Venue Environment System

Project Report

Selina Ceban 315235,
Lóránt Bekó 315189,
Adriana Leišytė 333678,
Rojus Paukštė 315221,
Matas Kairys 315188,
Nerijus Savickas 315173,
Javier Abreu Barreto 315237,
Daniel Lopes 315274,
Mikkel Bjørn Hansen 315167

Supervisor: Kasper Knop Rasmussen, Joseph Chukwudi Okika, Erland Ketil Larsen

VIA University College

Software Technology Engineering

Semester 4

2023

Table of content

Abstract 1
Introduction 2
Analysis 3
Requirements 3
Use Cases 6
Use Case Diagram 11
Non-Functional Requirements 13
Domain Model 14
Rojus 14
Design 16
Architecture 16
Architectural Qualities 17
Front End Design 18
Architecture 18
Technologies 18
Design Patterns 19
UI Design Choices 20
Cloud Design 23
Technologies
Lolek 23
Design Patterns 23
Class diagram 24
IoT Design 27
Architecture 27
Technologies 28
Design Patterns 28
UI Design Choices 29
Implementation 30
IoT Implementation 30
Cloud Implementation 31
Web Implementation 35
Continuous Integration & Development 42

Web CI/CD 42
Git, Branching & Merging 43
DevOps Tools Used 43
Integration of DevOps into Workflow 43
Impact of DevOps Tools/Methods 44
IoT CI/CD 44
General workflow decisions 44
Git and version control 45
Branching strategy 45
GitHub Workflows 46
Development Tools 47
Cloud CI/CD 49
Git and Version Control 49
Branching Strategy 49
GitHub Workflows 50
IDE/Development 50
Deployment 50
Testing 51
Black-box testing (Acceptance tests) 51
IoT Test 53
Front-End Test 54
Results and Discussion 57
Conclusions 58
Project future 60
Sources of information 62
Appendices 63

• Abstract

Rojus

The overall goal of the project is to implement a system that would aim towards improving air quality within indoor areas globally. It will contain all the necessary features needed, to achieve such a goal within one system. The solution will take atmosphere measurements, display them and react to them, as such, helping indoor venue or room owners to take care of the air quality within their property. A system like this could potentially create a safer and healthier environment for social gatherings around the world.

The resulting solution is a 3-tier distributed system. Different tiers are dedicated for key parts of the system: Front-end for user interface, Cloud/Back-end for data management and communication, IoT for data gathering and device management. All 3 tiers are using different middleware for communication between one another: Front-end and Cloud/Back-end use Spring Web API, IoT and Cloud/Back-end use Websockets in combination with LoraWAN network.

The solution grants their users the ability to view sensor data, manage logs of that data and toggle climate control that uses their own chosen limit values to adjust the atmosphere within the venue

● Introduction

Venue management system focuses on improving air quality within indoor locations. Throughout their living period, humans tend to often spend their time indoors. A big concentration of them usually results in poor air quality. Air quality has a massive impact for all living beings.

As a matter of fact, more than 15% of deaths worldwide are caused by diseases forming due to environmental pollution. This pollution is especially relevant to indoor venues, where second hand smoking, high levels of humidity due to a mass of people are all present. An example of these types of places is a nightclub. Apart from pollution and high humidity, CO2, light and sound levels, along with a possibility of a burglary of theft also pose a threat to customers and employees. As such, it is important to keep track of the surrounding atmosphere to preserve the well-being of people within the indoor venues. With this in mind, the team has come up with an idea to create a product to measure and manage possible threat factors in clubs as there are little to none similar systems on the market.

The main goal is to take a weight off of the club owner's shoulders, as the system is to notify them if one of the measured levels reach a critical limit. Furthermore the system will also take certain actions to deescalate values that are too high or too low. However, legal atmosphere level requirements will not be taken into consideration. The team has carefully analysed the problem statement, to create proper requirements on which the use cases are based upon. Use cases are a fundamental part of the Domain Model, and therefore the design of the system, leading to precise implementation and hence, testing.

● Analysis

An analysis has been conducted to help the team implement a system, capable of bringing satisfactory results according to the clients' needs. This section puts focus on the process of analysis of the requirements formed by the product owner. Amongst multiple requests by the product owner, who has put emphasis on having the system being connected to a specific IoT device and using LoraWAN network to communicate between services. Moreover, the product owner has requested to have 3 types of users – an administrator, a manager and a venue guest, all of which would have access to different functionalities within the system.

○ Requirements

Rojus

The following list of requirements was provided by the product owner in the form of user stories. They have been sorted by priority and importance, to aid the team in organising their work. As mentioned previously, there will be 3 different roles of user actors within the system.

Critical

1. As a manager, I want to be able to view the current temperature, so that I can keep track of whether it is safe for people to stay inside.
2. As a manager, I want to be able to view current humidity levels, so that I could prevent bad air quality.
3. As a manager, I want to be able to view current CO2 levels, so that I know if there is enough oxygen in the venue.

4. As a manager, I want to be able to let the system control the air conditioner automatically, so that the temperature and humidity levels of the room stay balanced without my intervention.
5. As a manager, I want to be able to let the system control the air ventilation automatically, so that the CO2 levels within the venue stay balanced without my intervention.

High Priority

6. As a manager I want to extract a log of the current sensor values, and give a comment, so that I can share them with my employees.
7. As a manager I want to be able to see critical values of sensor values in the logs, so that I would know when values have exceeded the set limits.
8. As a manager I want to see previous sensor recordings, so that I can reference the data for later use.
9. As a manager I want to delete logs, so that redundant or old data is not within the system.
10. As a manager, I want to see the currently set limits of acceptable temperature values, so that I can reference them after they are set.
11. As a manager, I want to set the limits of acceptable temperature values, so that the system could trigger certain events automatically, such as printing logs and controlling the air conditioner.
12. As a manager, I want to set the limits of acceptable CO2 values, so that the system could trigger certain events automatically, such as printing logs and controlling the air conditioner.
13. As a manager, I want to set the limits of acceptable humidity values, so that the system could trigger certain events automatically, such as printing logs and controlling the air conditioner.
14. As a manager, I want to be notified when CO2 levels exceed the set limits, so that I would know when I need to take action towards regulating air quality.

15. As a manager, I want to be notified when temperature has exceeded the set limits, so that I would know when I need to take action towards regulating the temperature.

16. As a manager, I want to be notified when humidity levels exceed the set limits, so that I would know when I need to take action towards regulating the humidity.

Low Priority

17. As a manager, I want to be able to view current PIR sensor values, so that I know if there is movement in the venue.

18. As a manager, I want to see the warning PIR sensor showing activity at a set time, so that I know if there is movement in the venue when the venue is closed.

19. As a manager, I want to have control over when the PIR sensor is active, so that I could avoid setting off the sensor during working hours.

20. As a manager, I want to be able to view recent light sensor values, so that I could prevent economically inefficient use of power within the venue.

21. As a manager, I want to be able to view recent sound sensor values, so that I know that sound is at acceptable levels.

22. As a manager, I want to be informed about connection errors, so that I could alert those responsible.

23. As a venue guest, I want to be able to view the sensor values, so that I can keep track of whether it is safe for me to stay inside.

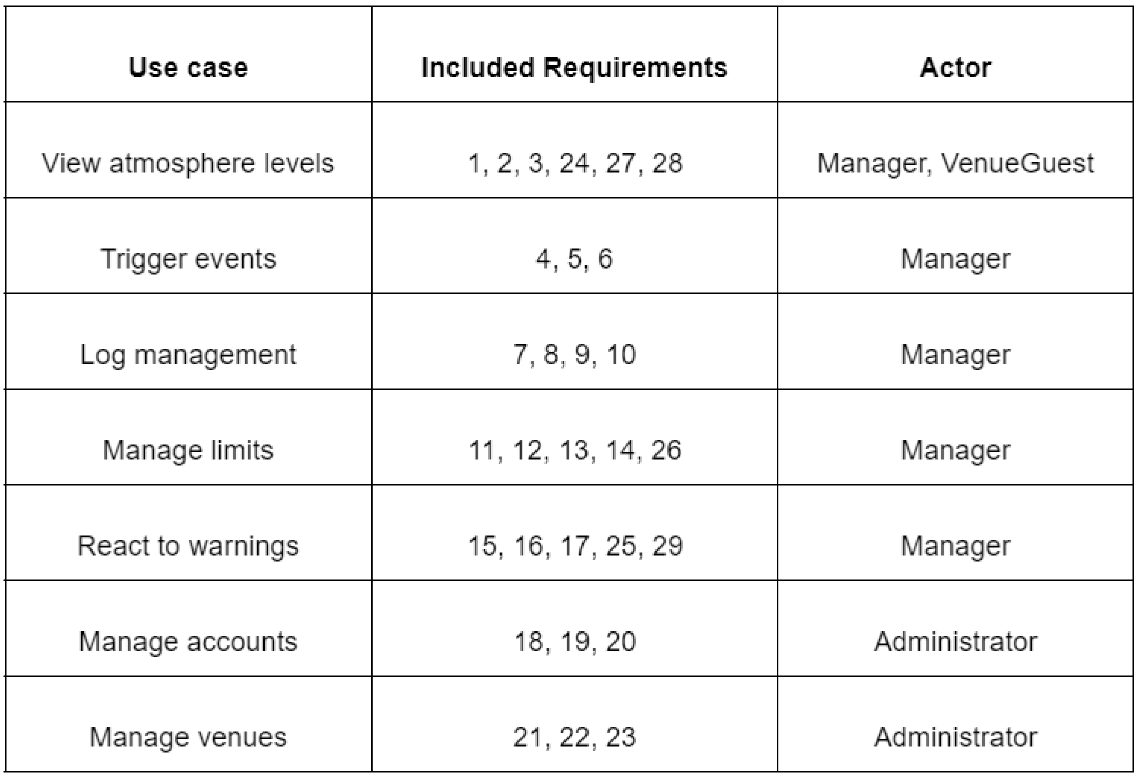
Future

24. As an administrator, I want to create manager accounts, so that I could assign them to be responsible for a certain venue.
25. As an administrator, I want to edit manager accounts, so that I could alter their assigned venue.
26. As an administrator, I want to remove manager accounts, so that I could restrict access to the system for people who are no longer part of the staff.

27. As an administrator, I want to create venues, so that I could keep track of multiple facilities at a time.
28. As an administrator, I want to edit venues, so that I could keep track of multiple facilities at a time.
29. As an administrator, I want to remove venues, so that I could keep track of multiple facilities at a time.

- **Use Cases**

Rojus, Lolek



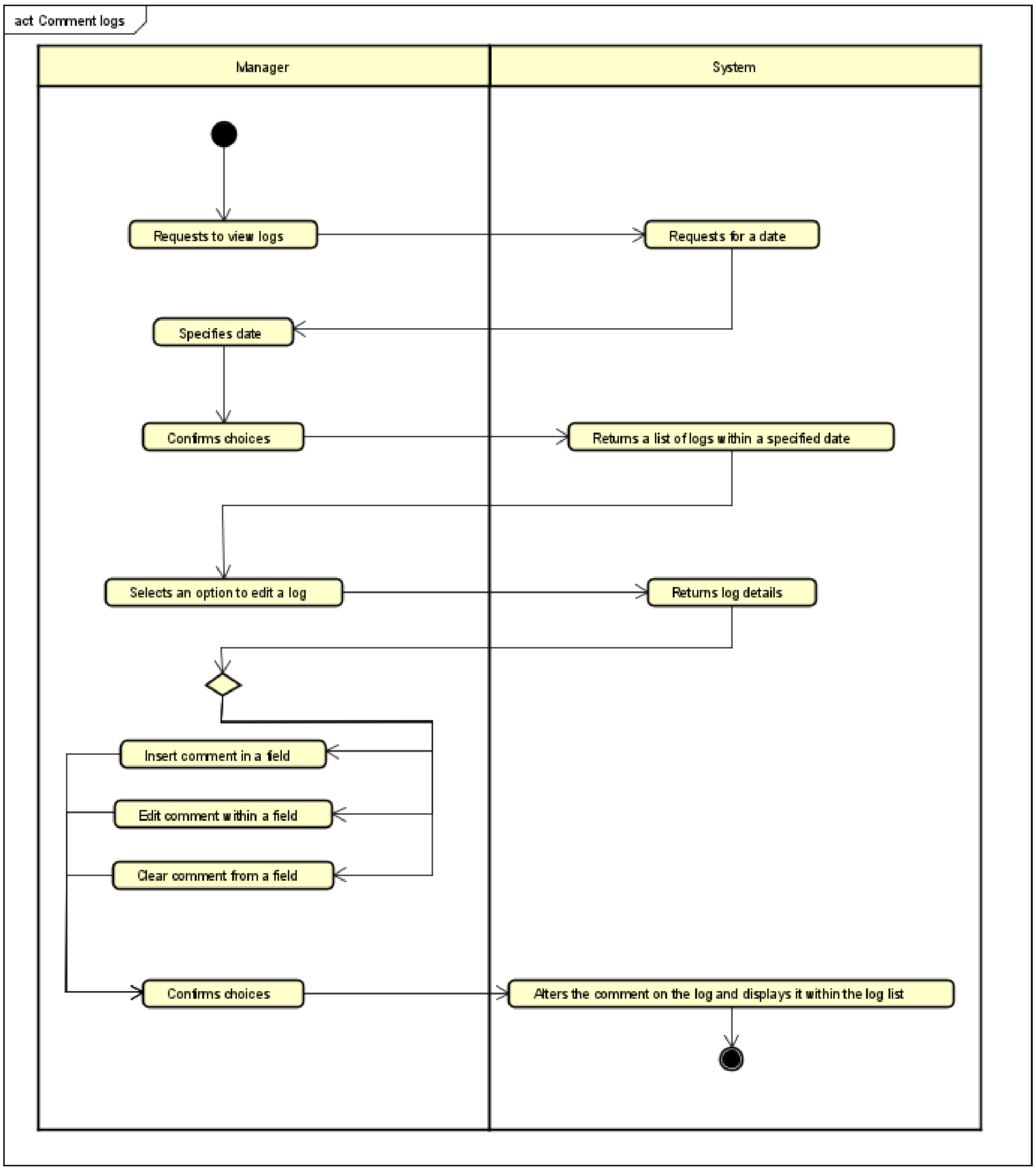
Use case	Included Requirements	Actor
View atmosphere levels	1, 2, 3, 24, 27, 28	Manager, VenueGuest
Trigger events	4, 5, 6	Manager
Log management	7, 8, 9, 10	Manager
Manage limits	11, 12, 13, 14, 26	Manager
React to warnings	15, 16, 17, 25, 29	Manager
Manage accounts	18, 19, 20	Administrator
Manage venues	21, 22, 23	Administrator

Use cases were produced by grouping up the functional requirements, based on their focus. Their aim is to provide guidelines for the team on what the user experience should be for the system. In addition, the use cases also have their respective descriptions to elaborate on them in more detail.

