PROJECT REPORT

Zuper - Library Climate Control
IoT System

Students

Amahdya Delkescamp - 256523

Andrei Cioanca - 266105

Claudiu Rediu - 266129

Dominika Kubicz - 266148

Flemming Vindelev - 251398

Michal Ciebien - 266908

Mihail Kanchev - 266106

Nikita Roskovs - 266900

Stefan Harabagiu - 266116

Supervisors

Kasper Knop Rasmussen

Knud Erik Rasmussen

Erland Ketil Larsen

Acknowledgements

The team wishes to formally acknowledge VIA University College’s active implication in the evolution and completion of this project work. All documents contained or referred to from this project are based on VIA University College’s templates and designs. All information present in this document, while original, could not have been made possible without VIA’s resources.

Abstract

The Climatizer Project is an Internet Of Things project that was made for use in the VIA Library in order to monitor the climate levels and to help maintain an optimal learning environment for students. The application would achieve this through the use of various sensors and applications. User requests start at the Android application, sent to a Web API, and are sent to a Bridge Application. At the Bridge Application, a socket connection is used to send the request over the LoRaWAN to an Arduino board that is equipped with sensors. The Arduino responds to the request by sending data back through the LoRaWAN connection, to the Bridge Application, where the data is cleaned and sent to the correct receivers, which are either the MongoDB where the data will be persisted into the database or to the Web API so the data can be displayed on the Android application.

To build this project the team of 9 was broken up into groups of 3 representing 3 specializations: Embedded Engineering, Data Engineering, and Cross Media. Each team utilized their own architectures and processes. The Embedded group used FreeRTOS to act as an operating system that will allow tasks to be performed for the Arduino board. The Data Engineering group used Dimensional Modeling and the ETL process to store climate data to a database and to extract, clean, and load the SQL Server data. The Cross Media group used MVVM architecture to build the Android app and display the readings from the sensors and database to the user.

The result of the project is a complete IoT system where the user can request the current climate levels of the VIA Library using an app that communicates with an installed sensor and can also view the climate readings over a period of time.

Table of Contents

Introduction

A library is a collection of information and resources that are made accessible to a community for reference and borrowing. Libraries provide either physical or digital access to various materials. As a place where people can spend their time, regardless if it is in a room or the library building itself, it is important to maintain comfort. (Merriam-Webster, 2019; Lawrence, 1997) As mentioned in (educationcorner, 2019), is it counterproductive to study for extended hours at a time in an uncomfortable environment. The library fulfills the needs of most people in creating a proper environment for learning because they offer individual cubicles, group study rooms, tables, couches, and other resources. These studying areas should be kept under close watch to maintain the best conditions for the people inside it.

There are two sides to libraries, the books and the people inside the library. Both are affected by the humidity, however, the people inside are also susceptible to variations in temperature and CO2 levels. For humans, high humidity together with high temperature can lead to a rise in the core temperature, which makes the body work harder to cool itself down. On the other hand, low humidity increases water loss through skin and respiration leading to dry eyes, sore throat, dry skin and many other symptoms (Allergy, 2019). Several studies conducted indicate that CO2 commonly present indoors can impair cognitive function. It is relevant for the efficiency of the people inside the library to keep the CO2 levels below 1000 ppm (parts per million). Reducing high levels of CO2, reduced the instances of headaches and tiredness while improving the satisfaction (ecoadmin, 2016).

In the case of books, controlling the temperature and relative humidity is important to preserve the state of the library. Heat accelerates deterioration, and every increase in temperature of 10° C doubles the rate of deterioration. High relative humidity in the air contains moisture that can cause damaging chemical reactions. Extremely low relative humidity can lead to drying and cracks in the book's material. Temperature and relative humidity should be measured and recorded to support the documentation of current environmental conditions (Ogden, 2019).

Libraries are lacking a way to monitor information related to temperature, humidity and CO2 levels to create a better environment for books and people.

Requirements

Functional Requirements

1. A user should be able to see the current temperature in the library.
2. A user should be able to see the current humidity in the library.
3. A user should be able to see the current CO2 levels in the library.
4. An administrator should be able to open a window in the library remotely.
5. A user should be able to switch between different measurement scales. (ex. Celsius -> Fahrenheit)

Non-Functional Requirements

1. The system will be written in a multitude of languages including, C, Java and SQL
2. The system will be developed under an IoT network

System Analysis

The libraries have specific requirements when it comes to the actual data that the system reads, stores and shows to the user. More specifically, the system must be able to read and process CO2, temperature and humidity levels.

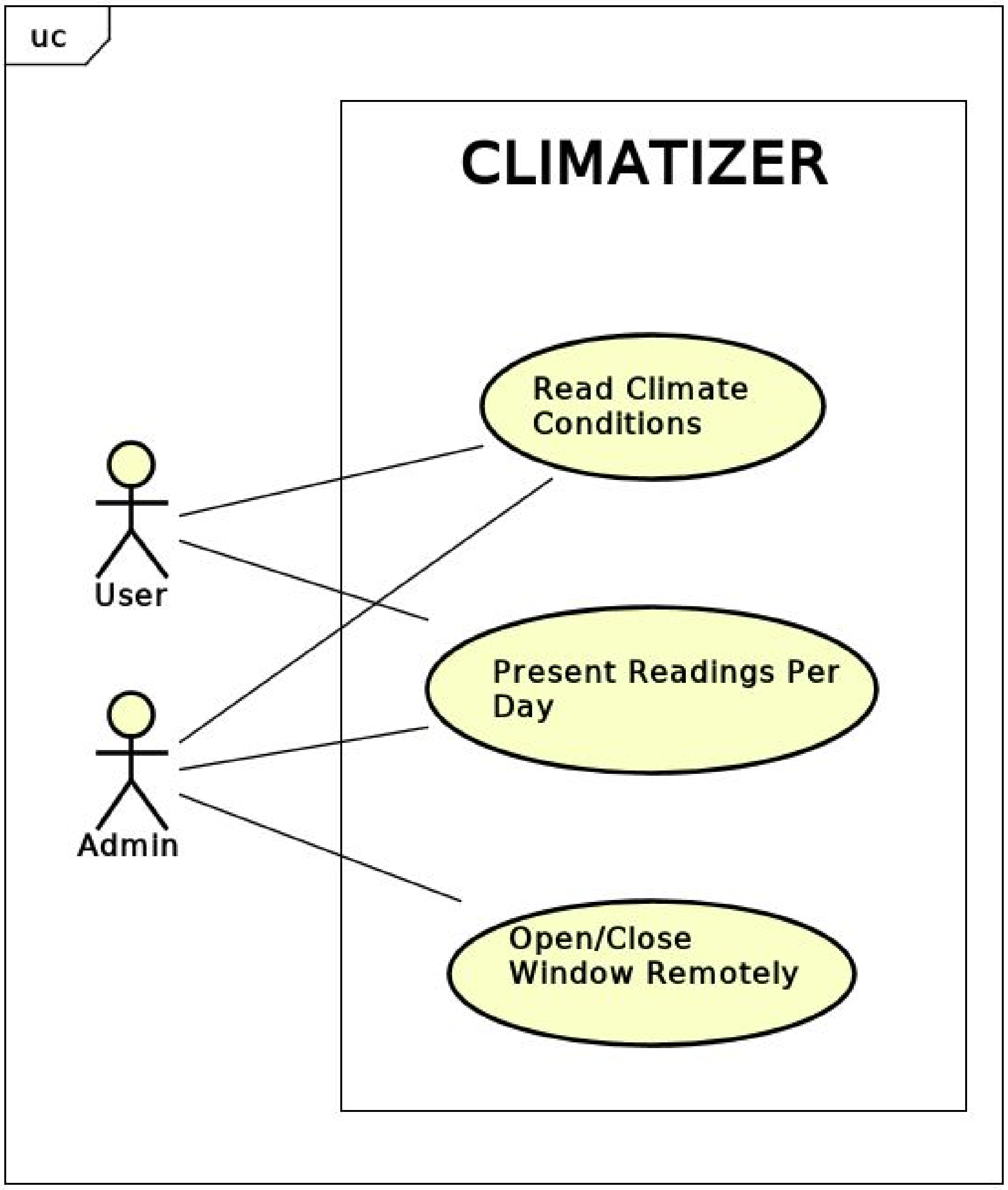
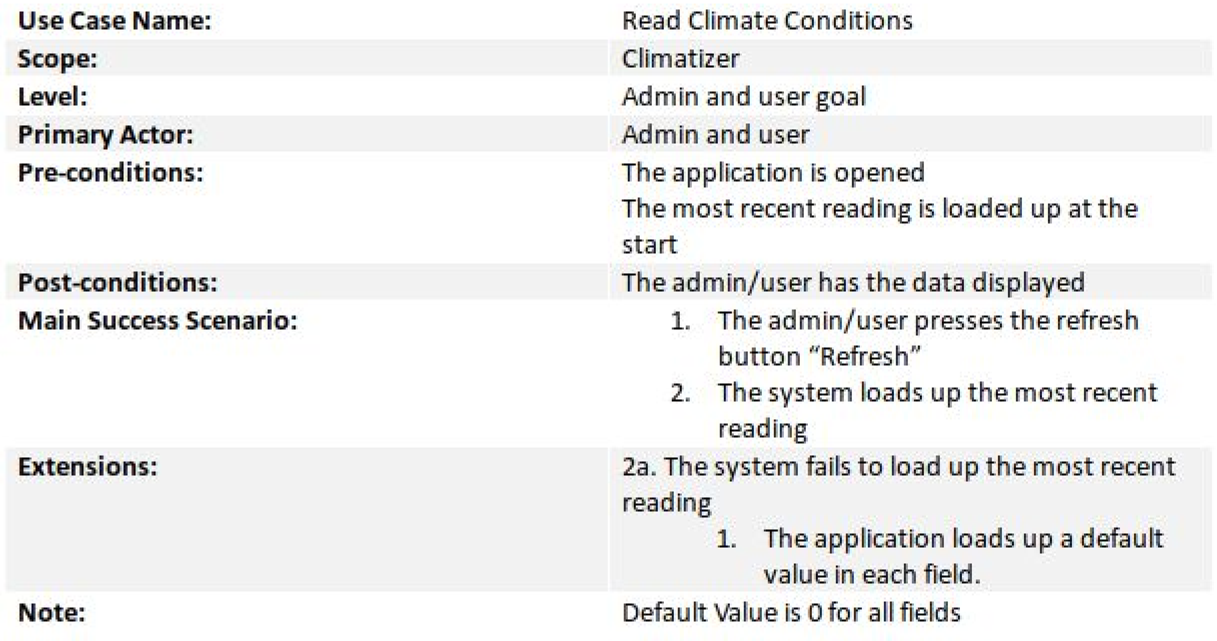


Figure 1 System Analysis - Use Cases

There are several problems to analyze. One of them is the grouping and processing of those readings. Another problem would be knowing when such a reading took place.



Use Case Name:	Read Climate Conditions
Scope:	Climatizer
Level:	Admin and user goal
Primary Actor:	Admin and user
Pre-conditions:	The application is opened The most recent reading is loaded up at the start
Post-conditions:	The admin/user has the data displayed
Main Success Scenario:	1. The admin/user presses the refresh button "Refresh" 2. The system loads up the most recent reading
Extensions:	2a. The system fails to load up the most recent reading 1. The application loads up a default value in each field.
Note:	Default Value is 0 for all fields

Figure 2 System Analysis - Read Climate Conditions Use Case

To solve these issues, the system was built with these "readings" in mind. Each reading encompasses CO2, temperature and humidity levels, the system also registers the current date and time whenever there is a new reading. Thus allowing to display all the readings that have been done in a day.

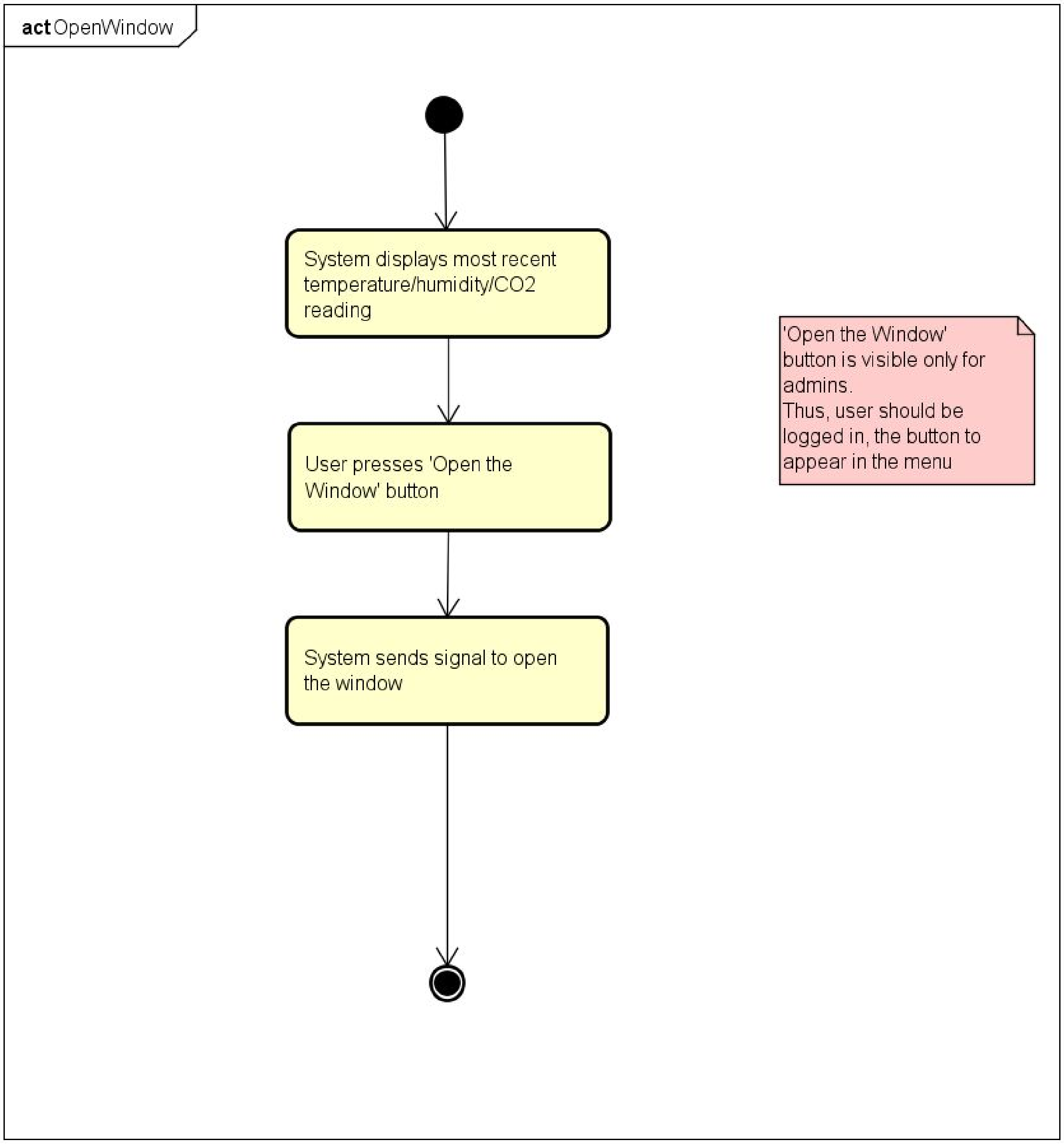

act OpenWindow
System displays most recent temperature/humidity/CO2 reading
User presses 'Open the Window' button
System sends signal to open the window
'Open the Window' button is visible only for admins. Thus, user should be logged in, the button to appear in the menu

Figure 3 in System Analysis - Activity Diagram for Open/Close Window Use Case

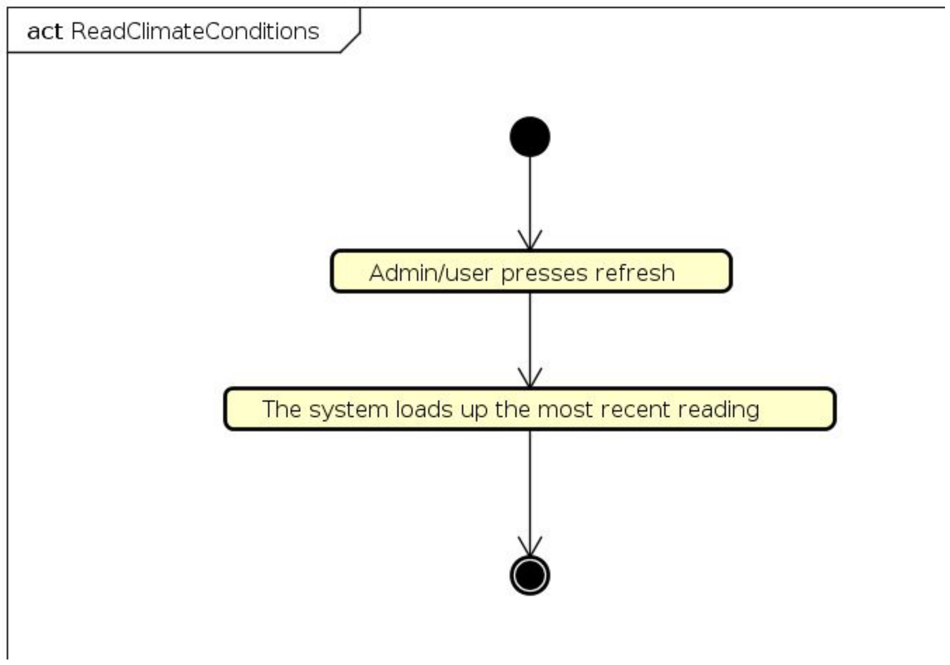

act ReadClimateConditions
Admin/user presses refresh
The system loads up the most recent reading

Figure 4 in System Analysis - Activity Diagram for Reading Climate Conditions Use Case

Both default users and administrators are able to press a button and receive updated readings.

The following domain model came as a result of analyzing the problem domain and the requirements:

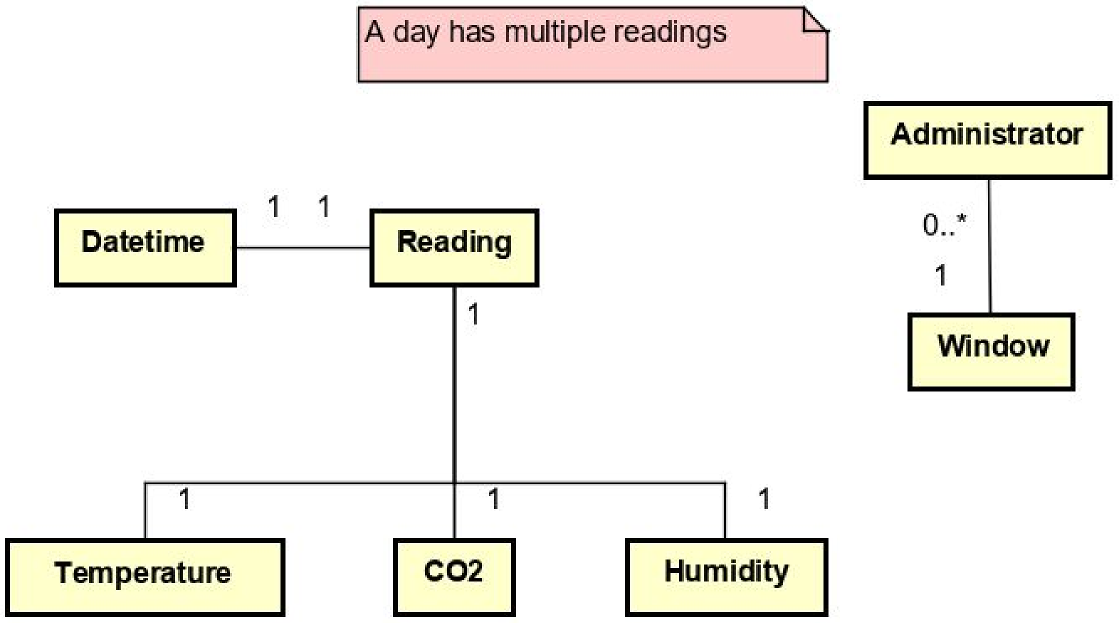


Figure 5 in System Analysis - Domain Model

The objects and their relationships have been identified in the problem domain.

