chapterIntroduction sectionMotivation

sectionObjectives

sectionReport Outline/Document Organization

chapterBackground

sectionIoT Systems RD: breve intro aos sistemas IoT.

subsectionSensors sensor de pH - descrever funcionamento - vou partilhar convosco descrição do circuito de pH

subsectionEmbedded Systems falar da função e dos varios candidatos - trazer para aqui a parte do "WhY"

subsectionBackend

sectionExisting Solutions RD: trabalho do Bernardo e outros + soluções industriais/profissionais Acrescentar pequena comparitiva das soluções

chapterProposed IoT System Architecture sectionIntroduction

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

01 – 05 – 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 feed 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.

Contents

List of Figures

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. 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 behavior 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 some 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) technology to automate the monitoring and maintenance of the filtration system. By integrating a microcontroller unit (MCU) 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 advanced 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. 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:

- Considering that the microcontroller unit will be integrated into an existing product, it is important to take the cost of the unit into account as a significant factor.

- For the system to minimize the need for maintenance, it is essential to have a prolonged battery life.

- The creation of comprehensive documentation is crucial to ensure that individuals with varying levels of background can understand and effectively operate our system.

- Develop clean, easily maintainable code that facilitates unit testing for each component.

- To develop a system that is able to read the pH data from inside the filter and share that data with a server

- Send a message to the owner or manager of the filter if any value is over a predetermined threshold

- To design a highly intuitive and user-friendly website that enables easy navigation and access to the necessary 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.

1.3.1 Introduction

In this 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.

1.3.2 Background

The project report's background 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.

1.3.3 Proposed IoT System Architecture

The purpose of this 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 IotSystems

The Internet of Things (IoT) technology influences a variety of aspects of our daily lives. In just a few years, it has already impacted the lives of millions of people.These systems are composed of both digital and physical elements. 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, and more

2.1.1 Sensors

To achieve our ultimate 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 concentrate. It serves as a vital tool for evaluating the productivity of the 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. 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.

As shown in the figure, 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 these data, the I/O pin has to be momentarily made low and then held high as shown in the timing diagram below:

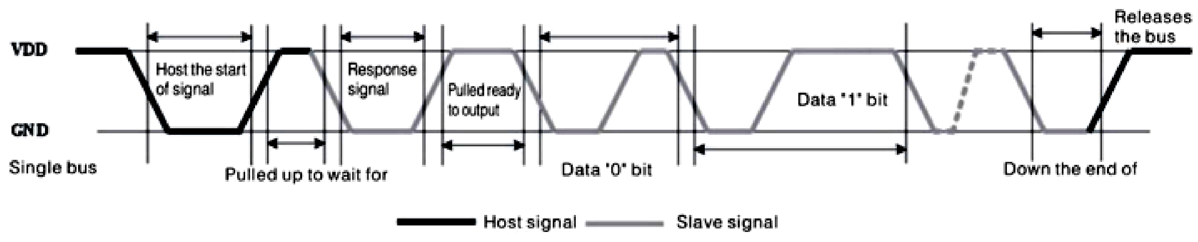

Figure 2.3: DHT11 Sensor Module

To detect water leaks in the filter, the "Water Sensor" module is a good option. Is very cheap and easy to use. This simple sensor is 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.

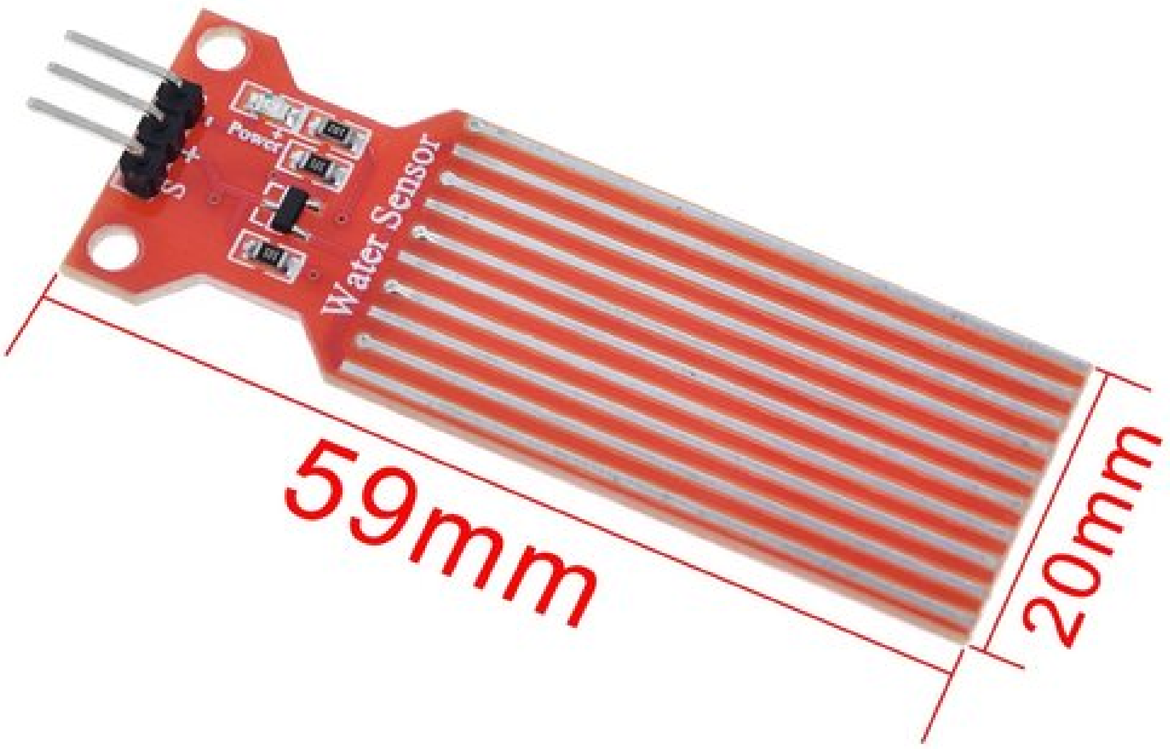

Figure 2.4: DHT11 Sensor Module

2.1.2 Embedded Systems

An embedded system is a microprocessor-based computer hardware system with software that is designed to perform a dedicated function.

When a project includes the collection of sensor data, most times, a microcontroller is needed, to compute all this collected data. He is the one who devices when the data should be collected, how to store it and how it is transited.

2.1.2.1 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 important factor of this project was the overall low cost of the system, since we are imitating an industrial project as close as possible. The battery consumption of the selected microcontroller was another crucial consideration, since 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 board, but we also took into account MCUs from other manufactures 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

Figure 2.5: 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

Figure 2.6: 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. 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. 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, fulfills our project requirements adequately. 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 remarkable choice for our project due to its exceptionally low power consumption, despite having a slightly less powerful processor compared to the other options. 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 boasts a remarkable combination of features, including low power consumption, a built-in Wi-Fi module, and similar characteristics to the ESP32 - S2. 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 boasts superior power consumption, making it an energy-efficient choice. However, it lacks a built-in Wi-Fi module, which presents a challenge. 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, 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.

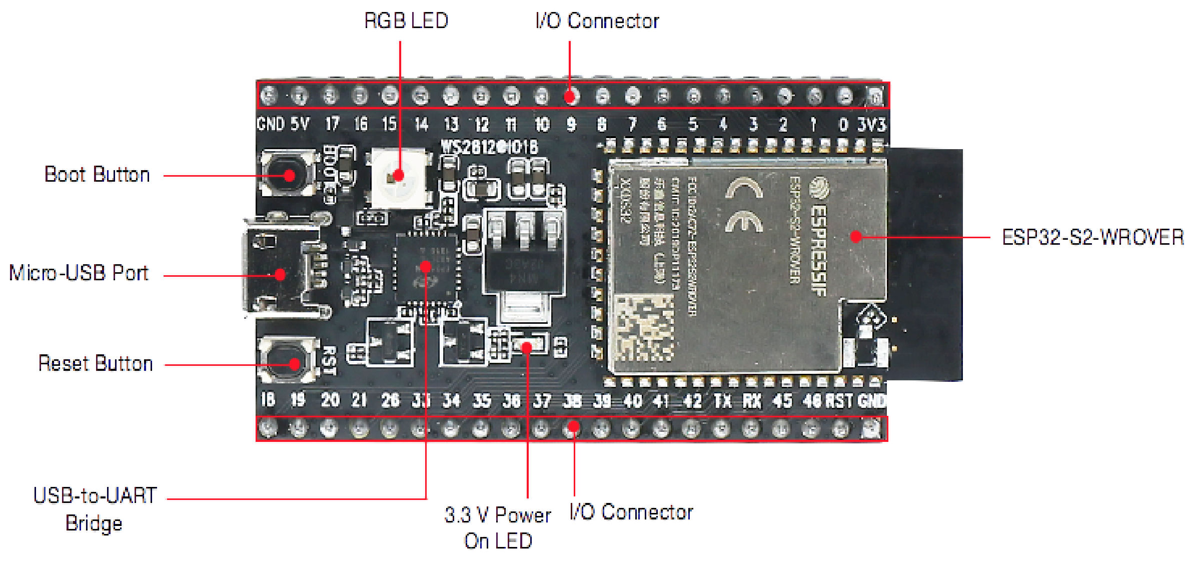

Figure 2.7: Módulo Wi-Fi - Espressif ESP32-S2-SAOLA-1M (WROOM)

2.1.2.2 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 allow us to optimize performance in a variety of tasks. Another advantage is the strong documentation freely available. On the other hand, many tasks, for instance connecting to the WiFi or interacting with the sensors, becomes a complex task duo to the non-existent abstraction that other more general 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:

2.1.3.1 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. The use of Spring 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. 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.

2.1.3.2 Database

Data processing and storage: The backend receives data from IoT devices, processes it, and stores it in a 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). They offer faster querying and can handle large volumes of data. 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. In order 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

To allow the interaction between different elements within our system, we had to verify and analyse different communication protocols to take the best decision for each specific route. We aggregated in the following table some of the most interesting protocols in the ambit of our project:

HTTP:

Advantages:

- Well-established protocol, widely used and supported by many devices and platforms.
- Easy to implement and understand, with numerous libraries and tools available for working with it.
- It can operate over both TCP and UDP, providing flexibility in network configuration.
- Allows easy integration with web technologies, such as web servers and RESTful APIs.

Disadvantages:

- Not very efficient in terms of bandwidth utilization, especially compared to lighter protocols like MQTT and CoAP.
- It can be slow and resource-intensive, particularly when dealing with large amounts of data.

- Requires a complete request-response cycle, which can introduce additional latency in real-time applications.
- May not be suitable for use cases with limited or intermittent network connectivity, as it relies on persistent connections.

MQTT:

Advantages:

- Lightweight protocol, making it suitable for IoT devices with limited processing power and memory.
- Simple and easy to use, making it popular in the IoT community.
- Supports both TCP and TLS, providing secure communication over the network.
- Reliable message delivery, with the option to implement Quality of Service (QoS) levels.

Disadvantages:

- Limited support for large payloads, which can be problematic when transmitting large amounts of data.

DDS:

Advantages:

- Specifically designed for distributed systems, making it suitable for IoT applications.
- Supports a wide range of data types and message formats, making it flexible.
- Offers advanced features such as real-time communication, data filtering, and automatic device discovery.

