INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Department of Engenharia de Electrónica e Telecomunicações e de Computadores

IoT System for pH Monitoring in Industrial Facilities

47185 : Miguel Agostinho da Silva Rocha (a47185@alunos.isel.pt)
47128 : Pedro Miguel Martins da Silva (a47128@alunos.isel.pt)

Report for the Curricular Unit of Project and Seminar
of Bachelor's Degree in Computer Science and Software Engineering

Supervisor: Doctor Rui Duarte

06 - 2023

Abstract

In Germany there are several heavy restrictions on the discharge of waste resulting from manufacturing activities into nature, Therefore, our group has teamed up with the German company ”Mommertz” which manufactures boiler filters to regulate the acidity of the boiler discharge liquid before it is released into the public sewage system. The issue at hand is that, over time, these filters undergo wear and tear, leading to a decline in their original filtering capacity, necessitating regular servicing. Currently, the company’s strategy entails advising customers to periodically test the pH of the filter unit, and if the pH exceeds a specific threshold, maintenance will be required for the filter. In this project, we propose a solution to monitoring these pH filters. This solution is based on an embedded system and server-side computation. We choose which components to use to make each sub-system, and at the present moment we have a system that can read pH of the environment in the filter and can share that data to a central server which will process the data and send a signal to the manager of the device, via email, indicating that a value surpassed a threshold.

Table of Contents

List of Figures

List of Tables

1
Introduction

This document describes the implementation of the IoT System for pH Monitoring in Industrial Facilities project, developed as part of the final project in the Bachelor in Computer science and engineering course.

1.1 Motivation

In Germany, there are strict laws for the discharge of water into the sewage. This is duo to the protection of residual waters, and the sewage pipes, that conduct them. Any entity that posses condensing boilers, is obligated to neutralize the pH of water before discharging it into the public sewage system.

In the course of our final project, we collaborated with a German company called Mommertz GmbH[1]. The main aim of Mommertz's business model is to produce specialized filters designed to neutralize the pH of water resulting from the action of heating systems, thus ensuring compliance with legal obligations.

The filter comprises granules that serve to neutralize the pH of the water, including the water flowing from the boiler and passing through the filter. This system, like any industrial product, may suffer from several problems during its lifetime. As the water flows and undergoes neutralization, the granules gradually decompose, resulting in a reduction in the effectiveness of neutralization. Consequently, the pH of the water will not be fully neutralized.

Conversely, if an excessive amount of granules is used, the pH of the water may become excessively high. Additionally, there can be water leaks over time and or the water can, unexpectedly, cease flowing. Therefore, it will require maintenance, currently, all the inspection for possible problems in the mechanism is done manually, which has significant drawbacks because these kinds of systems are frequently located in challenging-to-access locations.

Additionally, manual inspections result in needless maintenance and are prone to human error. This heavy human dependency in the inspection for a possible malfunction in the behaviour of the mechanism is what our system aims to fix.

The very nature of this problem seems a very good candidate for the use of IoT technology in order to automate the device inspection process. Using sensors to monitor physical aspects of the neutralization system, we can, not only, extract key information about the device, but also notice the owner of the device when the next maintenance will be needed.

1.2 Objectives

Our project seeks to implement IoT technology for automation: The project aims to leverage **Internet of Things (IoT)** [2] technology to automate the monitoring and maintenance of the filtration system.

By integrating a microcontroller unit (MCU)[3] with sensors, the system will gather data on key filter variables and transmit it to a central server.

Establish a robust central server as the backbone of the system: One of the key objectives of our project is to develop a robust central server that serves as the backbone of the entire system. This central server will play a crucial role in storing, processing, and managing the collected data from the filtration system. It will have multiple functionalities:

- **Data storage and analysis**: The central server will include data storage mechanisms such as databases to efficiently store and manage the collected data. It will also employ analytics to process and analyse the data, extracting meaningful insights and patterns.
- **Broker functionality**: The central server will incorporate a broker component that acts as an intermediary between the server and the microcontroller unit (MCU). This broker facilitates seamless communication and data exchange between the MCU and the server, ensuring reliable and efficient transmission of information.
- **Web API for system integration**: To enable seamless integration with other systems and applications, the central server will expose a *Web API*[4]. This API will allow different systems to consume the collected data and access specific functionalities provided by the server. It will provide a standardized interface for easy and secure interaction with the system.

Enable remote monitoring and data visualization: The development of a user-friendly web application serves as a crucial objective in this project. It will allow users to remotely access and visualize the collected data from the filtration system. This interface will provide easy interaction and comprehensive insights into the system's performance, enabling efficient decision-making and proactive maintenance.

To ensure the project's value, competitiveness, and appeal, several requirements need to be fulfilled. Here are the guidelines we have agreed upon to steer the project in the right direction:

1. Cost-effective integration of the microcontroller unit into the existing product.
2. Prolonged battery life for minimal maintenance requirements.
3. Comprehensive documentation for user-friendly operation and understanding.
4. Clean and maintainable code for easy unit testing.
5. pH data reading from the filter and sharing it with a server.

6. Notification to the filter owner or manager if any value exceeds a set threshold.
7. Intuitive and user-friendly website design for easy navigation and access to information.

1.3 Document Organization

This section provides an overview of the organization of the chapters of this report, outlining the main topics and sections covered.

The document organization ensures a logical flow of information and facilitates understanding of the project's scope, objectives, and findings.

In the **first chapter**, it is provided information about the problem we are trying to solve and how our project solution is an interesting way to approach it, we also state the overall project objectives and the requirements we proposed to follow.

The **second chapter** provides important contextual information. It lays the groundwork for comprehending the project's strategies and technologies used, as well the decisions taken to choose each element of the system.

The purpose of the **third chapter** is to provide an in-depth explanation of the system architecture, dividing the problem in different modules and make a clear explanation of how each part of the problem was approached and the challenges we encountered during the implementation process.

2 Background

In this chapter will be provided the necessary information for the understanding of the core elements and technologies of our proposed architecture, as well as the analysis of the different alternatives for each component selected. Additionally, the chapter will offer general insights into IoT (Internet of Things) technology, providing a broader understanding of its principles and applications.

2.1 IoT Systems

The Internet of Things (IoT) refers to the vast network of interconnected devices, objects, and systems that are capable of collecting, exchanging, and sharing data over the internet. Relative to the physical elements, one key aspect of IoT systems is the deployment of sensors, they are responsible for collecting data from your surroundings. The collected data is transmitted to a central system or cloud platform for processing, storage, and analysis. That central system is responsible to centralize data management, process incoming sensor data, and facilitate communication and coordination between connected devices. With this architecture we can create a system that enable seamless communication between physical devices, collect and analyse data from those devices, and utilize the insights gained to improve efficiency, automation, and decision-making in various domains such as healthcare, transportation, manufacturing among others.

2.1.1 Sensors

To achieve our solution, we needed to implement several sensors. The **pH measurement** circuit uses a *TLC4502* operational amplifier to provide an analogue output for pH measurement. The pH probe oscillates between positive and negative voltages, and the circuit includes a voltage divider and a unity-gain amplifier to ”float” the pH probe input and produce a voltage output proportional to the pH value. This sensor plays a crucial role in our system as it is responsible for analysing the pH of the liquid [5]. It serves as a tool for evaluating the productivity of the neutralization mechanism, providing valuable insights into its performance.

The **ambient temperature and humidity measurement** were carried out by sensor DHT11. This sensor is also factory calibrated, making it easy to interface with other microcontrollers. Moreover, it is capable of measuring temperatures ranging from 0 °C to 50 °C and humidity from 20% to 90%, providing an accuracy of ±1 °C and ±1%. The values of the temperature and humidity are outputted as serial data by the sensor's 8-bit microcontroller[6]. This sensor provides knowledge about the environment in which the mechanism is working.

The DHT11 Sensor is factory calibrated and outputs serial data, making it highly easy to set up. The connection diagram for this sensor is shown below.

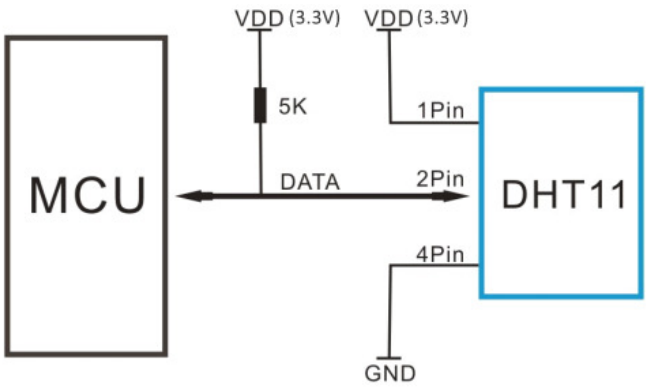


Figure 2.1: DHT11 Sensor Circuit

As shown in the Figure 2.1, the data pin is connected to an I/O pin of the MCU and a 5K pull-up resistor is used. This data pin outputs the value of both temperature and humidity as serial data.

The output given out by the data pin will be in the order of 8bit humidity integer data + 8bit the Humidity decimal data +8 bit temperature integer data + 8bit fractional temperature data +8 bit parity bit. To request the *DHT11* module to send data, the I/O pin has to be momentarily made low and then held high, as shown in the Figure 2.2.

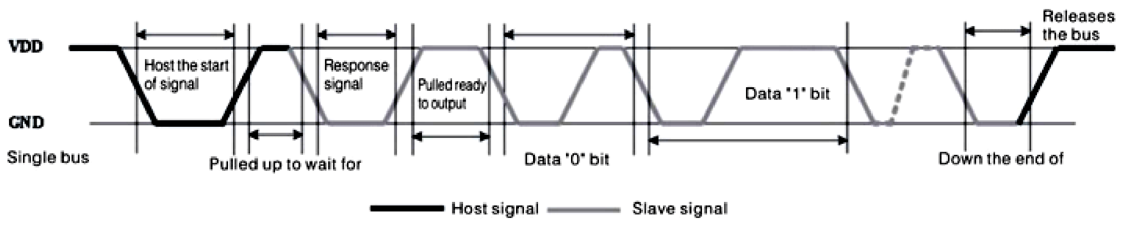


Figure 2.2: DHT11 Data Timing Diagram

To detect **water leaks** in the filter, we chose a simple water detector sensor that would be attached to the valve of the mechanism, it has a 3-pin module that outputs an analogue signal that indicates the approximate depth of water submersion. Overall, the more water it gets, the more conductivity and the voltage increases. When the sensor is dry, it outputs zero voltage.The principal criteria for the selection of this sensor was the fact that it has the right shape for our filter.

2.1.2 Embedded Systems

Embedded systems are compact and energy-efficient computers that are integrated into various mechanical or electrical systems. These systems are characterized by their low cost and power consumption. Typically, an embedded system consists of a processor, power supply, memory, and communication ports. These communication ports facilitate the exchange of data between the processor and peripheral devices[7].

MCU

The primary requirement for selecting the MCU was its compatibility with integrating a Wi-Fi module or having built-in Wi-Fi capabilities, as the device needed to transmit data over the internet. One requirement of this project was the overall low cost of the system, since we are imitating an industrial project as closely as possible. The battery consumption of the selected microcontroller was another requirement, as the device's ability to successfully automate the water inspection process relied on its ability to operate for an extended period of time. During this phase, we encountered a constraint related to the time-sensitive nature of the project, which limited our choice of microcontrollers (MCUs) available in the market since they could not be obtained within the required timeframe.

