



---

# Project Report

IAQ

---

## Students

Ainis Skominas 266756  
Aleksandr Zorin 224493  
Balkis Ibrahim 260092  
Fadi Atia Dasus 266265  
Muhammad Nadeem 266704  
Oskars Arajs 266534  
Robert Misura 269381  
Roza Ibrahim Hasso 266235  
Vladimir Rotaru 266914  
Yasin Issa Aden 267276

## Supervisors

Kasper Knop Rasmussen  
Knud Erik Rasmussen  
Lars Bech Sørensen  
Erland Ketil Larsen

**129654 Characters**

**ICT Engineering**

**4th Semester**

**2019/05/15**



## Table of content

Abstract	8
1. Introduction	9
2. User Stories and Requirement	10
2.1. User Stories	10
2.2. Functional Requirements	11
2.3. Non-Functional Requirements	12
3. Analysis	13
3.1. Scenarios	13
3.2. Use Case Diagram	14
3.3. Use Case Description	15
3.4. Domain Model	16
3.5. Activity Diagram	17
4. Design	17
4.1. Embedded	18
4.1.1. Embedded Sequence Diagram	18
4.1.4. Web Socket Bridge Application	20
4.1.5. Web Socket Bridge App Technologies	21
4.2. Data	22
4.2.1. EER Diagram	22
4.2.2. Data Warehouse - ETL	25
4.2.3. SQL Server Job (Scheduling)	25
4.2.4. Power BI	26
4.2.5. Data Web Services Class Diagram	27
4.2.6. Web Service Sequence Diagram	29



4.2.7.	Data Technologies	30
4.2.8.	Java Extractor	31
4.2.9.	Web application	33
4.3.	Android	36
4.3.1.	Android Class Diagram	36
4.3.2.	Android Sequence Diagram	38
4.3.3.	Android Architecture	39
4.3.4.	Android Technologies	39
4.3.5.	Android Design Patterns	40
4.3.6.	UI Design Choices	41
5.	Implementation	44
5.1.	Embedded	44
5.2.	Data	48
5.3.	Java Extractor	48
5.4.	Web Application	50
5.5.	Data warehouse	52
	Triggers	54
	Web Service	56
5.6.	Android	57
5.6.1.	Single Activity Principle	57
5.6.2.	Web Services (Network Helper)	59
5.6.3.	Firebase Cloud Messaging (FCMHelper)	61
5.6.4.	MVVM Structure	62
5.6.5.	UI Implementation	64
5.6.6.	Charts	68
6.	Test	70



6.1.	Embedded	70
6.1.1.	Test Specifications	70
6.2.	Data	71
6.2.1.	Postman	71
6.2.2.	Test Specifications	72
8.1.1.	Java Extractor	73
8.1.2.	Web application	73
8.2.	Android	74
8.2.1.	Test Specifications	74
8.3.	Test Specifications	75
9.	Results and Discussion	76
10.	Conclusions	77
11.	Project future	78
12.	Sources of information	79
13.	Appendices	80

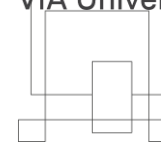
## List of figures and tables

Figure 1 - Use Case Diagram .....	14
Figure 2 - Use Case Description for Visualizing Data .....	15
Figure 3 - Domain Model.....	16
Figure 4 - Activity Diagram for Visualizing Data .....	17
Figure 5 - Embedded Sequence Diagram .....	18
Figure 6 - Embedded Class Diagram .....	19
Figure 7 - Web Socket Sequence Diagram .....	20
Figure 8 - Web Socket Bridge Class Diagram .....	21
Figure 9 EER Diagram.....	22
Figure 10 - Data Warehouse Design.....	23
Figure 11- ETL Process Steps .....	25
Figure 12 - SQL Server Job Scheduling.....	26
Figure 13 - Power BI 4 Chart Visualization.....	27
Figure 14 - Web Services Class Diagram .....	27
Figure 15 - ORM Life Cycle.....	28
Figure 16 – Web Service Sequence Diagram .....	29
Figure 17 - Java Extractor Domain Model .....	31
Figure 18 - Sequence Diagram of UpdateCO2.....	32
Figure 19 - Web Application Login Form.....	34
Figure 20 – Web Application List of rooms.....	34
Figure 21 - Web Application Create Room.....	35
Figure 22 - 1st Part of Android Class Diagram .....	36
Figure 23 - 2nd Part of Android Class Diagram.....	37
Figure 24 - Android Sequence Diagram.....	38
Figure 25 - Android Architecture Diagram .....	39
Figure 26 - Login Fragment.....	41
Figure 27 - Room Choice Fragment.....	41
Figure 28 - Room Main Fragment .....	42
Figure 29 - Warning List Fragment.....	42
Figure 30 - Report Fragment.....	43
Figure 31 - CO2 Sensor Initializer .....	44



Figure 32 - CO2 Sensor Interface .....	45
Figure 33 - Temperature/Humidity Measure Method .....	45
Figure 34 - Lora Connection Request .....	46
Figure 35 - LoraMessage Serialization.....	46
Figure 36 - LorA Message Parsing For MongoDB.....	47
Figure 37 - Parsing Methods.....	47
Figure 38 - EUIMongo Class.....	48
Figure 39 - CO2 Class .....	48
Figure 40 - UpdateCO2 Method.....	49
Figure 41 - Web App Authentication .....	51
Figure 42 - Web App Controller.....	52
Figure 43 - Removes null values from temporary fact tables.....	53
Figure 44 - Replacing the surrogate keys in the temporary fact table with the primary keys from each dimension .....	53
Figure 45 - Loads the dimension with the new data .....	54
Figure 46 – Temperature Trigger .....	54
Figure 47 - Updates Warning Table .....	55
Figure 48 - NotificationsService Class Schedule Example .....	56
Figure 49 - Navigation Controller Usage in MainActivity.....	57
Figure 50 - Navigation Example When Toolbar Menu Item is Clicked .....	57
Figure 51 - Navigation after logging in and selecting a room inside of Android app.....	58
Figure 52 - GetAllCo2sByRoomIdToday Class .....	59
Figure 53 - SensorsAPI CO2 Requests.....	59
Figure 54 - GetAllCo2sByRoomIdToday Example in NetworkHelper Singleton Class ..	60
Figure 55 - ReceiveDataService Class.....	61
Figure 56 - UpdateLiveData Method in FCMHelper class.....	61
Figure 57 - MVVM Packaging Structure .....	62
Figure 58 - LiveDataViewModel Initialization.....	62
Figure 59 - Subscribe/Unsubscribe from FCM .....	62
Figure 60 - LiveDataViewModel methods for subscribing/unsubscribing to FCM .....	63
Figure 61 - LiveDataRepository methods for calling FCMHelper class to subscribe unsubscribe from FCM.....	63

Figure 62 - Endpoint of MVVM Structure of Subscribe/Unsubscribe example of Firebase Cloud Messaging .....	64
Figure 63 - Room Choice Fragment.....	64
Figure 64 - Login Fragment.....	64
Figure 65 - Report Fragment.....	65
Figure 66 - Room Main Fragment .....	65
Figure 67 - Today's Data Fragment.....	66
Figure 68 - Warning List Fragment.....	66
Figure 69 - Example how minimum and maximum values are found in retrieved data	67
Figure 70 - TodaysDataFragment onCreateView method .....	68
Figure 71 - Methods from TodaysDataFragment.....	68
Figure 72 - Displays 1st part of setupCo2Chart method .....	69
Figure 73 - LoRaWan Screenshot confirming received packets .....	70
Figure 74 - Postman Test.....	71
Figure 75 - JUnit Test for loadDevice() & updateDevice() .....	73



## Abstract

Project purpose is to build a system that measures data from sensors which are installed at a certain location, then it is sent to database for analysis. This data later is exposed to Android applications through web services. Sensors consists of an Arduino Atmega2560 with a custom-built shield, that is running C code application and sends measured data to the LoRaWan server from which with a web socket connection we send data to MongoDB.

To retrieve data from MongoDB a bridge application is built. This application is named Java Extractor. It maps to both the SQL server and the MongoDB document. Then repositories for each is created handling all the queries. The application is then responsible for initializing the data from MongoDB inside the SQL Server and continuously update new data here.

SQL data is later returned into REST API server that exposes gathered data through web services to Android where visualization is done according to the received data.



## 1. Introduction

In the current era we live in, because of severe industrialization and the upcoming futuristic projects all around the world, the carbon dioxide (CO<sub>2</sub>) levels are rising faster than expected. Presently, the maximum acceptable carbon dioxide levels are between 600-1000 ppm, while the current atmospheric values are around 450-500 ppm and even a bit higher in larger cities. Hypersensitivity and allergies seem to flourish with poor learning and increased absenteeism as the main consequences. Researchers and scientists seem to agree that poor indoor air quality is to be blamed. They find that high levels of carbon dioxide influence people's health, particularly the brain. They find that even low levels of CO<sub>2</sub> are dangerous in closed rooms. The same issues apply to business meeting rooms. This means that poor indoor air quality may be the cause of poor productivity and increasing health problems in general among many people.

Improving the ventilation in most buildings would help solve these issues and notifying people about this problem is part of the project focus. The challenge is to make people aware of the increased levels of CO<sub>2</sub> in rooms and act accordingly. CO<sub>2</sub> makes up a part of every breath humans exhale causing the levels to rise as more people gather. Therefore, it is imperative to notify the dangers of the increased level of CO<sub>2</sub> all the time. By targeting the teaching institutions where there is a lot of focus involved, the teachers are warned about the CO<sub>2</sub> levels, thus keeping them informed, which they can act accordingly to the situation. Naturally, there will be more to this project than meets the eye.

## **2. User Stories and Requirement**

The purpose of this chapter is to create the user stories that help the developers understand what the client wants from the system. After user stories are drawn, they are referenced to create simplified requirements which will help the developing team know what is expected of the system.

### **2.1. User Stories**

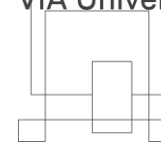
User stories are a simple way of getting a better understanding of the functionality of the system between the developers and customers. The customer explains what he needs from the system so that later on the developers can elaborate on these stories.

1. As an admin I want to add users to the system so that all the users will have access to the app.
2. As an admin I want to delete users from the system so to block users access to the app.
3. As an admin I want to edit users from the system so users change their credentials.
4. As an admin I want to add rooms to the system so that I can track the measurements from that room.
5. As an admin I want to delete rooms to the system so that I remove non existing rooms.
6. As an admin I want to edit room to the system so that I can edit the information about the rooms
7. As an admin, I want to add a new device so that I can track CO2, temperature, and humidity.
8. As an admin, I want to delete a device so that I can remove unused device.
9. As an admin, I want to edit a device so that I can change the room of the device.
10. As a user I want to check CO2 levels in the classroom so that I ensure the best IAQ (Indoor Air Quality).

11. As a user I want to check the temperature in the classroom so that I ensure the best IAQ.
12. As a user I want to check the humidity in the classroom so that I ensure the best IAQ.
13. As a user I want to see a diagram about CO<sub>2</sub>, temperature and humidity levels in the classroom so that I ensure the best IAQ.
14. As a user I want to read a report about CO<sub>2</sub>, temperature and humidity levels in the classroom so that I ensure the best IAQ.
15. As a user I want to check warnings history so that I can see where that happened and when.
16. As a user, I want to be able to see live data from one particular room. so that I keep an eye on IAQ

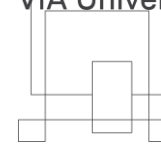
## **2.2. Functional Requirements**

1. The system must display the co<sub>2</sub> levels, temperature and humidity of a specific room where the device exists.
2. The system must generate a report about the collected data in a specific room.
3. The system must visualize live data collected from the sensors.
4. The system must generate a co<sub>2</sub> warning.
5. The system must save warnings history (when that happened, in which room).
6. The system must analyze co<sub>2</sub>, temperature and humidity data.
7. The system must allow admin to add a new room.
8. The system must allow admin to remove room.
9. The system must allow admin to edit room.
10. The system must allow admin to add a new device.
11. The system must allow admin to remove device.
12. The system must allow admin to edit device.
13. The system must allow user to choose a classroom.



### **2.3. Non-Functional Requirements**

1. The system must collect data 24/7 in order to make a comprehensive report.
2. The system must respond to searches within 10 seconds.
3. The system must be implemented in Java, C# and C.
4. The system must use business intelligence tools to analyze data.
5. The system must be able to transfer data using Lora Wan.
6. The system must be able to expose data using web services.
7. The system must be able to receive live data by using Firebase Cloud Messaging.
8. The system must be able to make an analysis of the data and create reports.
9. The system must be able to provide a responsive user interface.
10. The system must be able to authenticate user.



### 3. Analysis

In the Analysis section, the scenarios, use case diagrams and use case descriptions are being discussed.

#### 3.1. Scenarios

##### **Choose a Classroom Scenario:**

Precondition for this scenario is that user is logged in. A list of rooms appears, user selects a room.

##### **Visualize Data Scenario:**

User logs in, room list appears, user selects a room, visualized data appears.

##### **Visualize Report Scenario:**

Precondition for this scenario that room was selected. User selects to visualize report and a generated report appears.

##### **Check Warning History Scenario:**

User clicks on Check Warning History button, a list of warnings appears

##### **Display Warning Notification Scenario:**

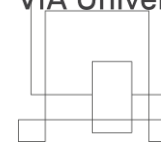
Precondition for the scenario is that abnormal levels of co2, temperature or humidity got captured. User gets a notification about a warning.

### 3.2. Use Case Diagram



Figure 1 - Use Case Diagram

In this use case it is clear that there are only two actors: Admin; User;  
The Administrator can add, edit, and remove admins, rooms and devices. The user is the actual one that is interacting with the system. the User has 4 use cases and each of those use cases include another use case "Choose classroom".



### 3.3. Use Case Description

Visualize data / UseCase Description	
ITEM	VALUE
UseCase	Visualize data
Summary	Visualize the collected live data.
Actor	User
Precondition	User has chosen a classroom.
Postcondition	User is able to view live data.
Base Sequence	1. System provides live visualized data according to the selected room.
Branch Sequence	
Exception Sequence	1.a. System notifies that live data is not accessible.
Sub UseCase	Choose classroom
Note	

*Figure 2 - Use Case Description for Visualizing Data*

As per the name a use case description is a report type description that states the sequence of steps taken to achieve the main success scenario and states whether or not there are any preconditions that have to be met and if there are branch/exception sequences possible.

In the example above it has only one step in its base sequence meaning that when the user chooses to utilize this part of the system, he just has to take one additional step after the precondition is met.

### 3.4. Domain Model

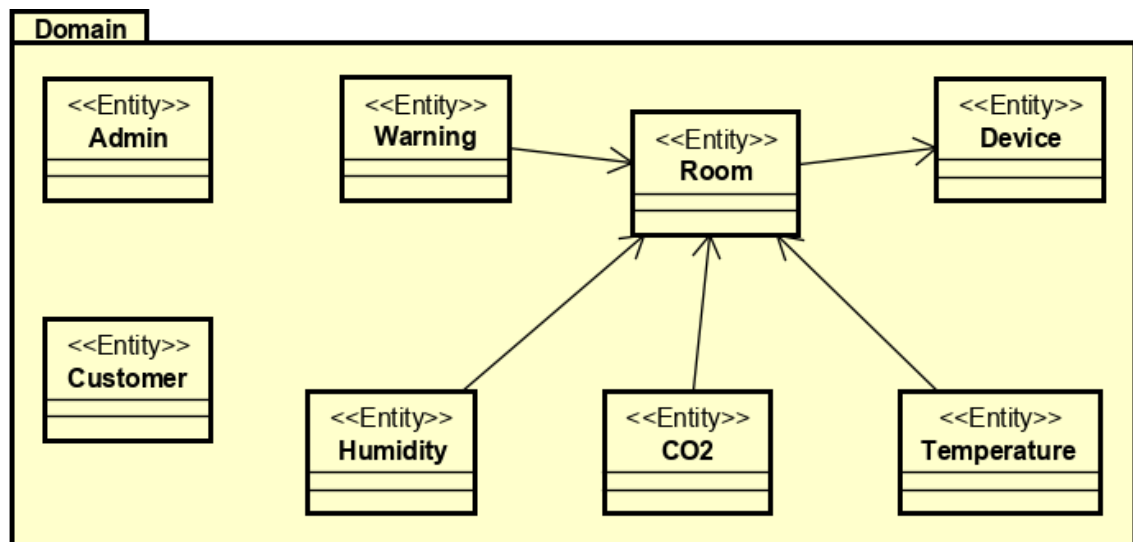
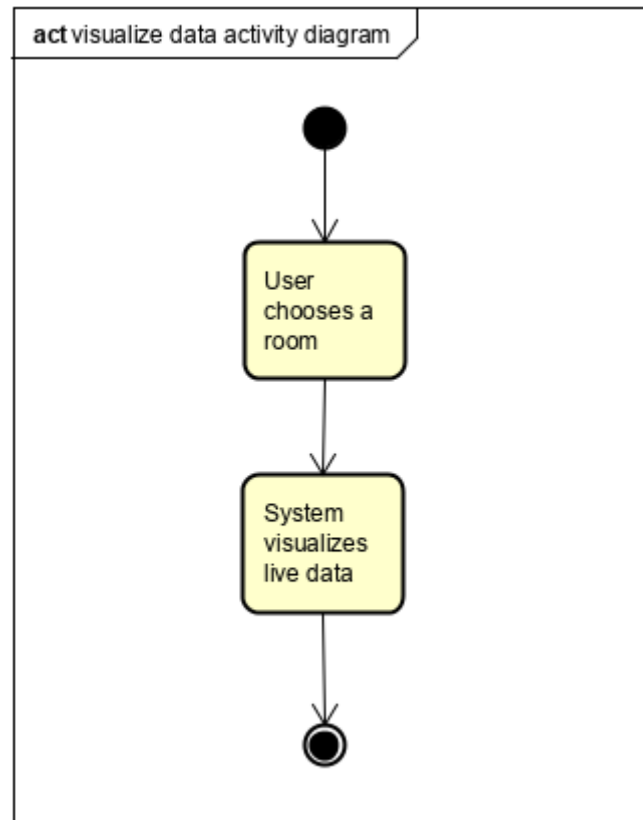


Figure 3 - Domain Model

The domain model shows in an abstract level what parts of the system are connected to what. It's a simple thing to reflect on since it gives the developer a clear overview of how the system is intertwined.