Disadvantages:

- More complex than MQTT, which can make its implementation more challenging.

- Requires more processing power and memory than MQTT.
- It may not be as widely adopted as MQTT, limiting the availability of tools and resources.

AMQP:

Advantages:

- Supports both reliable messaging and publish-subscribe message patterns, making it flexible.
- Offers advanced features such as message batching, message priorities, and message expiration.
- Designed to be platform-agnostic, making it suitable for heterogeneous IoT systems.

Disadvantages:

- More complex than MQTT, which can make its implementation more challenging.
- Requires more processing power and memory than MQTT.
- It may not be as widely adopted as MQTT, limiting the availability of tools and resources.

CoAP:

Advantages:

- Specifically designed for IoT devices, making it lightweight and suitable for constrained environments.
- Supports both request-response and publish-subscribe message patterns.
- Offers advanced features such as caching, discovery, and resource observation.
- Supports both UDP and TCP, providing flexibility in network communication.

Disadvantages:

- Lack of widespread adoption compared to MQTT, which can limit the availability of tools and resources.

- Limited support for security features, although this can be addressed with DTLS.

After careful consideration, we opted for the MQTT protocol to allow the comuntication between 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. 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.

2.1.4.1 Broker

One of the fundamental components of the solution is the presence of a broker. In 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. We evaluated various alternative brokers that could meet this requirement in our project:

Broker	**Description**	**Key Features**
HiveMQ Community Edition	Open-source MQTT broker	- Lightweight and scalable - High-performance messaging - Community support
Mosquitto	Open-source MQTT broker	- Lightweight and easy to install - Simple and reliable - Good community support
RabbitMQ	Feature-rich message broker	- Supports multiple protocols - Message queuing and routing - Persistence and scalability
EMQ X	Scalable and highly available MQTT broker	- MQTT 5.0 support - Clustering and fault tolerance - High-performance IoT applications
VerneMQ	Distributed MQTT broker	- Clustering and scalability - Message replication and persistence - MQTT 5.0 support
AWS IoT Core	Managed MQTT broker	- Secure and scalable messaging - Integration with AWS services - Device shadow and management
IBM Watson IoT Platform	Cloud-based MQTT broker	- Robust and integration capabilities - Device management and visualization - Analytics and data processing

Figure 2.8: Alternative MQTT Brokers

To implement the broker, we have decided to utilize the free version of the HiveMQ broker. This selection aligns with our requirements, particularly our focus on message security. The HiveMQ broker incorporates TLS/SSL encryption, 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.

2.1.4.2 MCU Configuration

When developing an IoT system, it is always interesting to allow the operator/user to configure it's own IoT Device, for example the MCU, with it's own specific desired settings, 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 WiFi-enabled micro controllers. The free ESP Touch mobile app uses this protocol to connect to the ESP Device, and thus, transmitting, not only the WiFi configuration, but also a sting of bytes of custom data.

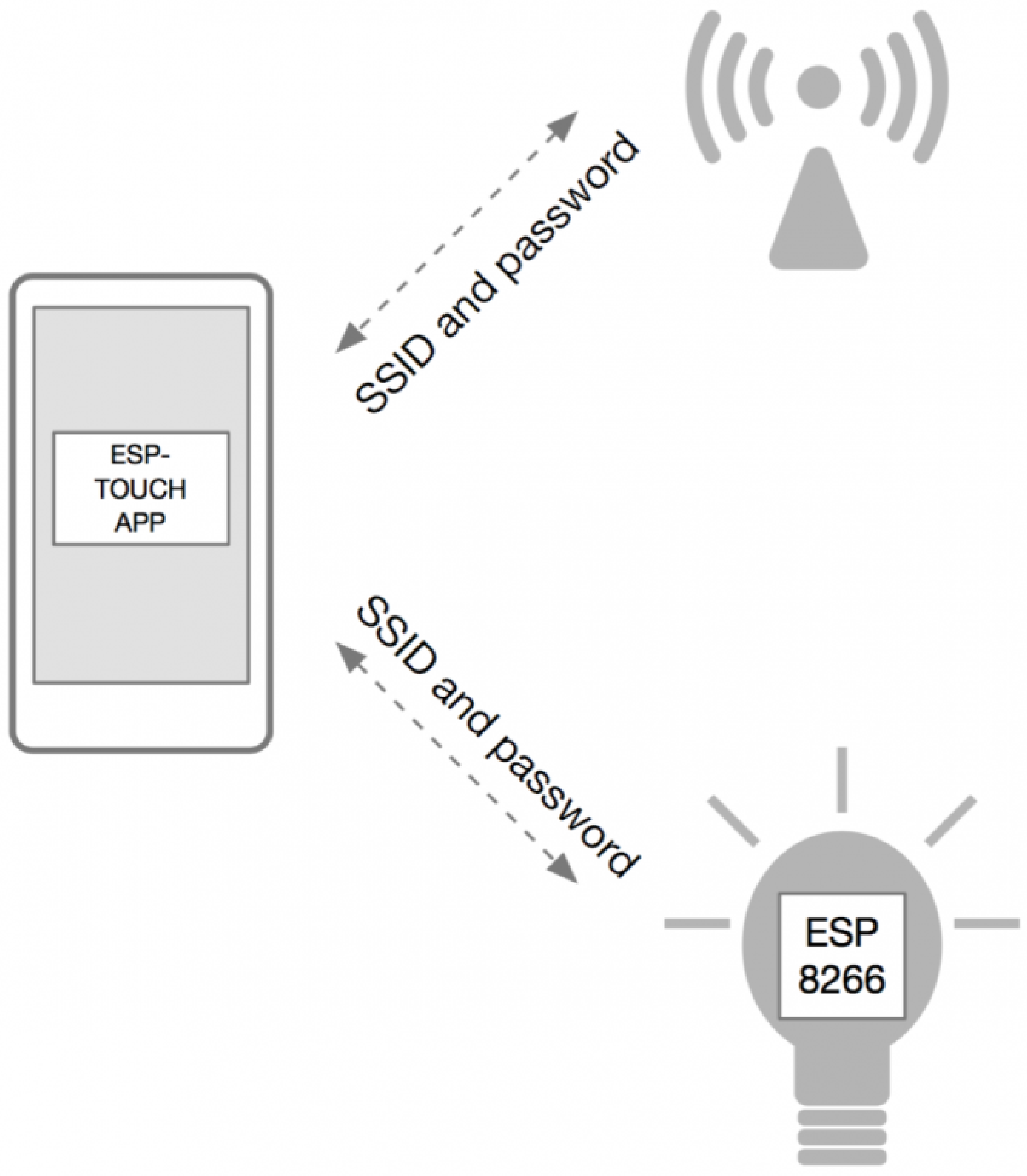

Figure 2.9: ESP Touch - Smart Config Protocol in the ESP8266

Therefore, duo to our choice of working with the ESP32-S2, we chose to integrate this protocol in our system to deal with the MCU configuration.

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 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. After careful consideration, we identified the following as the most significant ones:

Web App Client Frameworks		
Framework	Advantages	Disadvantages
React	High performance, easy to learn, reusable components, good for complex applications	Steep learning curve for beginners, requires a deep understanding of JavaScript
Angular	Comprehensive, good for large-scale applications, good tooling support, well-suited for enterprise applications)	Steep learning curve for beginners, can be overly complex for smaller projects
Vue.js	Lightweight, easy to learn, good for smaller projects, high performance	Limited ecosystem compared to React and Angular
Django	Comprehensive, follows MVC architecture, includes an ORM, good for rapid development, good for large-scale projects	Can be overly complex for smaller projects, limited flexibility compared to microframeworks
Flask	Lightweight, flexible, easy to learn, good for smaller projects, good for rapid prototyping	Limited functionality compared to comprehensive frameworks like Django

Figure 2.10: Web App Client Frameworks Comparison

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.

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.

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.

