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

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.

For this reason, many MCUs were analyzed. The following list summarizes the main characteristics of each one:

(INSERIR AQUI TABELA)

Given the cost-effectiveness, Wi-Fi connectivity, and low power consumption of the ESP32-S2 microcontroller unit (MCU), its primary competitor would be the MSP430FR413x. While the MSP430FR413x boasts exceptional low power consumption and affordability, it lacks a built-in Wi-Fi module, necessitating an additional purchase and installation. Considering the project's time constraints and the readily available access to the ESP32-S2 and the high price of the MSP430FR413x, we opted for the most secure and convenient option.

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 front-end interface that provides users with visualizations of the data. This user-friendly interface allows users to interact with and gain insights from the collected data in a visually appealing manner. Since we during our academic path worked with react, a popular framework used to build front-end applications, for building our front-end user interface, we opted for using it in the project, we also used React bootstrap,for styling purposes, though at at minimal scale, the majority of website styleing was done using pure css. We also used Webpack witch is a free and open-source module bundler for JavaScript, and typescrit, witch benefit from type checking, enhanced code editor support, and improved code documentation.The webpack was configured to use a typescript loader,

2.2 Existing solutions

3 Architecture

In this chapter, the system architecture will be decomposed. Each main system component will be addressed.

3.1 Overall

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 see/analyze relevant filter-collected data

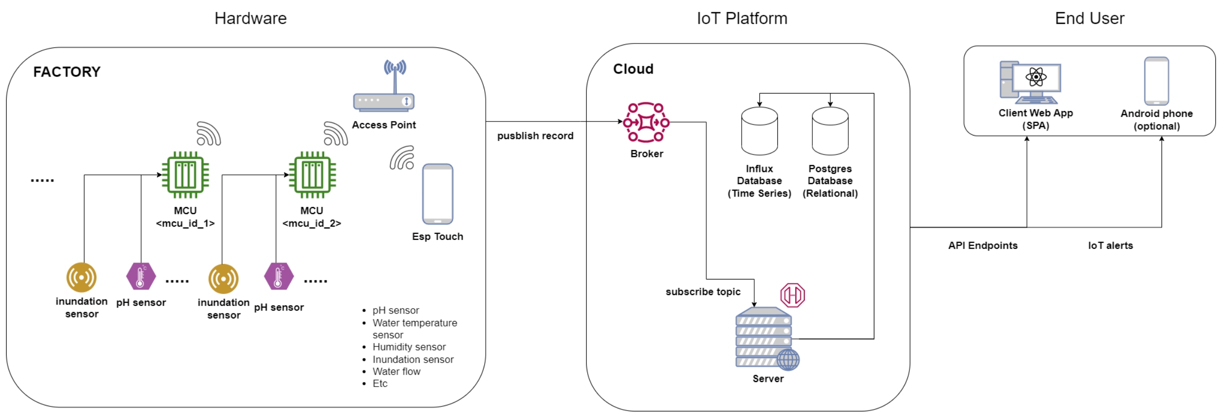

Figura 3.1: Main system architecture

Data flows from left to right. Begins in the MCU, which, then, sends the data to the Backend, where it is processed and analyzed. The Backend also exposes an Web API, where Frontend applications, like the one described here - web application - will use to ask for available data. This includes users, devices, sensors, etc.

3.2 Factory

This component involves 4 sub-components:

- Sensors
- Microcontroller (MCU)
- Access Point (AP)
- ESP Touch

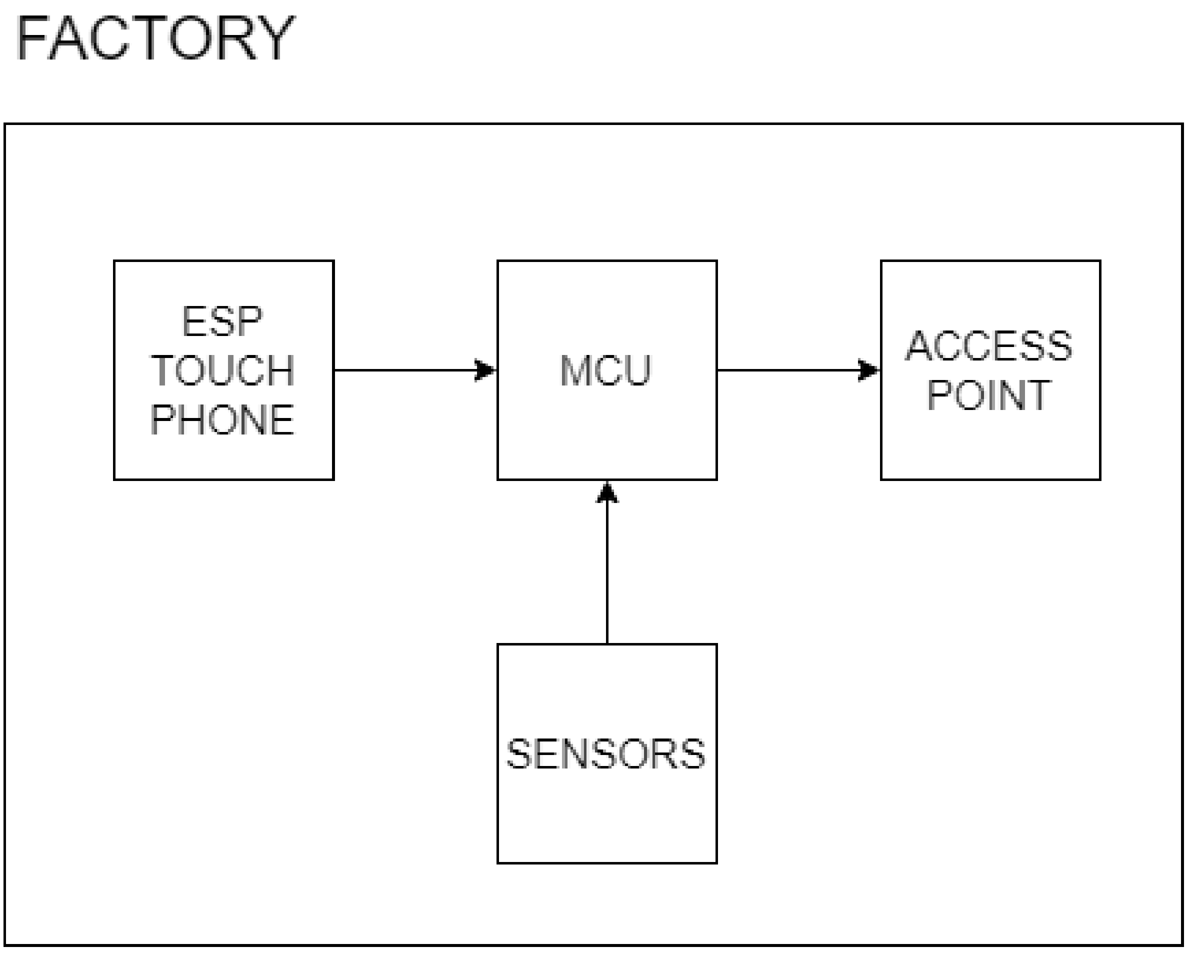

Figura 3.2: Main system architecture

The user will need the ESP Touch android app to configure the MCU, for the first time. By using this, he will use a special network protocol, 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, passing through the AP, which should be basis router.

3.2.1 Sensors

There are 7 main 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 MCU

This component comprises a single hardware microcontroller device, the ESP32-S2. This device is connected with the sensors, and instructs them to make readings, and, eventually, will send them to the Backend, using the Access Point, installed in the room.

3.2.3 Access Point

This is typically a router, allowing the MCU to connect to the internet.

3.2.4 ESP Touch

In order to allow the ESP MCU series to connect to an access point, via WiFi, the ESP Touch - an Android App - is used by the operator, configuring the ESP for the first time, to pass the network credentials, and other information, if necessary.