### 3.5. Activity Diagram



*Figure 4 - Activity Diagram for Visualizing Data*

An activity diagram shows the steps the user takes in the application in order to achieve the desired goal.

In the example above the first step, the user has to take is "Choose a room". This is so the system knows on which room to focus and not retrieve irrelevant data but rather return exactly what the user wants.

## 4. Design

In this section the different parts of the project group starts branching out and creating subsections to better explain their own relevant parts of the system.

## 4.1. Embedded

### 4.1.1. Embedded Sequence Diagram

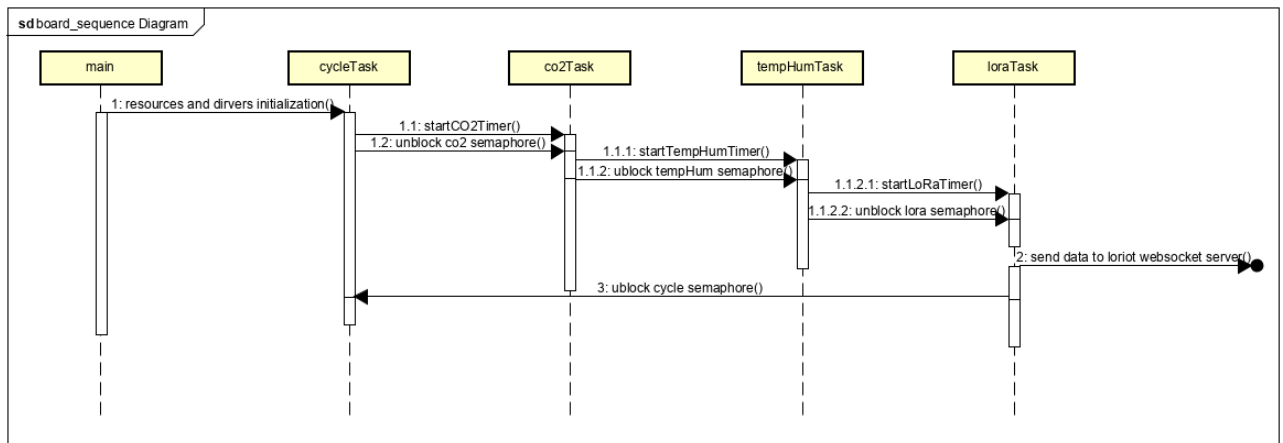


Figure 5 - Embedded Sequence Diagram

The board sequence diagram shows the process of starting the process of data measurements and sending the measured data over LoRaWan module to the web socket server. When the system is started, the main module initializes resources and drivers. FreeRTOS was used as a minimalistic operation system, to help the system run in safe thread way and in given order. The timers and semaphores provide a window for the measuring of data to take place. All of the tasks are initialized at the same time but the measuring of data takes place one-by-one thanks to the semaphores. The final task is the loraTask that calls the `lora_send_data()` method to send the measured data to Lorient web socket server.

### 4.1.2. Embedded Technologies

#### Hardware

In addition to the Arduino ATMEGA2560 a Arduino shield was mounted on it to give us the use of different sensors and communication abilities with the LoRa server.

The main components for this project included:

- 1) MH-Z19 module - CO2 sensor
- 2) HIH8120 module temperature/humidity sensor

- 3) LoRaWAN RN2384 module
- 4) VIA shield

**MH-Z19** is a small CO2 NDIR sensor. Meaning its a **nondispersive infrared sensor** that measures how many CO2 particles pass through it's embedded gas tube.

**HIH8120** is a digital temperature and humidity sensor with an accuracy of +/- 2.0% for relative humidity (RH) and a temperature accuracy of +/- 0.5°C.

**LoRaWAN RN2384** module is based on LoRa technologies that utilizes long-range communication opportunities with different sensor technologies.

**VIA SHIELD** is a specialized arduino shield that has sensors, LED's, LoRa module and JTAG debugging pins all embedded into it for ease of use.

### Software

The board is running on **FreeRTOS** software. This software helps the program run in a safe thread way and handle necessary resource memory allocations. This allows the system to run the tasks in given order and make sure that system will not do anything malicious.

#### 4.1.3. Embedded Design Patterns

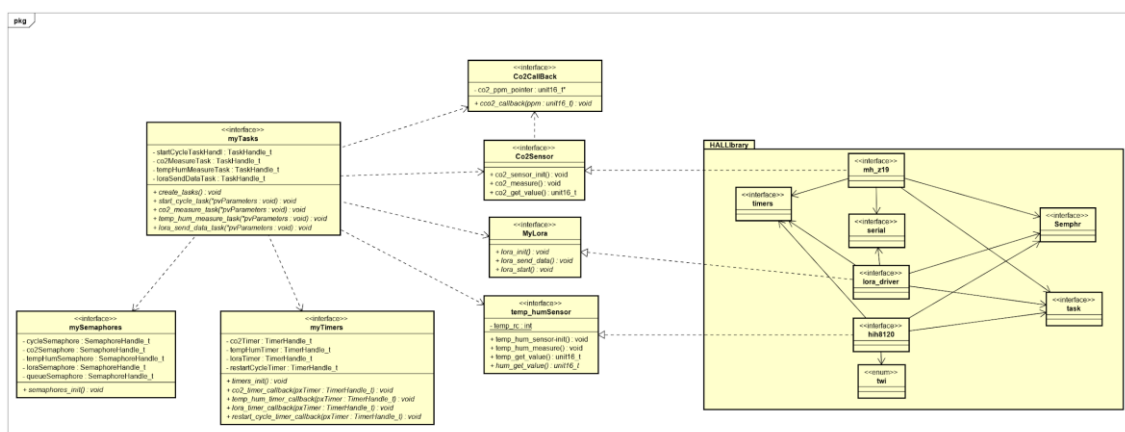


Figure 6 - Embedded Class Diagram

Embedded class diagram consists of multiple interfaces containing methods that are meant to initialize the drivers that are necessary for the system to calibrate itself with the proper settings.

MyTask interface holds the methods for scheduling tasks needed to perform measurements from the sensors and also to send the data through the GateWay and to Lorient Server.

MySemaphores interface holds the methods needed for creating mutexes needed for each task so that it can work uninterrupted.

The return codes are of typedef enum for ease of use so that we can return a String set when passing corresponding value and to avoid “magic numbers”.

As per the implementation of methods, all of it is done inside their corresponding packages.

#### 4.1.4. Web Socket Bridge Application

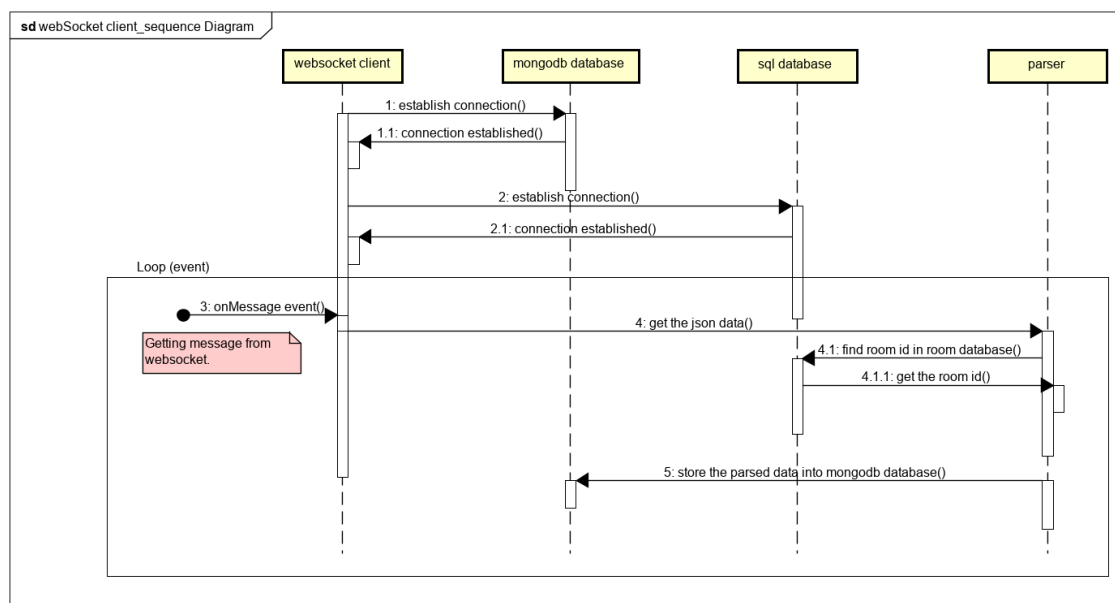


Figure 7 - Web Socket Sequence Diagram

The web socket sequence diagram shows the process of script waiting for event that happens when Lorient server sends message to web socket client (script). In order to store data to database, connection between script and MongoDB cluster is established first. Connection to SQL database must be established also, in order to get all necessary information that needs to be stored in our MongoDB database. Because messages from Lorient web socket server contains too many fields, that system does not need for further computing and displaying data, message first goes through parsing process and

requested data are stored in a new JSON object. In the last step, this object is sent to MongoDB database. Process repeats every time that script gets the message from web socket server.

#### 4.1.5. Web Socket Bridge App Technologies

For the application, two versions of script were made. Due to lack of information, the group decided to use Python language to create script. Script was successful and used for initial testing. After conversation with teachers, the script was also coded in Java language. Final version of system contains both builds of scripts.

#### 4.1.6. Web Socket Bridge App Class Diagram

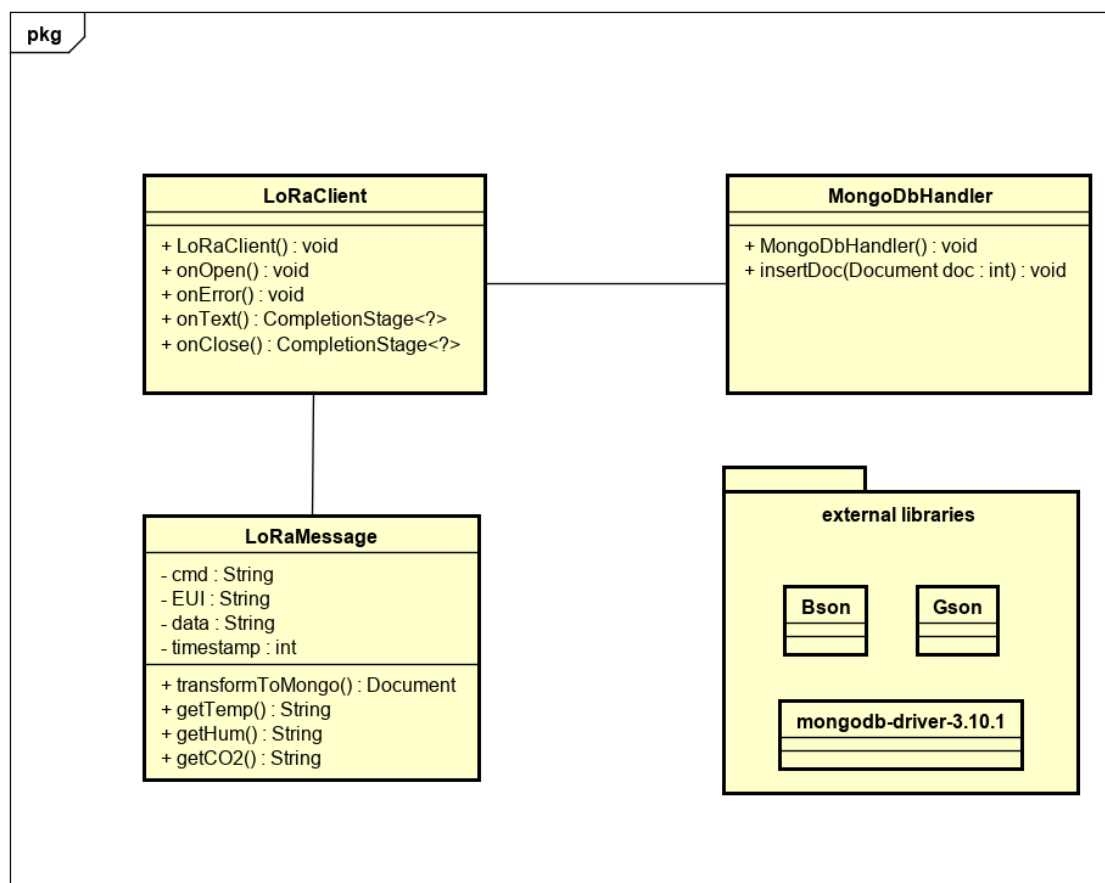


Figure 8 - Web Socket Bridge Class Diagram

Class diagram consist of three classes. LoRaClient is a listener to the web socket and waits for the event (onText()), when message from web socket server is sent. After message was received, this message is transformed into MongoDB JSON document and is immediately sent to database. Both MongoDBHandler and LoRaClient constructors hold the code for initializing and establishing connections between database (mognodb) and web socket (LoRa).

## 4.2. Data

### 4.2.1. EER Diagram

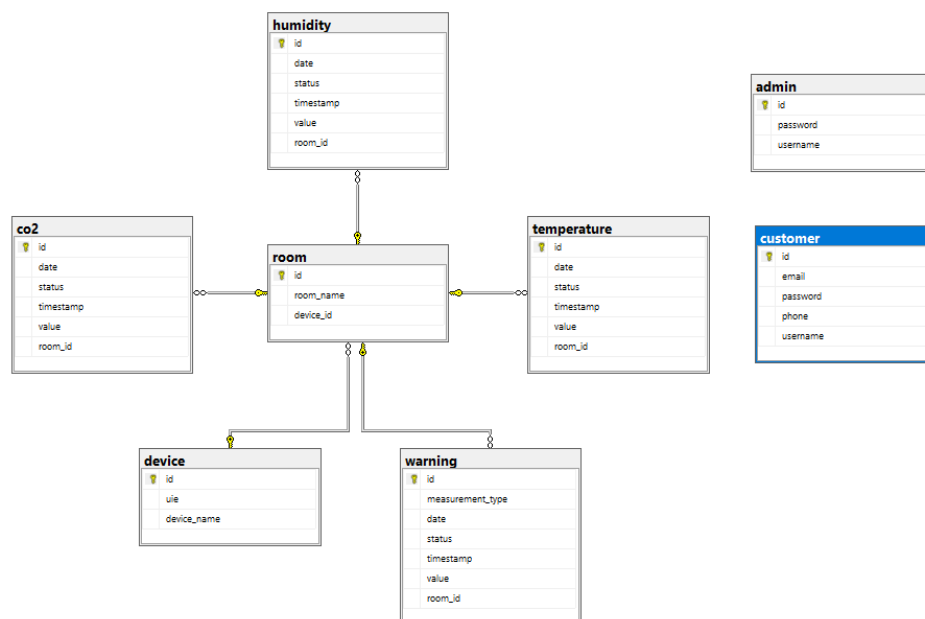


Figure 9 EER Diagram

After analyzing the use cases for the system, the EER diagram has been designed with the following entities: Room works as a middleware between the entities, [BT1] it is presented as a foreign key in almost all the tables. Co2, Temperature, and Humidity are the main entities in this diagram, they present the measurements. All the tables of measurement include attributes about: date, timestamp, the value that has been recorded from the sensor, room id, and finally the status field that will be set to either NORMAL, HIGH or LOW. The table of warnings will hold all the rows from co2, humidity and temperature where the status attribute is set to either LOW or HIGH. The device is holding information about the device used for measuring the data including its uie. The admin table will be used to authenticate the admin through the login function in the web app. The customer table might be used to authenticate android app users in the future. The database has been generated using the Hibernate persistence framework to persist Java objects in a relational database. All the ids are primary key and a clustered index. No rows will be saved into the warning table directly.

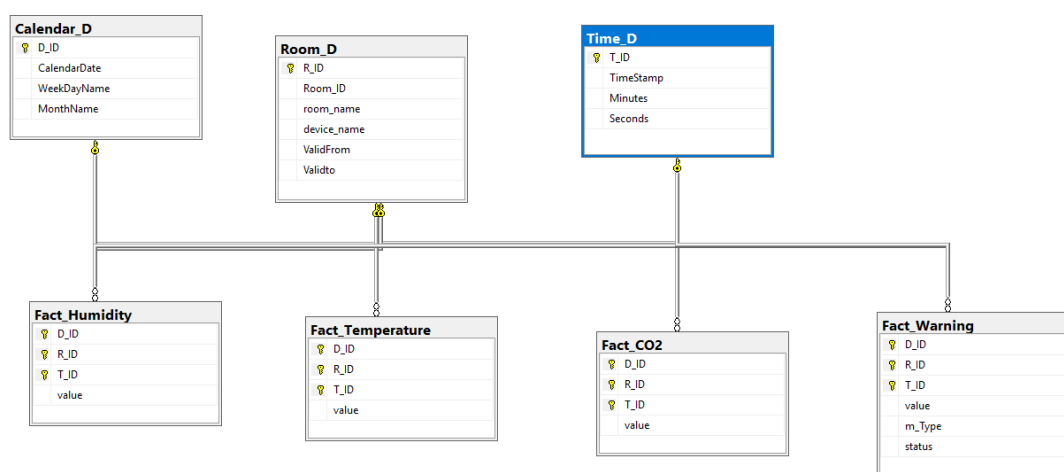
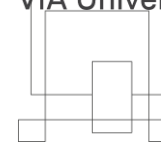


Figure 10 - Data Warehouse Design



The diagram above shows the data warehouse design. It has four fact tables and three dimensions. Each fact table has a relationship with all the dimensions. there is no other relationship neither between the fact tables nor the dimensions. Four steps needed to reach the dimensional design.