System Design

Android App

(Authors: Amahdya Delkescamp, Andrei Cioanca, Nikita Roskovs)

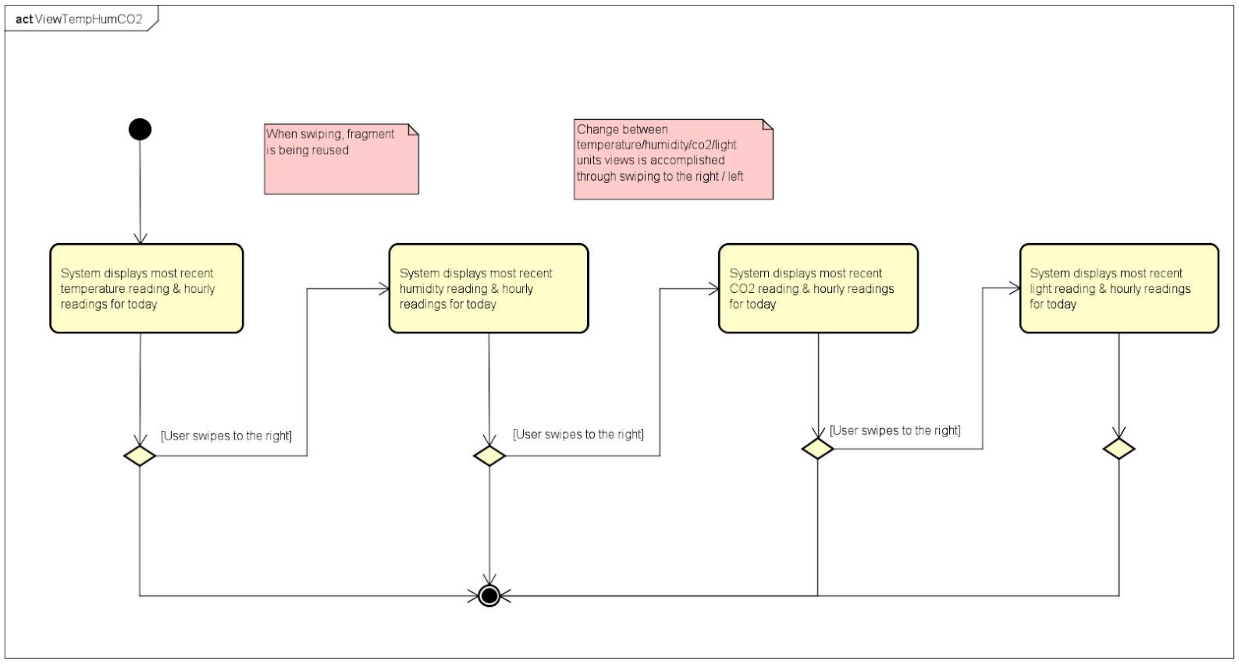

act ViewTempHumCO2
When swiping, fragment is being reused
Change between temperature/humidity/co2/light units views is accomplished through swiping to the right / left
System displays most recent temperature reading & hourly readings for today
System displays most recent humidity reading & hourly readings for today
System displays most recent CO2 reading & hourly readings for today
System displays most recent light reading & hourly readings for today
[User swipes to the right]
[User swipes to the right]
[User swipes to the right]

Figure 1 in System Design - View Readings Activity Diagram

The Android application has been built with ease of access in mind. By facilitating the user's interaction with the app through an intuitive UI, the functionality of the app is readily available and only a few swipes away.

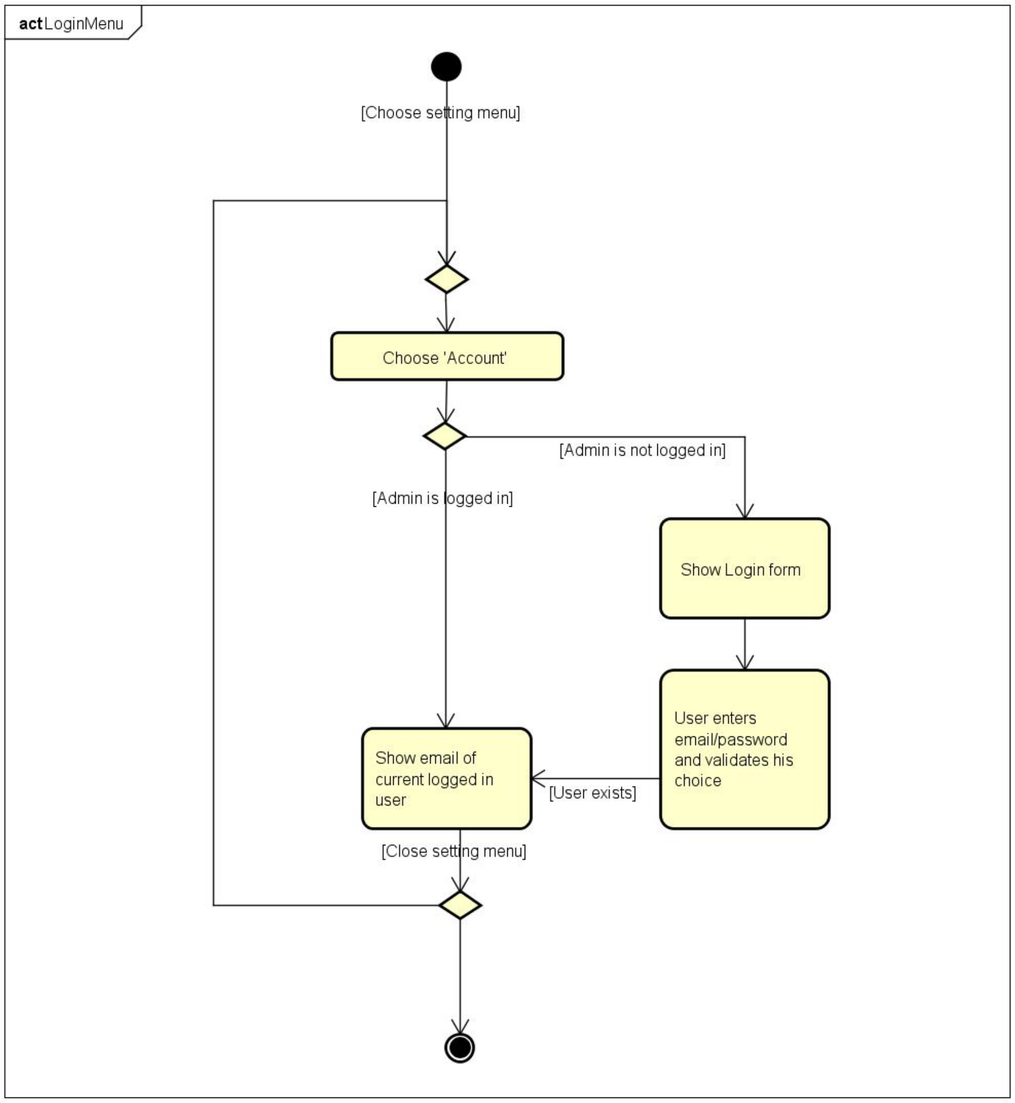


Figure 2 in System Design- Login Menu

The system is built with two kinds of users in mind. The default user that needs no authentication to use the system, while not having access to the window opening/closing controls. The other kind of user is the administrator that can also use the window controls.

The application makes use of the MVVM (Model - View - ViewModel) system architecture. The following diagram shows how the system is structured in broad strokes:

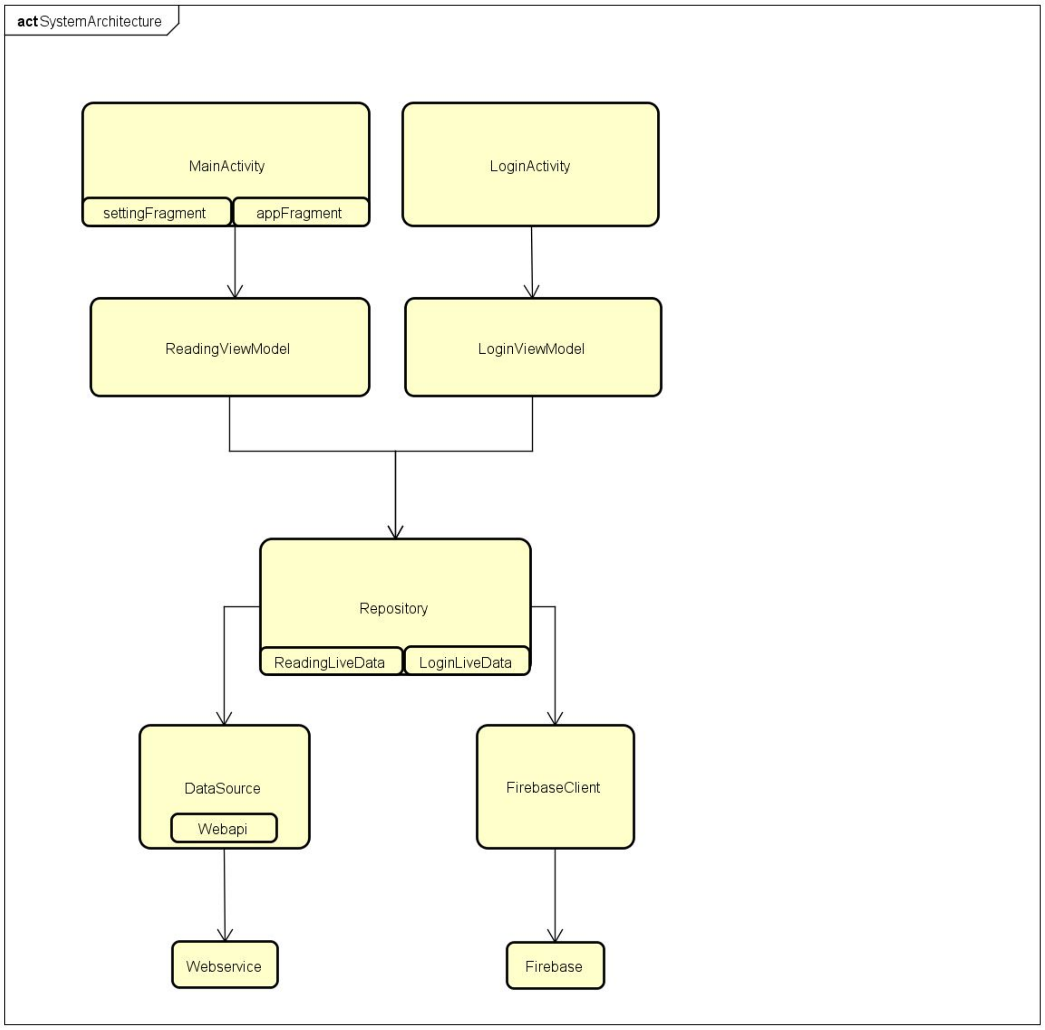


Figure 1 in Android app - App Structure

The data flows in only one direction thus ensuring low-coupling between the different parts of the app. The MainActivity is responsible for displaying most of the information in the app. Here we display the list of readings and the most recent reading we received through the API. Since there are three types of values that need to be displayed (CO2, humidity and temperature), the MainActivity contains a PagerView that keeps reusing the AppFragment for different values of a reading.

The MainActivity also contains multiple material design widgets such as a toolbar and tabbed activities that work together with the PagerView but also a horizontal scrollview that displays the readings of a user-selected date. The toolbar contains a drop-down button with Settings and Login.

Settings adds a new fragment on top of the MainActivity that handles the SharedPreferences. From here the user can set the temperature to either fahrenheit or celsius.

The LoginActivity is where the users can enter their login information in order to authenticate themselves and be able to open a window inside the library remotely. The authentication process is done through firebase.

Navigating between those two activities is done through intents. Each activity has its own ViewModel through which they receive their updated data.

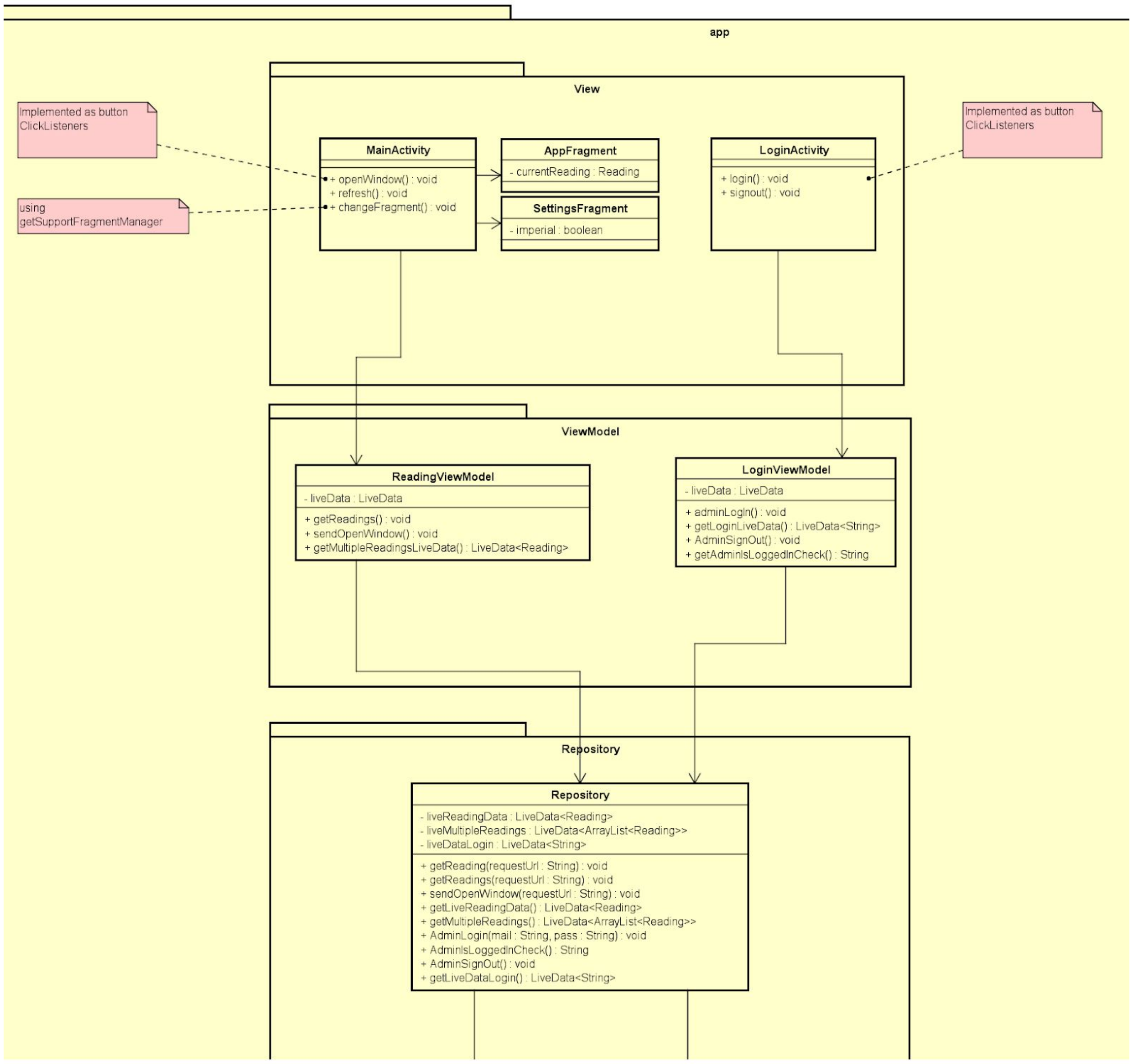
The temperature, humidity and CO2 values are not locally stored so the app must have access to the internet in order to display data properly.

All requests sent to the API go through the WebAPIClient.
There are three types of requests:

1. Request on start-up the most recent reading (a reading is comprised of the most recent CO2, Temperature and Humidity values, multiple readings per day) and also all the per-hour readings of the current day.
2. Manually request an update through the user’s input for the most recent reading. This includes receiving the most recent reading again
3. Request that is sent through the API and the bridge application to the embedded part of the system and tells it to use the window inside the library.

The first two requests are GET requests and the subsequent replies from the API will provide the application with the necessary information to display to the users.
The third request is a PUT request that is sent through the API and through the bridge application to the embedded part of the system and when read, tells the system to open the window in the library.

The diagram below goes further into detail with the design choices made above:


app
View
Implemented as button ClickListeners
using getSupportFragmentManager
MainActivity
+ openWindow() : void
+ refresh() : void
+ changeFragment() : void
AppFragment
- currentReading : Reading
SettingsFragment
- imperial : boolean
LoginActivity
+ login() : void
+ signout() : void
Implemented as button ClickListeners
ViewModel
ReadingViewModel
- liveData : LiveData
+ getReadings() : void
+ sendOpenWindow() : void
+ getMultipleReadingsLiveData() : LiveData<Reading>
LoginViewModel
- liveData : LiveData
+ adminLogin() : void
+ getLoginLiveData() : LiveData<String>
+ AdminSignOut() : void
+ getAdminIsLoggedInCheck() : String
Repository
Repository
- liveReadingData : LiveData<Reading>
- liveMultipleReadings : LiveData<ArrayList<Reading>>
- liveDataLogin : LiveData<String>
+ getReading(requestUrl : String) : void
+ getReadings(requestUrl : String) : void
+ sendOpenWindow(requestUrl : String) : void
+ getLiveReadingData() : LiveData<Reading>
+ getMultipleReadings() : LiveData<ArrayList<Reading>>
+ AdminLogin(mail : String, pass : String) : void
+ AdminIsLoggedInCheck() : String
+ AdminSignOut() : void
+ getLiveDataLogin() : LiveData<String>

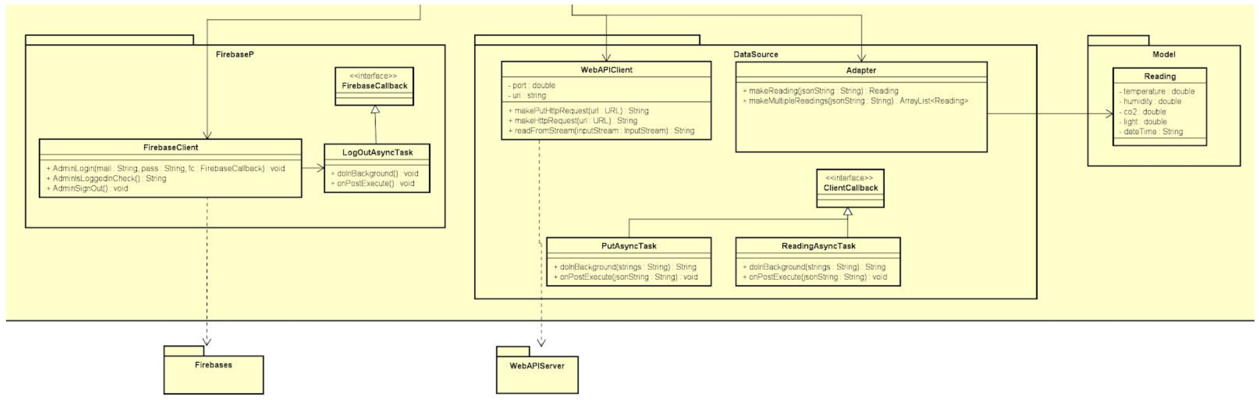

FirebaseP
<<interface>>
FirebaseCallback
FirebaseClient
+ AdminLogin(mail : String, pass : String, fc : FirebaseCallback) : void
+ AdminIsLoggedInCheck() : String
+ AdminSignOut() : void
LogOutAsyncTask
+ doInBackground() : void
+ onPostExecute() : void
DataSource
WebAPIClient
- port : double
- uri : string
+ makePutHttpRequest(url : URL) : String
+ makeHttpRequest(url : URL) : String
+ readFromStream(inputStream : InputStream) : String
Adapter
+ makeReading(jsonString : String) : Reading
+ makeMultipleReadings(jsonString : String) : ArrayList<Reading>
<<interface>>
ClientCallback
PutAsyncTask
+ doInBackground(strings : String) : String
+ onPostExecute(jsonString : String) : void
ReadingAsyncTask
+ doInBackground(strings : String) : String
+ onPostExecute(jsonString : String) : void
Model
Reading
- temperature : double
- humidity : double
- co2 : double
- light : double
- dateTime : String
Firebases
WebAPIServer

Figure 3 in Android app - Architecture

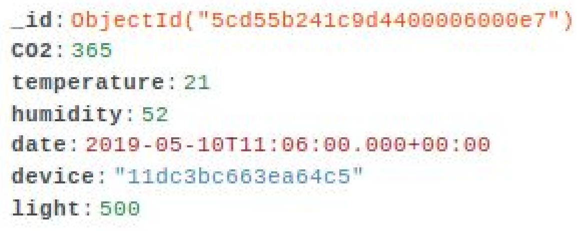
Data

(Authors: Claudiu Rediu, Mihail Kanchev, Stefan Harabagiu)

Cloud-Based Database

In designing an IoT system, a need emerged for a cloud-based database that could receive and transmit data, while providing both synchronous and asynchronous interaction. MongoDB was chosen because it fulfills these requirements. Designing the schema is different from the usual RDBMS database. In the current case, disk space is cheap in comparison with compute time and it is easily optimizable for the most frequent use cases.