The main options we took into account were part of the *espressif* family: the *ESP32 - S3*, *ESP32 - S2*, *ESP8266EX*, *ESP32 - C3* microcontrollers. However, we also considered MCUs from other manufacturers, such as the MSP430FR413x from *Texas Instruments*. The following tables reflect the main criteria under analysis:

Microcontroler Comparison				
	ESP32 - S3	ESP32 - S2	ESP8266EX	ESP32 - C3
Active Mode (mA)	91 - 340	68 - 310	56 - 170	95-240
Modem Sleep (mA)	13.2 - 107.9	10.5 - 32.0	15	18 -28
Light Sleep (uA)	240	750	900	130
Deep Sleep (uA)	7-8	20 -190	20	5
Power Off (uA)	1	1	0.5	1
Wifi	yes	yes	yes	yes
Bluetooth	yes	yes	yes	yes
GPIO	34	43	17	22
CPU	dual-core 160MHz to 240 MHz	single-core 240 MHz	single-core processor that runs at 80/160 MHz	single-core 160 MHz
Price (€)	14.1	7.52	————	8.46

Table 2.1: Comparison Between different MCUs from espressif family, prices consulted from Mouser Online Store at 10/03/2023

Microcontroler Comparison	
	ESP32 - S3
Active Mode (mA)	126 uA / MHz
LPM0	158 - 427 uA (20 uA / MHz)
LPM3	1.25 - 1.99 uA, 25ºC (1.2uA)
LPM4	0.57 - 0.75 uA, 25ºC (0.6uA without SVS)
LPM3.5	0.70 - 1.25 uA, 25ºC (0.77uA with RTC only)
LPM4.5	0.013 - 0.375 uA, 25ºC
SHUTDOWN	13 nA
Wifi	————
Bluetooth	————
GPIO	14.1
Price (€)	17.49

Table 2.2: MSP430FR413x analysis, price consulted from Mouser Online Store at 10/03/2023

During the evaluation process, we primarily considered three key factors for selecting the board. First, the presence of a built-in Wi-Fi module was crucial as the board needed to connect to Wi-Fi networks. This feature ensured seamless wireless connectivity for our project.

Secondly, we paid close attention to the number of *GPIO* pins available on the board. This factor was vital for the future expandability of the project, allowing us to connect a sufficient number of sensors and peripherals as needed.

Lastly, we focused on the power consumption during *deep sleep* mode, as this state would be the board's primary operational mode for extended periods[8]. Additionally, we considered the power consumption during active mode, which would occur when the board collected data from the sensors and transmitted it. These power consumption figures received significant attention during our evaluation, as we aimed to optimize the energy efficiency of the board during its most common operating states.

By carefully considering these factors, we were able to make an informed decision regarding the selection of the board for our project.

The **ESP32 - S3** is a high-end MCU that offers an impressive array of features, including a built-in Wi-Fi module, low power consumption, and a powerful processor [9]. However, it comes with a relatively high price tag. While the ESP32 - S3 provides a wide range of possibilities for our project, one of the requirements is to maintain awareness of the final product's cost-effectiveness. Considering the overall economy of the project, we opted against choosing the ESP32 - S3 due to its higher cost.

The **ESP32-S2**, although it may have a slightly weaker processor compared to the ESP32-S3, fulfils our project requirements adequately [10]. Its processing power is more than sufficient for the tasks at hand, eliminating the need for unnecessary expenses.

The **ESP8266EX** board stands out as a choice for our project due to its exceptionally low power consumption, despite having a slightly less powerful processor compared to the other options[11]. Its built-in Wi-Fi capability is also a valuable feature. One drawback of the ESP8266EX is its limited number of GPIO (General Purpose Input/Output) pins. This limitation could potentially restrict the number of sensors that can be added to the project, thereby limiting its expandability in the future.Although we were unable to select this MCU for our project at this stage due to stock shortage, it is worth mentioning as a notable competitor that we considered.

The **ESP32 - C3** is a noteworthy contender for our project. It has a combination of features, including low power consumption, a built-in Wi-Fi module, and similar characteristics to the ESP32-S2 [12]. Although it runs on a slightly less powerful CPU, it delivers excellent deep sleep consumption. However, it is important to note that the ESP32-C3 is slightly more expensive compared to the ESP32-S2. Despite this cost difference, its overall performance and feature set make it a strong candidate for consideration in our project.

The **MSP430FR413x** microcontrollers has superior power consumption, making it an energy-efficient choice. However, it lacks a built-in Wi-Fi module, which presents a challenge [13] . To incorporate Wi-Fi functionality, an additional purchase and installation would be required, adding complexity and potentially increasing costs. Considering our project's time constraints and the readily available access to the ESP32-S2, it was important to choose a solution that offered convenience and efficiency.

Given the project's time constraints and the readily available access to the ESP32-S2, coupled with the high price and additional requirements of the MSP430FR413x line-up, we made the decision to prioritize convenience, security, and cost-effectiveness. The ESP32-S2 emerged as the most suitable and practical option for our project, ensuring smooth implementation without compromising essential features. Figure 2.3 shows the version of the chosen microcontroller.

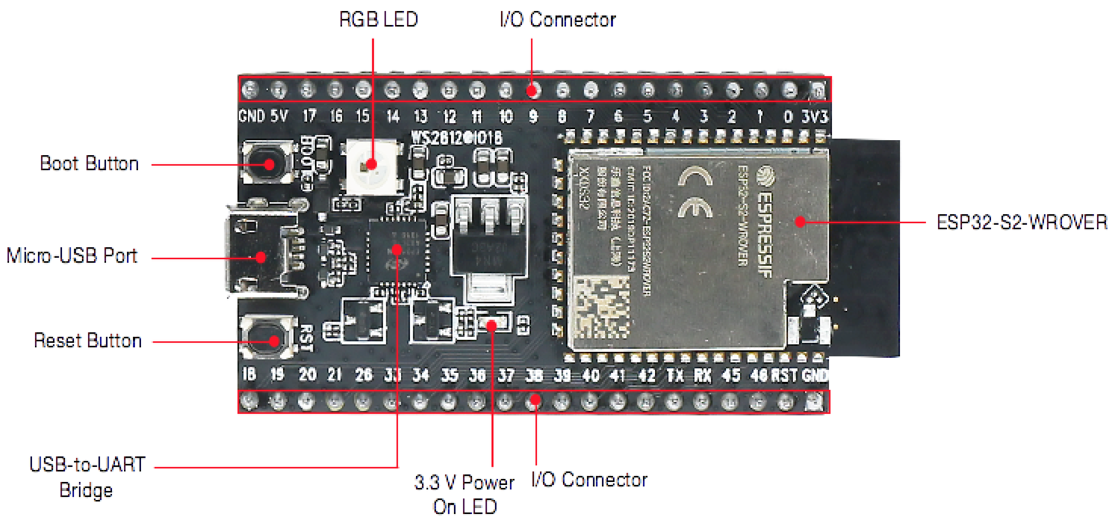


Figure 2.3: Espressif ESP32-S2-SAOLA-1M (WROOM)

Framework

We made the decision to utilize the **ESP-IDF** (Espressif IoT Development Framework) framework, the official ESP's devices programming framework, instead of the more commonly used and less versatile Arduino and PlatformIO frameworks for programming the MCU.

This choice was motivated by the project's requirement for an economic power device. The ESP-IDF framework provides us with direct access to the underlying hardware and peripherals, giving us more control and flexibility over the microcontroller. This allows us to optimize performance in a variety of tasks.

Another advantage is the documentation freely available. On the other hand, many tasks, for instance connecting to the Wi-Fi or interacting with the sensors, becomes a more complex task duo to the non-existent abstraction that other frameworks provide.

2.1.3 Backend

In an IoT system, the backend serves as the central component responsible for managing and processing the data collected from connected devices. The main functions of the backend in an IoT system include:

Technology Stack

In the backend development of our IoT project, we adopted a robust technology stack consisting of the Spring framework and the Kotlin programming language. In the backend development of our IoT project, we adopted a robust technology stack consisting of the Spring framework and the Kotlin programming language.

The **Spring framework** served as the foundation of our backend implementation. We leveraged its comprehensive features, including dependency injection, MVC architecture, transaction management, and seamless integration with other components[14].

The use of this framework provided us with a scalable and flexible infrastructure for building our backend services.

Kotlin, a modern and expressive programming language, was chosen as the primary language for our backend development[15]. Its concise syntax, null safety, interoperability with Java, and rich standard library proved to be valuable assets in our project. Kotlin allowed us to write clean and readable code, enhancing our productivity and reducing the likelihood of errors.

Due to our prior exposure to and experience with both of these technologies throughout our course, we have developed a strong familiarity with them. This familiarity significantly influenced our decision to incorporate Spring and Kotlin into our project's tech stack.

Database

To store the data we decided to separate the data in two sections: The **static data**, that refers to the portion of data within a database that remains constant and does not change frequently or during the operation of a system, such as information about the users, error logs and associated devices. The main considerations in selecting a technology for storing this type of data were scalability and support for intermediate querying capabilities. *PostgreSQL* emerged as an ideal choice that fulfilled these criteria. Furthermore, our prior exposure to *PostgreSQL* during our course expedited the development process of this module, enhancing overall time efficiency.

The **sensor data**, which is derived from the continuous collection of data by the device's sensors, such as pH values, is characterized by its time-stamped nature and represents a continuous stream of values or events. With this design in mind, we chose to utilize a time series database.

Time series databases are specifically designed to store and retrieve data records associated with timestamps (time series data), due to the uniformity of time series data, specialized compression algorithms can provide improvements over regular compression algorithms designed to work on less uniform data. They offer faster querying and can handle large volumes of data[16][17][18]. However, it's important to note that these databases may not perform as effectively in domains that do not primarily work with time series data.

We chose to utilize the *InfluxDB* time series database for our project, which is widely recognized as an excellent option for storing sensor data in Internet of Things applications. *InfluxDB* offers a range of advantages that we previously discussed, and it seamlessly integrates with Kotlin, the server-side language we have adopted. One disadvantage of opting for the open-source version of *InfluxDB* technology is its lack of support for horizontal scaling in the free version. To keep the project cost-effective, we had to make the sacrifice of not being able to utilize horizontal scaling capabilities provided by this technology.

2.1.4 Communication protocols

After careful consideration, we opted for the **MQTT** protocol to allow the communication between the device and the server. MQTT is extensively utilized in the IoT industry due to its exceptional lightweight and efficient nature. This efficiency enables devices to conserve energy more effectively in comparison to other protocols such as HTTP[19][20]. MQTT achieves this by employing a smaller packet size and lower overhead than its counterparts. To allow the communication between the web app client, we used the HTTP protocol. Our application provides an API (Application Programming Interface) that allow other applications or services to interact with them. APIs are typically based on HTTP and use various HTTP methods (such as GET, POST, PUT, DELETE) to perform operations like retrieving data, creating resources, updating resources, or deleting resources. API clients make HTTP requests to these endpoints to interact with the web application and exchange data.

Broker

One of the fundamental components of the solution is the presence of a broker. On an IoT system, data is consistently transmitted from the devices to the central server. Therefore, it is crucial to select a communication protocol that is both lightweight and offers a dependable communication channel.

The role of the broker in the MQTT protocol is to serve as an intermediary, facilitating the transmission of data from publishing clients to subscribing clients. In the specific context of our project, the publishing clients are represented by various MCUs (Microcontroller Units), while the subscribing client is the central server tasked with analysing the data.

To implement the broker, we have decided to utilize the free version of the *HiveMQ* broker[21]. This selection aligns with our requirements, particularly our focus on message security.

The *HiveMQ* broker incorporates TLS/SSL encryption[22], ensuring secure communication between *HiveMQ* and MQTT clients (both publishers and subscribers). Additionally, it provides robust support for handling substantial amounts of data and is compatible with Kotlin. While the free version of *HiveMQ* does have limitations, such as a maximum allowance of 100 devices, it is deemed sufficient for the scope of our project.

MCU Configuration

When developing an IoT system, it is always interesting to allow the operator to configure his own IoT device, for example, to connect the device to a specific local network.

The company behind the ESP devices has developed a solution. The ESP Touch protocol is a wireless communication protocol developed by *Espressif Systems* specifically for their *ESP8266* and *ESP32* series of Wi-Fi enabled microcontrollers. The free ESP Touch mobile app uses this protocol to connect to the ESP device, and thus, transmitting, not only the Wi-Fi configuration, but also a string of bytes of custom data.

Due to our selection of the ESP32-S2 microcontroller, we have decided to incorporate this protocol into our system for MCU configuration purposes.

2.1.5 Frontend

The Frontend refers to the client-side part of a software application that users interact with directly. It is responsible for presenting the user interface (UI) and handling user interactions, such as inputting data, submitting forms, and receiving and displaying information. In our project, we have developed a user-friendly frontend interface that offers users visualizations of the collected sensor data. This interface allows users to interact with the data and gain meaningful insights in an aesthetically pleasing manner.

To develop the frontend, we had to select a web app client framework. The web app client, which comprises the code and resources delivered to the user's browser and executed there, is responsible for creating the user interface and facilitating user interactions. It serves as the practical realization of the frontend using web technologies. To select the appropriate technology, we evaluated several options and conducted thorough analysis.