**1. Choosing the business processes.**

monitoring the air quality

**2. Declaring the grain.**

measurement every 2 minutes

**3. Identify the dimension.**

Three dimensions were included: Calendar, Room and Time. Regarding the room dimension, three attributes were needed to cover the business requirements, therefore combining two attributes instead of creating two separate dimensions was the best decision.

**4. Identify the fact tables.**

The fact tables contain the measurement value, and the foreign keys from all the dimension.

Regarding the warning fact it has extra fields to know for which type does the warning belongs and its status.



#### 4.2.2. Data Warehouse - ETL

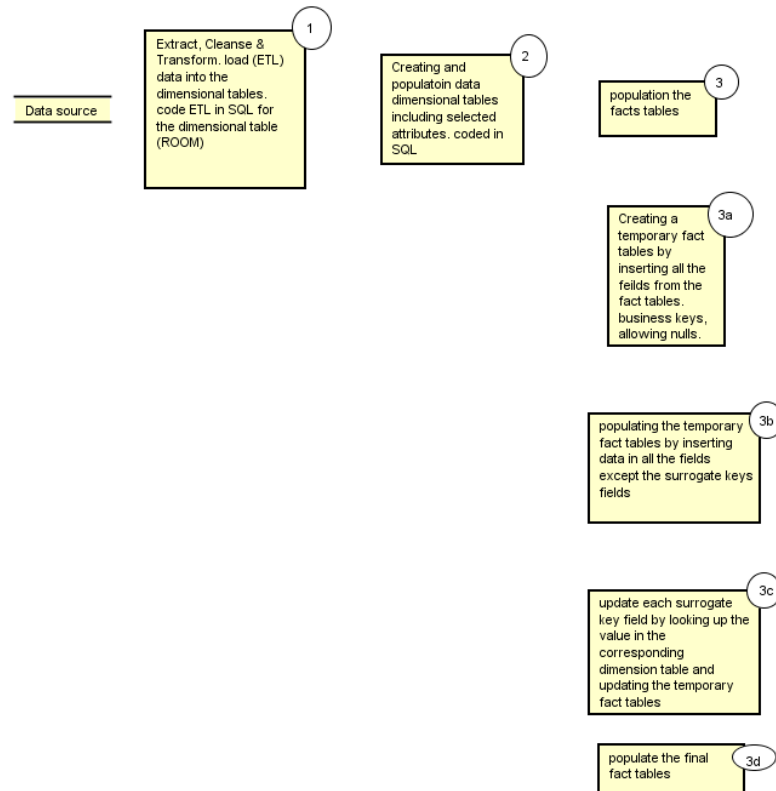


Figure 11- ETL Process Steps

#### 4.2.3. SQL Server Job (Scheduling)

SQL Server job is used to schedule queries to make some amendments in tables or columns. It is three steps procedure. First is define job where name and description are filled, second is to create the new step where write the queries to perform and third is scheduling to create new schedule, when it will start.

Figure 12 represents server job and it is starting everyday night at 00:00. So, data will be inserted from entities to data warehouse every day.

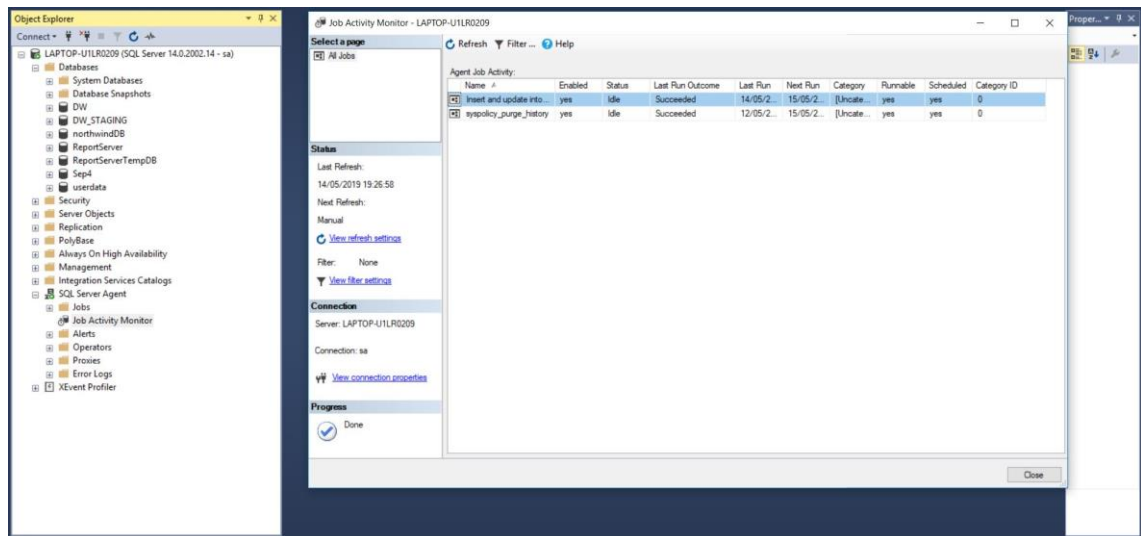


Figure 12 - SQL Server Job Scheduling

#### 4.2.4. Power BI

Power bi is used to visualized data and create business Intelligence reports. Based on data warehouse of Co2, Temperature and Humidity, three pages are created to visualize data on each. In *Figure 13*, four different kind of graphs are used to represent trends in humidity (example). Line Chart represents daily fluctuations and trends in humidity. Humidity values are lying between high and low depending on the air quality of room on that day. Column Chart and Pie chart represents the average humidity on base of specific room name. Gauge is also used to show how much average humidity in specific room for all days. By clicking on column chart for any room, trends and average humidity is shown for specific room. Report server is also created to publish on web but deployed on local server. <http://laptop-u1lr0209/reports/powerbi/Co2-Humidity-Temperature>



Figure 13 - Power BI 4 Chart Visualization

#### 4.2.5. Data Web Services Class Diagram

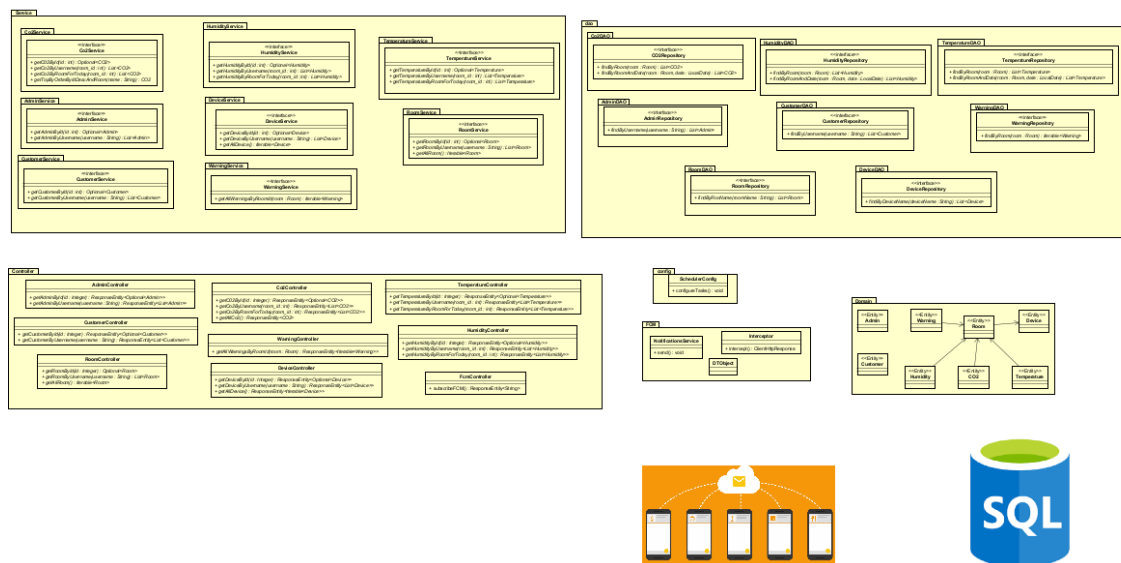


Figure 14 - Web Services Class Diagram

The Web services class diagram consists of 6 packages, each of which has some sub-packages. Starting with the controller's package, it has one controller for each

entity in the model. The controllers are defining the endpoint for the web service, and it maps HTTP GET requests to the database through the service package.

The service package communicates with the appropriate DAO package in order to request data from the database and return it to the controller. All the logic happens in the service package, here the system implies its business logic to the entities and maps it back to its controller.

In the dao package, the system has a collection of Crud Repositories that communicate directly with the database server.

## ORM Architecture

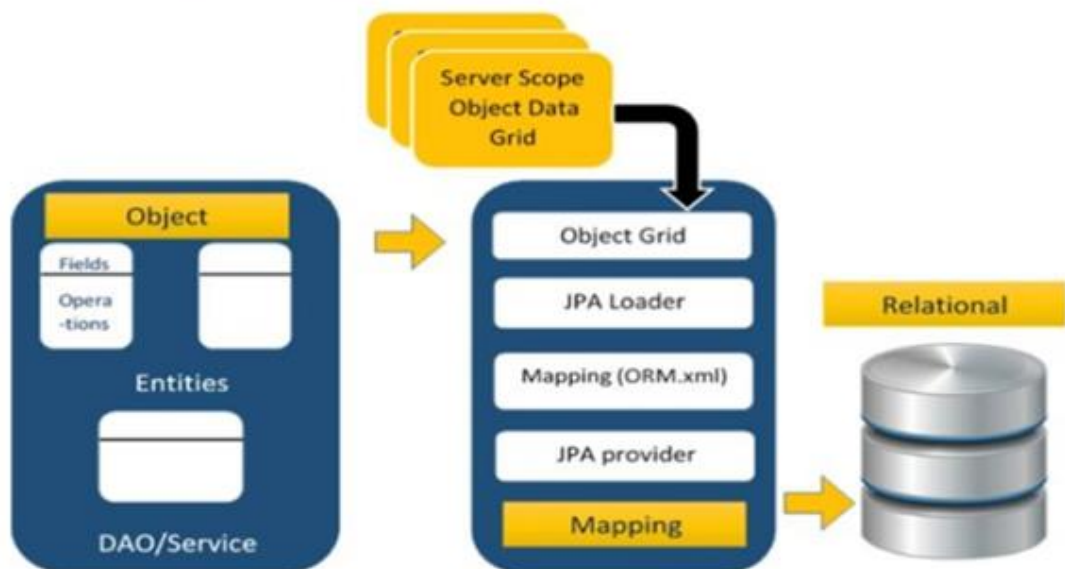


Figure 15 - ORM Life Cycle

The configuration package organizes the scheduling jobs and it enables synchronization tasks for managing multithreading execution in the program.

Finally, the firebase cloud messaging package, which includes three classes for managing sending notifications to firebase which will forward the notification to the

Android app. The message basically contains the latest updated data from the database in order to view live data in android.

SOLID principles have been followed during the design and the implementation for this system, which leads to a high-quality product in the end. All the dependencies in this application have been managed by spring container using the Autowire injection mechanism provided by the spring container.

#### 4.2.6. Web Service Sequence Diagram

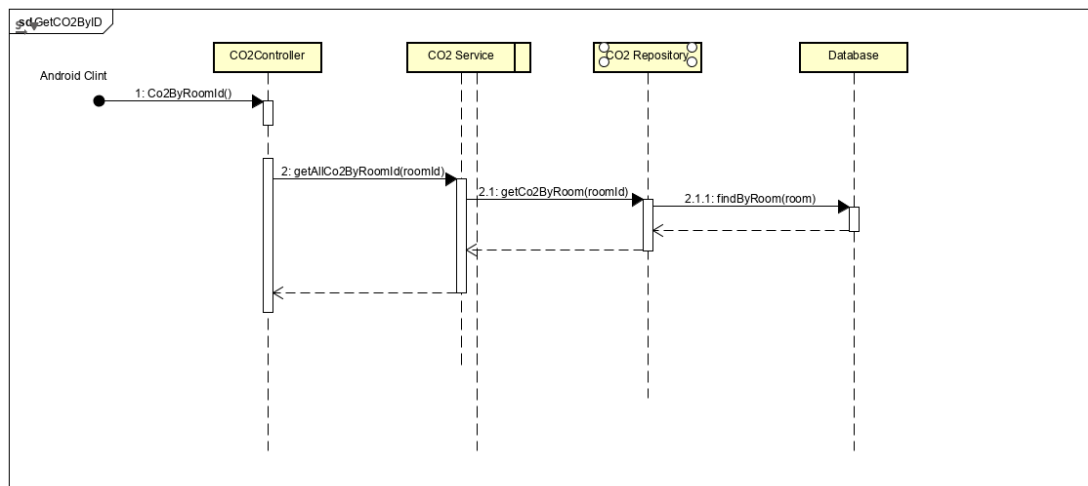
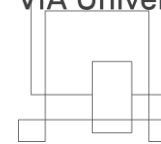


Figure 16 – Web Service Sequence Diagram

This sequence diagram illustrates retrieving co2 measurements based on room id. The controller receives a GET request contains the room id, based on that id. The controller calls `getAllCo2ByRoomId(id)` from the co2 service interface which in turn make another call to `getCo2ByRoom(id)` from the co2 Repository interface. Furthermore the repository interface extends `CrudRepository` interface which interprets the method `findByRoom()` to a sql query behind the scenes. The query returns all co2 rows from the database for the room object passed as a parameter in the method `findByRoom()`.



#### 4.2.7. Data Technologies

**Firebase Cloud Messaging (FCM)** - provides a reliable and battery-efficient connection between the server and devices that allows delivering and receiving messages and notifications on iOS, Android, and the web at no cost. FCM has been used in the system to send live data to the android app.

**Spring Boot** - A brand new framework from the team at Pivotal, designed to simplify the bootstrapping and development of a new **Spring** application. The framework takes an opinionated approach to configuration, freeing developers from the need to define boilerplate configuration.

**ATLAS MongoDB** - For connecting to mongo db the system uses mongoDb Atlas in the cloud, so everyone can have access to the same data all the time no matter where the user will run the app from.

**Hibernate ORM** - an object relational mapping tool for the java programming language. It provides a framework for mapping an OO domain model to a relational database. Hibernate handles object relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

**Actuator** - App monitor. The main benefit of this library is that it provides production grade tools without having to actually implement these features ourselves.

#### 4.2.8. Java Extractor

When developing the system some sort of connection is needed to be established between the SQL Server and the mongodb. Since the embedded team is sending data from the sensors to mongodb using the LoRaWan. To save this data into the SQL Server a bridge application handling this needed to be established. An application named Java extractor was made. This application was developed in Java using the Spring framework, and its responsibilities would be to load and continuously update new data present in the mongodb into the database.

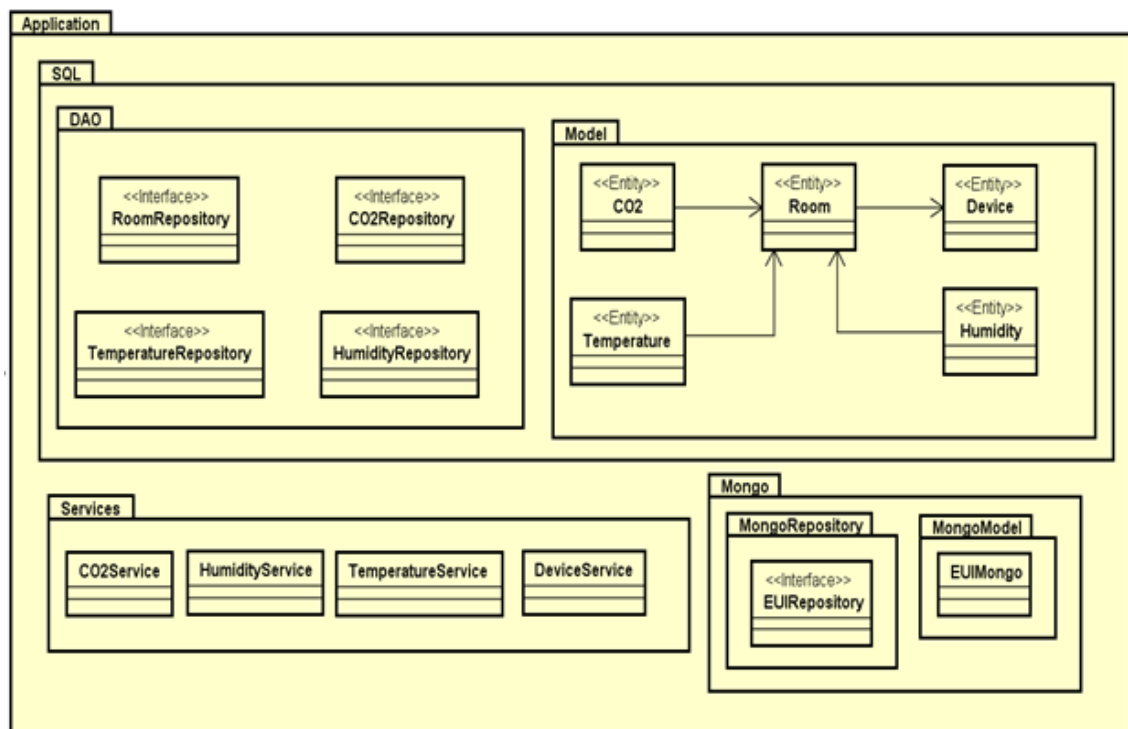


Figure 17 - Java Extractor Domain Model

The diagram above illustrates the class diagram for this application. The model inside the SQL package consist of the classes that maps to the SQL Server(database). First and foremost, the three measurements are located here since they are the primary values that are retrieved from the embedded team. They are all connected to a Room, and the Room is connected to a Device. In addition, the embedded group will also send the device the measurements are coming from,

and which room the measurements is measured. The DAO consists of the repositories. These classes allow users to perform queries such as save, load, delete etc. Inside the Mongo package a parallel model that maps to the mongodb and its repository is located. Lastly, there is the Services package. Here all the classes that perform the desired business logic is located.

