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 Final Project
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.

Índice

Lista de Figuras

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 analyze 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 analyze 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 analog 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.

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 decision

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 for this board were the MSP430FR413x, ESP32-S2, and ESP32-S3 boards, there are several factors to consider. While each board has its own strengths, the ESP32-S2 stands out as a favorable choice for 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.

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. Additionally, the ESP32-S2 includes integrated Wi-Fi functionality, allowing seamless communication and integration with IoT systems and networks.

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.

2.1.2.2 Framework

We made the decision to utilize the ESP-IDF(Espressif IoT Development Framework) framework instead of the more commonly used and less versatile Arduino framework for programming the board. This choice was motivated by the project's requirement for an extended battery life. The ESP-IDF framework provides us with enhanced control over the device, enabling us to leverage various sleep modes that effectively preserve the board's battery when it is not actively collecting or transmitting data.

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.

2.1.3.3 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. After careful consideration, we opted for the MQTT protocol. 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. 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 analyzing the data. For the technology chosen 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 Frontend

In our project, we have developed a user-friendly front-end 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.

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.

Projeto do bernardo

3

Proposed IoT System Architecture

3.1 Introduction

In this chapter, the proposed system architecture will be presented. First, the system high level view will be presented. This includes defining each module/segment and how they all interact with each other, composing the final solution. Then, for each component presented, the details of it's implementation will be addressed, along with the chosen technologies. 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 analyze relevant filter-collected data, and manage devices

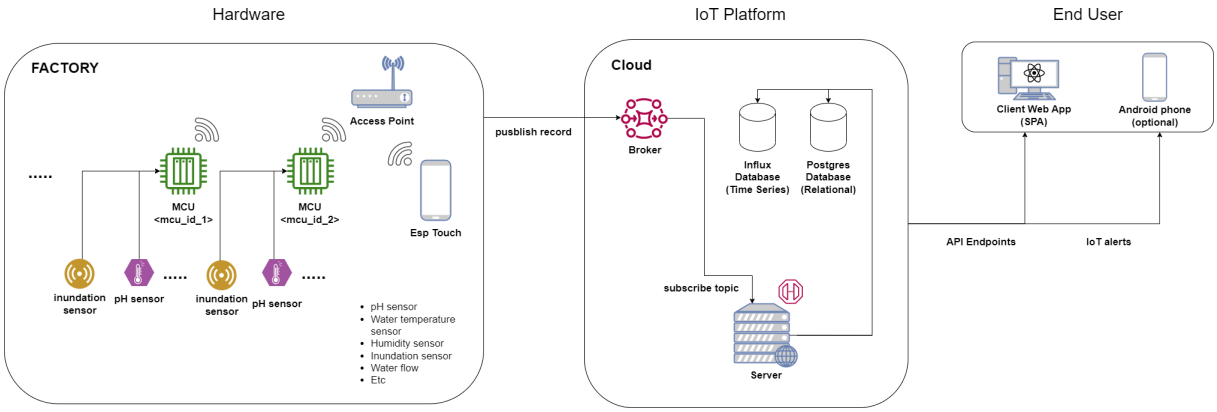

Figura 3.1: Main system architecture

Overall, data flows from left to right. All begins in the MCU collecting data (sensor records) and then, sending the data to the Backend Broker module. The Backend will, eventually, receive the data and analyze it accordingly. It also exposes a Web API, where Frontend applications, like the one described here - web application - will use to interact with the system, for instance to query for available data, adding devices, etc.

3.2.1 Factory

This component involves 4 sub-components:

- Sensors: collect neutralization filter data
- Microcontroller (MCU): coordinate when to collect and send the data to the Backend
- Access Point (AP): allows the factory to connect with the internet
- Small MCU App: App to configure the MCU

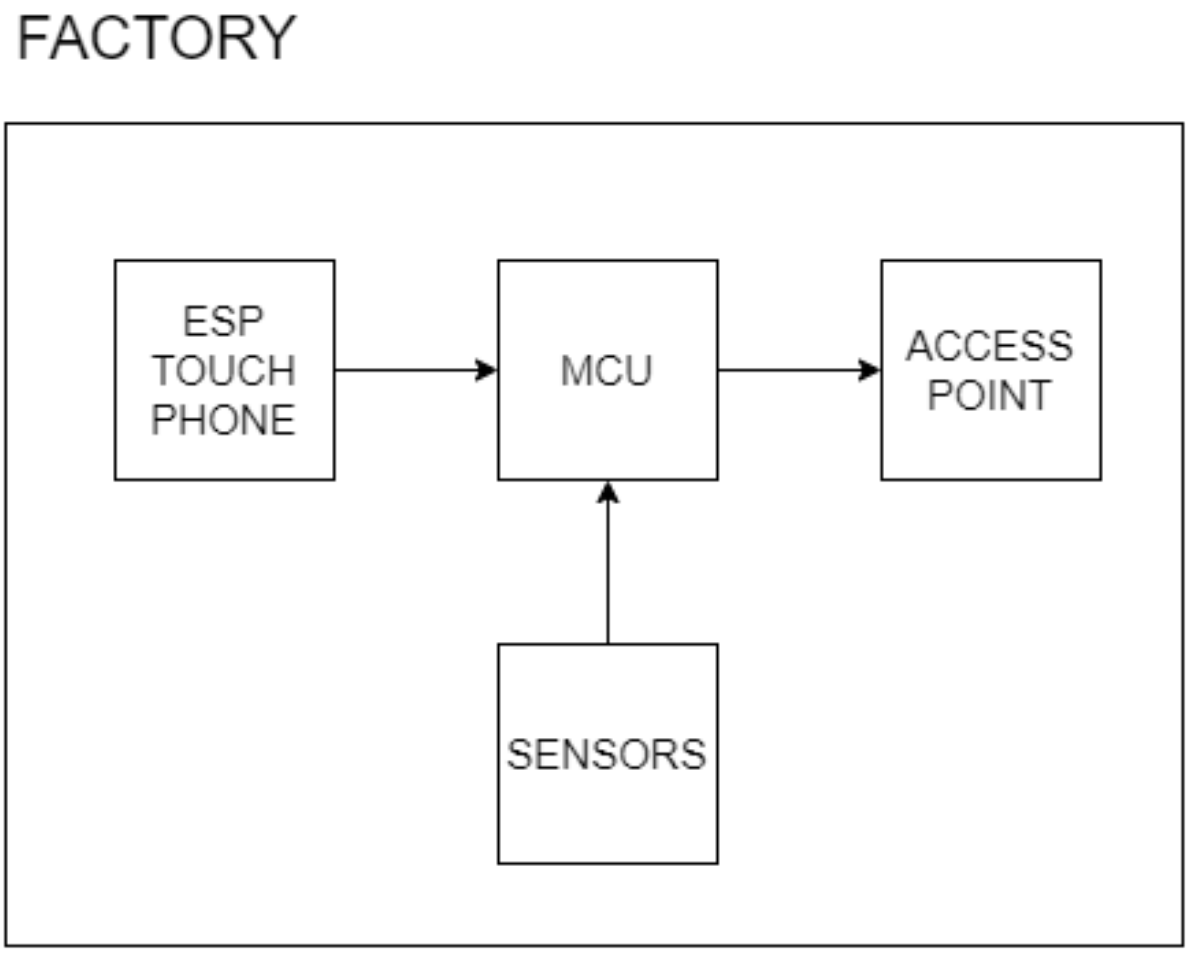

Figura 3.2: Main system architecture

The user will interact with a small Android app to configure the MCU, for the first time. This includes setting up a device ID and sending the WiFi network credentials directly to the MCU. After normal initialization, the MCU will use the sensors to collect data, and send it to the Backend Broker, with the use of the Access Point.