Given our previous experience with **React**, a widely adopted framework for building front-end applications, we made the deliberate choice to utilize it in our project. Additionally, we incorporated React Bootstrap to assist with styling, although its usage was limited. The majority of the website's styling was accomplished using pure CSS.

To streamline our development process, we integrated **Webpack**, a free and open-source module bundler for JavaScript. We also took advantage of TypeScript, which provides benefits such as type checking, enhanced code editor support, and improved code documentation. Webpack was configured to employ a TypeScript loader responsible for transpiling TypeScript files into JavaScript, thereby ensuring compatibility with various browsers[23].

By adopting these technologies and tools, we successfully constructed a visually appealing and interactive front-end interface while leveraging the advantages of React, React Bootstrap, Webpack, and TypeScript.

2.2 Existing solutions

Significant advancements have been made in this field, with well-established companies like **CropX** leading the way. CropX specializes in delivering IoT-based soil monitoring systems that incorporate pH sensors. Their extensive experience in the market has enabled them to provide real-time data on soil conditions, empowering farmers to make informed decisions regarding irrigation and nutrient management. The fundamental principles of this industrial solution align closely with our own project, as both aim to equip users with valuable insights derived from data, thereby facilitating better decision-making processes.They also provide an intuitive UI (User Interface) for the user to analyse the data[24].

In 2022, a project was developed by Bernardo Marques, under the guidance of professor Rui Duarte. This project is called **SmartDoorLock-IoT**. It consists of a system composed by a microcontroller, in particular an ESP32-S2, that hosts a socket based server and an Android application that interacts with the MCU/Server to lock and unlock a door. Behind the curtains, the system uses security features, like encryption through symmetric and asymmetric keys, Hash, MAC, databases, Wi-Fi communication. Using an IoT device, the system is able to take advantage of features only possible with microcontrollers, like the ultra low power mode, available in the ESP32-S2, which is very important in systems of this kind. Overall, this project/system takes advantage of the IoT technology to implement a secure IoT system capable of simplifying the life of a human being.

3
Proposed IoT System Architecture

3.1 Introduction

In this chapter, the proposed system architecture will be presented. Starts with the system high level view. This means defining each module and how they all interact with each other to compose the final solution. Then, for each component presented, the details of its implementation will be addressed. This way, the reader can, not only, understand the proposed system architecture without relying on the implementation details, but also deep dive into each component specific implementation.

3.2 System Overview

There are three main system components:

1. Factory: Collects the data from the neutralization filter;
2. Backend: Receives and processes the data received from the MCU;
3. Web Application: Allows the user to analyse relevant filter-collected data, and manage devices.

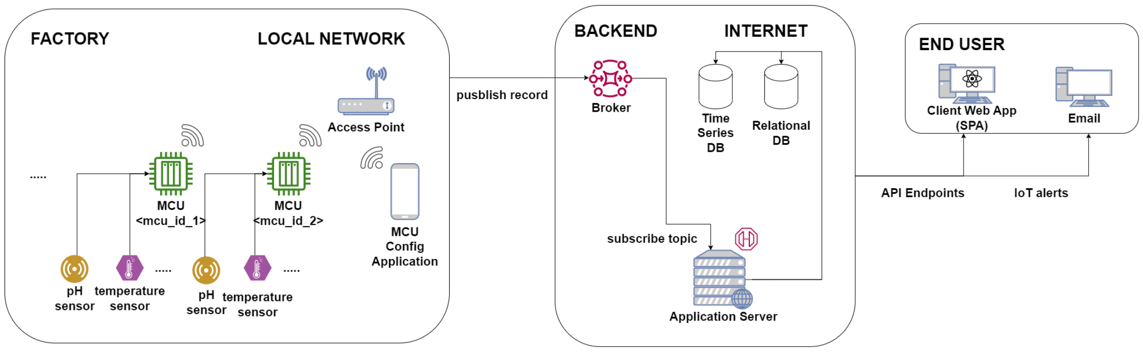


Figure 3.1: Main system architecture

As shown in the Figure 3.1, data flows from left to right. All begins in the MCU collecting data - *sensor records* - and then, sending the sensor data to the *Backend Broker* module. The Backend is composed by the Broker, which is the middleman between the *IoT Node* and the application server. The latter receives the sensor data from the broker to process it, accordingly. Eventually, the application server might send an alert email to the manager of the neutralization mechanism if the data indicates an abnormal condition. The server also provides an API that allows other applications, such as the web application described in this report, to interact with the system. This API enables functionalities like querying available sensor data, creating users, and adding devices.

3.2.1 IoT Node

Each client utilizing the system outlined in this report will have a replicated component consisting of four subcomponents, as presented in the figure 3.2.

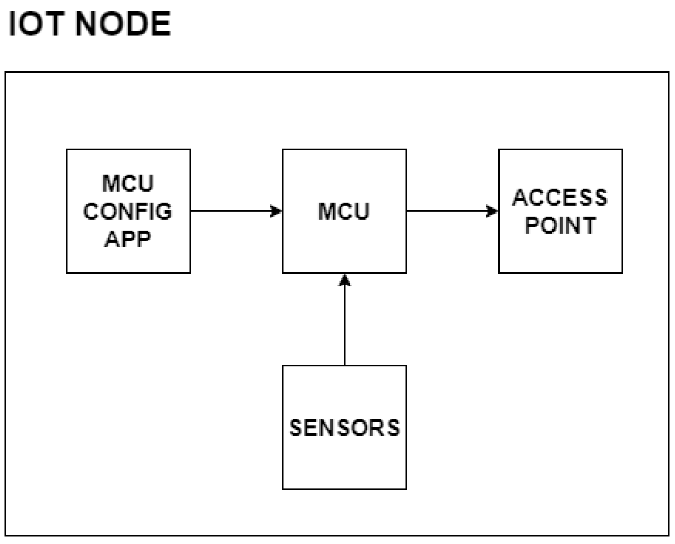


Figure 3.2: IoT Node Module Architecture

The sub-components have the following functions:

1. Sensors: collect neutralization filter data;
2. Microcontroller (MCU): coordinate when to collect and send the data to the Backend;
3. Access Point (AP) to enable the MCU to connect to the internet
4. Android app to configure the MCU

The user interacts with an android app provided by **Expressif** called **Esp Touch** to configure the MCU, for the first time. This includes setting up a device identifier and sending the WiFi network credentials directly to the MCU. After, the MCU will instruct the sensors to collect data, and send it to the backend, through the access point.

MCU

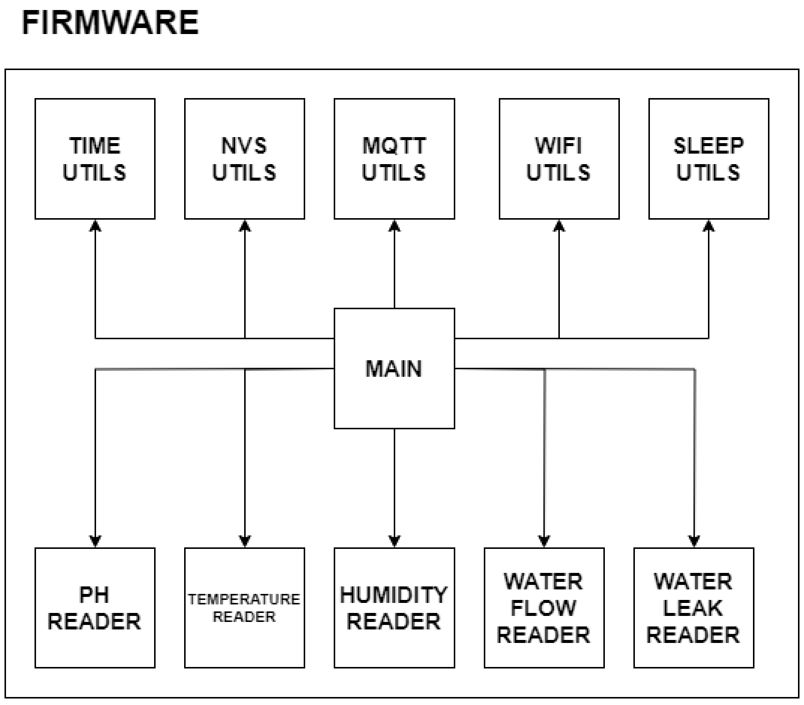


Figure 3.3: MCU firmware architecture

As represented in the Figure 3.3, the *Main* component controls the main flux algorithm. It interacts with other utility components such as the sensor reader modules to collect data from the sensors, the *MQTT UTILS* module to send data using the MQTT protocol or the *TIME UTILS* module to obtain an accurate time.

Sensors

There are a total of seven sensors:

1. Initial pH sensor to obtain the solution pH in the beginning of the filter system (before being neutralized)
2. Final pH sensor to obtain the solution pH in the end of the filter system (after being neutralized);
3. pH sensor to obtain the pH of the solution present in the filter system (before being neutralized);
4. Ambient temperature sensor;
5. Air humidity sensor;
6. Water flow sensor to estimate the amount of water that flows in the system;
7. Water detection sensor to alert if water has exited the system through a place it is not supposed to.

The primary objective of this project is to develop a system that automates the inspection process for potential issues in a neutralization mechanism. All this sensors give us the data necessary to accomplish that. However, it is also interesting to develop a system that is capable of providing us with meaningful insights, regarding the utilization of the filter, for instance the amount of water that flows through it, or the neutralization effectiveness, given the pH of the solution before and after being neutralized by the system. Currently, our system is only using a single pH sensor.

MCU Config App

An android application is used to configure the MCU, in particular to connect it to the Access Point, and transmit other necessary information like defining the unique Device/MCU ID.

3.2.2 Backend

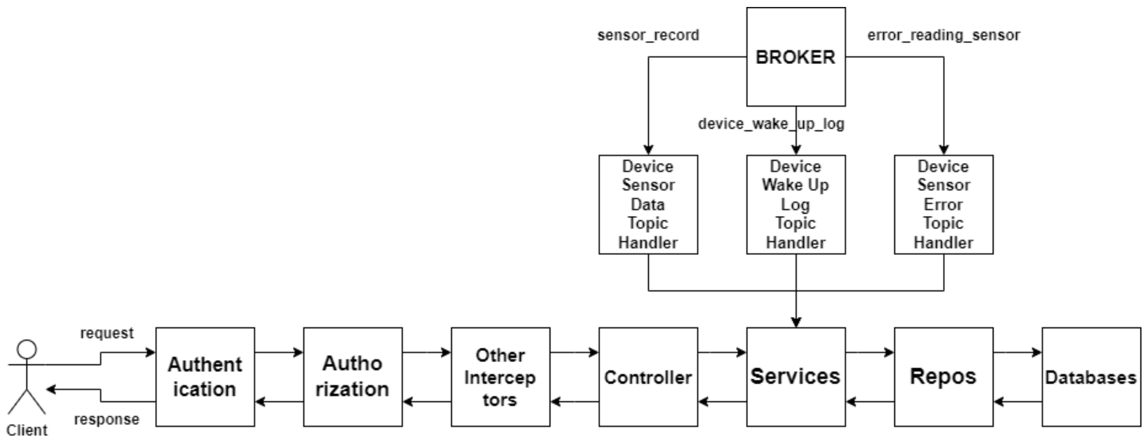
This component is shared by all the clients, and serves a an IoT platform, supporting all IoT nodes scatter across the multiple clients using this system. It should be deployed in the cloud, so all IoT nodes have access to it. It is composed by the Application Server, the Broker and Databases.

Application Server

The server computes all system involved data. All the business logic is enforced here. This application serves two purposes:

- Exposes a Web API, so clients (ex: frontend) applications can interact with the system, providing user-friendly interfaces.
- Store and compute all system data, enforcing business logic.
- Receive, analyse and store the data that coming from the Broker module.

It also exposes a Web API, so that Frontend applications can interact with the system.



The Figure 3.2.2 shows the main modules involved:

- Authentication and authorization module will be used when the request is attempting to access protected resources;
- Other interceptors, such as request logging interceptors,

- The controllers to define the API operations;
- The services display available system actions which will guarantee atomicity and enforce business logic while accessing and manipulating system data;
- The repositories implement the database ”drivers”, meaning they handle all the logic associated with the database access;
- The MQTT topic handlers, responsible to receive with data coming from the broker, and deal with them. This includes analyzing sensor data, storing coming records, etc.