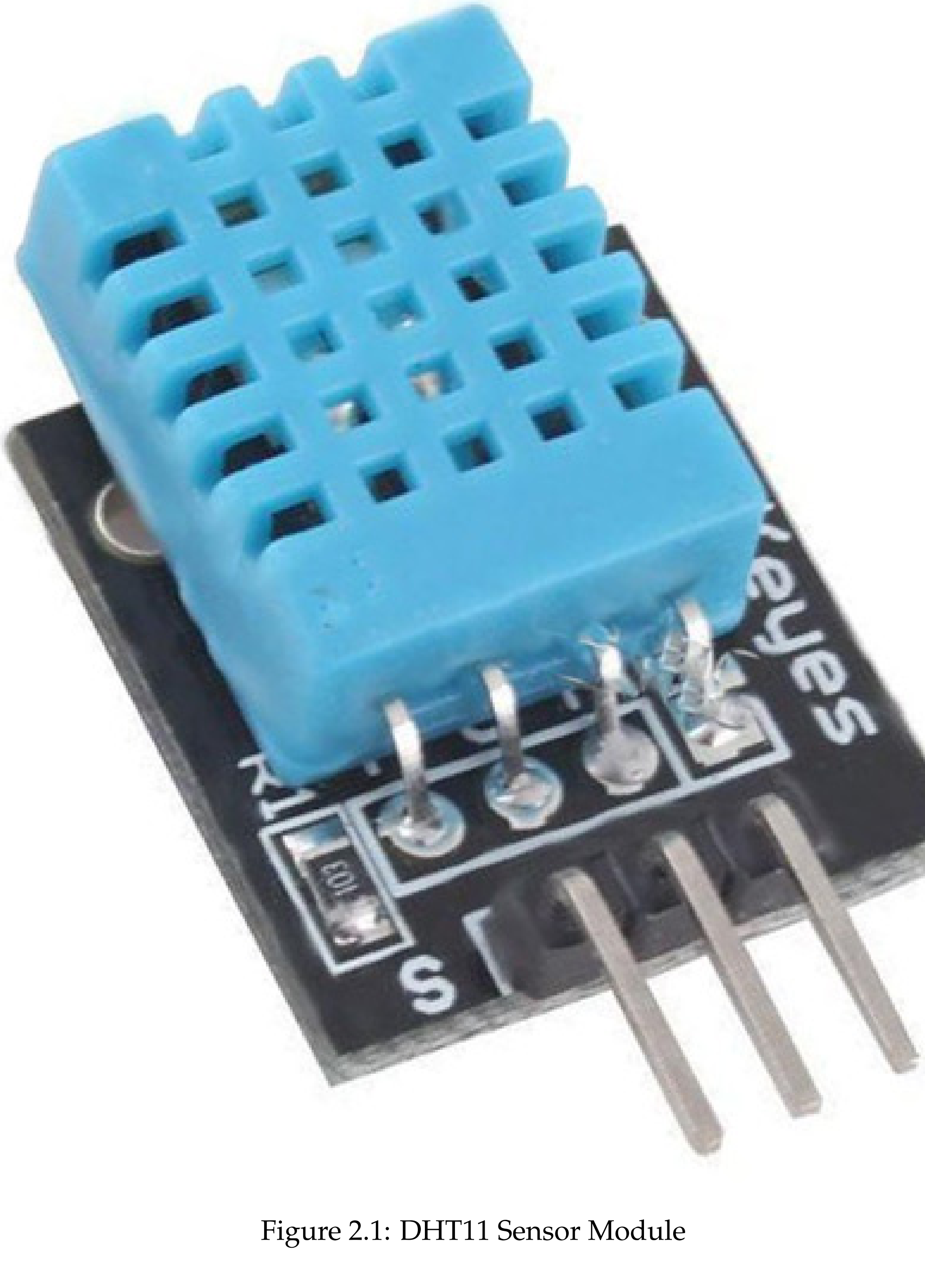

Figure 2.1: DHT11 Sensor Module

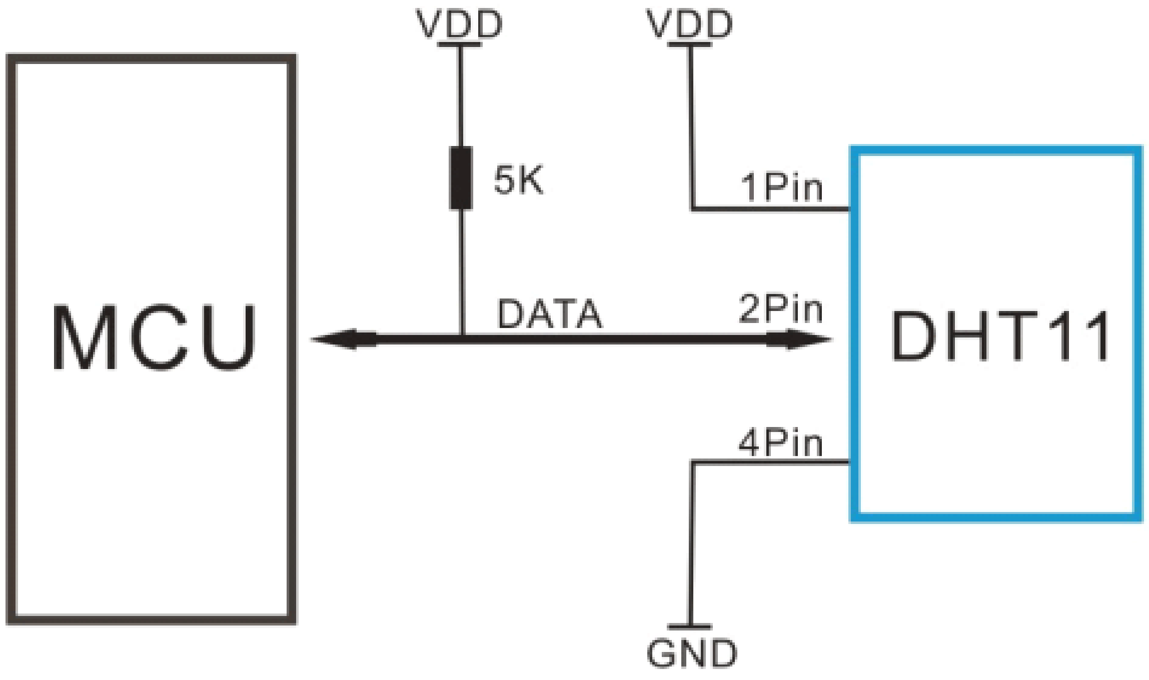

Figure 2.2: DHT11 Sensor Circuit

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.

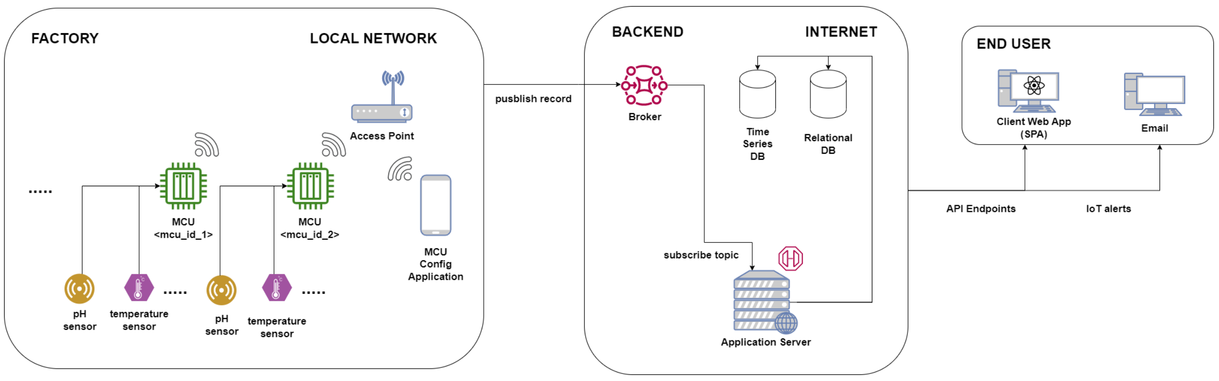

Figure 3.1: Main system architecture

Overall, 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 *Factory* 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 exposes an API, where other applications, like the one described in this report - web application - will use to interact with the system, for instance to query for available sensor data, create users, add devices, etc.

3.2.1 Factory

This component will be replicated for each client using the system described in this report.

Each one involves 4 subcomponents:

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

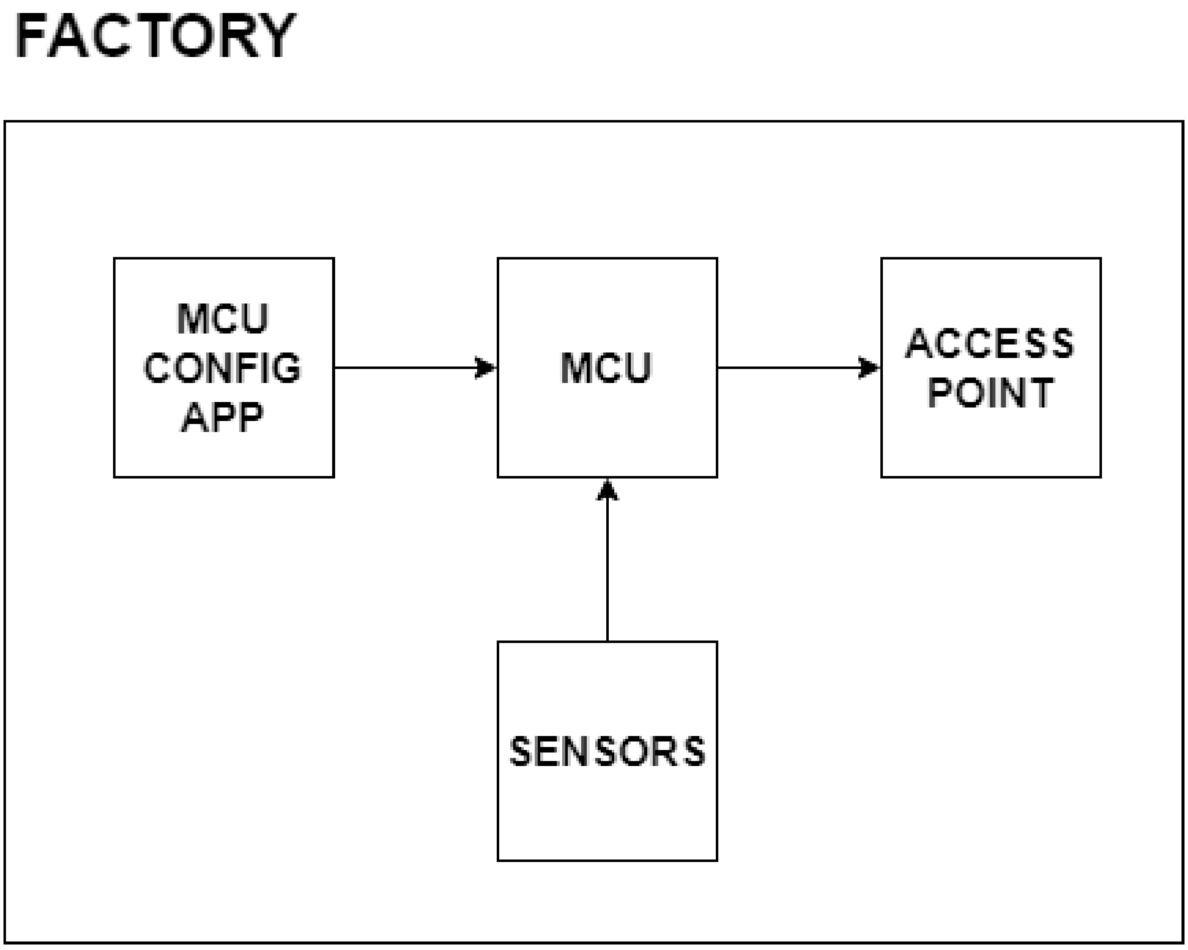

Figure 3.2: Factory Module Architecture

The user interacts with an android app provided by **Expressif** called **Esptouch** to configure the MCU, for the first time. This includes setting up a device identifier and sending the Wi-Fi 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. EXPLICAR O PROCESSO DE CONFIGURAÇÃO

3.2.1.1 MCU

This is the MCU firmware component architecture:

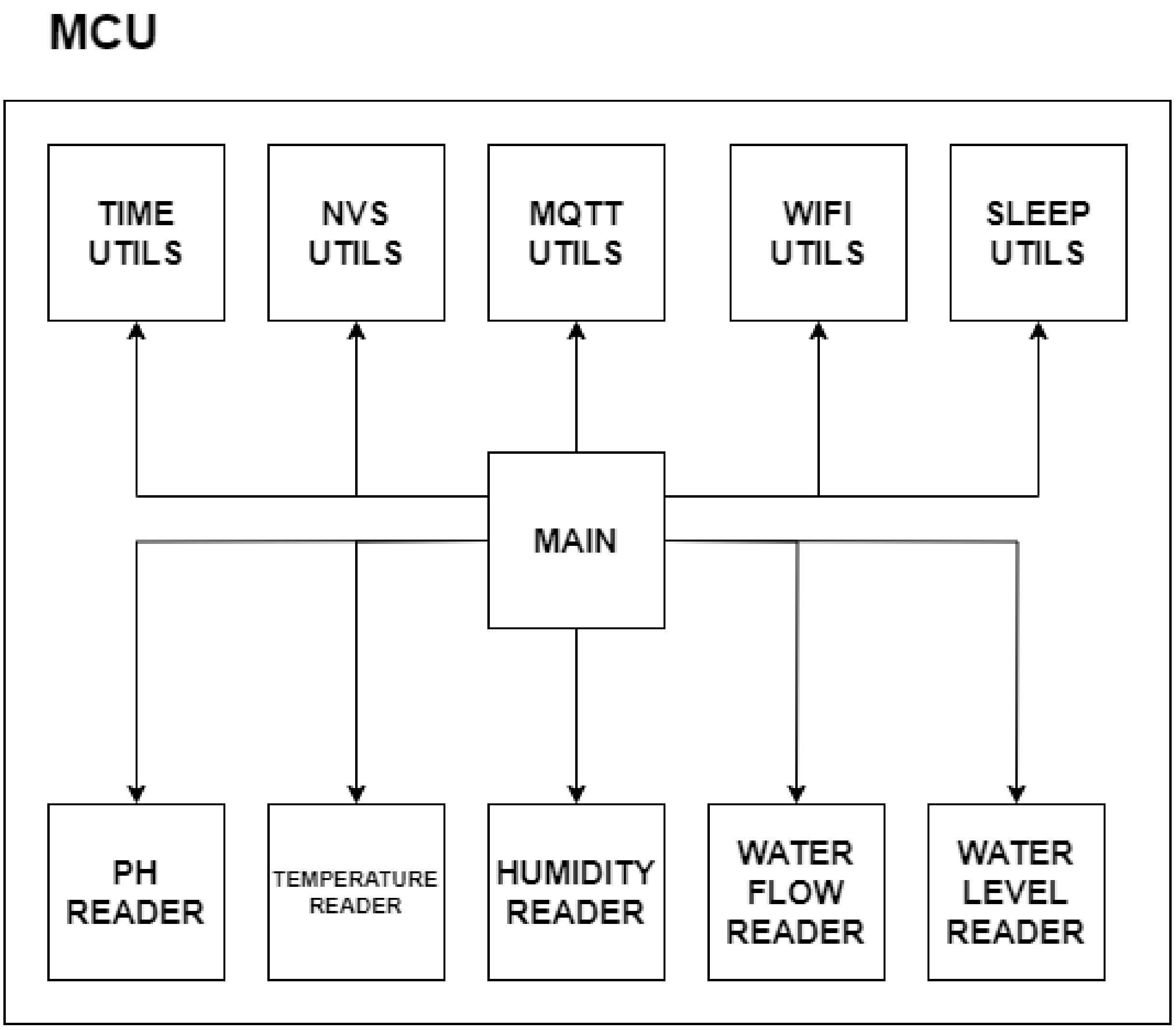

Figure 3.3: MCU firmware architecture

The Main component controls the main flux algorithm. It interacts with other utility components, like the sensor reader modules to collect data from the sensors, the MQTT to send data using the MQTT protocol, the time module to obtain accurate time, etc.

3.2.1.2 Sensors

There are a total of seven sensors:

- Initial pH sensor to obtain the solution pH in the beginning of the filter system (before being neutralized);
- Final pH sensor to obtain the solution pH in the end of the filter system (after being neutralized);

- Ambient temperature sensor;
- Air humidity sensor;
- Water level sensor to detect if the filter water level is too high;
- Water flow sensor to estimate the amount of water that flows in the system;
- Water leak 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.

3.2.1.3 MCU Config App

Inside the room, where the MCU is located, a simple 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

Unlike he Factory component, this one is unique and shared by all the clients having one or more neutralization systems. Typically, this will be located in the Cloud and will function as an IoT platform to receive and handle all system data, namely the collected sensor data.

3.2.2.1 Application Server

This is the component that computes and persists all the system involved data. All the business logic will be enforced here. It also exposes a Web API, so that Frontend applications can interact with the system.

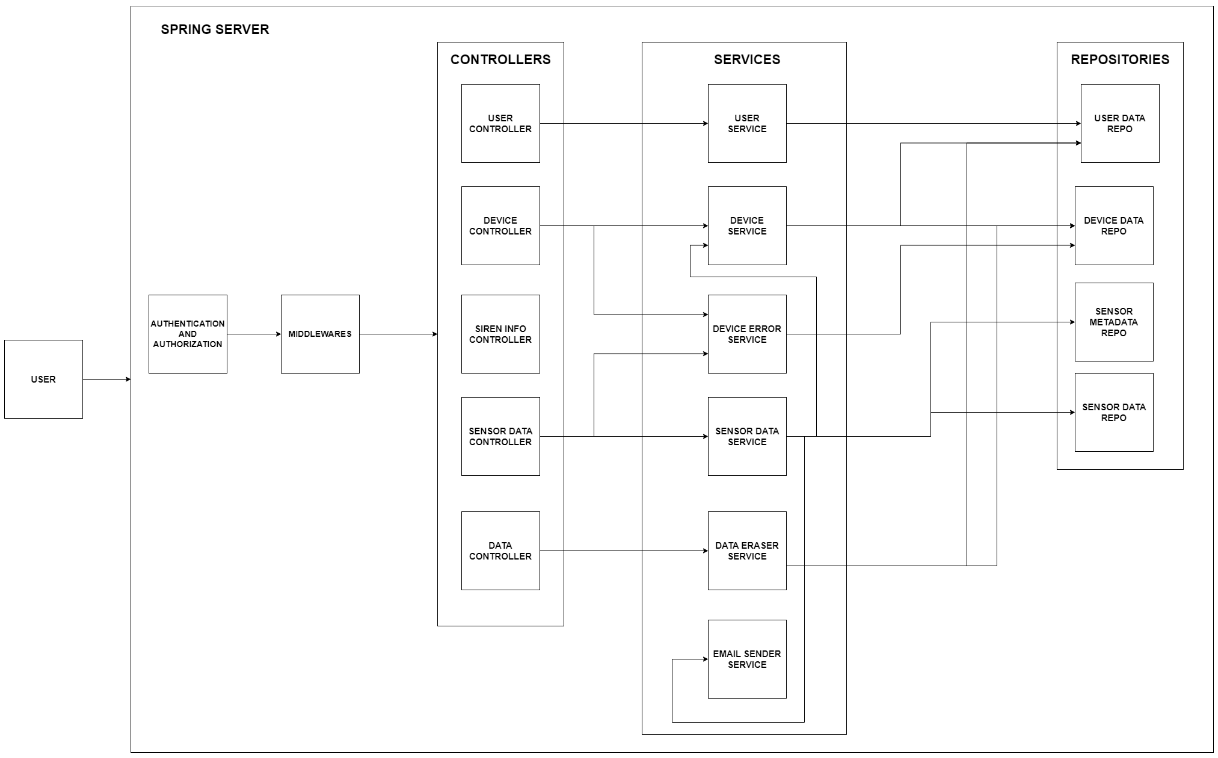

Figure 3.4: Spring Server Overview

The figure above shows the main modules involved:

- Authentication and authorization modules will be used when the request is attempting to access protected resources;
- Other Interceptors, for instance to log the requests;
- The controllers to define the API operations: endpoints + request formats;
- 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.

Each time a client sends a request, the Controllers will use the Services to fulfil the request. Likewise, when the services need to access/store data, they will utilize 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.

Most Backend layers are divided into the following components:

- User: Handles all operations related to system users, such as user creation, retrieval, and deletion;
- Device: Manages all operations related to devices, including device creation, retrieval, and deletion;
- Sensor Data: Handles all operations related to sensor data, such as requesting available sensor types for each device and retrieving sensor data.

3.2.2.2 Broker

The Broker stores the data that is sent from the IoT devices, in the Factory. In our specific use case, the Broker will send the data straight to the Server.

3.2.2.3 Database

Duo 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. PostgreSQL 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.

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 relational database has the following scheme:

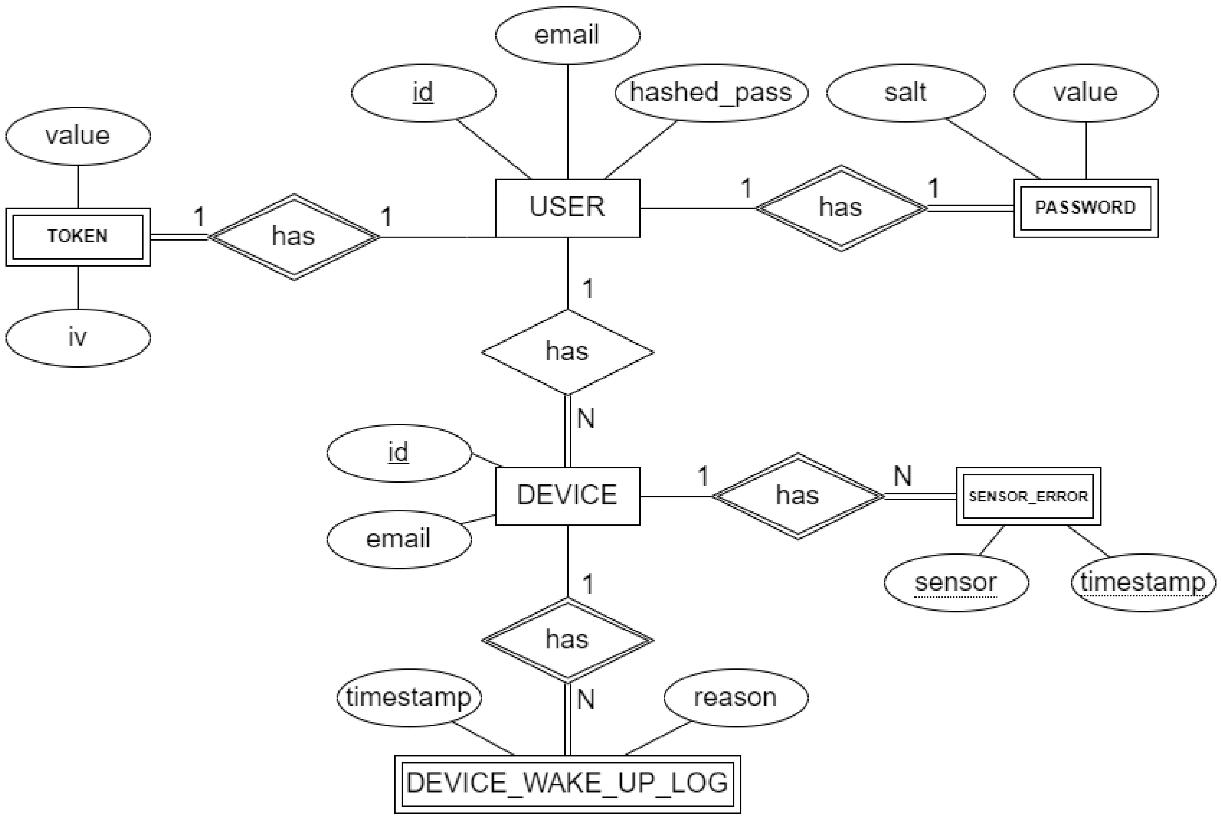

Figure 3.5: Postgres DB Entity Model

The user can have multiple devices. 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:

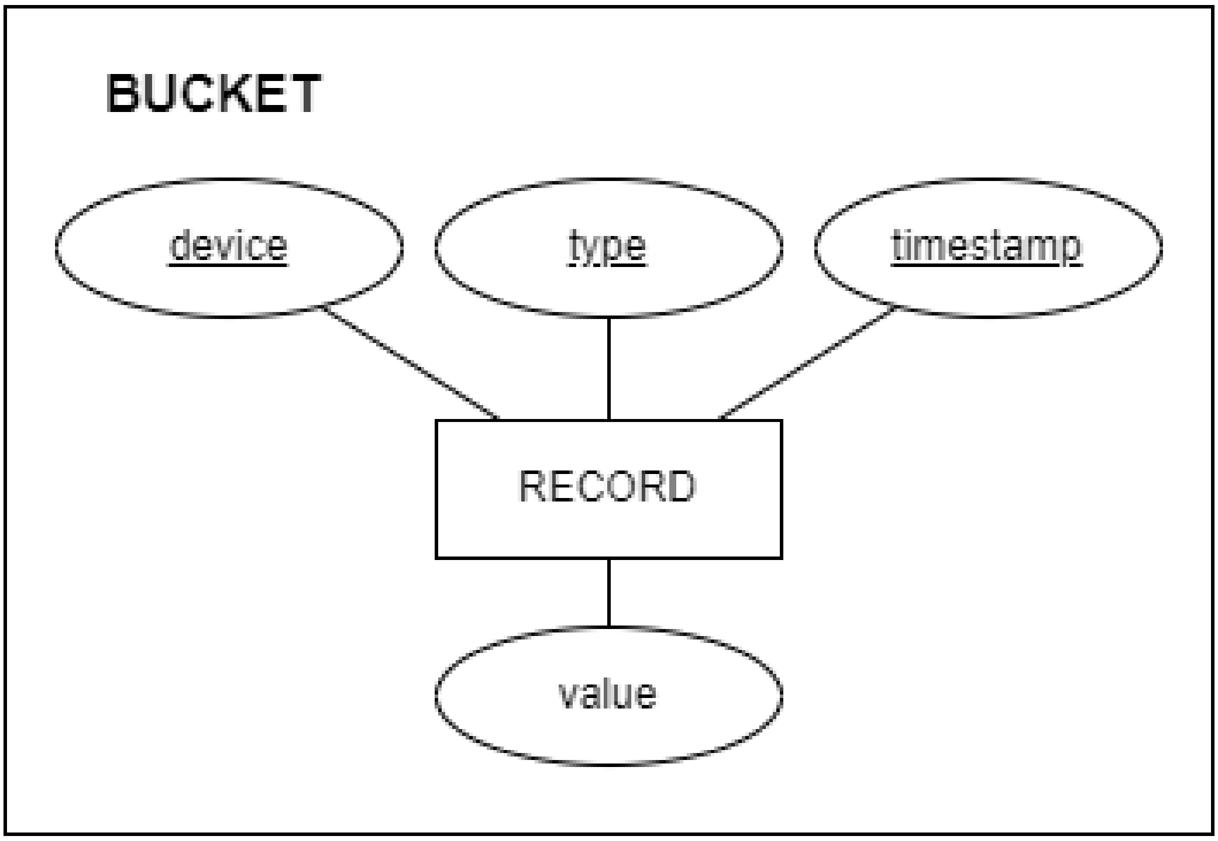

Figure 3.6: Postgres DB Entity Model

A record is always composed of the device, the type/sensor, timestamp and the value/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 the following:

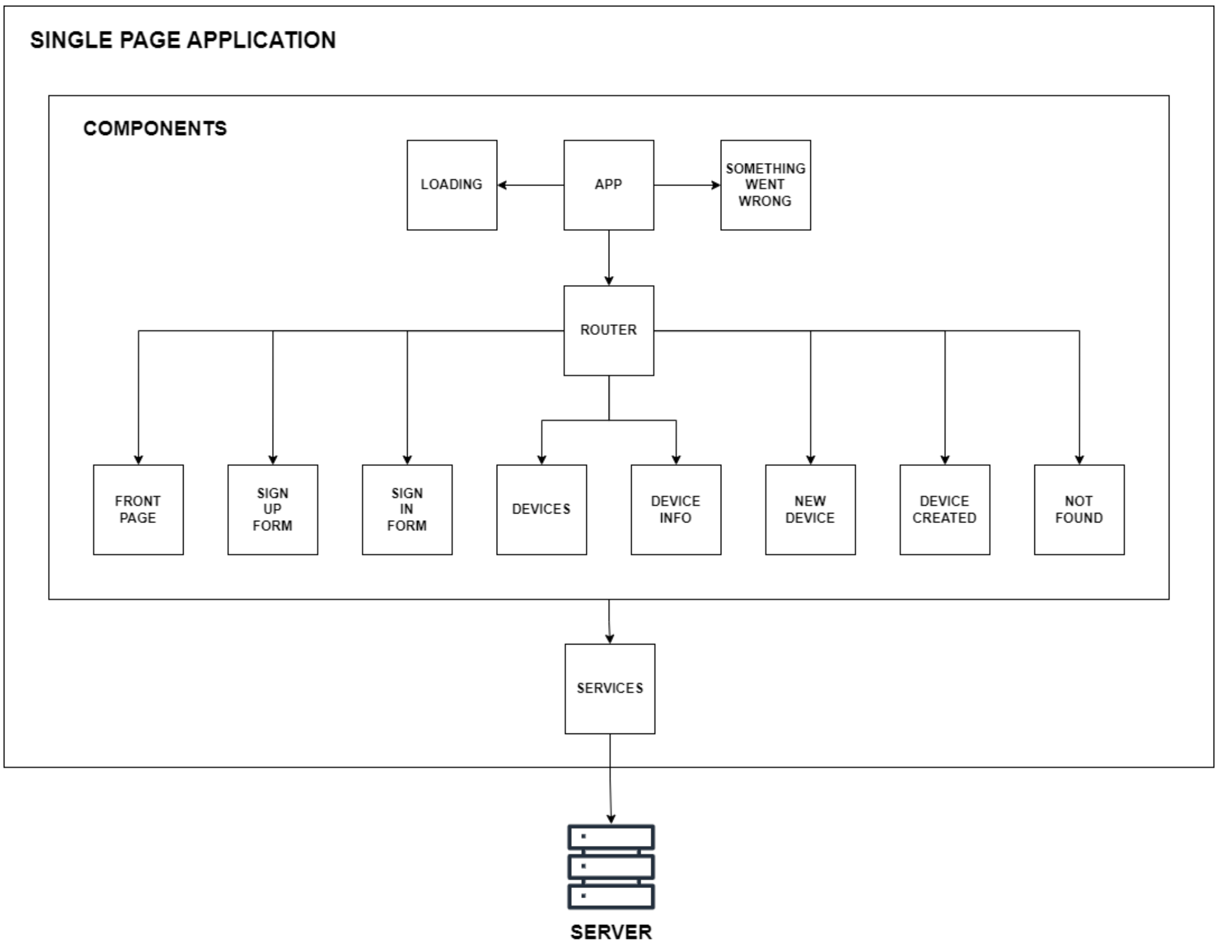

Figure 3.7: Website Diagram

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/component to access the Backend Server. This allows them to request and manipulate system data efficiently. 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 Factory

3.3.1 MCU

3.3.1.1 Behaviour

The MCU behaviour can be summarized by the following ASM (Algorithmic state machine) chart:

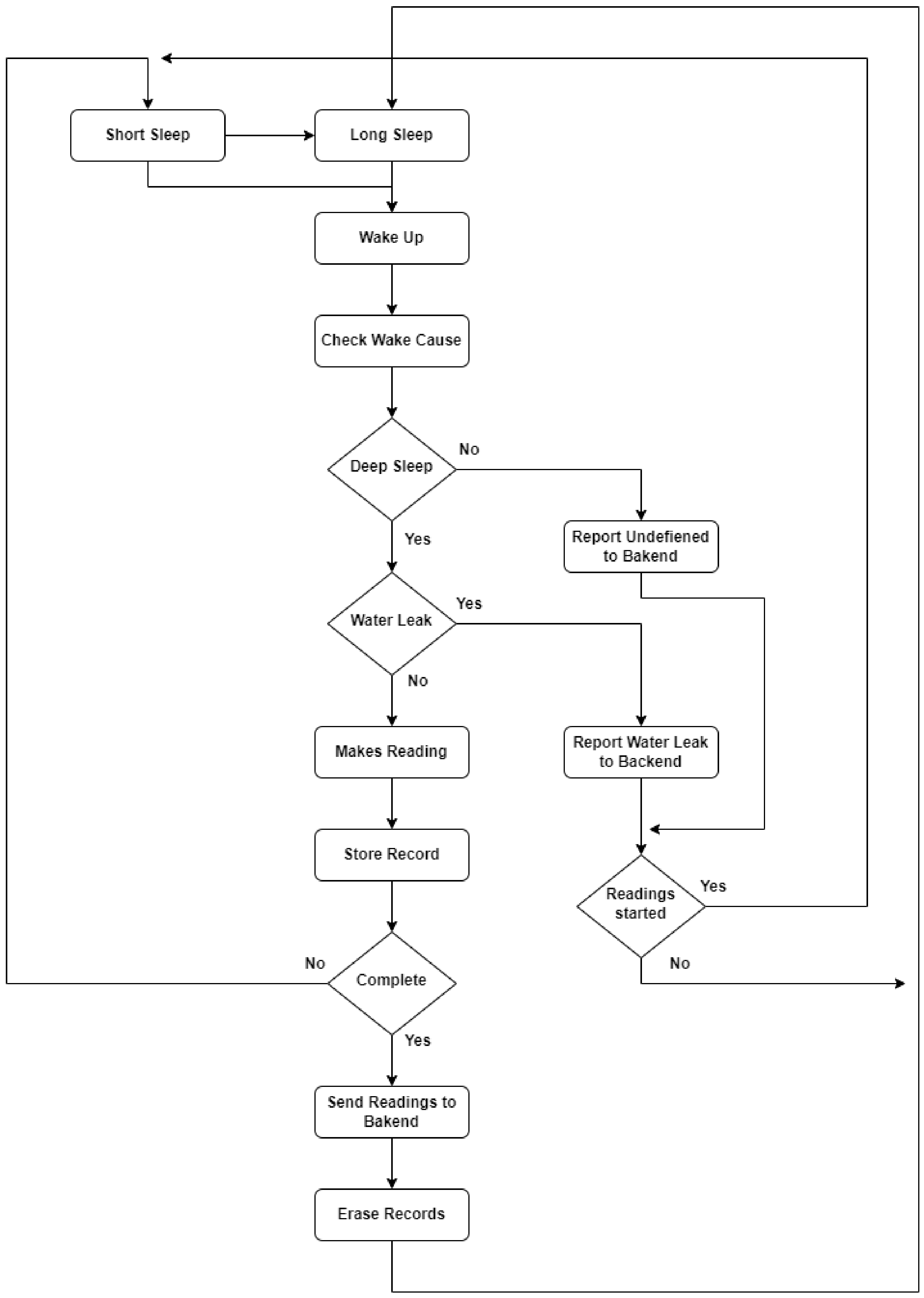

Figure 3.8: ESP Touch - Smart Config Protocol in the ESP8266

Most of the time, the MCU will be sleeping. This is because it is not worth it to make constant readings. The filter conditions will not change every minute. Besides, since the power consumption is a concern, each second the MCU is running counts!

When the device wakes, it must check its cause. This is because it can happen for a variety of reasons. The device can wake up only because it was powered on. Yet, the most interesting reasons, for our specific use case, is to know if it woke up from a *deep sleep* or a *panic exception*.

Typically, the occurrence you described happens when the device encountered an unexpected termination or error during its previous execution. The ESP32-S2 will re-start execution when this happens. In this project, IoT devices are programmed once and deployed after. Even if the programmer is still working on bugs, if the "board" is deployed in the factory 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.

The next thing to do is checking if the wake cause is duo to a water leak. This is something that can happen in the neutralization filter. Again, the Backend is immediately reported about such event.

In all this cases, a log is reported to the Backend, using the MQTT protocol to send a message to the Broker, informing the device wake up reason.

In the case the device wake up reason is duo to *deep sleep*(a low-power mode that allows the MCU to significantly reduce its power consumption while preserving the internal state and certain functionalities wake-up timer) wake-up timer, since it is not an abnormal wake-up, it is not necessary to inform the Backend.

