UNIVERSIDADE SÃO JUDAS TADEU

Allan Picoli Pasqualino

Ana Carolina da Silva

Joao Paulo Martins Rodrigues

Matheus Moura Nepomuceno Pereira

Monitoramento IoT para Sistema de Gás Industrial

São Paulo

2021

Allan Picoli Pasqualino
Ana Carolina da Silva
Joao Paulo Martins Rodrigues
Matheus Moura Nepomuceno Pereira

Monitoramento IoT para Sistema de Gás Industrial

Trabalho de Conclusão de Curso de Graduação em Engenharia da Computação da Faculdade de Tecnologia e Ciências Da Universidade São Judas Tadeu como requisito para obtenção do título de Bacharel em Engenharia da Computação
Orientador: Prof. Dr. Jeison Willian Gomes da Fonseca.

São Paulo
2021

FICHA DE APROVAÇÃO

Data: _____

Horário: _____

Sala: _____

Título: _____

Nome completo dos alunos	RA
Aluno 1	
Aluno 2	
Aluno 3	
Aluno 4	

Observações sobre o trabalho:

_____

_____

_____

Professores da Banca examinadora	Assinatura
Professor 1 (orientador)	
Professor 2	
Professor 3	

Resultado: Somente aprovado ou reprovado

DEDICATÓRIA

Dedicamos este trabalho a família, aos amigos, a todos aqueles que nos apoiaram nas horas sombrias, mas principalmente aqueles que duvidaram da nossa força de vontade.

RESUMO

Desde tempos remotos a humanidade convive com gases inflamáveis, sendo responsáveis por cavernas assombradas que traziam um misterioso fim para aqueles que se aproximavam, mas também por serem a força motriz de enormes indústrias e matrizes energéticas de países inteiros. Apesar desses gases serem de extrema importância para diversos setores da vida cotidiana, eles representam um elevado risco para aqueles que lidam com o mesmo. Estes gases, em seu estado natural muitas vezes, indetectável, possuem uma grande capacidade de causar acidentes com um alto potencial letal.

O trabalho desenvolvido teve como principal objetivo criar um sistema eficaz e viável para sua detecção, e tendo em vista o cenário atual de distanciamento sua proposta inicial foi desenvolver tal sistema de uma maneira que possa ser acessado e monitorado por um usuário remoto através de seu terminal próprio. Integrando o sistema a um banco de dados é possível monitorar o sensor em tempo real ou observar sua variação em um determinado período de tempo já decorrido.

Palavras-Chave: Gás; Sensor; React; Server; Monitoramento.

ABSTRACT

Since remote times, humanity lives together with flammable gases, being responsible for haunted caves which brought a mysterious end for those who approached, but as well for being the driving force of huge industries and energy matrices of whole countries although these gases are of extreme importance in many sectors of daily life, they represent a high risk for those who deal with them. These gases in their natural state are many times undetectable, possessing a high capacity of causing accidents with a high letal potential.

The developed work had as main objective to create an effective system and viable for your detection, having on sight the actual scenario of social distancing, the initial proposal was to develop such system in a manner that it can be accessed and monitored by an remote user in his own terminal. Integrating the system to a database its possible to monitor the sensor in real time or to observe its variation in a specific time band already past.

Keywords: Gas; Sensor; React; Server; Monitoring.

LISTA DE FIGURAS

Figura 32 - Estrutura de pastas do Frontend

Figura 33 - Site apresentando gráfico em tempo real e consulta no histórico

LISTA DE ABREVIATURAS

API - *Application Programming Interface*

CSS - *Cascading Style Sheets*

DAC - *Digital to Analogic Converter*

GND - *Graduated Neutral Density Filter*

GLP - Gás Liquefeito de Petróleo

GSM - *Groupe Special Mobile*

HTML - *HyperText Markup Language*

HTTP - *Hypertext Transfer Protocol*

IDE - Integrated Development Environment

IoT - *Internet Of Things ou Internet*

MVC - *Model, View and Controller*

PPM - Parte por milhão

REST - *Representational State Tranfer*

USB - *Universal Serial Bus*

WLAN - *Wireless Local Area Network*

SQL - *Structured Query Language*

SPA - *Single Page Applications*

Sumário

1 Introdução

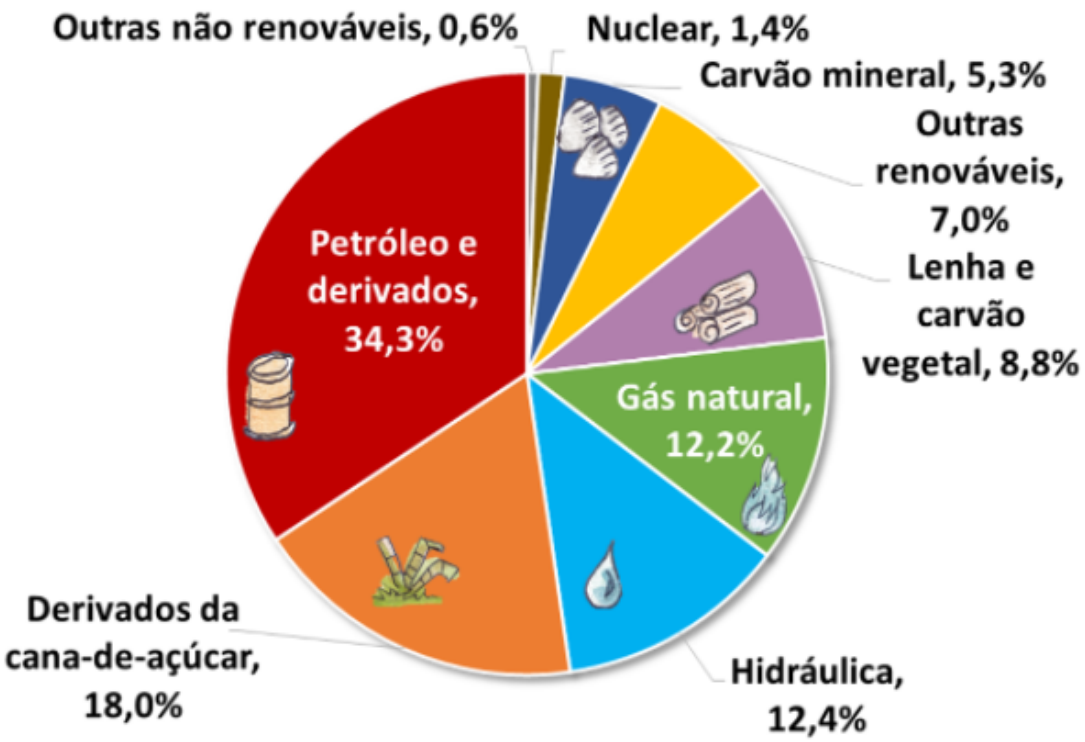
O gás natural é composto por diversos tipos de hidrocarbonetos presentes em formações subterrâneas porosas que podem ser encontradas em solo continental ou em alto mar, apesar de não ser uma fonte renovável de energia é considerado uma forma de energia limpa. Existem dois tipos de gás natural, o tipo associado, que está dissolvido no petróleo, é utilizado na produção de petróleo, e o tipo não associado está presente na formação rochosa que contem o petróleo e é utilizado para produzir o gás natural propriamente dito (eCycle, 2021).

A partir da década de 40 apenas o gás natural começou a ser utilizado como fonte de energia devido aos avanços de condicionamento e transporte desenvolvidos durante o pós-guerra. Seus primeiros usos em grande escala foram feitos na Europa depois da invenção do queimador bunsen utilizado para efetuar o aquecimento de compostos em laboratório (eCycle, 2021).

Na indústria o gás natural é amplamente utilizado para a produção de Metanol, amônia, uréia e na geração de energia elétrica. Como combustível ele é utilizado em ônibus, caminhões e outros automóveis, sua queima é mais limpa, o que acaba por dar uma vida útil maior aos motores que o utilizam, ele pode ser até 70% mais barato do que gasolina e álcool (GasNet, 2021).

No Brasil o gás natural representa 12,5% da matriz energética conforme demonstrado na figura 1, seu uso foi intensificado após a descoberta do pré-sal que é uma área extensa de reserva petrolífera contendo um petróleo de maior qualidade do que o petróleo comum (Mundo educação, 2021).

Figura 1 - Matriz Energética Brasileira de 2019 (BEN, 2020)



Os vazamentos de gás representam a maior parte dos acidentes domésticos com vítimas fatais tanto devido a explosões geradas pela sua combustão quanto devido a asfixia quando não se nota o vazamento em lugares fechados (Alves, 2020).

O gás liquefeito de petróleo é utilizado tanto no meio doméstico quanto no meio industrial para, aquecimento de água, calefação, combustível para empilhadeiras, combustível para oxicorte (corte de metais por combustão), fonte energética no ramo da metalurgia e outros diversos setores (Ultragaz, 2021).

Sua produção é feita por meio da destilação fracionada do petróleo onde se separam os seus derivados como a gasolina, o óleo diesel, o querosene e outros. Então os subprodutos serão destinados a outros processos que irão refina-los e transformá-los nos produtos finais (Petrobrás, 2021).

Diversos métodos de detecção têm sido utilizados para compor modelos de sistemas de segurança contra vazamentos e incêndios utilizando microcontroladores, genericamente são utilizados um controlador Arduino e um sensor da série MQ que também detectam, metano, gás carbônico, hidrogênio e outros gases como visto em Kiran (2016).

Alguns métodos de monitoramento via microcontroladores visam a detecção local do vazamento de GLP (Gás Liquefeito de Petróleo), utilizando o sistema para gerar um alarme que soa localmente. Recentemente o foco tem sido a detecção remota que consiste no sistema estar em um ambiente diferente do indivíduo responsável pelo monitoramento, como por exemplo demonstrado em Kiran (2016), o uso de um módulo GSM (*Groupe Special Mobile*, ou sistema global para comunicações móveis) para o encaminhamento do alerta via celular através de mensagens de texto. Outro método amplamente usado, não só para a detecção de gases mas para gerenciamento e monitoramento de sistemas físicos, é utilizando a arquitetura IoT (*Internet Of Things ou Internet das coisas*).

1.1 Justificativa

O gás natural está presente na vida de todos desde sempre. Da antiguidade até os dias atuais. Conforme a evolução da indústria seu uso está se tornando cada vez maior e mais importante nesse meio.

Apesar disso, o uso de gás na indústria, residências ou automóveis é muito delicado. O gás normalmente usado é inflamável com pontos de ignição muito altos geralmente usados para produção de calor ou energia, como em chuveiros para o aquecimento da água ou nos carros e maquinários industriais para o uso de motores.