Use Case	**Log Management**	
Covered user stories	6, 7, 8, 9	
Actor	Manager	
Precondition	For the manager to interact with logs, they must first exist in the system within a specified date.	
Postcondition	The manager may view, comment and delete log data of past sensor data	
Base sequence – View logs	1. 2. 3. 4. 5.	The manager selects an option to view logs The system requests for a date The manager selects a specific date from the date picker The manager confirms their choices The system displays a list of logs for the selected date
Base sequence – Comment logs	1. 2. 3. 4. 5. 6. 7. 8. 9. 10.	The manager selects an option to view logs The system requests for a date The manager selects a specific date from the date picker The manager confirms their choices The system displays a list of logs for the selected date The manager selects an option to edit one of the logs The system opens up a pop-up with the selected log details The manager alters the content within the comment comment field The manager saves changes The system changes the comment on the log and displays it in the log list

Branch sequence		N/A
Sub use case	N/A	
Note	● For the logs to exist, the IoT device must have been online to record and send sensor value data to the system at the given date.	

Moreover, interaction diagrams have been created alongside the use cases to describe a series of user actions and the systems reactions that must be taken to achieve the postcondition of the use case. Activity diagrams also specify additional information regarding some series of actions the user must take.

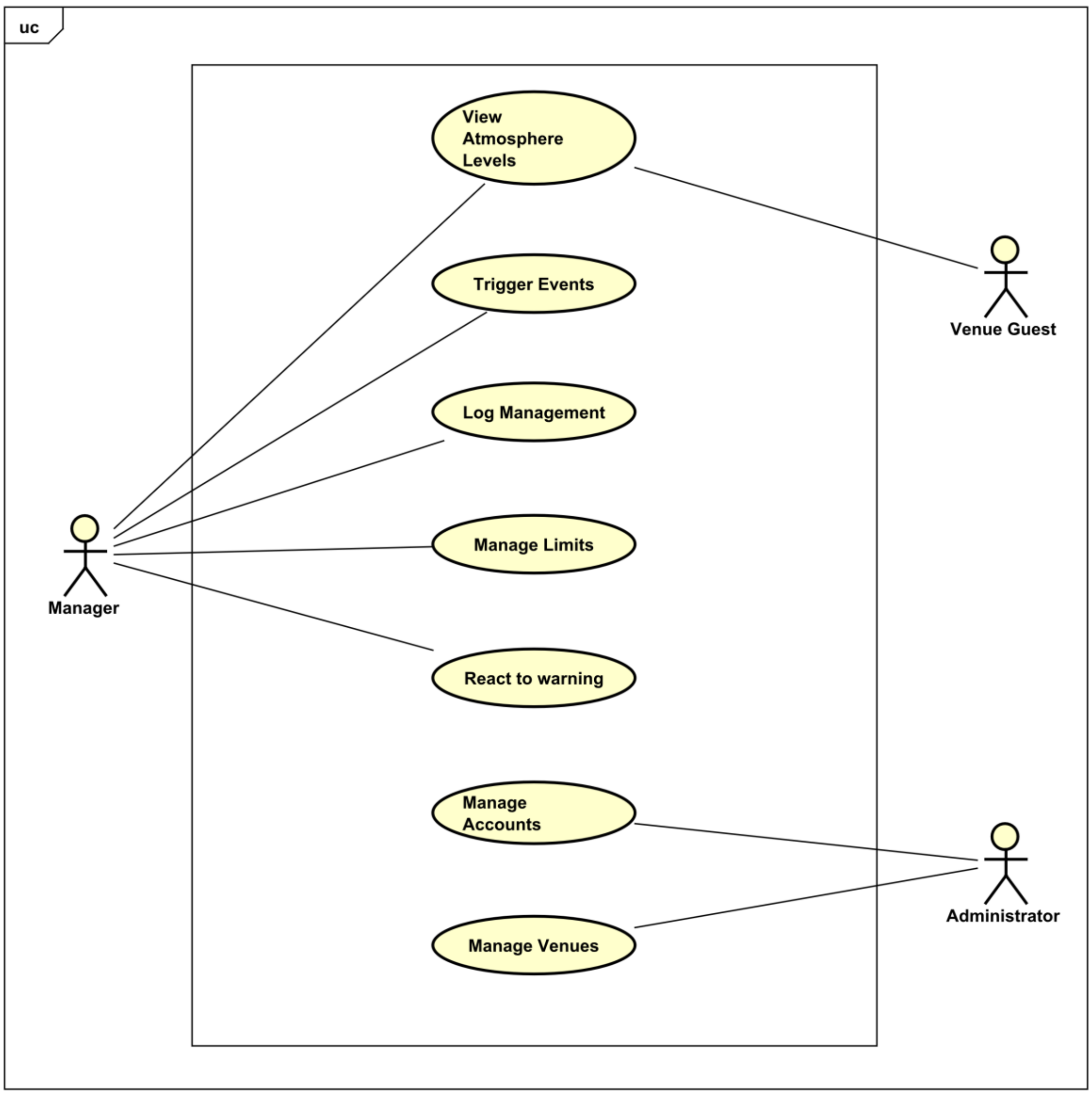

act Comment logs
Manager
System
Requests to view logs
Requests for a date
Specifies date
Confirms choices
Returns a list of logs within a specified date
Selects an option to edit a log
Returns log details
Insert comment in a field
Edit comment within a field
Clear comment from a field
Confirms choices
Alters the comment on the log and displays it within the log list

- **Use Case Diagram**

Rojus

As indicated within the Use Case Diagram, The plan was to have 3 types of actors: Administrator, Manager and Guest. All of these actors were supposed to have access to different parts of the system and receive unique user experience compared with each other. Currently in the system there is only one type of user - manager.

- **Non-Functional Requirements**



Rojus

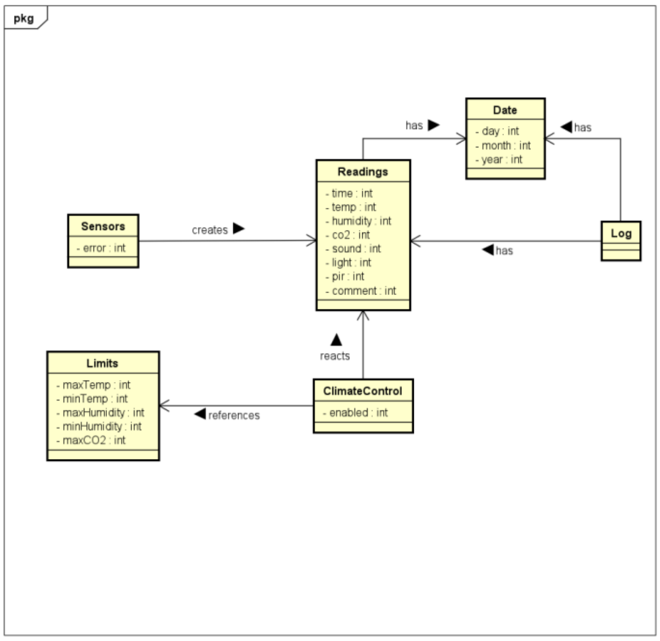
The non-functional requirements listed below were made with the goal of making sure the system is implemented to be usable and effective. In the case of not following the requirements listed, the implemented solution could become inconsistent, slow and insecure. All of these factors could potentially lead to customer dissatisfaction.

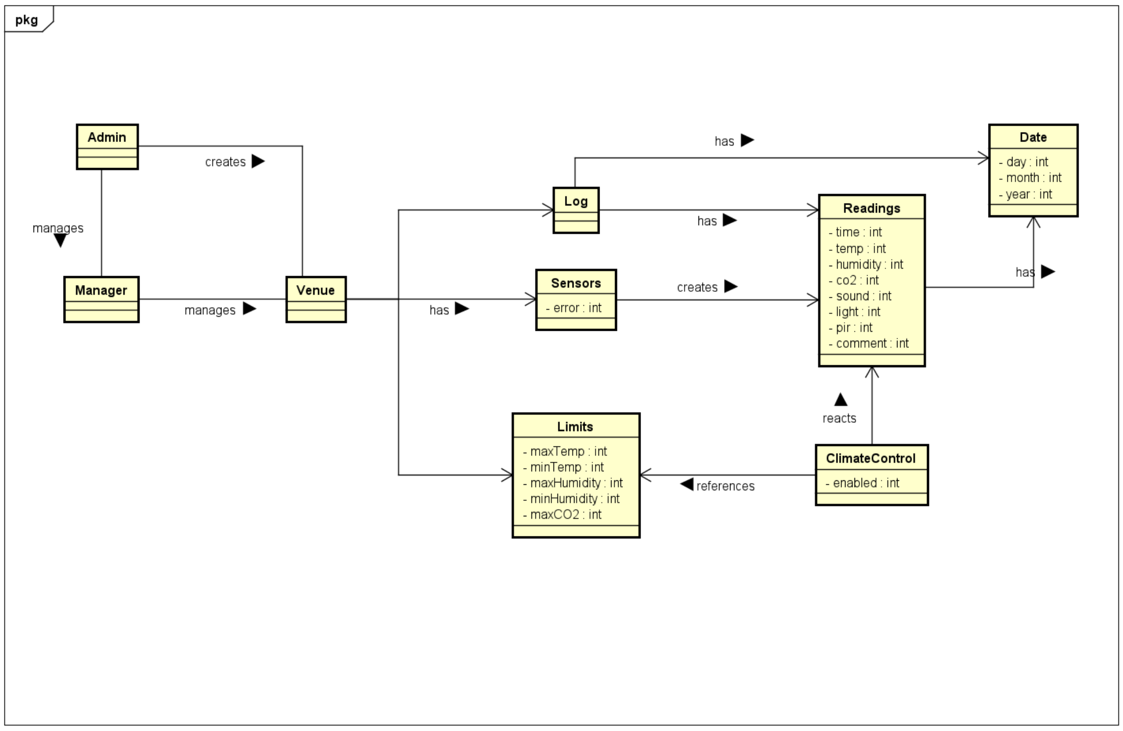
- The system must be accessible on Google Chrome.
- The system must be able to run on freeRTOS for Internet of Things Team, on Linux for the Cloud Team and on Windows 10 and 11 for Front End Team.
- The web app language must be English.

○ **Domain Model**

Rojus

The Domain model is the key part of project analysis, as it serves as an overall visual presentation of the system and the interactions between each entity within it. The goal of it is to guide the development team towards implementing a system that fulfils the expectations set by the product owner. It acts as an interface or common ground for both parties.



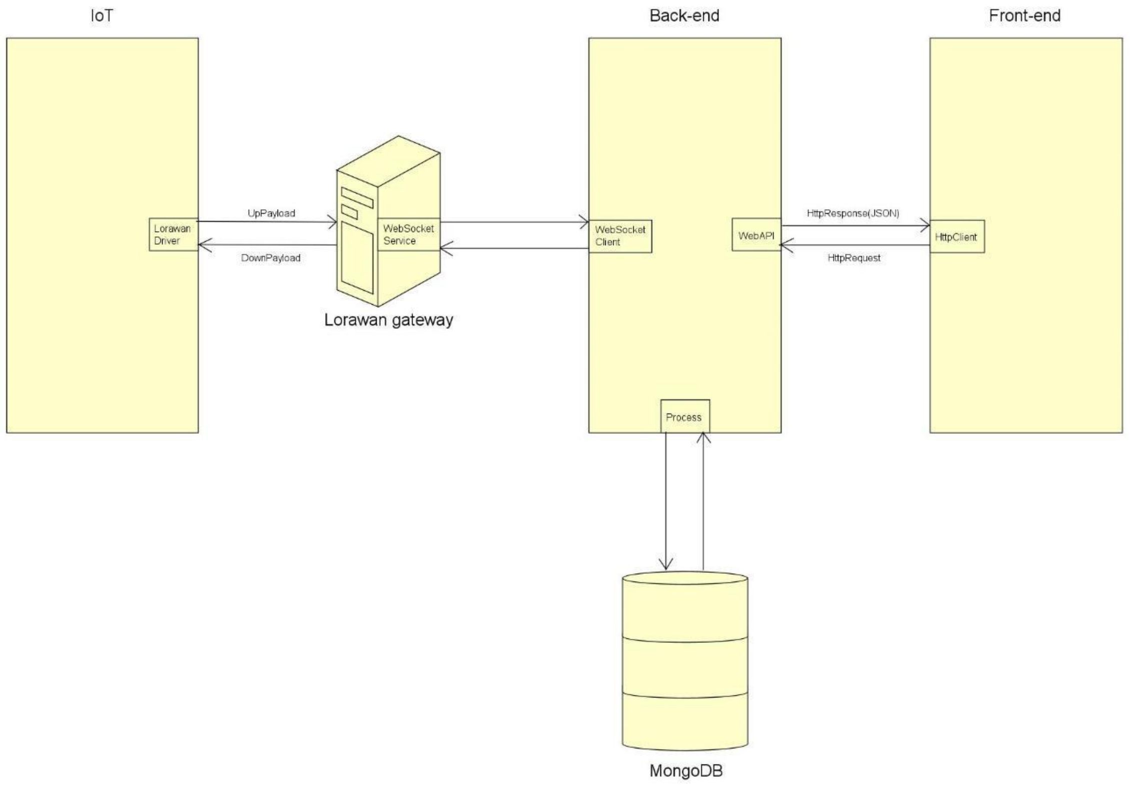


Like stated previously, the solution will have 3 roles. The Administrator will be responsible for managing venues within the system and assigning staff to be responsible for them. The staff assigned take on the role as Managers, who will have access to log and limit management, as well as viewing all readings received from the IoT device. The guests will have access to a website where the most relevant up to date reading data will be displayed.

● Design

○ Architecture

To follow this semester's requirements, the team has decided to split the system into 3 parts, them being IoT, Cloud and Front-End. The IoT and Cloud communicate via the LoRaWAN protocol, while Cloud provides a web API for Front-End to connect to. All parts run separately on different servers.