Each time a client sends a request, the controllers will use the services to fulfil the request. Likewise, when the services need to access or store data, they will use the repositories to interact with the physical databases. In essence, the request will follow a left-to-right flow, while the response will flow in the opposite direction.

Database

Due to the nature of this project, there are two types of data to be persisted:

- Aggregate data: Users, devices, etc;
- Time based data: Sensor data

A relational database (RDB) was chosen for the first type of data that requires strong aggregation. This type of data involves relationships, such as users having devices and devices having logs. A relational database is specifically designed to enforce structure and maintain relationships between data entities. Additionally, when dealing with sensitive data like user information, it is crucial to handle it with care.

In this project, *PostgreSQL* was selected as the relational database. It is an open-source, powerful, and highly reliable database system. Its robust features, scalability, and strong community support make it a suitable choice for handling complex data structures and ensuring data integrity [25].

On the other hand, we chose to use the *InfluxDB* database to store sensor record data. This is a time series database, which is specially designed for this kind of data, where each recording involves a value and a timestamp.

To build each Database, we started by designing their data models. By doing this, we are selecting the requirements for our specific domain.

The Figure 3.4 represents the schema of a relational database designed to manage user, device, log, sensor error, token, and password information. The schema defines the relationships between these entities, ensuring the proper association and integrity of data within the system.

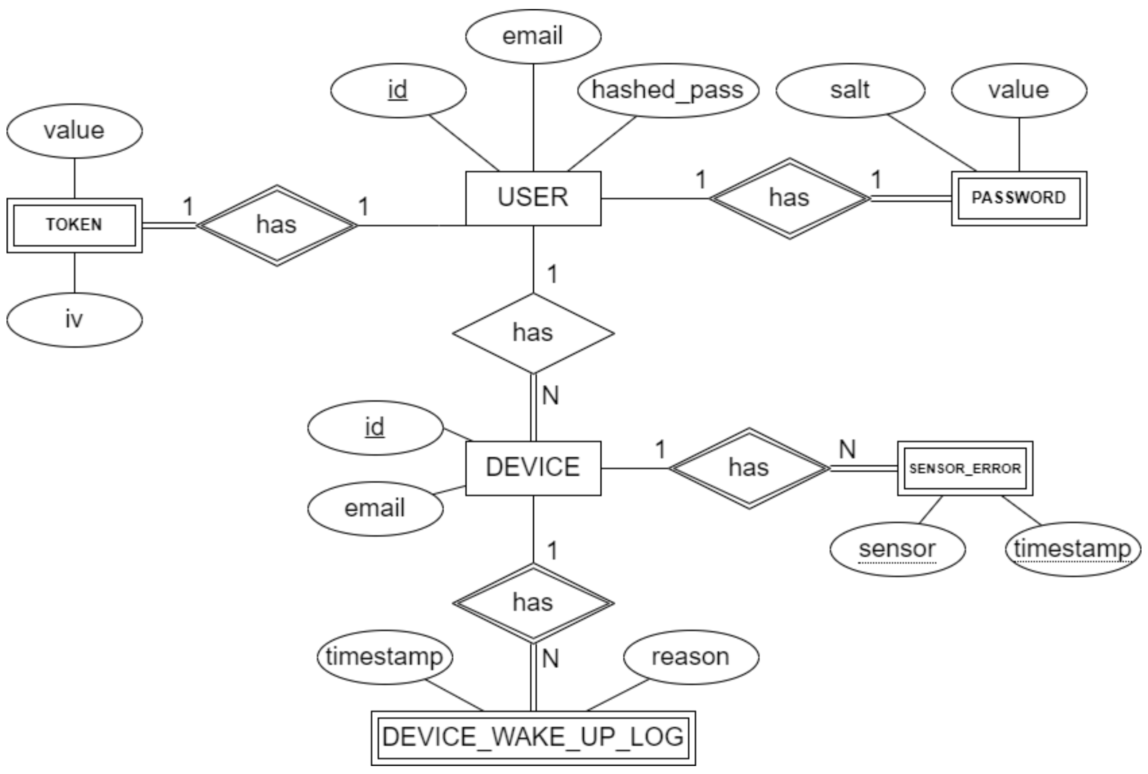


Figure 3.4: Postgres DB Entity Model

The user can have multiple devices operating. Each device must always be tied to one and only one user. Each device can have a set of logs and sensor errors. This can only exist when associated to the device. The same goes for the tokens and passwords, as they can only exist tied to a user. To summarize the relationships:

User and Device: One-to-Many (One user can have multiple devices). Device and Logs/Sensor Errors: One-to-Many (One device can have multiple logs and sensor errors). User and Tokens/Passwords: One-to-One (One user has one set of tokens and one password).

These relationships ensure the proper association and integrity of data within the system, allowing for effective management and organization of user, device, log, sensor error, token, and password information.

As InfluxDB is a non-relational database, the standard relational data representation approach does not apply. However, the following diagram offers a clear understanding of the type of data it stores:

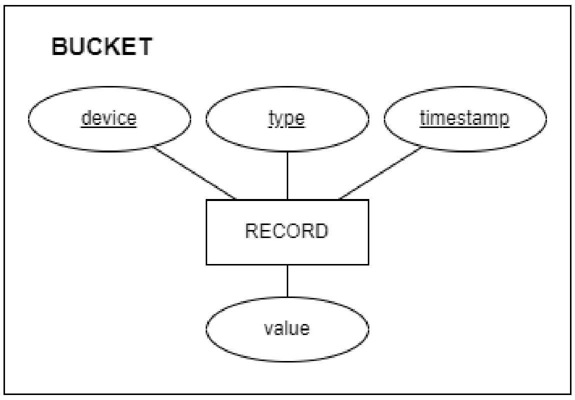


Figure 3.5: Time Series DB representation

A record always consists of the device, the type or sensor, the timestamp, and the corresponding value or reading.

3.2.3 Web Application

As the third system component, the Web application appears as a user-friendly entry point to the system. It interacts with the exposed Web API, in the Backend, to manipulate system data.

The main website architecture is represented in the Figure 3.6, it presents an overview of the different modules and connections that constitute the website.

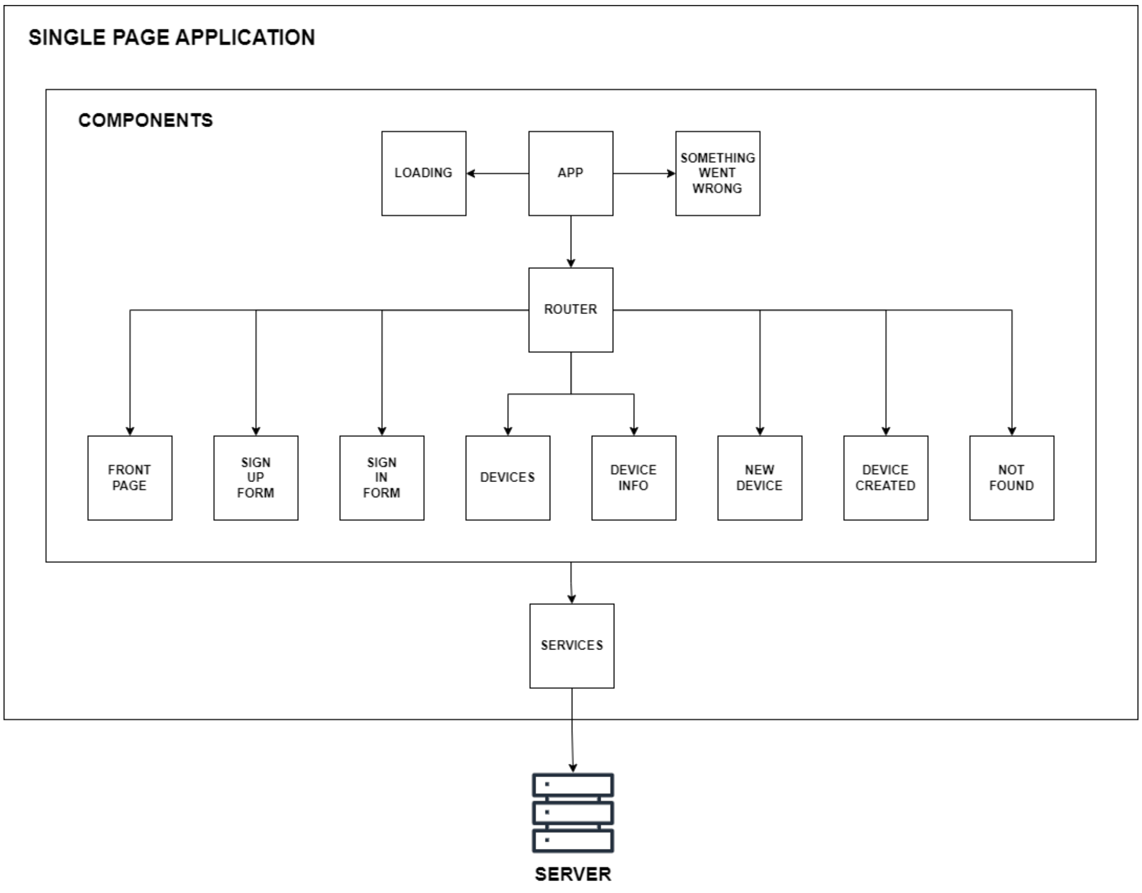


Figure 3.6: Single Page Application components diagram

The App module is responsible for loading the Router, which is responsible for selecting the component that the user will see.

The majority of components within the application rely on the services layer to access the backend server. This allows them to request and manipulate system data.

The services layer acts as an intermediary between the components and the backend server, handling the communication and data operations on behalf of the components.

3.3 IoT Node

3.3.1 MCU

Behaviour

The MCU behaviour can be summarized by the following ASM (Algorithmic state machine) chart of the Figure 3.7.

Most of the time, the MCU will be in **deep-sleep** mode, the CPUs, most of the RAM, and all digital peripherals that are clocked from APB_CLK(Advanced Peripheral Bus Clock) are powered off [8] since it is not necessary to make constant readings due to the nature of the neutralization mechanism because conditions will not change in a short space of time. Considering the importance of power consumption, every second of the MCU being active has a significant impact.

Upon awakening, the device needs to determine the cause of its wake-up event. While one possible reason is simply that it was powered on, the most relevant factors for our specific use case are identifying whether it woke up from a *deep sleep* or as a result of an unexpected event.

Typically, the ESP will re-start its execution after an exception is that was not handled. When this happens, we can conclude that the device did not wake up because of a timeout (this corresponds to the normal behaviour of the MCU to wake up and take readings) and consequently no further action is required. After reporting the action that cause the wake-up, the device goes to sleep again.

In this project, IoT devices are programmed once and deployed after. Even if the programmer is still working on bugs or the device is deployed, it is not trivial to update the firmware, since to do that, the programmer will have to send the new update explicitly. For this reason, if an error occurs, it is of great importance to report to the backend, so the users get notified about such event.

After identifying the cause of the wake-up event, the next step is to power on the sensors and perform the necessary readings. Once all the required readings are obtained, the MCU sends the data to the backend for further processing. Subsequently, the device enters sleep mode once again, conserving energy until the next wake-up event occurs.

Sometimes, the sensor readings may not accurately reflect reliable values. In order to address this issue, the device performs multiple readings before transmitting them to the backend. There are two approaches to achieve this. The first approach involves taking a reading and then putting the device back to sleep for a short period, thereby conserving energy. However, this method requires powering up the sensors again upon waking up, and it also necessitates allowing sufficient time for stabilization. Consequently, the total time taken is longer compared to simply keeping the MCU blocked between readings, even when it is active. Hence, the second option was selected. By spacing out the readings, the quality of the sensor records improves, making them more representative of the actual filter condition.