De maneira geral, toda a cadeia de tratamento, transportes, manuseio e utilização do gás precisa ser monitorada para a segurança das pessoas e um bom funcionamento dos equipamentos, pois geralmente é usado em altas temperaturas e pressões oferecendo grandes riscos para as pessoas que trabalham com ele.

Face aos fatos expostos acima, a realização deste trabalho justifica-se por associar a literatura e o aprendizado em sala de aula para aumentar a segurança no espaço de trabalho das pessoas e em suas casas. Apresentando uma solução para ser utilizada na prevenção de vazamentos de gases.

1.2 Objetivos

1.2.1 Objetivo Geral:

Desenvolver um sistema capaz de monitorar em tempo real vazamentos de gás em instalações.

1.2.2 Objetivos Específicos

Implementar um sistema de sensores IoT que deve realizar as medições enviando as mesmas para o painel de monitoramento na nuvem.

I. Construir o sistema de sensores, e usar o *NodeMCU* para coletar as informações.
II. Desenvolver a *API* (*Application Programming Interface*) a partir do *NodeJS* para armazenar as informações no banco de dados e abrir uma comunicação *WebSocket*.
III. Desenvolver o painel de monitoramento a partir do *ReactJS* que irá tratar os dados armazenados.

2.Revisão Bibliográfica

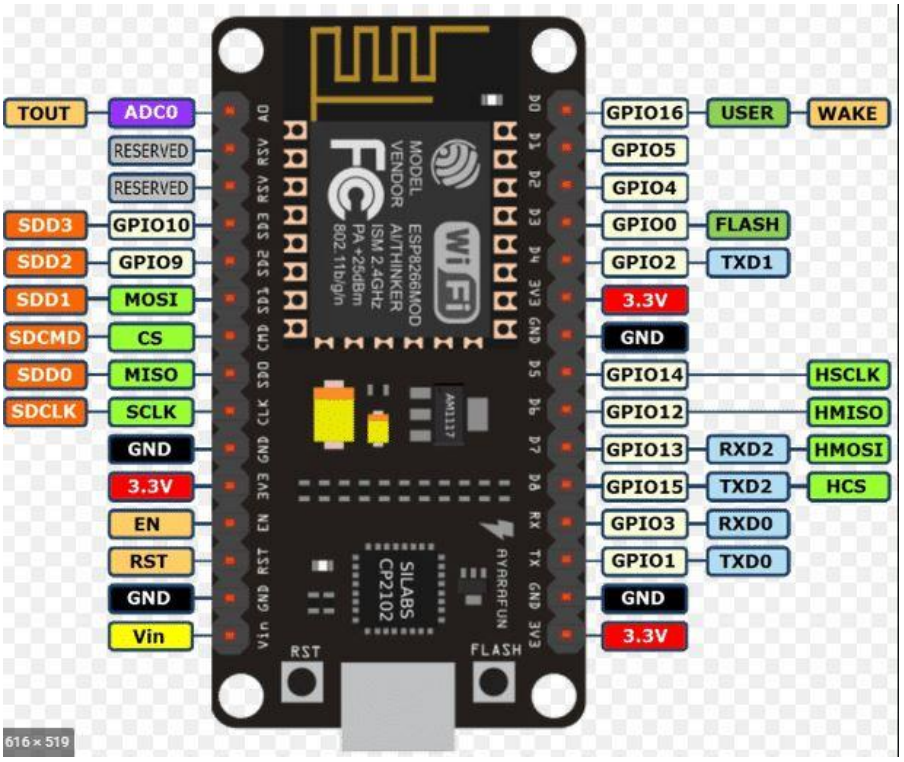
A detecção de gases pode ser feita de diversas maneiras, mas a utilização de um sistema eletrônico provê uma confiabilidade maior do que apenas utilizando o olfato para captar o odor de Mercaptan presente no GLP justamente para identificar o vazamento do gás.

2.1 Detecção.

Para a detecção do GLP no ar foi utilizado o módulo sensor de gás MQ-9, capaz de detectar concentrações de 20 ppm até 2.000 ppm para monóxido de carbono e de 500 a 10.000 ppm para GLP, metano, butano, dentre outros gases. Conforme mostrado em Santos, 2012, os sensores do modelo MQ são capazes de detectar diversos gases inflamáveis e até mesmo fumaça, utilizando a variação de temperatura do composto SnO2 (Dióxido de Estanho), como material sensível aos gases desejados. A mudança de condutividade do composto quando em contato com o gás é convertida, de acordo com sua concentração, num sinal elétrico que é captado pelo microcontrolador.

Os microcontroladores têm a função de controlar o *hardware* do projeto, utilizando o *NodeMCU ESP8266* demonstrado na figura 2, projetado para funções dentro do campo da IoT (Ronsani, 2018). O módulo ESP8266 está integrado à placa do NodeMCU, sua função é receber as informações do sensor e transmiti-las para o *NodeJS*, realizando a função de *Client Socket.*

Figura 2 - NodeMCU ESP8266 (electronicshub)



2.2 Comunicação remota.

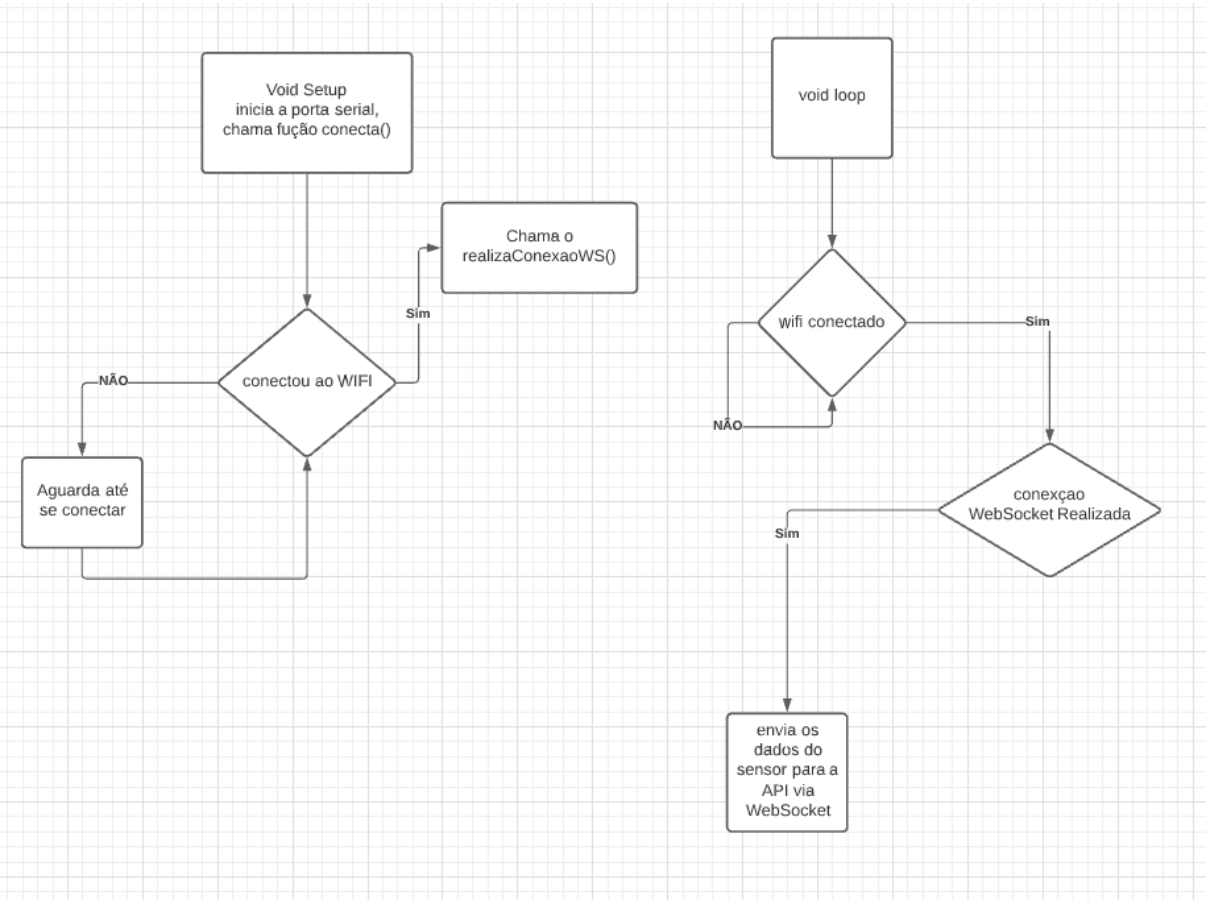
Utilizando novamente Kiran (2016) e Santos (2012) como exemplo podemos notar o uso da placa GSM como uma escolha comum, contudo, seu uso acaba sendo uma limitação para o projeto. A rede GSM foi desenvolvida nos anos 80 para permitir a comunicação de celulares de uma maneira mais estável do que seus antecessores, podemos então encontrar uma alternativa no conceito de IoT. A arquitetura IoT permite que diversos dispositivos possam ser interconectados e ao mesmo tempo podem se comunicar com uma aplicação, (Silva, 2017). A arquitetura é baseada em diversas camadas que representam todo um serviço de interconexão entre objetos do mundo real e uma aplicação, englobando principalmente o sensoriamento, o envio de dados para o banco de dados e o tratamento dos dados pelo usuário final (Ronsani, 2018).

2.3 Microcontrolador e Recepção dos Dados

O funcionamento do projeto se inicia com o sensor ligado ao *NodeMCU* (Figura 3) através de sua porta analógica, inicialmente foi idealizado o uso de um DAC (*Digital to Analogic Converter*) para converter o sinal analógico em digital para uma melhor leitura do sensor, mas essa necessidade logo foi descartada mediante a presença de ambas portas digital e analógica no *NodeMCU*

Para a realização dos testes com o microcontrolador foi utilizado um potenciômetro comum para simular a variação de tensão gerada pelo sensor na presença do gás para o qual ele estaria calibrado. Mediante a variação de tensão, o *NodeMCU* através do módulo ESP8266, que por sua vez, através da conexão *Web-server/Web-socket* os dados são enviados para a API e então a tensão é convertida para PPM (Parte por milhão) do gás.

Figura 3 - Fluxograma da conexão ESP8266



O desenvolvimento do programa no microcontrolador, foi projetado para executar 3 operações: conectar ao wifi, estabelecer *websocket* com a API, capturar os dados do sensor e repassar para a API.