○ Architectural Qualities

This architecture provides great scalability, as, if need be, infinitely many new functions can be implemented without changing the whole architecture of the system. With all the logic being handled in Cloud, it is easy to maintain the system. As the users do not have a direct connection to the database, the number of security issues have greatly decreased as well.

Despite all the advantages, all parts of the system have to be running in order for the program to achieve the required goals, while efficiency highly depends on network latency between Cloud and Front-End.

○ Front End Design

▪ Architecture

When it comes to front-end architecture, the web application is designed as a single-page application (SPA). This choice was made to provide a seamless and interactive user experience where the user will not have to wait for pages to load fully. Because of this, the UI is divided into reusable components that allow better maintainability of the application. The component-based approach makes it easier to organise the code in the future, and is chosen with the attempt to make it testable.

▪ Technologies

React.js: For the development of the web application, a popular JavaScript framework called React was chosen. It helps with building user interfaces that are interactive. *<<Matas>>*

Tailwind CSS: For styling the web app, the front end team decided to use Tailwind CSS, - a utility first CSS framework. This framework provides a highly customizable and efficient way to build user interfaces. It differs from other CSS frameworks by focusing on utility classes that can be applied directly to HTML elements, allowing for rapid development and easy customization.

Tailwind CSS was chosen as the CSS library for this project due to several advantages:

- Rapid Development: Tailwind CSS offers a vast collection of pre-built components, making rapid styling without the need to create styles from scratch.

- Consistent and Responsive Design: Tailwind CSS provides visually appealing and consistent styling for all components. Additionally, it supports responsive design.
- Optimization: A key selling point of Tailwind CSS is its unique "Purge CSS" feature. During the build process, only the CSS classes that are actually used in the application are included, resulting in a smaller file size. This optimization enhances loading times and overall performance of the application. <<*Daniel*>>

Recharts: To make the implementation of the charts easier, a popular open-source charting library for React called Recharts was used. It provides a comprehensive set of customizable and interactive charts, allowing developers to easily visualise data in their web applications.

Jest: For testing, the team picked the testing framework that is used for unit testing React.js components. The choice was made due to its popularity and the fact that it has been taught in class.

▪ Design Patterns

A design pattern present in the application developed by the front end team is Component Based Architecture. This helps with separation of concerns and modularity. Each component has a certain functionality, and components can be re-used across other components.<<*Matas*>>

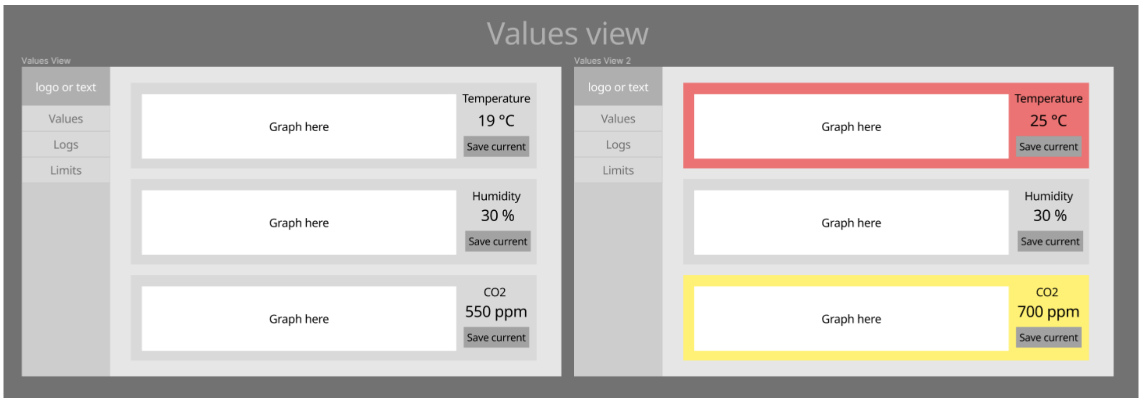
▪ **UI Design Choices**

The chosen UI design for the web app follows a minimalistic and clean approach, aiming for a sleek and modern appearance. The primary color scheme revolves around a navy blue tone, which creates a sense of elegance and sophistication. The navy blue colour is predominantly used in the navigation bar, providing a visually striking element against the overall white background.

To maintain a consistent and cohesive design, a minimalist aesthetic is applied throughout the app, with a focus on simplicity and clarity. The use of white text on the navy blue navigation bar ensures readability and contrast, while black text on a white background is employed for the remaining pages, promoting a clean and easy-to-read interface.

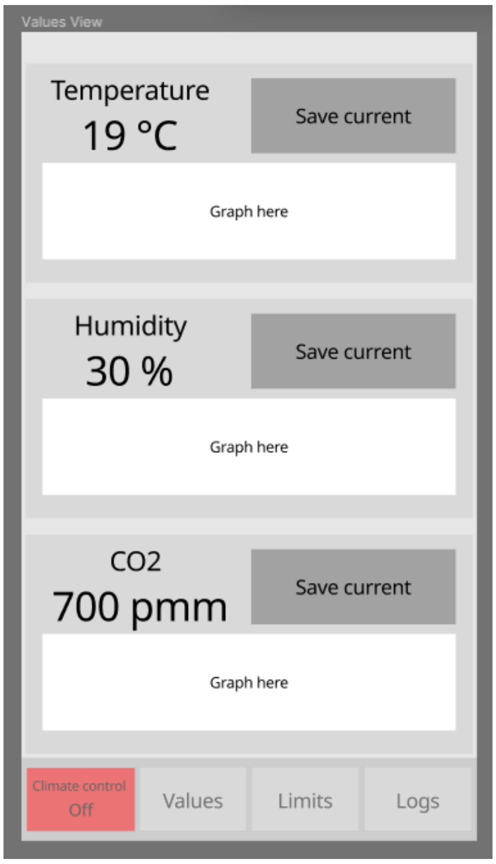
Overall, the UI design choices aim to create a visually appealing, user-friendly, and minimalistic web app experience. The combination of the navy blue color scheme, minimalistic approach, and the utility-first approach of Tailwind CSS contributes to a streamlined and efficient development process while ensuring a visually cohesive and engaging user interface.

While this web app isn't a massive project, it serves an important purpose, which is why it's design was made simple. The addition of a few quality-of-life enhancements was made as well to improve its usability. <<*Selina*>>



Reading View showing the latest information in the venue

In the main view, each measurement is separated for an easy overview of the current situation. To help the user check if the values are between the limits, a decision was made to simplify the process of lighting up the reading red if it is on or above its limit. This makes it a lot easier to manage the venue's environment and make decisions while doing so.



The Readings Graph View is a page user interacts with most.

In the large screen version, all important information is displayed on the screen in a number of graphs. Users can view the current values and a graph of previous values. The Navigation Bar at the top allows simple navigation through the system,

In a small-screen version of the app, not much changes aside from the Navigation Bar moving to the bottom. Later on, as will be seen in implementation, the design changed slightly. *<<Selina>>*

o Cloud Design

▪ Technologies

Lolek

- **SpringBoot** - provides versatile utilities that were useful in creating the application. Additionally it provides other features such as dependency injection, embedded web servers, a custom packaging method with a custom ClassLoader, among many other benefits.
- **Apache Tomcat** - is a web-server that can be embedded into a Java application that provides the system with HTTP capabilities.
- **WebSockets** - are used to communicate with the LoRaWAN Gateway, a WebSocket client was implemented. It handles the connection, receiving and sending data as well as error messages.
- **MongoDB** - is a NoSQL database that stores data as “Documents”, which are stored in BSON format, which is a superset of JSON, with binary encoded and built specifically for speed
- **Docker** - is a container engine that allows for virtual spaces called containers that simulate environments for applications to run in

▪ Design Patterns

Javier

- Facade pattern: Abstract details from an *API* by providing a different one that better fits the current use case, hiding details while reducing complexity, one example of this is **`DataRepository`**, which provides data access capabilities while simultaneously hiding an abstracting the implementation of said data access, such as the fact that we're using *MongoDB* as the database. If the database were to ever change, only the implementation of **`DataRepository`** should change, since the system only ever interacts with data through the **`DataRepository`** facade.

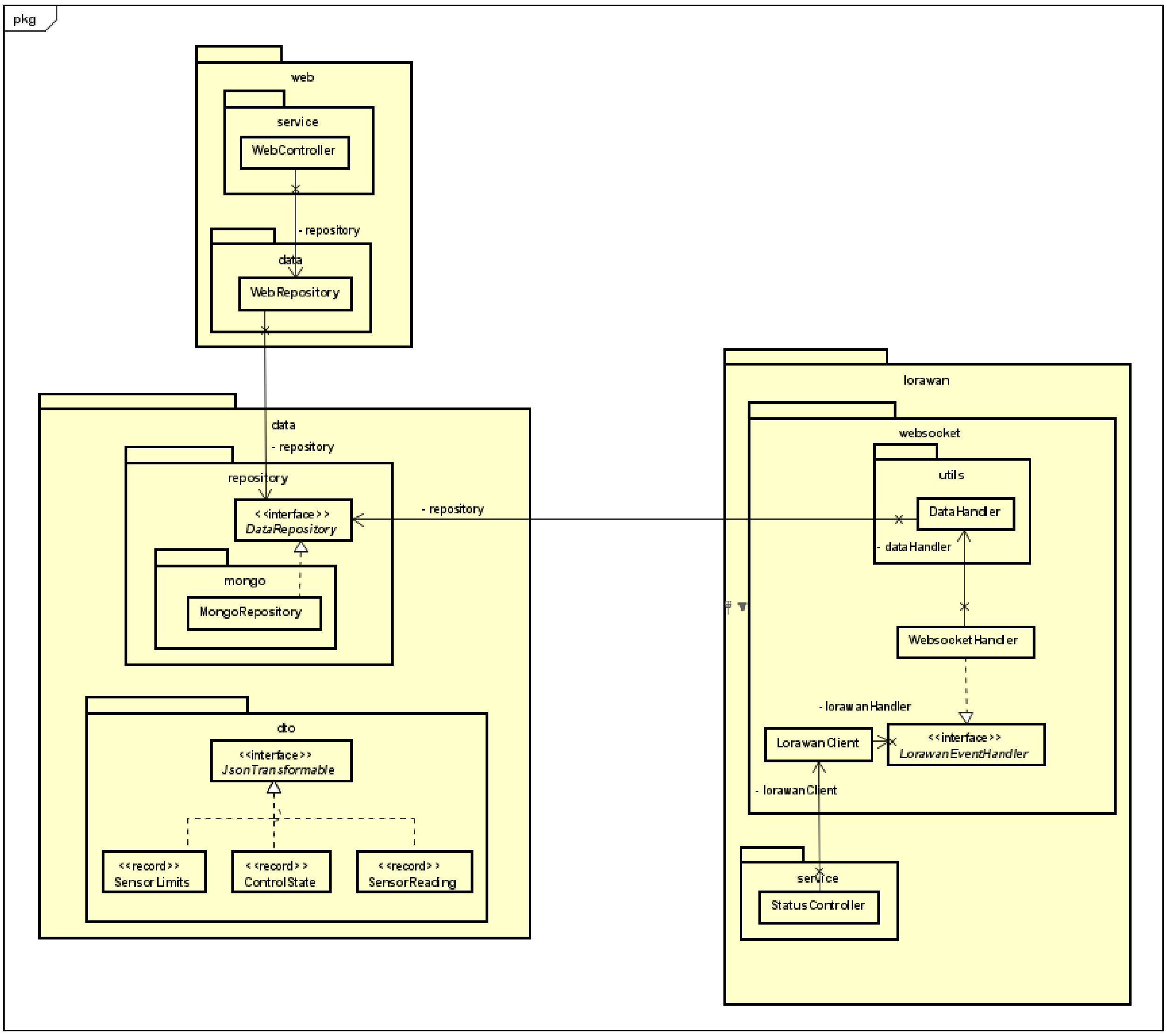
- Adapter pattern: Changes one *API* to fit into another, basically deferring the adapted functionality to the adapter's. For example, **`WebRepository`** adapts **`DataRepository`**, by wrapping it and implementing many of the same methods, but changing the return types such that it natively returns objects in *JSON.*

- Chain of Responsibility: A pattern where each handler decides to process a request, or passes it to the next handler in the chain. An example of this can be found in the LoRaWAN module where the Listener receives a command from the server and decides whether to process it or pass it to the LorawanHandler, who in turn decides to process the commands or pass them on to the DataHandler class.

▪ Class diagram

Lolek

In the following section, a simplified class diagram can be seen to provide an overview of the cloud system and to show the connection between packages.

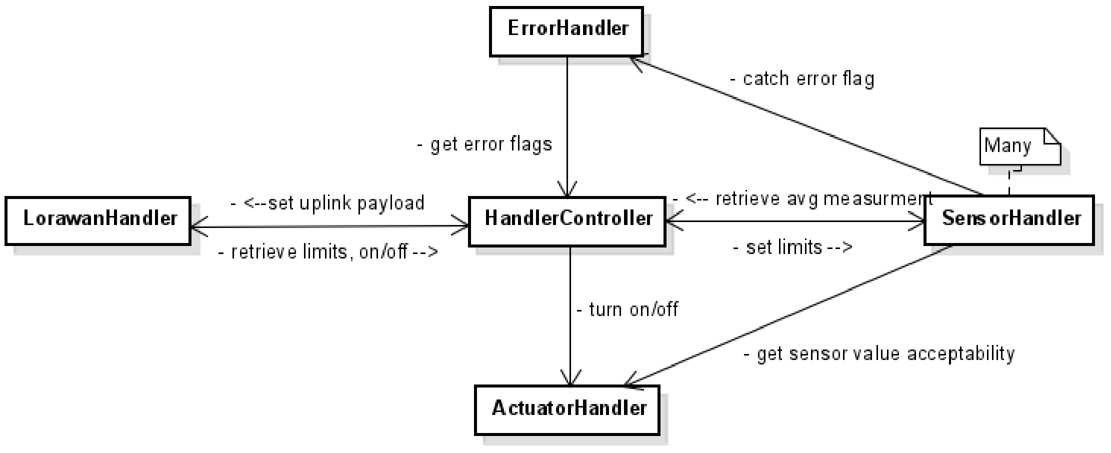


The lorawan package handles the connection to the LoRaWAN Gateway. It takes care of receiving and sending payloads, parsing the objects into correct formats. Then, all the data is sent to the data package to be stored in the database, which is handled by the MongoRepository class. In terms of connection to the frontend, the web package takes care of that via the WebController class, which provides endpoints for the website to use.