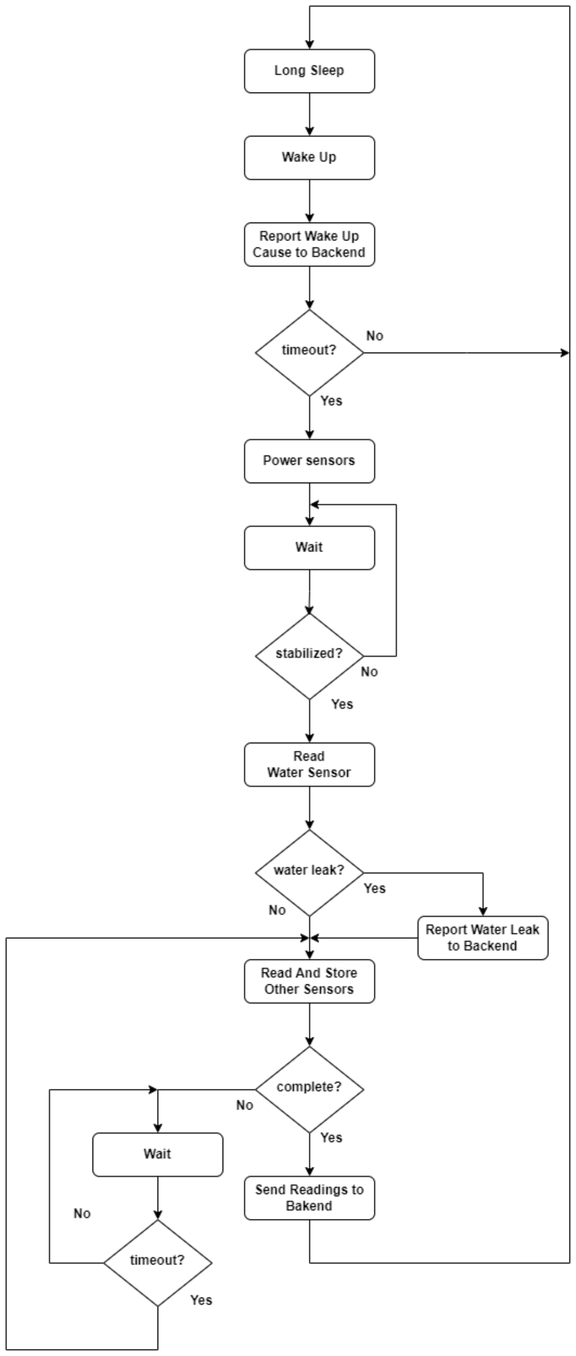


Figure 3.7: MCU general behaviour

Sleep

For this specific project use case, the MCU will be sleeping for two reasons. The first one is what we called *Long Sleep*. This is when the MCU will be sleeping for most of the day. The other reason is what we called *Short Block*. It is at this point that the MCU is sleeping for a short time, ideally seconds, waiting to make another reading. The *Long Sleep* state is characterized by putting the device in *Deep Sleep* mode, this mode maximizes the conservation of the energy by blocking the MCU of the great majority of its functionalities. The objective of *Short Block* is to make a brief pause between readings from the same sensor to ensure the reliability of the data.

3.3.2 Sensor Modules

Each one of the seven sensors require its own module that handles the sensor readings, since the code to read from the sensor is specific for that same sensor. Each module has a function that should be called to make the sensor reading, abstracting the caller of all the code required to interact with the sensor.

Since the beginning of the project, during the planning phase, we realized that the physical sensors might not be available to us. To address this challenge, we implemented a solution by substituting the *Sensor Reader* modules with fake implementations that simulated the actual sensor readings. Consequently, for each sensor reader component, two implementations were developed: a real implementation that interacts directly with the physical sensor, and a fake implementation that simulates the sensor procedure. This approach provided flexibility and independence from the availability of the physical sensors, allowing us to continue testing and developing various components of the system regardless of the sensor's presence.

3.3.3 Ph Sensor

To deploy the **pH sensor**, it was first necessary to make experimental readings using this device. The ESP32-S2 includes ADC (Analogue to Digital Converter) channels to make analogue readings. This input value, the ADC, is a value with 13 bits range, which means is able to read values between 0 and 8191. Zero means null voltage, while 8191 is the maximum voltage the MCU is able to read.

The maximum voltage upper bound is dependent on the *VRef* constant, which chip dependent[26]. Most round 1.1V. It is not possible to change this value, so, we resorted to the attenuation value. Attenuation affects the values read from the ADC by adjusting the input voltage range and precision. The ADC units in the ESP32-S2 support configurable attenuation settings to handle different voltage levels. When higher attenuation is set, the ADC's input voltage range is increased, allowing measurement of larger input voltages.

According to the documentation, 11dB (max) allows the chip to read from 0 - 2500mV. This means, that with an ADC value of 13 bits, and an attenuation of 11dB, the ADC is almost always sensible to voltage change between 0 and 2.5V, excluding extremes as with voltages near 0V and 2.5V. The read values should never achieve reach this extremes, and so, it imposes no problem to the readings.

Since the pH sensor outputs 5V (maximum) we used a voltage divider to downgrade the analogue input voltage on the chip, to 2500mV (maximum).

By choosing appropriate resistor values, we can create a voltage divider that scales down the 5V output from the pH sensor to a voltage that falls within the ESP32S2's range (0-2.5V). The formula for calculating the output voltage is:

$$Vout = Vin * \frac{R2}{R1 + R2} \tag{3.1}$$

To achieve a maximum output voltage (*Vout*) of 2.5V while operating with a maximum input voltage (*Vin*) of 5V, we carefully selected two resistances with equal power ratings. Our selection was guided by the consideration of minimizing current flow to conserve system energy, while simultaneously avoiding excessively high resistance values that could introduce signal interference, since the weaker signal can be easily overwhelmed or distorted by the interfering signals, leading to inaccuracies or disturbances in the system's operation.

We opted for 47kOhm for the value of both resistor(R1 and R2). R1 is the resistor connected to the input voltage (Vin), and R2 is the resistor connected to the ground or reference point as shown in the figure 3.8.

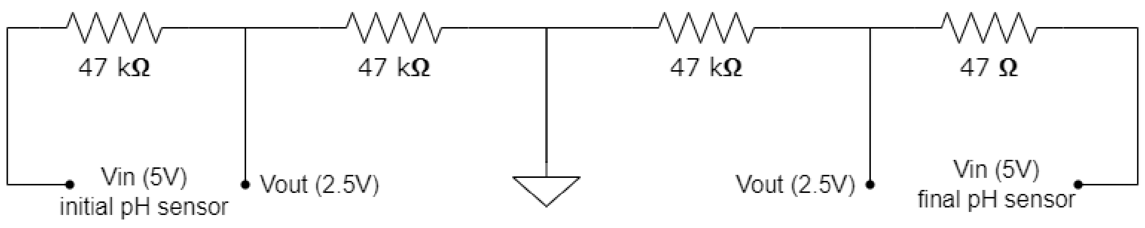


Figure 3.8: Voltage divider circuit used to achieve lower output voltage

After selecting the appropriate resistors, the next step was to make the association between the read ADC value and the correspondent pH value. There are two different ways of preforming this association.

The traditional one, which consists in the analysis of the characteristic curve of the sensor, which establishes the relationship between voltage and pH. This step provides essential insights into the behaviour of the sensor. Following that, the voltage output of the microcontroller unit (MCU) is examined. To ensure accuracy, it is necessary to calibrate the voltage readings using an external measuring device. This calibration process helps align the voltage measurements from the MCU with the reference values obtained from the external device. This calibration process would involve the analysis of the data present in the Figure 3.9, to adjust the voltage read from the MCU to the voltage read from the oscilloscope.

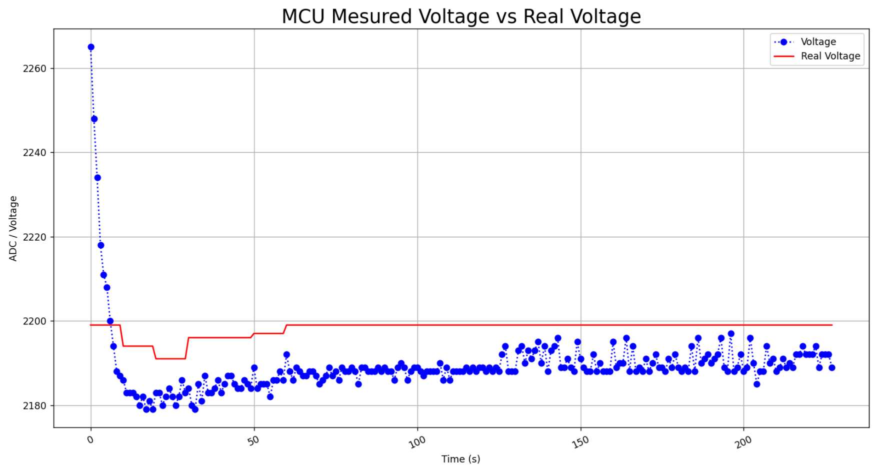


Figure 3.9: Graph illustrating the measured voltage with the MCU against the real voltage measured with an oscilloscope, of a liquid solution, pH equal to 8.

However, as we have liquids with known pH, we go directly from the ADC value to the pH value. According to the figure 3.10 the conversion between the ADV values and the voltage is almost linear, excluding both extremes. The voltage to pH transformation function is also linear.

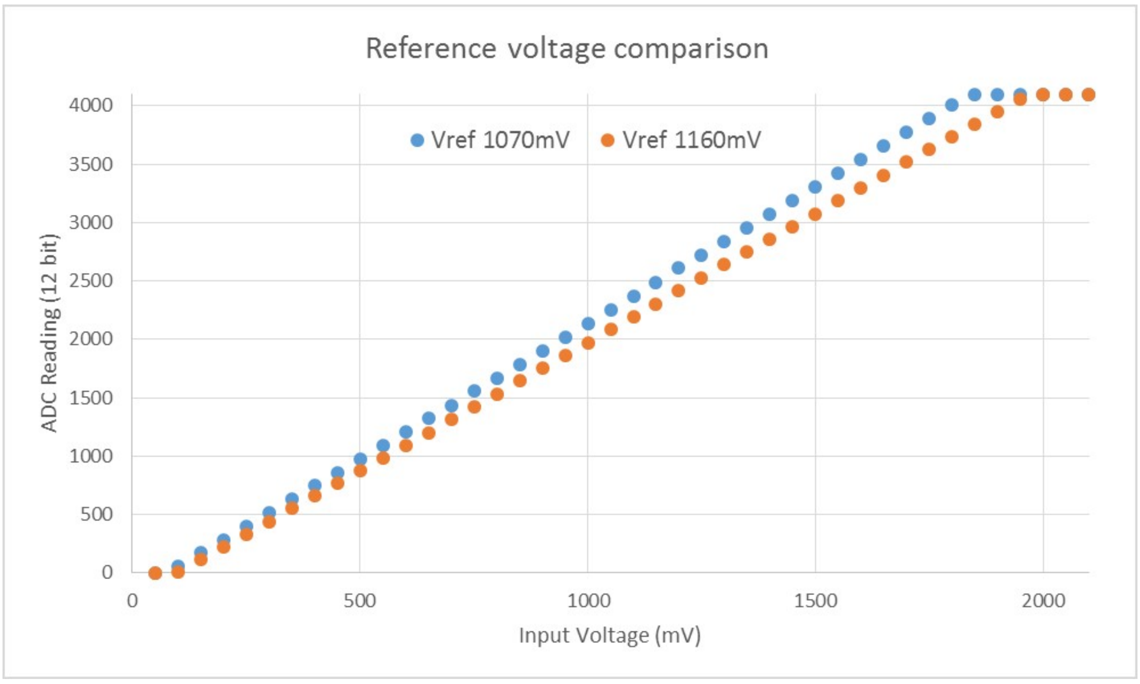


Figure 3.10: Graph illustrating effect of differing reference voltages on the ADC voltage curve.

PH values are relatively linear over a certain range (between pH 2 to pH 10), we need two calibration points to determine the linear line, and then derives the slope of the line so that we can calculate any pH value with a given voltage output, and therefore, ADC value.

Knowing this, we experimented a solution liquid, measuring pH equal to 4 and read the correspondent ADC value in the MCU. The same was done with another solution measuring pH equal to 8.8.

PH deduction:

7955 ADC corresponds to pH 4.0

7184 ADC corresponds to pH 8.8

$$
\begin{aligned}
m &= \frac{8.8 - 4.0}{7184 - 7955} \\
4.0 &= 7955 \cdot (-0.00622) + b \\
b &= 4.0 - 7955 \cdot (-0.00622) = \pm 53.4801 \\
\text{ph} &= \text{ADC} \cdot m + b \\
8.8 &= 7184 \cdot (-0.00622) + 53.4801
\end{aligned}
$$

We have a high level of confidence in the accuracy of the pH values. This confidence is supported by the manufacturer's assurance of the reliability of these values, and we have further validated them through the use of a commercial calibrated pH meter.