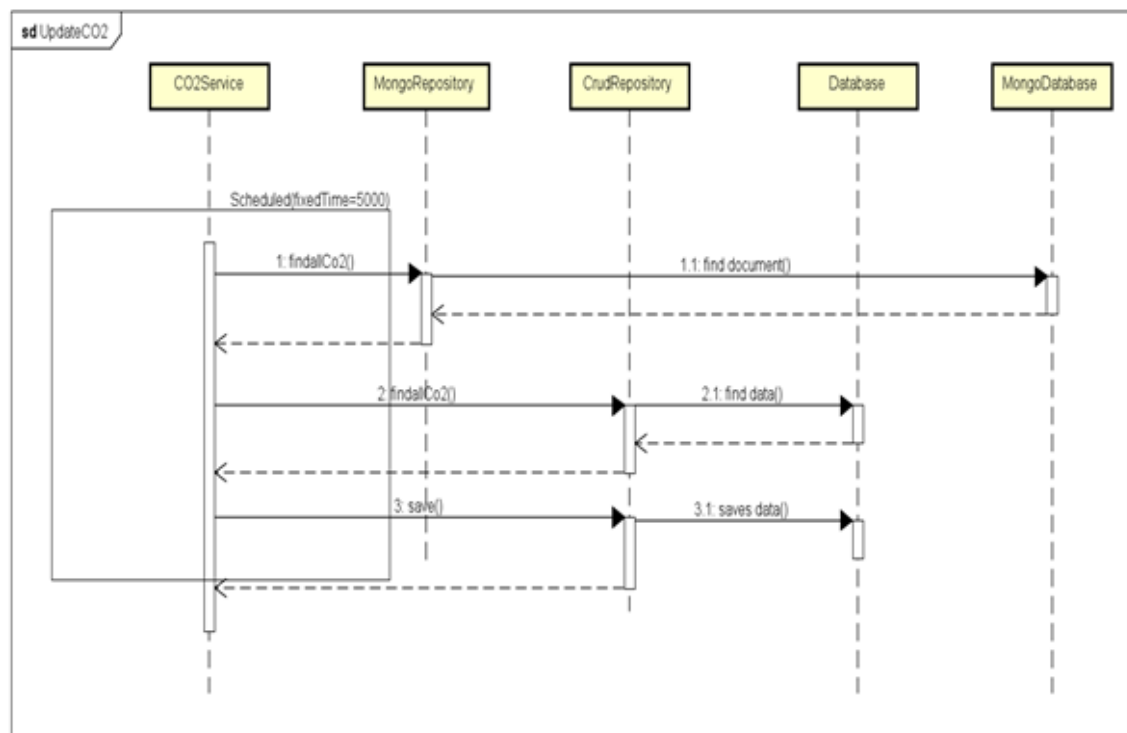


Figure 18 - Sequence Diagram of UpdateCO2

The sequence for the following method UpdateCO2() is illustrated above. First the method is run by CO2Service where it retrieves all the desired data from mongodb using the inbuilt method in Spring findAll(). This returns a list with all the data from the Mongoddb. Once the findAll() is called the MongoRepository that maps to the Mongoddb documents, will retrieve all the JSON objects and map them to Java CO2 objects. Finally, a list of the created CO2 objects is returned by the MongoRepository. Next, the method findAll() will be called again, but this time to the CrudRepository that maps to the CO2 table in the SQL Server. The CrudRepository will find data from the database and put them inside Java CO2 objects. The CrudRepository will then return a list with all CO2 object present in



the database. Next the method performs a comparison between the list using the timestamp to check for new readings from mongodb to be inserted into the database. The logic is performed using a for loop through all the objects from mongodb and comparing the timestamp with the last object from the database. If the CO2 objects located in mongodb has a timestamp that is after the timestamp of the last object in the database, it will be saved into the database. This allows the application to not only to load all documents from mongodb into the database, but also update the new ones.

#### **4.2.9. Web application**

When making the system the team decided to include rooms in the database. This was done to keep track of which specific room the current measurements were coming from and to create a report for the measurements in a given room. However, this meant that the room would be specified with an external system, since there would otherwise be no way to know which room the measurements were coming from. To accomplish this an external web application was made that would allow the admins to login using authentication and add, edit, view and delete a room. The application also allows the feature to add customers and future users, other admins and devices. In the future this application could be scaled to make the admin manually create new users to be used as authentication for the app as well. The application was built with razor pages using .net and C#. The entity framework core was used to map to the database and created models for the database entities. This allows generating controllers for the models efficiently. C# and .net also compliments the Microsoft SQL server, which was convenient.

Figure 19 - Web Application Login Form

The application consists of a several tabs that allows the user to add, edit, view and remove users, admins, rooms, devices. Lastly, there is also a login and logout tab.

room_name	device_name	
F307	SEP4_DEVICE	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
F306	SEP4_DEVICE	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
E303	SEP4_DEVICE	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
E500	SEP4_DEVICE	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
E600	SEP4_DEVICE	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

Figure 20 – Web Application List of rooms

This snippet above illustrates page for viewing the rooms. The admin can from this page also directly edit and delete the rooms.



**Application name**

[Home](#) [CreateUser](#) [ViewUsers](#) [CreateAdmin](#) [ViewAdmin](#) [CreateRoom](#)

room

room\_name

device\_id

SEP4\_DEVICE ▼

Create

[Back to List](#)

© 2019 - My ASP.NET Application

*Figure 21 - Web Application Create Room*

In the snippet above the page for adding room is illustrated. Since device is a foreign key to room there is drop down list to pick from existing devices.

### 4.3. Android

#### 4.3.1. Android Class Diagram

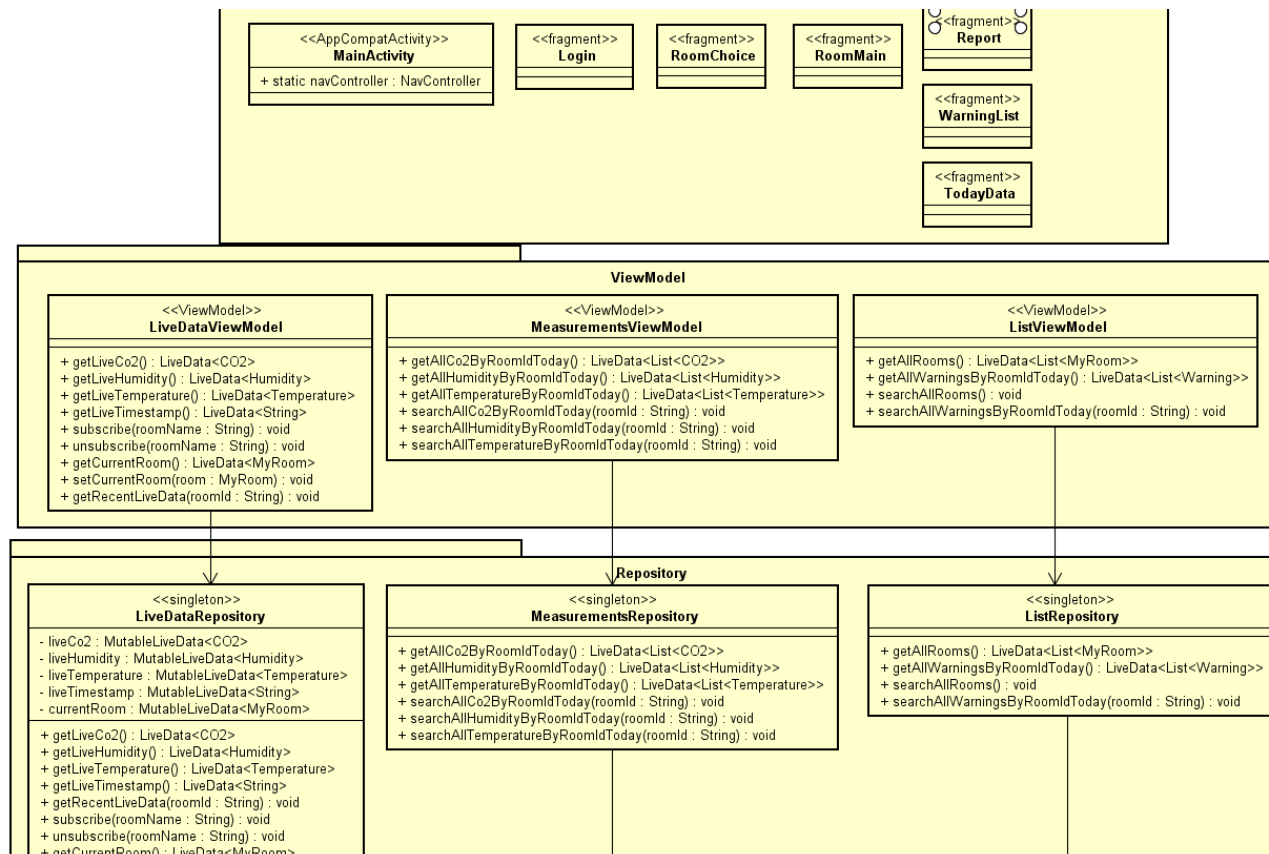


Figure 22 - 1st Part of Android Class Diagram

This is the first half of the full android class diagram. The main focus of the architecture is to maintain structure in the code. Firstly, MVVM is used for the project with the help of in-built Lifecycles library to maintain data in the application after configuration changes occur in the life cycles. Lifecycles library includes LiveData class object which implements the observer pattern. The observers are located in the fragments which are observing the data changes, and if there are changes, the observer is notified about them and updates the views. The repository pattern is used to abstract the calls to the webservice and Firebase

Cloud Messaging which allows following the Single Source of Truth principle which defines one place to return data from multiple data sources.

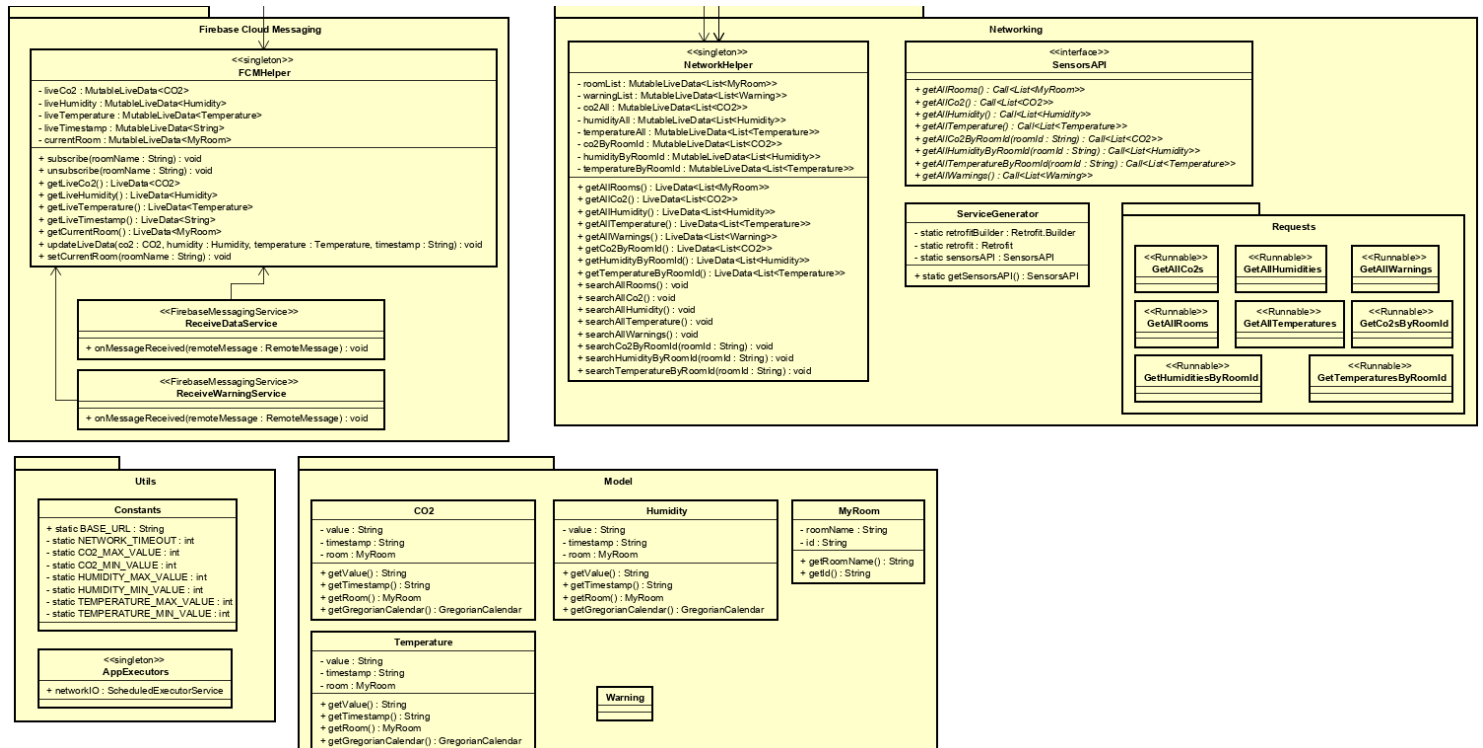


Figure 23 - 2nd Part of Android Class Diagram

This is the second half of the class diagram. The Sensors API interface contains network calls which are later initialized in one of the Requests Runnables classes with the help of Retrofit2 library. Network Helper class and FCMHelper contains MutableLiveData objects which contains data sent from web services or Firebase Cloud messaging and once MutableLiveData changes, the observers in the view get notified about the changes. The system has two data sources: Web Services and Firebase Cloud Messaging. Web services expose lists of co2, temperature and humidity of today and Firebase Cloud Messaging helper receives live data over the Firebase servers about current data of sensors.

#### 4.3.2. Android Sequence Diagram

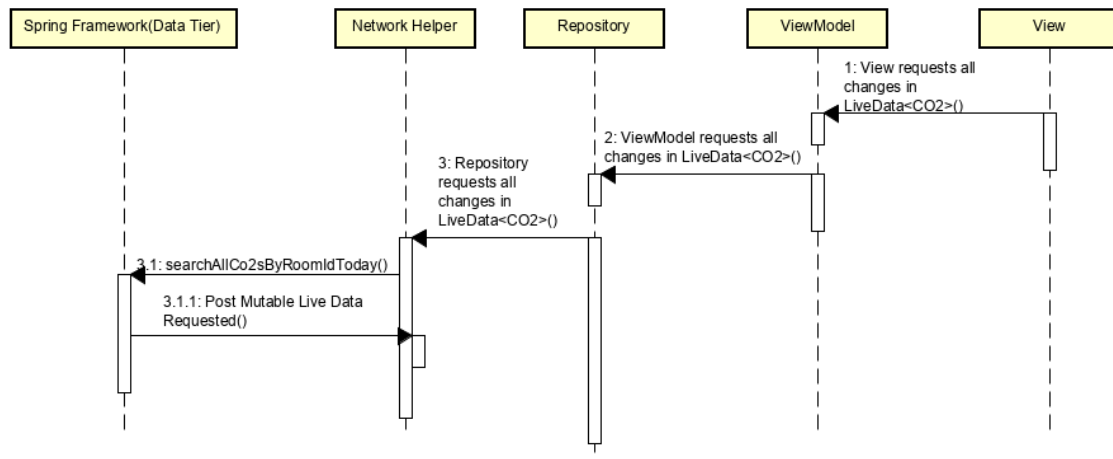


Figure 24 - Android Sequence Diagram

In the figure above, the sequence diagram is of a manual request from the view to the Spring framework to retrieve today's data of CO2. The view observes changes of LiveData CO2 from the ViewModel. The ViewModel observes the same from the Repository, and the Repository observes from the Network Helper which is updated by searchAllCo2sByRoomIdToday(roomId : String roomId) method which makes a network request to Spring framework. The framework gives a response that is later posted inside the Mutable Live Data.

#### 4.3.3. Android Architecture

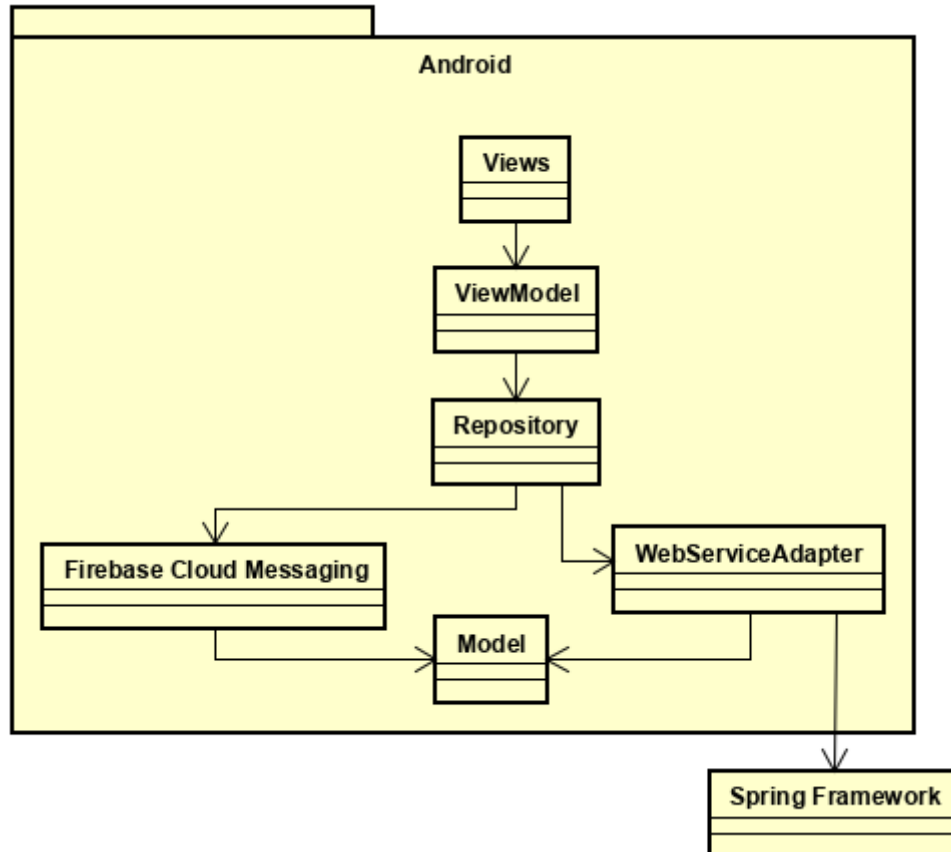


Figure 25 - Android Architecture Diagram

The figure above displays a simplified version of Android App Architecture. This diagram serves for a strong base of structure, implementing well known design patterns such as MVVM, Repository, Singleton and Observer.

