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 Silva (a47128@alunos.isel.pt)

Relatório para a Unidade Curricular de Final Project
da Bachelor's Degree in Computer Science and Software Engineering

Professor : Doctor Rui Duarte

01 – 05 – 2023

Abstract

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 person/company that posses condensing boilers, is obligated to neutralize the pH water.

Mommertz is a German company that sells "filters" that neutralize the water, so that the restrictions obligations are 100 percent full filed according to the law.

This filter is composed of granulates (little rocks) that neutralize the water pH, plus the water that comes from the boiler and runs through the filter.

As the water flow and is neutralized, the granulates begin to decompose, and so, the neutralization effectiveness begins to decrease. The pH wont be neutralized. On the contrary, if too much granulates are placed, the pH will be too high Also, with time, there could be water leaks. The water, for some reason, could stop flowing. Like any other industrial system, it will eventually need maintenance. Currently, this is done, manually, which has some disadvantages, since this type of systems are usually in places of difficult access, like basements in Germany. Also, manual checkups introduce human error, and unnecessary maintenance. Our system aims to solve this problem.

This system uses IoT technology to automate the above described system. It is composed by three main components. The first one comprises an MCU, connected to multiple sensors, that monitor key filter variables. This data is to be sent to the central server (Backend). The Backend makes the second component, which is composed by the Spring Server that receives and analyses the data, collected by the sensors, and exposes a Web API, so other systems can consume that same data. It is also composed by the databases used to persist the data, and a broker, being the intermediary between the MCU and Backend. Finally, as the third component, and to provide a way to the

user observe the collected data, a user friendly Web application will serve as the entry point to the system.

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

1.1.1 Project choice

We choose to develop a project which involves as many topics as possible, addressed in our course. This one includes, not just Software, but also Hardware. Hardware is an area that sometimes gets less attention, compared with Software. This might be duo to the fact that the most jobs available focus on Software. Maybe because hardware component might not seem to be as "exciting"as the software or it might be be harder, and it requires special skills in specific languages, as the C and C++ language. Even so, we choose a project that involved hardware, not just to develop our skills in that same component, but also wanted to build a full system in which the hardware interacted with the software.

1.1.2 Background

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 person/company that posses condensing boilers, is obligated to neutralize the pH water.

Mommertz is a German company that sells "filters"that neutralize the water, so that the restrictions obligations are 100 percent full filed according to the law.

This filter is composed of granulates (little rocks) that neutralize the water pH, plus the water that comes from the boiler and runs through the filter.

As the water flow and is neutralized, the granulates begin to decompose, and so, the neutralization effectiveness begins to decrease. The pH wont be neutralized. On the contrary, if too much granulates are placed, the pH will be too high Also, with time, there could be water leaks. The water, for some reason, could stop flowing. Like any other industrial system, it will eventually need maintenance. Currently, this is done, manually, which has some disadvantages, since this type of systems are usually in places of difficult access, like basements in Germany. Also, manual checkups introduce human error, and unnecessary maintenance.

The very nature of this problem seems a very good candidate to the use of IoT technology. Using sensors to monitor some aspects of the neutralization system, we can, not only extract key information about the "filter"use, but also predict when the next maintenance will be needed.

Note: our system wont eliminate the need for human intervention. Manual maintenance will always be needed, specially done by a professional in the area. In other way, it will automate the control of the "filter"process, improving not only that, since the sensors wont be subjected to the human error, but also improving costs, since the user doesn't have to pay a professional to check the filter, when it is not necessary to do so.

1.1.3 Document organization

In the second chapter, the main project architecture is introduced. In the third chapter, the micro-controller component is described, along with other related modules, like the associated sensors and an app used to make the MCU connection with the WiFi. In the forth chapter, the Backend component is addressed, including every sub-component that it includes: Spring Server, Web API, Broker and Databases. In here, the chosen

operations exposed by the Web API are explained, the data models, why we chose to have two different databases, etc. In the fifth chapter tiers down the Web Application developed, describing the type of Website developed, the logical page navigation, etc. The third, forth and fifth chapter also explain the choice of the technologies, their advantages, disadvantages and trade offs.