One potential approach considered was to report the sensor readings to the Backend immediately after they were obtained by the sensors. However, in a neutralization system, the readings can oscillate a bit. That is why we choose to make some readings, intercalated by a fixed time, and then compute the average and send the result to the Backend. As said above, each minute the MCU is running counts, and so, to avoid wasted power consumption and computation, this one goes to sleep for a short time, in case the readings are not completed. Otherwise, the readings result is sent to the Backend.

In general, the MCU will follow the following sequence of operations: wake up, report the cause of the wake-up to the Backend. If the wake-up cause is abnormal, it will proceed to make sensor readings and send them to the Backend. If the wake-up cause is deemed appropriate, the MCU will go back to sleep mode again.

All this behaviour takes place in the main MCU module.

3.3.1.2 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 time can be changed in the code itself. The other reason is what we called "Short Sleep". This is when the MCU is sleeping for a short time, ideally seconds, waiting to make another reading. This is accomplished by using the deep sleep mode, that is available on the ESP32-S2, in order to conserve the most amount of energy.

3.3.1.3 Sensor Modules

There is a total of seven sensors. Consequently, 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 dual implementation 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.

To deploy the pH sensor, it was first necessary to make experimental readings using the sensor. The ESP32-S2 includes ADC channels to make analog readings. This input value, the ADC, is a value with 13 bits range, which means is able to read in the 0 - 8191 range. Zero means null voltage while 8191 is the maximum voltage the MCU is able to read. This maximum voltage is dependent on the VRef constant, which depends from chip to chip. Most round 1.1V. It appears it is not possible to change this value, so, if we want to read more than this, attenuation is required. According to the documentation, 11DB (max) allows the chip to read from 0 - 2500mV. This is unfortunate, since the pH sensor outputs 5V (maximum). So, a voltage divider was used to downgrade the analog 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 * (R2 / (R1 + R2))

R1 = R2 = 47kOm

Vout = 2.5V

MAKE VOLTAGE DIVIDER DIAGRAM

After selecting the appropriate residences, the next step was to make the association between the read ADC value and the correspondent pH value. One might be naive and try to convert the ADC value to voltage and then to pH. But this is actually unnecessary. The transformation function between the ADC and the voltage value is something like this:

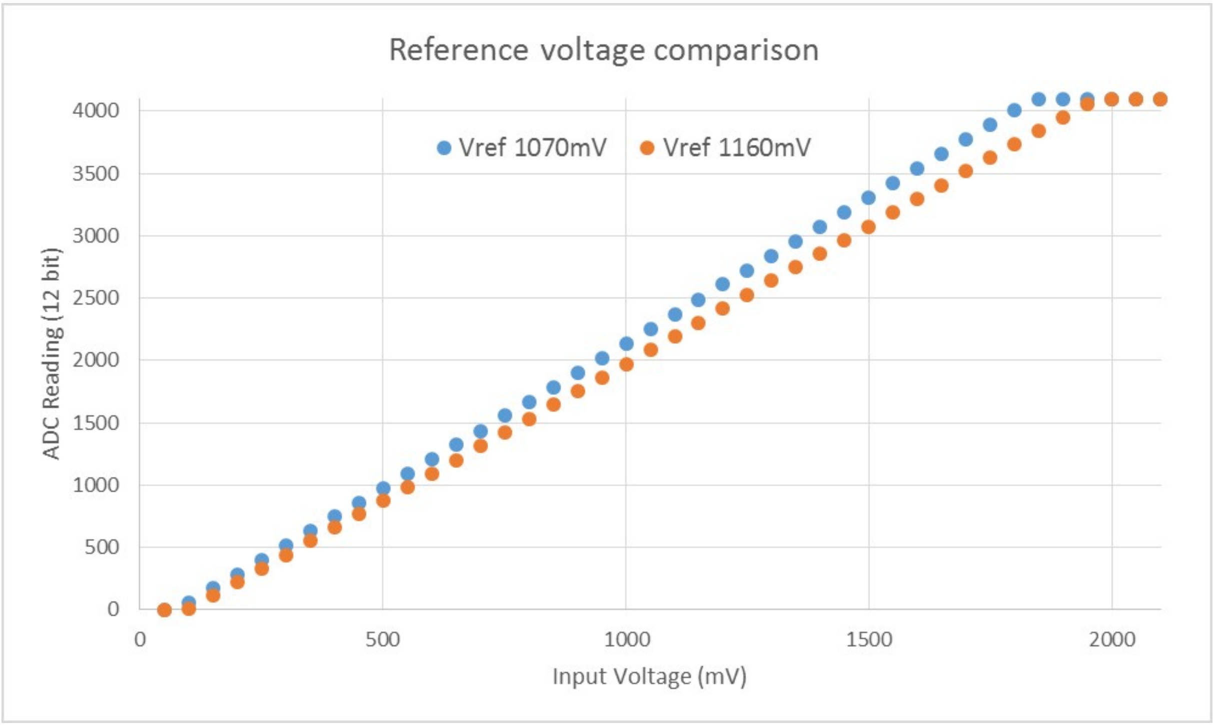

Figure 3.9: ESP Touch - Smart Config Protocol in the ESP8266

meaning, excluding both extremes, the conversion is almost linear. The voltage to pH transformation function is also linear. This means that is possible to convert from ADC to pH with a simple calculation. Knowing this, a solution liquid, measuring pH equal to 4 was used to read the correspondent ADC value in the MCU. The same was done with another solution measuring pH = 8.8. With this, 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 following graph shows:

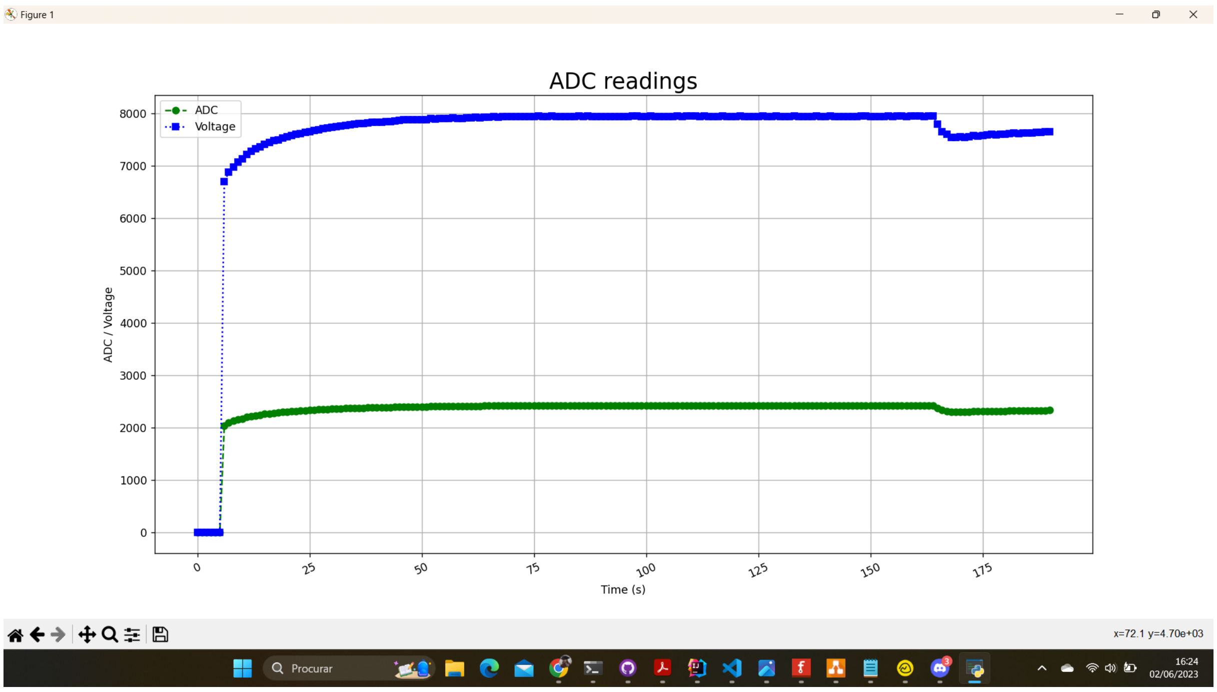

Figure 3.10: ESP Touch - Smart Config Protocol in the ESP8266

The graph above shows the experimentation done to determine the amount of time the sensor takes to stabilize the output with pH equal to 4. It takes approximately 60 seconds to stabilize. Therefore, it was decided that after 60 seconds, the sensor is considered suitable for readings.

Nevertheless, a final experimentation was done, not to measure the correct pH but to estimate how bad *CONTINUE HERE*

To make the ambient humidity and temperature readings, the MCU component that handles this needs to respect the behaviour of the chosen sensor module, described in the Background chapter. This reveals as one of the biggest difficulties while dealing with hardware devices. One might use the Arduino framework which requires maybe two calls to get the readings from the sensor. The framework provides such an abstraction that the programmer doesn't have to deal with synchronization issues, splitting the bits coming from the digital sensor, and other implementation problems that might arise. With the ESP IDF framework, all this small details have to be met to make proper readings. During development, we made a little research to find the implementation mostly done, avoiding to have to implement this low level code.

MAYBE SHOW PLOT WITH RESULTS

As explained in the Background chapter, the "Water Sensor" module will output a certain voltage, each time is in touch with water. At first, we decided to use the ESP32S2 functionality of waking up the chip whenever an input pin the digital 1 value. The water sensor would be used here to wake up the MCU, if water was detected. We encounter two problems:

- After making experimental tests, it was concluded that the module, even after fully immersed in water, it only outputs 1.7V. This is not enough for the MCU to considered a digital one input;

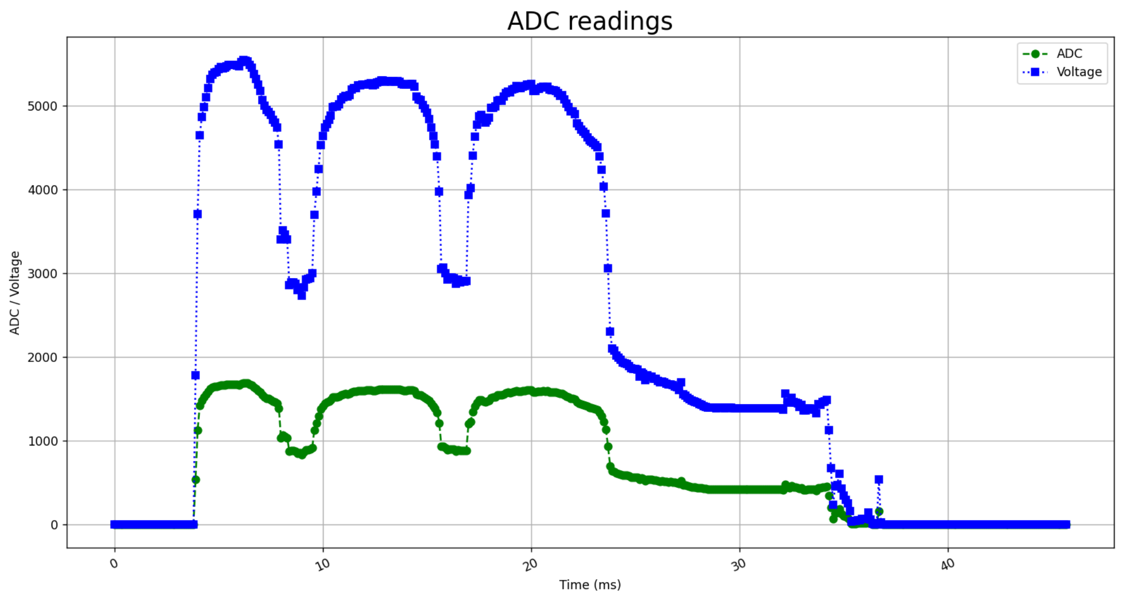

Figure 3.11: ESP Touch - Smart Config Protocol in the ESP8266

- The sensor would have to be constantly on, resulting in a greater energy consumption. This goes against the established requirements, where the hardware is supposed to be economic. Also, it is assumed that whenever there is a water leakage in the system, it's the rate is very slow.