3.2.1.1 MCU

The component architecture is the following one:

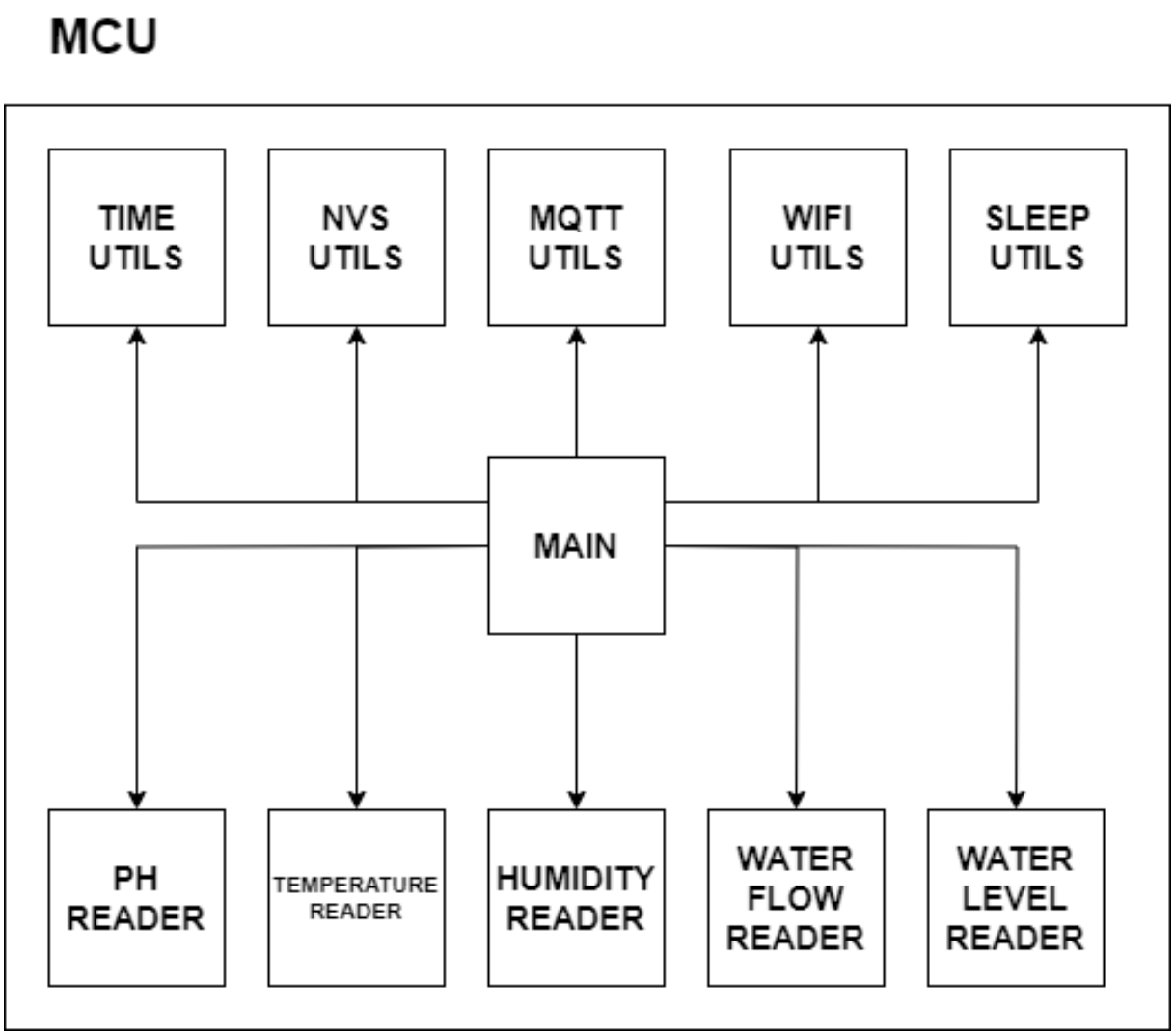

Figura 3.3: MCU firmware architecture

The Main component controls the main flux algorithm. It interacts with other utility components, like the sensor readers to collect records, from the sensors, the MQTT to send the records using the MQTT protocol, etc.

3.2.1.2 Sensors

There are a total of seven sensors:

- Start pH sensor
- End pH sensor
- Air temperature sensor

- Humidity sensor
- Water level sensor
- Water flow sensor
- Water leak sensor

The main objective, with this project, is to develop a system that automates the neutralization filter. All this sensors give us the data necessary accomplish that. Although, it is also interesting to develop a system that is able to give us meaningful insights about the use of the filter, like the amount of water that passes, or the neutralization effectiveness, giving the initial and final pH.

3.2.2 Backend

This component is composed of the fallowing sub-components:

- Server
- Broker
- Database

All the sensorial data, coming from the factory, will be sent to the Broker. The Server, will receive that same data, from the Broker, and persist it, using two databases.

3.2.2.1 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.

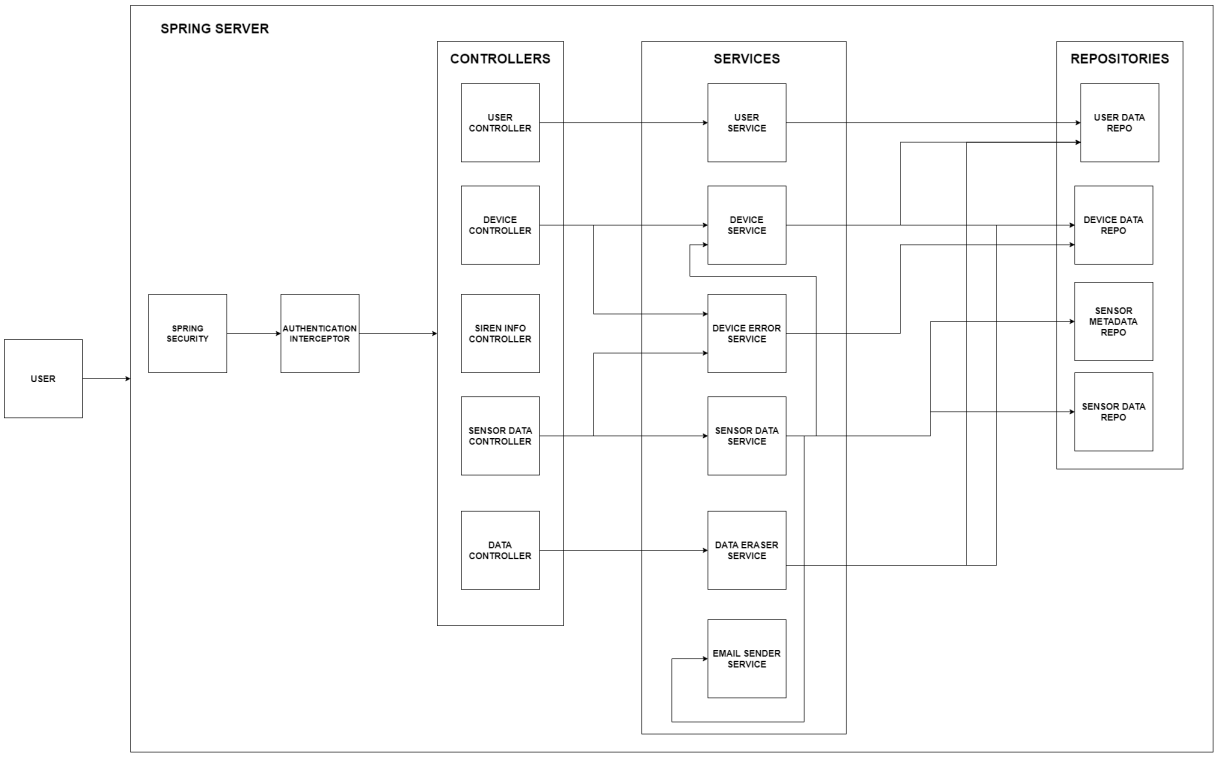

Figura 3.4: Spring Server Architecture

The figure above shows the main modules involved.