#### 4.3.4. Android Technologies

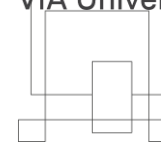
The android app is using external libraries to help abstract and structure code.

**Retrofit2** - a library that gives a layer of abstraction to the code for handling web service calls.

**Firebase Cloud Messaging** - a library that is automatically subscribed to the data from sensors. This library helps the app to receive current data from the sensors.

**Navigation Architecture Components** - a library that enforces the single-activity architecture by using Fragments instead of activities to display the view.

**ButterKnife** - a view binding library that provides more elegance to the code.



**Lifecycles** - a library that helps to manage lifecycles of the android application with Live Data and View Model. Live Data implements the observer pattern, which automatically notifies the view to update if there are any changes to what it is observing, and View Model maintains data even when configuration changes occurs for android app.

**RecyclerView** - a library that recycles a list of items to prevent loading of hundreds of items in the list and only forces to load items that are currently being viewed by the user.

**Custom Gauge** - a library that creates a custom gauge in the view of a fragment.

**MPAndroidChart** - a chart library which allows creating different types of charts for displaying co2, humidity and temperature data.

**Firestore Authentication** - a library that handles the authentication process and provides an end-to-end identity solution, supporting email and password accounts.

#### 4.3.5. Android Design Patterns

The project is using 4 distinct design patterns.

**MVVM design pattern (Model, View, ViewModel)** - This is the core structure of the app. MVVM helps maintain integrity, scalability, testability and structure throughout the code of the app. The main focus of the MVVM design pattern are the View Models. The View Models help maintain data that is displayed during the activities and fragments lifecycles. The data will not be lost if for example the user rotates his phone, which destroys the activity and restarts it.

**Observer design pattern** - The observer design pattern is implemented in the Live Data library. This pattern helps the view to observe specific data constantly. When the observed data has changed in anyway, the observer is notified of the change, which then notifies the view of the change, and the view updates accordingly to the change.

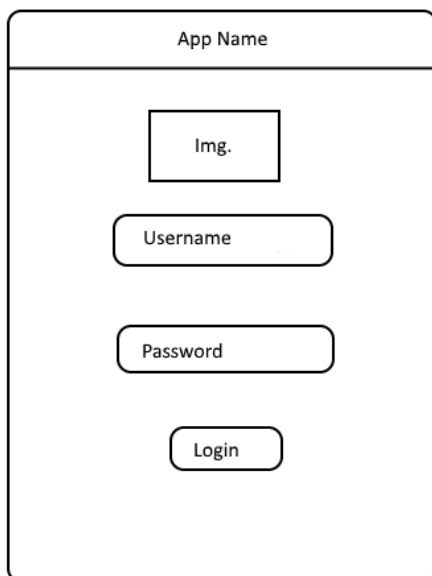
**Repository pattern** - Design patterns that helps to have one access point in code to control multiple data sources. By following this pattern, it is possible to integrate single source of truth principle to have one place for returning data from multiple data sources.



**Singleton pattern** - Ensures that the system creates only one class object without creating multiple instances. It is widely used in the system so many helper classes do not create itself referencing incorrect locations in the code.

#### 4.3.6. UI Design Choices

The main focus in the choices of the UI is the single-activity architecture, that is why all of the Views are Fragments. Fragments give a smoother transition between Views for the app and they are able to share Views. For the single-activity architecture, the views are implemented using the Navigation Architecture Component.

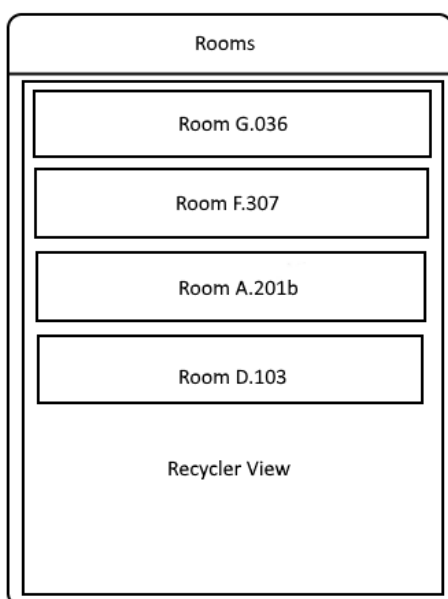


The Login Fragment UI design is a vertical rectangle. At the top is a header bar labeled "App Name". Below the header is a large rectangular area containing four elements: a square placeholder labeled "Img.", a rounded rectangular input field labeled "Username", another rounded rectangular input field labeled "Password", and a rounded rectangular button labeled "Login".

Figure 26 - Login Fragment

#### Login Fragment

The front-end is designed in a simplistic manner. There is no Register button because the users are added in manually so that the right people can access the contents of the app. After a successful login, the user is taken to the Room Choice Fragment.



The Room Choice Fragment UI design is a vertical rectangle. At the top is a header bar labeled "Rooms". Below the header is a list of four room names, each in its own rectangular box: "Room G.036", "Room F.307", "Room A.201b", and "Room D.103". Below these boxes is a larger rectangular area labeled "Recycler View".

Figure 27 - Room Choice Fragment

#### Room Choice Fragment

After the successful login, the user is taken to the Room Choice Fragment, which uses a Recycler View to display the rooms that are clickable on. After the user has picked out and clicked the room in the view, he is taken to the Room Main Fragment.

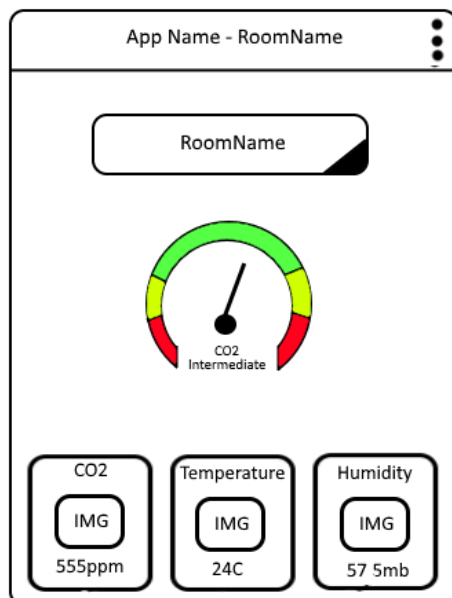


Figure 28 - Room Main Fragment

### Room Main Fragment

After a room is chosen, the user is sent to the Room Main Fragment, where all the live data is displayed. This fragment has a custom toolbar, that will contain buttons. The buttons are Report List, Warning List and Log Out. The first two buttons when clicked on, send the user to the appropriate section. In the middle the name of the room is displayed, and a Spinner attached to it. It will allow to change rooms on the spot. In the middle there is a gauge that displays relevant live data. Below the gauge, there are 3 measurements displayed and the images corresponding them are clickable. After an image is clicked on, the gauge changes and displays relevant live data regarding the image clicked on. If the Temperature image is clicked, the gauge will adjust to that.

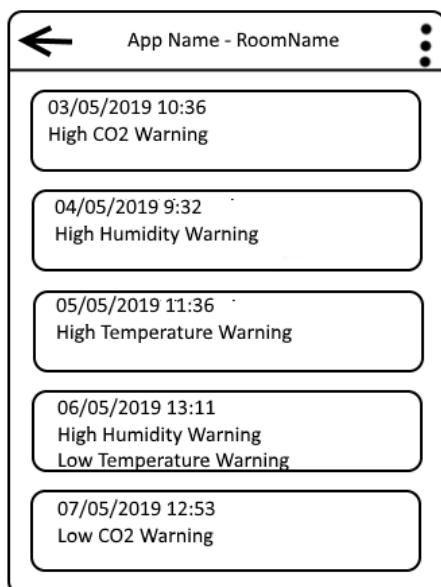
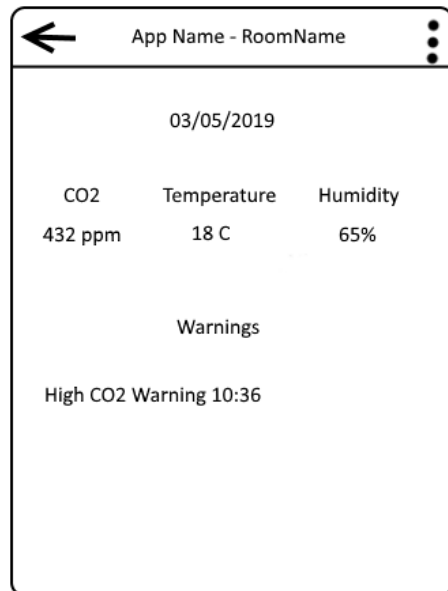
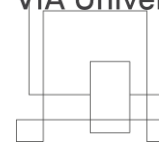


Figure 29 - Warning List Fragment

### Warning List Fragment

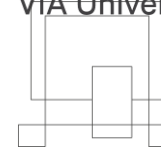
If the user selected Warning List from the previous fragment, he will be sent to this view that is a RecyclerView that contains data about the warnings. The data displayed is a history of warnings that are clickable. In this fragment, the custom toolbar also has the back button, to return to the Room Main Fragment. Also, the same attributes apply for the toolbar, it has the Report List and Logout buttons.



## Report Fragment

After the user has selected a report from the list in the previous fragment, the app shows this fragment, which displays detailed data in that timestamp. Any warnings that happened in that timestamp are shown below the measurements and at which time they happened. The toolbar has the same functionality as in the Warning Fragment.

*Figure 30 - Report Fragment*



## 5. Implementation

### 5.1. Embedded

#### 5.1.1. C Code:

After creating the class diagrams, The system was implemented in C language using Atmel Studio 7. The project was provided with a number of drivers which have been included in the project, the second step was to implement the interfaces for each class which are: CO2Sensor, temp\_humSensor and MyLora.

Each interface has a method to create and initialize the driver, the driver has to be created once for each sensor and for the Lora Wan server.

In the CO2Sensor interface, the method for initializing the driver has two parameters: the first parameter is the USART port the MH-Z19 sensor is connected to - in this case USART3.

The code below shows the initialization of CO2 sensor driver:

```
// create drivers
void co2_sensor_init() {
    mh_z19_create(ser_USART3, co2_callback);
}
```

*Figure 31 - CO2 Sensor\_INITIALIZER*

The call back function will be called by the driver when a new CO2 value is returned by the sensor. So, the second parameter is the address of the call back function.

The second method is to measure the CO2 value which can be performed after creating and initializing the driver.

And the last method is the get value method which is returning the CO2 value.

The screenshot below shows the methods used in the CO2Sensor interface:



```
#define CO2SENSOR_H_
#include "ATMEGA_FreeRTOS.h"
#include <stdbool.h> #include <stdint.h> #include <stdio.h> #include <stdlib.h> #include <serial.h> #include <mh_z19.h>

void co2_sensor_init();
void co2_callback(uint16_t co2_ppm);
void co2_measure();
uint16_t co2_get_value();

#endif /* CO2SENSOR_H_ */
```

Figure 32 - CO2 Sensor Interface

In temp\_humSensor class, the same steps were followed as in CO2Sensor class, in order to perform a measuring for temperature and humidity, the sensor must be woken up from power down, after the [hih8120Wakeup\(\)](#) call, the sensor will need a minimum of 50 ms to be ready and to give the results.

The code below shows the method which is responsible for measuring the temperature/humidity in a specific room:

```
// measuring function
void temp_hum_measure() {
    hih8120DriverReturnCode_t rc;

    vTaskDelay(1000/portTICK_PERIOD_MS);
    if ( HIH8120_OK != ( rc = hih8120Wakeup() ) )
    {
        printf("temp_hum_SENSOR_ERROR --> %d", rc);
    }
    //printf("temp_hum_rc --> %d", rc);
    vTaskDelay(50/portTICK_PERIOD_MS);
    if ( HIH8120_OK != ( rc = hih8120Measure() ) )
    {
        printf("temp_hum_SENSOR_ERROR1 --> %d", rc);
    }
    //printf("temp_hum_rc --> %d", rc);
    vTaskDelay(1000/portTICK_PERIOD_MS);
}
```

Figure 33 - Temperature/Humidity Measure Method

In LoraWan module the method to create and initialize the driver has only one parameter, which defines the port to be used for communication with the RN2483 module, in this case the port number 1 was used.

After implementing the necessary methods for sensors and Lora Wan module, The next step was to schedule tasks to perform measurements. That is why three different classes were created to execute tasks: myTasks, myTimers and mySemaphores.

The semaphores and timers are used to achieve tasks synchronization in the multiprocessing environment.

All the tasks have the same priority.

The timer cycle is set to 10 minutes and between each task and the next task there is a timer set to 1 minute.

### 5.1.2. The WebSocket Application (Web Socket client)

The web socket client bridge application was implemented in Java. Because of libraries used for this project(java.net.http) , Java 11 (sdk-11 or above) is needed. It is the first version, where these new libraries were added.

```
public LoRaClient() {
    HttpClient client = HttpClient.newHttpClient();
    CompletableFuture<WebSocket> ws = client.newWebSocketBuilder()
        .buildAsync(URI.create("wss://iotnet.teracom.dk/app?token=vnoRdgAAABFpb3RuZXQuOGVybWVkbS5kZmV5LkZlbnRlbnQ="), this);
    .buildAsync(URI.create("wss://iotnet.teracom.dk/app?token=vnoRdgAAABFpb3RuZXQuOGVybWVkbS5kZmV5LkZlbnRlbnQ="), this);

    mongoHandler = new MongoDBHandler();
    gson = new Gson();
}
//onOpen()
public void onOpen(WebSocket webSocket) {
    // This WebSocket will invoke onText, onBinary, onPing, onPong or onClose methods on the associated listener (i.e. receive methods) up to n more times
    webSocket.request(1);
    System.out.println("WebSocket Listener has been opened for requests.");
}
```

Figure 34 - Lora Connection Request

After opening connection between web socket server and this client, program runs in an infinite while loop, while listening to the server and waiting for a message.

```
//onText()
public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {
    //System.out.println(data);
    LoRaMessage message = gson.fromJson(data.toString(), LoRaMessage.class);
    mongoHandler.insertDoc(message.transformToMongo());
    webSocket.request(1);
    return null; // new CompletableFuture().completedFuture("onText() completed.").thenAccept(System.out::println);
};
```

Figure 35 - LoraMessage Serialization

When the application receives the message, message is serialized to LoRaMessage and transformed into mongoDb document and immediately sent to the database to be stored through mongoHandler, which was created in LoRaClient constructor.



```
public Document transformToMongo() {  
    if(!cmd.equals("rx"))  
        return null;  
  
    return new Document("UIE", EUI)  
        .append("Room", 1) // change room to room id from sql server  
        .append("Name", "RAND_NAME")  
        .append("Timestamp", new Date(timestamp))  
        .append("CO2", getCO2())  
        .append("Humidity", getHum())  
        .append("Temperature", getTemp());  
}
```

Figure 36 - LorA Message Parsing For MongoDB

```
public String getHum() {  
    return Integer.parseInt(data.substring(0, 4), 16)+"";  
}  
  
public String getTemp() {  
    return Integer.parseInt(data.substring(4, 8), 16)+"";  
}  
  
public String getCO2() {  
    return Integer.parseInt(data.substring(8, 12), 16)+"";  
}  
}
```

Figure 37 - Parsing Methods

Before sending document into the MongoDB database, a message from the websocket server must be parsed and data must be decoded. Since the data comes in one string containing all measurements in hexadecimal format, they need to be split each by 4 bytes and decoded to decimal numbers.



## 5.2. Data

### 5.2.1. Java Extractor

Spring framework is an application framework for the Java platform. It supports a lot of other popular frameworks and hence called the framework of frameworks. Some of its supported libraries include SQL DAO mapping, MongoDB document mapping, Scheduling and many more. Because of those supported features, building the application would be significantly easier than otherwise. First SQL DAO mapping was used to create/map to the database from code using annotations.

```
@Entity
@Table(name = "Co2")
public class Co2 {
    |   @Id
    |   @GeneratedValue(strategy = GenerationType.IDENTITY)
    |   @Column(name = "Id")
    |   private int id;

    |   @ManyToOne
    |   @JoinColumn(name="room_id")
    |   private Room room;

    @Column(name = "status")
    private String status;

    @Column(name = "value")
    private double value;

    @Column(name = "date")
    private LocalDate date;

    @Column(name = "timestamp")
    private Timestamp timestamp;
```

Figure 39 - CO2 Class

```
@Document(collection = "EUI")
public class EUIMongo {

    @Id
    private ObjectId _id;

    @Field("UIE")
    private String uie;

    @Field("Name")
    private String name;

    @Field("RoomId")
    private int roomId;

    @Field("Timestamp")
    private Date timestamp;

    @Field("CO2")
    private String co2;
```

Figure 38 - EUIMongo Class

In the code snippets above the CO2 object maps/create a corresponding table in the database using the annotations. First the specific entity is specified above the class followed by the columns for each instance variables. Spring also supports a

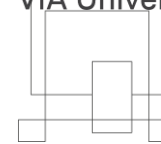


similar feature for mongodb. In the snippet besides the first one an EUIMongo class is created. This class will map to mongodb document and the specific collection. Here annotations are used to specifying the collection and the fields from mongodb. In order to map to the right mongo server and database a connection string is set up in the application.properties file beforehand. When this is done the respective repositories are made for these classes. The repositories responsibility is to handle the queries. Another important feature from Spring was scheduling. Scheduling allows the users to schedule methods for specific intervals. An example could be a method would be run every 5 seconds while the application is running, or conversely and initial delay could also be added as well for 10 seconds. This feature came in handy when checking for new updates in mongodb that would be added to the database.

```
51 @Scheduled(initialDelay = 1200, fixedRate = 5000)
52 public void updateCO2() {
53     EUI = er.findAll();
54     if(EUI != null && co2.findAll() != null) {
55         if(co2.findAll().size() != 0) {
56             tl = co2.findAll().get(co2.findAll().size() - 1).getTimestamp();
57         } else {
58             tl = new Timestamp(0);
59         }
60         try {
61             for(int i = 0; i < EUI.size(); i++) {
62                 t = new Timestamp(EUI.get(i).getDate().getTime());
63                 ld = mm.parse(strDate = mm.format(EUI.get(i).getDate())).toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
64                 Co2_value = EUI.get(i).getCo2();
65                 co2IntValue = Double.parseDouble(Co2_value);
66                 timestamp = t;
67                 room = rr.findAll().get(EUI.get(i).getRoomId());
68                 co2New = new Co2(co2IntValue, ld, timestamp, room);
69
70                 if(t.after(tl)) {
71                     co2.save(co2New);
72                 } else {
73                     System.out.println("no update");
74                 }
75             }
76         } catch (Exception e) {
77             e.printStackTrace();
78         }
79     }
```

Figure 40 - UpdateCO2 Method

In the snippet above the main logic of the application is illustrated. In this example updateCO2() is used. As mentioned before this method is scheduled using the annotation shown. A 1200 milliseconds delay from when the program starts and a fixedRate where the method reruns at 5000 milliseconds is set. In line 53 the



EUI(MongoRepository) repository is set, and the `findAll()` is called giving all the objects in the `MongoRepository` as a list. This is also done for CO2 with the `co2.findAll()` that would give all the objects in the database as a list. In the next line an if-statement is made to check if there is and any values in `mongodb` or the database. Next on line 55 the CO2repository is checked to see if there are any values here. This is done in order to check for the timestamp of the last CO2 object in the database. If the database is empty a default timestamp set at 0 would be used, else the timestamp from the last object would be used. Now all the new objects from `mongodb` need to be initialized inside the CO2 object using the for loop that begins at line 60. This loop runs through all the objects in `mongodb` and initializes them into a CO2 object. At line 69, an if-statement checking if the timestamp for the object from `mongodb` is after the timestamp from last object in the database is made. If the statement is true, the object would be added to the database. Hence this method will prevent adding duplicates to the database.