Começando pelo void setup, temos apenas a inicialização da porta serial, e a chamada da função conecta(), nessa função utiliza os métodos de *WiFi* que é disponibilizado pela biblioteca ESP8266WiFi.h, o primeiro método usado é WiFi.begin(nome_da_rede, senha_da_rede) para iniciar a conexão, dentro dessa função possui um while, que só permite que o programa avance se o wifi.status == WL_CONNECTED, que é o status retornado quando a conexão com o wifi ocorreu corretamente, com a conexão wifi funcionando, já pode chamar a função RealizaConexaoWS, essa função faz uso da biblioteca WebSocketClient, inicia com client.connect(endereço_servidor, porta_da_conexao_websocket), o client conseguindo se conectar ao servidor, inicia o *handshake* (aperto de mão, processo no qual duas ou mais máquinas afirmam o reconhecimento uma da outra e estão prontas para se comunicar). e altera a variavel conexaoWs para *true*, passando para o void loop, temos a conexão com o *Wifi* e já é possível realizar a comunicação via *WebSocket*, então é feito a leitura da porta analogica onde o sensor estara ligado, e utiliza o metodo WebSocketClient.sendData(Envia_uma_string), o formato da informação enviado para a API é "nome'nome_do_sensor_lido'valor'valor_lido_da_porta_analogica'unidade'unidade_de_medida" dessa forma a API consegue quebrar essa string e transformar em um objeto.

2.4 API

Para que o projeto possa ser utilizado por um usuário remoto, os dados gerados pelo *NodeMCU* precisam fluir para a camada de aplicação da arquitetura IoT, em (Sant'Anna, 2018) utiliza-se a função nativa de criação de rede WLAN (*Wireless Local Area Network*) propicia a criação de um web-server, contudo para a criação de uma API mais robusta é necessário a utilização de novo ambiente de execução para o Web-server. Em (Basso, 2014) utiliza-se a plataforma *Node JS* para que a execução do JavaScript Server seja executada com o foco em sistemas de tempo real. E em adição ao *Node JS*, pode-se trocar o padrão de arquitetura MVC (*Model, View and Controller*) pela biblioteca *React*, tornando o sistema mais escalável e ainda permitindo a separação dos aspectos da aplicação como demonstrado em (Cechinel, 2012). O *NodeJS* será responsável por gerir a API, sua escalabilidade facilita o desenvolvimento do *WebServer* utilizando diversas características que o diferenciam de outros servidores como por exemplo o Apache ou o Glassfish (Basso, 2018).

API RESTful é uma interface que fornece dados em um formato padronizado baseado em requisições HTTP (*Hypertext Transfer Protocol*). Antunes (2019)

Os conceitos do **REST** foram submetidos à tese de doutorado de **Roy Fielding** nos anos 2000, onde o princípio fundamental é usar o protocolo HTTP para comunicação de dados (Lima, 2020).

REST (*Representational State Tranfer*) é um modelo de arquitetura e não uma linguagem ou tecnologia, fornece diretrizes para sistemas distribuídos se comuniquem corretamente, a comunicação é realizada através de requisições utilizando o protocolo HTTP, os métodos mais comuns são:

POST: Criar dados.

GET: Buscar(leitura) dados.

PUT: Atualizar dados.

DELETE: Excluir dados.

Normalmente sua resposta é dada em JSON (*JavaScript Object Notation*), construída a partir de rotas, exemplo:

Requisição

GET http://siteExemplo.com/users realiza um GET (buscar dados) na rota users

Resposta (formato JSON)

```
{
   “user”: {
        “nome”: “Exemplo user”
        }
}
```

Status(200)

Uma mesma rota pode receber vários métodos HTTP diferentes.

A API sempre retorna um status em sua resposta para identificar a situação da solicitação, alguns dos estados mais comuns

200 – OK sucesso na requisição.

201 – CREATED criado com sucesso (utilizado quando o método POST foi bem sucedido).

400 – BAD REQUEST a requisição foi mal sucedida, inúmeros motivos pode retornar esse status

404 – NOT FOUND rota não encontrada (um erro muito comum de ser visto na internet quando um site ou uma página está fora do ar).

2.5 Backend

O backend da aplicação foi desenvolvido utilizando a tecnologia *NodeJS*, que permite a execução de código JavaScript fora do navegador Web.

Começando pela comunicação *WebSocket*, para estabelecer essa comunicação foi usado a biblioteca *WebSocket* que permite estabelecer um servidor, e esse servidor pode repassar as informações a diversos clientes que estejam conectados.

Para se conectar ao banco de dados foi utilizado a biblioteca Knex.js que é uma "query builder" criador de consultas, a principal vantagem dessa query builder é que ele permite escrever uma consulta SQL (*Structured Query Language*) que possa ser usada em qualquer banco de dados seja um MySQL(que está sendo usado nesse projeto), SQL Server, Oracle entre outros, dessa forma só é necessário configurar um arquivo para que possa ser usado em qualquer banco.

Analisando a estrutura da API, o primeiro arquivo é o package.json, esse arquivo é o principal arquivo de qualquer aplicação criada com *NodeJS*, server para guardar informações como nome do projeto, versão autores, e principalmente Scripts(trechos de códigos que sejam úteis no projeto) e as Dependências que são as bibliotecas que o projeto depende para o correto funcionamento.

knexfile.js nesse arquivo é onde se realiza a configuração de qual banco será usado e as credenciais necessárias como host, user, password, database e port.

Passando para pasta SRC onde se encontra todo o funcionamento da API. A pasta Componentes serve para alocar códigos que serão reutilizados algumas vezes durante a execução da API, mas que não estão diretamente ligado a regra de negócio,
No caso dos componentes utilizados temos DataAtual, HorarioAtual que são usados durante a persistência dos dados no Banco.

Pasta database possui um arquivo chamado connection.js esse exporta a conexão com o banco de dados, dessa forma não é necessário ficar instanciando o Knex em todos os arquivos que for necessário se conectar com o banco.

Pasta Erro, possui os arquivos de erros que devem ser retornados ao cliente quando ocorrer algum problema na API, CampoInvalido.js retorna uma mensagem de erro alertando que o cliente preenche de forma errada algum campo da requisição. NãoEncontrado.js, retorna um aviso informando que não foi encontrado o que o cliente está solicitando a API. dessa forma é possível criar outros arquivos para erros mais específicos tornando as respostas da API mais eficientes.

Pasta Rotas nessa pasta se encontra as rotas que serão utilizadas durantes as requisições, no caso da API possui outra pasta chamada Sensor, dentro dessa pasta temos o arquivo DAOSensor.js esse arquivo é o responsável por utilizar o arquivo Connection para poder persistir os dados no banco, dentro desse arquivo possui 5 métodos o Inserir(que recebe um objeto do tipo Sensor) e persiste as informações desse objeto no banco de dados, Listar(Lista todas as informações contida na tabela Sensor),PegarPorId(Realiza uma consulta de um sensor específico), Atualizar(Atualiza os dados de algum sensor específico) e por ultimo remover que deleta do banco um registro específico. nesse arquivo também são utilizados os códigos da pasta Erro caso o cliente passe algum dado errado ou faça alguma solicitação incorreta remover ou atualizar um Sensor(ID) que não exista no banco.

Arquivo Sensor.js esse arquivo é a Classe Sensor onde define o construtor do sensor estabelecendo os atributos necessário e define os métodos dessa classe, a classe Sensor utiliza os métodos do arquivo DAOSensor, dessa forma a classe Sensor não acessa diretamente o banco de dados apenas passa os dados que devem ser persistidos no banco

No arquivo Index.js começam as rotas do programa, o cliente faz uma requisição o arquivo index intercepta ela e realiza uma ação de acordo com o Método HTTP que foi enviado.

Ex: cliente faz uma requisição na rota api/sensor/ utilizando o método POST, para que essa requisição seja concluída com sucesso a API espera que o cliente envie no corpo da requisição as informações básicas para a criação do sensor, assim que recebe essa informações instancia um novo objeto da classe Sensor e chama o método Criar para que seja persistido no banco de dados essa informação , se tudo ocorreu corretamente a api retorna um status(201) que significa “created”(sucesso na criação) e as informações do sensor criado.

Voltando a pasta SRC, o Arquivo Server.js que inicia a API, é utilizado a biblioteca Express, que serve para criar o web server e poder trabalhar com as requisições HTTP, esse arquivo então fica responsável por receber as requisições HTTP e repassar para as rotas corretas, ao mesmo tempo que instancia o *WebSocket* e repassa as informações para os cliente conectados, dessa forma é possível receber as informações do microcontrolador, persistir essas informações no banco de dados e repassar essas informações em tempo real para o cliente front end

2.6 Front-end

O React é uma biblioteca de JavaScript, responsável pelo *View* na aplicação, ou seja, o React trabalha diretamente com o usuário. O React permite o uso de uma arquitetura chamada Flux que diferentemente de uma arquitetura MVC, considerada demasiadamente complexa, facilita o entendimento e modificações no código

O React é uma biblioteca front-end e tem como um de seus objetivos facilitar a conexão entre diferentes partes de uma página, portanto seu funcionamento acontece através do que chamamos de **componentes** (Roveda, 2016).

A criação do Front-end irá utilizar o conceito de SPA(*Single Page Applications*), dessa forma o carregamento de recursos(HTML, CSS e JavaScript) ocorrem uma única vez, quando o usuário transita pelo site, não existe a necessidade de realizar uma nova requisição ao servidor para que ele envie novamente os recursos, o *ReactJS* (*Frameworks*) através de uma requisição HTTP, recebera um JSON com as informações necessárias, e a partir dos recursos já carregados anteriormente irá gerar a página solicitada pelo usuário, dessa forma desloca todo o esforço de renderização para o cliente, permitindo um tráfego mais suave entre cliente e servidor.

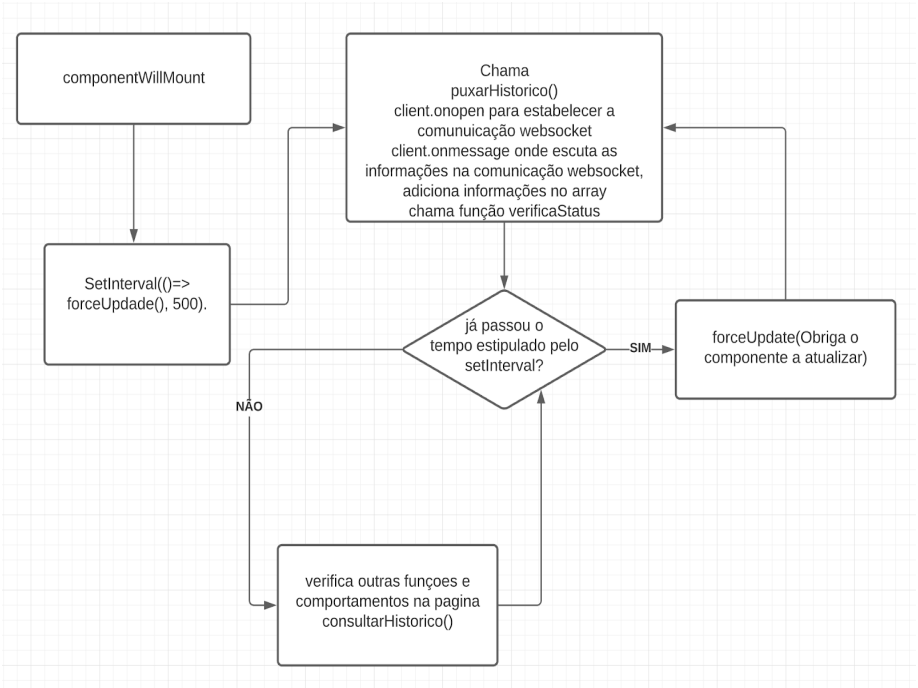
A finalidade do font, é exibir ao usuário um gráfico em tempo real como sinalização de status, e uma consulta no histórico existente, para alcançar tal objetivo foi utilizado o React para criar a SPA, onde é realizada uma conexão WebSocket para alimentar as informações do gráfico de tempo real, e outra conexão para realizar requisições HTTP com a API.