- Authentication
- Other Interceptors
- The controller: handles the http API,
- The services: control each possible Server operation
- The repositories: handle data persistence.

Authentication and authorization will be used when the request is trying to access protected resources. The interceptors will then be called to process the request, as for checking cookies and tokens or to log the request. Each time a client makes a request, the controllers will use the services to full-fill the request. Similarly, each time the Services need to access data, they will use the repositories to fetch/store necessary data. In other words, the request will flow from left to right, and the response will flow in the opposite way.

The server is organized into the fallowing components:

- User: deals with all operations related to the Users, ex: creating, requesting, deleting users;
- Device: deals with all operations related to the devices, ex: creating, requesting, deleting devices;
- Sensor Data: deals with all operations related to the Sensor Data, ex: requesting available sensor types, for each device and questing sensor data;
- API Info: used to retrieved Server API information

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 o the Server.

3.2.2.3 Database

Duo to the nature of this project, there is two types of data to be persisted:

- Aggregate data: Users, Devices, etc;
- Time based data: sensor data

A relational DB was used for the first kind. This type of data needs a strong aggregation. Users have Devices, Devices have Logs, etc. A relational DB is specially designed to enforce structure between data. Another reason is when includes sensible data, in this case User information, which should be carefully manipulated. PostgreSQL was chosen duo to being an open source, powerful and high reliable database.

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.

3.2.2.4 Data Model

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 squeme:

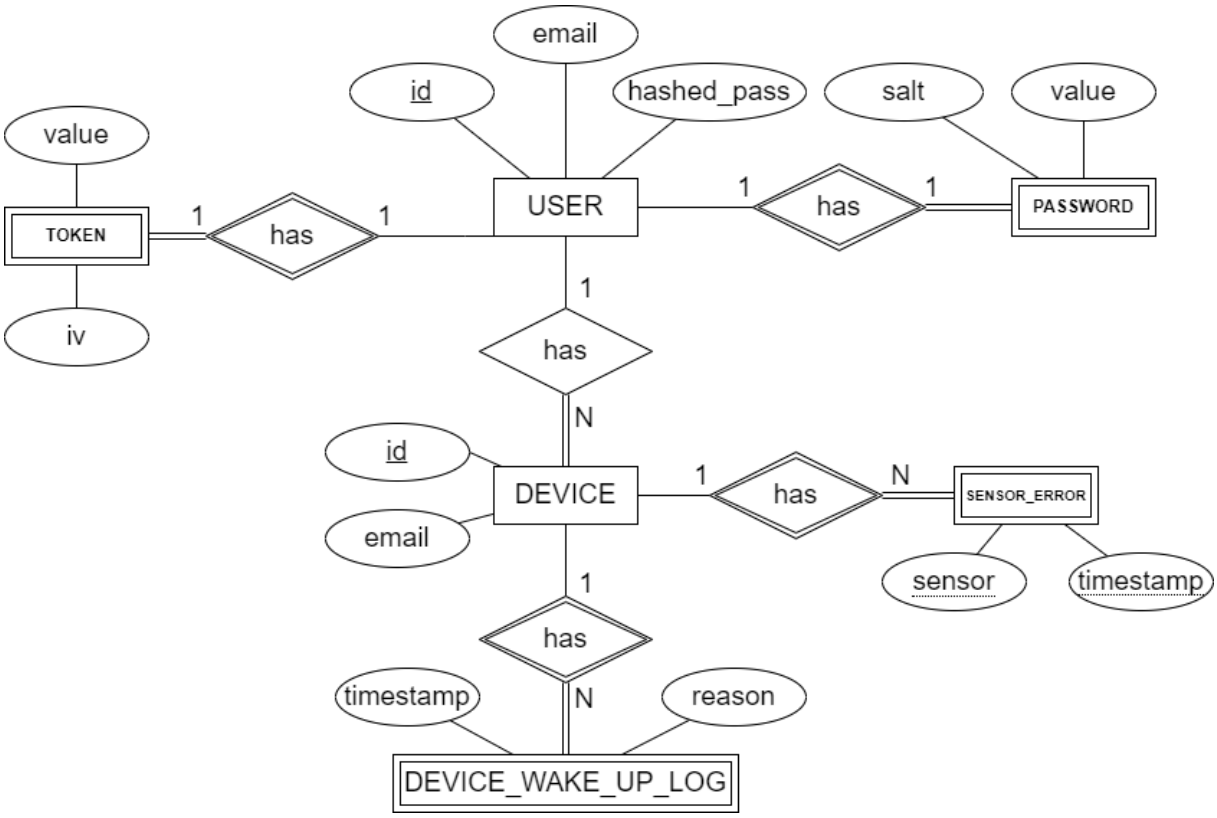

Figura 3.5: Postgres DB Entity Model

The User can have multiple many 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.

Since the InfluxDB is not a relational DB, we couldn't find a standard way to represent the data. Yet, the following diagram provides a good insight of the kind of data that it stores:

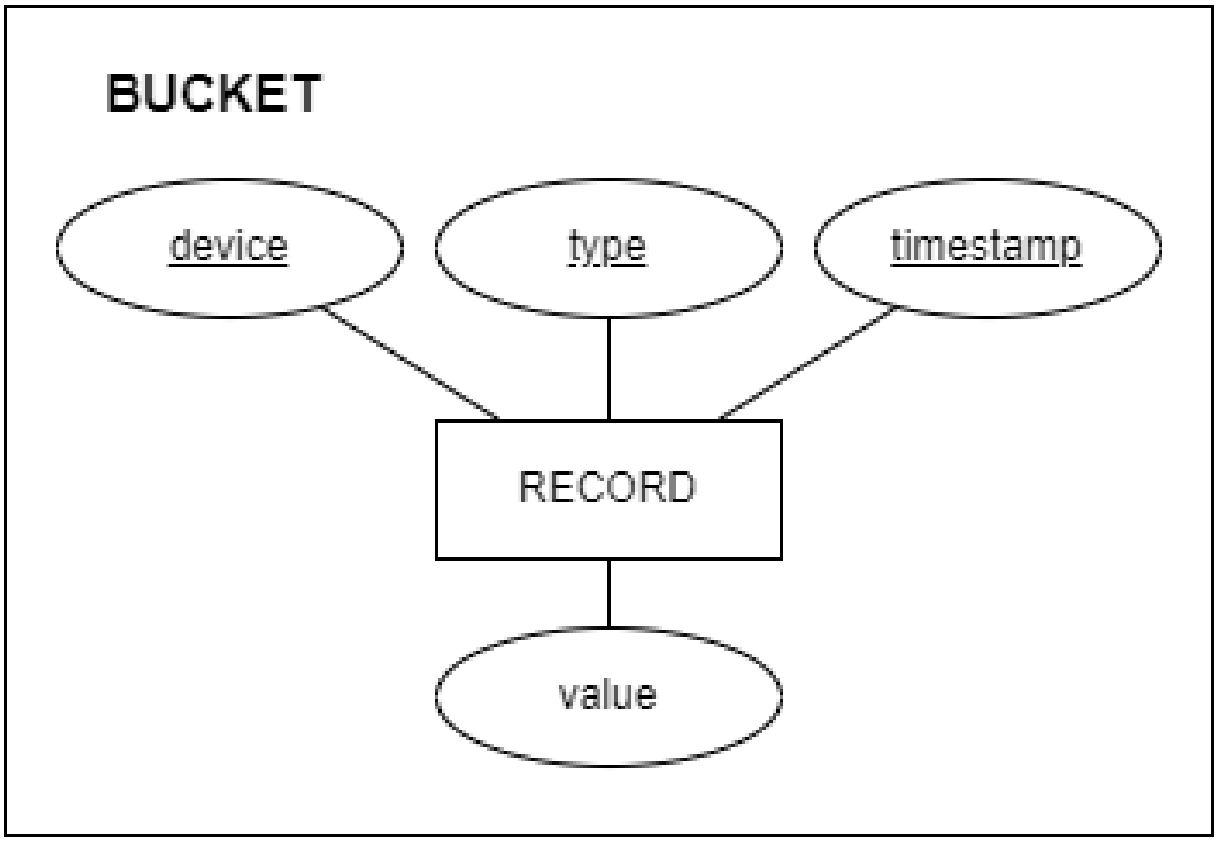

Figura 3.6: Postgres DB Entity Model

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.

3.3 Factory

3.3.1 MCU

3.3.1.1 Device Selection

The two main requirements for this project, involving the equipment, was the low cost of the IoT devices and it's power consumption.

One important aspect is that, since all the sensor data is to be sent to the Backend, which is not in the same physical place as the neutralization filter, and consequently the MCU, WiFi was also required to send the data, over the internet.

Given all this aspects, cost, connectivity and power consumption, the ESP series appear to have the best cost-benefit. In particular, the ESP32-S2 was the chosen one.

3.3.1.2 Programming Framework

Each MCU needs to be programmed to operate. The chosen framework is the ESP IDF, since is the official ESP series framework, which give us more control of the hardware, and is backed by a strong documentation support. On the other hand, the C and C++ are the languages available, which are not known as flexible languages. This increases the complexity of the final firmware program, comparing with other more general solutions, like the Arduino or PlatformIO.

3.3.1.3 Behaviour

The MCU behaviour can be summarized by the state ASM chart:

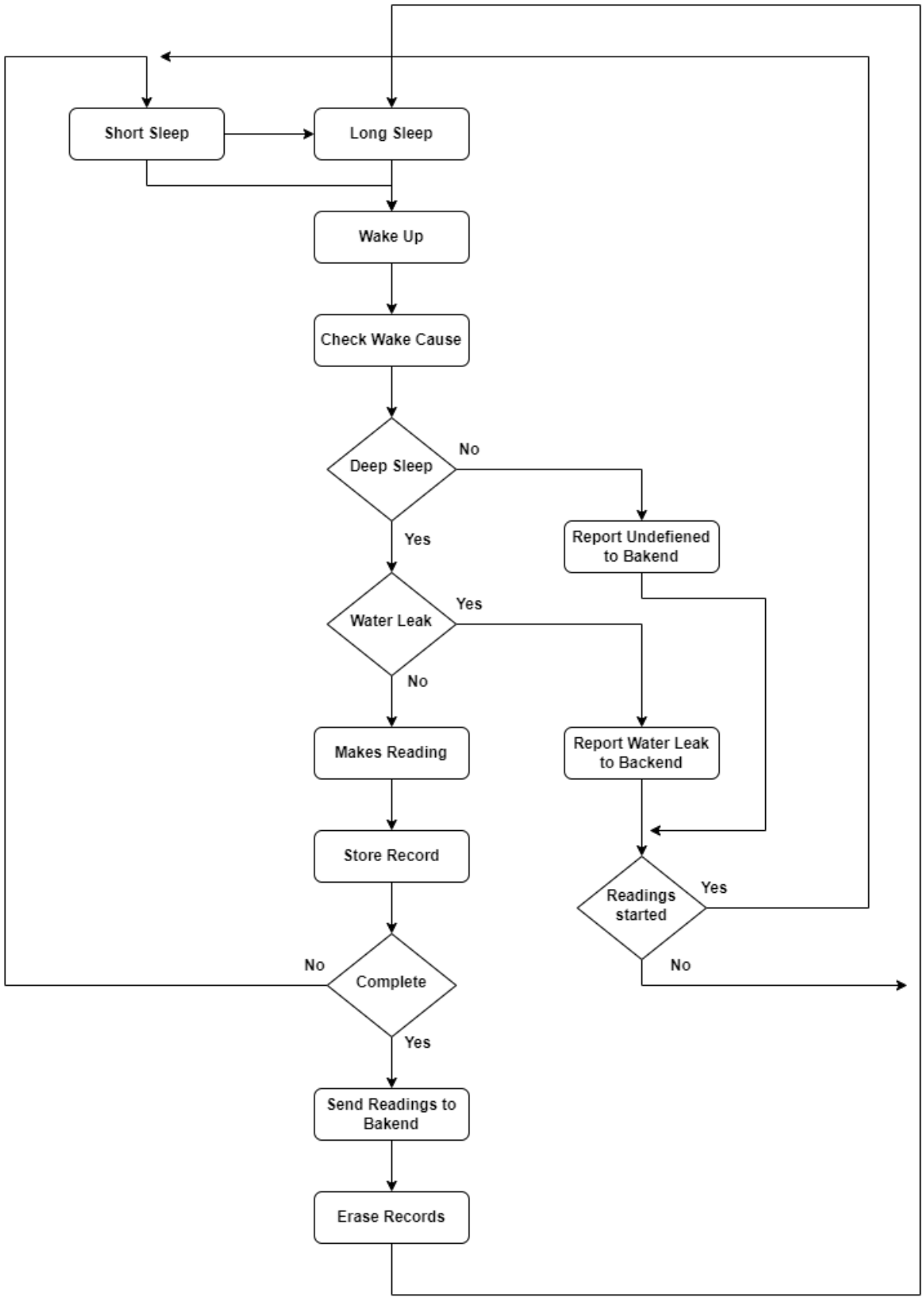

Figura 3.7: ESP Touch - Smart Config Protocol in the ESP8266

For the most time, the MCU will be sleeping. This is because it is not worth it to make constant readings. The filter conditions doesn't change every minute. Besides, since the power consumption is a concert, each second the MCU is running counts!

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

Usually, the last one happens when the device aborted in the 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, using a cable, to the MCU. For this reason, if an error occurs, it is of great importance to report on 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 wake up timer, since it is not an abnormal wake up, it is not necessary to inform the Backend.

One possibility was to make the reading and report it immediately to the Backend. Sometimes, in the neutralization system, the readings can oscillate a bit. That is why we choose to make some readings, intervalated 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.

Overall, the MCU will wake up, report the wake up cause to the Backend, if abnormal, make readings, send the readings to the Backend, if appropriately, and go back to sleep again.

All this behaviour takes place in the main MCU module.

3.3.1.4 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.

3.3.1.5 Sensor Modules

There is a total of seven sensors. Most sensors require it's own module that handles the sensor readings, since the code to read from the sensor is specific for that same sensor.