### 5.2.2. Web Application

The application is secured by Authentication and Authorization. To authenticate a user into the application the user must be an admin. The admins to authenticate are taken from the SQL server that are used for the system. The pages are all authorization as well. In other words, only the admin can enter and modify the data. To create security the .net library `System.Web.Security`; is used, which allows the authentication of users with forms.

```
19 [HttpPost]
20 public ActionResult Login(admin a)
21 {
22     Sep4_GroupX2Entities db = new Sep4_GroupX2Entities();
23     var count = db.admins.Where(x => x.username == a.username && x.password == a.password).Count();
24     if (count == 0)
25     {
26         ViewBag.Msg = "Invalid User";
27         return View();
28     }
29     else
30     {
31         FormsAuthentication.SetAuthCookie(a.username, false);
32         return RedirectToAction("Index", "Home");
33     }
34 }
35
36 // GET: Account
37 public ActionResult Logout()
38 {
39     FormsAuthentication.SignOut();
40     return RedirectToAction("Index", "Home");
41 }
42 }
43 }
```

Figure 41 - Web App Authentication

The snippet above shows the method that authenticates if the admin is from the current system. At line 22 the database is initialized. The following line creates a count that checks if the admin is inside the database. If count==0, the user gets returned to the home page and won't have access to the authorized features, but if the count is 1, FormsAuthetication is set to the user, and they are now able to login and have access to the authorized features. Once logged in the admin will by default be redirected to the home page, but they will be able to enter all the authorized tabs. To logout the FormAuthetication is set to SignOut() and the user is redirected to the home page.

```
[Authorize]
public class roomsController : Controller
{
    private Sep4_GroupX2Entities db = new Sep4_GroupX2Entities();

    // GET: rooms
    public ActionResult Index()
    {
        var rooms = db.rooms.Include(r => r.device);
        return View(rooms.ToList());
    }

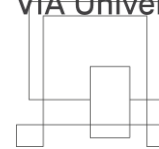
    // GET: rooms/Details/5
    public ActionResult Details(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        room room = db.rooms.Find(id);
        if (room == null)
        {
            return HttpNotFound();
        }
        return View(room);
    }
}
```

Figure 42 - Web App Controller

In the snippet above the room controller is illustrated. This class is responsible for creating the endpoints for the controller. It is also marked with an annotation above the class called authorize, to prevent unauthenticated users to enter the endpoints. The razor pages also creates a view for the controllers, which displays the controllers view once the endpoint is accessed. The view is made using razor annotation, which is a .net language that is a combination of html, css, and some new razor features.

### 5.2.3. Data warehouse

First step to implement the data warehouse was creating the star schema tables which are the four fact tables (CO2, humidity, temperature and warning), also the dimensions (Room, time, calendar).



Second creating the staging area where the data was extracted from the source code. Surrogate keys were generated, null values were deleted from the room dimension.

```
-- filtering the data and remove NULL values if it exists
UPDATE [DW_STAGING].[dbo].[TEM_FACT_CO2] SET value = 'NO VALUE' WHERE value IS NULL;

-----populate to the fact table

select * from [TEM_FACT_CO2];
```

Figure 43 - Removes null values from temporary fact tables

Regarding the temporary fact tables, it has the surrogate keys for each dimension the business key and the value for the measurement. Date and time attributes were used to update the surrogate keys in the temporary tables. After populating the tables, the data got cleaned and loaded into the fact tables.

```
go
-----update
)UPDATE [TEM_FACT_CO2] SET D_ID =(select D_ID from [DW].[dbo].[Calendar_D] where [DW].[dbo].[Calendar_D].CalendarDate = TEM_FACT_CO2.date)
UPDATE [TEM_FACT_CO2] SET T_ID =(select T_ID from [DW].[dbo].[Time_D] where [DW].[dbo].[Time_D].TimeStamp = TEM_FACT_CO2.time)
UPDATE [TEM_FACT_CO2] SET R_ID =(select R_ID from [DW].[dbo].[Room_D] where [DW].[dbo].[Room_D].Room_ID = TEM_FACT_CO2.Room_ID)
```

Figure 44 - Replacing the surrogate keys in the temporary fact table with the primary keys from each dimension

### Updating the data warehouse with the new data:

the code snippet below was used to load the dimension with the new added data from the source code each day using SQL Server Job (Scheduling)

```
use Sep4;

insert into [DW].[dbo].[Room_D]
(
    Room_ID, room_name, device_name, ValidFrom, Validto )
select
    a.id, a.room_name , b.device_name, '05/02/2019', '01/01/2099' -- LAST UPDATE +1
    FROM Sep4.dbo.room a JOIN Sep4.dbo.device b
    ON a.device_id = b.id

where a.id in
((
    --- today
    select [id]

from [Sep4].[dbo].[Room]
)

EXCEPT

--- yesterday

( select Room_ID from [DW].[dbo].[Room_D]
)
)
```

Figure 45 - Loads the dimension with the new data

## 5.2.4 Triggers

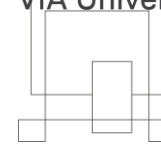
```
-- This trigger insert a row in the warning table if the status field is either LOW or HIGH
ALTER TRIGGER [dbo].[TemperatureWarningConfigureStatus]
ON [dbo].[temperature]
AFTER UPDATE
AS
BEGIN
    -- Declaring the variables
    DECLARE @status AS VARCHAR(15)
    DECLARE @timeStamp AS DATETIME
    DECLARE @value AS INT
    DECLARE @roomID AS INT
    DECLARE @type AS VARCHAR(10) = 'Temperature'
    DECLARE @date AS DATE

    -- assigning values
    SELECT @status = status, @timeStamp = timestamp, @value = value, @roomID = room_id , @date = date
    FROM Inserted

    -- checking the condition and inserting if true
    IF (@status = 'HIGH' OR @status = 'LOW')
        INSERT INTO dbo.warning ( [measurement_type] ,[status] ,[timestamp] ,[value] ,[room_id] ,[date]) values (
            @type, @status, @timeStamp, @value, @roomID, @date)
END
```

Figure 46 – Temperature Trigger

The database has 6 triggers that set the status field in co2, humidity and temperature based on their recorded value. For temperature table, the trigger will update the status field to HIGH when it detects a temperature value over 30C. After updating the status field, another trigger will run to insert this particular row into the warning table.



```

ALTER TRIGGER [dbo].[TemperatureConfigureStatus]
ON [dbo].[temperature]
AFTER INSERT
AS BEGIN
    -- Declaring variables
    DECLARE @status AS varchar(15)
    DECLARE @max AS INT = 35
    DECLARE @min AS INT = 0
    DECLARE @originalValue AS INT
    -- set the default value
    UPDATE Co2 SET status = 'NORMAL'
        WHERE Co2.id IN (SELECT id FROM Inserted)
    SELECT @status = status
    FROM Inserted
    SELECT @originalValue = value
    FROM Inserted
    -- update the default value based on the condition
    IF (@originalValue > 30)
        UPDATE [temperature]
        SET status = 'HIGH'
        WHERE [temperature].id IN (SELECT id FROM Inserted)
    ELSE IF (@originalValue < 0)
        UPDATE [temperature]
        SET status = 'LOW'
        WHERE [temperature].id IN (SELECT id FROM Inserted)

END

```

Figure 47 - Updates Warning Table



## 5.2.5 Web Service

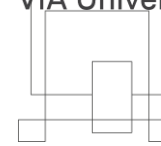
```
@Scheduled(fixedDelay = 100000,initialDelay = 60000)
@Async
public void send() {
    if (isEnabled) {
        DTOObject dto = getDataForTheTopic();
        if (dto != null) {
            JSONObject body = new JSONObject(); body.put("to", "/topics/" + topic); body.put("priority", "high");
            JSONObject data = new JSONObject(); data.put("co2_value", dto.getCo2()); data.put("hum_value", dto.getHumidity());
            data.put("temp_value", dto.getTemperature()); data.put("timestamp", dto.getTime().toString());
            data.put("topic", this.topic); body.put("data", data);
            HttpEntity<String> request = new HttpEntity<>(body.toString());
            CompletableFuture<String> pushNotification = init(request);
            CompletableFuture.allOf(pushNotification).join();}}}

@Async
public DTOObject getDataForTheTopic() {
    Co2 co2 = co2Repository.findTopByRoomOrderByIdDesc(this.room);
    Temperature temperature = temperatureRepository.findTopByRoomOrderByIdDesc(this.room);
    Humidity humidity = humidityRepository.findTopByRoomOrderByIdDesc(this.room);
    if (co2 != null && temperature != null && humidity != null) {
        DTOObject dto = new DTOObject(co2.getValue(), temperature.getValue(), humidity.getValue(),
            co2.getTimestamp());
        System.out.println(dto.toString());
        return dto;
    }
    return null;
}
```

Figure 48 - NotificationsService Class Schedule Example

The code snippet above has been taken from the NotificationsService class, it declares a scheduled component provided by spring core. It schedules a job with an initial delay of 60 seconds, and a fixed delay of 100 second. The method starts its work when the isEnabled variable is set to true, and that happens through an HTTP GET call to the FCM controller. The caller will send a topic that he wants to subscribe to, in this case the topic is a room name. Based on the room name the caller will receive notifications about the live data in that particular room. A helper method called getDataForTheTopic will package the data from the database into a data transfer object. All the calls to the database is managed by a connection pool manager, moreover both methods are annotated with the @Async annotation which will allow multithreading for both methods. In other words, the application has the ability to handle multiple android devices at the same time, each of which will have its own thread, and the most important part is all the threads will use the same database connection provided by the connection pool manager. The notification or the DTO will be send to the firebase cloud message console which in turn will send the message as a notification to all those who has subscribed to listen to this particular topic.





### 5.3. Android

#### 5.3.1. Single Activity Principle

Recently Google<sup>1</sup> started recommending single activity principle which follows having only one activity in the entire application. Such methodology helps to have more control over user interface lifecycle with the help of Navigation Architecture Component released in Android Jetpack.

```
navController = Navigation.findNavController( activity: this, R.id.nav_fragment);  
AppBarConfiguration appBarConfiguration = new AppBarConfiguration.Builder(R.id.loginFragment, R.id.roomChoiceF  
NavigationUI.setupWithNavController(toolbar, navController, appBarConfiguration);  
NavigationUI.setupWithNavController(bottomNav, navController);  
navController.addOnDestinationChangedListener(navDestinationListener());
```

*Figure 49 - Navigation Controller Usage in MainActivity*

Figure above displays how simplified Android navigation is by using 5 lines of code. Toolbar is configured correctly to have accurate back stack navigation and bottom navigation view is included into MainActivity. Navigation is done by using only one line of code where the only requirement is to locate action created by Navigation component UI inside of Android Studio interface.

```
switch(id){  
    case R.id.action_today_data: {  
        navController.navigate(R.id.action_global_todayDataFragment);  
    }  
}
```

*Figure 50 - Navigation Example When Toolbar Menu Item is Clicked*

All of the navigation is managed inside of Android Studio where it is possible to create actions which navigates from one fragment to another.

---

<sup>1</sup> <https://youtu.be/2k8x8V77CrU>

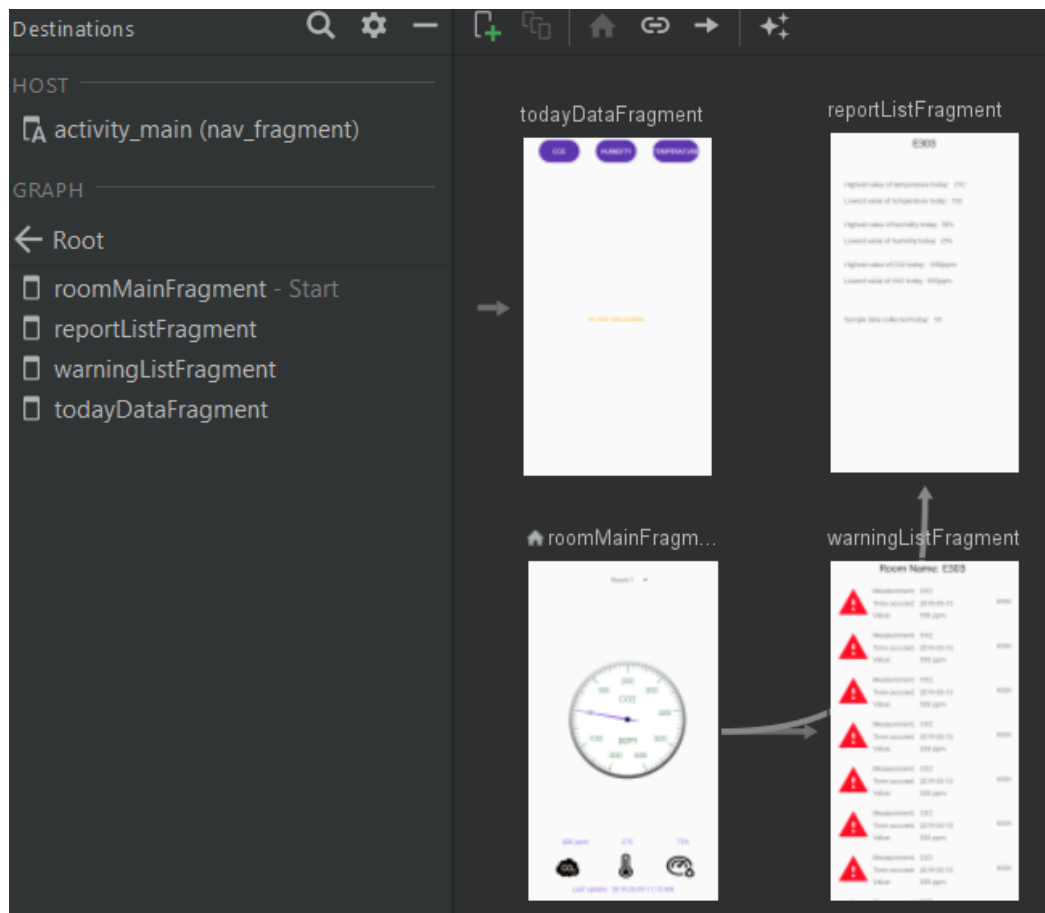


Figure 51 - Navigation after logging in and selecting a room inside of Android app

With the help of Navigation Architecture component it is easier to control the flow of fragments. There is no need to handle fragment transactions or back stack. Everything is done by the library in the NavHostFragment that is an already made Fragment in the library.

### 5.3.2. Web Services (Network Helper)

Networking is done using Retrofit2 library that helps to create web API requests easier. Each network request has an individual class that implements Runnable for threading. This is done for scalability and future reasons of better error handling. Also such code structure gives better overview of the codebase.

```
public class GetAllCo2sByRoomIdToday implements Runnable{

    private ServiceGenerator sg;                //Returns Call for Retrofit Response
    private MutableLiveData<List<C02>> data;      //LiveData reference from NetworkHelper
    private String TAG;                          //TAG for debugging
    private String roomId;                      //Room id reference for API

    public GetAllCo2sByRoomIdToday(String tag, MutableLiveData<List<C02>> list, String roomId){
        this.data = list;
        this.TAG = tag;
        this.sg = ServiceGenerator.getInstance();
        this.roomId = roomId;
    }
}
```

Figure 52 - GetAllCo2sByRoomIdToday Class

Figure above displays what each runnable has inside of it. ServiceGenerator is used to get API calls by creating a Singleton reference to Retrofit2 instance which initializes SensorsAPI interface that uses Retrofit2 annotations for preparing API request.