Iniciando pela pasta public, o index.html é um arquivo *html*, que possui algumas configurações no *head* e dentro do *body* uma *div* com id = “root”, é dentro dessa *div* que o *React* irá renderizar todo o código da aplicação.

Na pasta SRC o, index.js, é o primeiro arquivo a ser executado, importa o React e o ReactDom, que será responsável por manipular e renderizar o componente App.js na Div id=”root” na dom do arquivo Index.html

App.js o componente principal da aplicação, a página consiste em um loop para que haja a constante atualização do gráfico em tempo real. Assim que o componente App é criado o método de ciclo componentWillMount() é chamado, mas esse método só executa uma única vez assim que o componente é renderizado, para sanar isso é utilizado logo em seguida o setInteval com um tempo de espera de 500 milisegundos, enviando o método forceUpdate() que fará o componente renderizar novamente. Ainda dentro desse método é chamado à função puxaHistórico(), e inicia a conexão com o *WebSocket*, recebendo uma mensagem que atualiza o array do gráfico de tempo real (Figura 4).

Figura 4 - Fluxograma do React



A função consultaHistorico() realiza uma requisição a API na rota api/sensor/datas, que é um método POST que recebe um JSON, o conteúdo desse *JSON* é uma data a qual quer realizar a consulta.

Tanto o gráfico de tempo real quanto o de consulta histórica utilizam o mesmo componente BarChart.

O arquivo style.css, contém as estilizações da página.

Na Pasta componentes temos o BarChar, esse componente é disponibilizado através de uma ferramenta da Google Charts, essa ferramenta tem como finalidade a criação de gráficos nos mais diversos formatos.

Dentro do arquivo index.js do BarChart temos a importação da biblioteca do react-google-charts que permite utilizar e customizar o componente Charts dessa biblioteca, esse componente recebe alguns parâmetros para seu correto funcionamento,

dataMapa recebe um array contendo as informações do gráfico no formato(“eixo x”, eixo y), title o título que o gráfico irá receber, corMapa a cor do mapa, status se o gráfico está ou não disponível para visualização (“ready”, “not”) .

3 Metodologia

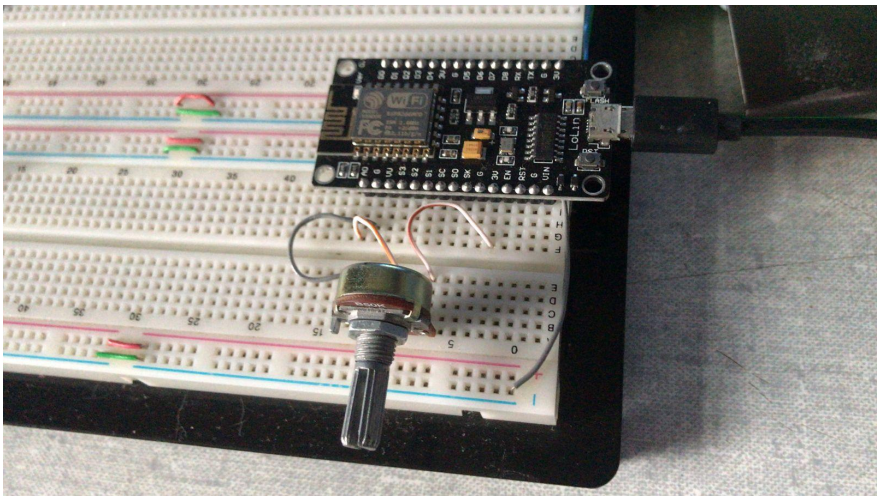
Ao longo deste capítulo serão apresentados as etapas de desenvolvimento do projeto para a compreensão geral de sua implementação prática.

3.1 Preparação dos Sistemas de Hardware

O NodeMCU ESP8266 foi acoplado a uma protoboard, sendo sua alimentação via USB 5V. Junto à ele um potenciômetro de 50K Ohms tendo suas extremidades alimentadas com as portas positivo (3.3V) e negativo (GND) DC do próprio NodeMCU. O pino intermediário do potenciômetro onde há a variação da tensão é ligado a porta A0 do NodeMCU.

A entrada de dados para a programação do NodeMCU também é feita pela entrada USB, compartilhada com a sua alimentação DC, conforme demonstrado na figura 5.

Figura 5 - NodeMCU ligado ao potenciômetro

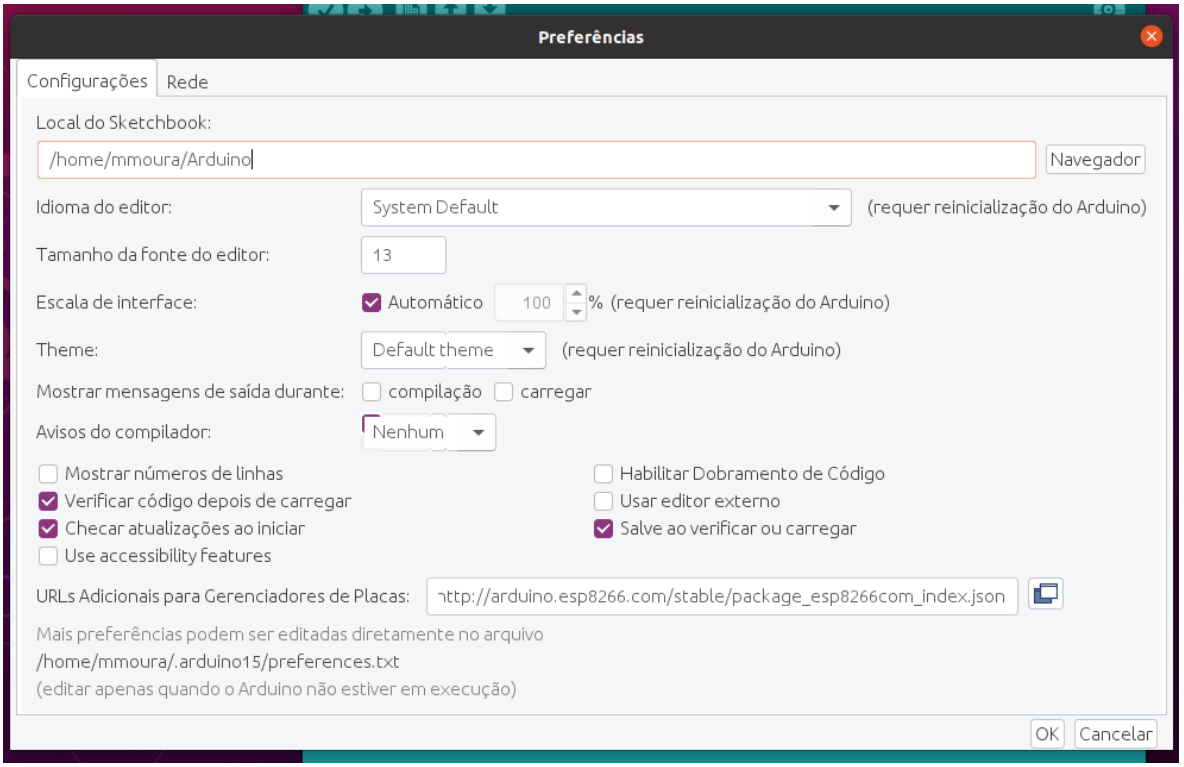


3.2 Preparação dos Sistemas de Software

Para a programação do NodeMCU ESP8266 foi necessário instalar o software open-source Arduino IDE e realizar alguns ajustes nas preferências para o funcionamento pleno do hardware.

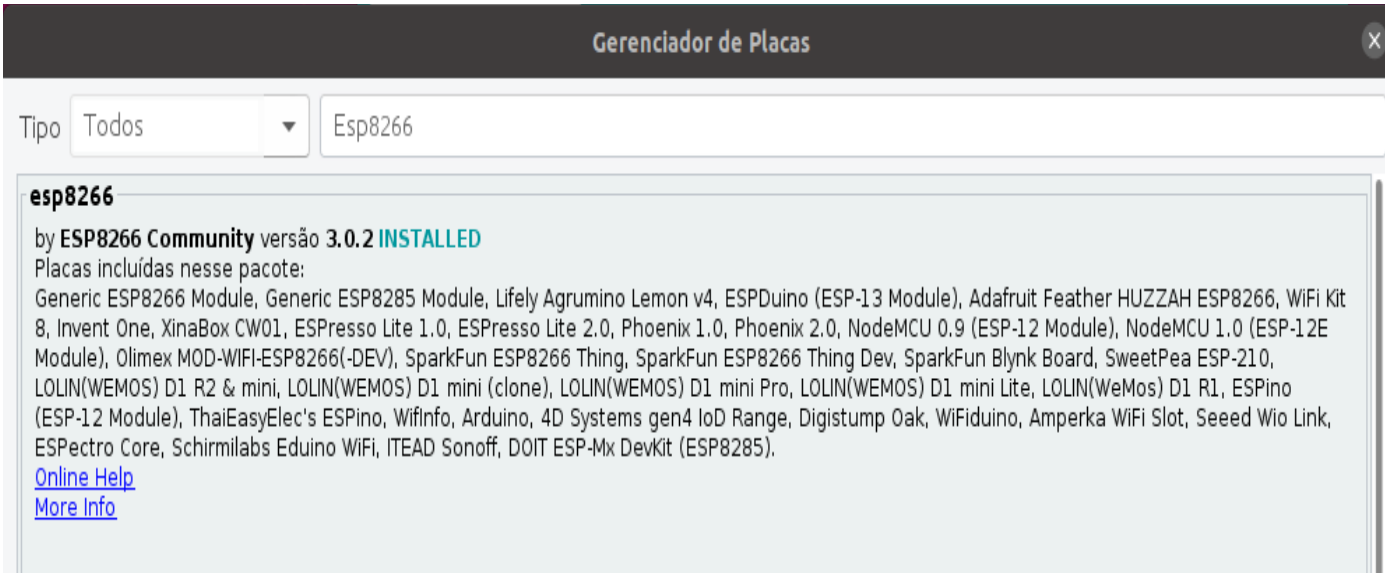
Em Arquivos > Preferências no campo “URLs Adicionais para Genrenciadores de Placas” utiliza-se o link “http://arduino.esp8266.com/stable/package_esp8266com_index.json “ conforme demonstrado na figura 6.