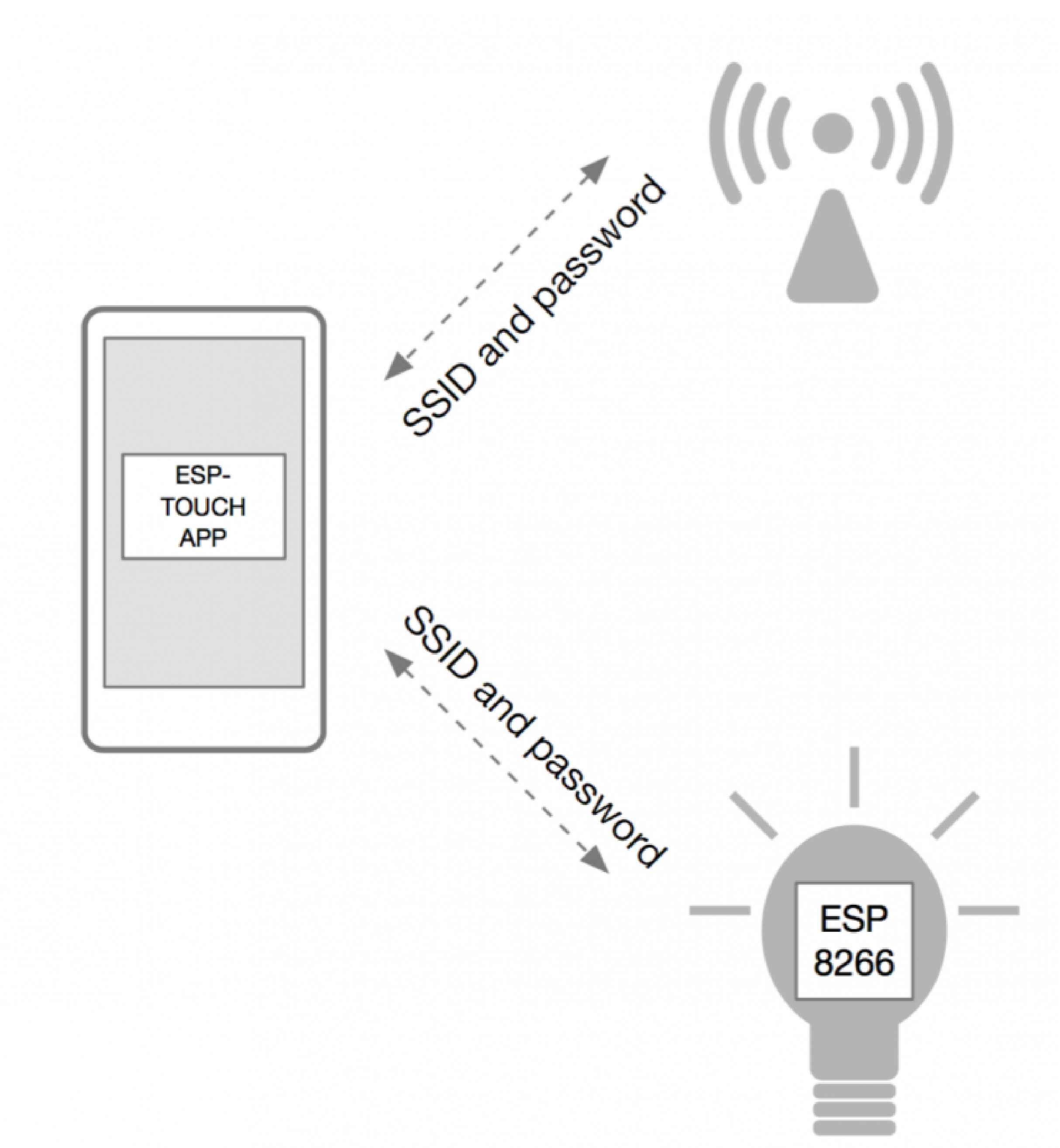

Figura 3.3: ESP Touch - Smart Config Protocol in the ESP8266

3.3 Backend

This component is composed of the fallowing sub-components:

- Spring Server
- MQTT Broker
- Databases

All the sensorial data, coming from the factory, will be headed to the MQTT Broker. The Spring Server, will receive that same data, from the Broker, and persist it, using the two databases, used in this project.

3.3.1 Spring Server

This is the component that stores, fetches and processes all the system necessary data. All the business logic will be enforced here. It also exposes an Web API, so that Frontend applications can interact with the automation system.