○ IoT Design

Nerijus, Mikkel, Adriana

▪ Architecture

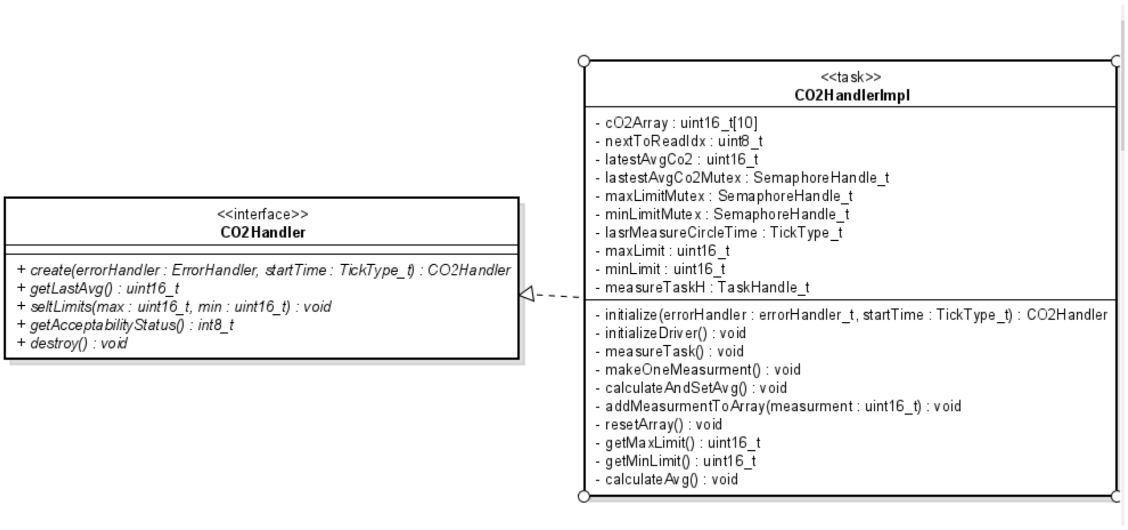


Distributed collaborative control pattern was used to design architecture of the IoT device application, because it was decided that control should be distributed across FreeRTOS tasks. Handler controller has the most control, it creates all other handlers, retrieves latest average sensor measurements, retrieves error flags, forms and sets LoRaWAN uplink payload from sensor measurements and error flags, retrieves downlink payload from LoRaWAN handler, based on which it sets sensor limits and turns on/off environmental control actuators. Actuator handler retrieves current average sensor value acceptability status from sensor handlers and controls climate control actuators for ventilation and air conditioning. The Lorawan handler controls a task that sends and receives payloads from the LoRaWAN gateway. Sensor handlers control sensor measurement tasks.

▪ Technologies

- FreeRTOS was chosen as the real-time operating system for the IoT device. Real time OS guarantees to meet deadlines of task execution, which makes it valuable for the venue environmental control system, since venue ventilation and air conditioning control must react in a timely manner when sensors detect values exceeding limits set for temperature, co2 and humidity. The FreeRTOS is trusted world-wide and is able to support a wide range of devices. It is small and energy efficient, which makes it fitting for low memory embedded devices even if they are powered by batteries. There is a wide range of information regarding the use and development using FreeRTOS.
- LoRaWAN was used as the networking protocol. It is low-power and it provides long-range bidirectional communication between end devices and network gateways.

▪ Design Patterns


<<interface>>
CO2Handler
+ create(errorHandler : ErrorHandler, startTime : TickType_t) : CO2Handler
+ getLastAvg() : uint16_t
+ seltLimits(max : uint16_t, min : uint16_t) : void
+ getAcceptabilityStatus() : int8_t
+ destroy() : void
<<task>>
CO2HandlerImpl
- cO2Array : uint16_t[10]
- nextToReadIdx : uint8_t
- latestAvgCo2 : uint16_t
- lastestAvgCo2Mutex : SemaphoreHandle_t
- maxLimitMutex : SemaphoreHandle_t
- minLimitMutex : SemaphoreHandle_t
- lasrMeasureCircleTime : TickType_t
- maxLimit : uint16_t
- minLimit : uint16_t
- measureTaskH : TaskHandle_t
- initialize(errorHandler : errorHandler_t, startTime : TickType_t) : CO2Handler
- initializeDriver() : void
- measureTask() : void
- makeOneMeasurment() : void
- calculateAndSetAvg() : void
- addMeasurmentToArray(measurment : uint16_t) : void
- resetArray() : void
- getMaxLimit() : uint16_t
- getMinLimit() : uint16_t
- calculateAvg() : void

Facade Design Pattern was used as a design pattern for the IoT part of a system. It is a well-known design pattern in the industry and is popular because of its versatility. It was used in order to have a front-facing interface masking more complex underlying code. It improves readability and usability of the software code and it provides a context-specific interface for generic functionality. Abstract data types are used to implement this design pattern for IoT device's environmental control application as can be seen in the given **CO2Handler**.

Mutex concurrency pattern was used as a primary pattern to handle mutual exclusion of resources accessed by several FreeRTOS tasks. In the given example of **CO2Handler** it can be seen that mutexes were used to mutually exclude latest average measurement, minimum and maximum limits. For example, **latestAvgMeasurmentMutex** is used when last sensor average sensor measurement is accessed by CO2 measurement task, controller handler task, and actuator handler task.

Event queue design pattern was used to collect errors from FreeRTOS measurement tasks of sensor handlers. Since the IoT device cannot send error notifications as they occur because of the limitations of LoRaWAN, errors need to be queued. Throughout sensor measurement task execution errors are being collected into a queue and are being released from it into an uplink payload whenever the IoT device is ready to send the uplink message again.

- **UI Design Choices**

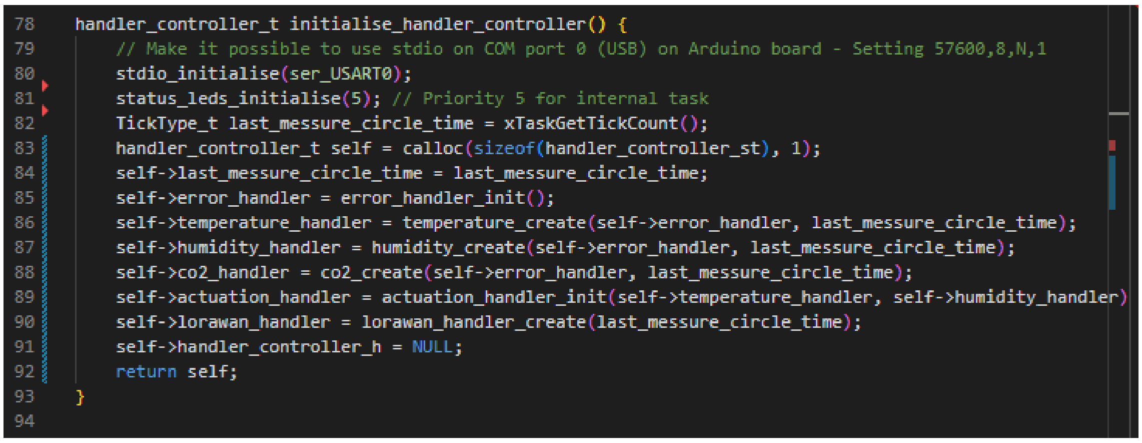
The screen on the IoT device cycles through 6 bit numbers corresponding to each of the sensors, which indicates that there are issues with the given sensors. As there are only 4 digits on the available display, and the potential maximum supported sensor count was 6.

● Implementation

Adriana, Nerijus

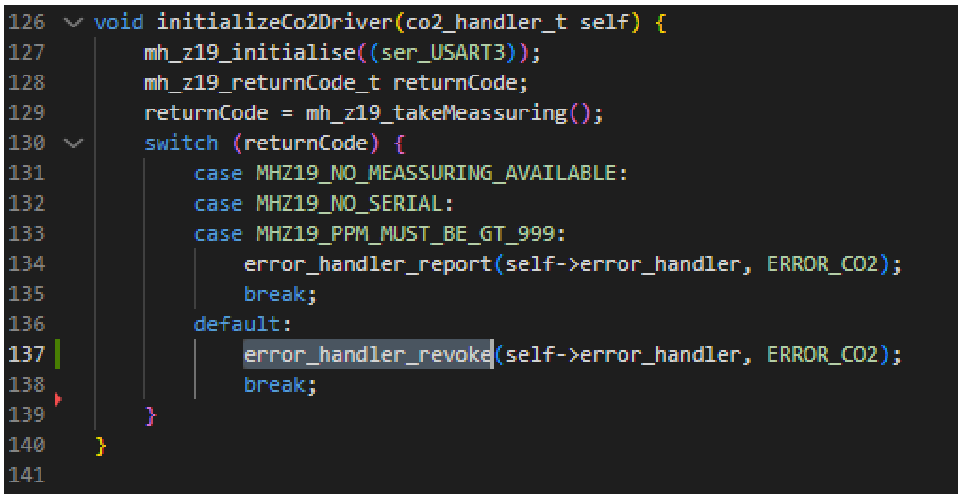
○ IoT Implementation

HandlerController creates all sensor handlers, **LorawanHandler**, **ActuatorHandler**, **ErrorHandler** and stores them as member variables.



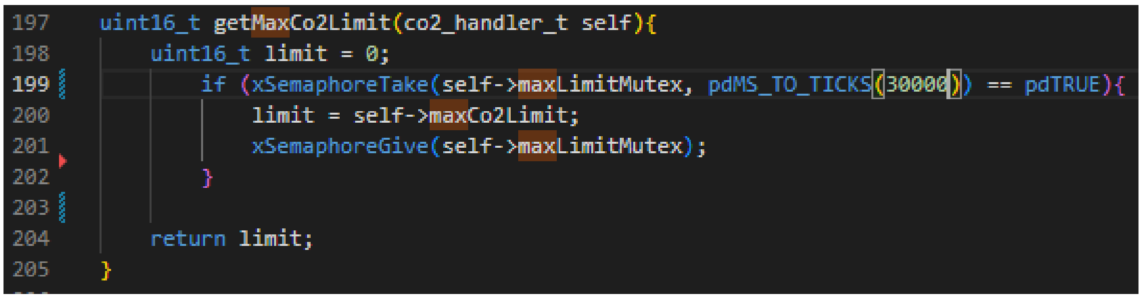
```c
handler_controller_t initialise_handler_controller() {
    // Make it possible to use stdio on COM port 0 (USB) on Arduino board - Setting 57600,8,N,1
    stdio_initialise(ser_USART0);
    status_leds_initialise(5); // Priority 5 for internal task
    TickType_t last_messure_circle_time = xTaskGetTickCount();
    handler_controller_t self = calloc(sizeof(handler_controller_st), 1);
    self->last_messure_circle_time = last_messure_circle_time;
    self->error_handler = error_handler_init();
    self->temperature_handler = temperature_create(self->error_handler, last_messure_circle_time);
    self->humidity_handler = humidity_create(self->error_handler, last_messure_circle_time);
    self->co2_handler = co2_create(self->error_handler, last_messure_circle_time);
    self->actuation_handler = actuation_handler_init(self->temperature_handler, self->humidity_handler)
    self->lorawan_handler = lorawan_handler_create(last_messure_circle_time);
    self->handler_controller_h = NULL;
    return self;
}
```

During sensor handler initialization sensor drivers need to be initialised as well. If any errors arise during driver initialization, an error flag is passed to the error handler, where it is added to the queue.



```c
void initializeCo2Driver(co2_handler_t self) {
    mh_z19_initialise((ser_USART3));
    mh_z19_returnCode_t returnCode;
    returnCode = mh_z19_takeMeassuring();
    switch (returnCode) {
        case MHZ19_NO_MEASSURING_AVAILABLE:
        case MHZ19_NO_SERIAL:
        case MHZ19_PPM_MUST_BE_GT_999:
            error_handler_report(self->error_handler, ERROR_CO2);
            break;
        default:
            error_handler_revoke(self->error_handler, ERROR_CO2);
            break;
    }
}
```

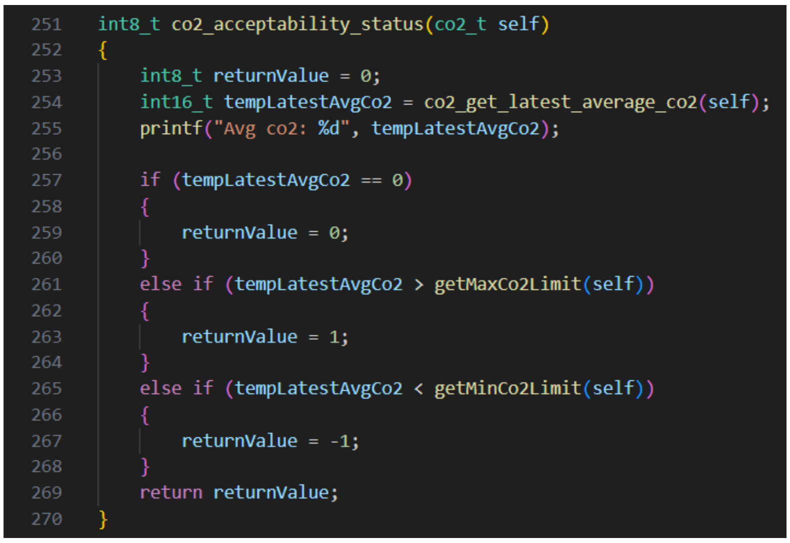
As it is mentioned before, mutex is used to mutually exclude sensor limits, because the limits are being accessed by sensor measure tasks, environment control actuator handler task and handler controller task.



```
uint16_t getMaxCo2Limit(co2_handler_t self){
    uint16_t limit = 0;
        if (xSemaphoreTake(self->maxLimitMutex, pdMS_TO_TICKS(30000)) == pdTRUE){
            limit = self->maxCo2Limit;
            xSemaphoreGive(self->maxLimitMutex);
        }

    return limit;
}
```

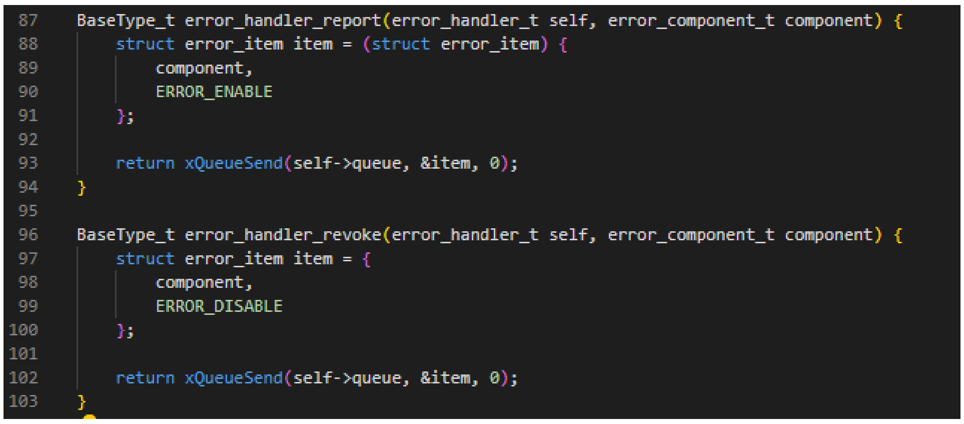
Sensor acceptability status is accessed by the environmental control actuator handler task. Sensor acceptability status is a value of 1, 0, or -1 and it shows if the last calculated average measurement exceeds the limits set by the handler controller task after parsing a downlink message.



```
int8_t co2_acceptability_status(co2_t self)
{
    int8_t returnValue = 0;
    int16_t tempLatestAvgCo2 = co2_get_latest_average_co2(self);
    printf("Avg co2: %d", tempLatestAvgCo2);

    if (tempLatestAvgCo2 == 0)
    {
        returnValue = 0;
    }
    else if (tempLatestAvgCo2 > getMaxCo2Limit(self))
    {
        returnValue = 1;
    }
    else if (tempLatestAvgCo2 < getMinCo2Limit(self))
    {
        returnValue = -1;
    }
    return returnValue;
}
```

Every sensor handler reports errors related to the sensor failures to the error handler. In the error handler’s report function, the reported error flag is sent to the error queue. Error flag is removed from the error queue when the sensor successfully resumes its work and error handlers revoke function is called.