With this, it is possible to trace the slope, allowing the conversion from ADC to pH. Even so, after powering up the pH sensor, it is necessary to wait for the sensor to stabilize, as the figure 3.11 shows. Starting from the data point at the 102 second(ADC: 7947), we observe a sequence of consecutive data points where the ADC values remain within the range of 7947 to 7953 (±3 units).All the values after 102 seconds consistently fall within the defined stabilization criteria of ±3 units.Therefore, it was decided that after 102 seconds, the reading is considered valid.

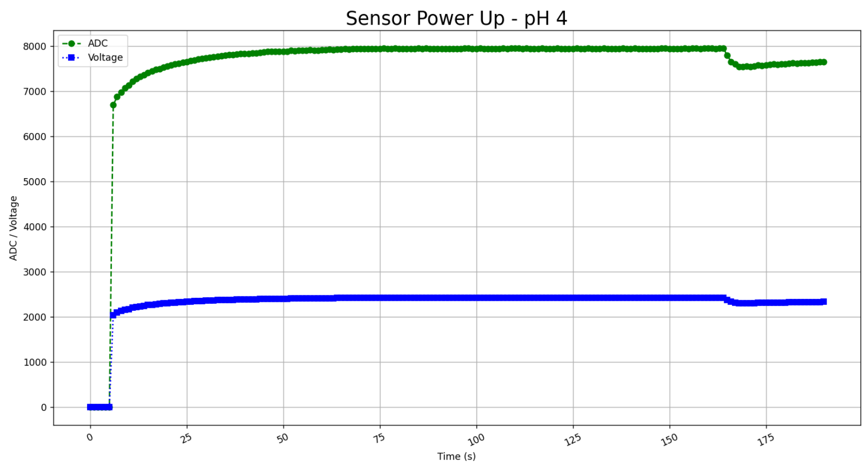


Figure 3.11: Graph illustrating the measured ADC and voltage of a liquid solution, pH equal to 4.

Ambient humidity and temperature sensor

To make the ambient humidity and temperature readings, the MCU component that handles this needs to respect the behaviour of the chosen sensor module, as described in the subsection 2.1.1.

During the development process, we conducted some research to identify existing implementations that could be leveraged, thus avoiding the need to write low-level code from scratch.

However, to determine if calibration was necessary, we conducted an experiment to verify if the values were close to the real environment condition. The test results indicated that after the 30 seconds to that took the sensor to stabilize, the values were correspondent to the room temperature and humidity, in Figure 3.12, the data relative to the reading of the sensor is presented graphically.

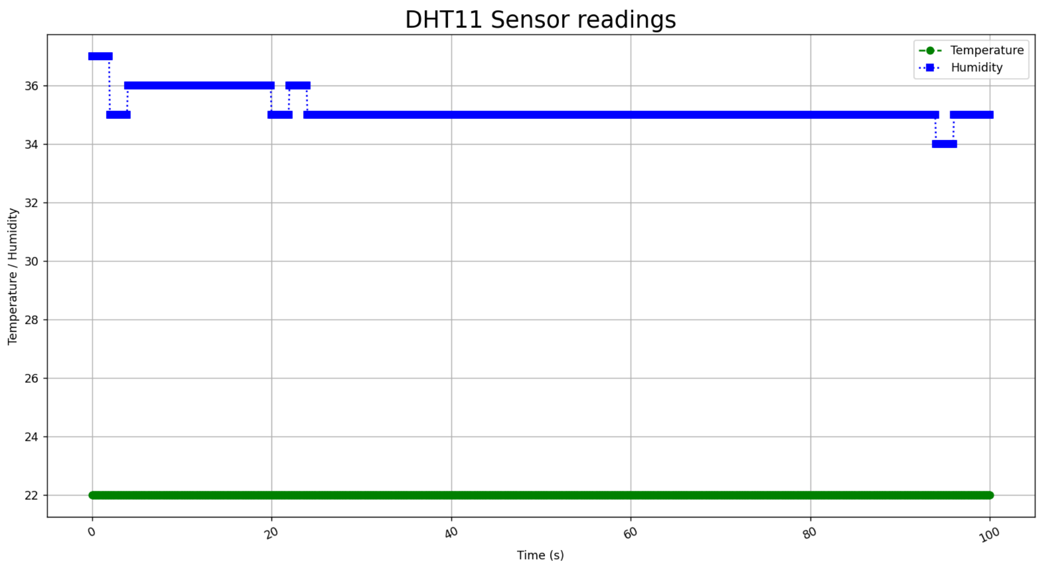


Figure 3.12: Graph illustrating the measured humidity and temperature, from with the DHT11 sensor, at room temperature.

Water leaks sensor

Relative to the water leaks sensor presented in the subchapter 2.1.1, the sensor will output a certain voltage, each time it is in contact with water.

At first, we decided to use the ESP32-S2 functionality of waking up the microcontroller whenever an input pin receives the digital 1 value from the sensor. We identified two problems with this approach:

1. After conducting experimental tests, it was concluded that the module, even when fully immersed in water, only outputs 1.7V. This voltage level is insufficient for the MCU to recognize it as a digital logic high. The graph in Figure 3.13 illustrates the measured ADC and voltage when the sensor is immersed in water and taken out.
2. The sensor would have to be constantly on, resulting in greater energy consumption. This goes against the established requirements, where the hardware is supposed to be energy-efficient. Additionally, it is assumed that any water leakage in the system would occur at a very slow rate.

Therefore, in the context of our problem, it is not advantageous to constantly monitor for water leakages. It's more appropriately to check for water leaks, only after the MCU wakes up, i.e. this sensor would have a behaviour similar to the others.

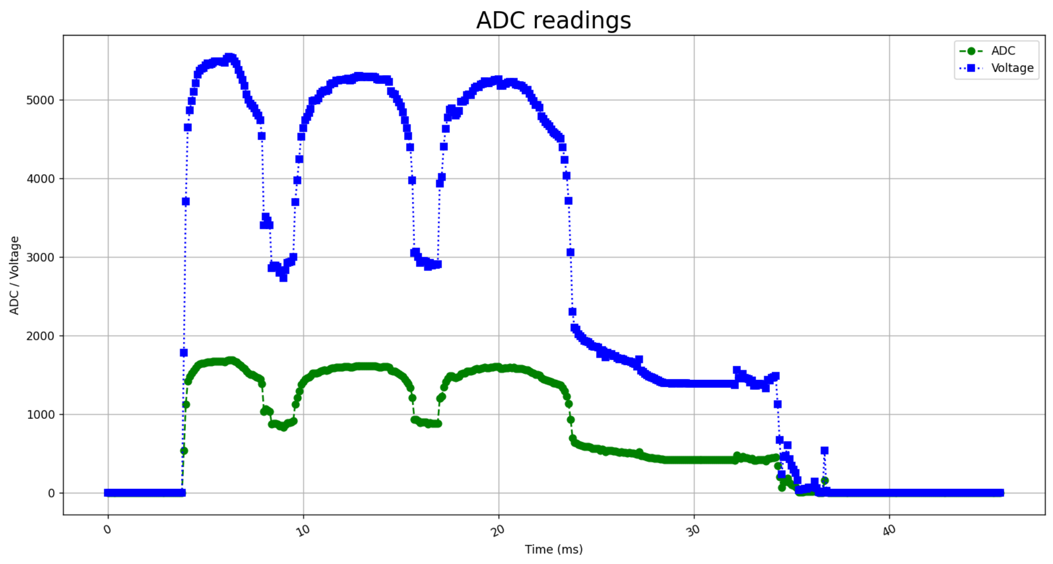


Figure 3.13: Graph illustrating the measured ADC and voltage when the sensor is immersed in water and taken out.

3.3.4 Assembly

The Figure 3.14 shows the schematics of our assembly at the time of the beta phase. In order to optimize power consumption, the sensors will not be constantly powered on. Instead, each one will be connected to an I/O pin, allowing them to be powered only when the chip is awake and actively using them. All sensors, except the pH one need 3.3V to operate. Since each GPIO can be used as digital, operating on 3.3V, when outputting digital one, they can be used as digital gates, for each sensor. Therefore, each one is connected to an individual GPIO. Concerning the pH sensor, which requires a total of 5V to operate, there is only one PIN outputting 5V. This should be used to power both pH sensors. The problem is that it's not possible to turn down the power with software. The consequence is that the sensors would always be running, and unnecessary consumption would be drained. To solve this, a transistor was used to "cut" the power whenever necessary. Since both pH sensors have the same stabilizing power timings, will be powered always at the same time, and used almost at the same time, we opted to use the same transistor. Another GPIO pin will be used to control the gate.

As a result, the experiments presented above play a crucial role in determining the required operational time for the sensors.

Both pH sensors have their own resistances to lower down the output voltage. Since both pH sensors have different purposes, each one is connected to a different MCU ADC channel, respectively ADC1_0 (GPIO 2) and ADC1_1 (GPIO 3), so it's possible to read two different values: initial and final pH. They are fallowed by the water sensor, which, which is connected in the ADC1_2 MCU channel (GPIO 4). The DHT1 sensor is digital and so, connected to the GPIO 9, configured as digital. The order and GPIO pins choice is not very important since there are enough pins to shelter all necessary sensors. Instead, the only concern was to occupy only one side of the MCU GPIO pins, since the breadboard used is not big enough to place the MCU in the middle and still have pins available in both sides. This should not be a concern in production.

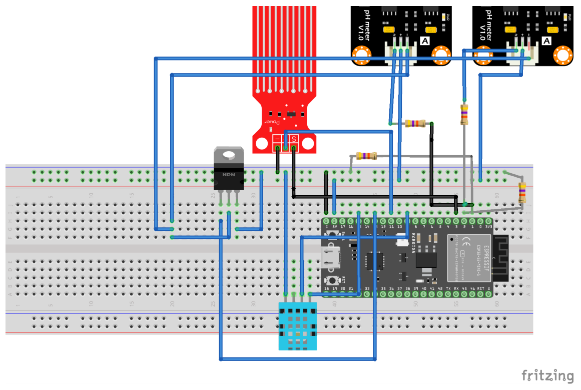


Figure 3.14: Current IoT Node assembly in the breadboard

Wi-Fi

To connect the MCU to the Wi-Fi, two properties are required: **SSID** and **Password**. The SSID and the password are the credentials used to connect to the Wi-Fi network. Firstly, the MCU tries to recover the Wi-Fi configuration and, if available, tries to connect to the Wi-Fi. If not, the MCU will initiate the *ESP Touch protocol* to allow users to utilize the *ESP Touch* Android app to transmit the Wi-Fi configuration to the MCU.

Once the user sends the credentials to the ESP32-S2, the data will be stored in the persistent memory, for later to reconnect to the same SSID. Through the Touch Android application, the user has also the ability to send a costume string, as described in the subsection 2.1.4, which in our project we define this string as the unique **device ID** that corresponds to an identifier exclusive for each microcontroller.

To incorporate the ESP Touch Protocol and Wi-Fi modules into our project, we referred to the **SmartDoorLock-IoT** project mentioned in section 2.2. This project had already implemented these modules, so we utilized and customized them to suit our specific use case.

MQTT

To send the sensor collected data to the backend broker, we developed the MQTT˙util module. Most of the implementation is taken from the extensive ESP IDF documentation, which deals with the MQTT connection to the target endpoint, and other low level details. Yet, we adapted with other procedures to send sensor data, according to our domain.

In our IoT system, where devices may operate in an unreliable network environment and energy efficiency is crucial, we have considered various MQTT data formats. While a compact payload format can contribute to system efficiency, we have opted to use the JSON format. Despite JSON potentially having a slightly larger payload compared to other compact formats, such as binary or compressed formats, we have determined that the benefits of its flexibility and ease of parsing outweigh the minimal size difference.

The content of the JSON file includes the device ID, the timestamp, plus the sensor value or the device wake up reason. If, for instance, the payload included a list of undefined objects, a different serialization mechanism would be required. Also, the MCU doesn't receive messages from the exterior. One inefficiency related to the JSON format is the de-serialisation process, which never happens in the current system.

One deficiency in our system is the need to configure the MQTT broker URL in the firmware itself. One possibility to solve this problem is to provide it through the ESP Touch protocol, using the already described Android App. This would be necessary only for the first time the user interacts with the IoT MCU device.