3.3.2 MQTT Broker

When a system involves IoT Devices, a new concern emerges: device consumption. Usually, when this happens the traditional solution to send data over the internet - HTTP protocol - is not sufficient anymore. For that reason, a Publish Subscribe protocol was used, for this project - MQTT protocol. The clients will subscribe topics about messages they want to be notified. They also publish messages on specific topics. The Broker constitutes the main software piece that receives, stores and re-sends the messages.

3.3.3 Databases

For this project, we choose to use two different databases:

- Postgres SQL: relational DB
- InfluxDB: Time series DB.

3.3.3.1 Postgres SQL

Even thought, in current project, there is not much data to be persisted, there is some data TODO!!!

3.3.4 Time Series DB

From the moment a system includes multiple sensors collecting data, and sending them to a central Server, the way we store those sensor records, is very important. Usually this type of data doesn't have much aggregation. For this reason, a relational DB wouldn't bring any advantage. A time series DB, on the other hand, is primarily build for this kind of data. It supports fast queries, for this kind of data.

3.4 Web Application

As the third system component, the Web Application appears as a user friendly entry point to the system. It is an Single Page Application (SPA), that interacts with the exposed Web API, in the Backend, in a client-server protocol.

4

Factory

In this chapter, the Factory component will be addressed. Starting with the chosen technologies, then the architecture, connecting all the sub-components involved and finally the implementation of each component.

4.1 MCU

4.1.1 Why?

When a project includes the collection 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 is transited.

4.1.2 Device Selection

One of the requirements for this project was the low cost of the IoT devices. For this reason, many MCUs were analyzed. Texas MCUs, Arduino's, ESP's, etc.

The following list summarizes the advantages and disadvantages of each one:

// TODO: include device table

Another aspect, to consider, and one that was part of the requirements, was the cost.

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.

4.1.3 Firmware

Each MCU needs to be programmed to operate. There are many frameworks we could use to program and load the firmware onto the MCU:

- ESP IDF Framework
- Arduino Framework
- Platform IO

We chose the first one, 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 solution, comparing when used the Arduino framework, for instance.

4.1.3.1 Architecture

The firmware architecture is the following one:

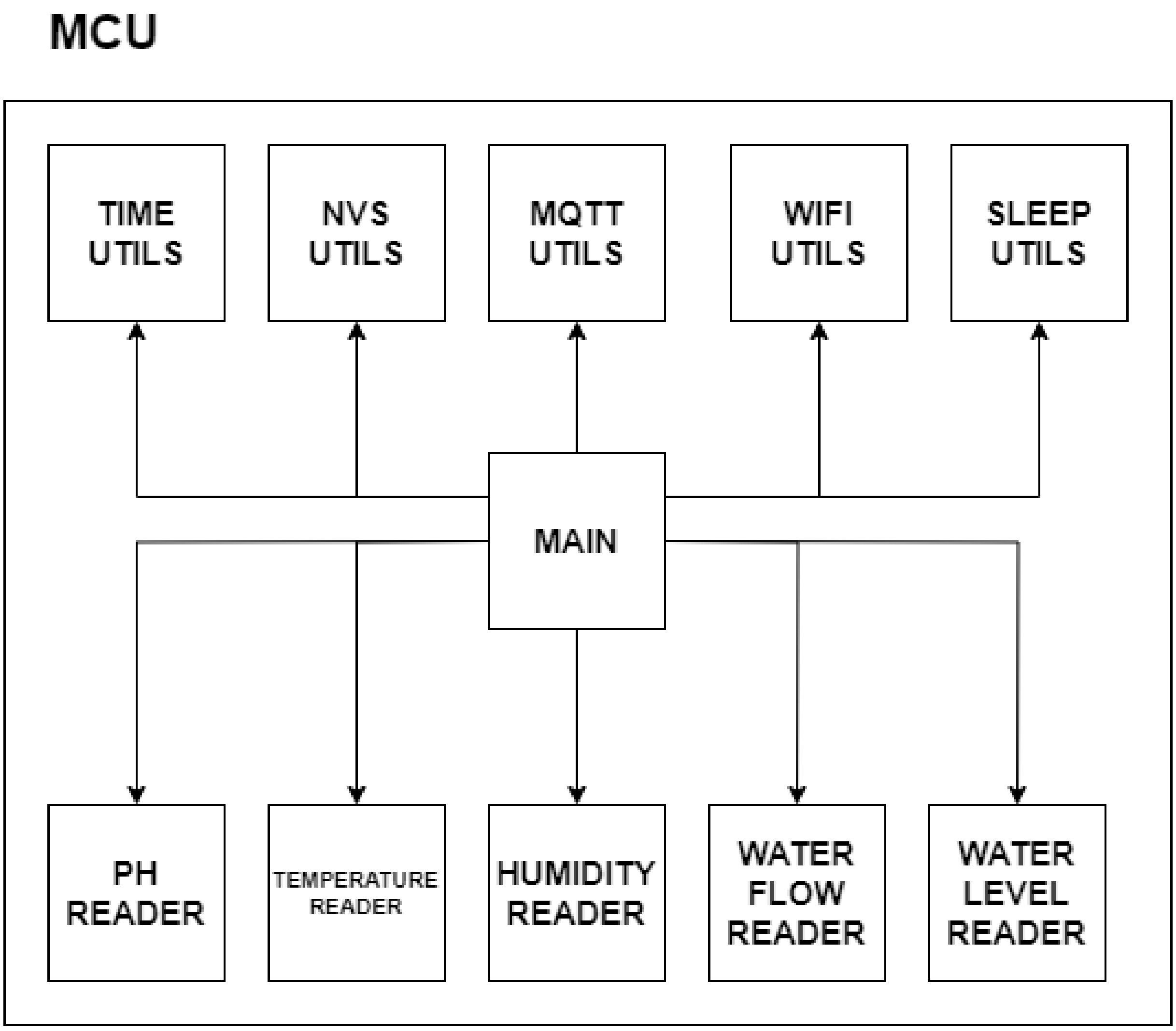

Figura 4.1: MCU firmware architecture

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.

4.1.3.2 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.