Figura 6 - Adição da biblioteca ESP8266



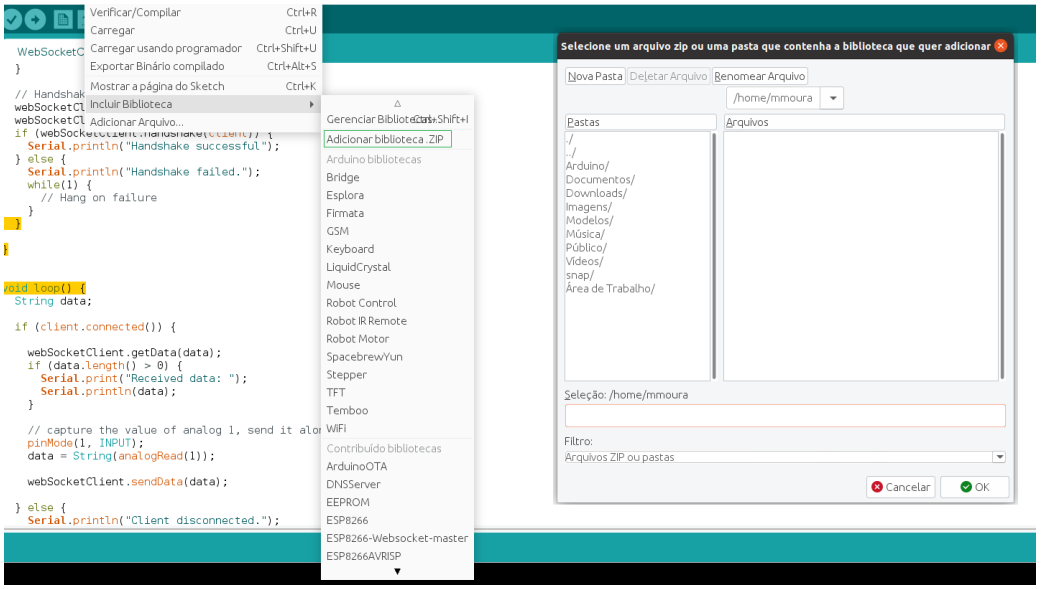
Em Ferramentas > Placa > Gerenciador de Placas pesquisamos por “ESP8266” e seleciona-se a biblioteca ESP8266 by ESP8266 Community conforme demonstrado na figura 7.

Figura 7 - Instalação dos drivers e bibliotecas do ESP8266



Foi instalada outra biblioteca, baixando um zip no GitHub. Na IDE, Sketch > Incluir Biblioteca > Adicionar Biblioteca .zip onde seleciona-se o arquivo baixado conforme demonstrado na figura 8.

Figura 8 - Adicionando biblioteca websocket



Para o desenvolvimento do Back-End e Front-End foi necessário instalar a IDE Visual Code, sem configurações adicionais necessárias.
Para a execução do código JavaScript foi necessário instalar o Node JS, sem configurações adicionais necessárias.
Para o Banco de Dados foi necessário instalar o software MySQL, sem configurações adicionais necessárias.

3.3 Implementação do código para o NodeMCU

Baseados em um modelo pronto disponibilizado pela biblioteca instalada, encontrado em Arquivo > Exemplos > ESP8266-WebSocket-Master > WebSocketClient_Demo.

Foram feitas a alteração dos valores de algumas variáveis:

const char* ssid para o nome da rede WiFi
const char* password para a senha da Rede WiFi
char* host para o endereço IP do Servidor WebSocket

O código conforme demonstrado na figura 9 foi refatorado para criar funções mais simples e reutilizáveis para os próximos procedimentos, mas que em suma não altera nada da lógica inicial do código.

Figura 9 - Código completo do NodeMCU ESP8266

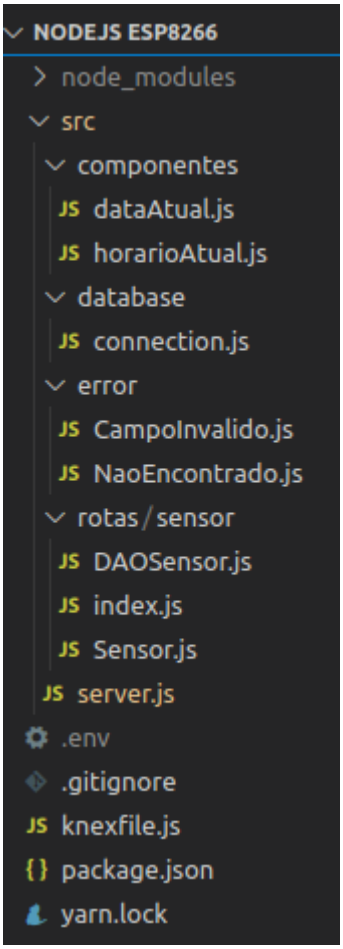
```ino
#include <ESP8266WiFi.h>
#include <WebSocketClient.h>

boolean conexaoWS = 0;
int val = 0;
String dados = "";
char path[] = "/";
const char* ssid     = "Familia Moura";
const char* password = "F4m1l14@";
char* host = "192.168.1.14";
const int espport = 3003;
unsigned long tempo;


WiFiClient client;
WebSocketClient webSocketClient;

void conecta() {
  Serial.print("Iniciando a conexão: ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("...tentando se conectar");
  }
  Serial.println("Conexão WiFi realizada com sucesso!!!");
  Serial.print("Endereço IP: ");
  Serial.println(WiFi.localIP());
  realizaConexaoWS();
}

void realizaConexaoWS() {
  if (client.connect(host, espport)) {
    Serial.println("Iniciando conexao WebSocket!!!!");
    webSocketClient.path = path;
    webSocketClient.host = host;
    if (webSocketClient.handshake(client)) {
      conexaoWS = true;
      Serial.println("Conexao WebSocket realizado com sucesso!!!!");
    } else {
      Serial.println("Houve falha na conexão WebSocket, tentaremos novamente ...");
      conexaoWS = false;
    }
  } else {
    Serial.println("Cade o servidor ???");
    conexaoWS = false;
  }
}


void setup() {
  Serial.begin(9600);
  delay(10);
  conecta();
  delay(500);
  tempo = millis();
}

void loop() {

  if (client.connected()) {
    if ((millis() - tempo) >= 1000) {
      if (conexaoWS == true) {
        Serial.println("chegou aqui!!! depois do conexaoWS");
        val = analogRead(A0);
        val = map(val, 0, 1023, 200, 10000);
        dados = "nome'Sensor da Caldeira'valor'" + (String)val + "'unidade'ppm";
        Serial.println(dados);
        webSocketClient.sendData(dados);//send sensor data to websocket server
        tempo = millis();
      }
    }
  }
}
```

```ino
WebSocketClient_Demo
#include <ESP8266WiFi.h>
#include <WebSocketClient.h>

const char* ssid     = "SSID HERE";
const char* password = "PASSWORD HERE";
char path[] = "/";
char host[] = "echo.websocket.org";

WebSocketClient webSocketClient;

// Use WiFiClient class to create TCP connections
WiFiClient client;

void setup() {
  Serial.begin(115200);
  delay(10);

  // We start by connecting to a WiFi network

  Serial.println();
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  delay(5000);


  // Connect to the websocket server
  if (client.connect("echo.websocket.org", 80)) {
    Serial.println("Connected");
  } else {
    Serial.println("Connection failed.");
    while(1) {
      // Hang on failure
    }
  }

  // Handshake with the server
  webSocketClient.path = path;
  webSocketClient.host = host;
  if (webSocketClient.handshake(client)) {
    Serial.println("Handshake successful");
  } else {
    Serial.println("Handshake failed.");
    while(1) {
      // Hang on failure
    }
  }

}


void loop() {
  String data;

  if (client.connected()) {

    webSocketClient.getData(data);
    if (data.length() > 0) {
      Serial.print("Received data: ");
      Serial.println(data);
    }

    // capture the value of analog 1, send it along
    pinMode(1, INPUT);
    data = String(analogRead(1));

    webSocketClient.sendData(data);

  } else {
    Serial.println("Client disconnected.");
    while (1) {
      // Hang on disconnect.
    }
  }

  // wait to fully let the client disconnect
  delay(3000);

}
```

3.4 Implementação do código para o Back End

Inicialmente é utilizado o comando “npm init” no console/terminal que irá criar o arquivo package na raiz do projeto. A estrutura de pastas ficará conforme demonstrado na figura 10.

Figura 10 - Estrutura de pastas e arquivos da API



Após isso realiza a configuração do arquivo .env que recebe os seguintes parâmetros para a conexão com o banco de dados

DB_USER='root' //Usuário do banco

DB_PASS='root'// Senha do usuário

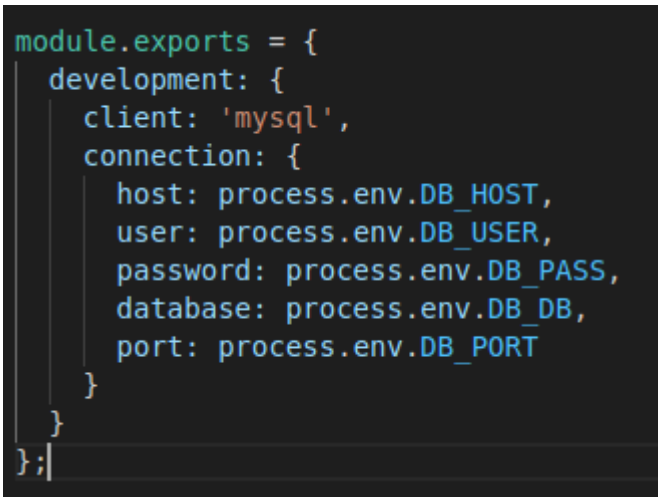
DB_HOST='localhost' // host onde está hospedado o banco de dados

DB_DB='ProjetoTCC' // nome do Banco de dados

DB_PORT='3306' // porta que está sendo executada o serviço do banco de dados

Em seguida vem a configuração do arquivo Knexfile.js que irá consumir os parâmetros do arquivo .env conforme demonstrado na figura 11.