Storage

To persist data in the ESP32-S2, the Non-volatile Storage ESP IDF library was used in this project. Non-volatile storage (NVS) library is designed to store key-value pairs in flash, it is a type of non-volatile storage device, meaning that even if it has no power, it will retain the information saved in it [27]. This way, even when the device is powered down, data won't be lost.

In the MCU firmware, there is a module called nvs_utils to interact with the NVS partition, providing an interface to store and retrieve the device ID and Wi-Fi credentials.

Time

Each time a sensor reading is performed, it is necessary to measure the time. There are many ways to read the current time. Some ways are more straightforward since they do not require internet connection to sync the time. For example, using the Real-Time Clock (RTC) which tracks the time, even when the device is in sleep model.

In our system, we have selected the Network Time Protocol (NTP) to acquire accurate timestamps. However, it is important to note that utilizing NTP comes with certain disadvantages, primarily the reliance on an internet connection for synchronization. While this dependency on internet connectivity poses a challenge, the advantage of obtaining highly accurate time readings justifies this trade-off.

Each time the MCU wakes up, it initiates a log transmission to the Backend Server, necessitating an internet connection. Considering that the Wi-Fi connection already introduces certain overhead, the inclusion of an additional NTP request does not significantly impact efficiency. While the NTP request does add an extra step, the overall impact on efficiency is minimal when compared to the existing Wi-Fi connection requirements.

3.4 Backend - Application Server

3.4.1 Users

As with any other system, a User needs to be identified. We opted to use a UUID, since the probability of replications is very small.

To satisfy the company requirements, the system must support users. Each one will be associated with Devices and should be alerted when something goes wrong. Yet, we opted to define two types of users:

- The Admin that has access to all users and devices, aswell as each device sensor data and logs. This user is not authorized to create devices. His mainly objective is to monitor other users and correspondent devices.
- The User, which is able create devices and is authorized to see just it's own devices, sensor data and logs.

The second one was not request by the company, but a choice of the authors. This project doesn't just try to automate the neutralization filter monitoring, but also to extract relevant data, so it's possible to evaluate the system efficiency. The admin will have access to all this data.

3.4.2 Devices

With the Devices identifiers, the story is a little different... This system projects that the client will have to configure it's own devices, meaning that the Device ID will have to be passed to the Device manually. Using the same format above - UUID - would guarantee uniqueness. Yet, the poor client would have to write the complete ID into the Device. So, to avoid this inconvenient, a domain ID generations was developed. This system doesn't aim for an enormous number of client. Therefore, an ID composed of 32 bits should be more than enough, to distinguish all users. This means a total of 4,294,967,296 possible identifiers. Yet, to avoid the need for the user to have to insert all 32 bits, using alphanumerical characters, is possible to reduce the overall size of the ID.

Each device is associated with an alert email, defined upon creation. This will be used to notify the user if the filter may need human atention.

3.4.3 Sensor Analysis

Upon receiving sensor data from the Broker, the server needs to analyse it. This means evaluating if the received sensor records indicate dangerous values in the filter. There are many possibilities to achieve this. The Server only needs to know the thresholds (boundaries) associated with each sensor type. For example, the pH lower and upper boundaries might be 6 and 7. One possibility would be for each device to send, along with each value record, the sensor thresholds. The server would receive the value and the respective thresholds and process it immediately. The main disadvantage is that it require more bandwidth, which deeply impacts the power consumption on the MCUs. The advantage would be, it would be possible to customize threshold values among different devices. This system assumes that all filters belong to the same company, and thus, they should function under the same conditions. Therefore, the current solution is to store the sensor thresholds inside a configuration file, in the Backend server. This implies that all devices sensor data will be treated similary.

3.4.4 Spring Configuration

One notable feature of the Spring framework is its robust dependency injection mechanism. This feature offers a powerful abstraction for programmers when implementing software layers within the Spring framework.

This system effectively utilizes this feature. For instance, the server repository classes rely on a *Handle* object, which represents a connection to the database system and is closely linked to the database. During application testing, such as integration tests, it becomes necessary to provide a different database to the *Repository* class. This is where the Spring configuration comes into play. In this project, various configurations are defined to accommodate different databases, depending on whether the application is running in a production or testing environment.

3.4.5 Interceptors

An interceptor is a class that acts as a Spring component and, as the name implies, intercepts the request before and after it passes through the controller. With access to the HTTP request information, interceptors can be used to perform additional processing on requests before they are sent to the appropriate handler. They can also modify or stop the request if necessary [28].

In the developed server, there are two interceptors: the **LoggerInterceptor** and the **AuthInterceptor**.

The first is used to log the requests, giving the developer means to inspect the requests that the server is receiving.

The *Authentication interceptor* serves for a different purpose. It uses its ability to inspect the request and metadata to make, not only the authentication process, but also the authorization process.

By implementing this approach, if the handler tries to a resource linked to a user, the interceptor, access to the authorization field, in the HTTP request, tries to authenticate the user, and resolve the request.

If the process fails and the argument cannot be resolved, it signifies that the necessary information or data required for authentication cannot be obtained. In this scenario, the interceptor determines that the request should be terminated because the user is classified unauthorized, responding with a *401 Unauthorized status code*. This means that the user's request is immediately halted, and a response indicating the lack of authentication or authorization is sent back to the client.

The 401 Unauthorized status code is a standard HTTP response status code used to indicate that the request requires authentication, but the user's credentials are either missing or invalid. By terminating the request and returning a 401 Unauthorized response, the client is notified that they need to provide valid authentication credentials to access the requested resource. This approach ensures that unauthorized access attempts are promptly rejected, and the client is informed of the authentication requirement. It helps enforce security measures and protects sensitive resources from unauthorized access attempts.

Similarly, if the controller method has the **Authorization annotation**, the interceptor will try to authenticate the user and evaluate if the user possesses the necessary role defined in the annotation to be allowed to proceed with the request. If not allowed, a 403 Forbidden status code is returned. The 403 Forbidden status code is an HTTP response status code that signifies that the server understood the request but refuses to authorize it. It is used to indicate that the server has explicitly forbidden the requested action, even if the user is authenticated.

3.4.6 Controllers

In Spring, a Controller class is always proceeded with a *@Controller* annotation. In particular, in our application, the *@RestController* annotation is used over the other one, because it pre-defines a set of standard assumptions, common to Rest API implementations. For example, the **@RequestMapping** methods assume **@ResponseBody** semantics by default.

For organization reasons, the controllers are separated into multiple components, described in the architecture.

As an example, the module *DeviceController* includes the method *addDevice*, which has the *@PostMapping(URI)* annotation. By including the *Authorization* annotation, the annotated method becomes the designated handler that will be invoked when a client sends a request to the specified URI (Resource Identifier) mentioned in the annotation.

The method takes a single parameter of type *User*. This informs Spring to resolve the requested parameter automatically before executing the handler. By requesting this parameter in the method signature, the *Authentication Interceptor* described in the previous section ensures that only authenticated requests reach the controller.

All handler methods have another type of annotations: API documentation annotations. These are used to generate automatically the Swagger specification API documentation[29].

Inside the controllers, an *Authorization* annotation is used. This is not part of Spring. This one is an application domain annotation, and is used to mark the controller methods that require the user to have a specific Role: User or Admin.

3.4.7 Services

The primary objective of this component is to provide communication between the *Controller* and *Repository* layers, enforcing business logic. One of our key priorities was to ensure the atomicity of database transactions within the system. Atomicity refers to the property of transactions where they are guaranteed to either fully occur or have no effects at all. This guarantee of atomicity is crucial in maintaining consistent and reliable data across the system. To address this propriety, the mojority of the services rely on the *TransactionManager* interface, which consists of a single method: *run()*. This method ensures atomicity of operations. If the execution of the method completes successfully without any exceptions, the changes made during the transaction are committed. However, if an exception occurs, a rollback is triggered, undoing any modifications made within the transaction. This ensures that concurrent operations only observe consistent and committed data, and any intermediate changes are not visible to other processes until they are finalized through a successful commit.

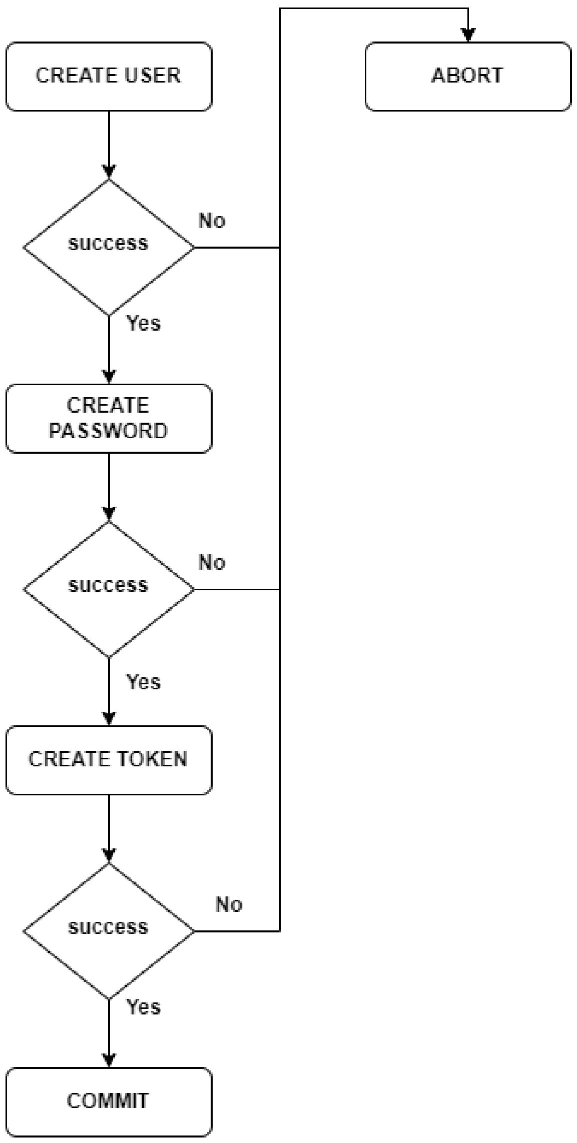


Figure 3.15: Creating a User in the Services

The diagram depicted in Figure 3.15 illustrates the process of creating a user, involving multiple actions. These actions must be executed successfully in a sequential manner. However, if any of the actions encounter an error or failure, the system employs a rollback mechanism.

This is accomplished by the use of the **Jdbi library**, which provides a convenient and efficient way to work with databases in a transactional manner. By leveraging its capabilities, the class is able to ensure that the transactional boundaries are properly defined and that the necessary operations are executed within the transaction. In particular, in this project, it interacts with the Postgres DB.

As previously stated, the *Service* layer interact with the *Controller* layer. Several challenges can arise during the execution of an operation, leading to inconveniences or complications. When an operation encounters a failure, it is beneficial to provide information about the cause of the error. By doing so, the Controller classes can adapt the type of response they generate based on the specific error.

```
fun getSensorRecordsIfIsOwner(deviceId: String, userId: String,
    sensorName: String): SensorDataResult {
    return if (!deviceService.existsDevice(deviceId))
        Either.Left(SensorDataError.DeviceNotFound)
    else if (!deviceService.belongsToUser(deviceId, userId))
        Either.Left(SensorDataError.DeviceNotBelongsToUser(userId))
    else
        Either.Right(sensorDataRepo.getSensorRecords(deviceId,
            sensorName))
}
```

As an example, in the snipped above, the function first check if the device itself exists, and if so, it checks if it belongs to the entity making the request. Two points of failure can happen. To handle the cases where there are multiple possible points of failure, was developed a strategy to represent the outcome of the operations. It consists in using the *Either.Right* type to represent a successful operation and the *Either.Left* to represent a failed one.

The implementation of these different types is presented in the following list:

```
sealed class Either<out L, out R> {
    data class Left<out L>(val value: L) : Either<L, Nothing>()
    data class Right<out R>(val value: R) : Either<Nothing, R>()
}
```

3.5 Website

The main objective of the website is to provide an interactive user interface to allow the buyers of the neutralization mechanism a way of graphically visualizing the data that is being collected.

3.5.1 Single Page Application

We opted to implement a Single Page Application (SPA) instead of the more traditional approach, a Multi Page Application (MPA), since the SPA offers a clearer separation between the Frontend and Backend tasks.