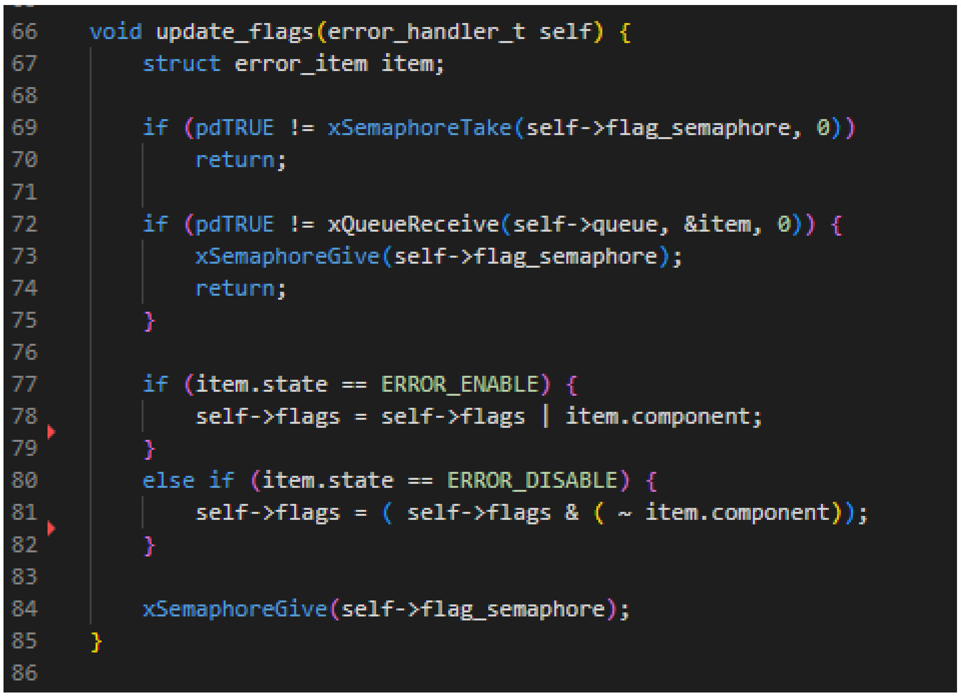
```c
BaseType_t error_handler_report(error_handler_t self, error_component_t component) {
    struct error_item item = (struct error_item) {
        component,
        ERROR_ENABLE
    };

    return xQueueSend(self->queue, &item, 0);
}

BaseType_t error_handler_revoke(error_handler_t self, error_component_t component) {
    struct error_item item = {
        component,
        ERROR_DISABLE
    };

    return xQueueSend(self->queue, &item, 0);
}
```

Every 15 seconds the error handler executes a task that updates the error flag byte, which later is being sent as a part of a LoRaWAN payload.



```c
void update_flags(error_handler_t self) {
    struct error_item item;

    if (pdTRUE != xSemaphoreTake(self->flag_semaphore, 0))
        return;

    if (pdTRUE != xQueueReceive(self->queue, &item, 0)) {
        xSemaphoreGive(self->flag_semaphore);
        return;
    }

    if (item.state == ERROR_ENABLE) {
        self->flags = self->flags | item.component;
    }
    else if (item.state == ERROR_DISABLE) {
        self->flags = ( self->flags & ( ~ item.component));
    }

    xSemaphoreGive(self->flag_semaphore);
}
```

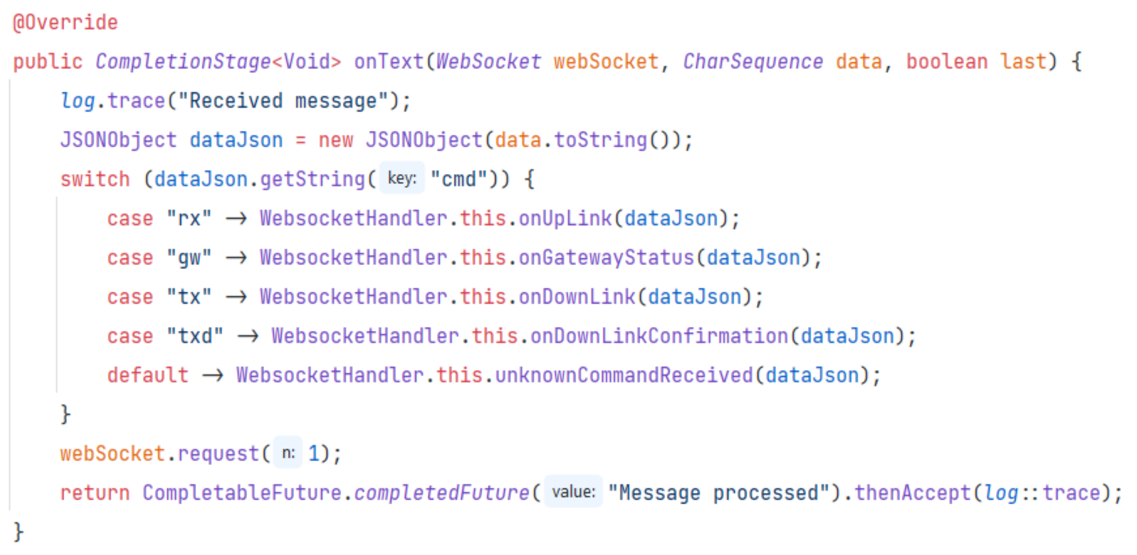
○ Cloud Implementation

Javier

The Cloud infrastructure is the backbone connecting the frontend web-app to the data measured by the *IoT* device. In order to achieve this, multiple microservices sharing a persistence layer ensure the data flow through the system.

The first of these microservices is the *LoRaWAN* connection, a module that will open a WebSocket connection with the server and use it to communicate back and forth with the IoT, receiving data and sending limits back to the device.

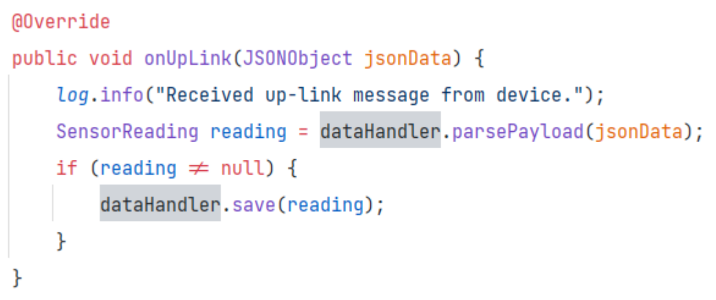
The websocket listeners will read out the command from the *LoRaWAN* socket and delegate into specific methods to handle each kind of message, as can be seen in the following picture



```
@Override
public CompletionStage<Void> onText(WebSocket webSocket, CharSequence data, boolean last) {
    log.trace("Received message");
    JSONObject dataJson = new JSONObject(data.toString());
    switch (dataJson.getString( key: "cmd")) {
        case "rx" → WebsocketHandler.this.onUpLink(dataJson);
        case "gw" → WebsocketHandler.this.onGatewayStatus(dataJson);
        case "tx" → WebsocketHandler.this.onDownLink(dataJson);
        case "txd" → WebsocketHandler.this.onDownLinkConfirmation(dataJson);
        default → WebsocketHandler.this.unknownCommandReceived(dataJson);
    }
    webSocket.request( n: 1);
    return CompletableFuture.completedFuture( value: "Message processed").thenAccept(log::trace);
}
```

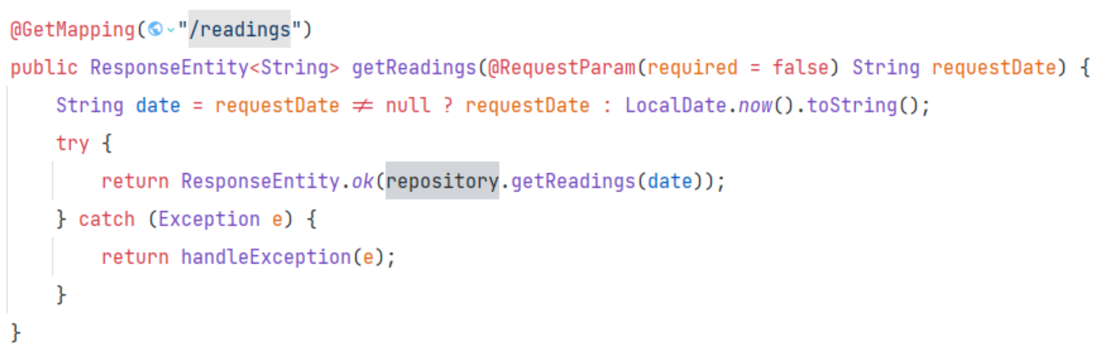
The received data is propagated to the command and acted upon, for example, when receiving an uplink message from the device, the message will have a command “**rx**”, and the corresponding method will call on the **DataHandler** to parse it, and then save the result to the database. This is just an example of how *OOP* principles are implemented into the codebase.

Further examples can be found in the use of design patterns as the chain of responsibility, as illustrated also by the former example, where the **WebsocketHandler.Listener** delegates processing text commands to the **WebSocketHandler**, and that in turn delegates some methods to the **DataHandler**. Another pattern used is the facade pattern in the case of **DataRepository**, where all data operations are defined and then implemented in **MongoRepository**. This removes a lot of the complexity for data access and also abstracts the decision to persist data with *MongoDB* from the system, meaning the database could be swapped out at any time for a different implementation of the **DataRepository**. In fact, this is exactly what was done in order to replace it with a fake repository for testing.



```
@Override
public void onUpLink(JSONObject jsonData) {
    log.info("Received up-link message from device.");
    SensorReading reading = dataHandler.parsePayload(jsonData);
    if (reading ≠ null) {
        dataHandler.save(reading);
    }
}
```

The other service that the system relies on defines a *REST-API* capable of serving data over *HTTP*-requests to the web client. This *API* is defined using the Spring Web framework



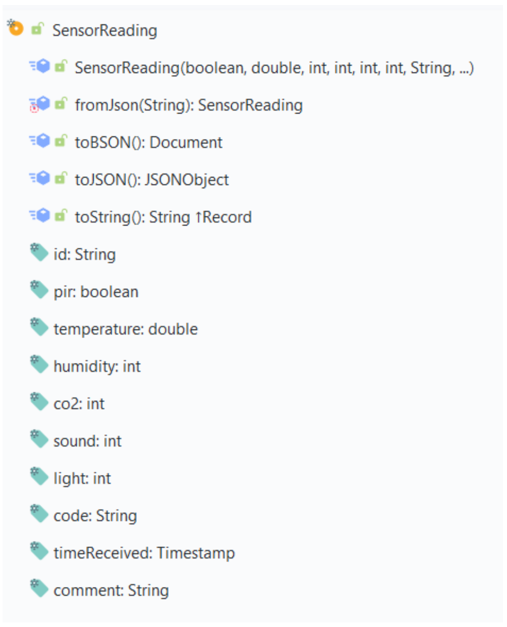
```java
@GetMapping("/readings")
public ResponseEntity<String> getReadings(@RequestParam(required = false) String requestDate) {
    String date = requestDate ≠ null ? requestDate : LocalDate.now().toString();
    try {
        return ResponseEntity.ok(repository.getReadings(date));
    } catch (Exception e) {
        return handleException(e);
    }
}
```

Different mappings are set, alongside *HTTP*-method verbs such as *GET* or *PUT*, defining endpoints and their output. This contract was documented into easy to digest, natural language documentation for ease of communication with the other teams.

The *API* also implements multiple patterns, such as the **`WebRepository`** being an adapter class for **`DataRepository`**, which allows for the Repository to return ***JSON*** in the form of **`String`** rather than the data objects.

It should now be evident that most of the application relies on the **data module**, which is in charge of providing data access capabilities to the services that depend on it. The microservices don't communicate with each other, but share their data so that the state is constantly synced between all services.

As far as functionality, the data module relies heavily on the Data repository to perform operations in the database, and the data transfer objects (*DTO*) which are implemented as records (immutable data classes with automatic **`constructors`** / **`getters`** / **`equals`** / **`toString`** methods) that can transform their state to *JSON* or *BSON* in order to easily convert to and from the Database format to the WebAPI format while existing as regular objects within the system where they can be passed around between methods and conveniently accessed.



```
SensorReading
    SensorReading(boolean, double, int, int, int, int, String, ...)
    fromJson(String): SensorReading
    toBSON(): Document
    toJSON(): JSONObject
    toString(): String ↑Record
    id: String
    pir: boolean
    temperature: double
    humidity: int
    co2: int
    sound: int
    light: int
    code: String
    timeReceived: Timestamp
    comment: String
```

Additionally, these services have been set up so that they can be deployed into Docker containers. For this, a Dockerfile defines a Java environment suitable for the application and then places the packaged software in it, where it is run following a short that fetches the secrets from the environmental variables of the container.