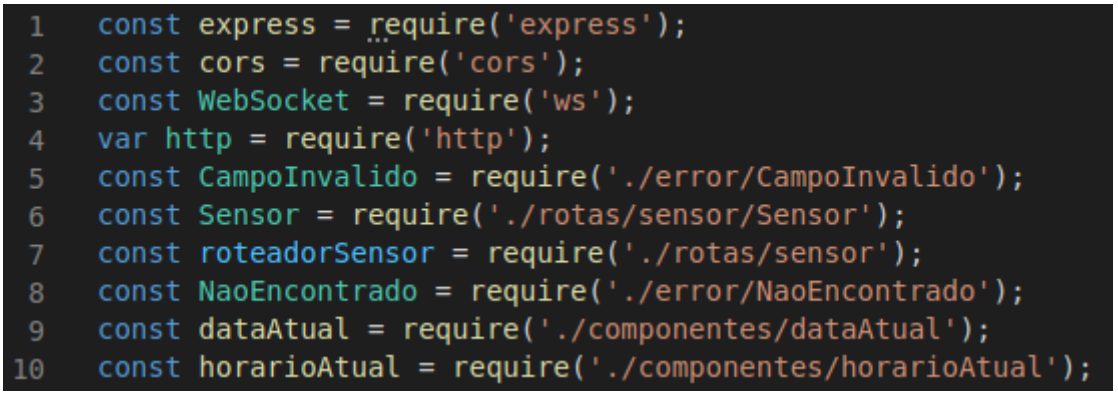
Figura 11 - Configuração do banco de dados



```
module.exports = {
  development: {
    client: 'mysql',
    connection: {
      host: process.env.DB_HOST,
      user: process.env.DB_USER,
      password: process.env.DB_PASS,
      database: process.env.DB_DB,
      port: process.env.DB_PORT
    }
  }
};
```

Criando o arquivo Server.js inicia-se a importação de algumas bibliotecas e componentes que serão utilizados conforme demonstrado na figura 12.

Figura 12 - Importações de bibliotecas e arquivos necessários



```
1   const express = require('express');
2   const cors = require('cors');
3   const WebSocket = require('ws');
4   var http = require('http');
5   const CampoInvalido = require('./error/CampoInvalido');
6   const Sensor = require('./rotas/sensor/Sensor');
7   const roteadorSensor = require('./rotas/sensor');
8   const NaoEncontrado = require('./error/NaoEncontrado');
9   const dataAtual = require('./componentes/dataAtual');
10  const horarioAtual = require('./componentes/horarioAtual');
```

Na sequência o server com a biblioteca “http” cria o WebSocket Server conforme demonstrado na figura 13.

Figura 13 - Criação do servidor WebSocket

```
const server = http.createServer(app);
require('dns').lookup(require('os').hostname(), function (err, add, fam) {
    console.log('addr: ' + add);
})
const webSocket = new WebSocket.Server({ server });
```

Utilizando o método .on do WebSocket que transmite o parâmetro 'connection', cujo funcionamento depende do Servidor WebSocket estar ligado, dentro dele foram montadas mais duas sub funções, ambas utilizam o .on com parâmetros diferentes, a primeira é ‘message’ quando o websocket recebe alguma mensagem de algum cliente e dentro dessa terá mais uma função chamada, sendMessage, que ficará responsável por enviar as informações já tratadas para os demais clientes conectados, e o ‘close’ quando algum cliente sai da conexão conforme demonstrado na figura 14.

Figura 14 - Utilização métodos do Websocket

```
webSocket.on('connection', async function (ws, req, proximo) {
    ws.on('message', async function (message) {…
    });
    ws.on('close', function () {
        console.log("cliente desconectado ...");
    });

    console.log("cliente conectado ...");
}
);
```

Por fim foram definidas as portas em que cada serviço será executado, e criamos uma rota para que o client Front end possa utilizar os dados da API conforme demonstrado na figura 15.

Figura 15 - Criação das portas e rotas da API

```
85  const portAPI = 3005;
86  const portWS = 3003;
87
88  app.use('/api/sensor', roteadorSensor);
89  app.use((erro, requisicao, resposta, proximo) => {
90      let status = 500
91
92      if (erro instanceof CampoInvalido || erro instanceof NaoEncontrado) {
93          status = 400
94      }
95
96      resposta.status(status);
97
98      resposta.send({
99          status_requisição: erro.message,
00      })
01
02  })
03
04  app.listen(portAPI, () => {
05      console.log(`A API está sendo executada na url http://localhost:${portAPI}/`);
06  });
07
08  server.listen(portWS);
09  console.log(`o WebSocket no endereço: http://localhost:${portWS}/`);
```

A rota "/api/sensor" será responsável por devolver os dados da consulta do front essa que estará rodando na porta 3005, enquanto na porta 3003 estará o serviço do WebSocket.

Na pasta componente, possui códigos essenciais para a criação do objeto sensor, como o dataAtual.js Aqui utilizamos a biblioteca Date presente no próprio JavaScript para coletar os valores necessários conforme demonstrado na figura 16.

Figura 16 - Função data atual

```
function dataAtual() {
    const datas = new Date;
    let mes = parseInt(datas.getMonth()) + 1;
    if (mes < 10) {
        mes = '0' + mes;
    }

    const atual = `${datas.getFullYear()}-${mes}-${datas.getDate()}`;
    return atual;
}


module.exports = dataAtual;
```

No caso desse componente é utilizado o .getMonth para coletar o mês, .getFullYear para o ano e getDate para o dia.

O horarioAtual.js, segue o mesmo e padrão que o componente dataAtual, utiliza a biblioteca Date, usando os métodos getHours para coletar as horas, getMinutes para os minutos e getSeconds para os segundos.

Na pasta error, o primeiro arquivo é o CampoInvalido.js essa estende a classe Error do próprio JavaScript assim podemos criar um erro customizado, recebe uma variável campo que será concatenada ao texto que irá retornar ao cliente conforme demonstrado na figura 17.

Figura 17- Classe campo inválido que extende a classe error

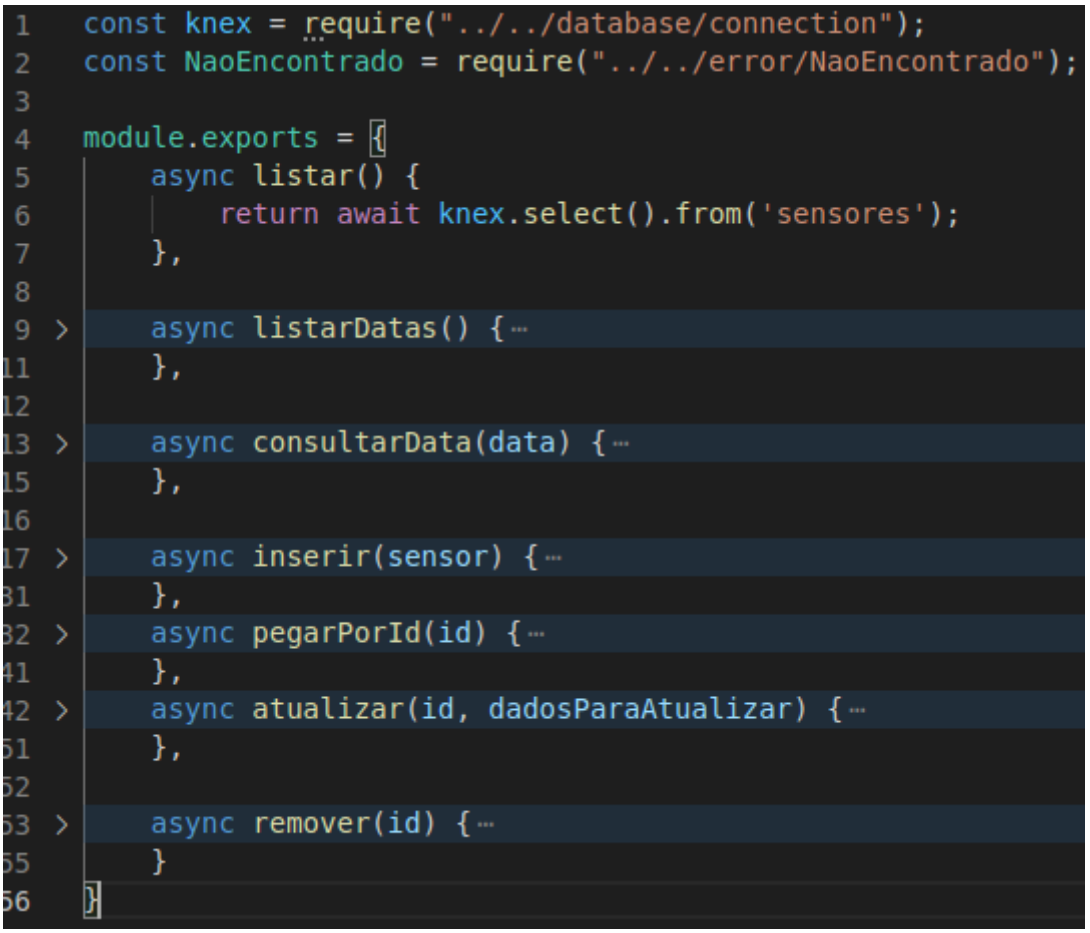
```
class CampoInvalido extends Error {
    constructor (campo) {
        const mensagem = `O campo '${campo}' está inválido, por favor corriga antes de realizar outra requisição!!!`
        super(mensagem)
        this.name = 'CampoInvalido'
        this.idErro = 1
    }
}

module.exports = CampoInvalido
```

O arquivo NãoEncontrado segue o mesmo padrão, apenas mudando a variável que recebe nesse caso é id e o texto que será passado ao cliente.

Na pasta Rotas/ sensor, o arquivo SensorDAO conforme demonstrado na figura 18 começa importando o arquivo connection e o arquivo de erro naoEncontrado, a partir deste ponto utilizando os próprios métodos do knex são criadas as querys de consulta ao banco de dados.

A primeira função é a listar() essa que realiza um return no resultado do knex.select().from('sensores'); é necessário o await antes do método do knex pois a consulta pode demorar.

Figura 18 - DAO do objeto sensor



No arquivo sensor.js conforme demonstrado na figura 19 é onde será criado o modelo de negócio e classe sensor que será utilizada durante a execução da API.

Inicia-se importando o arquivo DAOSensor, e então criando a class sensor, em seguida vem a definição do construtor, na sequência os métodos dessa classe listar() esse utiliza o método listar do DAOsensor, Criar() utiliza o método inserir, carregar() utiliza o método pegarPorId e caso não localize o id devolve o erro NaoEncontado, atualizar() esse utiliza dois métodos do DAOSensor, o pegarPorId e o Atualizar() que receberá um id e os dados para atualizar, por fim o remover() que utiliza o metodo remover() do DAOSensor.