With an SPA, the Backend does not need to handle page rendering or selection for the Frontend applications. For example, it does not have to handle the different UI layouts, depending on the different targets (Android, Web, Tablet, etc). Instead, this responsibility is delegated to the Frontend applications, allowing the Backend to focus on its main concern: data management. SPA also have the advantage of not reloading the page each time the user navigates or interacts with the website, thus allowing a more fluid navigation.

By adopting this strategy, we achieve a more modular and efficient architecture, where the frontend is responsible for handling the user interface and navigation, while the backend focuses on providing data and APIs to support the frontend functionality.

3.5.2 Deep Linking

Deep linking allows users to access specific pages or sections of our SPA directly through unique URLs. With Webpack, we were able to configure the necessary routing and mapping of URLs to corresponding components or routes within our application. This ensures that users can bookmark and share specific links, and the application will correctly navigate to the corresponding content. To enable *Deep Linking* functionality, we leveraged the capabilities of the *Webpack bundler*.

Each time the user refreshes the page or loads the website on a particular web page, the website will try to make a request to a URL. Since that URL is not registered, the browser should receive a 400s status code. Upon receiving a 404 status code, Webpack will fall back to a default URI, which will load the index.html. By loading the default page, the website will be loaded from the beginning, the API information fetched again, from the backend, and the *React Router* will render the appropriate component, based on the current URL. This behaviour is demonstrated in the ASM chart present in the Figure 3.16

Main Website Behaviour

When the *React App* component is loaded, the first thing to do is to load all the siren API information, from the Backend.

In a Siren-based API, resources are represented as JSON objects with specific properties such as ”class,” ”properties,” ”links,” and ”actions.” The ”links” property provides links to related resources, allowing clients to navigate between them. The ”actions” property describes the available actions or operations that can be performed on a resource, along with their associated input parameters. This is used to facilitate the Frontend development, since it is useful for the website to have to know only one the base URL and from this endpoint, it should be able to get all the API information, in order to navigate all the API endpoints, without having prior knowledge of the API structure. If this information cannot be obtained, the website will only show a static error message. This behaviour is because if the SPA is not able to obtain all the expected Siren Info, it would cause a bad user experience. For instance, it might not be able to access devices from the Backend. Therefore, the action is to display

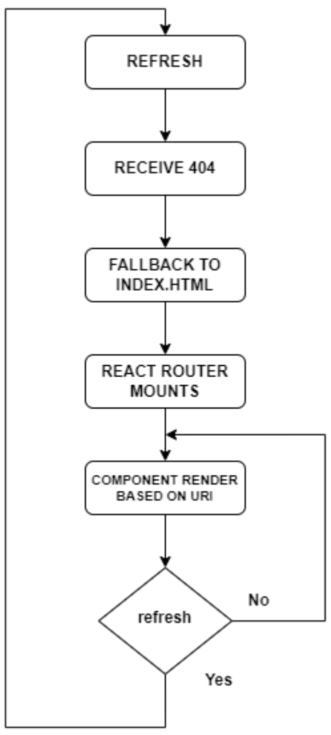


Figure 3.16: Webpack with Deep Linking

a message to the user, showing that something went wrong. A future idea would be to register the SPA developer to receive alerts each time this happens, since is considered a severe error.

One possibility which wasn't yet implemented is the use of local cache to store the API information. This ends up being a trade off. In a way, it is possible to make refresh loads without erasing local information, such as the siren information, or the login state. On the other way, it might be a risk, since the cache could become outdated. This is very unlikely in the case of the API/Siren information. But could emerge as an issue on the latter case (login state).

If the Siren information is obtained successfully, the SPA will setup the React Router, and the authentication state will be initialized.

Protected Resources

Most website resources/pages are protected, meaning the standard users should not be able to access them, but only authorized ones. Therefore, the website provides a sign-up/sign-in page, for authentication purposes. After users login, they will be redirected to an authorized page. Yet, nothing forbids the user from trying to access a protected resource directly, for instance by inserting the page URL manually in the browser search bar (deep linking). For some reason, there is a special component - **RequireAuthn** - that wraps the protected component, ensuring that the user is logged-in.

His behaviour is demonstrated in the ASM chart represented in the Figure 3.17.

By using this component, it ensures protected components are only accessed by authenticated users.

The login state is managed by the *AuthnContainer* component, and accessible to most SPA React components, so they can check if the user is logged in, and use that information, for example to make an API request.

The reader may be thinking: ”the authentication state might be safe in the *AuthnContainer* component, once the user logs-in. But what happens if the user refreshes the page, and the authentication state is lost?”. Should the user have to log in again?

In our system, we store a token inside a cookie to hold the login state. This is used, not for the browser to know that it is logged in, but for the server to know the user is logged-in. The source of truth lies in the token validity. Since the cookie, sent by the server, cannot be accessed in JavaScript, to protect from cross-site scripting (XSS) attacks, the browser/SPA needs to make an API request to know if the client is authenticated or not. This happens when the user uses deep linking to access a protected resource, and the authentication state is not yet known by the SPA. Yet, the token stored in the browser might still be valid, and he should not have to go through all authentication steps again. Similarly, to make the logout, the client browser makes an API request, so the server can remove the cookie, therefore eliminating the client session.

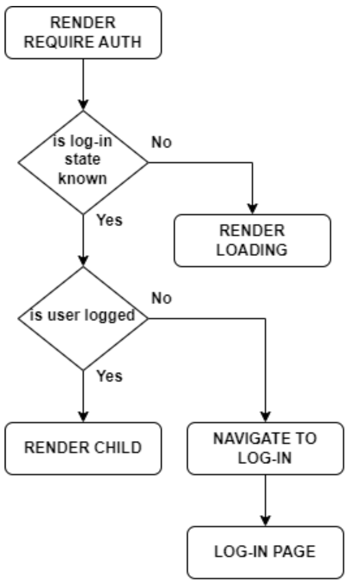


Figure 3.17: Authentication control in the SPA

Error Handling

Most unpredictable events will come from the Backend server (ex: when the user requests its own devices, sensor data, etc). In this situations, after making a request, the response could be a 400s or a 500s. The response will, most times, be a problem+json, and have a detailed description about what caused the error outcome. Despite this, it is not always partical for an SPA to know all possible errors, and know how to deal with them in particular. Some errors are expected. For instance, the React component that shows the login/sign-in page and makes the Backend request to create a User, will expect possible errors related to the credentials. Yet, in other request types, sometimes the error is a 500 internal server error and the website might not know how to deal with it. Therefore, the solution was to use a React component - *ErrorContainer* - that wraps another (the component that the user should see if everything goes well) and re-renders itself each time an error happens. By doing this, it is possible to control the user experience, specially when something unexpected happens, by not just letting the error pass unnoticed.

Services

To facilitate website development, we created a Service interface along with two implementing classes: *FakeServices* and *RealServices*. The *Service* interface defines the contract for interacting with the backend server API. The *FakeServices* class was specifically designed to support frontend development by simulating the behaviour of the backend server API. This allows developers to work independently without directly relying on the actual backend implementation.

However, in this report, we will not go into detail of the *FakeServices* class since its primary purpose is to provide a mock implementation for frontend development. Instead, we will focus on the *RealServices* class, which represents the actual implementation that communicates with the Backend Server API.

The majority of the service functions need to make an API request, hence they need to know the specific URL, the HTTP method and other information. The backend server API works with the media type: Siren. In particular, the only known URI endpoint - siren-info - responds with all the information the website needs to know to navigate the API. The function *getBackendApiInfo()* will make a request to this endpoint, and extract all necessary information. This includes the Siren Links and Actions, allowing the website to function.

When a service function needs a link or action that is not yet known, it throws an exception, so the caller can know the problem. They can also throw exceptions for other reasons. For instance, the fetch function will throw an exception if the API response in not the one expected, i.e. 400s. If the response is a 400s, the fetch will try to convert the error response into a *Problem JSON*, throwing an exception with the *Problem* title, allowing the caller to know more information about the request error response.

3.6 System Tests

The main system logic is concentrated in the Spring Backend server. Therefore, extensive tests were made to this component. In our development process, we have placed a strong emphasis on automated testing, particularly for the server-side layers of our application. This includes testing the Services, Repositories, Domain, and Controllers through integration testing.

To test the MCU firmware, we opted for simple programs that need to be loaded in the hardware. These are not automatic tests as with the Spring Server. For instance, to calibrate the pH sensor, a simple program that only reads from the sensor, is loaded into the hardware and the developer observes the logs, which print the values. Next, the values are copied and pasted in a .csv and loaded by a python script to display the values, and thus, providing a more comfortable analysis.

Another example of testing is to use a python script that emulates the hardware device to send synthetic sensor data to the Backend application server, so it is possible to test the server receiving sensor data, without relying on the real hardware.

4
Conclusion

During the development of the project we had the opportunity to exercise the technologies we learned throughout the course such as React and Spring Framework and that are widely used in the industry.

We also had to learn to use other technologies and protocols such as time series database and MQTT protocol and beyond that we had contact with hardware, more specifically the programming of the ESP32-S2 microcontroller, which was one of the main challenges during the project since both elements of the group doesn't have any experience in this area.

Despite these challenges, the group managed to successfully meet the requirements that were proposed for the project.

Bibliography

[1] "Mommertz," Mommertz Corporation. [Online]. Available: https://mommertz.de/

[2] "Internet of things," Amazon. [Online]. Available: https://aws.amazon.com/what-is/iot/

[3] "Microcontroller," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Microcontroller

[4] "Web api," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Web_API

[5] H. Cheung, "Measure ph with a low-cost arduino ph sensor board." [Online]. Available: https://www.e-tinkers.com/2019/11/measure-ph-with-a-low-cost-arduino-ph-sensor-board/

[6] "Dht11–temperature and humidity sensor," components101. [Online]. Available: https://components101.com/sensors/dht11-temperature-sensor

[7] B. Lutkevich, "Embedded system." [Online]. Available: https://www.techtarget.com/iotagenda/definition/embedded-system

[8] "Sleep modes," Espressif. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html

[9] "Esp32-s3 series datasheet," Espressif. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf

[10] "Esp32-s series datasheet," Espressif. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf

[11] "Esp8266ex series datasheet," Espressif. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[12] "Esp32-c3 series datasheet," Espressif. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf

[13] "Msp430fr413x series datasheet," Texas Instruments. [Online]. Available: https://www.ti.com/lit/ug/slau595b/slau595b.pdf?ts=1684329937122

[14] "Introduction to spring framework," Spring Framework Reference Documentation. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html

[15] “What are the benefits of kotlin,” Sagara Technology Idea Lab. [Online]. Available: https://sagaratechnology.medium.com/what-are-the-benefits-of-kotlin-d7fdcd1cfc0

[16] “Time series database,” Hazelcast. [Online]. Available: title={https://hazelcast.com/glossary/time-series-database/},

[17] “Relational databases vs time series databases,” Influxdata. [Online]. Available: https://www.influxdata.com/blog/relational-databases-vs-time-series-databases/

[18] J. Tao, “Time-series database basics.” [Online]. Available: https://devops.com/time-series-database-basics/

[19] “What is mqtt?” Amazon. [Online]. Available: https://aws.amazon.com/what-is/mqtt/

[20] L. Dallinger, “Http vs mqtt: Choose the best protocol for your iot project.” [Online]. Available: https://cedalo.com/blog/http-vs-mqtt-for-iot/

[21] “The free public mqtt broker,” HiveMq. [Online]. Available: https://www.hivemq.com/public-mqtt-broker/

[22] “Hivemq security,” HiveMq. [Online]. Available: https://www.hivemq.com/solutions/technology/security/

[23] “Webpack documentation,” Webpack. [Online]. Available: https://webpack.js.org/concepts/

[24] “Field data management know to grow,” Cropx. [Online]. Available: https://cropx.com/cropx-system/field-data/

[25] “Postgres - about,” Postgres. [Online]. Available: https://www.postgresql.org/about/

[26] “Analog to digital converter,” Expressiff. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/release-v4.2/esp32s2/api-reference/peripherals/adc.html#adc-calibration

[27] “Non-volatile storage library,” Expressiff. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/release-v4.2/esp32s2/api-reference/peripherals/adc.html#adc-calibration

[28] “Spring framework documentation - interface handlerinterceptor,” Spring Framework. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html

[29] “Spring framework documentation - interface handlerinterceptor,” springdoc-openapi v1.7.0. [Online]. Available: https://springdoc.org/