A special struct was constructed to store the sensor readings.

```
#include "ph_values_struct.h"

#ifndef SENSOR_RECORDS_H
#define SENSOR_RECORDS_H

#define MAX_PH_VALUES 1

typedef struct sensors_records_struct {
    struct ph_records_struct ph_readings;
    struct water_flow_records_struct water_flow_readings;
    struct water_level_records_struct water_level_readings;
    struct temp_records_struct temp_readings;
    struct humidity_records_struct humidity_readings;
} sensor_records_struct;

#endif
}
```

```
#ifndef SENSOR_RECORD_H
#define SENSOR_RECORD_H

typedef struct sensor_record {
    char* sensor_name;
```

```
6     float value;
7     int timestamp;
8  } sensor_record;
9
10 #endif
```

Then, each module has a function that should be called to make the sensor reading. For example, for the pH:

```
1  #include "sensor_record.h"
2
3  int read_start_ph_record(sensor_record *ph_record);
4  int read_end_ph_record(sensor_record *ph_record);
```

Fakes During development, one restraint was that the sensors might not be immediately available. To fight go around this, we replaced the "sensor reader"modules with fakes, that simulated the real readings. Even if the sensors were available, since the beginning of development, having fakes is always a good practice, easing the testing of other components.

Overall, for each sensor reader component, there is two implementations. The real one that interacts with the sensor and the fake one that simulates that procedure, allowing not to depend on the sensors.

Temperature and Humidity Between the existent sensors, we opted for the DHT11 sensor module, since is able to read temperature and humidity, is low cost and very easy to deploy with our current hardware.