The system has taken a lax approach with security, but at least secrets are kept in a separate file that needs to be created in dev (and setup as environmental variables for containers). This files are retrieved from a .properties file by using an external library

○ Web Implementation

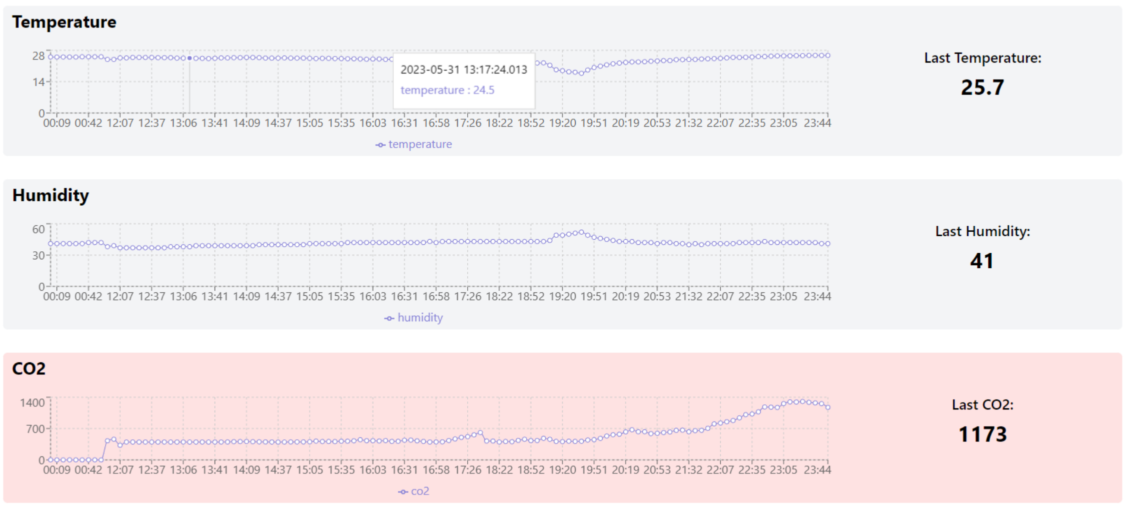
Daniel & Selina

After technologies have been decided and the wireframes have been created, they are implemented in React.js. during each of the sprints. In the following subsections, an overview of what goes on behind the front end part of the system can be found.

In the front-end implementation of the system one of the main components is ReadingsComponent.js. This component provides functionality to analyse current values and compare them with historical readings through a graphical representation.

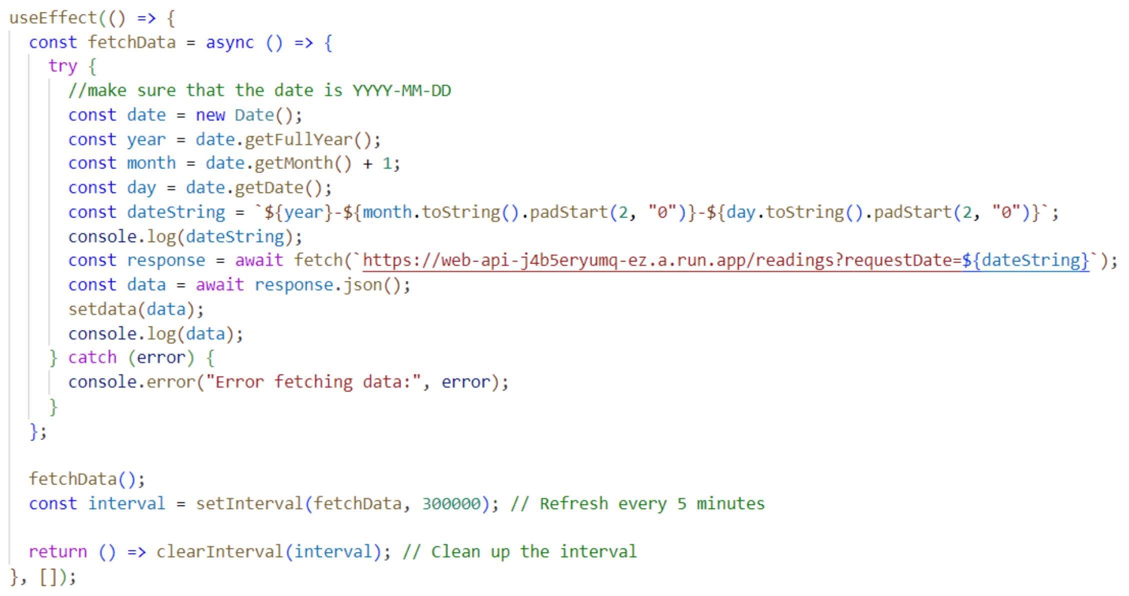
ReadingsComponent.js

This is where you can analyse the current values and compare them in a graphic with all the readings from the current day, as it can be seen on the image below:



The CO2 graph readings are specifically highlighted in red to indicate when they reach a predefined limit. This visual cue alerts the user to potential issues. Achieving this result involves three crucial factors: data fetching, graph rendering, and handling limits.

Data Fetching

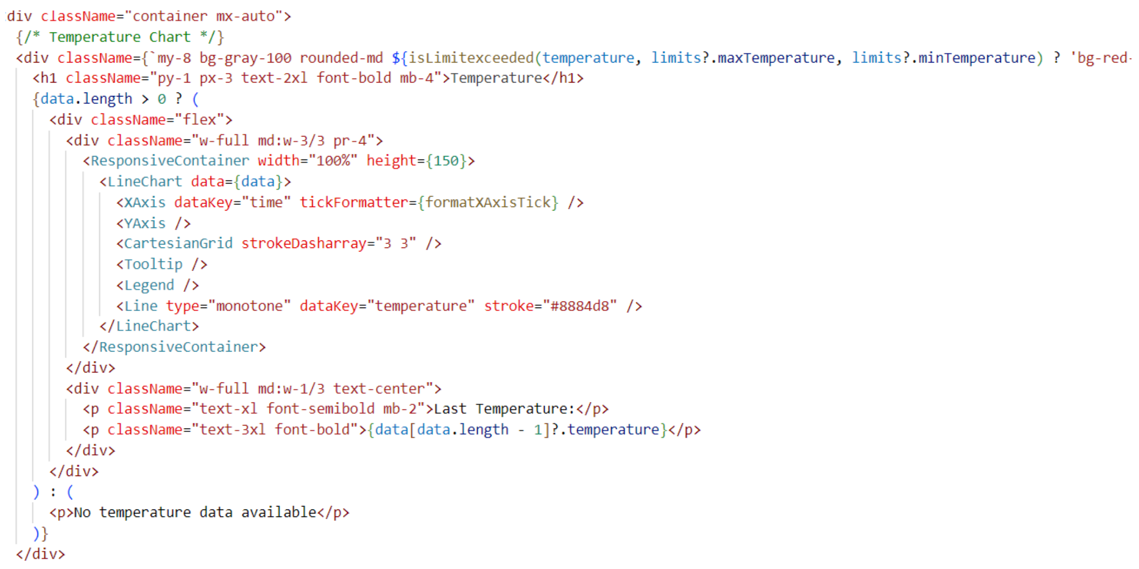


```
useEffect(() => {
  const fetchData = async () => {
    try {
      //make sure that the date is YYYY-MM-DD
      const date = new Date();
      const year = date.getFullYear();
      const month = date.getMonth() + 1;
      const day = date.getDate();
      const dateString = `${year}-${month.toString().padStart(2, "0")}-${day.toString().padStart(2, "0")}`;
      console.log(dateString);
      const response = await fetch(`https://web-api-j4b5eryumq-ez.a.run.app/readings?requestDate=${dateString}`);
      const data = await response.json();
      setdata(data);
      console.log(data);
    } catch (error) {
      console.error("Error fetching data:", error);
    }
  };

  fetchData();
  const interval = setInterval(fetchData, 300000); // Refresh every 5 minutes

  return () => clearInterval(interval); // Clean up the interval
}, []);
```

To fetch the necessary data, the component utilises an API. The current day is obtained and converted into a format that the API understands. Once the data is retrieved from the API, it is used to generate the graphs.



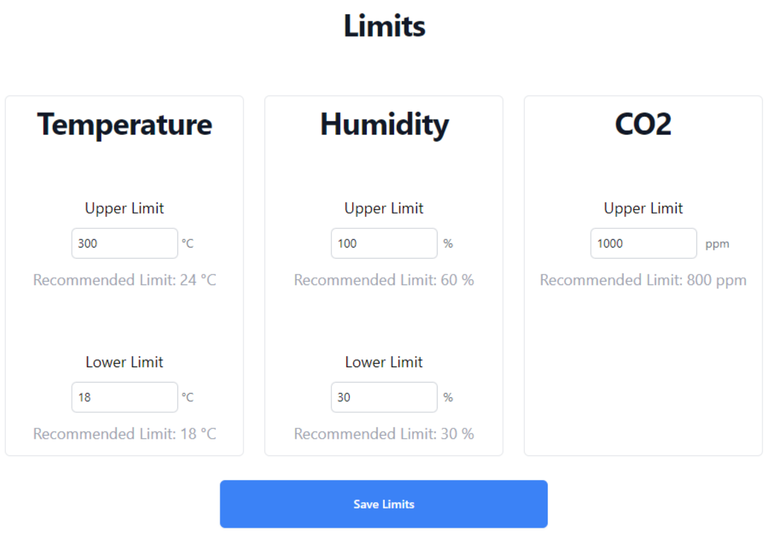
```
div className="container mx-auto">
 {/* Temperature Chart */}
 <div className={`my-8 bg-gray-100 rounded-md ${isLimitexceeded(temperature, limits?.maxTemperature, limits?.minTemperature) ? 'bg-red-
   <h1 className="py-1 px-3 text-2xl font-bold mb-4">Temperature</h1>
   {data.length > 0 ? (
     <div className="flex">
       <div className="w-full md:w-3/3 pr-4">
         <ResponsiveContainer width="100%" height={150}>
           <LineChart data={data}>
             <XAxis dataKey="time" tickFormatter={formatXAxisTick} />
             <YAxis />
             <CartesianGrid strokeDasharray="3 3" />
             <Tooltip />
             <Legend />
             <Line type="monotone" dataKey="temperature" stroke="#8884d8" />
           </LineChart>
         </ResponsiveContainer>
       </div>
       <div className="w-full md:w-1/3 text-center">
         <p className="text-xl font-semibold mb-2">Last Temperature:</p>
         <p className="text-3xl font-bold">{data[data.length - 1]?.temperature}</p>
       </div>
     </div>
   ) : (
     <p>No temperature data available</p>
   )}
 </div>
```

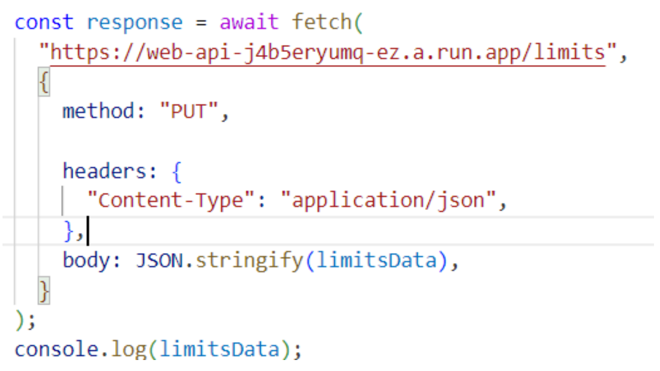
To handle the limits and their impact on the UI, the component dynamically changes the colour of certain div elements. When a limit is reached, the corresponding div turns red, drawing the user's attention to the issue.

For the graphical representation, the ReCharts framework is employed. This framework simplifies the process of creating graphs by providing features such as automatic scaling and data visualisation. By feeding information for the X and Y axes, the framework handles the graph generation.

LimitsComponent.js

The Limits component offers a user-friendly view where users can effortlessly monitor the current limits and conveniently set new ones according to their preferences. This intuitive interface allows users to effectively manage the system's thresholds, as showcased in the image below:





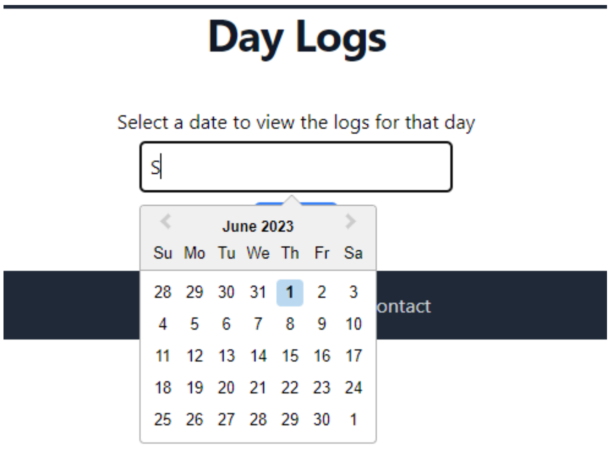
```
const response = await fetch(
  "https://web-api-j4b5eryumq-ez.a.run.app/limits",
  {
    method: "PUT",

    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(limitsData),
  }
);
console.log(limitsData);
```

To implement this functionality, the component leverages a PUT request to the API. This request enables seamless communication between the front-end and back-end, ensuring that updated limit settings are accurately reflected in the system

LogsComponent.js

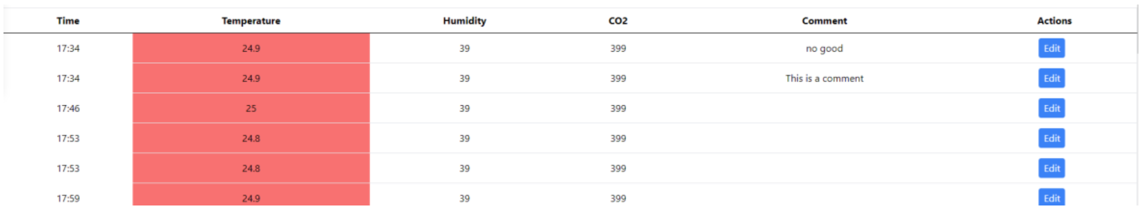
The Logs component encompasses a wide range of functionality. Its journey begins by prompting the user to select their desired date using a simple yet effective date picker interface.