4.1.3.3 Behaviour

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

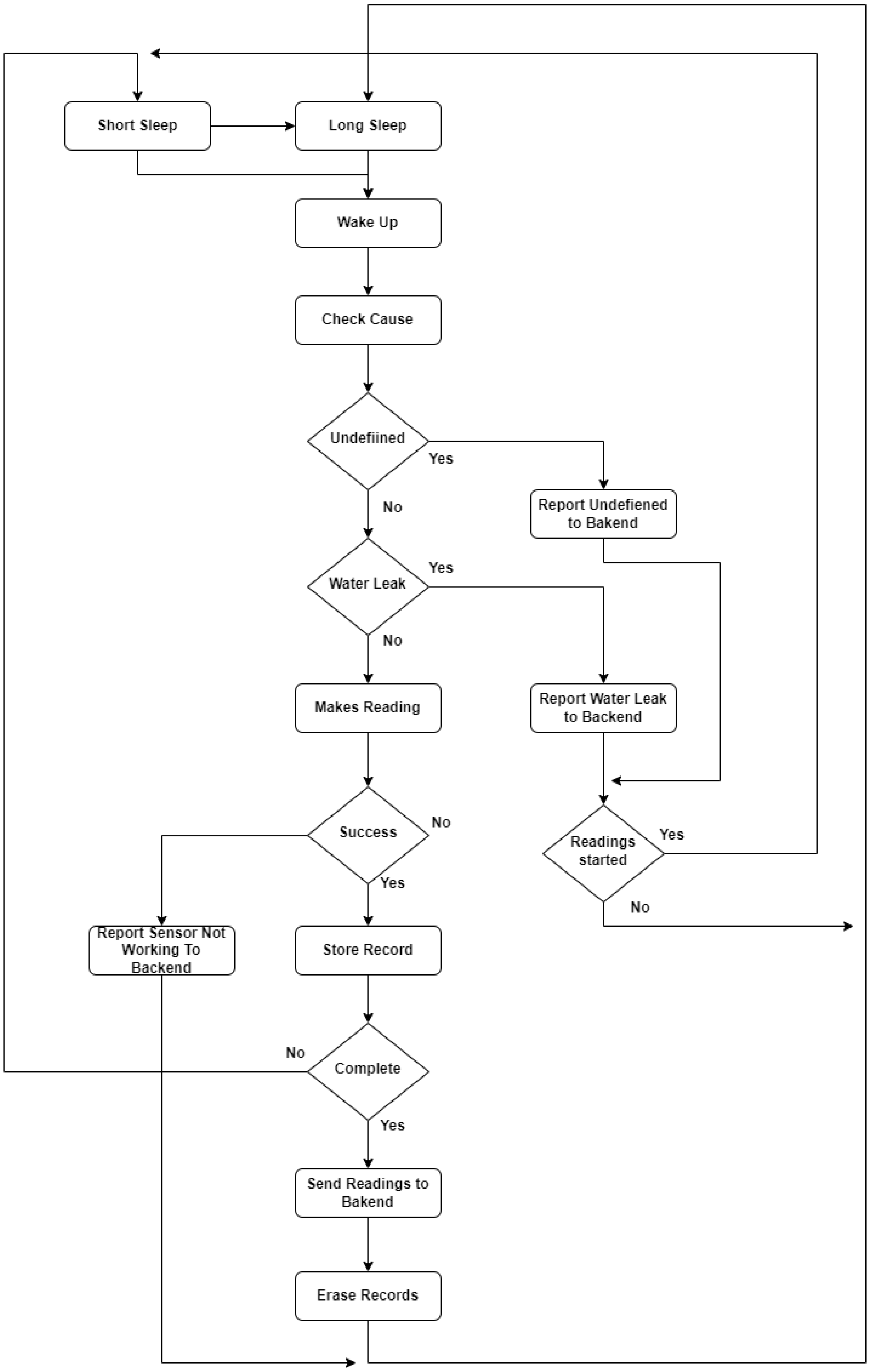

Figura 4.2: ESP Touch - Smart Config Protocol in the ESP8266

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 up counts.

When the devices wakes, it must check it's cause. This is because it can happen for at least three reasons. One is when the cause is undefined. Usually, this happens when the an error occurred 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.

If the wake up is not one of these, means it is time to make a reading. 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 count, 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. When trying to make a reading, the sensors that are not able to make readings, for any reason, are reported to the Backend, so the user is notified about the specific sensors that are malfunctioning.

All this behaviour takes place in the main MCU module.

4.1.3.4 Other modules

Other MCU firmware modules wont be addressed in this report. They were mostly implemented as utility modules. Their implementation details do not contribute to the deep understanding of the final solutions. If the reader wants to know more about this modules, he is welcomed to access the source code for this project, and read the documentation/comments in it.

4.1.3.5 MCU Scheme

TODO: insert MCU skeleton and sensor connections

5 Backend

This chapter will dissect all aspects about the Backend, diving down into the implementation details about each component that makes the complete Backend solution.

We begin with the Spring Server, the technology used, it's architecture, and other details. Then, we address the way the Broker was setup in the Backend. Finally, we see the combined database solution and why we need two of them.

5.1 Spring Server

In this section, all details about the Spring Server will be addressed.

5.1.0.1 API Endpoints

5.1.1 Technology

Even before starting to "code"anything, a technology has to be chosen. There is quickly three choices to implement the Server:

- Spring framework
- Node. js / Express framework
- With Sockets, inside the Board

The third option might seem strange, but is certainly possible. The MCU would not just be responsible to collect data, using the sensors, but also to implement all the Server features, allowing him to compute collected sensor data, persist that data, and expose an API, so that the Web application could use to consume it. This way, the Backend deploy would be necessary, as the Broker component. The Web application would connect directly with the MCU, given they were in the same network. There are some disadvantages with this approach. First, web sockets would be necessary, so that the Frontend app would communicate with the MCU Server. This can become a tricky task, since C, the language to "code"many MCUs, is a low level language, when compared with languages like Kotlin, Python, etc. Second, and since power consumption is an important aspect, the Board would have to be running most of the time, killing any chance of trying to save power.