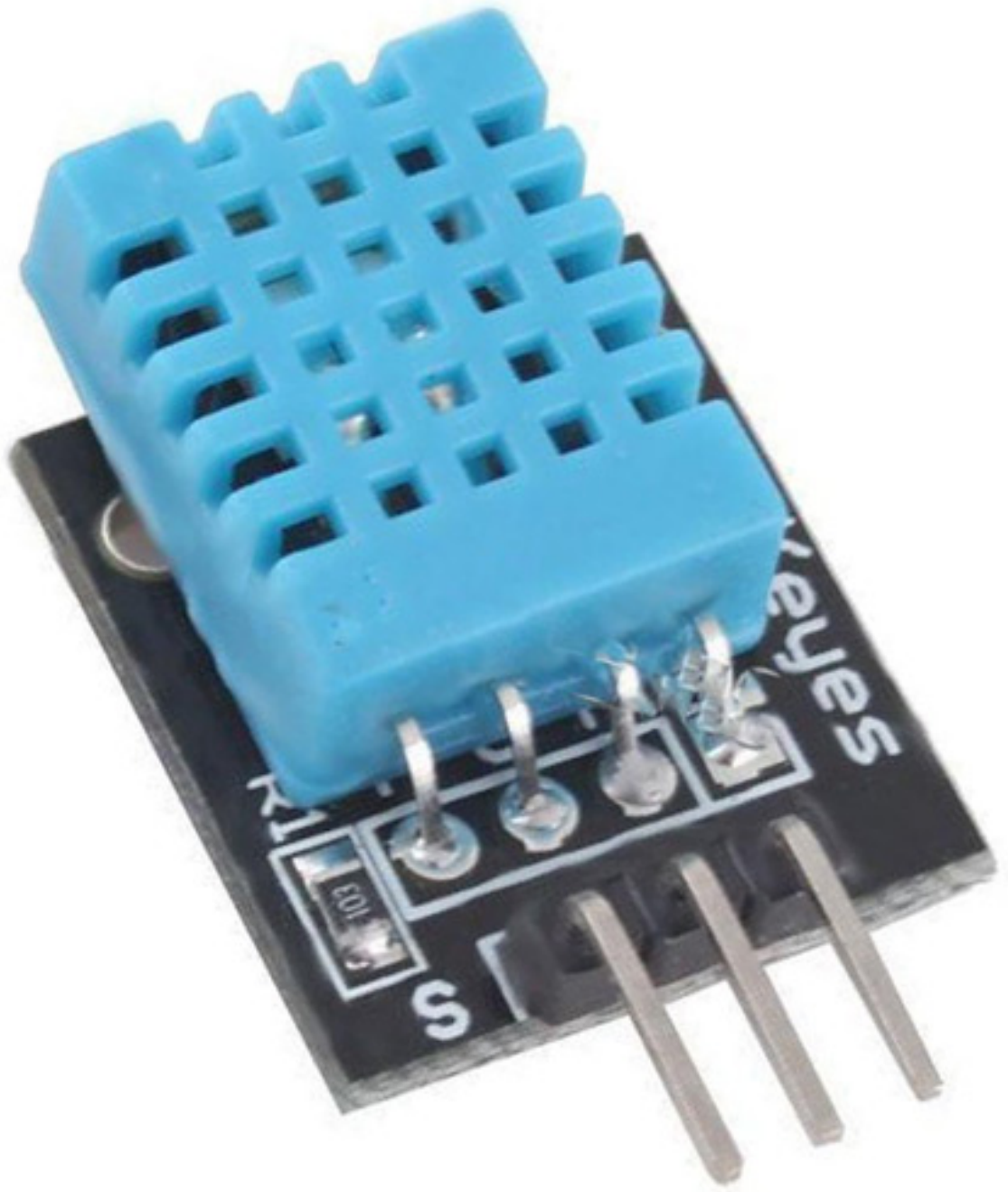

Figura 3.8: DHT11 Sensor Module

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

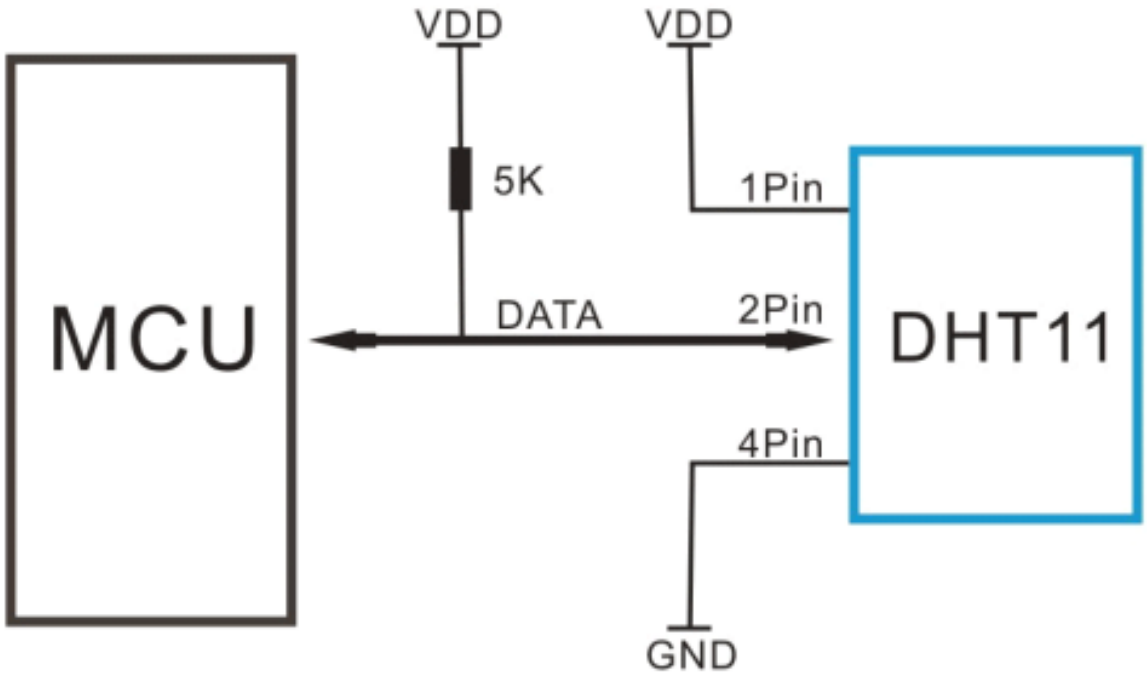

Figura 3.9: DHT11 Sensor Module

As you can see 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

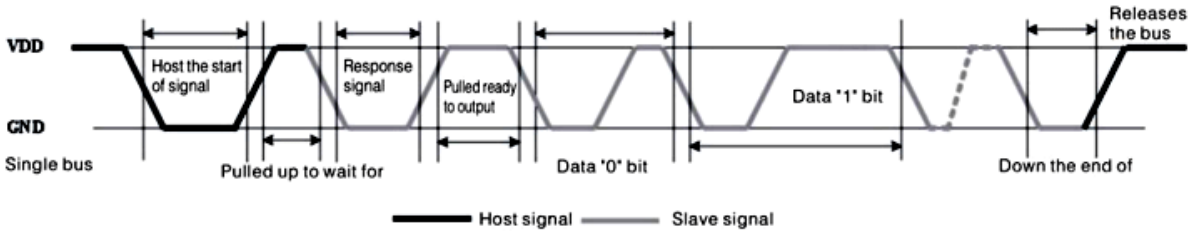

Figura 3.10: DHT11 Sensor Module

The module to read from this sensor is relatively simple. It is composed of two main functions:

```
1 void DHT11_init(gpio_num_t gpio_num);
```

```
2 struct dht11_reading DHT11_read();
```

The first one initiates the GPIO to read from. It also makes the appropriate sensor initialization. The second one performs the reading. This involves sending the appropriate signals, checking for errors and wait for the bits that represent both the temperature and humidity.

Water Leak Sensor <TODO: falar do water leak sensor>

3.3.1.6 WiFi

To connect the "Board"to the WiFi, two information's that are required: chosen access point and password. The MCU will try to recover the WiFi configuration, if available. If existent, the MCU will try 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.

```
1  void setup_wifi(void) {
2      strcpy(action, "seting_up_wifi");
3      ESP_LOGE(TAG, "Setting up WiFi...");
4
5      char* deviceID;
6      wifi_config_t wifiConfig;
7
8      if (get_saved_wifi(&wifiConfig) == ESP_OK && get_device_id(&deviceID) ==
         ESP_OK )
9      {
10         if(!connect_to_wifi(wifiConfig)) esp_touch_helper(&deviceID);
11     } else
12     {
13         esp_touch_helper(&deviceID);
14     }
15
16     ESP_LOGE(TAG, "Finished setting up WiFi");
17 }
```

ESP Touch The ESP Touch protocol is a wireless communication protocol developed by Espressif Systems specifically for their ESP8266 and ESP32 series of WiFi-enabled microcontrollers.

The free ESP Touch Android app uses this protocol to connect to the ESP Device, and thus, transmiting, not only the WiFi configuration, but also a sting of bytes as costum data. In the current system, this string of bytes will be used to configure the device ID.

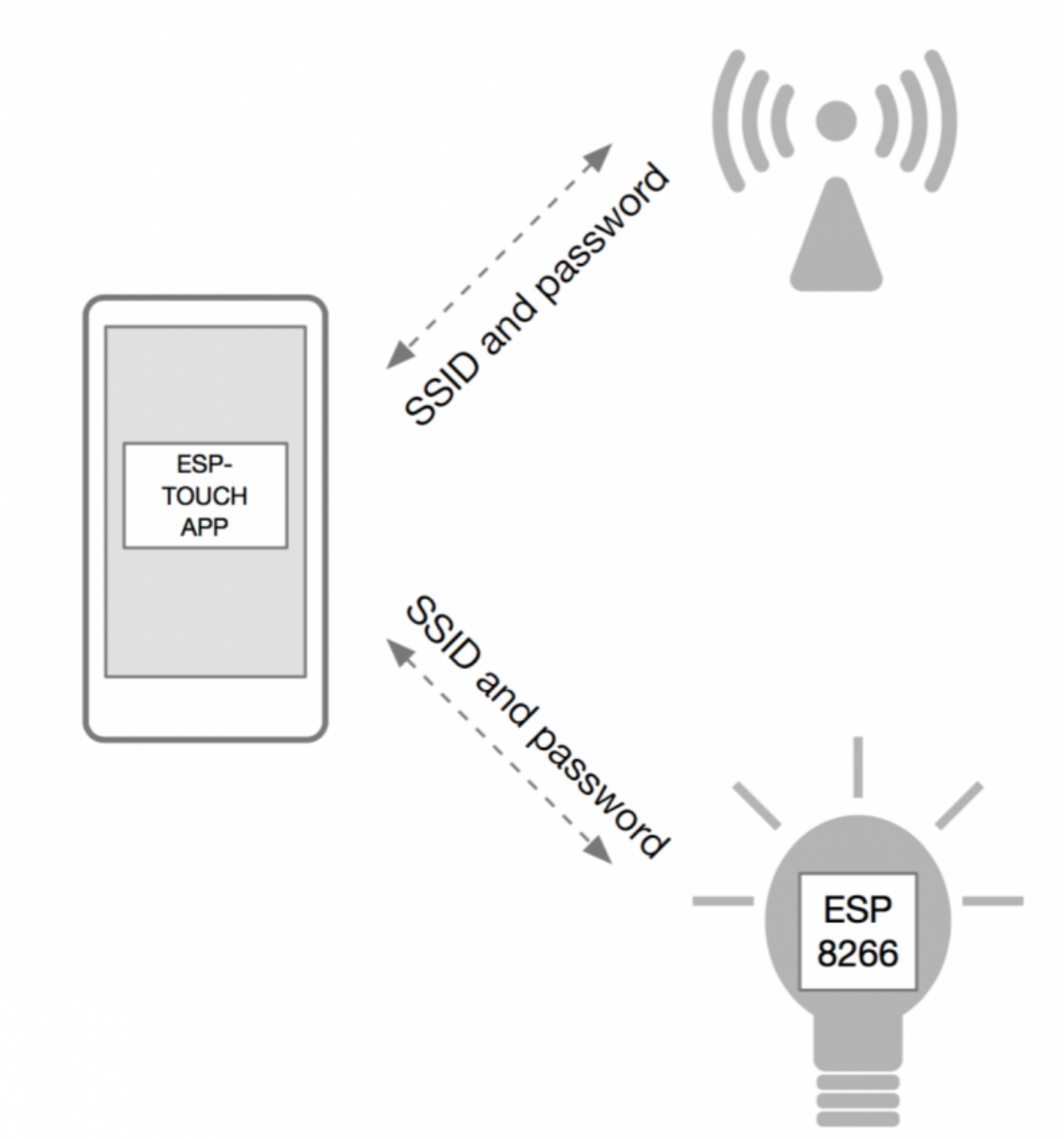

Figura 3.11: ESP Touch - Smart Config Protocol in the ESP8266

Code Itself To implement this two modules - ESP Touch Protocol and WiFI - we resorted to the Bernardo previous work, where he also implemented this modules.

```
1  static void event_handler(void* arg, esp_event_base_t event_base, int32_t event_id,
       void* event_data) {
2      ...
3      memcpy(ssid, evt->ssid, sizeof(evt->ssid));
4      memcpy(password, evt->password, sizeof(evt->password));
5      ESP_LOGI(TAG, "SSID: %s", ssid);
6      ESP_LOGI(TAG, "PASSWORD: %s", password);
7
8      if (evt->type == SC_TYPE_ESPTOUCH_V2) {
9          ESP_ERROR_CHECK( esp_smartconfig_get_rvd_data(rvd_data, sizeof(
       rvd_data)) );
10         ESP_LOGI(TAG, "RVD_DATA: %s", (char*)rvd_data);
11         *device_id = malloc(strlen((char*)rvd_data));
12         strcpy(*device_id, (char*)rvd_data);
13     }
14
15     set_saved_wifi(&wifi_config);
16     set_device_id((char*)rvd_data);
17
18     ESP_ERROR_CHECK( esp_wifi_disconnect() );
19     ESP_ERROR_CHECK( esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
20     esp_wifi_connect();
21     ...
22 }
```

From the listing above, we can see thta the function "eventHandler"will receive the ssid and password WiFi credentials. Not only, the handler will access the custom data in line 10, defining it as the device ID. Finally, both configurations will be stored internally.