The first step in designing it is making choices related to the collection. The current collection will be capped(If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size), will have the index auto-generate and a size(needed for the capped collection). Making it capped will help with the limited available space for the cloud.



```
_id: ObjectId("5cd55b241c9d4400006000e7")
co2: 365
temperature: 21
humidity: 52
date: 2019-05-10T11:06:00.000+00:00
device: "11dc3bc663ea64c5"
light: 500
```

Figure 1 in Data - Document in MongoDB

A document will contain the values detected by the sensors together with the id of the device and a date that contains both the day and the time at which the reading took place.

SQL Server

Data from the MongoDB should be backed up in Microsoft SQL Server. The data that will be stored in there will go through the ETL process to result in data that can be easily visualized for business intelligence.

The SQL Server is responsible for the storage of all data recorded by the sensors and for warehousing that data as well. The data is taken from the MongoDB and stored in a transactional database that serves as the source database for the ETL process.

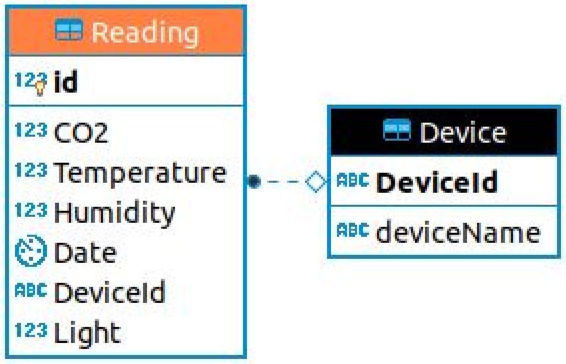


Figure 2 in Data - Source Transactional Database

The source database follows the structure shown in the figure above and based on it the dimensional model is going to be created. For the display reading process, the fact grain is the value of the temperature, CO2, humidity and, light by the date and the time the reading took place at. For displaying a reading, it is also needed the information about the device that recorded it. Therefore, the following dimensional model was created:

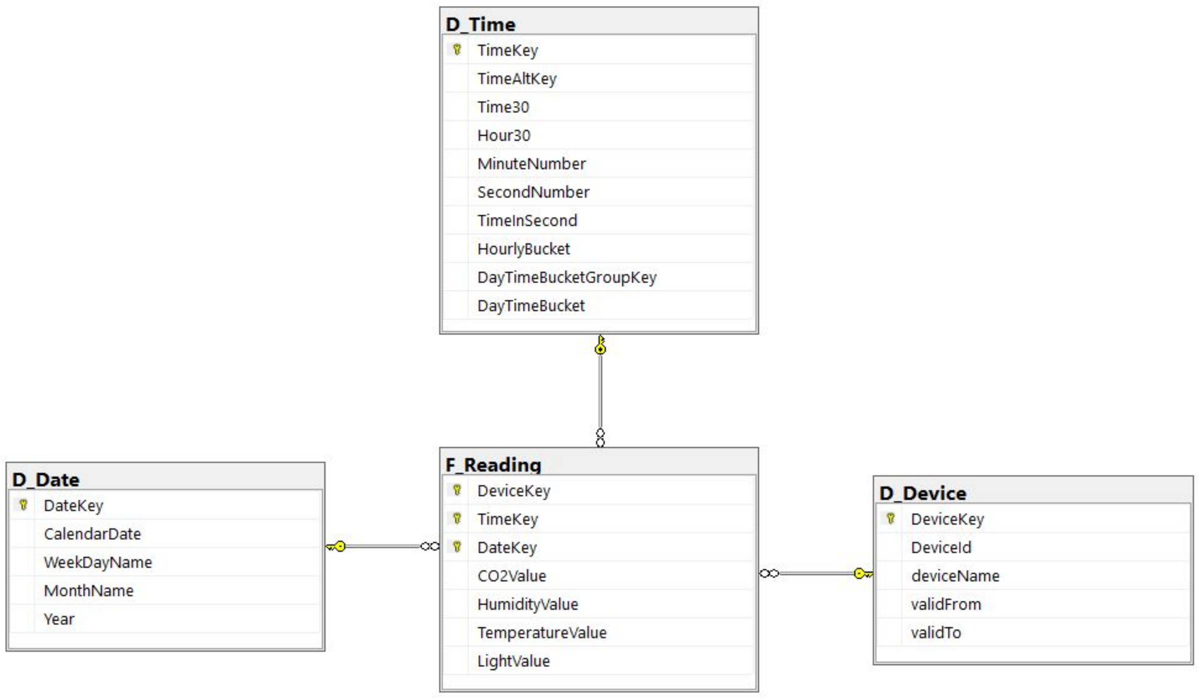


Figure 3 in Data - Reading Dimensional Model

The date dimension allows for grouping of data by either a specific date or by more specific parameters such as the day of a week (e.g.: all Mondays), the moth or year. The time dimension is separate from the date in order to allow more in-depth analysis based on the measurement values at specific hours, as attributes such as temperature change quite frequently. The device dimension allows identifying the specific piece of equipment that took the measurement.

For the purpose of displaying the analyzed data in a meaningful and easily understandable manner, several graphs are created. In the case of temperature values, there will be graphs that show the fluctuations every hour, as well as the daily and monthly average.

Pipeline Data

A new problem arises when trying to transfer the data from the cloud to the local storage. For this purpose, a JAVA program is used to enable communication. It is created to update the local database at a certain interval that would fit the needs of the SQL Server. Once enabled, it will run on its own at acertain time interval. The following diagram (Figure 4 in Data) would showcase this in more detail.

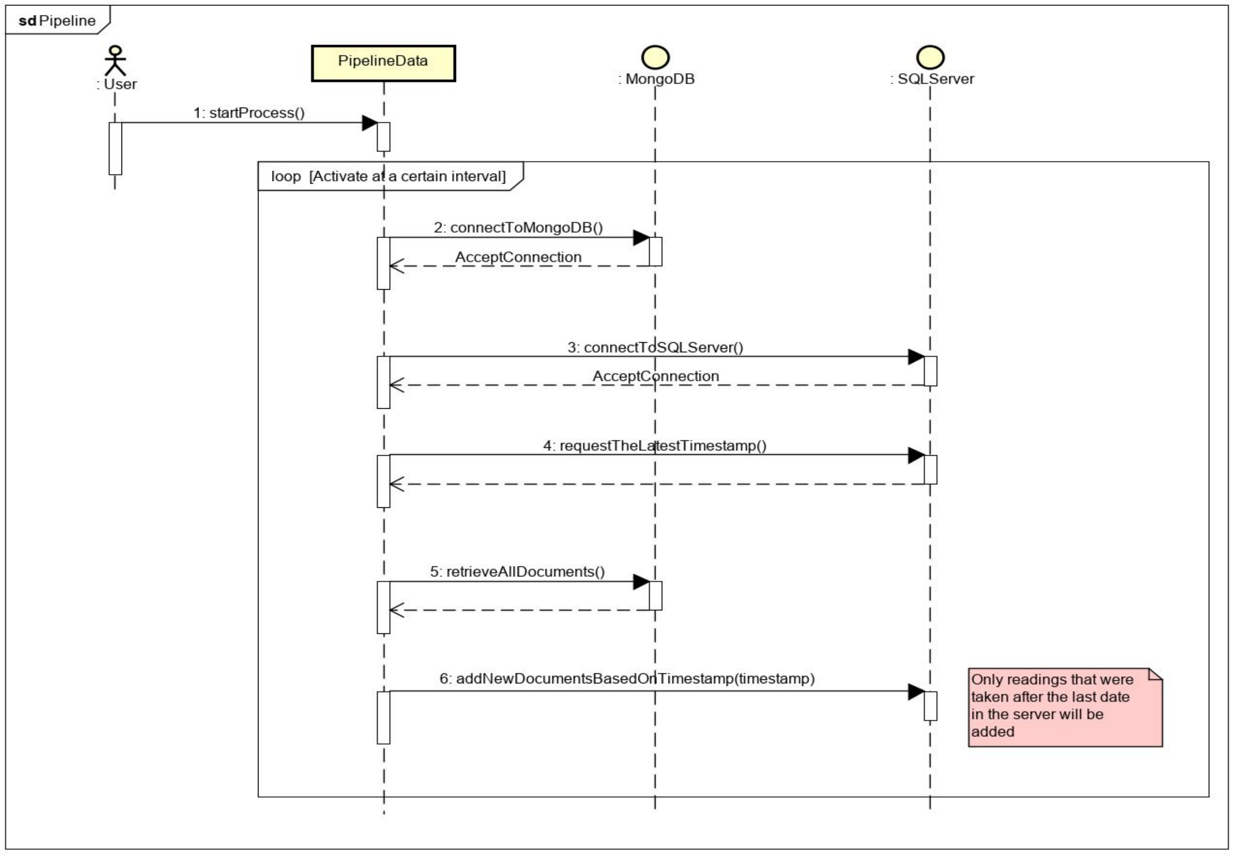

sd Pipeline
: User
PipelineData
: MongoDB
: SQLServer
1: startProcess()
loop [Activate at a certain interval]
2: connectToMongoDB()
AcceptConnection
3: connectToSQLServer()
AcceptConnection
4: requestTheLatestTimestamp()
5: retrieveAllDocuments()
6: addNewDocumentsBasedOnTimestamp(timestamp)
Only readings that were taken after the last date in the server will be added

Figure 4 in Data - Sequence Diagram for the Pipeline

Web API

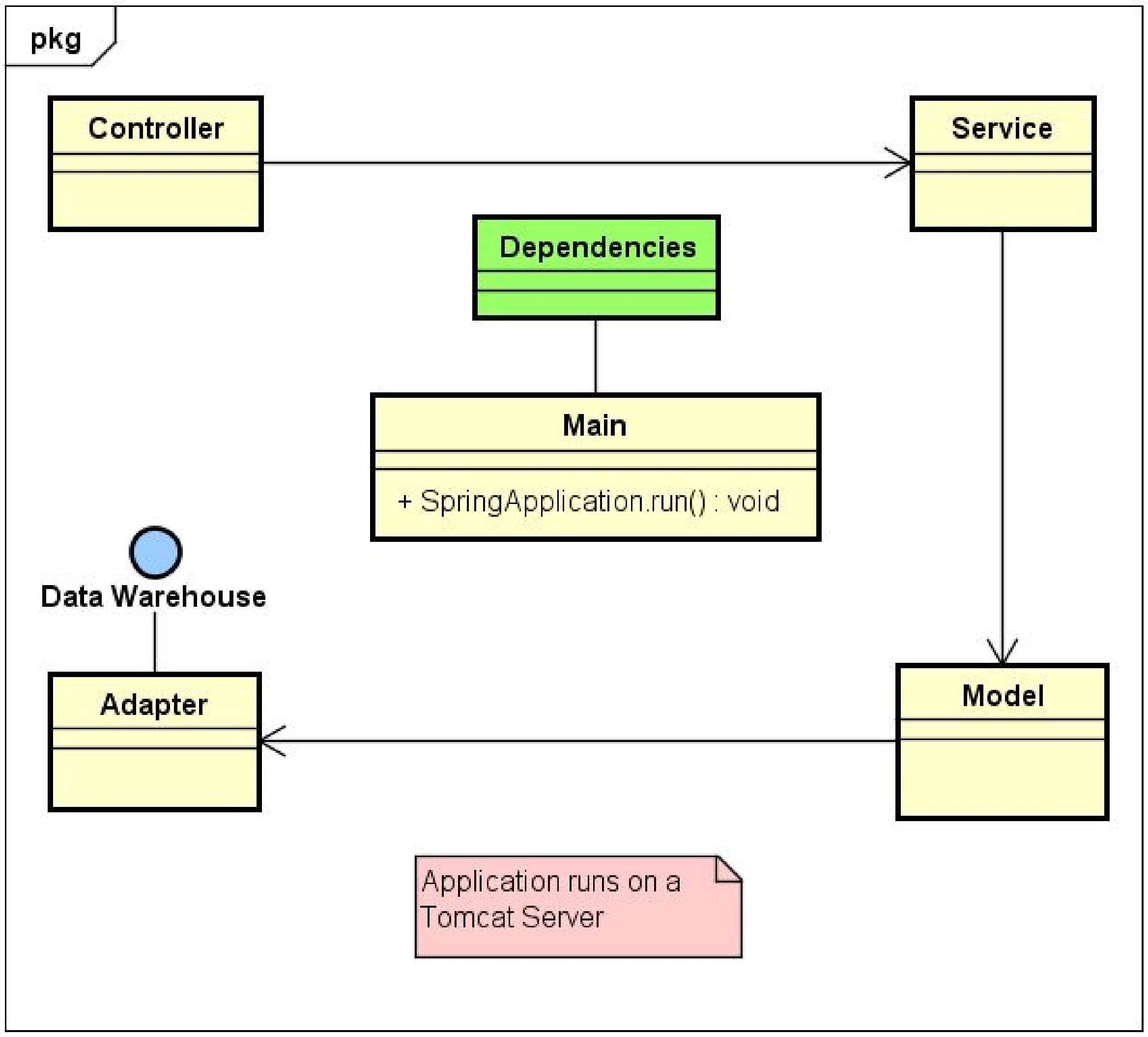

pkg
Controller
Service
Dependencies
Main
+ SpringApplication.run() : void
Data Warehouse
Adapter
Model
Application runs on a Tomcat Server

Figure 4 in Data - Diagram for API Design

- **The API design makes use of the Spring Boot framework and runs on a TomTom server.**

- **Dependencies are handled by the framework (Maven).**

- **JDBC is used for database access.**

Request monitoring is scheduled to be implemented as a server View, but currently has low priority and is not part of the design.

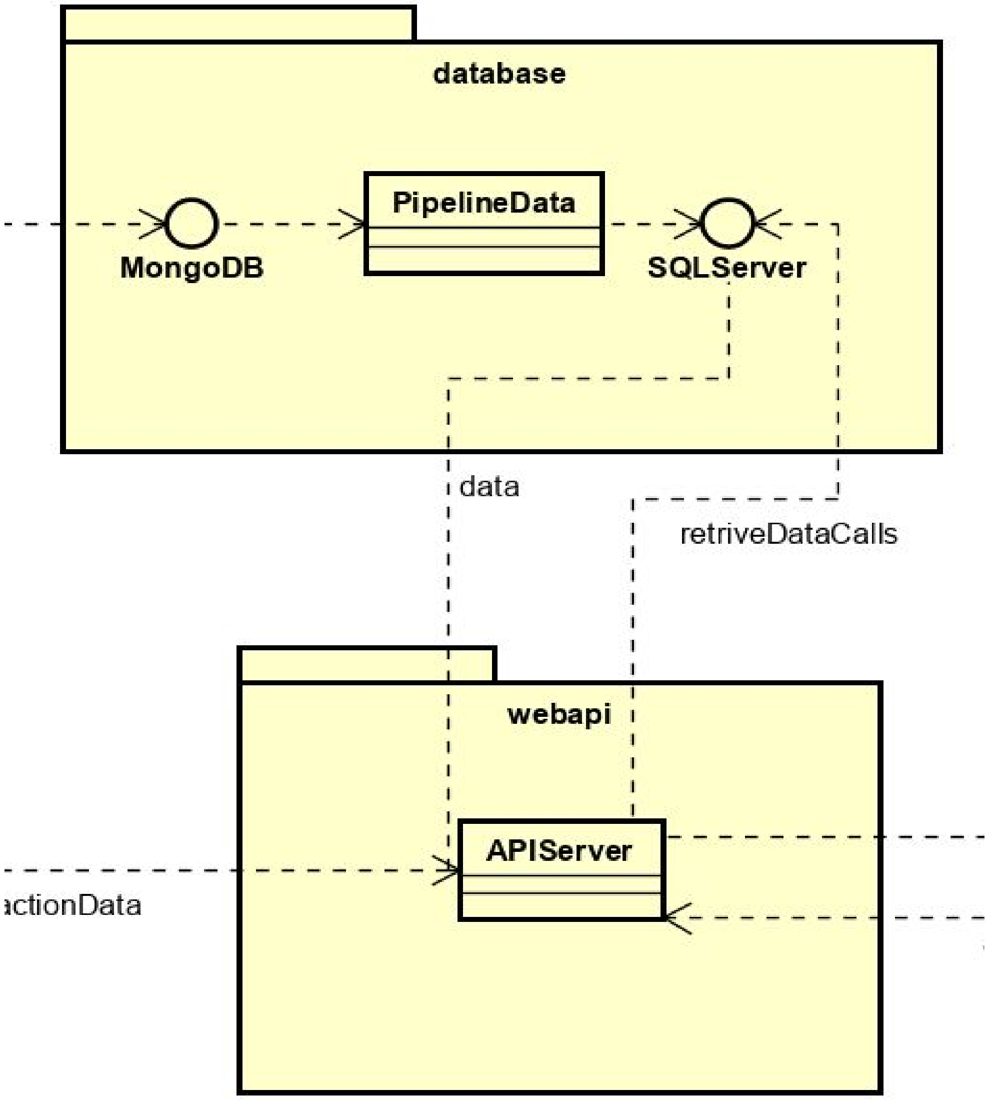


Figure 5 in Data - Data - Part of an IoT System

A result of the design is Figure 4 which is part of the bigger picture that consists the IoT System. To satisfy all the needs of the client, a web api is employed to provide services(visualization, provide data) related to the data stored in the NoSQL and SQL Servers.

Embedded system

(authors: Michal Ciebien, Dominika Kubicz)

The Bridge Application

The bridge application is a very important part of communication between the arduino, database and the API. It handles the cleans the data and sends the right information to the correct receiver.
It consists three classes handling communication with three different components. First one is the APIListener which is waiting for a call from the WebAPI. Second one is the LoRaClient which handles the websocket connection with the LoRaWan. It also uses the UplinkMessageFormatter which creates the right format of the message that will be sent further. Third one is the MongoDBHelper which sends the adequate data to the database.