During our course, we made a lot of web server's, using either Kotlin/Java and JavaScript. Between Spring vs Express, we chose Spring with Kotlin since Spring is a framework which creates an abstraction, dealing with the majority aspects evolving connectivity with the Client. This allows the engineer to focus on the most important task: the domain functionality. Another reason is the Kotlin language, which is not as verbose as Java, but on the other hand, more powerful than the first one.

5.1.2 Architecture

With Spring, we used the following layers

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

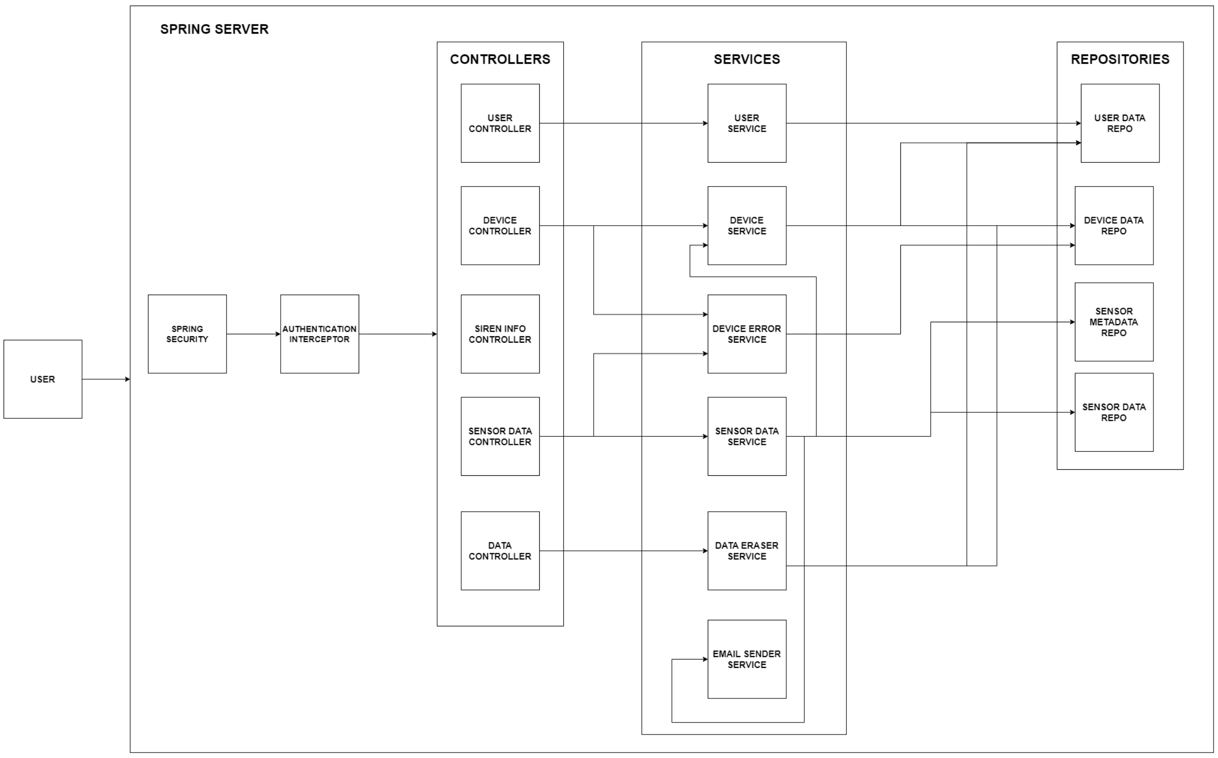

Figura 5.1: Spring Server Architecture

Spring Security will be used only in few situations, as when the user intents to use Google authentication. The interceptors will the 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.