3.3.1.7 MQTT

In order to send the sensor records to the Backend Broker, it was developed the module mqtt_util, with the fallowing functions:

```
esp_mqtt_client_handle_t setup_mqtt();

void mqtt_send_sensor_record(esp_mqtt_client_handle_t client, struct sensor_record1
    *sensor_record, char* deviceID, char* topic);

void mqtt_send_device_wake_up_reason_alert(esp_mqtt_client_handle_t client, int
    timestamp, char* deviceID, char* wake_up_reason);
}
```

To implement this module, we resorted to the ESP IDF tutorials that already provide the code that handles the MQTT protocol. A Broker URL is necessary.

```
...
static const char *CONFIG_BROKER_URL = "mqtt://2.tcp.eu.ngrok.io
    :15281/";
...
}
```

This is the URL to reach the Broker.

<TODO: falar do formato das mensagens MQTT>

3.3.1.8 Storage

To persist data in the ESP32-S2, the Non-volatile Storage Library was used in this project. Non-volatile storage (NVS) library is designed to store key-value pairs in flash. This way, even when the device is powered down, data wont be lost.

In the MCU firmware, there is a module called nvs_utils to interact with the NVS partition, having functions to store and retrieve the device ID and WiFi credentials.

3.3.1.9 Time

Each time a sensor reading is performed, it is necessary to request the time. To obtain the current time, it is necessary to check if the time is valid. If so, the time is obtained. Otherwise, it is necessary to connect to the WiFi to synchronize the time.

```
1  void obtain_time(void)
2  {
3
4  #ifdef LWIP_DHCP_GET_NTP_SRV
5      sntp_servermode_dhcp(1);      // accept NTP offers from DHCP server, if any
6  #endif
7
8      /* This helper function configures Wi-Fi or Ethernet, as selected in menuconfig.
9       * Read "Establishing Wi-Fi or Ethernet Connection" section in
10      * examples/protocols/README.md for more information about this function.
11      */
12
13     if (isTimeValid()) return;
14
15     initialize_sntp();
16
17     // wait for time to be set
18     time_t now = 0;
19     struct tm timeinfo = { 0 };
20     int retry = 0;
21     const int retry_count = 15;
22     while (sntp_get_sync_status() == SNTP_SYNC_STATUS_RESET && ++retry <
         retry_count) {
23         ESP_LOGI(TAG, "Waiting for system time to be set... (%d/%d)",
         retry, retry_count);
24         vTaskDelay(2000 / portTICK_PERIOD_MS);
25     }
26     time(&now);
27     localtime_r(&now, &timeinfo);
28 }
29 }
```

3.3.1.10 MCU Scheme

TODO: insert MCU skeleton and sensor connections

3.4 Backend

3.4.1 Server

To implement the Server, we chose to use the Spring framework, since is already familiar to us, and allows for a good separation of concerns: controller, services, data model and middlewares.

3.4.1.1 Spring Configuration

A very strong feature caracteristic of the Spring framework is the dependency injection mechanism. This provides abstraction for the programmer implementing a software layer, inside Spring. All he needs to do is specify the service that the layer depends. Spring will take care of the rest.

Yet, Spring will need to scan the project to add available classes to it's context, so, later it can satisfy necessary dependencies. Some classes can be added to the Spring Context directly, using the @Component annotation, or others similar, like the @Controller, @Services, etc. Still, sometimes, it is impossible to add classes to the Context because they are part of some already made up library, like the InfluxDBClientKotlin class, used in this Server. To go around this, the @Configuration annotation is used, so Spring is informed that this class declares one or more Bean methods, meaning methods that return classes to be added to the Spring Context, in order to satisfy dependencies. Sometimes, declaring beans is also useful, not only because we don't have direct access to the library class, but also because we want to make a custom initialization of that class.

```
1  @Configuration
2  class TSDBProductionConfig {
3      private val tsdbConfig = TSDBBuilder("production")
4      @Bean
5      fun getInfluxDBClientKotlin(): InfluxDBClientKotlin {
6          return tsdbConfig.getClient()
7      }
8      @Bean fun getBucket(): Bucket {
9          return tsdbConfig.getBucket()
10     }
11 }
12 ...
```

In the above snippet, two beans are being declared. In particular, this will satisfy the dependencies of the InfluxDB repository class.

3.4.1.2 Spring Security and Google Authentication

TODO

3.4.1.3 Interceptors

An interceptor is a class Spring component that, like the name implies, intercepts the request before and after he passes through the controller. Since the interceptor has access to the HTTP request information, we can use them to process requests, before sending them to the specific handler, and even "cut"the request if desirable.

In the present Server, there is only two interceptors: the Logger intercetor and the Authentication interceptor.

```
1  @Component
2  class AuthenticationInterceptor(
3      val service: UserService
4  ) : HandlerInterceptor {
5      override fun preHandle(request: HttpServletRequest, response: HttpServletResponse
       , handler: Any): Boolean {
6          if (handler is HandlerMethod) {
7              val authorization = handler.getMethodAnnotation(Authorization::class.java)
8
9              // deals with authentication
10             if (handler.methodParameters.any { it.parameterType == User::class.java }) {
11                 // enforce authentication
12                 val token = request.cookies?.find { it.name == "token" }?.value
13                 if (token != null) {
14                     val user = service.getUserByToken(token)
15                     if (user != null) {
16                         UserArgumentResolver.addUserTo(user, request)
17                         return true.also { logger.info("Request: ${request.method} ${
       request.requestURI} - Authorized") }
```

```
18              }
19          }
20          response.status = 401
21          // response.addHeader(NAME_WWW_AUTHENTICATE_HEADER,
    AuthorizationHeaderProcessor.SCHEME)
22          return false.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Unauthenticated") }
23        }
24
25        // deals with authorization
26        if (authorization != null) {
27          val role = authorization.role
28          val token = request.cookies?.find { it.name == "token" }?.value
29          if (token != null) {
30            val user = service.getUserByToken(token)
31            if (user != null) {
32              if (user.userInfo.role == role) {
33                return true.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Authorized") }
34              } else {
35                response.status = 403
36                return false.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Forbidden") }
37              }
38            }
39          } else {
40            response.status = 401
41            // response.addHeader(NAME_WWW_AUTHENTICATE_HEADER,
    AuthorizationHeaderProcessor.SCHEME)
42            return false.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Unauthenticated") }
43          }
44        }
45      }
46      return true
47    }
48    ...
```

In this particular code listing, we can see how useful the interceptor can be. Not only the interceptor has access to the request information, but also he can see metadata about the controller handler to be called. 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 case, the request is immediately terminated, and responded with a 401 Unauthorized.