Day Logs

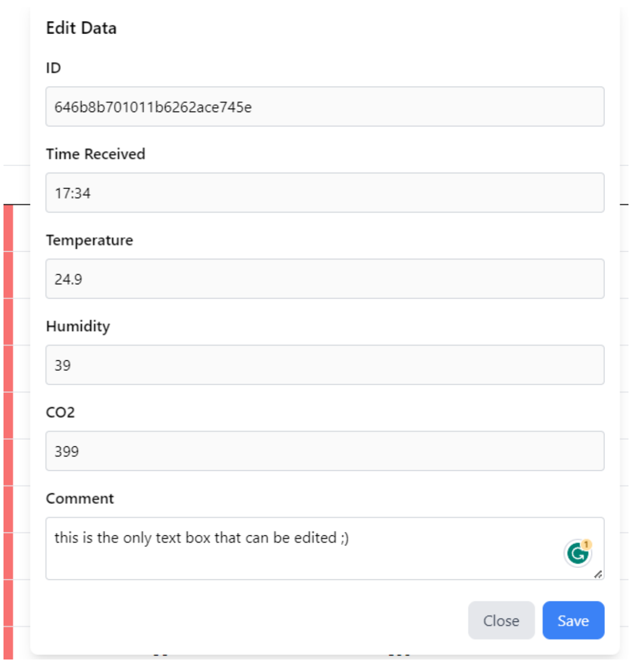
Select a date to view the logs for that day

Upon submitting the chosen date, an extensive table is generated, showcasing all the readings recorded on that specific day. This table is generated using a similar approach as the Readings Component, with the key distinction lying in the presentation of the data. Notably, each row in the table features an "edit" button, allowing users to effortlessly modify and enhance the associated readings.



Time	Temperature	Humidity	CO2	Comment	Actions
17:34	24.9	39	399	no good	Edit
17:34	24.9	39	399	This is a comment	Edit
17:46	25	39	399		Edit
17:53	24.8	39	399		Edit
17:53	24.8	39	399		Edit
17:59	24.9	39	399		Edit

The edit functionality is facilitated by a secondary component integrated within the Logs Component. This auxiliary component efficiently retrieves all the relevant information for the selected row by utilising the unique "id" assigned to each reading. As demonstrated in the image below, the component offers a popup interface where users can conveniently update or provide additional comments for the selected reading.



Edit Data

ID: 646b8b701011b6262ace745e

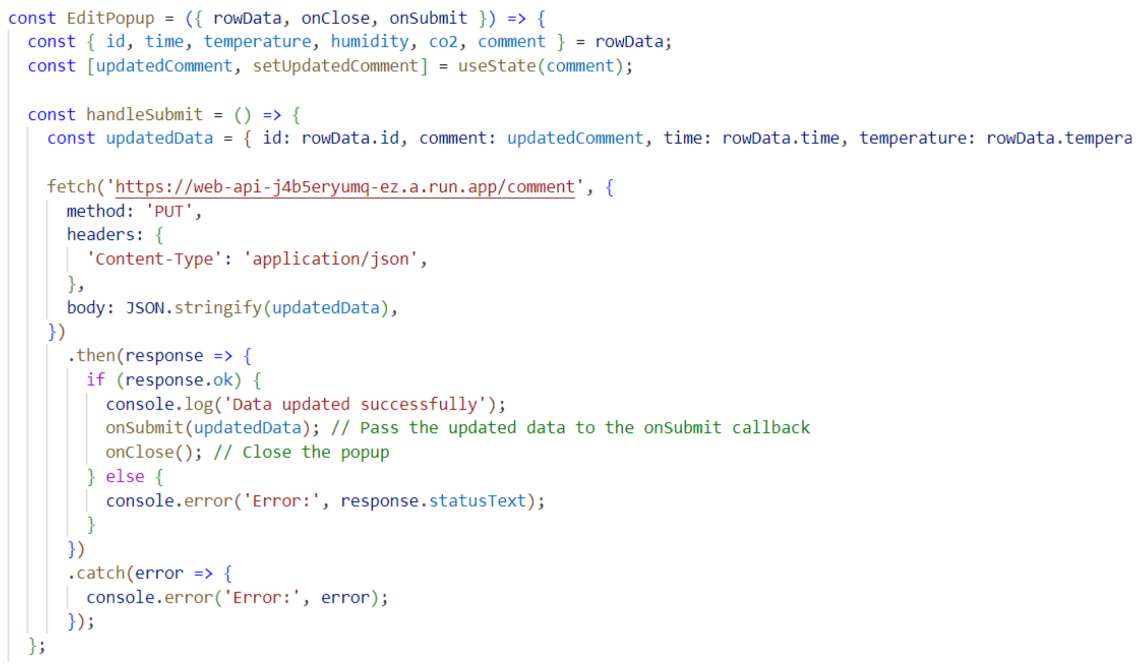
Time Received: 17:34

Temperature: 24.9

Humidity: 39

CO2: 399

Comment: this is the only text box that can be edited ;)



```
const EditPopup = ({ rowData, onClose, onSubmit }) => {
  const { id, time, temperature, humidity, co2, comment } = rowData;
  const [updatedComment, setUpdatedComment] = useState(comment);

  const handleSubmit = () => {
    const updatedData = { id: rowData.id, comment: updatedComment, time: rowData.time, temperature: rowData.tempera

    fetch('https://web-api-j4b5eryumq-ez.a.run.app/comment', {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(updatedData),
    })
      .then(response => {
        if (response.ok) {
          console.log('Data updated successfully');
          onSubmit(updatedData); // Pass the updated data to the onSubmit callback
          onClose(); // Close the popup
        } else {
          console.error('Error:', response.statusText);
        }
      })
      .catch(error => {
        console.error('Error:', error);
      });
  };
```

.

The seamless integration of the edit functionality is made possible through the utilisation of a PUT method. This method efficiently transmits the row's corresponding id along with the user's modified or new comments, ensuring accurate and prompt updating of the system's records.

● Continuous Integration & Development

○ Web CI/CD

Selina

- **Git, Branching & Merging**

The team has adopted a branching strategy, using standard branches such as the main/master branch, feature branches, and release branches. This allows for organised development and version control.

Commit messages are structured and formal to make them meaningful and useful. They follow a specific format with a first capital letter, period at the end, and present tense verb at the beginning.

Merging criteria have been established to ensure that features are fully implemented, code is reformatted and cleaned up, and commit messages provide clear progress.

- **DevOps Tools Used**

  - Git is used as the version control system, providing features like branching, merging, and commit history.
  - GitHub Workflow and GitHub Actions are used to create and run a continuous integration (CI) pipeline, ensuring automated testing and continuous integration.

- **Integration of DevOps into Workflow**

The team follows a general workflow that encompasses planning, coding, testing, merging, and deploying in a cycle.

Planning involves defining requirements and design, coding follows best practices and standards, testing is conducted to meet specifications, code is merged through pull requests, and deployment is achieved using a continuous deployment pipeline.

GitHub workflows are used to support and automate these steps, ensuring the code is always buildable and tests are passing.

- **Impact of DevOps Tools/Methods**

The DevOps tools and methods have likely improved the overall development process and collaboration within the team.

Using branches, commit message standards, and merging criteria help maintain code quality and track progress effectively.

GitHub Workflow and Actions enable continuous integration, automated testing, and deployment, which saves time and ensures that code changes are thoroughly validated.

The specific impact and effectiveness of these DevOps tools/methods would depend on the team's experiences and the project's requirements.

o IoT CI/CD

Nerijus, Mikkel

- **General workflow decisions**

Feature branches are only merged into main after thorough testing, and a review from one or multiple other team members, to prevent non-working code from entering the main branch.

Having only one IoT device, it was concluded that flashing and accessing it via cable is inefficient. For that reason the device was set up so it could be flashed and monitored remotely without the need to connect the device to different computers with physical usb cables. It made it more efficient and easy to use the device.

- **Git and version control**

During the development of the IoT system the development team relied on the git version control system, as well as the free to use code forge Github, which also provides free CI runners to public repositories.
In order to avoid commiting unnecessary files or sensitive information the ‘.gitignore’ file, a feature that has git ignore untracked files, which are specified in the previously mentioned ‘.gitignore’ config file.
The branching strategy used is similar to Github Flow, as it is simple and easy to implement for a small team.

- **Branching strategy**

  - Main branch - The active development branch. Larger changes, such as the implementation of a new feature, should be first implemented on a separate ‘Feature branch’. The changes can then be merged using a pull request, after all unit tests have succeeded.

- Feature branches - Each feature is expected to have its own development branch. Each feature branch is named after its given jira ticket, as such: “feature/(JIRA Ticket ID)”
- Bugfix branches - Branches created for bug fixes or other minor patches have no specific naming convention.
- Releases contain only practically usable code. They are managed using git tags and named with semantic versioning.

▪ **GitHub Workflows**

The CI pipelines used are composed of 3 separate github workflows, one running the test suite, one generating and publishing an api-reference and the last being a simple CD workflow. All of the workflows support manual activation.

The test workflow first runs the ‘scripts/init_lora.sh’ script. This generates the header LoRaWAN.h, as it is necessary to build the code. CMake is then used to configure the build environment and build the tests, the resulting binary of which is then executed. After the tests have been run a make target named ‘coverage’ is called that then uses both Gcov and Lcov in order to generate a static html page containing a coverage report. The report is then exported as an artefact and can be downloaded and viewed. The workflow is run on every push as well as any pull request against main.

The api-reference workflow runs on every push to the main branch, so the generated documentation is kept up to date. It runs a specialised github action of which sets up doxygen and then generates based on a given ‘Doxyfile’, documentation that is put into a directory named ‘html’. This directory is then given to another action alongside a github token which then publishes the generated documentation to github-pages.

The Platformio CD workflow first sets 3 environment variables, all of which are stored as repository secrets. The first two of the environment variables are the tokens needed by the LoRaWAN driver i.e the AppEUI and the AppKey, these are used by the previously mentioned script ‘scripts/init_lora.sh’ to generate the header file ‘LoRaWAN.h’. The keys are defined as constants ready to be used by the LoRaWANHandler. The workflow then sets up python alongside PlatformIO, the ‘remote’ api of which is then used to build and flash the iot device, so long as it is connected to a ‘remote agent’.

- **Development Tools**

  - PlatformIO is a cross platform build system and IDE for embedded system development. It supports easy remote flashing of hardware devices, which is useful for remote development as well as continuous deployment.

  - CMake - CMake is an open-source cross-platform tool designed to configure, test and package software builds. It is used to set up the Iot systems test environment in addition to coverage tests.

  - Google Test (gtest) is a unit testing framework for C and C++. It is used alongside Fake Function Framework (fff) to implement all of the unit tests.

- Fake Function Framework (fff) is a framework for easily producing fake c functions, which can be used to produce various types of test doubles.

- Gcov is a coverage testing tool, which is used to measure the quality of tests. It accomplishes this by, for example, checking how much of the source code is actually run during testing. The results can then be converted to a more readable format such as HTML using Lcov.

- Doxygen is the general standard tool for generating documentation from sources. Using Doxygen it is possible to extract code structure from source files. It is very useful with large scale source distributions. It was used in order to document the code and display it on github pages. Making the code easy to understand.

- **Cloud CI/CD**

Lolek, Javier

- **Git and Version Control**

Javier

During development the version control system git was used, as well as the free to use remote hosting platform Github in order to share this repository among the team members. This way, the team could take full advantage of distributed version control and track all changes throughout the process, reconciling changes made by individuals with the shared remote.

Originally, all changes were rebased into a single linear history, but rebases cause many conflicts. Eventually, the strategy shifted to merge commits.

- **Branching Strategy**

Lolek

The repository is developed through feature branches for implementing each single little feature for the system, named after what is implemented in them (e.g., ***web-api*** branch contains development of introducing Web-API into the system for the connection with the front-end). These will be merged into the ***development*** branch which will contain all the changes of the ongoing release. Upon completion of a released version the ***development*** branch will be reviewed and merged with the ***main*** branch, where all of the tested and working code will be stored.

This approach however, was not desirable, since there were problems keeping ***development*** and ***main*** in sync, in addition to complicating the workflow for little to no benefit. Instead, ***main*** was used as the target for merges from feature branches and tags were used instead to denote versions.

Merging into the MAIN branch is only possible if the automated test pipeline succeeds. Additionally, the approval of at least 2 team members must be received for the merger to happen (including the person proposing the changes)

▪ **GitHub Workflows**

There are two main workflows, one of them, runs all tests through maven, ensuring smooth integration, while the other deploys the images to DockerHub and the services to Google Cloud

▪ **IDE/Development**

Lolek

- IntelliJ IDEA – Writing, implementing, testing and building.
- Apache Maven - Project management, dependency management, packaging…

▪ **Deployment**

The application is deployed to Google Cloud Run through an automated workflow every time a version tag is pushed to GitHub. Google Cloud automatically pulls the DockerHub image that is created by the flow and hosts a managed kubernetes cluster serving the traffic to the container.