Figura 19 - Classe Sensor

```
const DAOSensor = require('./DAOSensor');

class Sensor {
    constructor({ id, nome, valor, unidade_medida, data_criacao, horario_criacao}) {
        this.id = id;
        this.nome = nome;
        this.valor = valor;
        this.unidade_medida = unidade_medida;
        this.data_criacao = data_criacao;
        this.horario_criacao = horario_criacao;

    }

    async listar() {…
    }

    async criar() {…
    }

    async carregar() {…
    }

    async atualizar() {…
    }

    async remover() {…
    }


    validar() {…
    }

}
module.exports = Sensor;
```

No arquivo index.js, começamos criando algumas const(constantes) roteador que pegara o método Router() da biblioteca express, a class Sensor e o DAOsensor

Esse roteador que foi criado será usado para criar as rotas do sensor.

A rota get(‘/’), assim que for chamada, será utilizado o método listar do arquivo DAOServer, se tudo correr bem devolve o status (200) mais os resultados que obteve do banco.

Figura 20 - Index da rota sensor

```
const roteador = require('express').Router()
const Sensor = require('./Sensor');
const DAOSensor = require('./DAOSensor');

roteador.get('/', async (requisicao, resposta) => {
    const resultados = await DAOSensor.listar();
    filtrar(resultados);
    resposta.status(200).send(resultados);
});

roteador.get('/datas', async (requisicao, resposta) => {
    const resultados = await DAOSensor.listarDatas();
    filtrar(resultados);
    resposta.status(200).send(resultados);
});

roteador.post('/datas', async (requisicao, resposta) => {
    try {
        const dados = requisicao.body;
        console.log(dados);
        const resultados = await DAOSensor.consultarData(dados.data);
        resposta.status(200).send(resultados);
    } catch (error) {
        proximo(error);
    }
});

roteador.post('/valor', async (requisicao, resposta, proximo) => {
    try {
        const dados = requisicao.body;
        const sensor = new Sensor(dados);
        await sensor.criar();
        resposta.status(201).send(sensor);

    } catch (error) {
        proximo(error);
    }
});
```

As demais rotas seguem o mesmo padrão, com exceção da rota post(‘/valor’) essa é necessário criar um objeto Sensor e passar os dados que foi recebido, e aí sim chamar o método Criar do objeto sensor conforme demonstrado na figura 20.

3.5 Implementação do código para o Front end

O projeto é iniciado no console/terminal usando o comando npx create-react-app nome_projeto esse comando irá criar um projeto base em reactjs, após alguns arquivos desnecessários serem removidos, o .env é criado com apenas um parametro URL_API='http://localhost:3005'. A pasta public é criada automaticamente pelo comando que é usado para iniciar esse projeto. Criando a pasta SRC, o primeiro arquivo a ser criado será o Index.js conforme demonstrado na figura 21.

Figura 21 - Arquivo index

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />,document.getElementById('root')
);
```

Inicialmente o React é importado, essa biblioteca não chega a ser usada diretamente no código, mas é necessária para que o projeto funcione. A biblioteca ReactDOM, que será usada para manipular a árvore DOM do HTML. e o componente APP. Nesse arquivo após as importações de biblioteca é realizada um ReactDOM.Render(), esse método Render que está sendo utilizado, irá renderizar todo o conteúdo do componente APP dentro do HTML no elemento que carrega o id=root.

Criando o arquivo APP.js, nesse arquivo algumas importações necessárias serão feitas conforme demonstrado na figura 22.

Figura 22 - Importação de bibliotecas, componentes e estilização

```
import React, { Component } from 'react';
import { w3cwebsocket as W3CWebSocket } from "websocket";
import BarChart from './components/BarChart';
import api from './services/api';
import './styles.css';
```

Configurando a conexão via Websocket passando o endereço ip e porta conforme demonstrado na figura 23.

Figura 23 - Configuração do endereço e porta do websocket

```
const client = new W3CWebSocket('ws://127.0.0.1:3003');
```

Agora é criada a função consultaHistoria() conforme demonstrado na figura 24 essa que irá realizar uma requisição HTTP com o método POST para a api na rota /api/sensor/datas. passando no corpo da requisição um objeto no formato {data: dataEscolhida}

Figura 24 - Função responsável por realizar a consulta histórica

```
async function consultarHistorico(e) {
  e.preventDefault();

  try {
    var dado = { data: dataEscolhida };
    const response = await api.post('sensor/datas', dado);
    var dados = response.data;
    if (historico.length > 2) {
      historico = [['horario', 'valor']];
    }
    dados.forEach(element => {
      historico.push([element.horario_criacao, parseInt(element.valor)])
    });
```

A próxima função criada é a puxaHistorico() conforme demonstrado na figura 25 onde outra requisição é realizada para a rota supracitada, mas dessa vez com o método GET, com a resposta que recebida da API é criado um array chamado Datas.

Figura 25 - Função histórico de datas

```
async function puxaHistorico() {
  try {
    const response = await api.get('sensor/datas');
    var dados = response.data;
    var i = 1;
    dados.forEach(element => {
      datas.push({ id: i, value: element.data_criacao });
      i++;
    });
    console.log(datas);
  } catch (error) {

  }

}
```

E por último é criada a função verificaStatus() demonstrado na figura 26 que recebe como parâmetro um valor, de acordo com o valor recebido altera a cor e mensagem que aparece no campo Status da página.

Figura 26 - Função verificadora de status

```
function verificaStatus(valor) {
  var div = document.getElementsByClassName("status-sensor");
  var p = document.getElementsByClassName("info-status");
  if (valor >= 2000) {
    div[0].style.backgroundColor = "#E00000";
    p[0].innerHTML = "Perigo grande vazamento de gás";
  } else if (valor >= 500) {
    div[0].style.backgroundColor = "#EF6416";
    p[0].innerHTML = "niveis de gás acima do normal";
  } else {
    div[0].style.backgroundColor = "#00E031";
    p[0].innerHTML = "niveis aceitáveis";
  }
}
```

Agora será criado o componente que será de fato renderizado em tela, será gerada uma classe App que estende a biblioteca Component do React, inicialmente configura-se o componentWillMont(), essa função pertence a biblioteca Component é chamada assim que o componente é montado, Dentro dessa função o setIntervel, atualizará a página após 500 milisegundos, nesse caso irá atualizar apenas aquilo que foi alterado. Após isso, a função puxaHistorico() é chamada para que a lista de datas seja criada, e por fim a conexão com o websocket é criada para começar a ouvir as informações.

O método client.onMessage intercepta as mensagens recebidas e transformá- las em um JSON, e então inclui essa informação no Array gráfico e chama a função verificaStatus passando o valor do sensor que recebeu conforme demonstrado na figura 27.

Figura 27- Criação da classe App

```
class App extends Component {
  componentWillMount() {
    setInterval(() => this.forceUpdate(), 500);
    puxaHistorico();

    client.onopen = () => {
      console.log('WebSocket Client Connected');
    };
    client.onmessage = (message) => {
      const result = JSON.parse(message.data);
      grafico.push([result.horario, parseInt(result.valor)]);
      status1 = 'yes';
      verificaStatus(parseInt(result.valor));
    };
  }


  render() {

    return (
      <section className="container">…
      </section>
    );
  }
}

export default App;
```

O método render() é o responsável por renderizar o componente dentro dele que possui um Return(); contendo todo o HTML da página para a criação dos gráficos, tanto em tempo real quanto o de consultas passadas, o mesmo componente conforme demonstrado na figura28 é utilizado.

Figura 28 - Utilizando o componente BarChart

```
<BarChart dataMapa={grafico} title={""} corMapa={'#ffa200'} status={status2} tituloVertical={"ppm"} />
```

Esse componente foi criado na pasta componente/Barchart, dentro dessa pasta temos um arquivo Index.js, iniciando com algumas importações conforme demonstrado na figura 29.

Figura 29 - Importa as bibliotecas React e Google Charts

```
import React from 'react';
import { Chart } from "react-google-charts";
import MapaComponent from "react-component-component";
```

O componente BarChart foi criado pelo Google, que disponibiliza a biblioteca Chart para a criação de gráficos dinâmicos conforme demonstrado na figura 30, feitas algumas alterações, ele se tornou um componente reutilizável na página como a propriedade chartData, que são os dados que o gráfico recebe, tornando-se uma variável dinâmica.

Figura 30 - Alterações no componente chart da biblioteca Google Charts

```
export default function BarChart(props) {


    return (
        <GraficoComponent
            initialState={
                { dataLoadingStatus: 'loading', chartData: [] }
            }
            didMount={

                async function (component) {
                    component.setState({
                        dataLoadingStatus: props.status,
                        chartData: props.dataMapa,
                    })
                }
            }
        >
            {component => {
                return component.state.dataLoadingStatus === 'ready'  ? (
                    <Chart
                        width={'75vw'}
                        height={'70vh'}
                        chartType="Line"
                        loader={<div>Carregando o grafico</div>}
                        data={component.state.chartData}
```

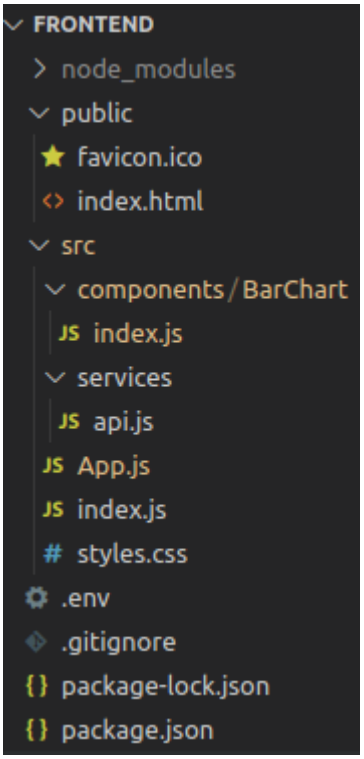
Criando a pasta services, que possuirá apenas um arquivo, o API.js será importada a biblioteca AXIOS que servirá para realizar as requisições HTTP conforme demonstrado na figura 31

.Figura 31 - Utilização da biblioteca Axios para conexão com a API