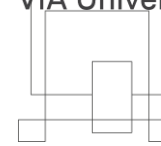
```
/*
 * C02
 */

@GET("co2/roomtoday/{id}")
Call<List<C02>> getAllCo2ByRoomIdToday(@Path("id") String roomId);

@GET("co2/room/{id}")
Call<List<C02>> getAllCo2ByRoomId(@Path("id") String roomId);
```

Figure 53 - SensorsAPI CO2 Requests

Following along all of these classes are connected and used in NetworkHelper which contains Mutable Live Data that is observed by the views and search methods which calls API and updates the Mutable Live Data.



```
public LiveData<List<C02>> getAllCo2sByRoomIdToday() { return co2ByRoomIdToday; }

public void searchAllCo2sByRoomIdToday(String roomId){
    //Networking Code
    if(getAllCo2sByRoomIdToday != null){
        getAllCo2sByRoomIdToday = null;
    }

    getAllCo2sByRoomIdToday = new GetAllCo2sByRoomIdToday(TAG, co2ByRoomIdToday, roomId);
    final Future handler = AppExecutors.getInstance().networkIO().submit(getAllCo2sByRoomIdToday);

    AppExecutors.getInstance().networkIO().schedule(() -> {
        handler.cancel( mayInterruptIfRunning: true);
    }, Constants.NETWORK_TIMEOUT, TimeUnit.MILLISECONDS);
}
```

*Figure 54 - GetAllCo2sByRoomIdToday Example in NetworkHelper Singleton Class*

In this figure Runnable class is initialized and later executed in Executor networkIO thread that is scheduled to run for 3000 milliseconds that is predefined in Constants.NETWORK\_TIMEOUT. When runtime of the thread surpasses 3000 milliseconds, thread is cancelled and here it is possible to catch a network timeout if web API request took too long to execute.

### 5.3.3. Firebase Cloud Messaging (FCMHelper)

Firestore Cloud Messaging solves the biggest issue the project had by allowing to listen to Firestore servers and receive data whenever Data tier posts it to a specific topic. Android tier subscribes to a topic named by the room name and listens for published data through `FirebaseMessagingService` extended class while Data tier is posting live data there. Android can receive the data and update the views accordingly to display current live data of the sensors. This is done by calling `FCMHelper` singleton's method `updateLiveData` and sending references of newly received data.

```
public class ReceiveDataService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        if(remoteMessage.getData().size() > 0){
            Map<String, String> map = remoteMessage.getData();
            String co2_value = map.get("co2_value");
            String hum_value = map.get("hum_value");
            String temp_value = map.get("temp_value");
            String timestamp = map.get("timestamp");
            MyRoom room = FCMHelper.getInstance().getCurrentRoom().getValue();

            CO2 co2 = new CO2(co2_value, timestamp, room);
            Humidity humidity = new Humidity(hum_value, timestamp, room);
            Temperature temperature = new Temperature(temp_value, timestamp, room);
            FCMHelper.getInstance().updateLiveData(co2, humidity, temperature, timestamp);
        }
    }
}
```

Figure 55 - ReceiveDataService Class

`updateLiveData` method then updates Mutable Live Data's of `FCMHelper` class, which are being observed by `RoomMain` fragment where the Gauge and live data of the sensors are displayed.

```
public void updateLiveData(CO2 co2, Humidity humidity, Temperature temperature, String timestamp){
    liveCo2.postValue(co2);
    liveHumidity.postValue(humidity);
    liveTemperature.postValue(temperature);
    liveTimestamp.postValue(timestamp);
}
```

Figure 56 - UpdateLiveData Method in FCMHelper class

#### 5.3.4. MVVM Structure

Android app is divided into many packages to follow MVVM structure and naming convention guidelines. There were two main packages: application which held adapters for recyclerviews, viewmodels and views that contain fragments. Business has data that has two additional packages (Firebase Cloud Messaging and Network Helper), models and repositories.

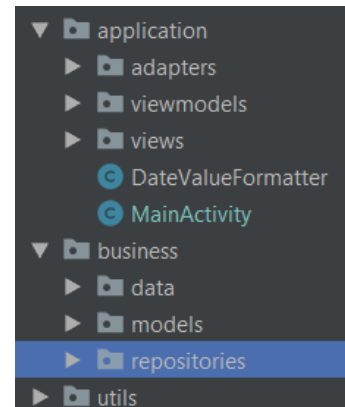


Figure 57 - MVVM Packaging Structure

MVVM pattern begins in the MainActivity where LiveDataViewModel is initialized

```
//Set up ViewModel
liveDataViewModel = ViewModelProviders.of( activity: this).get(LiveDataViewModel.class);
```

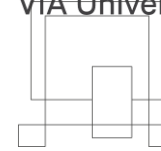
Figure 58 - LiveDataViewModel Initialization

MainActivity requires this view model in order to ensure when the user minimizes the application, he can successfully subscribe and unsubscribe from the Firebase Cloud Messaging so network resources are not wasted and subscription does not continuously work in the background.

```
@Override
protected void onResume() {
    super.onResume();
    if(LiveDataViewModel.getCurrentRoom().getValue() != null){
        MyRoom room;
        room = LiveDataViewModel.getCurrentRoom().getValue();
        LiveDataViewModel.subscribe(room);
    }
}

@Override
protected void onStop() {
    super.onStop();
    if(LiveDataViewModel.getCurrentRoom().getValue() != null){
        String roomName;
        roomName = LiveDataViewModel.getCurrentRoom().getValue().getRoomName();
        LiveDataViewModel.unsubscribe(roomName);
    }
}
```

Figure 59 - Subscribe/Unsubscribe from FCM



LiveDataViewModel in this case receives a method call from view and sends it to the repository.

```
/*
 * SUBSCRIBE & UNSUBSCRIBE
 */

public void subscribe(MyRoom room){
    repository.subscribe(room);
}

public void unsubscribe(String roomName) { repository.unsubscribe(roomName); }
```

*Figure 60 - LiveDataViewModel methods for subscribing/unsubscribing to FCM*

LiveDataRepository then calls FCMHelper singleton class to notify Firebase which topic is unsubscribed.

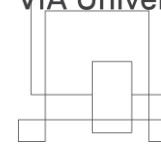
```
/*
 * SUBSCRIBE & UNSUBSCRIBE
 */

public void subscribe(MyRoom room) { fcmHelper.subscribe(room); }

public void unsubscribe(String roomName) { fcmHelper.unsubscribe(roomName); }
```

*Figure 61 - LiveDataRepository methods for calling FCMHelper class to subscribe unsubscribe from FCM*

The figure below displays the endpoint of MVVM structure where data source methods are called straight from the view. The whole lifecycle went from View -> ViewModel -> Repository -> Data source. In this case, FCMHelper class methods are called that are executing subscribe/unsubscribe functions from Firebase Cloud Messaging servers. Before subscribing to the topic currentRoom, Mutable Live Data is updated by MyRoom object that contains room name and room id which are essential for displaying current room name user is at and execute web api requests for specific room by currentRoom id.



```
public void subscribe(MyRoom room){
    if(room != null){
        currentRoom.postValue(room);
        firebaseMessaging.subscribeToTopic(room.getRoomName());
    }
}

public void unsubscribe(String roomName){
    firebaseMessaging.unsubscribeFromTopic(roomName);
}
```

Figure 62 - Endpoint of MVVM Structure of Subscribe/Unsubscribe example of Firebase Cloud Messaging

### 5.3.5. UI Implementation

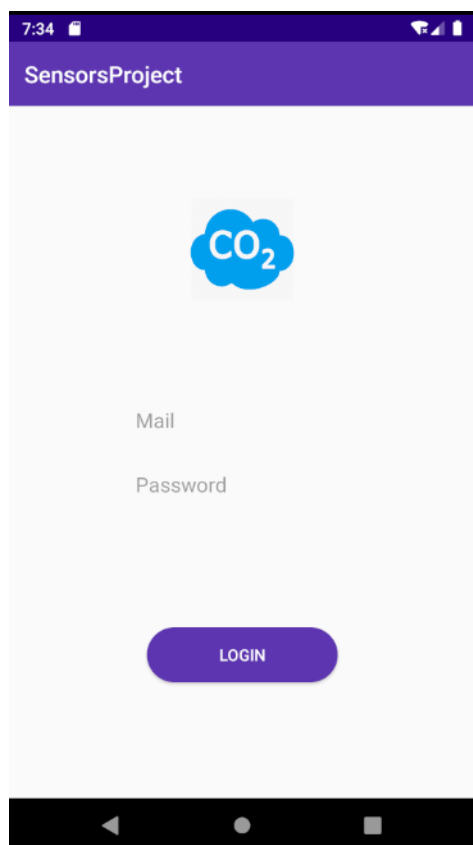


Figure 64 - Login Fragment

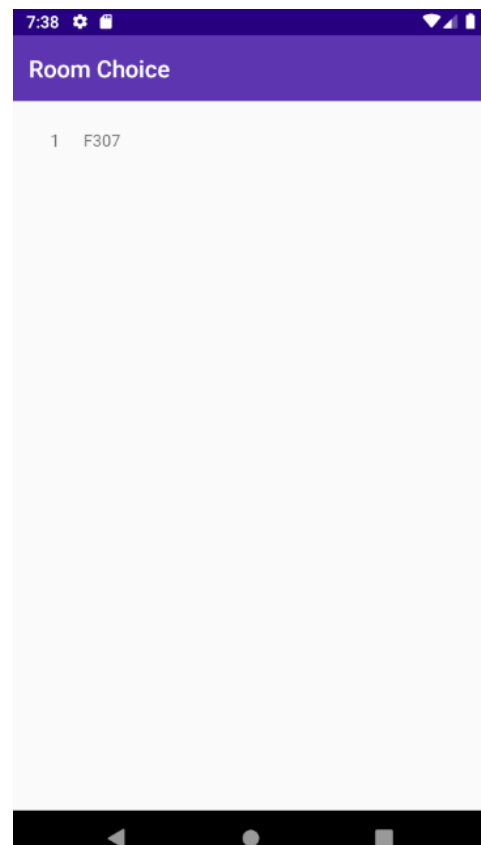


Figure 63 - Room Choice Fragment



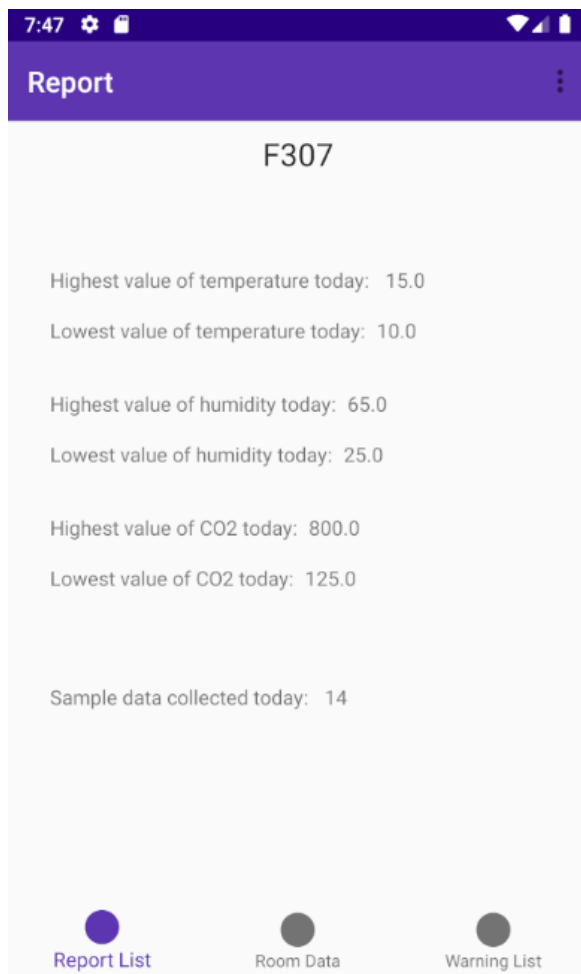


Figure 65 - Report Fragment

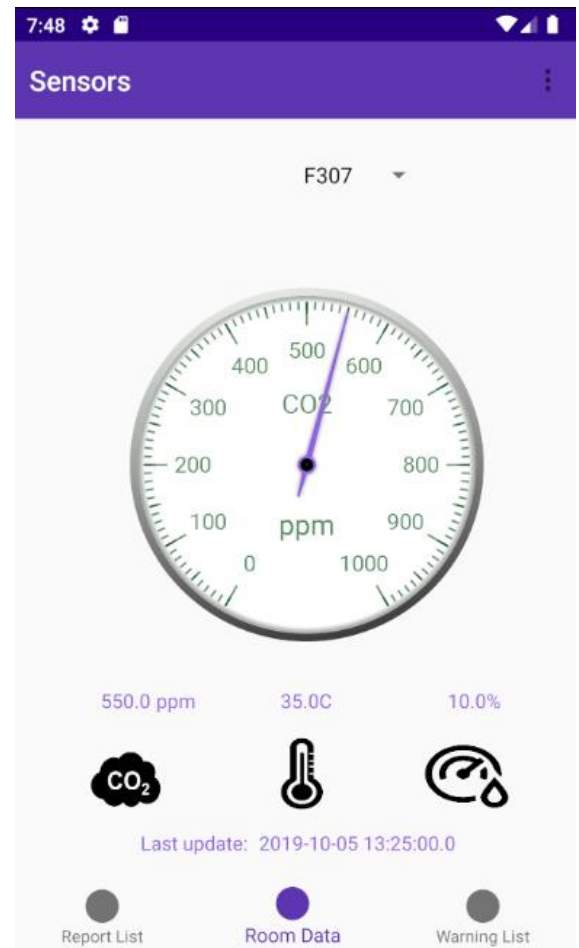


Figure 66 - Room Main Fragment

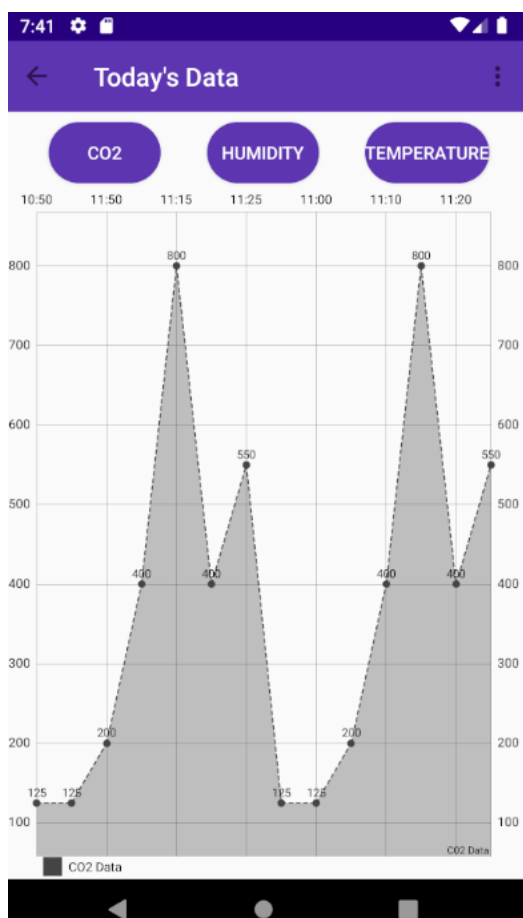


Figure 67 - Today's Data Fragment



Figure 68 - Warning List Fragment

Above there are all fragments of the Android app displaying project requirements. User interface follows Material Design coloring guidelines to match the contrast of colorPrimary, colorPrimaryDark and colorAccent in colors.xml res file.

**Figure 63 Login:** Implemented with FirebaseAuth help.

**Figure 62 Room Choice:** Fragment contains one recycler view that holds data retrieved from web services which gets a list of rooms. There is an onClickListener added for each ViewHolder of the recyclerview and whenever it is pressed, user is redirected to Room Main and subscription to Firebase Cloud Messaging occurs here.

**Figure 65 Room Main:** When this fragment is started, visibility of toolbar and bottom navigation view are enabled, 3 web api requests are sent to load up CO2, Temperature and Humidity data, recent data is extracted from data lists to display current live data of the sensors and after a specific time interval, Firebase Cloud Messaging receives live data from the sensors which is applied in this view.

**Figure 64 Report:** Simple report fragment which determines what were the highest and lowest values of the data today and how many samples were taken today. This is achieved by running a for loop throughout each of the lists of data and returning minimum and maximum values.

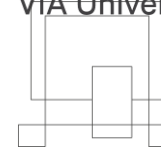
```
private void updateCo2Values(List<CO2> co2List){
    if(co2List != null && co2List.size() > 0){
        float maxValue = -9999;
        float minValue = 9999;
        for(int i = 0; i < co2List.size(); i++){
            CO2 currentCo2 = co2List.get(i);
            float value = Float.parseFloat(currentCo2.getValue());
            if(value > maxValue) maxValue = value;
            else if(value < minValue) minValue = value;
        }

        textCo2High.setText("" + maxValue);
        textCo2Low.setText("" + minValue);
        textSampleData.setText("" + co2List.size());
    }
}
```

*Figure 69 - Example how minimum and maximum values are found in retrieved data*

**Figure 67 Warning List:** This fragment has a recycler view implemented which defines Warning model in a ViewHolder. Purpose of this fragment is to satisfy a requirement for displaying history of occurred warnings in a specific room.

**Figure 66 Today's Data:** Last fragment displays charts that display today's gathered data of CO2, Humidity or Temperature. MPAndroidChart library was used to create the visual report here. More about it in the next section.



### 5.3.6. Charts

Charts are implemented to satisfy a requirement to visualize data of the gathered data. This is achieved by using a 3rd party library called MPAndroidChart which handles most of the logic how to display a chart.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_today_data, container, attachToRoot: false);
    ButterKnife.bind(target: this, view);

    roomId = LiveDataViewModel.getCurrentRoom().getValue().getId();

    subscribeCo2Data();
    subscribeHumData();
    subscribeTempData();
    setOnClickListeners();
    setupLineChart();

    return view;
}
```

Figure 70 - TodaysDataFragment onCreateView method

Figure above has 5 void methods that initialize observers for CO2, Temperature and Humidity, sets up onClickListeners and initializes LineChart which is an in-built chart from MPAndroidChart library.

