidade, flexibilidade e portabilidade.

O sistema possui alguns atrasos (previstos) no tempo de resposta, mas nada que seja significativamente prejudicial para o projeto em questão. No entanto, em um sistema com uma exigência de requisições em intervalos de tempo excepcionalmente menores, vale-se a pena atenta-se e tratar possíveis erros. Vale ressaltar que estas limitações não veem das tecnologias utilizadas e, sim da exigência de um conhecimento mais aprofundado e atenção minuciosa no uso dessas ferramentas.

REFERENCES

- [1] A. Rodriguez. (2008) RESTful Web services: The basics. [Online]. Available: <https://cs.calvin.edu/courses/cs/262/kvlinden/references/rodriguez-restfulWS.pdf>
- [2] M. Quimbundo. (2019) Introdução ao Flutter: O Básico. [Online]. Available: <https://bit.ly/3pnaddi>
- [3] Adam. (2018) How Flutter Works. [Online]. Available: <https://buildflutter.com/how-flutter-works/>
- [4] R. Teja. (2021) Getting Started with ESP32 — Introduction to ESP32. [Online]. Available: <https://www.electronicshub.org/getting-started-with-esp32/>
- [5] Morgan. Native Device Orientation. [Online]. Available: https://pub.dev/packages/native_device_orientation
- [6] O. Winter. NodeMCU-32s — ESP32 ESP-WROOM-32 Development Board. [Online]. Available: <https://circuitsetup.us/product/nodemcu-32s-esp32-esp-wroom-32-development-board/>
- [7] Dart. http. [Online]. Available: <https://pub.dev/packages/http>
- [8] S. Labs. CP210x USB to UART Bridge VCP Drivers. [Online]. Available: <https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers>
- [9] D. Butler. ESP32 NodeMCU-32S Arduino Environment Setup. [Online]. Available: <https://www.youtube.com/watch?v=uYQBLecCIIdI>

André Geraldo Guimarães Pinto graduando em Engenharia da Computação desde de 2018 pelo Instituto Federal de Ciência e Tecnologia de Mato Grosso. Possui experiência como técnico de informática. Entre 2019-2021 atuou na área de *Smart Grids* como estudante de iniciação científica. Atualmente é interessado na área de *Machine Learning*.