```
import axios from 'axios';
require('dotenv').config();

const api = axios.create({
    baseURL: process.env.URL_API,
})

export default api;
```

Por fim o frontend terá a seguinte estrutura conforme demonstrado na figura 32.

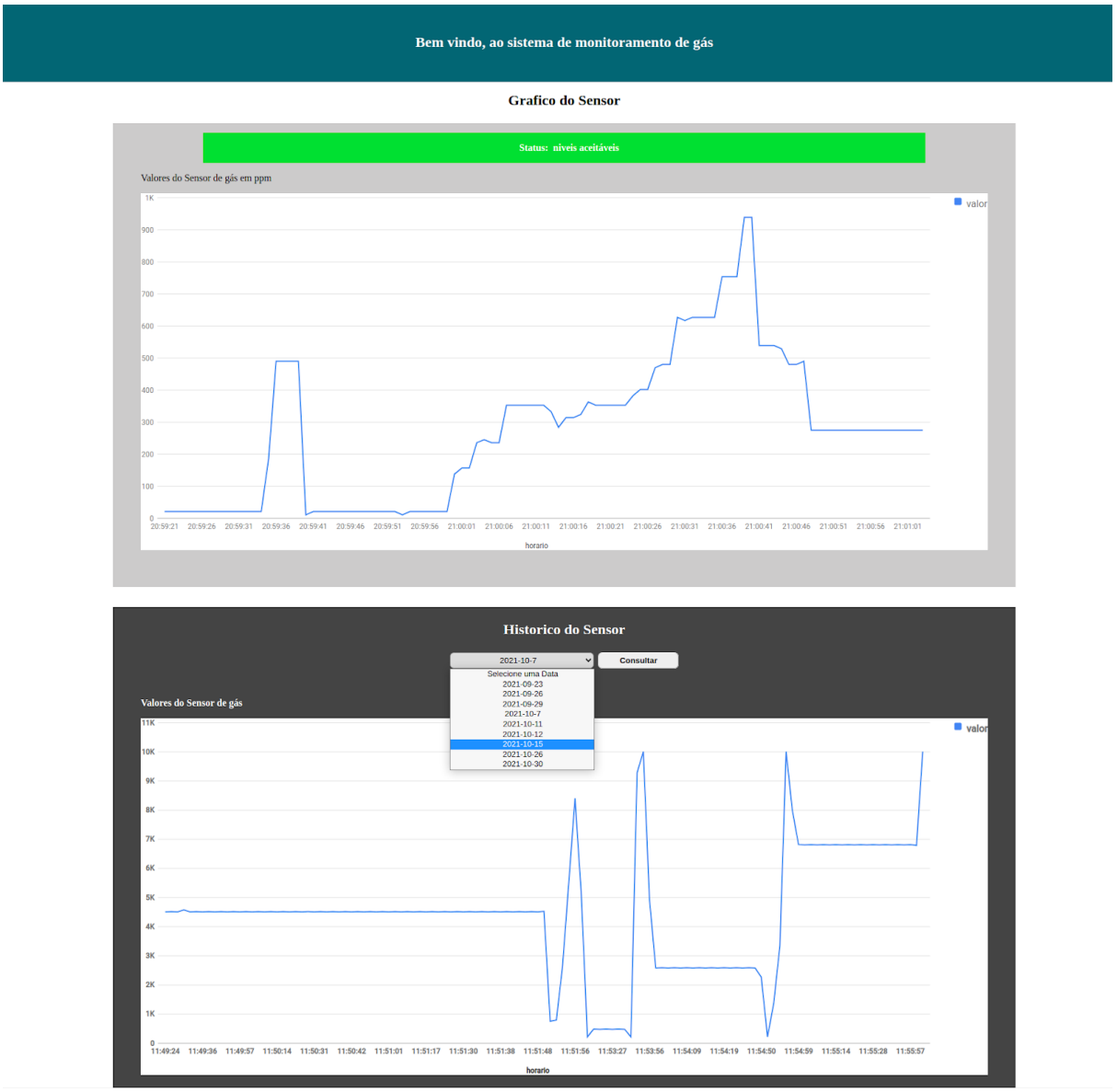
Figura 32 - Estrutura de pastas do Frontend



4 RESULTADOS

Analisando o resultado final conforme demonstrado na figura 33, vemos algo satisfatório, a criação de uma plataforma online onde o usuário possa monitorar um sensor remotamente.

Figura 33 - Site apresentando gráfico em tempo real e consulta no histórico



Na parte do front-end o ReactJs possibilitou a criação de gráficos em tempo real, consumindo informações da conexão WebSocket, e realizar consultas no banco de dados através de requisições HTTP. Já o Node JS se mostrou eficiente e satisfatório para a criação da API, que pode abrir uma conexão webSocket, receber dois clientes em sua comunicação, persistir os dados de um cliente (NodeMCU) no banco de dados e disponibilizar em tempo real um objeto com as informações do

sensor para o segundo cliente (ReactJS), tudo isso através da conexão webSocket, e liberar rotas HTTP para que possa consultar os dados que foram persistidos no banco de dados. No NodeMCU ESP8266 foi possível estabelecer uma comunicação WiFi de forma fácil, já que o mesmo já possui um módulo WiFi integrado e facilita esse processo, a comunicação webSocket acabou sendo mais complicada por quesitos de configurações, mas funcionou melhor do que o esperado.

De forma geral, a conexão WebSocket foi o que tornou possível esse monitoramento em tempo real, sem ela seria complicado gerar gráficos ou manter a comunicação do microcontrolador com a API, as requisições via HTTP são eficientes, porém são requisições que tornam obrigatório o ciclo de gerar uma requisição a cada nova atualização, então se o frontend conectado com backend fazendo requisições HTTP em tempo real, o gráfico de consulta e o microcontrolador conectado com backend fazendo requisições HTTP para enviar as informações do sensor, acontecendo simultaneamente, ocorrerá um congestionamento na API, com inúmeras requisições chegando, muitas vão acabar sem respostas ou até num pior cenário, derrubando a API.

Com o WebSocket fazendo esse papel, essas informações podem trafegar entre o server e os clients, enquanto a requisição HTTP fica disponível para consulta de dados, esses que podem ser de um dia inteiro de monitoramento, o que se torna uma consulta pesada pelo número de registros, essa consulta certamente irá demorar alguns segundos mas não causará impactos em na comunicação WebSocket, assim mantendo o gráfico em tempo real.

5 CONCLUSÃO

5.1 Considerações finais

Neste trabalho foi desenvolvido um sistema composto por uma API e um dispositivo, capazes de detectar GLP sendo monitorado remotamente por um usuário em seu terminal. Através da arquitetura IoT providenciada pelo NodeMCU com seu módulo ESP 8266, foi possível utilizar o dispositivo remotamente recebendo integralmente e em tempo real os dados gerados pelo mesmo.

A intenção deste projeto foi construir tal dispositivo com a missão de ser um item com a capacidade de prover segurança para residências e ambientes industriais. Apesar de na fase de testes um potenciômetro ter sido utilizado no lugar do sensor MQ-9, por motivos de segurança e praticidade, concluímos que o sistema funciona e é capaz de detectar GLP.

5.2 Propostas de melhoria posteriores

Diversas ideias foram propostas para melhorias no projeto durante seu desenvolvimento, dentre elas as mais relevantes foram:

A criação de uma “tela” de login onde os proprietários do sensor poderiam se cadastrar e vincular seus sensores ao site após sua instalação. Tal melhoria permitiria que múltiplos usuários acessassem seus sensores simultaneamente.

A opção de detecção de outros gases através do sensor MQ-9, realizando uma re-configuração do sensor através da plataforma de acordo com a necessidade do cliente.

A vinculação do sensor a um sistema de controle para válvulas de gás, para que o usuário possa controlar remotamente seu sistema de gás.

O aumento da quantidade de sensores operando simultaneamente.

Referências

T. ARPITHA, DIVYA KIRAN, V.S.N SITARAM GUPTA, PUNITHAVATHI DURAISWAMY. **FPGA-GSM based gas leakage detection system**, 2016. Disponível em: <https://ieeexplore.ieee.org/document/7838952/authors>.

ARTHUR HORMAN MEDEIROS CORREIA, TACIO VINICIUS MARINHO DOS SANTOS**. Sistema de proteção contra vazamento de gás inflamável e acúmulo de fumaça em ambientes fechados**, 2019. Disponível em: <https://semanaacademica.org.br/system/files/artigos/sistema_de_protecao_contra_vazamento_de_gas_inflamavel_e_acumulo_de_fumaca_em_ambientes_fechados_0.pdf>.

AMATUL MUNAZZA, TARUN KUMAR REDDY, RUPA TEJASWA, Mrs. SARANGA MOHAN. **IOT Based Gas Leakage Monitoring System Using FPGA**, 2020. Disponível em: <https://www.xajzkjdx.cn/gallery/59-may2020.pdf>.

IADY LORENZONI DA SILVA. **Desenvolvimento de uma Estufa Automatizada Baseada em IOT para uso Residencial**. 2020. Trabalho de Conclusão de Curso (Engenharia Elétrica). Universidade Regional do Nordeste do Estado do Rio Grande do Sul Departamento de Ciências Exatas e Engenharias Curso de Engenharia Elétrica, 2020.

JEFFERSON SILVA SANTOS. **Detector de Vazamento de Gás com Aviso por SMS**. 2012. Trabalho de Conclusão de Curso (Engenharia de Computação). Centro Universitário de Brasília Faculdade de Tecnologia e Ciências Sociais Aplicadas Curso de Engenharia da Computação, 2012.

RAFAEL SCHEFFER VARGAS. **Sistemas Embarcados: Acoplamento do Soft-Core Plasma ao Barramento OPB de um POWERPC 405**. 2007. Trabalho de Conclusão de Curso (Ciências da Computação). Universidade Federal de Santa Catarina, 2007.

FERNANDO LUIS BASSO. **Um Estudo Experimentativo e Comparativo da Plataforma Node.JS**. 2014. Trabalho de Conclusão de Curso (Tecnologia em Sistemas para Internet). Instituto Federal Sul-Rio-Grandense, *Campus* Passo Fundo, 2014.

ALEXANDRE CECHINEL. **Avaliação do framework Angular e das bibliotecas React e Knockout para o desenvolvimento do Frontend de aplicações Web**. 2017. Trabalho de Conclusão de Curso (Sistemas de Informação). Universidade Federal de Santa Catarina Departamento de Informática e Estatística Curso de Sistemas de Informação, 2017.

ESDRAS ALVES, LUIS EDUARDO PIRES DE LIMA. **DESSG Detector Elétrico de Segurança de Sistemas a Gás**. 2020. Trabalho de Conclusão de Curso (Eletrotécnica). Escola Técnica Estadual Frederico Guilherme Schmidt, 2020.

BERNARDO GUIMARÃES HARDUIM SANT´ANNA, LUIZ VINICIUS DA SILVA CAVALCANTI. **Automação residencial com *NodeMCU***. 2018. Trabalho de Conclusão de Curso (Engenharia de Telecomunicações). Universidade Federal Fluminense, 2018.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **ABNT NBR 6023:** Informação e documentação - Referências - Elaboração. Rio de Janeiro: ABNT, 2018.

REPOSITÓRIO DE CONFIGURAÇÕES PARA O ESP8266-Websocket
https://github.com/morrissinger/ESP8266-Websocket

APÊNDICE A

O código na íntegra pode ser acessado no repositório do gitHub https://github.com/0Z3R00/ttc_eng_comp