```
public void setupLineChart(){
    lineChart.setTouchEnabled(true);
    lineChart.setPinchZoom(true);
}

public void subscribeCo2Data(){
    measurementViewModel.getAllCo2sByRoomIdToday().observe(owner: this, co2s -> {
        if(co2s != null & co2s.size() > 0){
            setupCo2Chart(co2s);
        }
    });
}

private void setOnClickListeners(){
    buttonCo2.setOnClickListener(v -> {
        measurementViewModel.searchAllCo2sByRoomIdToday(roomId);
    });
}
```

Figure 71 - Methods from TodaysDataFragment

SubscribeCo2Data observes a list of CO2 from measurementViewModel which has Live Data reference from NetworkHelper that contains recent data retrieved from web API requests.

When buttonCo2 is clicked, then a method is fired from measurementViewModel to search for all co2's by room id gathered today with an argument of room id to define which room's data to retrieve. When this method is finished, the observer in subscribeCo2Data method notices the changes and calls setupCo2Chart method with the newly updated co2's list as an argument.

```
if(lineChart.getData() != null) {
    lineChart.getData().clearValues();
}
ArrayList<Entry> values = new ArrayList<>();
String[] dates = new String[co2List.size()];
for(int i = 0; i < co2List.size(); i++){
    float pos = (float) i;
    float value = Float.parseFloat(co2List.get(i).getValue());
    dates[i] = co2List.get(i).getTimestamp();
    values.add(new Entry(pos, value));
}

XAxis axis = lineChart.getXAxis();
axis.setValueFormatter(new DateValueFormatter(dates));
```

*Figure 72 - Displays 1st part of setupCo2Chart method*

In this part of the method data is set up according to the library needs. An Entry class is used as a way to determine X and Y axis of the chart. X axis is used to display a timestamp and Y axis is used to display the value of the data. An array of dates is created which is later used in a DateValueFormatter class that is extending ValueFormatter class from the library. In this class a custom formatter is set for X axis to use timestamp instead of numbers growing linearly.

## 6. Test

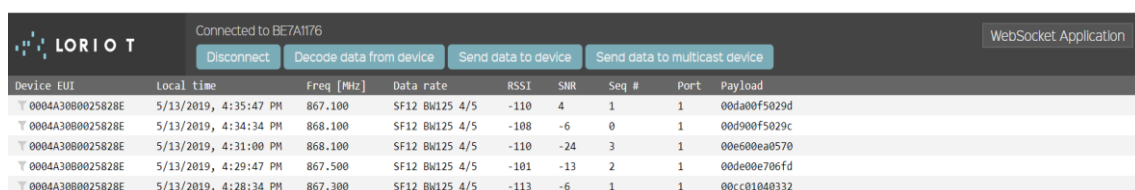
### 6.1. Embedded

#### 6.1.1. Test Specifications

The code responsible for the embedded part of the system was tested to verify that all the requirements are working perfectly. Black box testing was chosen for primary method of testing.

External program Hterm and external library was used to read messages sent from board to computer through a usb cable. Messages were original C language printf statements. Due to possibility to read messages from board, every functionality could be tested by printing for example measured values, results of creating drivers, results of sending packets from LoRa, etc.

The screenshot below shows the Lorient server application after receiving packets from LoRaWan module:

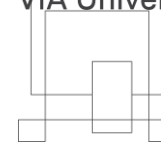


The screenshot shows the LORIoT web application interface. At the top, it says 'Connected to BE7A1176'. Below this are four buttons: 'Disconnect', 'Decode data from device', 'Send data to device', and 'Send data to multicast device'. On the far right, there is a 'WebSocket Application' button. The main part of the interface is a table with the following columns: Device EUI, Local time, Freq [MHz], Data rate, RSSI, SNR, Seq #, Port, and Payload. The table contains five rows of data, all from the same device EUI (0004A30B0025828E) and showing various frequency and data rate combinations.

Device EUI	Local time	Freq [MHz]	Data rate	RSSI	SNR	Seq #	Port	Payload
0004A30B0025828E	5/13/2019, 4:35:47 PM	867.100	SF12 BW125 4/5	-110	4	1	1	00da00f5020d
0004A30B0025828E	5/13/2019, 4:34:34 PM	868.100	SF12 BW125 4/5	-108	-6	0	1	00d900f5029c
0004A30B0025828E	5/13/2019, 4:31:00 PM	868.100	SF12 BW125 4/5	-110	-24	3	1	00e600ea0570
0004A30B0025828E	5/13/2019, 4:29:47 PM	867.500	SF12 BW125 4/5	-101	-13	2	1	00de00e706fd
0004A30B0025828E	5/13/2019, 4:28:34 PM	867.300	SF12 BW125 4/5	-113	-6	1	1	00cc01040332

Figure 73 - LoRaWan Screenshot confirming received packets

Bridge application did not require many tests. After testing the connection between application and MongoDB database, final black box tests were made. Final test consisted of comparing values sent from LoRa module to the values in MongoDB cluster.



Functionality	Result
CO2 sensor	Tested and working
Temperature/Humidity sensor	Tested and working
LoRa module	Tested and working
Tasks cycle	Tested and working
Sending data to websocket server	Tested and working
WS bridge app - receive data	Tested and working
WS bridge app - store data to MongoDB	Tested and working

## 6.2. Data

### 6.2.1. Postman

For testing web services, Post man has been used to verify the expected result from HTTP requests through the controller

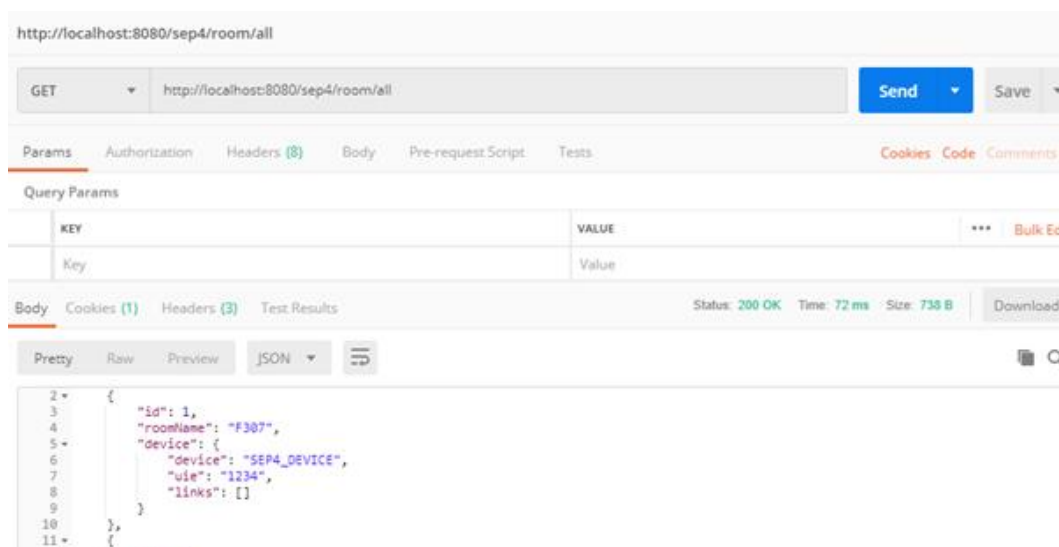
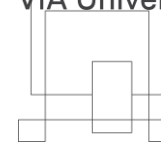


Figure 74 - Postman Test



### 6.2.2. Test Specifications

Functionality	Result
Firebase Cloud Messaging	Test Passed
Web Service Controllers	Test Passed
Database connection	Test Passed
Web application functionalities	Test Passed
ETL	Test Passed
Triggers in the database	Test Passed





### 7.1.1. Java Extractor

To test the java extractor JUNIT test cases were made here. Individual methods were continuously tested.

```

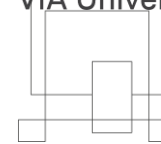
44     @Test
45     public void loadDevice() {
46         device.deleteAll();
47         EUI = er.findAll();
48         for(int i =0; i< EUI.size(); i++)
49         {
50             device.save(new Device(EUI.get(i).getName(),EUI.get(i).getUie()));
51         }
52
53         device.findAll().forEach(System.out::println);
54         System.out.println("-----");
55     }
56
57     @Test
58     @Scheduled(fixedRate = 5000)
59     public void updateDevie()
60     {
61         EUI = er.findAll();
62
63         int value = EUI.size()-co2.findAll().size();
64
65         for(int i =EUI.size()-value; i<EUI.size(); i++)
66         {
67             device.save(new Device(EUI.get(i).getName(),EUI.get(i).getUie()));
68         }
69     }

```

*Figure 75 - JUnit Test for loadDevice() & updateDevice()*

### 7.1.2. Web application

Here a combination of black/white box testing were used. In addition, the use of breakpoints were setup in combination with the white box testing.

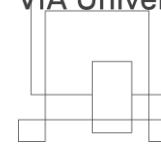


## 7.2. Android

### 7.2.1. Test Specifications

Android also was tested using black box testing methodology. Most of the functionality that was tested is related to the backend of the system. During the development Log.d command was used to test the functionality of the application and confirm specific parts of the codebase are working as intended and are being called in time, synchronized with other methods.

Functionality	Result
Firebase Authentication	Test Passed
Retrofit2 API Requests	Test Passed
Navigation Flow	Test Passed
Generate Report About Data	Test Passed
Receive Live Data From FCM	Test Passed
Visualize Data	Test Passed



### 7.3. Test Specifications

The purpose of the test phase is to accurately test the system in order to ensure its functionality. Black box testing method has been used in order to test the use cases.

In the table, on the left column which functionality is being tested and respectively on the right, results of the testing. Moreover, all except one of the use cases have successfully passed the testing.

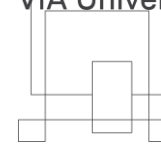
Use case	Result
Add Room	Test Passed
Edit Room	Test Passed
Delete Room	Test Passed
Add Device	Test Passed
Edit Device	Test Passed
Delete Device	Test Passed
Add Admin	Test Passed
Edit Admin	Test Passed
Delete Admin	Test Passed
Visualize data	Test Passed
Visualize report	Test Passed
Check warning history	Test Passed
Display warning notification	Test Failed
Choose classroom	Test Passed

## 8. Results and Discussion

The results from the section above showed that all functionalities planned for this system have been accomplished successfully. The necessary diagrams were done correctly and in time. Even if the system is fully functional, this will not be the release version of the software. Graphical user interface can be done better from the design perspective.

Scrum has been used during the project implementation. This methodology provides a well-designed structure of splitting the total workload into smaller pieces which should be completed in a limited amount of time. Total time spent for the task and person responsible for the task are noted down in a table which is called sprint backlog. It is built in a way so that introduced information allows the user to easily track down the progress made. Short Scrum meetings are organized at the start of each sprint so that all members are familiar with the tasks that are planned out. By following this methodology, it was much easier to organize the responsibilities in the group.

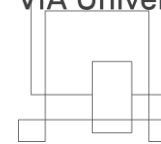
Considering the fact that during this project a group of ten people had to cooperate in order to build a complete final product. Good communication was established during the meetings.



## 9. Conclusions

The project goal was to ensure Indoor Air Quality by tracking CO<sub>2</sub>, Temperature and Humidity data. Embedded team fortunately managed to set up sensor devices to track room air measurements and successfully send them through a connection to LoraWan which transmits data to the Bridge Application. This Bridge Application is populating sensor data to the MongoDB. Later Data Tier extracts it from MongoDB using Java Extractor software to run it through dimensional modeling and send it to MSSQL. MSSQL then can execute triggers whenever dangerous levels of air measurements are saved and lists of received data are persisted. From here Web Service Application is taking data and is exposing it to the Android Applications through REST API Server.

In conclusion, the project requirements were satisfied, and the system transmits data successfully from sensors to the android application.



## 10. Project future

Future of this project has a high potential for indoor usage. Following improvements will make it more usable in the market. First of all, if the carbon dioxide level reaches the limit or even passes it a warning notification should be prompted, for example about uncomfortable CO<sub>2</sub> levels and windows could be open automatically. Secondly, in more severe cases where carbon dioxide levels rise and do not change after opening a window, for example, a sound should play to alert the people and evacuate the room immediately.

Moreover, an implementation that manipulates the temperature and humidity in a room will be very important. For example, in a greenhouse it will be much easier to control the well growing of the plants that are planted.

Android project was structured in such way so it could be scalable. As of now it does not handle errors well, but in the future, it is possible to catch networking requests. The application can survive configuration changes, however it is not fully responsive, due to the 3<sup>rd</sup> party library usage that is not optimized for responsiveness.

## 11. Sources of information

Baeldung, 2018. *Baeldung*. [Online]

Available at: <https://www.baeldung.com/queries-in-spring-data-mongodb>

Developers, A., 2018. *Single Activity: Why, When, and How (Android Dev Summit '18)*.

s.l.:<https://www.youtube.com/watch?v=2k8x8V77CrU&feature=youtu.be>.

Documentation, B., 2018. *Spring.io*. [Online]

Available at: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation-first-steps>

Introduction, 2018. *Spring.io*. [Online]

Available at: <https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#introduction>

jt, 2017. *SpringFramework*. [Online]

Available at: <https://springframework.guru/configuring-spring-boot-for-microsoft-sql-server/>

Mapping, 2018. *Spring.io*. [Online]

Available at: <https://docs.spring.io/spring-data/mongodb/docs/1.3.3.RELEASE/reference/html/mapping-chapter.html>

mongoDB, 2008. *mongoDB*. [Online]

Available at: <https://docs.mongodb.com/manual/reference/sql-comparison/>

Philipines, I., 2018. *IAQ Philipines*. [Online]

Available at: <http://www.iaqphilippines.com/indoor-air-pollutants-among-the-top-five-environmental-risks-to-public-health/>

Safety, O. H. &, 2019. *Occupational Health & Safety*. [Online]

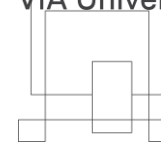
Available at: <https://ohsonline.com/Home.aspx>

Satish, U., 2012. *ehp Environmental Health Perspectives*. [Online]

Available at: <https://ehp.niehs.nih.gov/doi/10.1289/ehp.1104789>

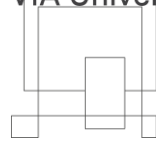
Thayalarajan, G., 2018. *thePro.io*. [Online]

Available at: <https://thepro.io/post/firebase-authentication-for-spring-boot-rest-api/>



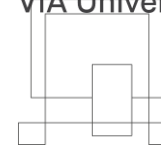
## **12. Appendices**





## Appendix A: Project Description

Located in the root folder of the hand-in...



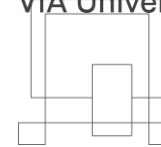
## Appendix B: Detailed List of Tasks

Located in the root folder of the hand-in...

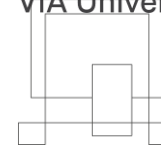
Project Report	
Abstract	Oskars. Revised by Ainis
Introduction	Aleksandr
Analysis	
Functional & Non-Functional Requirements	Oskars. Revised by Vladimir and Ainis
Use Case	Robert + Fadi
Domain Model	Fadi
Activity Diagram	Alex
Design	
<i>Embedded</i>	
Embedded Sequence Diagram	Robert + Roza
Embedded Technologies	Oskars
Embedded Design Patterns	Roza + Oskars
Websocket Bridge Application	Robert
Websocket Bridge App Technologies	Robert
Web Socket Bridge App Class Diagram	Robert
<i>Data</i>	



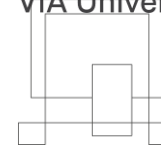
EER Diagram	Fadi
Data Warehouse - ETL	Balkis
SQL Server Job (Scheduling)	Nadeem
Power BI	Nadeem
Data Web Services Class Diagram	Fadi
Data Web Service Sequence Diagram	Fadi
Java Extractor	Yasin
Web Application	Yasin
<i>Android</i>	
Android Class Diagram	Aleksandr
Android Sequence Diagram	Aleksandr
Android Architecture	Aleksandr
Android Technologies	Aleksandr
Android Design Patterns	Aleksandr
UI Design Choices	Aleksandr
<b>Implementation</b>	
<i>Embedded</i>	
C Code	Robert



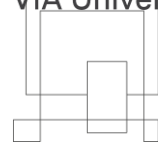
The WebSocket Application (Web Socket client)	Robert
<i>Data</i>	
Java Extractor	Yasin
Web Application	Yasin
Data Warehouse	Balkis
Trigger	Fadi
Web Service	Fadi
<i>Android</i>	
Single Activity Principle	Ainis
Web Services (Network Helper)	Ainis
Firebase Cloud Messaging	Ainis
MVVM Structure	Ainis
UI Implementation	Ainis
LiveDataRepository Implementation	Ainis
Charts	Ainis
<b>Testing</b>	
Embedded	Robert + Roza
Data	Fadi + Yassin
Android	Ainis



System Testing	Vladimir
<b>Project Report Ending</b>	
Results and Discussions	Vladimir
Conclusion	Vladimir
Project Future	Vladimir
Appendices	All
<b>Appendices</b>	
Appendix A Project Description	Ainis
Appendix B Scrum Sprint Tables	Vladimir
Appendix C Diagrams	Ainis
Appendix D Source Code	Ainis
Appendix E Demonstrational Video	Ainis
Appendix F Data Tier Scrum	Balkis
Appendix G Web Application User Guide	Yassin
Appendix H Github Link	Ainis
<b>Process Report</b>	
Introduction	Balkis

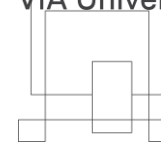


Group description	All
Project Initiation	Oskars
Project Description	Roza
Project Execution	Vladimir
Supervision	All
Conclusion	Vladimir + Alex + Ainis
Appendices	Vladimir



## Appendix C: Diagrams

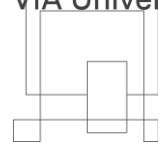
Located in the Appendix C Diagrams folder...



## Appendix D: Source Code

Located in the [Appendix D Source Code](#) folder...





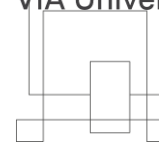
## Appendix E: Demonstrational Video

Located in the root folder of the hand-in...



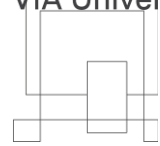
## Appendix F: Data Tier Scrum

Located in the Appendix F Data Tier Scrum folder of the hand-in...



## **Appendix G: Web Application User Guide**

Located in the root folder of the hand-in...



## Appendix H: Web Application User Guide

Located in the root folder of the hand-in...