5.1.2.1 Spring Security and Google Authentication

TODO

5.1.2.2 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.

```kotlin
@Component
class AuthenticationInterceptor(
  val service: UserService
) : HandlerInterceptor {
  override fun preHandle(request: HttpServletRequest, response: HttpServletResponse
    , handler: Any): Boolean {
    if (handler is HandlerMethod) {
      val authorization = handler.getMethodAnnotation(Authorization::class.java)

      // deals with authentication
      if (handler.methodParameters.any { it.parameterType == User::class.java }) {
        // enforce authentication
        val token = request.cookies?.find { it.name == "token" }?.value
        if (token != null) {
          val user = service.getUserByToken(token)
          if (user != null) {
            UserArgumentResolver.addUserTo(user, request)
            return true.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Authorized") }
          }
        }
        response.status = 401
        // response.addHeader(NAME_WWW_AUTHENTICATE_HEADER,
    AuthorizationHeaderProcessor.SCHEME)
        return false.also { logger.info("Request: ${request.method} ${
    request.requestURI} - Unauthenticated") }
      }

      // deals with authorization
      if (authorization != null) {
        val role = authorization.role
        val token = request.cookies?.find { it.name == "token" }?.value
        if (token != null) {
          val user = service.getUserByToken(token)
          if (user != null) {
            if (user.userInfo.role == role) {
                return true.also { logger.info("Request: ${request.method} ${request.requestURI} - Authorized") }
            } else {
                response.status = 403
                return false.also { logger.info("Request: ${request.method} ${request.requestURI} - Forbidden") }
            }
        }
    } else {
        response.status = 401
        // response.addHeader(NAME_WWW_AUTHENTICATE_HEADER, AuthorizationHeaderProcessor.SCHEME)
        return false.also { logger.info("Request: ${request.method} ${request.requestURI} - Unauthenticated") }
    }
  }
 }
 return true
}
...
```

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.

5.1.2.3 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. This is duo to the @RestController pre-defined 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:

- Siren Info Controller
- User Controller
- Device Controller
- Siren Info Controller
- Data Controller

```
@Tag(name = "Devices", description = "The Devices API")
@RestController
class DeviceController(
  val service: DeviceService,
  val deviceErrorService: DeviceErrorService,
) {
  @Operation(summary = "Add device", description = "Add a device
    associated with  email")
  @ApiResponse(responseCode = "201", description = "Device created", content
    = [Content(mediaType = "application/vnd.siren+json", schema = Schema(
    implementation = CreateDeviceOutputModel::class))])
  @ApiResponse(responseCode = "400", description = "Bad request - Invalid
     email", content = [Content(mediaType = "application/problem+json",
    schema = Schema(implementation = Problem::class))])
  @PostMapping(Uris.Devices.ALL)
  fun addDevice(
    @RequestBody deviceModel: DeviceInputModel,
    user: User
  ): ResponseEntity<*> {
    ...
  }
```

```
...
```

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 specifing 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.

```
...
@Operation(summary = "All devices", description = "Get all devices
    associated with our system")
  @ApiResponse(responseCode = "200", description = "Successfully
    retrieved", content = [Content(mediaType = "application/vnd.siren+
    json", schema = Schema(implementation = DevicesOutputModel::class))])
  @ApiResponse(responseCode = "401", description = "Not authorized", content
    = [Content(mediaType = "application/problem+json", schema = Schema(
    implementation = Problem::class))])
  @GetMapping(Uris.Devices.My.ALL)
  @Authorization(Role.ADMIN)
  fun getMyDevices(
     user: User,
     @RequestParam(required = false) page: Int?,
     @RequestParam(required = false) limit: Int?
  ): ResponseEntity<*> {
     ...
  }
...
```

In this listing, we can see the use of the Authorization annotation. This annotation is not part of Spring. This one is a project 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. Also, Spring provides the @RequestParam annotation, where is possible to define specific URI querie parameters.

5.2 Repository

In our problem domain, we have two kinds of data to persist:

- Aggregate data: Users, Devices, etc
- Temporal data: sensor data

A relational DB was used for the first kind. This type of data is related intrinsically: Users have Devices, Devices have errors, 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. PostgresSQL was chosen duo to being an open source, powerful and high reliable database.

On the other hand, we chose to use a time serial DB to store sensor records. This type of DB are specially designed to store this kind of data, where records are composed of a value that matches a given timestamp. This is data that is not sensible and thus not a problem for this kind of DB. We chose to use the Influx DB for being open source, and easy to deploy.

6

Conclusion