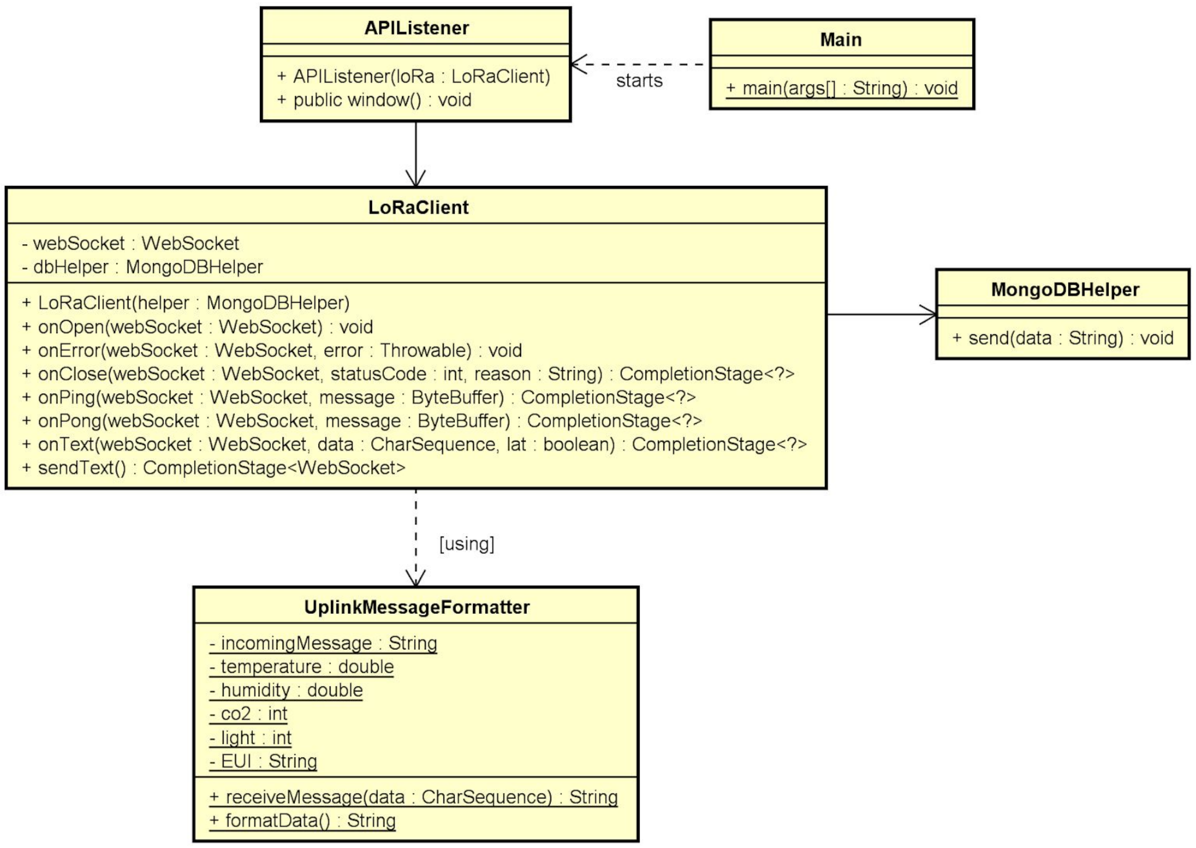

APIListener
+ APIListener(loRa : LoRaClient)
+ public window() : void
starts
Main
+ main(args[] : String) : void
LoRaClient
- webSocket : WebSocket
- dbHelper : MongoDBHelper
+ LoRaClient(helper : MongoDBHelper)
+ onOpen(webSocket : WebSocket) : void
+ onError(webSocket : WebSocket, error : Throwable) : void
+ onClose(webSocket : WebSocket, statusCode : int, reason : String) : CompletionStage<?>
+ onPing(webSocket : WebSocket, message : ByteBuffer) : CompletionStage<?>
+ onPong(webSocket : WebSocket, message : ByteBuffer) : CompletionStage<?>
+ onText(webSocket : WebSocket, data : CharSequence, lat : boolean) : CompletionStage<?>
+ sendText() : CompletionStage<WebSocket>
MongoDBHelper
+ send(data : String) : void
[using]
UplinkMessageFormatter
- incomingMessage : String
- temperature : double
- humidity : double
- co2 : int
- light : int
- EUI : String
+ receiveMessage(data : CharSequence) : String
+ formatData() : String

Figure 1 in Embedded System - UML for Bridge Application

In the diagram below the connection between the arduino and the bridge application is shown. The arduino sends the gathered data to the bridge application. When the user wants to open the window then the bridge application sends a notice to arduino. Both of these connections (sending and receiving) are handled by the LoRaClient (Figure 1).

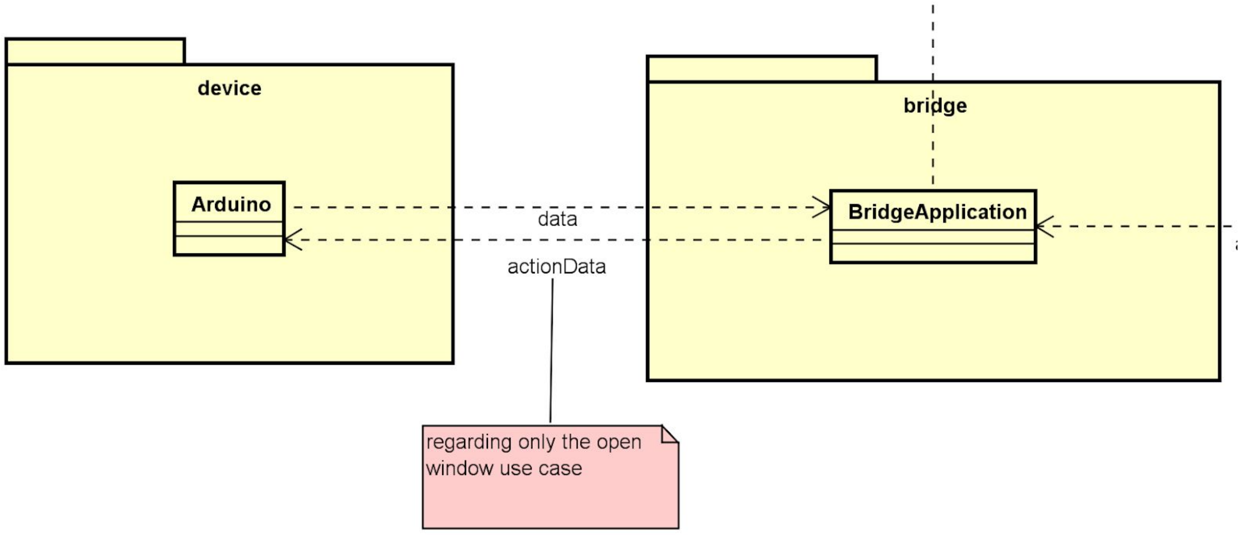


Figure 2 in Embedded System - Device & Bridge Connection

Figure 3 shows the connection between the APIServer and the bridge application. The APIListener handles this particular connection. It is listening for a call from the APIServer and when the call is made it handles it accordingly.

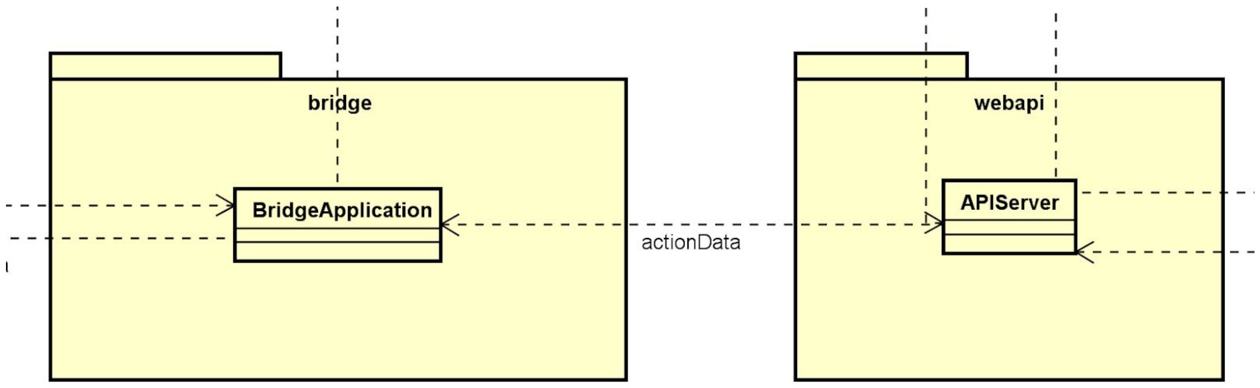


Figure 3 in Embedded System - Bridge & Web api Connection

The following diagram is focused on the connection with the MongoDB. The data that is meant to be sent first is cleaned by the UplinkMessageFormater. When the LoRaClient receives the readings back in the right format it will pass them to MongoDBHelper which sends the data to the database.

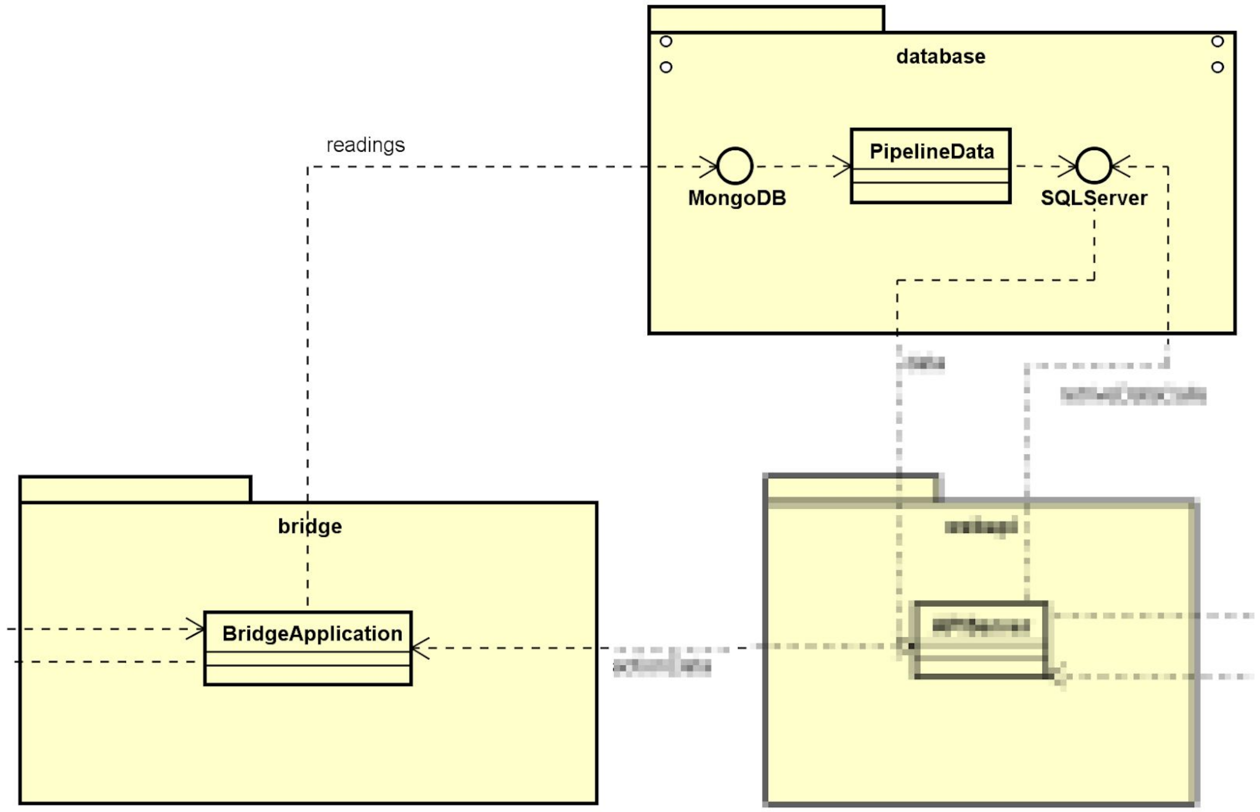


Figure 4 in Embedded System - Bridge & Database connection

Bridge Application Software

The bridge application is implemented using Java programing language. It is also done using Spring Boot framework and Maven.

The Arduino microcontroller

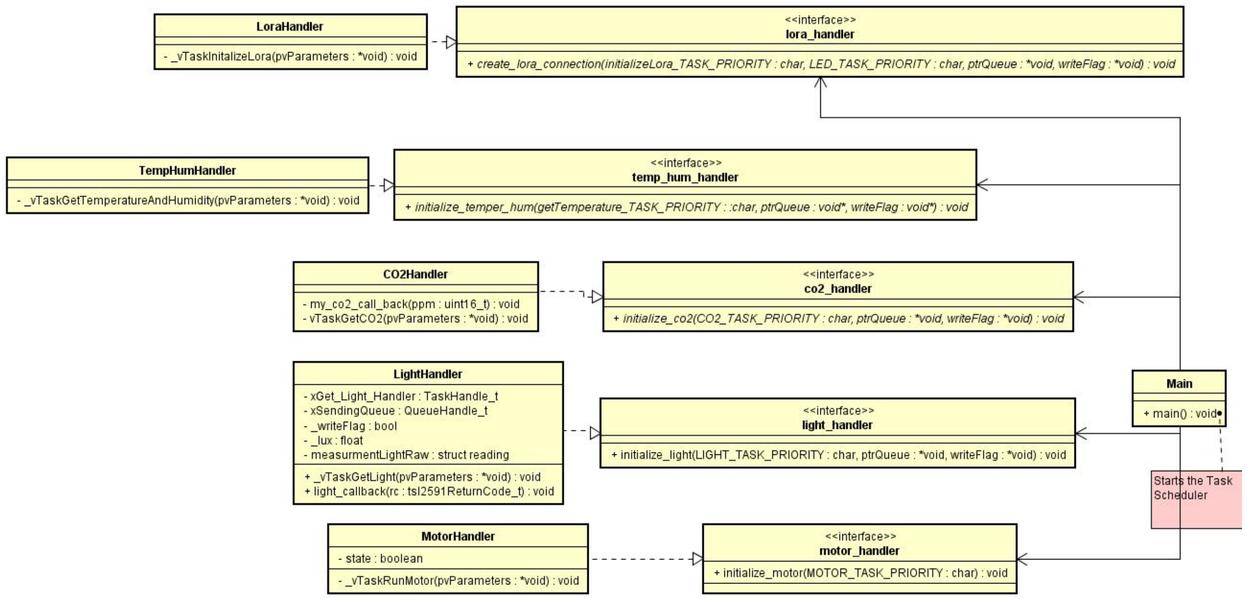

LoraHandler
- _vTaskInitalizeLora(pvParameters : *void) : void
<<interface>>
lora_handler
+ create_lora_connection(initializeLora_TASK_PRIORITY : char, LED_TASK_PRIORITY : char, ptrQueue : *void, writeFlag : *void) : void
TempHumHandler
- _vTaskGetTemperatureAndHumidity(pvParameters : *void) : void
<<interface>>
temp_hum_handler
+ initialize_temper_hum(getTemperature_TASK_PRIORITY : :char, ptrQueue : void*, writeFlag : void*) : void
CO2Handler
- my_co2_call_back(ppm : uint16_t) : void
- vTaskGetCO2(pvParameters : *void) : void
<<interface>>
co2_handler
+ initialize_co2(CO2_TASK_PRIORITY : char, ptrQueue : *void, writeFlag : *void) : void
LightHandler
- xGet_Light_Handler : TaskHandle_t
- xSendingQueue : QueueHandle_t
- _writeFlag : bool
- _lux : float
- measurmentLightRaw : struct reading
+ _vTaskGetLight(pvParameters : *void) : void
+ light_callback(rc : tsl2591ReturnCode_t) : void
<<interface>>
light_handler
+ initialize_light(LIGHT_TASK_PRIORITY : char, ptrQueue : *void, writeFlag : *void) : void
Main
+ main() : void
Starts the Task Scheduler
MotorHandler
- state : boolean
- _vTaskRunMotor(pvParameters : *void) : void
<<interface>>
motor_handler
+ initialize_motor(MOTOR_TASK_PRIORITY : char) : void

Figure 5 in Embedded System - UML for Arduino Microcontroller

The above diagram shows how the device architecture is structured. The Main class, responsible for starting the task scheduler can call initialize methods of different sensors and an actuator. This way, implementation of this methods can be done in the different ".c" files, which will improve the code structure and it's readiness. Specific sensors can initialize the drivers in the initializing methods and then create tasks, which will be started in the main method.

The final class diagram can be found in Appendix C.

The design of the sensors is presented on the following Activity Diagram.

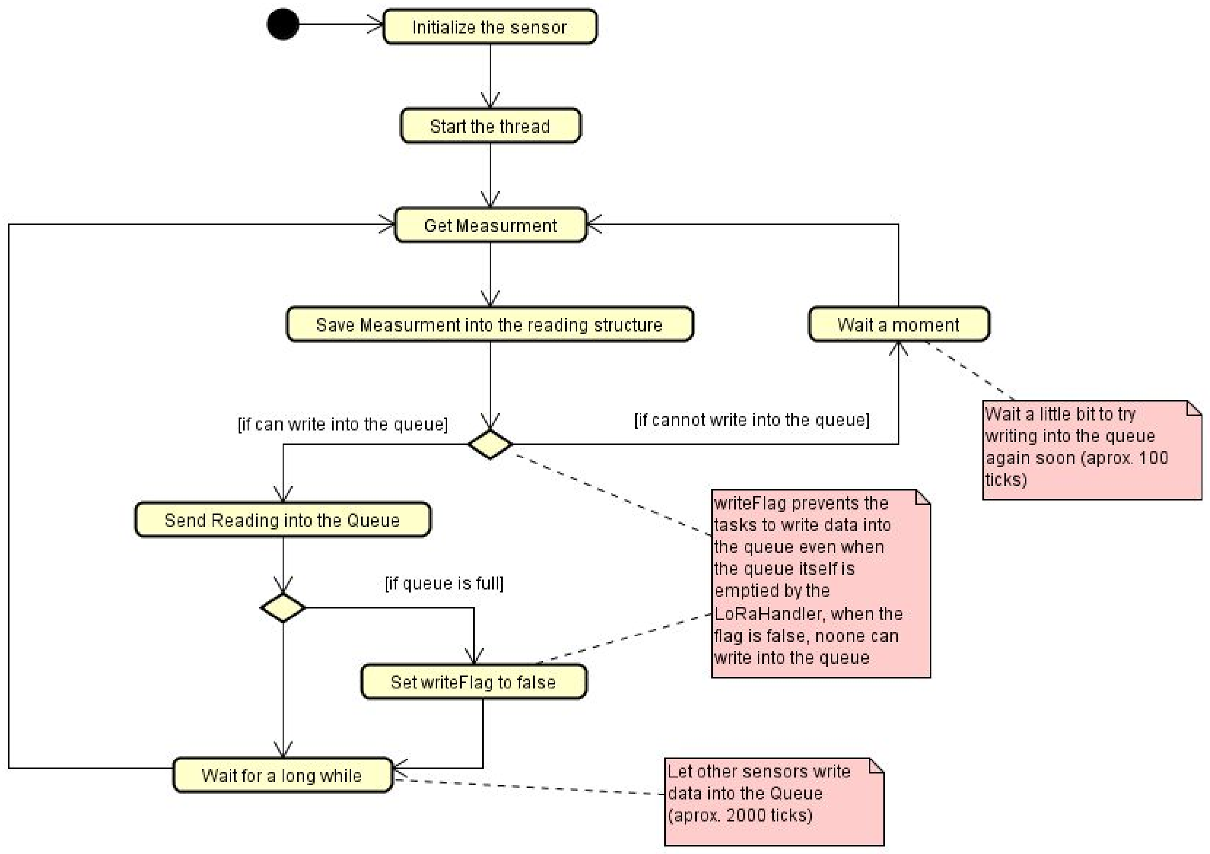

Initialize the sensor
Start the thread
Get Measurment
Save Measurment into the reading structure
Wait a moment
Wait a little bit to try writing into the queue again soon (aprox. 100 ticks)
[if can write into the queue]
[if cannot write into the queue]
Send Reading into the Queue
writeFlag prevents the tasks to write data into the queue even when the queue itself is emptied by the LoRaHandler, when the flag is false, noone can write into the queue
[if queue is full]
Set writeFlag to false
Wait for a long while
Let other sensors write data into the Queue (aprox. 2000 ticks)

Figure 7 in Embedded Implementation Sensors Activity Diagram

The diagram shows the work of the sensor and how different tasks should be handled including a thin abstraction layer to ease the understanding of the flow. After initialization of the task, the thread is being run. In the thread the task gets the measurement from the sensor, saves it in the formed structure and tries to write it into the queue. If it succeeds it checks if the queue is full and if so - sets the writeFlag to false. It then waits for a long while in order to allow other sensor tasks write their measurements into the queue.