1.1.4 State Of Art

// TODO: talk about Bernardo work

2

Architecture

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

2.1 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

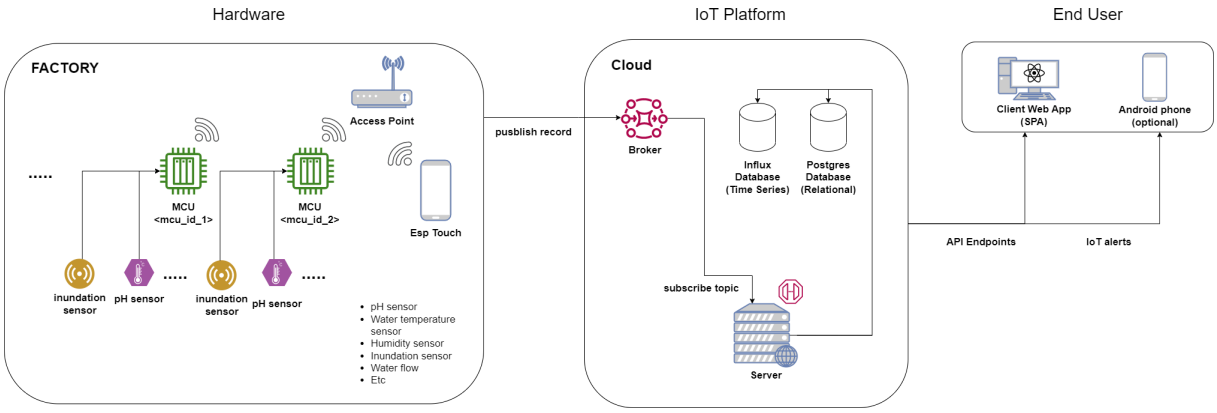

Figura 2.1: Main system architecture

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.

2.2 Factory

This component involves 4 sub-components:

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

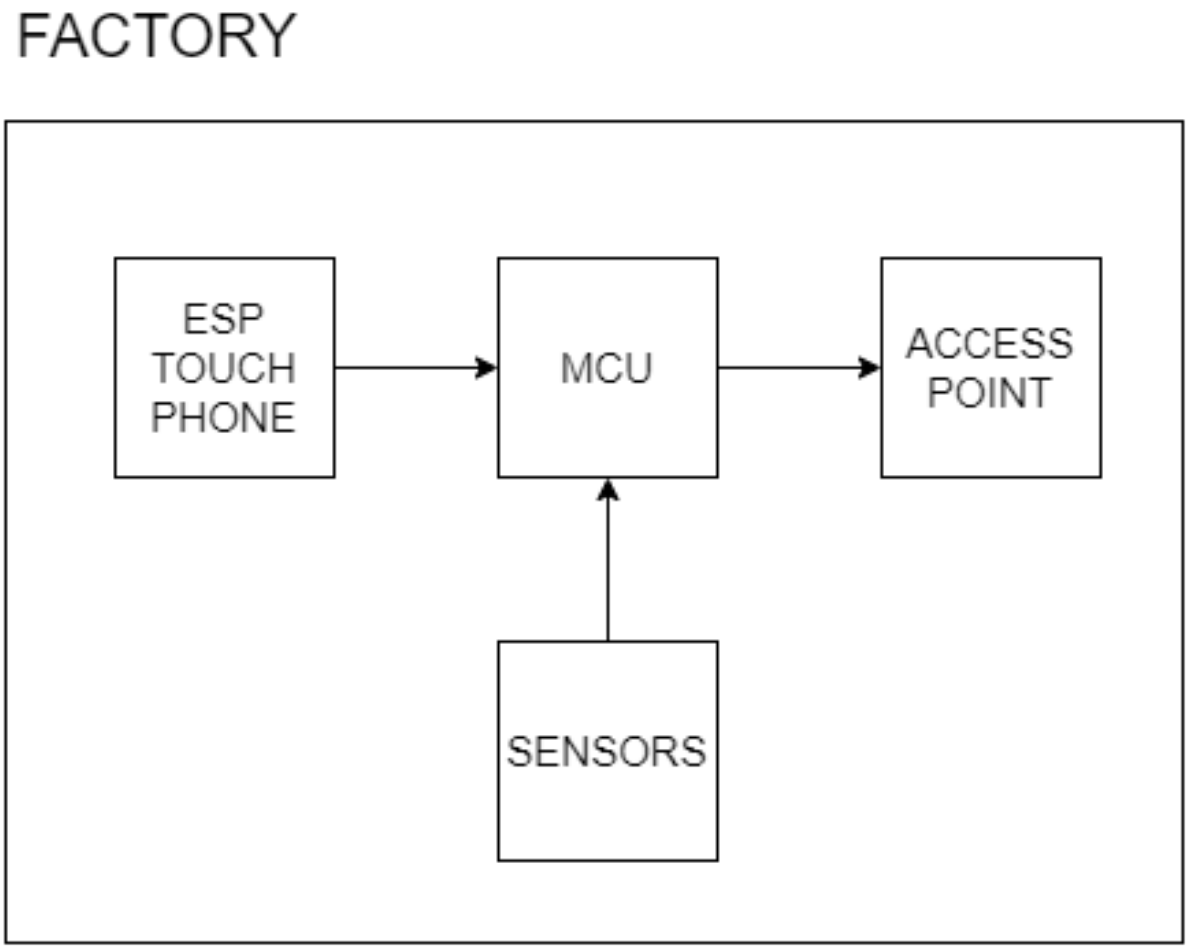

Figura 2.2: Main system architecture

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.

2.2.1 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.

2.2.2 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.

2.2.3 Access Point

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

2.2.4 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.

