

# Relatório 1

EEL7801

Aluno: Carlos Freitas

28 de Setembro de 2023

## Desenvolvimento Geral

O projeto é subdividido em 3 blocos, firmware, software e hardware, todas tiveram avanços em relação a proposta do projeto. As próximas seções terão foco no desenvolvimento individual de cada bloco com mais detalhes. Em relação ao desenvolvimento geral do projeto tem-se uma pequena mudança na estrutura do software, especificamente no servidor, tal mudança se refere a exclusão do processamento de machine learning, tendo em seu lugar um visualizador de dados, o qual permitiria a observação das medidas coletados pelos sensores.

## 1 Software

O software é composto por dois blocos, a visualização dos dados e o servidor. Sendo a conexão entre o bloco embarcado e o "mundo de fora", bem como o responsável por armazenar os dados coletados, o servidor é a parte fundamental do software, ele é feito na linguagem de programação Rust, linguagem que vem ganhando espaço no mercado web e de embarcados por sua segurança e performance. O código em si, utiliza das "crates" axum, tokio e serde que permitem a criação de um servidor web assíncrono, de alta performance. Além disso, o servidor utiliza a "crate" sqlx para a integração com o banco de dados, o qual é criado pela "engine" Sqlite, escrita em C e amplamente utilizada em ambientes mais restritos como em um celular, tablet, etc.

The image shows a code editor window with a dark background. The file name 'main.rs' is visible in the top left corner. The code is written in Rust and defines the main function of the server. It includes imports for 'anyhow::Result', 'axum::Router', and 'std::net::SocketAddr'. It also declares modules for 'handlers' and 'router'. The main function is marked with '#[tokio::main]' and is an asynchronous function 'fn main() -> Result<()>'. Inside, it creates a 'Router' instance, merges routes from the 'router' module, binds to the address '127.0.0.1:42068', and serves the routes using 'axum::Server'. The function returns 'Ok()' upon completion.

```
main.rs
use anyhow::Result;
use axum::Router;
use std::net::SocketAddr;

mod handlers;
mod router;

#[tokio::main]
async fn main() -> Result<()> {
    let routes_all = Router::new().merge(router::routes());

    let addr = SocketAddr::from([127, 0, 0, 1], 42068);

    axum::Server::bind(&addr)
        .serve(routes_all.into_make_service())
        .await?;
    return Ok(());
}
```

Figura 1: Arquivo fonte "main" do servidor

Com o servidor estando ligado é preciso definir como ele cuidará das requisições enviadas para ele, as quais, nesse caso, seriam de envio de dados provenientes do esp32 e requisição de dados tanto pelo esp32 como por requisições externas. Um esboço das funções que cuidam dessa função está

presente na figura 2. Nessa figura pode-se ver o protótipo de três dessas funções que manipulam as requisições "get\_complete\_data\_handler", "post\_measuring\_handler" e "get\_statistics\_handler", bem como três structs que representam as estruturas das mensagens, que teriam formato Json.

```

1 use anyhow::Result;
2 use axum::extract::Query;
3 use axum::response::IntoResponse;
4 use axum::Json;
5 use serde::{Deserialize, Serialize};
6
7 #[derive(Deserialize, Debug)]
8 pub struct DataParameters {
9     samples: Option<u32>,
10     stats: Option<bool>,
11 }
12
13 #[derive(Deserialize, Serialize, Debug)]
14 pub struct MeasurementSample {
15     temp: u16,
16     umidity: u16,
17     light: u16,
18     air_quality: u16,
19 }
20
21 #[derive(Serialize, Debug)]
22 #[serde(transparent)]
23 pub struct MeasurementsData {
24     dataset: Vec<MeasurementSample>,
25 }
26
27 pub async fn post_measuring_handler(Json sample: Json<MeasurementSample>) -> impl IntoResponse {
28     todo!();
29 }
30
31 pub async fn get_statistics_handler() -> Result<Json<MeasurementSample>> {
32     todo!();
33 }
34
35 pub async fn get_complete_data_handler(
36     Query data_param: Query<DataParameters>,
37 ) -> Result<Json<MeasurementsData>> {
38     todo!();
39 }

```

Figura 2: Arquivo fonte que processa as requisições

Quanto ao visualizador de dados, ele é um script feito com python para se emular uma requisição ao servidor e então utilizar os dados recebidos via http para se traçar gráficos do comportamento das medidas de cada sensor. O andamento do script pode ser visto na figura 3.

```

import argparse
import requests as rq
import json
import matplotlib.pyplot as plt
import numpy as np

parser = argparse.ArgumentParser(description='Simple method for visualizing data from monitoring-system api')
parser.add_argument('port', help='Port number of the api', type=str)
parser.add_argument('-n', '--samples', help='Defines the size request query', required=False, type=int)
parser.add_argument('-s', '--stats', help='Enables request for statistics', action='store_true', required=False)

args = parser.parse_args()

# TODO default case
if args.samples != None:
    samples = args.samples
else:
    samples = 0

if args.stats:
    stats = "True"
else:
    stats = "False"

# data = rq.get(f'http://127.0.0.1:{args.port}/data?')

```

Figura 3: Arquivo fonte que processa as requisições

## 2 Firmware

O Firmware do projeto será feito integralmente no framework esp-idf da espressif, empresa responsável pelo esp32, usando C++ como linguagem de programação. Esse bloco do projeto será responsável por coordenar a leitura dos sensores, o envio para o servidor e a visualização por meio

de um display. Nessa parte serão desenvolvidos drivers para o controle dos sensores e a interface da comunicação sem fio.

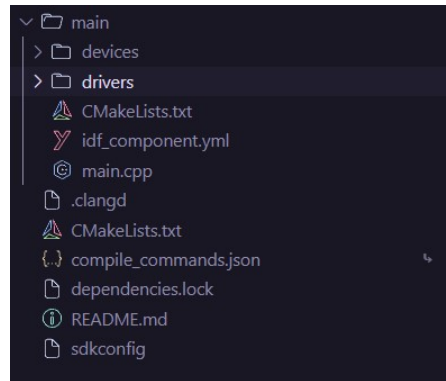


Figura 4: Estrutura de arquivos do Firmware

### 3 Hardware

O Hardware é bastante simples, sendo composto por um esp32, dois sensores de temperatura, um lm35 e um dht11, que funciona também como sensor de humidade, além disso há também um sensor de irradiância ultravioleta, gyml8511, e um sensor de qualidade do ar, cjmdu-811. Para visualização dos dados "localmente" tem-se um display.

Todos os dispositivos, display e sensores, serão gerenciados pelo esp32 levando em conta seus tempos de amostragem, formas de comunicação, etc. Infelizmente ainda não fiz o esquemático e realizei os testes físicos, pois nem todos os sensores foram entregues. No entanto, as conexões serão relativamente simples, com pouca ou nenhuma adição além de resistores de pull-up e semelhantes, isso se deve a inclinação da complexidade estar no software e firmware que manejariam a sincronia e a comunicação por código invés de por hardware.