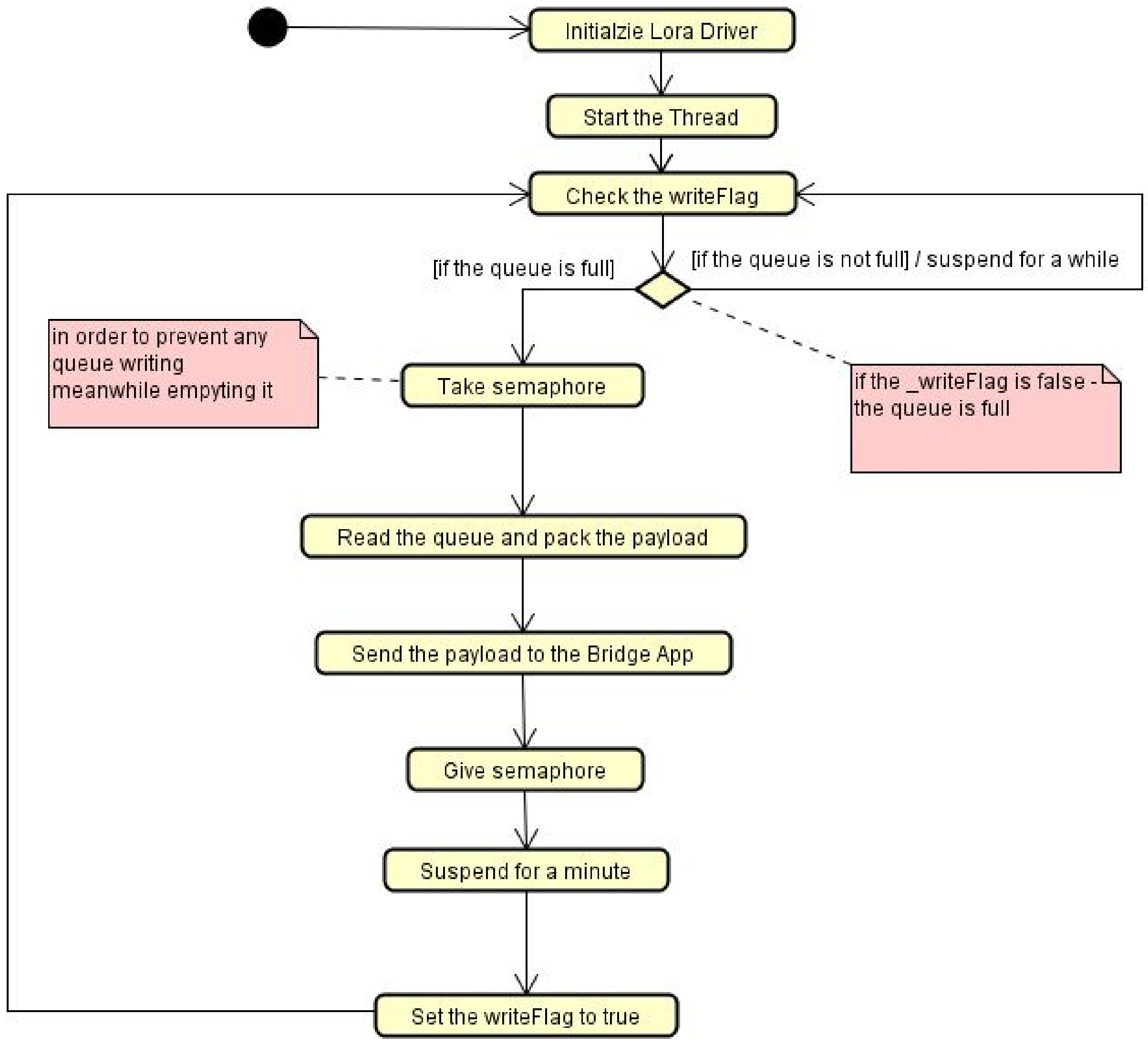

Initialzie Lora Driver
Start the Thread
Check the writeFlag
[if the queue is full]
[if the queue is not full] / suspend for a while
in order to prevent any queue writing meanwhile empyting it
Take semaphore
if the _writeFlag is false - the queue is full
Read the queue and pack the payload
Send the payload to the Bridge App
Give semaphore
Suspend for a minute
Set the writeFlag to true

Figure 8 in Embedded Implementation LoRaWAN Handler Activity Diagram

The above diagram shows the flow of the LoraHandler task. It is responsible for collecting data from different sensor tasks, packaging them into a fixed structure payload and then sending it to the Bridge Application. At first the LoRa driver should be initialized. After that the thread starts and checks the writeFlag (which indicates if the queue is full or empty). If the queue is empty, the thread gets suspended for a while and checks again. If the queue is full the task takes the semaphore, reads the queue and packs the payload to send it to LoRa Server. After the data is sent, the task gives out the semaphore allowing other tasks to run in parallel again. The semaphore is needed in this case to prevent any readings from emptying the queue[1] in the meantime.

Arduino Software and Hardware

In order to build the system, the decision to use uniquely designed VIA-board hardware was made. The board contains a list of components, including:

- HC-SR501 - PIR sensor
- HIH8120 - Temperature and Humidity Sensor
- RN2483 - LoRaWAN transmitter
- MH-Z19 - CO2 Sensor
- TSL2591 - Light Sensor

The VIA-Board also connects to the Atmel-ICE, which allows debugging of the system.

For the software part of the project the decision to use FreeRTOS was made. This tool provides us with means to develop in-parallel running software, which is one of the requirements of the device to work properly (especially important with regard of the actuator).

Conclusion to Design

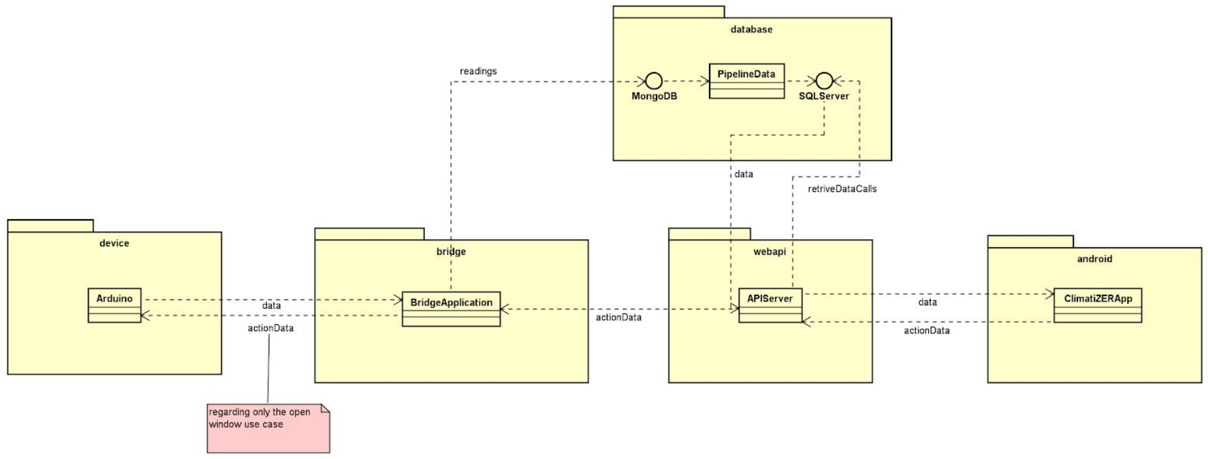


Figure 9 in Conclusion to Design - Design Diagram for IoT System

The conclusion of the design section comes as a diagram for the whole IoT System. It combines different aspects of the system (Embedded technology, Data and an application built for the Android Mobile OS). It showcases the design of a tool that can monitor information related to temperature, humidity and CO2 levels to create a better environment for books and people.

System Implementation

Data Implementation

(Authors: Claudiu Rediu, Mihail Kanchev, Stefan Harabagiu)

ETL Process

Initial Load

After the design of the dimensional model was created, the tables had to be populated using the data already existent in the source database. This process of initial load of the data warehouse also encompassed populating the date and time dimensions with auto-generated data. In order to conduct this process, it was needed to create two staging tables to serve as intermediaries between the source database and the data warehouse. The following activity diagram offers an overview of the initial load:

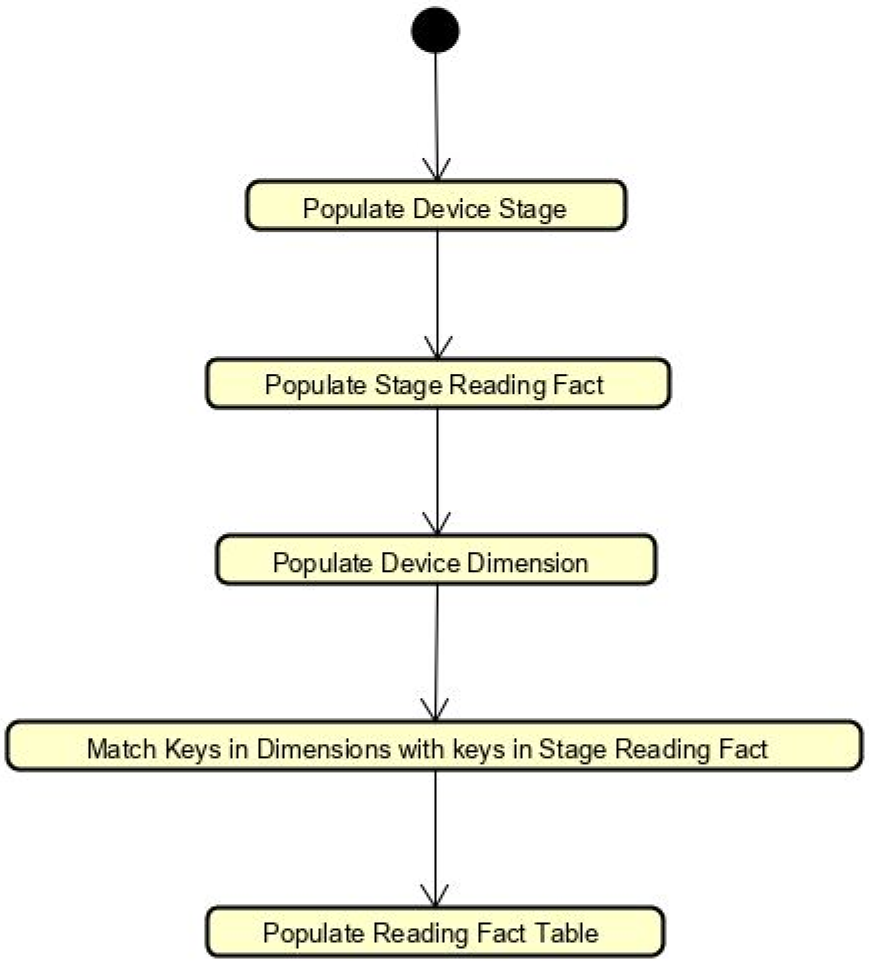
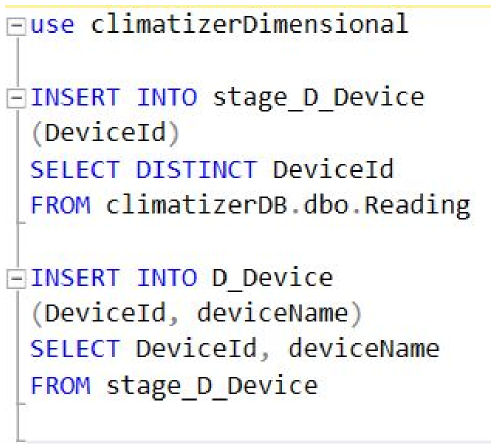

Populate Device Stage
Populate Stage Reading Fact
Populate Device Dimension
Match Keys in Dimensions with keys in Stage Reading Fact
Populate Reading Fact Table

Figure 1 in Data Implementation Initial Load

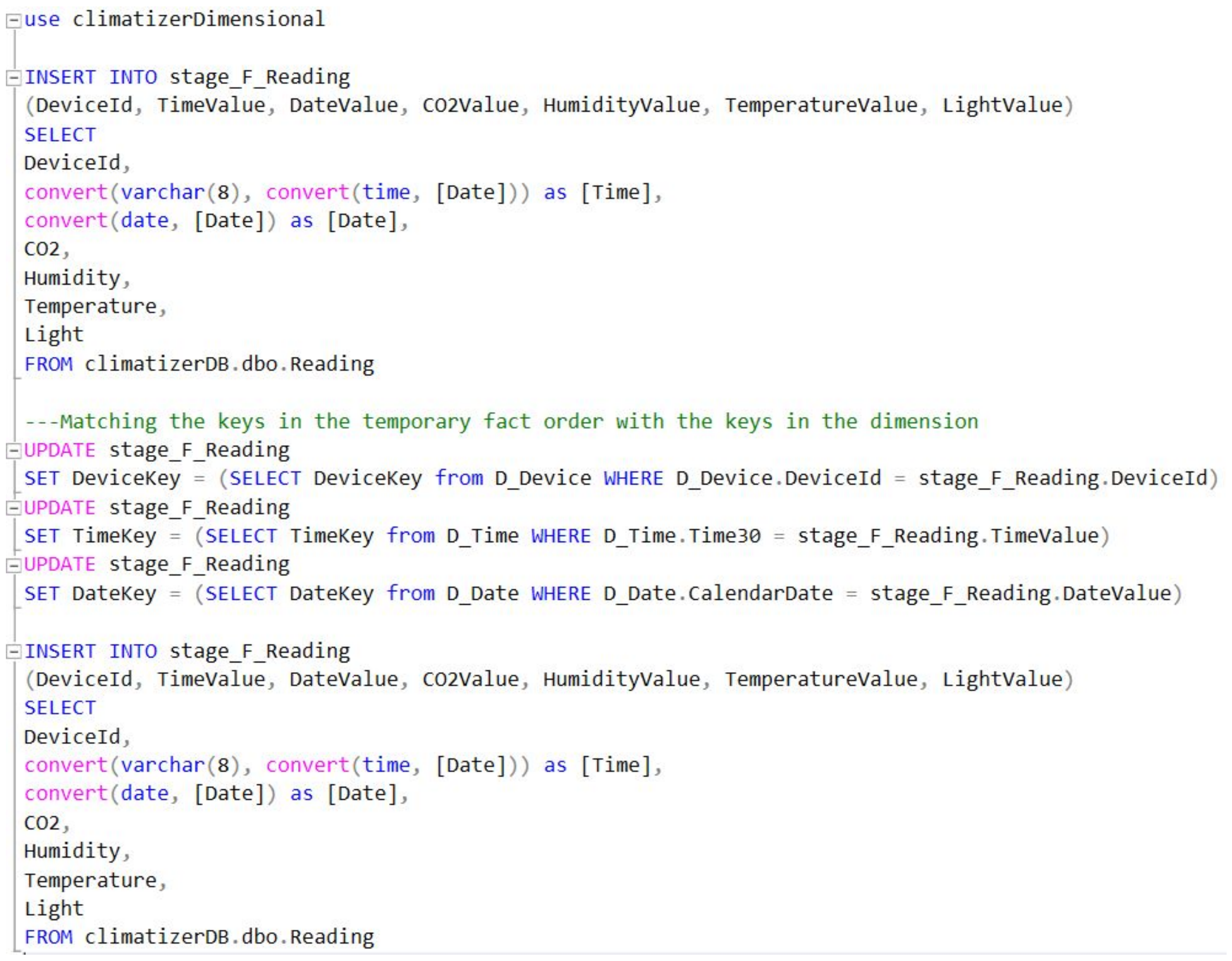
The first part is populating the staging table for the date dimension and then taking that data and putting it in the actual dimension.



```sql
use climatizerDimensional

INSERT INTO stage_D_Device
(DeviceId)
SELECT DISTINCT DeviceId
FROM climatizerDB.dbo.Reading

INSERT INTO D_Device
(DeviceId, deviceName)
SELECT DeviceId, deviceName
FROM stage_D_Device
```

Figure 2 in Data Implementation Insert Statement

The next part is inserting data in the fact staging table, matching the surrogate keys and then populating the fact table.

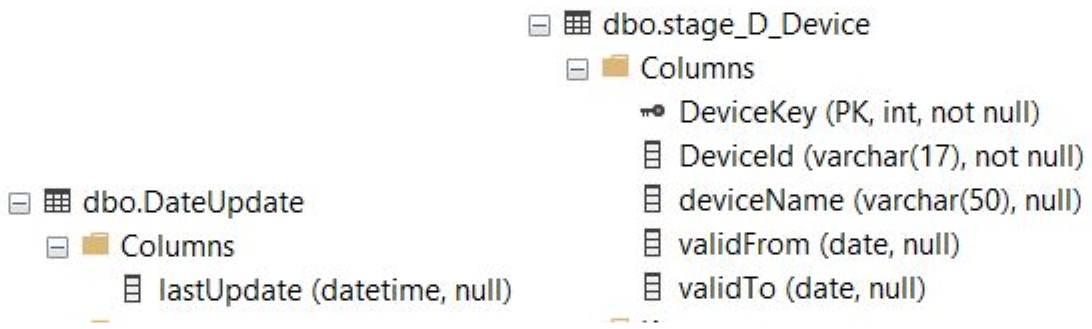


```sql
use climatizerDimensional

INSERT INTO stage_F_Reading
(DeviceId, TimeValue, DateValue, CO2Value, HumidityValue, TemperatureValue, LightValue)
SELECT
DeviceId,
convert(varchar(8), convert(time, [Date])) as [Time],
convert(date, [Date]) as [Date],
CO2,
Humidity,
Temperature,
Light
FROM climatizerDB.dbo.Reading

---Matching the keys in the temporary fact order with the keys in the dimension
UPDATE stage_F_Reading
SET DeviceKey = (SELECT DeviceKey from D_Device WHERE D_Device.DeviceId = stage_F_Reading.DeviceId)
UPDATE stage_F_Reading
SET TimeKey = (SELECT TimeKey from D_Time WHERE D_Time.Time30 = stage_F_Reading.TimeValue)
UPDATE stage_F_Reading
SET DateKey = (SELECT DateKey from D_Date WHERE D_Date.CalendarDate = stage_F_Reading.DateValue)

INSERT INTO stage_F_Reading
(DeviceId, TimeValue, DateValue, CO2Value, HumidityValue, TemperatureValue, LightValue)
SELECT
DeviceId,
convert(varchar(8), convert(time, [Date])) as [Time],
convert(date, [Date]) as [Date],
CO2,
Humidity,
Temperature,
Light
FROM climatizerDB.dbo.Reading
```

Figure 3 in Data Implementation Inserting in Reading Stage

Incremental Load

The next step in setting up the data warehouse is making the dimensional model capable of handling changes to the source database. Therefore, two new columns were added to the device dimension (validFrom and validTo), and a new auxiliary table, with one row and one column, was created to help keep track of the last time an update was performed on the warehouse.



dbo.DateUpdate
Columns
lastUpdate (datetime, null)

dbo.stage_D_Device
Columns
DeviceKey (PK, int, not null)
DeviceId (varchar(17), not null)
deviceName (varchar(50), null)
validFrom (date, null)
validTo (date, null)

Figures 4 in Data Implementation Date of the last update and device dimension

The following activity diagram was created to give an overview of an update of the data warehouse.