Therefore, its is not worth it to constantly monitor for water leakages. It's more appropriately to check for water leaks, only after the MCU wakes up.

3.3.1.4 WiFi

To connect the "Board" to the WiFi, two properties are required: SSID and Password. The MCU tries first to recover the WiFi configuration and, if available tries to connect to the WiFi, using the WiFi module. Otherwise, the MCU will initiate the ESP Touch protocol to allow users to utilize the ESP Touch Android app to transmit the WiFi configuration to the MCU. Once the User sends the credential to the ESP32-S2, he will store them, in the persistent memory, for later to reconnect to the same SSID. Through the Touch Android application, the user/operator has also the ability to send a costumn string, as described in the Background section, which in our project we define this string as the unique Device ID.

To implement this two modules - ESP Touch Protocol and WiFi - we resorted to the Bernardo previous work, where he also implemented this modules. We took the WiFi and ESP Touch protocol modules and adapted to our specific use case.

3.3.1.5 MQTT

In order 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 our domain. There is many possible MQTT data formats the system can use. We are describing a system, in particular, an IoT system, in which the devices might be in a unreliable network environment. Also, this devices need to be energy efficient. A compact body format may be the difference between an efficient system. Yet, we choose to use the JSON format because the payload will always be small. It only 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 deserialization 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 it self. 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 using the IoT MCU device, having the same behaviour as the WiFi credentials.

3.3.1.6 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. 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 WiFi credentials.

3.3.1.7 Time

Each time a sensor reading is performed, it is necessary to measure the time. There is 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. Yet, we chose to use NTP (Network Time Protocol) to obtain the correct timestamp. This has disadvantages since it requires internet connection to make the synchronization. The advantage is that the time reading is much more accurate. This trade off is sometimes harder to solve. Yet, even tho time accuracy is not very important for this system, every time the MCU wakes up, it sends a log to the Backend Server, thus requiring internet connection. Since the WiFi connection already introduces some overhead, the additional request NTP request wont make much difference in terms of efficiency. Plus the NTP uses a combination of algorithms and protocols to minimize network traffic making him much more efficient then a traditional HTTP request.

3.3.2 ESP Touch App

To configure the MCU, in particular the ESP32-S2,

3.4 Backend

3.4.1 Application Server

3.4.1.1 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. Instead of directly instantiating and managing dependencies, the programmer simply needs to specify the required service or dependency within the layer. Spring takes care of the rest by automatically managing the instantiation and injection of the dependencies. Being said, this system takes advantage of this feature. For example, the server repository classes need a Handle object. Yet, which represents a connection to the database system, and therefore, is tied to the database. When testing the application (ex: integration tests), a different database should be provided to the Repository class. This is were it comes the Spring configuration. In this project, different configurations (with different DBs) are defined, whether the App will run for production or for testing purposes.

3.4.1.2 Cors

TODO

3.4.1.3 Spring Security and Google Authentication

TODO

3.4.1.4 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.

In the present 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 difference pretense. It uses it's ability to inspect the request and metadata to make, not only the Authentication, but also the Authorization process. By doing this, if the handler asks for a User argument, the interceptor, having access to the Authorization field, in the HTTP request, tries to authenticate the user, and resolve the argument. If not possible, means that the argument cannot be resolved, and thus, the user cannot be authenticated. In this scenario, when the interceptor determines that the request should be terminated and the user is unauthorized, it responds 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 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.

3.4.1.5 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 the *@RestController* 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 classes/components, described in the architecture.

As an example, the 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, providing direct access to user information.

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

Inside the controllers, a *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 requires the user to have a specific Role: User or Admin.

3.4.1.6 Services

Each Service module is implemented as a class in Kotlin, that depends on another set of services/modules. These are specified in the Service class constructor. The Spring will then take care of injecting the necessary dependencies upon initialization.

The very nature of the Services is to provide communication between the Controller and Repository layers, enforcing business logic. Some operations require multiple accesses to the database, and some more than one database. Yet, atomicity (a property of database transactions which are guaranteed to either completely occur, or have no effects) must be guaranteed to maintain consistent data over the system. To address this issue, most 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.

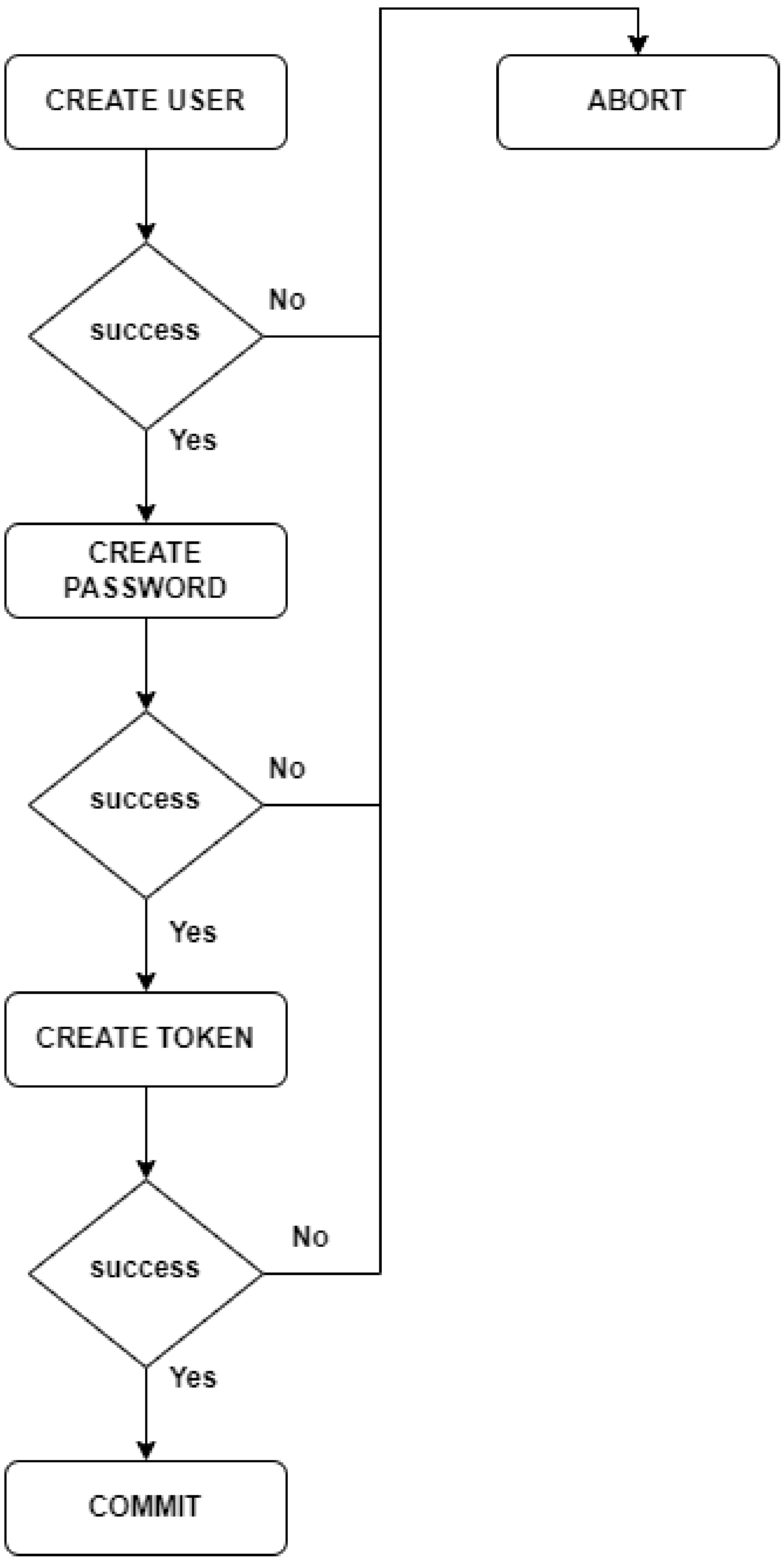

Figure 3.12: Creating a User in the Services

The diagram above shows the process of creating a *User*. Several actions need to be performed. If one fails, the actions are rollback.

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 said above, the *Service* layer is to be used by the *Controller* layer. Several challenges can arise during the execution of an operation, leading to inconveniences or complications. When an operation fails, it is helpful to inform the cause. Doing this, the Controller classes can adapt the type of response based on the 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))
}
```

In the snipped above, the function must first check if the device itself exists, and if so, must also check if it belongs to the entity making the request. Two points of failure can happen.

So, a strategy was developed to accomplish this: when there is more than one point of failure, the functions are going to return an *Either* type:

```
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>()
}
```

This is used to represent an outcome. The *Either.Right* represents a successful operation. On the other hand, the *Either.Left* represents a failed operation.

3.4.1.7 Repository

The repository classes constitute modules that interact directly with the Databases.

This report does not provide an in-depth analysis or detailed explanation of the implementation of these classes. For a comprehensive understanding of the implementation, it is recommended that users refer to the official support documentation, which provides detailed information on building PostgreSQL queries and utilizing the mentioned classes.

One important aspect is that the reader might wonder is why these two classes are not annotated with the *@Repository* annotation. This is because these classes don't need to be scanned by the Spring Context. They are created in the *JdbiTransaction* class and in the *JdbiTransactionManager*, being the last one annotated with the *@Component* annotation, allowing the *Services* to access the functions to interact with the Database.

The other repository class is the *SensorDataRepo*. This implements the access layer to the InfluxDB database.