Similarly, if the controller method has the Authorization annotation, the interceptor will try to authenticate the user and evaluate if the user has the necessary role, defined in the annotation, to be allowed to proceed with the request. If not allowed, a 403 Forbidden is returned.

3.4.1.4 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.

```
1 @Tag(name = "Devices", description = "The Devices API")
2 @RestController
3 class DeviceController(
4   val service: DeviceService,
5   val deviceErrorService: DeviceErrorService,
6 ) {
7   @Operation(summary = "Add device", description = "Add a device
      associated with  email")
8   @ApiResponse(responseCode = "201", description = "Device created", content
      = [Content(mediaType = "application/vnd.siren+json", schema = Schema(
      implementation = CreateDeviceOutputModel::class))])
```

```
9   @ApiResponse(responseCode = "400", description = "Bad request - Invalid
        email", content = [Content(mediaType = "application/problem+json",
      schema = Schema(implementation = Problem::class))])
10  @PostMapping(Uris.Devices.ALL)
11  fun addDevice(
12      @RequestBody deviceModel: DeviceInputModel,
13      user: User
14  ): ResponseEntity<*> {
15      ...
16  }
17  ...
```

The listing above shows a piece of the DeviceController class. This one includes the method addDevice. He, himself, includes the @PostMapping(URI) annotation. By having it, this method will became the handler that will be called when a client makes a request to the URI specified inside the annotation. There is only one parameter of the User type. This indicates Spring to resolve the requested parameter, before the handler execution. Just by specifying that he wants this parameter, the Authentication Interceptor, addressed in the last section, will guarantee that if the request arrives to the controller, mean that the request was indeed authenticated, and it has direct access to the user information.

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

```
1  ...
2  @Operation(summary = "All devices", description = "Get all devices
       associated with our system")
3    @ApiResponse(responseCode = "200", description = "Successfully
       retrieved", content = [Content(mediaType = "application/vnd.siren+
       json", schema = Schema(implementation = DevicesOutputModel::class))])
4    @ApiResponse(responseCode = "401", description = "Not authorized", content
       = [Content(mediaType = "application/problem+json", schema = Schema(
       implementation = Problem::class))])
5    @GetMapping(Uris.Devices.My.ALL)
6    @Authorization(Role.ADMIN)
7    fun getMyDevices(
```

```
8         user: User,
9         @RequestParam(required = false) page: Int?,
10        @RequestParam(required = false) limit: Int?
11    ): ResponseEntity<*> {
12        ...
13    }
14 ...
```

In this listing, we can see the use of the Authorization annotation. This annotation is not part of Spring. This one is a Server domain annotation, and is used to mark the controller methods that require a specific role to be met, before the client can full fill the request. Spring also provides the @RequestParam annotation, where is possible to define specific URI query parameters.

3.4.1.5 Services

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

```
1 @Service
2 class DeviceService (
3     private val transactionManager: TransactionManager,
4     private val userService: UserService,
5 ) {
6 ...
7 }
```

For instance, the snippet above shows the Device Service, which depends on the User Service and a Transaction Manager module.

The very nature of the Services is to provide to provide communication between the Controller and Repository layers, enforcing business logic. Some operations require multiple accesses to the database, and some to more than one database. Yet, attomicity must be guaranteed to maintain consistent data over the system. To solve this problem, most Services depend on the TransactionManager interface, which has only

one method: run(). This method provides atomicity. If successful, meaning no exceptions are drowned, the results are committed, otherwise, a rollback is performed. Other concurrent operations wont see the actions not yet committed. They will only see consistent data.

```
/**
 * Creates a new user.
 * @param password is optional. If not provided, the user will be created without a password.
 */
fun createUser(email: String, password: String?, role: Role): UserCreationResult {
    return transactionManager.run {
        ...
        it.userRepo.createUser(newUser)

        if (password !== null)
            saltPasswordOperations.hashPassAndPersist(password, userId)

        val tokenCreationResult = createAndGetToken(userInfo.email, password)
        ...
    }
}
```

In the snippet above, we, not only creating a User in the database, but also creating a token for that same user, which is stored in a different physical table. If the second step fails, the User must not be created, thus, the transaction must be aborted.

In particular, the JdbiTransactionManager class was implemented, which is specific to interact with the PostgresSQL DB, using the JDBI library, available in Kotlin.

```
@Component
class JdbiTransactionManager(
    private val jdbi: Jdbi
) : TransactionManager {

    override fun <R> run(block: (Transaction) -> R): R =
        jdbi.inTransaction<R, Exception> { handle ->
            val transaction = JdbiTransaction(handle)
            block(transaction)
        }
```

```
11 }
```

This class uses the Jdbi library to create a transaction for which will run a callback within it and commited if it finished normally.

As said above, the Service layer is to be used by the Controller layer. Many inconveniences can happen while fulfilling an operation. When an operation fails, it is helpfull to inform the cause.

```
1 fun getSensorRecordsIfIsOwner(deviceId: String, userId: String, sensorName: String):
      SensorDataResult {
2     return if (!deviceService.existsDevice(deviceId))
3         Either.Left(SensorDataError.DeviceNotFound)
4     else if (!deviceService.belongsToUser(deviceId, userId))
5         Either.Left(SensorDataError.DeviceNotBelongsToUser(userId))
6     else
7         Either.Right(sensorDataRepo.getSensorRecords(deviceId, sensorName))
8   }
```

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 the Either type:

```
1 sealed class Either<out L, out R> {
2   data class Left<out L>(val value: L) : Either<L, Nothing>()
3   data class Right<out R>(val value: R) : Either<Nothing, R>()
4 }
```

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.

Then, each Service class defines it's own domain classes that contain possible errors and successful results.

```
1 sealed class SensorDataError: Error() {
```

```
    object DeviceNotFound: SensorDataError()
    data class DeviceNotBelongsToUser(val userId: String): SensorDataError()
}
typealias SensorDataResult = Either<SensorDataError, List<SensorRecord>>

sealed class SensorErrorDataError: Error() {
    object DeviceNotFound: SensorErrorDataError()
    data class DeviceNotBelongsToUser(val userId: String): SensorErrorDataError()
}
typealias SensorErrorDataResult = Either<SensorErrorDataError, List<
    SensorErrorRecord>>
```

The above snippet illustrates this. When requesting for sensor data, for a specific Device, the provided device ID can be invalid or the user might not have authorization permissions to access that device information.

TODO: falar dos topic subscribe

3.4.1.6 Repository

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

Between the existent solutions to interact with the relational database, we opted to use the Jdbi library, since it is a very balanced solution, not being as low level as the JDBC but also not as high level as the Hibernate. Therefore, to connect to the PostgresDB, two classes were implemented: DeviceDataRepository and UserDataRepository, as shown in the Architecture section.

```
class JdbiDeviceDataRepository(
    private val handle: Handle
) : DeviceDataRepository {
    ...
}
```

Both classes rely on the Handle type, that represents a connection to the database system.

This report wont go further into detailing both classes since, to understand the implementation, the user should read the official support documentation available to build PostgreSQL queries.

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

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

```
1 @Repository
2 class SensorDataRepo(
3     private val client: InfluxDBClientKotlin,
4     bucket : Bucket
5 ) : CollectedDataRepository {
6     private val bucketName = bucket.name
7     val mutex = Mutex() // Use Mutex for synchronization
8     private val MEASUREMENT_PREFIX = "my_sensor" // Modify this line
9     ...
```

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.

Once again, the implementation details to understand how this class interacts with the DB, are not worth mentioning here, since it don't bring any meaningfull knoladge to the comprension of the chosen solution. The reader should, instead read the offitial documentation available in the InfluxDB website.

4

Conclusion