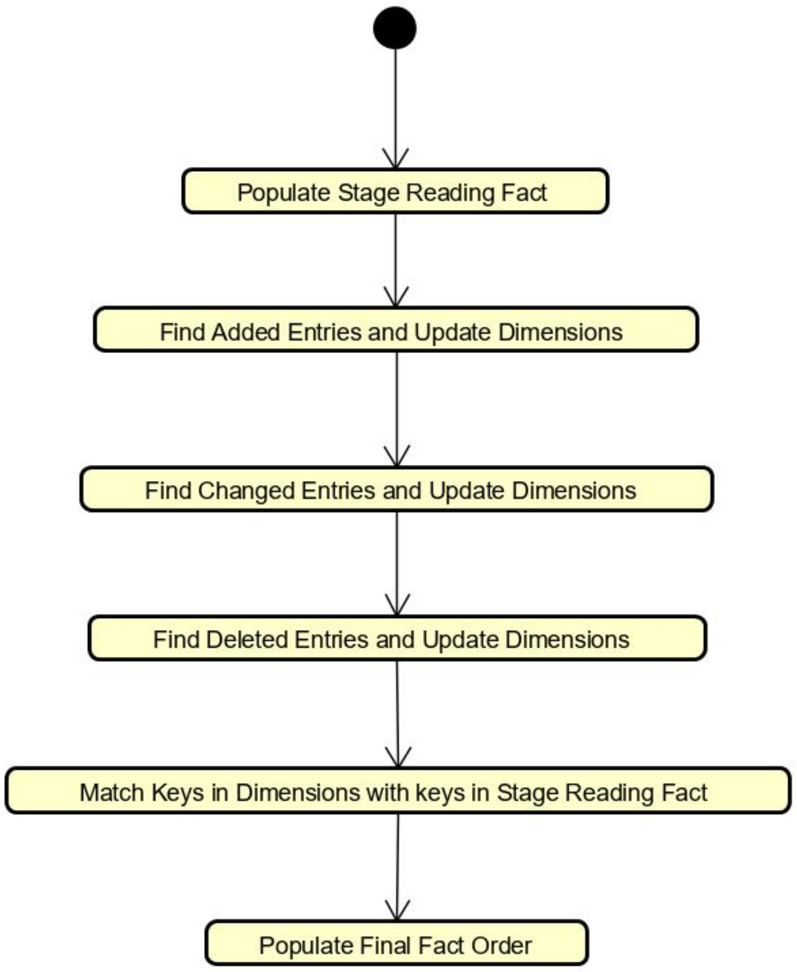
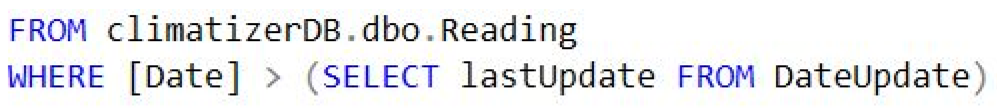

Populate Stage Reading Fact
Find Added Entries and Update Dimensions
Find Changed Entries and Update Dimensions
Find Deleted Entries and Update Dimensions
Match Keys in Dimensions with keys in Stage Reading Fact
Populate Final Fact Order

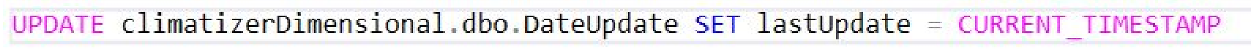
Figure 5 in Data Implementation Incremental Load

All that is left is writing the queries for detecting changes in the device table in the source database (new device added, device deleted or device data changed) and for checking if there are any new readings in the source database (by using lastUpdate).

The query for updating the fact table is similar to the one from the initial load with the addition of a date check and updating the DateUpdate table.



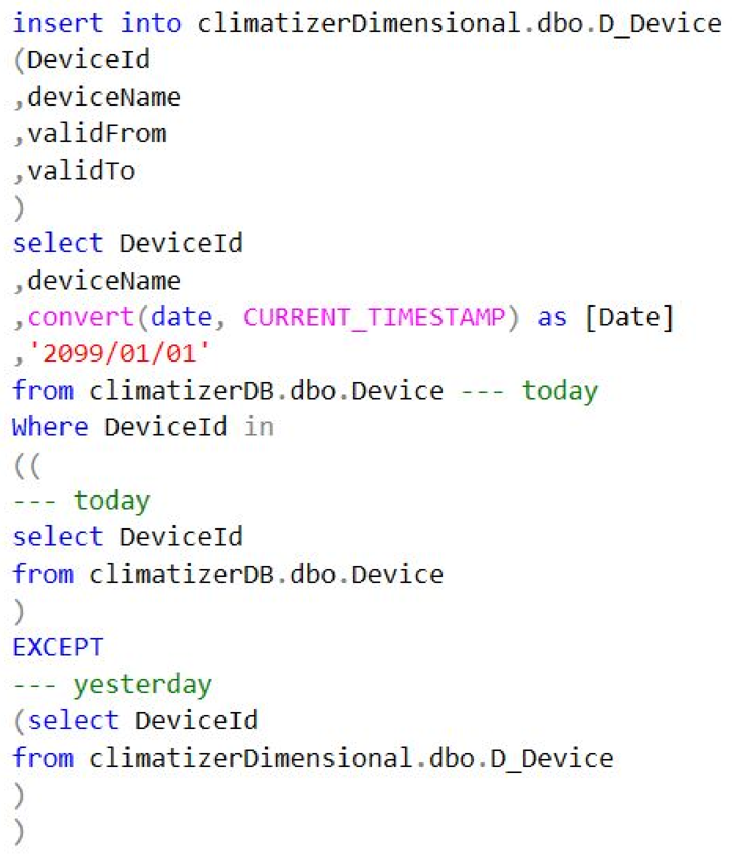
```
FROM climatizerDB.dbo.Reading
WHERE [Date] > (SELECT lastUpdate FROM DateUpdate)
```



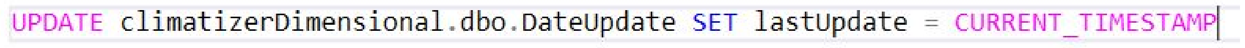
```
UPDATE climatizerDimensional.dbo.DateUpdate SET lastUpdate = CURRENT_TIMESTAMP
```

Figures 6 in Data Implementation Adding to the Reading table

In order to showcase the use of the validFrom and validTo columns, the query for detecting new devices in the source database and adding them to the dimension is presented below.



```
insert into climatizerDimensional.dbo.D_Device
(DeviceId
,deviceName
,validFrom
,validTo
)
select DeviceId
,deviceName
,convert(date, CURRENT_TIMESTAMP) as [Date]
,'2099/01/01'
from climatizerDB.dbo.Device --- today
Where DeviceId in
((
--- today
select DeviceId
from climatizerDB.dbo.Device
)
EXCEPT
--- yesterday
(select DeviceId
from climatizerDimensional.dbo.D_Device
)
)
```



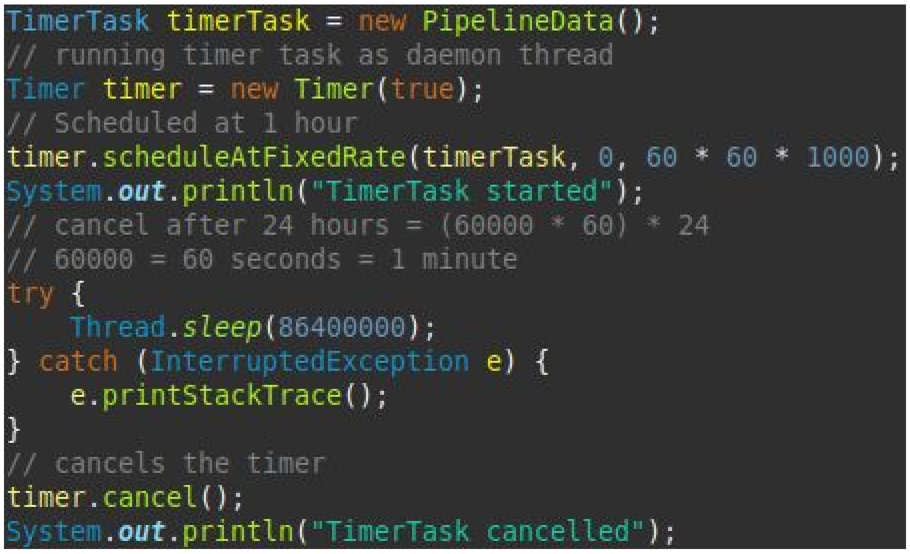
```
UPDATE climatizerDimensional.dbo.DateUpdate SET lastUpdate = CURRENT_TIMESTAMP
```

Figure 7 in Data Implementation Inserting in the Device dimension

The new row in the dimension will have the validFrom attribute set to the current timestamp and the validTo attribute to something far off in the future.

JAVA Pipeline

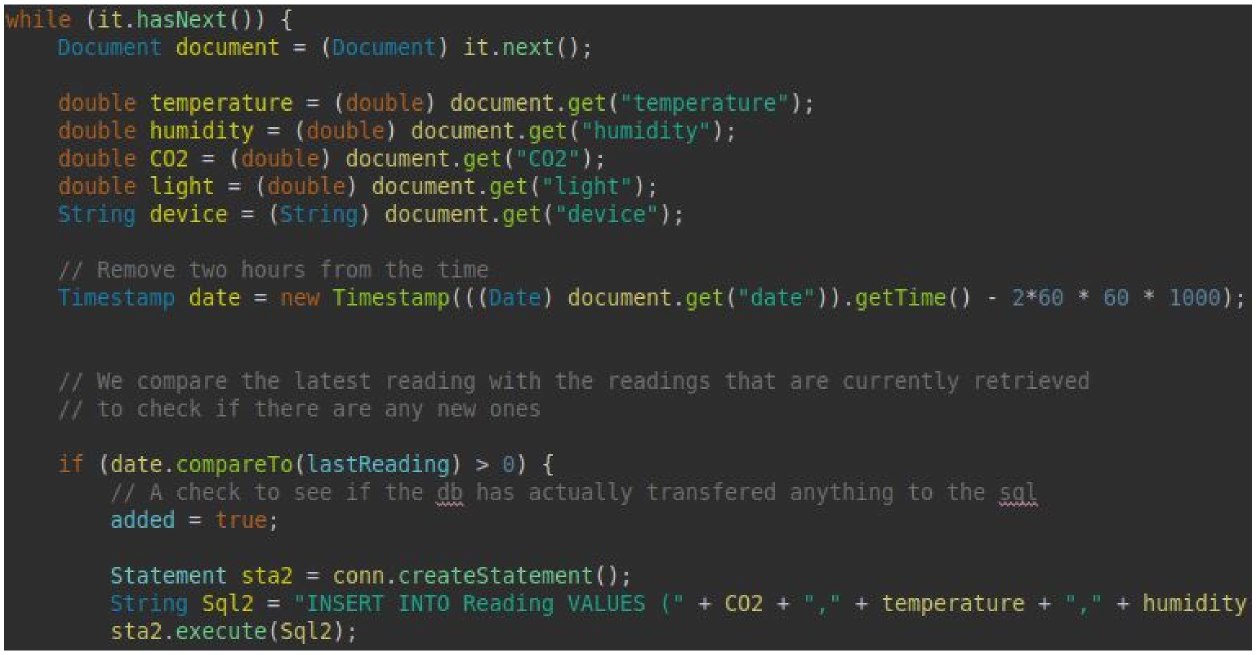
The pipeline has been written in JAVA and it uses the TimerTask to perform the transfer at one hour. It will cancel itself after 24 hours. This might be changed in the future depending on the needs.



```java
TimerTask timerTask = new PipelineData();
// running timer task as daemon thread
Timer timer = new Timer(true);
// Scheduled at 1 hour
timer.scheduleAtFixedRate(timerTask, 0, 60 * 60 * 1000);
System.out.println("TimerTask started");
// cancel after 24 hours = (60000 * 60) * 24
// 60000 = 60 seconds = 1 minute
try {
    Thread.sleep(86400000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
// cancels the timer
timer.cancel();
System.out.println("TimerTask cancelled");
```

Figure 8 in Data Implementation - Setting up the TimerTask

The time interval in which it happens is determined by how much time it has until it cancels itself. This is why the thread is put to sleep for 24 hours. It is scheduled at a fixed rate in the current time frame, so it will synchronize with the ETL process.



```java
while (it.hasNext()) {
    Document document = (Document) it.next();

    double temperature = (double) document.get("temperature");
    double humidity = (double) document.get("humidity");
    double CO2 = (double) document.get("CO2");
    double light = (double) document.get("light");
    String device = (String) document.get("device");

    // Remove two hours from the time
    Timestamp date = new Timestamp(((Date) document.get("date")).getTime() - 2*60 * 60 * 1000);


    // We compare the latest reading with the readings that are currently retrieved
    // to check if there are any new ones

    if (date.compareTo(lastReading) > 0) {
        // A check to see if the db has actually transfered anything to the sql
        added = true;

        Statement sta2 = conn.createStatement();
        String Sql2 = "INSERT INTO Reading VALUES (" + CO2 + "," + temperature + "," + humidity
        sta2.execute(Sql2);
```

Figure 9 in Data Implementation - Adding the next documents

To make sure that only new documents are added, the date in the document is compared with the latest date that has been added to the source database. After retrieving all the values from the document, they are put in an SQL statements that executes and inserts a new row of data in the SQL database.

JAVA Web API

Java was the chosen language for the web services because of the control a developer has over the sent and received requests. Upon further analysis the team settled on the *Spring Boot* framework as it provides all functionality that is required and the code can easily be read because of how abstract the framework is.

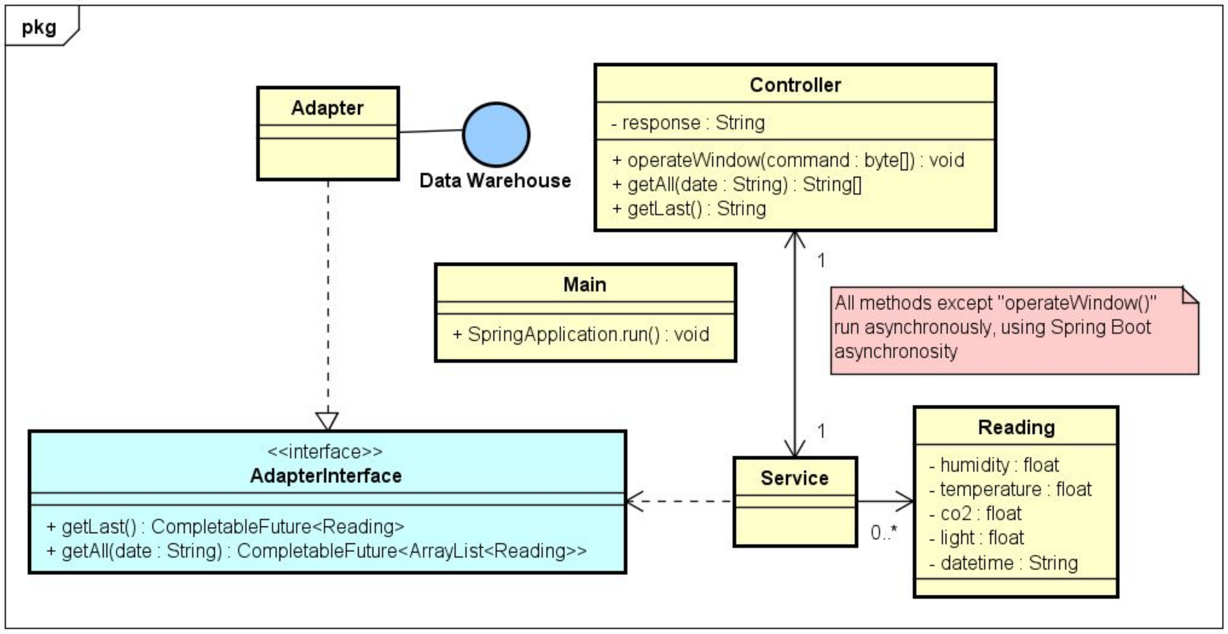
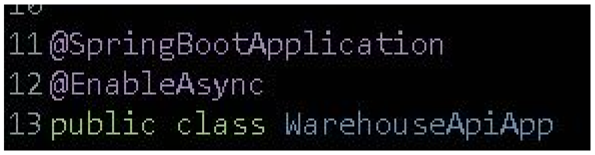
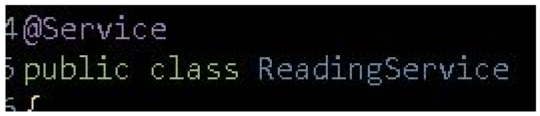

pkg
Adapter
Data Warehouse
Controller
- response : String
+ operateWindow(command : byte[]) : void
+ getAll(date : String) : String[]
+ getLast() : String
1
Main
+ SpringApplication.run() : void
All methods except "operateWindow()" run asynchronously, using Spring Boot asynchronosity
1
<<interface>>
AdapterInterface
+ getLast() : CompletableFuture<Reading>
+ getAll(date : String) : CompletableFuture<ArrayList<Reading>>
Service
0..*
Reading
- humidity : float
- temperature : float
- co2 : float
- light : float
- datetime : String

Figure 10 in Data Implementation

System responsibility is split amongst different components. To distinguish all components, the *Spring Boot* framework makes use of annotations to mark its classes.



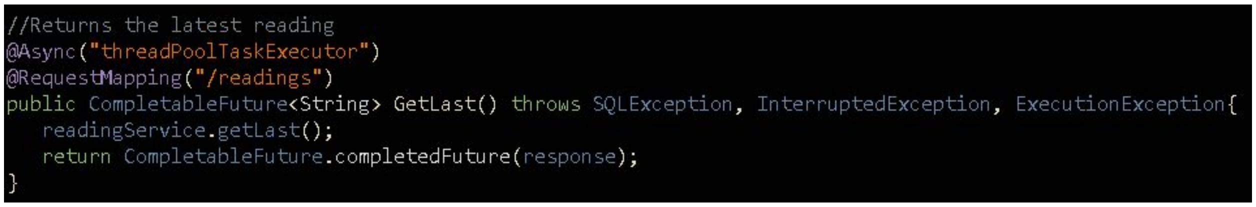
```
@SpringBootApplication
@EnableAsync
public class WarehouseApiApp
```



```
@Service
public class ReadingService
```

Figures 11 in Data Implementation

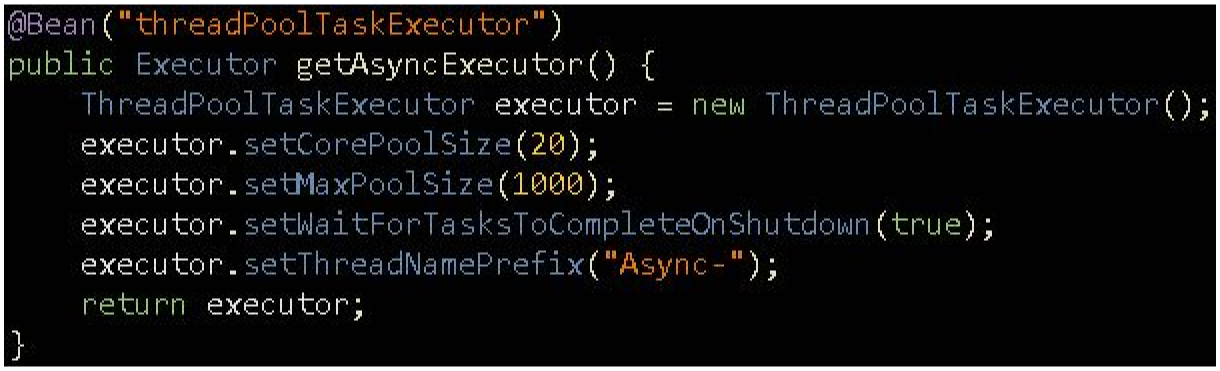
Controller handles request types. It holds an instance of **Service** in order to call the appropriate methods. A private String value **“response”** holds any requested response and is sent upon response completion. The decision behind that field was the asynchronous design of the system.



```
//Returns the latest reading
@Async("threadPoolTaskExecutor")
@RequestMapping("/readings")
public CompletableFuture<String> GetLast() throws SQLException, InterruptedException, ExecutionException{
   readingService.getLast();
   return CompletableFuture.completedFuture(response);
}
```