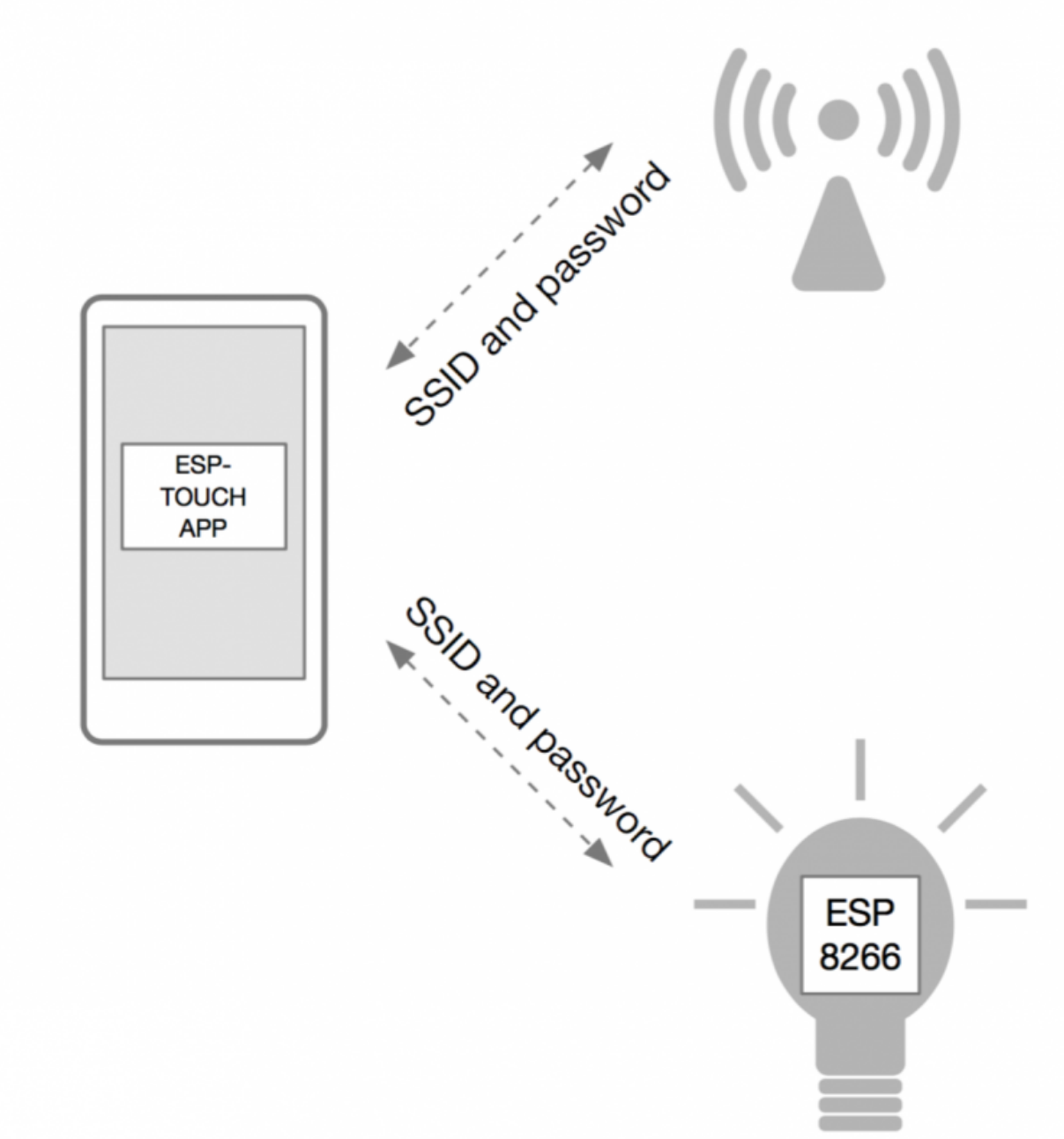

Figura 2.3: ESP Touch - Smart Config Protocol in the ESP8266

2.3 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.

2.3.1 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.

2.3.2 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.

2.3.3 Databases

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

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

2.3.3.1 Postgres SQL

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

2.3.4 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.

2.4 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.

3

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.

3.1 MCU

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.3.1 Architecture

The firmware architecture is the following one:

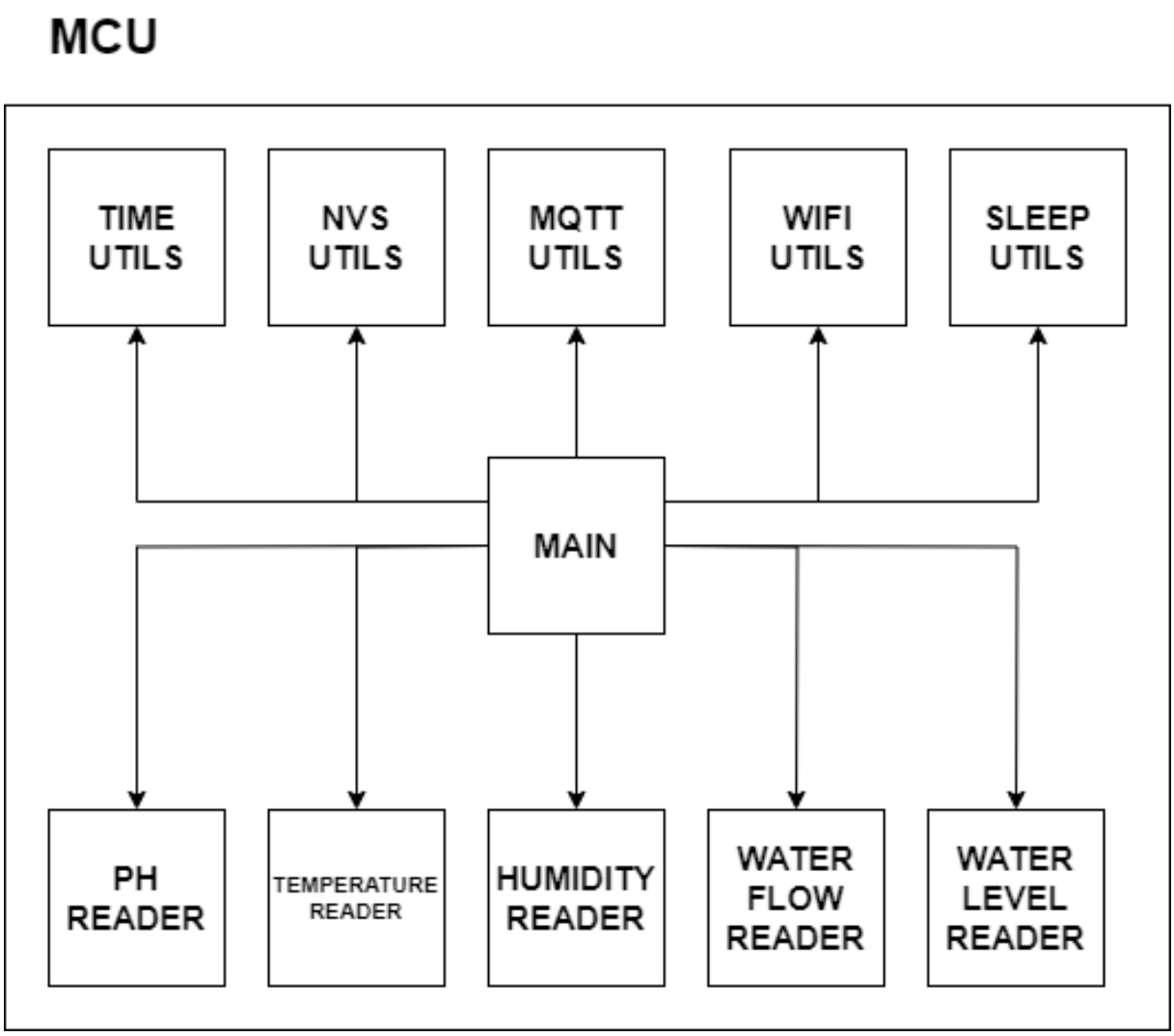

Figura 3.1: MCU firmware architecture

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.

3.1.3.2 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.