● Testing

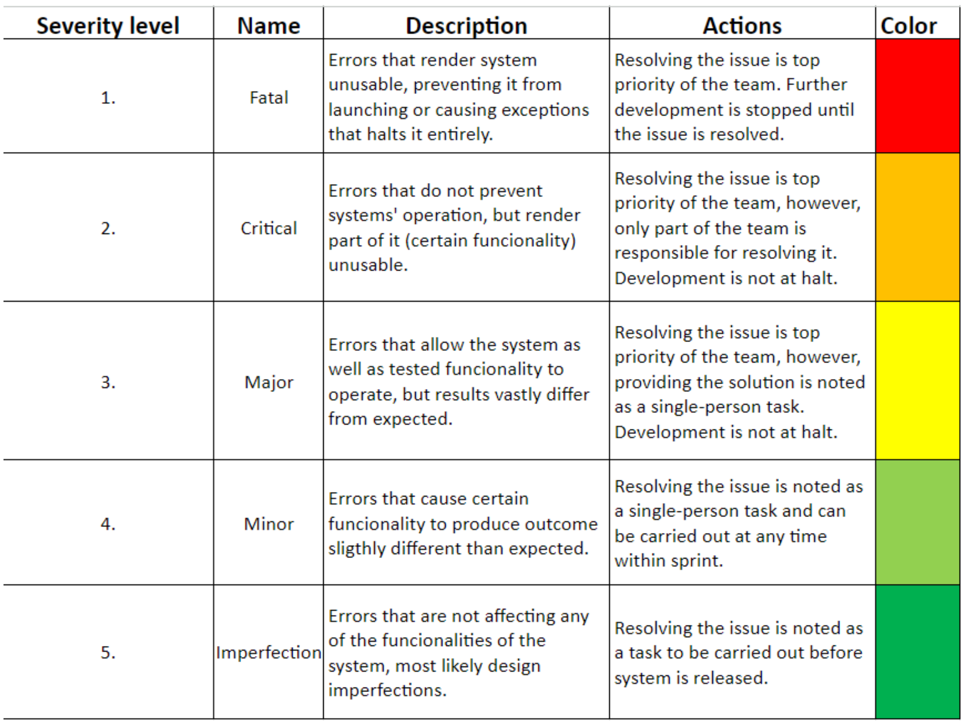
○ Black-box testing (Acceptance tests)

Rojus

The team conducted a variety of tests with the aim of ensuring the quality of user experience while using the system. In order to ensure that the system as a whole satisfies all of the requirements provided by the product owner, the team utilised black-box testing in the form of acceptance testing. This was done to prevent any possible bugs a user of the system may encounter. The tests go through each base and exception sequence of every single use case, thus testing all the possible action scenarios the user may go through. As part of the project appendices, the team created an excel file that was used as a test-book. The book consists of 3 tables:

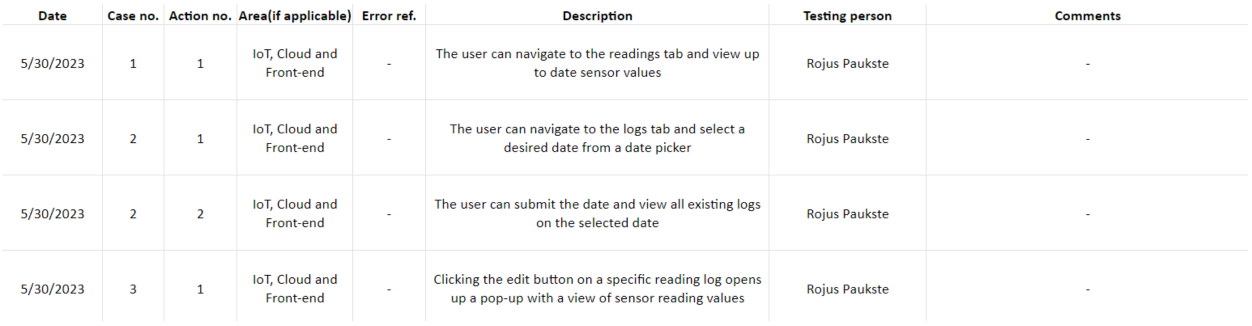
No.	Test Case	Ref. Use Case	Actors	Preconditions	Expected outcome	Action no.	Action	Reaction
1	View recent readings	View atmosphere levels	Manager	N/A	Atmosphere levels of the current time are displayed in separate graphs	1	Navigate from the main page to the readings tab	Verify that the system is displaying the sensor data correctly
2	View logs	Manage logs	Manager	N/A	A list of logged readings from a selected date is displayed	1	Navigate to the log tab and select a date from the date picker	Verify that the date picker works and that a selected date is displayed
						2	Submit the date	Verify that the list of logs from the selected date is displayed

The first table describes all the testing scenarios the team would go through. They reference the use case that is tested, the outcome to be expected, actions to be performed and result verification process.



Severity level	Name	Description	Actions	Color
1.	Fatal	Errors that render system unusable, preventing it from launching or causing exceptions that halts it entirely.	Resolving the issue is top priority of the team. Further development is stopped until the issue is resolved.	🟥
2.	Critical	Errors that do not prevent systems' operation, but render part of it (certain funcionality) unusable.	Resolving the issue is top priority of the team, however, only part of the team is responsible for resolving it. Development is not at halt.	🟧
3.	Major	Errors that allow the system as well as tested funcionality to operate, but results vastly differ from expected.	Resolving the issue is top priority of the team, however, providing the solution is noted as a single-person task. Development is not at halt.	🟨
4.	Minor	Errors that cause certain funcionality to produce outcome sligthly different than expected.	Resolving the issue is noted as a single-person task and can be carried out at any time within sprint.	🟩
5.	Imperfection	Errors that are not affecting any of the funcionalities of the system, most likely design imperfections.	Resolving the issue is noted as a task to be carried out before system is released.	🟩

The second table displays all the different kind of result types. They determine the severity of a test failure and describe what the team should do if such a result type is encountered. The result types are ranked by their importance.



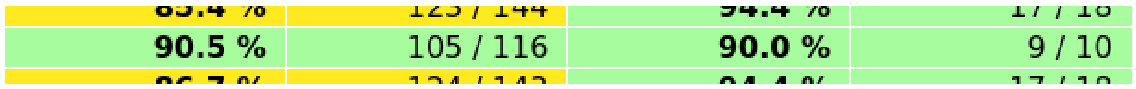
Date	Case no.	Action no.	Area(if applicable)	Error ref.	Description	Testing person	Comments
5/30/2023	1	1	IoT, Cloud and Front-end	-	The user can navigate to the readings tab and view up to date sensor values	Rojus Paukste	-
5/30/2023	2	1	IoT, Cloud and Front-end	-	The user can navigate to the logs tab and select a desired date from a date picker	Rojus Paukste	-
5/30/2023	2	2	IoT, Cloud and Front-end	-	The user can submit the date and view all existing logs on the selected date	Rojus Paukste	-
5/30/2023	3	1	IoT, Cloud and Front-end	-	Clicking the edit button on a specific reading log opens up a pop-up with a view of sensor reading values	Rojus Paukste	-

The third table displays the logs for performed acceptance tests by the team. Each log has a date, referenced test case, referenced action, tier coverage, error reference, description, comment and testing person.

○ IoT Test

Nerijus, Mikkel

The IoT team made use of the test framework “Google Test”, as well as the “Fake Function Framework” to implement a White Box unit-test suite.



90.5 %	105 / 116	90.0 %	9 / 10

Coverage report for lorawan_handler.c

lorawan_handler Test Suite	
Name	**Test Description**

lorawan_handler_create	Verifies that the correct functions are called during creation e.g. xTaskCreate.
lora_uplink_task	Mocks a successful uplink task startup, and verifies all necessary functions are called.
lora_upload_uplink_no_downlink	Mocks a successful uplink without a downlink payload, and verifies it exits unchanged.
lora_upload_uplink_receive_downlink	Mocks a successful uplink with a downlink payload, and verifies both exit unchanged.
lorawan_handler_destroy	Provides a handler and verifies the task is destroyed.

o Front-End Test

Matas

The front-end team used “Jest” to test the Limits Component and Climate Control Toggle Component. The testing technique for the Limits, which has the most functionality out of all components by far, is the White Box testing. This means that the tests test specific functions and the internal implementation of the code. The opposite applies for the Climate Control Toggle Component, which uses the Black Box testing technique. This technique is about testing the program from the user's point of view. The testing coverage approach chosen was the testing of all parts of a few units.

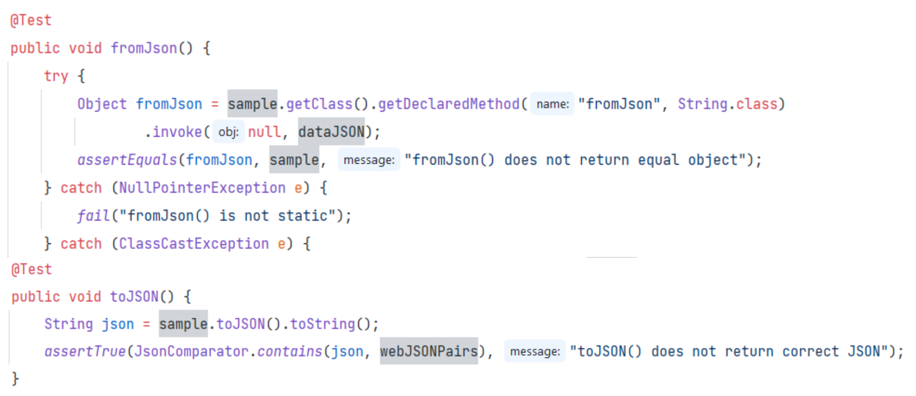
Limits test use case results:

Name	Action	Result
handleSubmit	Mocks the input values and simulates the "save limits" submission event to see if the limits will be saved and sent.	**Pass**
handleCO2Change	Inputs a value to the CO2 field, which has not allowed symbols, and tests if the symbols get removed.	**Pass**
handleUTempChange	Inputs a value to the Upper Limit of the temperature field, which has not allowed symbols, and tests if the symbols get removed.	**Pass**
handleLTempChange	Inputs a value to the Lower Limit of the temperature field, which has not allowed symbols, and tests if the symbols get removed.	**Pass**
handleUHumChange	Inputs a value to the Upper Limit of the Humidity field, which has not allowed symbols, and tests if the symbols get removed.	**Pass**

handleLHumChange	Inputs a value to the Lower Limit of the Humidity field, which has not allowed symbols, and tests if the symbols get removed.	**Pass**
checkIfValid (All correct)	Mocks the correct input values to see if the Limits will be saved	**Pass**
checkIfValid (wrong CO2)	Mocks the incorrect CO2 input values to see if the Limits will not be saved	**Pass**
checkIfValid (wrong Hum)	Mocks the incorrect Humidity input values to see if the Limits will not be saved	**Pass**
checkIfValid (wrong Temp)	Mocks the incorrect Temperature input values to see if the Limits will not be saved	**Pass**

Cloud Test

Tests in the cloud system are run using the JUnit5 engine and api. This allows assertions based on whether certain functionality works and whenever the system changes, it can be verified to ensure that the functionality is still there. Most of the implemented tests regard data processing in one way or another. This is because testing the functionality of a library should be left to the developers of said library.



```
@Test
public void fromJson() {
    try {
        Object fromJson = sample.getClass().getDeclaredMethod( name: "fromJson", String.class)
                .invoke( obj: null, dataJSON);
        assertEquals(fromJson, sample,  message: "fromJson() does not return equal object");
    } catch (NullPointerException e) {
        fail("fromJson() is not static");
    } catch (ClassCastException e) {
@Test
public void toJSON() {
    String json = sample.toJSON().toString();
    assertTrue(JsonComparator.contains(json, webJSONPairs),  message: "toJSON() does not return correct JSON");
}
```

● Results and Discussion

Matas

The result of this semester is a working system that has all of the necessary functionality to monitor the environment. It consists of an IOT device which successfully gets the data from the sensors and sends them to the cloud, from which the web app can take the data and display it. The user can see the current temperature, humidity and CO2 levels on the web app. In addition to the current limits, as well as override the upper/lower limits of temperature and Humidity, and the higher limit of CO2 levels. These limits are sent back to the IOT device through the cloud. The user is able to check the previous logs, which contain the readings from the selected date. The logs are in the cloud’s database and can also have a comment added by the user. The web app will warn the user of readings that surpass its limits by marking the said reading red.

● Conclusions

Rojus

The project puts focus on developing a system that targets to improve air quality within indoor locations with high concentration of people, mostly bars, clubs and other social gathering venues, globally.

The purpose of the Venue management system is to have all the features needed for measuring, displaying and automatically reacting to atmosphere levels within areas where the device sensors are placed, in one system. The solution is a distributed system, highly based on the 3-tier architecture.

The team started their work by performing analysis over the project requirements. They were formatted to user stories and sorted by priority. The result of this process was a list of 29 requirements and 3 user roles. The user stories would be processed in analysis, design, implementation and testing iteratively with each sprint instance. The team would create use cases, their descriptions, activity diagrams and the domain model - all of which laid plans for design.

Throughout the design part of the project, the team would focus on sketching up a more technical architecture model for the system prior to implementing it. Artefacts such as an architectural diagram, class diagram, sequence diagrams would contribute greatly towards constructing the foundation for implementation.

The team used various forms of testing depending on which tier their work was conducted in. Overall both white-box and black-box approaches were used. For black-box testing the team went through the system as a whole using acceptance testing from the users perspective, whereas for white-box the team utilised Google tests, JUnit tests and Jest testing framework.

The end product of the project was reviewed and evaluated by the team at the end of working on the project. The team's thoughts and plans for the future of the solution were laid out in the "Results and Discussions" and "Project future" chapters respectively.

● Project future

Nerijus

Given the limited time allocated to the project there are a number of features that were planned but not yet implemented. That being the case, below are some things that could be implemented in order to improve the project if it was to be updated in the future.

Multiple venues:

One of the more interesting additions would be to be able to have multiple venues. So that it would be possible to see and compare different venues and see which one is doing better than the other one and maybe be able to detect the reason. It would also become more appealing for larger companies because of its scalability.

Multiple accounts:

Having multiple accounts in correlation with having multiple venues would increase the versatility of the system. Enabling the higher ups of the companies to assign different individuals with separate venues. It would add better capabilities of management of the venues.

Multiple devices in one venue(zones):

The possibility of having more than one device in a venue would make it plausible to separate the venue into different zones. That being the case the client would receive more accurate information of the venue. Making it easier and faster to detect problem areas.

Use of Sound sensor:

Adding a sound sensor data while having the zones feature to the venue would make it easy to see where the volume in the venue is the highest and where it is the lowest.

Display of sensor limit:

Display the limits in the graph while not yet exceeding it would be useful and handy because it would allow the client to visually see if the values are reaching near the limit and at what times.

Use of Light sensor:

The use of a light sensor would help in determining light levels in the venue. Making it possible to make it adjust the light when it is bright or is the day time. That would help save money for the companies.

Use of PIR sensor:

PIR would enable the companies to know at what time there was motion in the venue making it convenient to see camera footage from that specific time. It would also save on electricity for when there are no people in the venue it would turn off the lights.

On/Off button:

Having an on/off button on the IoT device would be very useful, because it would make transporting the IoT device easier without having to disconnect it. It would also help because it could be turned off when not in use.

Administrator:

Because of time constraints administration functionality was not implemented. Having administration functionality would allow for addition and removal of venues. That being the case it would improve the management part of the venue system.

● Sources of information

● Appendices

- Appendix E - Project Description
- Appendix F - Use Case Description
- Appendix G - Diagrams in SVG
- Appendix H - User Guide
- Appendix I - Installation Guide
- Appendix J - Astah Files
- Appendix K - GitHub Links
- Appendix L - Video Presentation Link