Figure 12 in Data Implementation

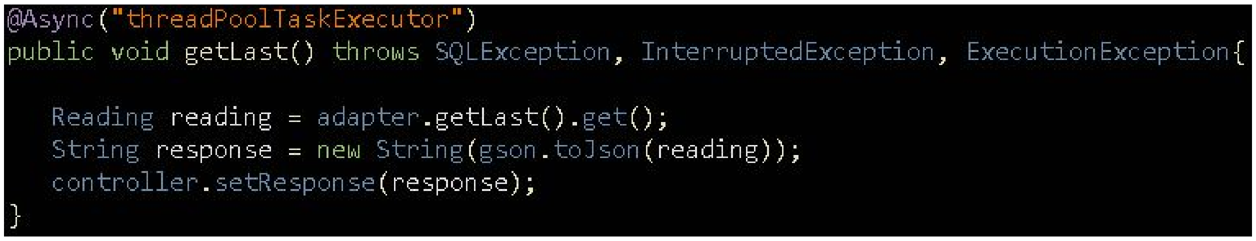
Shown on the picture above is how a simple *CRUD* request is handled. The default request being *GET* and the address being *"ip:port/readings"*. The "*Async*" annotation marks the method as being asynchronous and sets a custom task executor. *RequestMapping* handles the path of the request as a single controller is used for multiple requests towards different addresses. Upon method initiation, the instance of **Service** gets used by calling **getLast()**, which upon completion writes the result onto the local **response** String.



```
@Bean("threadPoolTaskExecutor")
public Executor getAsyncExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(20);
    executor.setMaxPoolSize(1000);
    executor.setWaitForTasksToCompleteOnShutdown(true);
    executor.setThreadNamePrefix("Async-");
    return executor;
}
```

Figure 13 in Data Implementation

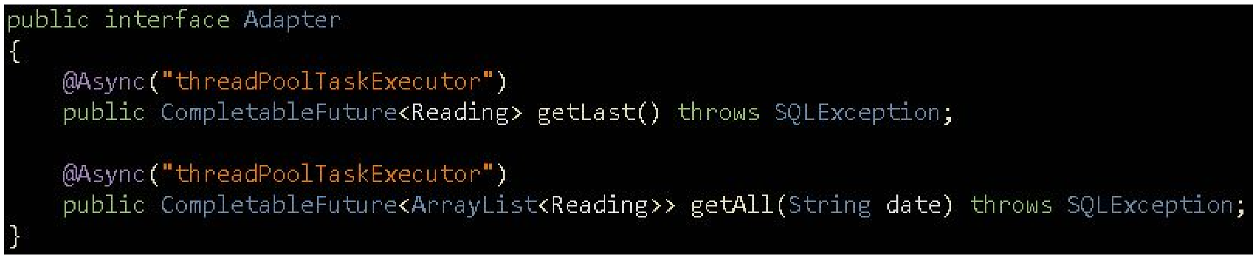
A custom *ThreadExecutor* is used. The "*Bean*" annotation tells the *Spring Boot* framework to initiate the following code upon the initiation of a **Service** class.



```
@Async("threadPoolTaskExecutor")
public void getLast() throws SQLException, InterruptedException, ExecutionException{

   Reading reading = adapter.getLast().get();
   String response = new String(gson.toJson(reading));
   controller.setResponse(response);
}
```

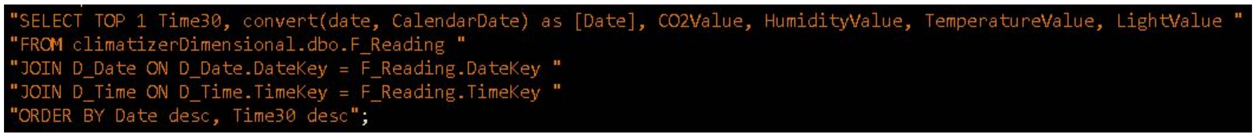
Figure 14 in Data Implementation

The request continues further down the system with the **Service** class. The **Service** class holds an instance of **Controller** in order to get access to its local **response** String. Inside the method the **Model** is being used and the **Adapter** is being called. Whenever the **.get()** method gets called, the entire operation starts on another thread and the method is paused until a result is received.



```java
public interface Adapter
{
    @Async("threadPoolTaskExecutor")
    public CompletableFuture<Reading> getLast() throws SQLException;

    @Async("threadPoolTaskExecutor")
    public CompletableFuture<ArrayList<Reading>> getAll(String date) throws SQLException;
}
```

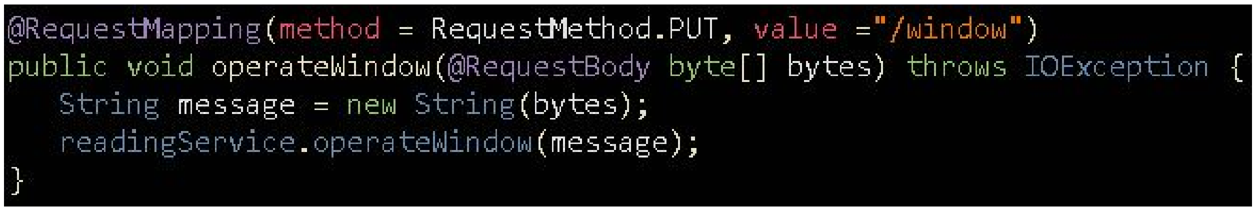
The Adapter interface can be seen on the picture above. It too makes use of the asynchronosity of Spring Boot. It connects to the dimensional database and queries the requests.



```
"SELECT TOP 1 Time30, convert(date, CalendarDate) as [Date], CO2Value, HumidityValue, TemperatureValue, LightValue "
"FROM climatizerDimensional.dbo.F_Reading "
"JOIN D_Date ON D_Date.DateKey = F_Reading.DateKey "
"JOIN D_Time ON D_Time.TimeKey = F_Reading.TimeKey "
"ORDER BY Date desc, Time30 desc";
```

An example of a query

Database querying makes use of JDBC.



```java
@RequestMapping(method = RequestMethod.PUT, value ="/window")
public void operateWindow(@RequestBody byte[] bytes) throws IOException {
    String message = new String(bytes);
    readingService.operateWindow(message);
}
```

As it can be seen above, the window operation was designed and implemented but not finalized due to Bridge application complications. Currently the signal is received from the Android device but no further action is taken.

Embedded Implementation

(authors: Michal Ciebien, Dominika Kubicz)

Implementation details of the Bridge Application

One of the Use Cases states that the user should be able to read climate conditions. The bridge application in order to complete this use case has to transfer data from the LoRaWan to the Mongo

Database server. The diagram below shows steps that are taken in order to send the correct information the database.

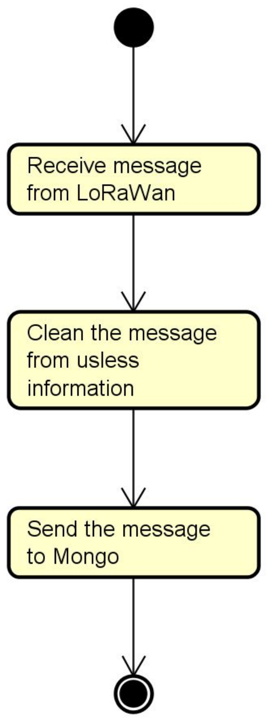
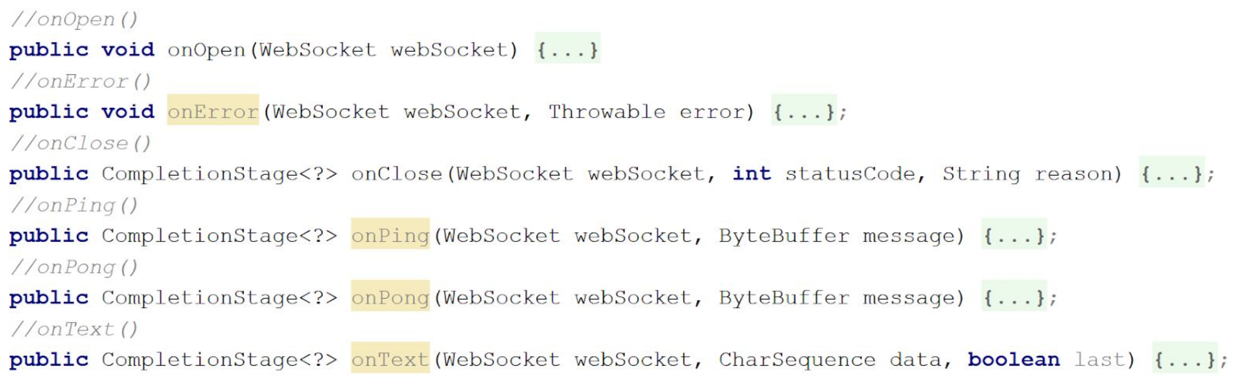

Receive message from LoRaWan
Clean the message from usless information
Send the message to Mongo

Figure 1 in Embedded Implementation Sending Message to Mongo Activity Diagram

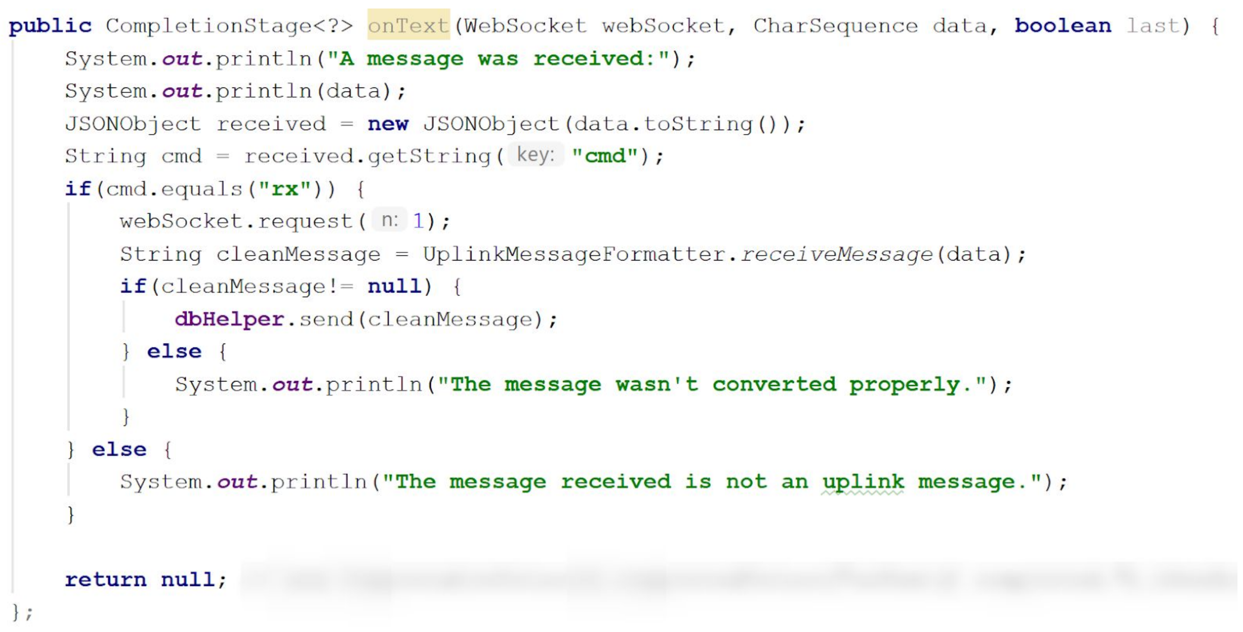
The message from LoRaWan is received through a websocket. The websocket is initialized through dependency injection. The websocket connection is waiting for LoRaWan to invoke its methods. The methods that can be invoked are shown in *Figure 2 in Embedded Implementation.* In case of receiving message the onText method is called.



```
//onOpen()
public void onOpen(WebSocket webSocket) {...}
//onError()
public void onError(WebSocket webSocket, Throwable error) {...};
//onClose()
public CompletionStage<?> onClose(WebSocket webSocket, int statusCode, String reason) {...};
//onPing()
public CompletionStage<?> onPing(WebSocket webSocket, ByteBuffer message) {...};
//onPong()
public CompletionStage<?> onPong(WebSocket webSocket, ByteBuffer message) {...};
//onText()
public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {...};
```

Figure 2 in Embedded Implementation - Methods Which Can Invoked By LoRaWan

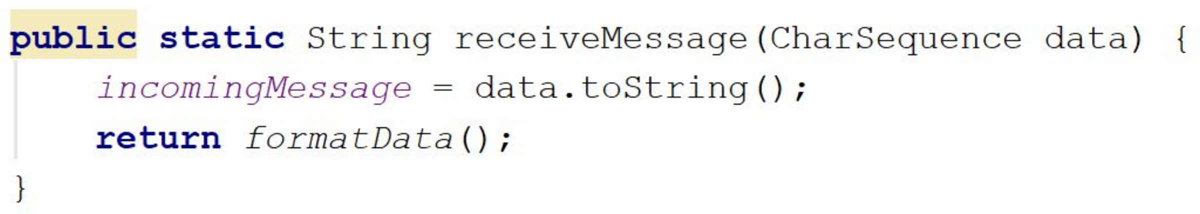
The method onText first creates an Json object out of the received data. Next the “cmd” field is checked whether it is “rx”. It needs to be checked in order to establish, if this is an uplink message. In the second case the “cmd” field would have a different value. Next, using the UplinkMessageFormatter class the data is cleaned and by calling the send message it is passed to the dbHelper which sends it to the database.



```
public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {
    System.out.println("A message was received:");
    System.out.println(data);
    JSONObject received = new JSONObject(data.toString());
    String cmd = received.getString( key: "cmd");
    if(cmd.equals("rx")) {
        webSocket.request( n: 1);
        String cleanMessage = UplinkMessageFormatter.receiveMessage(data);
        if(cleanMessage!= null) {
            dbHelper.send(cleanMessage);
        } else {
            System.out.println("The message wasn't converted properly.");
        }
    } else {
        System.out.println("The message received is not an uplink message.");
    }

    return null;
};
```

Figure 3 in Embedded Implementation - OnText Method Implementation

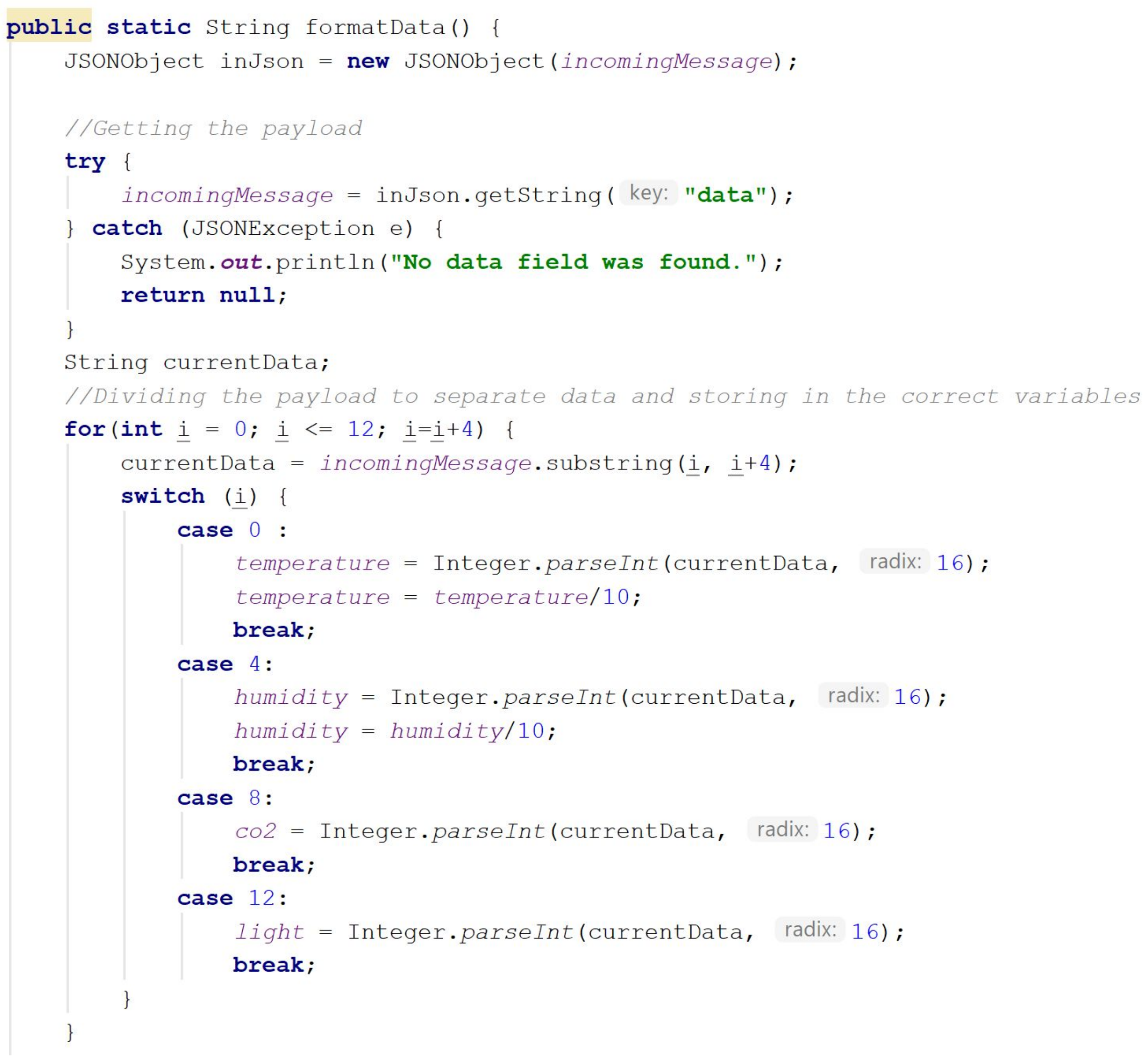
The receiveMessage method takes the charSequence that is given by the LoRaClient and changes it to string format. Next the formatData method is called which cleans and formats the data.



```
public static String receiveMessage(CharSequence data) {
    incomingMessage = data.toString();
    return formatData();
}
```

Figure 4 in Embedded Implementation - receiveMessage Method Implementation

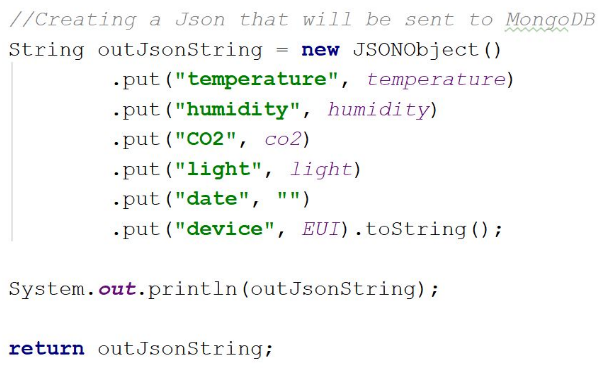
The data received from lora is formatted in Json. So a JSONObject is created. The only data that needs to best to the database are the measurements taken by the device which are passed in the data field. The measurements are put in the incomingMessage variable. Since they are sent in hexadecimal they need to be converted to decimal. The first four characters are representing the temperature, next four the humidity, next CO2 and last four is the light. The conversion is run 4 times for each of the measurement. Each converted measurement is saved in a different variable.



```
public static String formatData() {
    JSONObject inJson = new JSONObject(incomingMessage);

    //Getting the payload
    try {
        incomingMessage = inJson.getString( key: "data");
    } catch (JSONException e) {
        System.out.println("No data field was found.");
        return null;
    }
    String currentData;
    //Dividing the payload to separate data and storing in the correct variables
    for(int i = 0; i <= 12; i=i+4) {
        currentData = incomingMessage.substring(i, i+4);
        switch (i) {
            case 0 :
                temperature = Integer.parseInt(currentData,  radix: 16);
                temperature = temperature/10;
                break;
            case 4:
                humidity = Integer.parseInt(currentData,  radix: 16);
                humidity = humidity/10;
                break;
            case 8:
                co2 = Integer.parseInt(currentData,  radix: 16);
                break;
            case 12:
                light = Integer.parseInt(currentData,  radix: 16);
                break;
        }
    }
```