3.1.3.3 Behaviour

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

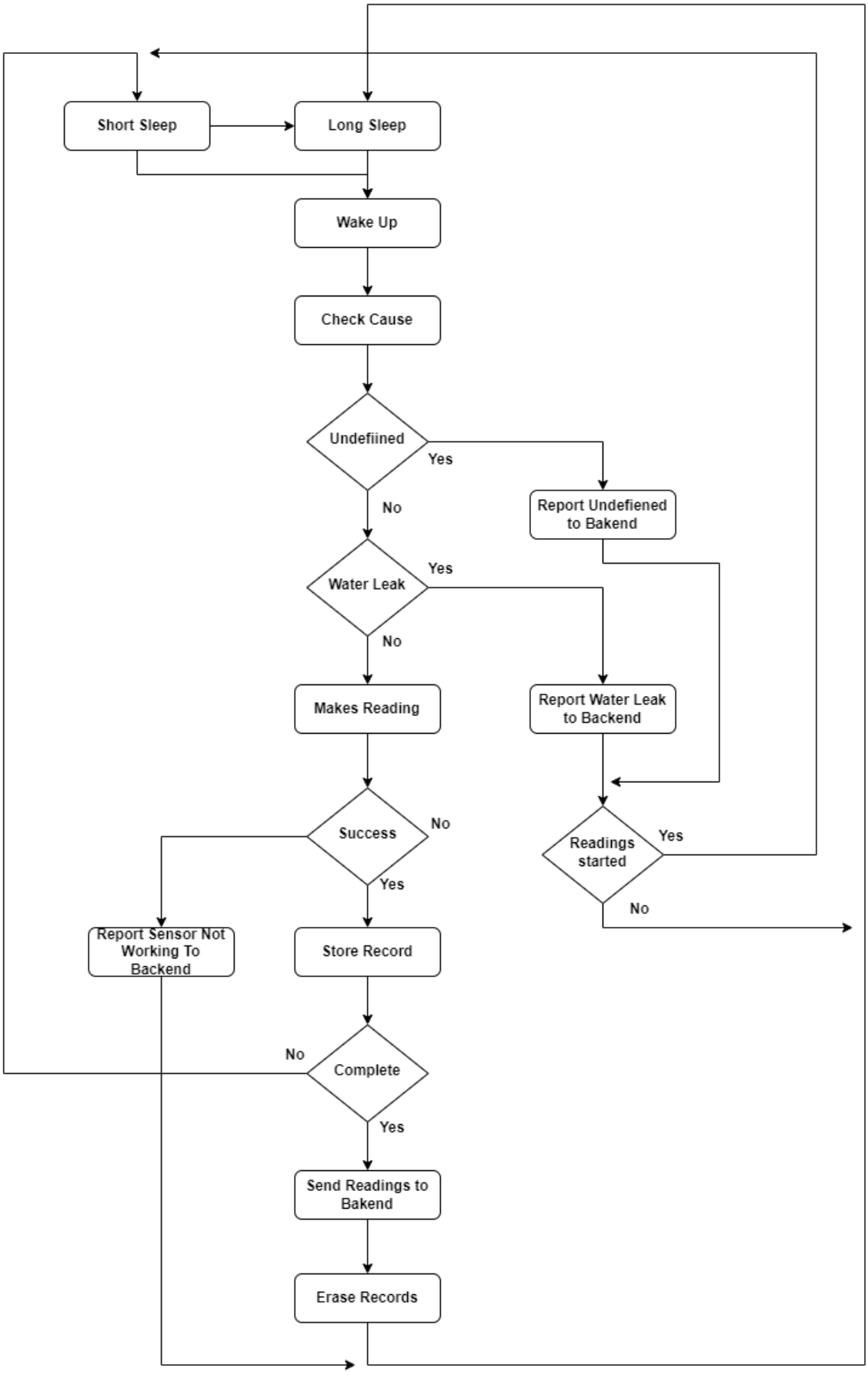

Figura 3.2: ESP Touch - Smart Config Protocol in the ESP8266

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.

3.1.3.4 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.

3.1.3.5 MCU Scheme

TODO: insert MCU skeleton and sensor connections

4

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.

4.1 Spring Server

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

4.1.0.1 API Endpoints

4.1.1 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.

4.1.2 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.

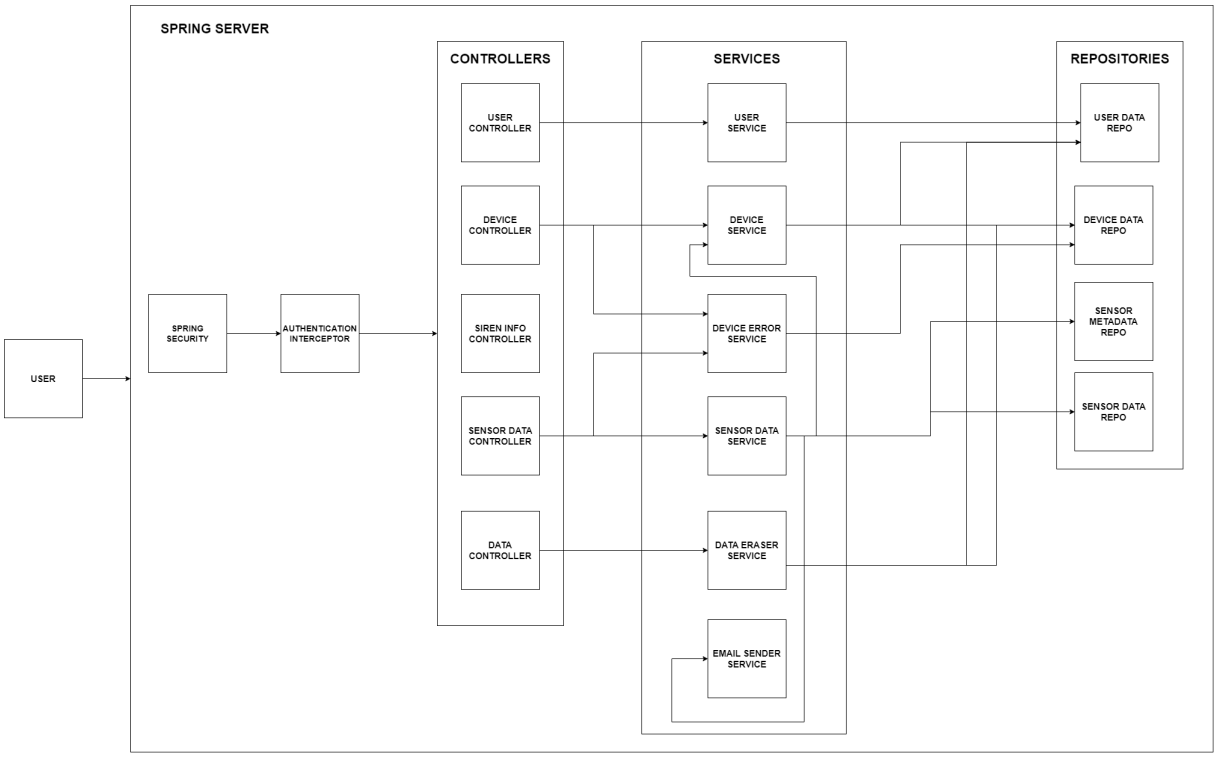

Figura 4.1: Spring Server Architecture

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.

4.1.2.1 Spring Security and Google Authentication

4.1.2.2 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.

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.

4.1.2.3 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

```
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 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.

```
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 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.

4.2 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.

5

Conclusion