```
@Repository
class SensorDataRepo(
  private val client: InfluxDBClientKotlin,
  bucket : Bucket
) : CollectedDataRepository {
  private val bucketName = bucket.name
  val mutex = Mutex() // Use Mutex for synchronization
  private val MEASUREMENT_PREFIX = "my_sensor" // Modify this line
  ...
```

Unlike the classes that interacted with the PostgresDB, this one is annotated with the *@Repository* annotation, so Spring can scan this as a Repository class.

The implementation details of how this class interacts with the database are not mentioned here, as they do not contribute significantly to the understanding of the chosen solution. Instead, it is recommended that the reader refer to the official documentation available on the InfluxDB website. The official documentation provides comprehensive information and examples that can help users understand how to effectively interact with the InfluxDB database using this class.

3.4.1.8 Sensor Analysis

When this application receives the sensor data, it needs to analyse it. Well, this means evaluating if the received sensor records indicate dangerous values in the filter. There are many possibilities to achieve this. First, the server needs to know each sensor thresholds values (boundaries). For example, the pH lower and upper boundaries might be 6 and 7. This system assumes that all filters belong to the same company, and thus, they should function in the same conditions. Since the application server is shared by all the users, and therefore, all devices, the server only needs to know the values for each sensor. One possibility would be for each device sending, along with each value record, the sensor thresholds. The server would receive the value and the respective thresholds and process it immediately. The disadvantages. The main one 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. Yet, as said above, this is not a requirement. So, the current solution is to load ... *CONTINUE HERE*

*TODO: falar o facto do servidor usar sensores genericos

*TODO: falar sobre a geraçao de Device Ids

TODO: falar sobre os alertas por email

TODO: falar dos topic subscribe

TODO: falar de error handling

3.5 Website

3.5.1 Approach

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 components. With an SPA, the Backend does not need to handle page rendering or selection for the Frontend applications. For example, it doenst 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. By adopting SPA, 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

To enable Deep Linking functionality, we leveraged the capabilities of the Webpack bundler, which provided a straightforward solution. Deep Linking allows users to access specific pages or sections of our Single Page Application (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. 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 that doesn't exist. Therefore, 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.

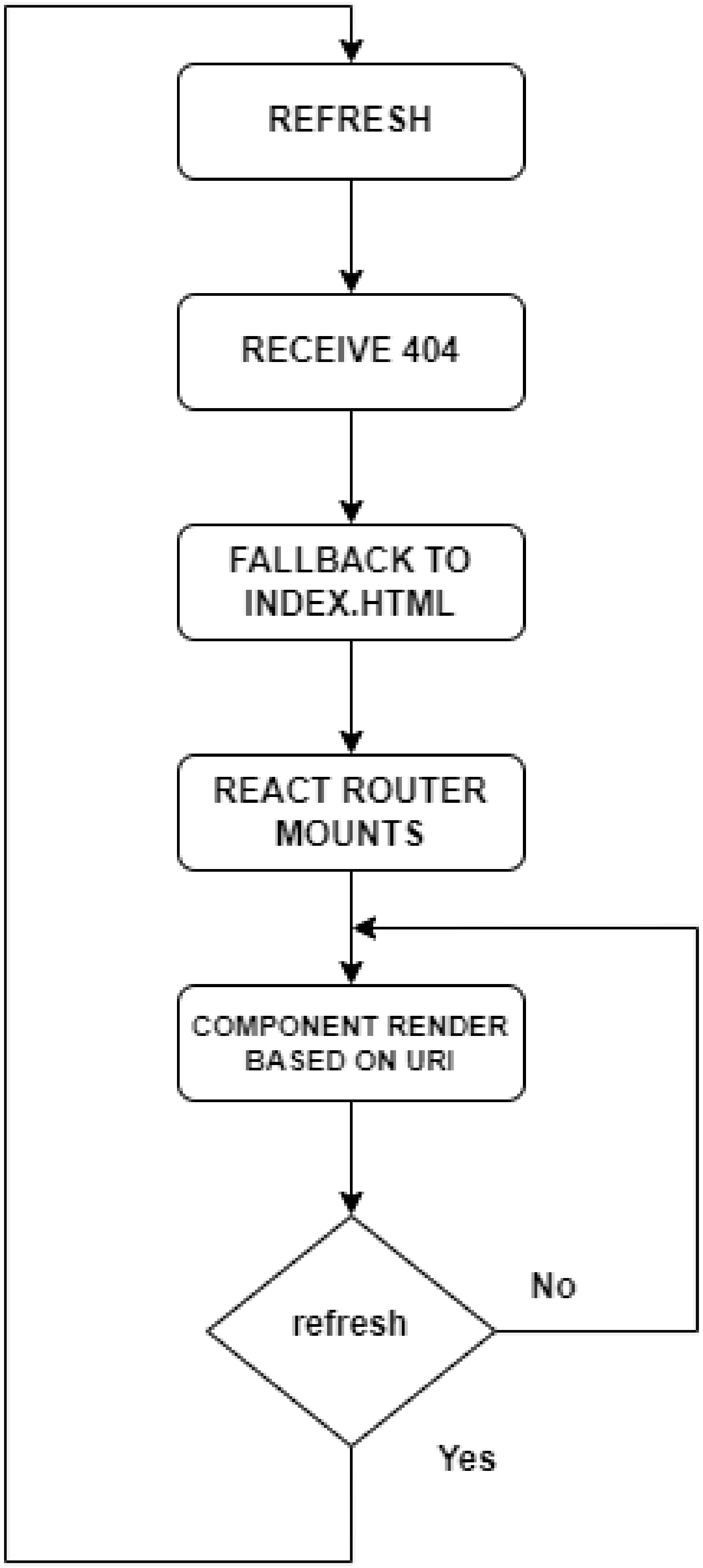

Figure 3.13: Webpack with Deep Linking

3.5.2.1 Main Web Site Behaviour

When the React App component is loaded, the first thing to do is to load all the Siren API information, from the Backend. This is used to facilitate the Frontend development, since it is useful for the website to have to know only one Server URL - the base URL. From this endpoint, the Website should be able to get all the API information, in order to navigate all the API endpoints. If this information cannot be obtained, the website will only show a static error message. This is because it doesn't make sense to allow any operation since the website is blind, not knowing how to interact with the remote API. All functionalities, from creating Users to requesting device information, are all dependent from the API. 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).

3.5.2.2 Protected Components

Most website components are protected, meaning no ordinary/standard user should be able to access/render them, but only authorized ones. Because of this, the website provides a sign-up/sign-in page. After users login, they will be redirected to an appropriate page. Yet, nothing forbids the user from trying to access a protected resource by using deep-linking (inserting the page URL manually in the browser). Therefore, there is a special component that wraps the protected component, ensuring that the user is logged-in: <RequireAuthn />

His behaviour is pretty simple:

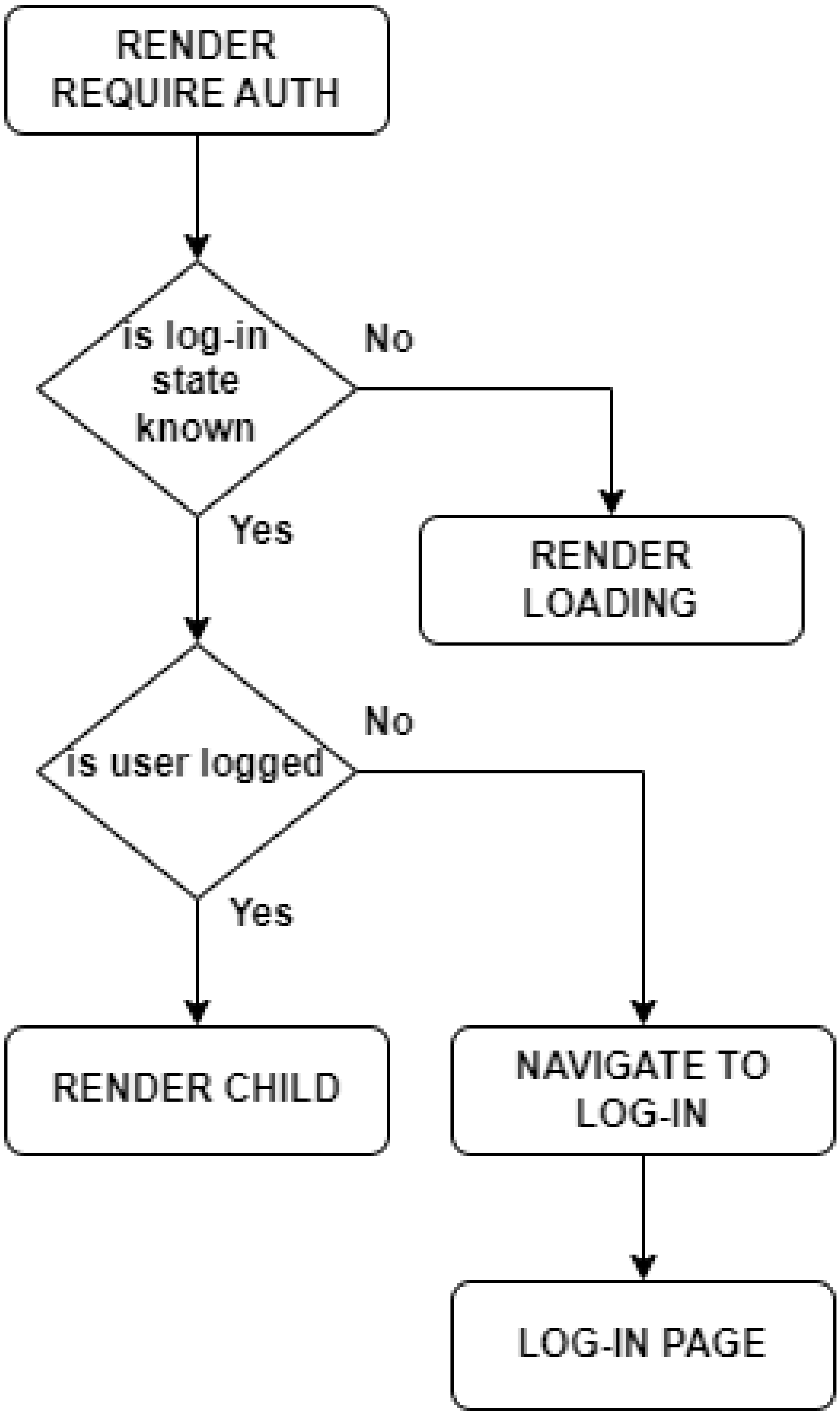

Figure 3.14: Webpack with Deep Linking

By using this made up component, it ensures protected components are only seen by authenticated users, even with URL direct accesses.

The login state is managed by the *AuthnContainer* component, loaded by the *Router* component, so that its components 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 component might be save the authentication state, once the user logs-in. But what happens if the user refreshes the page, and the auth state is lost?" Should the user have to log-in again? One possibility, which was described above, is to hold the state in cache. Like it was said, this is not bullet proof, since the state might become outdated. Well, in our system we use a token inside a cookie to hold the login state. Yet, 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. After all, the authentication is done against the server. The source of truth lies in the token validity!. Even so, there is another problem. The cookie sent by the server, cannot be accessed in Javascript. This is to protect from cross-site scripting (XSS) attacks. So, knowing all this, the browser needs to make an API request to know if the Client is authenticated or not. Similarly, to make the logout, the Client browser needs to make an API request so the Server can remove the cookie/token, therefore eliminating the User session.

3.5.2.3 Error Handling

Most unpredictable events will come from the Backend Server (ex: when the user requests its own devices, sensor data, etc). When a component uses the *Services* to fulfil a request, the call will return a result, which could be positive or an error if something goes wrong. In case is the latter, a string is available, indicating the error in detail. The component *ErrorContainer* behaves similar to the *AuthContainer*. If a component receive an error from the services, it sets a property which re-renders the ErrorContainer, re-render the page with an error.

TALVEZ FAZER ASM

3.5.2.4 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 delve into the details 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. This 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 the use a python script that emulates the hardware device to send synthetic sensor data to the Backend application server, so is possible to test the server receiving sensor data, without relying on the real hardware.

4 Conclusion