Figure 5 in Embedded Implementation - Converting Data

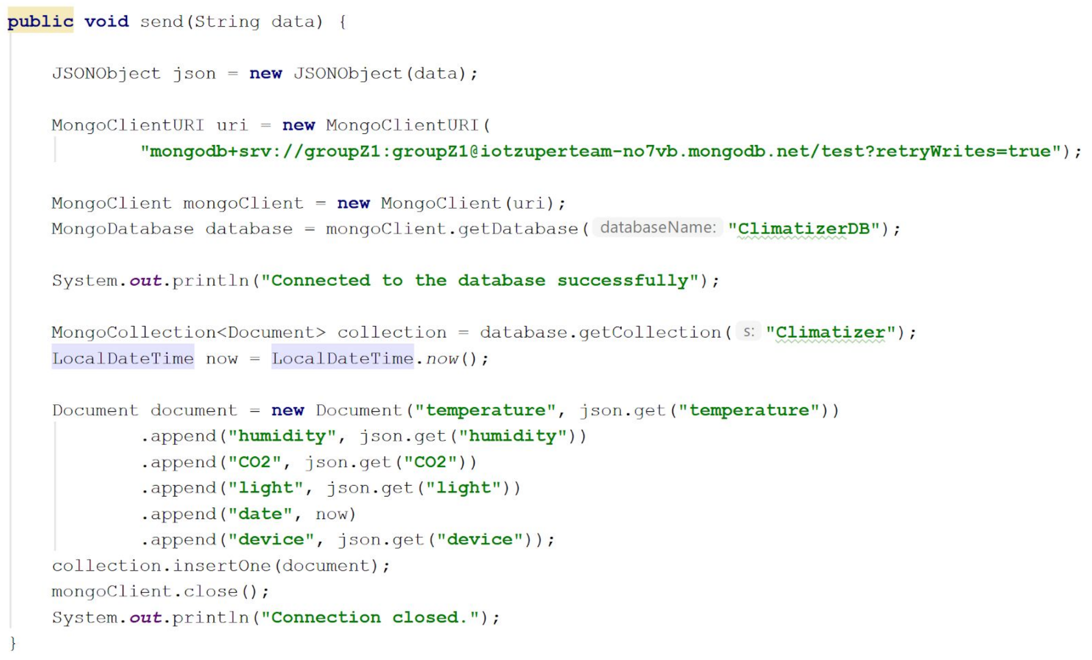
After all the data is already converted, a String for a JSONObject is created. It consists of the data that will be sent to Mongo. In the date field nothing is put, because the time would be inaccurate, so it will be put right before sending.



```
//Creating a Json that will be sent to MongoDB
String outJsonString = new JSONObject()
        .put("temperature", temperature)
        .put("humidity", humidity)
        .put("CO2", co2)
        .put("light", light)
        .put("date", "")
        .put("device", EUI).toString();

System.out.println(outJsonString);

return outJsonString;
```

Figure 6 in Embedded Implementation - Creating New Json

First a JSONObject is created out of the String that was received. Next the connection to the database is created by connecting to it through a uri and the ClimatizerDB is fetched. Next the collection that the data is saved at is obtained. A new document is created and by taking the fields from the JSONObject and inputting them to the document. Only a new instance of DateTime is taken and put in the object so the time is accurate. In the end the document is inserted in the database and the connection is closed.



```
public void send(String data) {

    JSONObject json = new JSONObject(data);

    MongoClientURI uri = new MongoClientURI(
            "mongodb+srv://groupZ1:groupZ1@iotzuperteam-no7vb.mongodb.net/test?retryWrites=true");

    MongoClient mongoClient = new MongoClient(uri);
    MongoDatabase database = mongoClient.getDatabase( databaseName: "ClimatizerDB");

    System.out.println("Connected to the database successfully");

    MongoCollection<Document> collection = database.getCollection( s: "Climatizer");
    LocalDateTime now = LocalDateTime.now();

    Document document = new Document("temperature", json.get("temperature"))
            .append("humidity", json.get("humidity"))
            .append("CO2", json.get("CO2"))
            .append("light", json.get("light"))
            .append("date", now)
            .append("device", json.get("device"));
    collection.insertOne(document);
    mongoClient.close();
    System.out.println("Connection closed.");
}
```

Figure 7 in Embedded Implementation - Connection With Mongo Database

The whole process of sending the message from receiving it in the LoRaClient class untill sending it from the MongoDBHelper is shown on *Figure 7 in Embedded Implementation*.

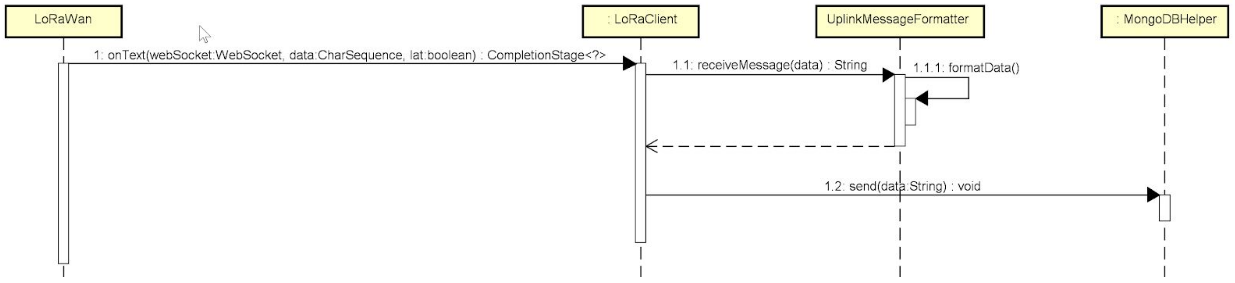
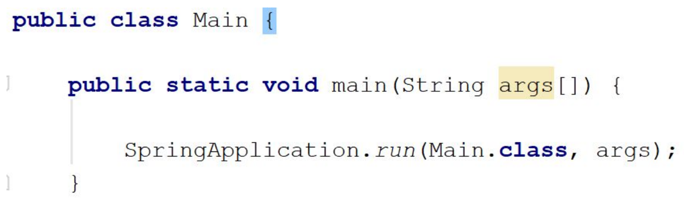

LoRaWan
: LoRaClient
UplinkMessageFormatter
: MongoDBHelper
1: onText(webSocket:WebSocket, data:CharSequence, lat:boolean) : CompletionStage<?>
1.1: receiveMessage(data) : String
1.1.1: formatData()
1.2: send(data:String) : void

Figure 7 in Embedded Implementation - Send Message To Mongo Sequence Diagram

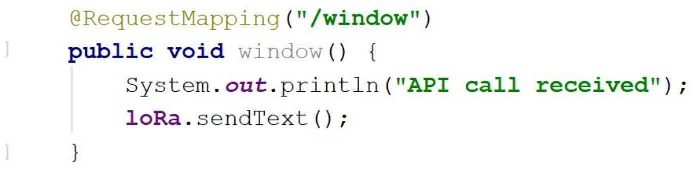
For the Open/Close Window Remotely Use Case the connection from the WebAPI to LoRaWan is needed. This is implemented by using the spring framework. The Main class runs the Spring Boot application and then a controller is listening for calls.



```
public class Main {

    public static void main(String args[]) {

        SpringApplication.run(Main.class, args);
    }
```

Figure 8 in Embedded Implementation - Main Method

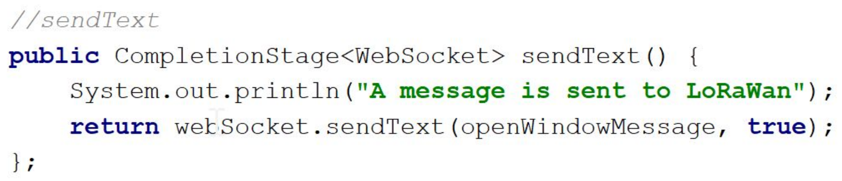
The window() method can be called using "/window" mapping. When the method is call it initiates the method sendText in LoRaClient class.



```
@RequestMapping("/window")
public void window() {
    System.out.println("API call received");
    loRa.sendText();
}
```

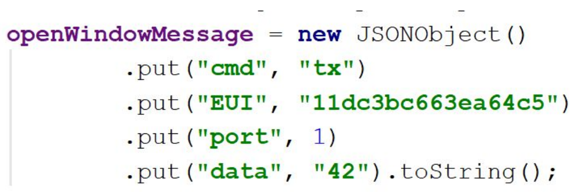
Figure 9 in Embedded Implementation - Window Method

In the sendText method a simple call on the webSocket is made with the openWindowMessage and true as arguments. The second argument means that this is the last message of the sequence and it can be sent. The openWindowMessage is initialized in the constructor of LoRaClient class - *The figure below*.



```
//sendText
public CompletionStage<WebSocket> sendText() {
    System.out.println("A message is sent to LoRaWan");
    return webSocket.sendText(openWindowMessage, true);
};
```

Figure 10 in Embedded Implementation - Window Method



```
openWindowMessage = new JSONObject()
        .put("cmd", "tx")
        .put("EUI", "11dc3bc663ea64c5")
        .put("port", 1)
        .put("data", "42").toString();
```

Figure 11 in Embedded Implementation - Open Window Message

The whole process and invoked methods that are needed to complete the Open/Close Window Remotely Use Case are shown on the diagram below.

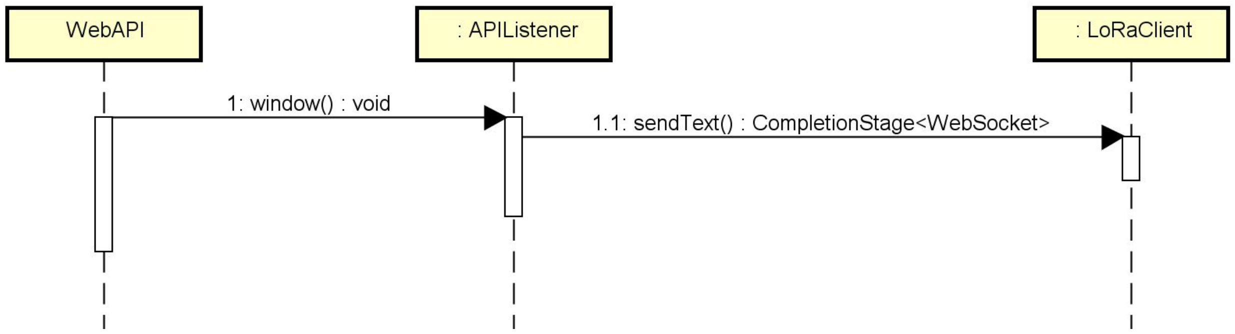
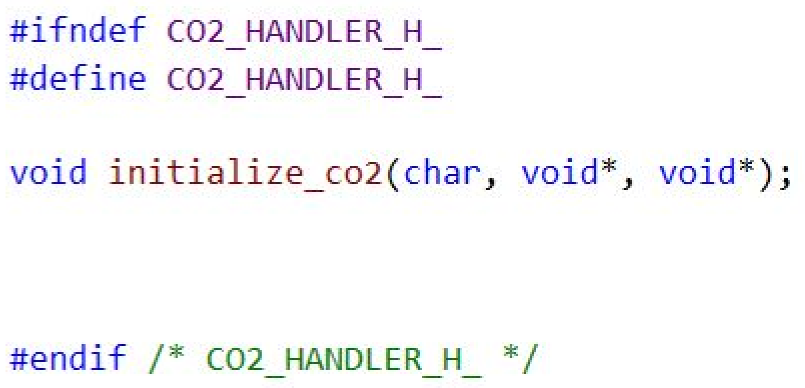


Figure 12 in Embedded Implementation - Open Window Sequence Diagram

Implementation details of the Arduino microcontroller

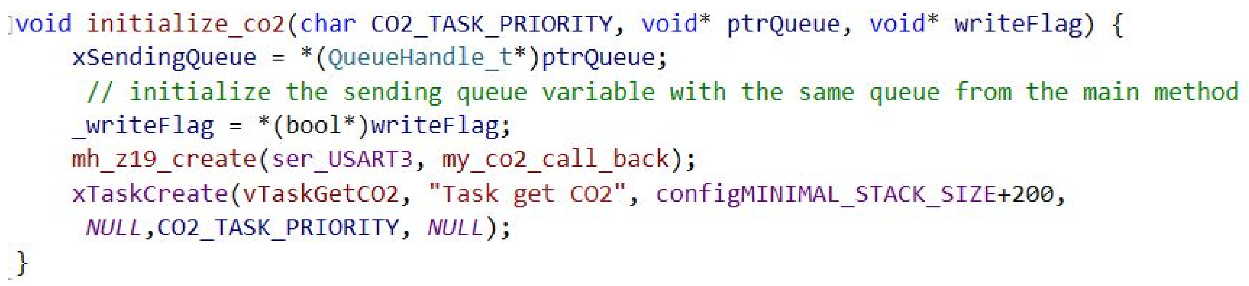
As one of the Use Cases of the project states, the readings of the device should be send with a fixed period of time. In order to achieve that the task should gather the data first, send it to the LoRa Server wait for a fixed period, and then do that action again. Using the Queue built into FreeRTOS it is possible to gather the data from the sensors. The problem is that after the Queue is emptied by the sending task, the sensors' tasks see that the queue is empty and therefore they can write things into it without waiting until the fixed period passes. One way to solve it would be to use the "writeFlag" - a global variable, that the sensors would have to check before they will try to write things into the queue.
This way the Queue could be emptied and only after time elapses the new data will land in the queue, ready to be send again.
The following code snippets show how this was implemented in code giving an example of one of the Sensor Tasks - the CO2 Sensor and LoRaWAN Handler Task.



```c
#ifndef CO2_HANDLER_H_
#define CO2_HANDLER_H_

void initialize_co2(char, void*, void*);



#endif /* CO2_HANDLER_H_ */
```

Figure 13 in Embedded Implementation - co2_handler.h file code snippet

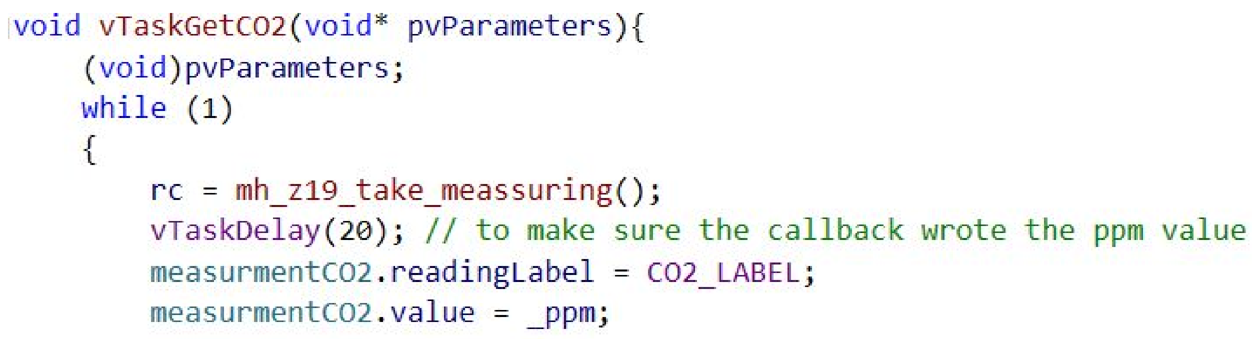
As can be seen above, all of the sensors use header files, that contain one initialization method, that can be then used in the main method.



```c
void initialize_co2(char CO2_TASK_PRIORITY, void* ptrQueue, void* writeFlag) {
    xSendingQueue = *(QueueHandle_t*)ptrQueue;
     // initialize the sending queue variable with the same queue from the main method
    _writeFlag = *(bool*)writeFlag;
    mh_z19_create(ser_USART3, my_co2_call_back);
    xTaskCreate(vTaskGetCO2, "Task get CO2", configMINIMAL_STACK_SIZE+200,
     NULL,CO2_TASK_PRIORITY, NULL);
}
```

Figure 14 in Embedded Implementation - CO2Handler.c file code snippet

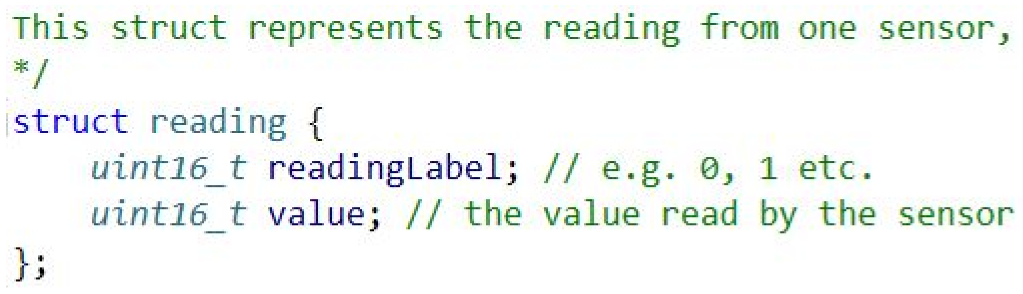
The parameters passed into the method are then saved as global for this file and used in the xTaskCreate method. You can also see the mh_z19_create method that initializes the driver for the CO2 sensor that takes the callback methods as one of the parameters.



```c
void vTaskGetCO2(void* pvParameters){
    (void)pvParameters;
    while (1)
    {
        rc = mh_z19_take_meassuring();
        vTaskDelay(20); // to make sure the callback wrote the ppm value
        measurmentCO2.readingLabel = CO2_LABEL;
        measurmentCO2.value = _ppm;
```

Figure 15 in Embedded Implementation - CO2Handler.c file code snippet

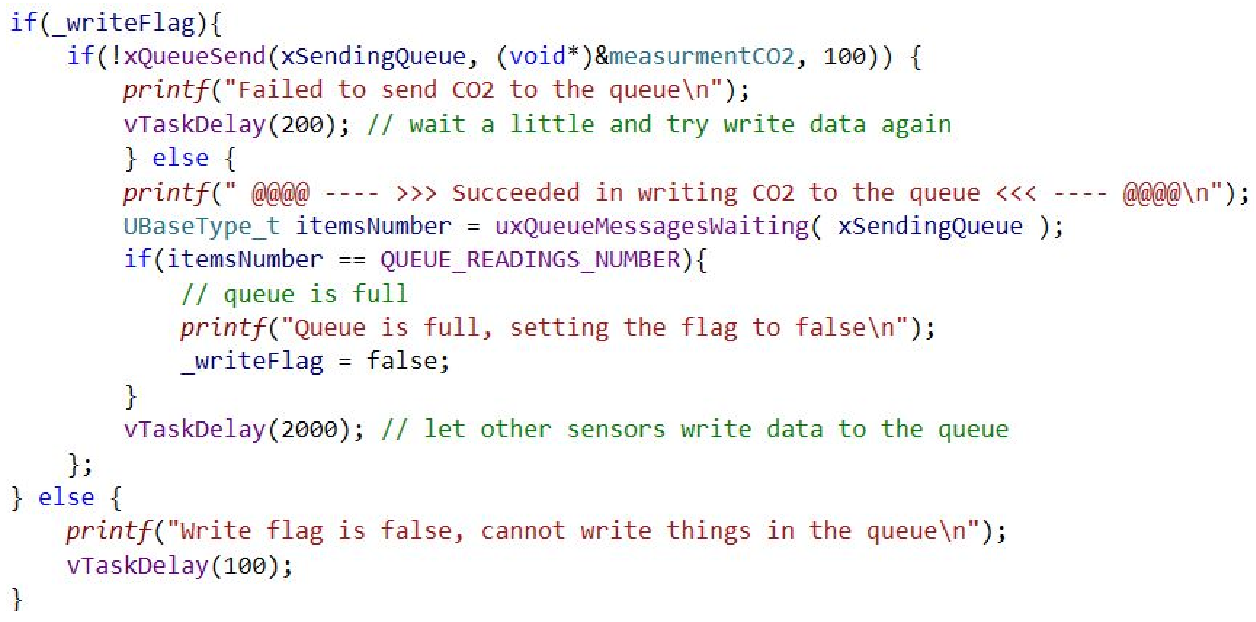
On the above code snippet the vTaskGetCO2 method is shown. It starts the infinite while loop, starting the thread, calls the measurement method, and then saves the data retrieved from the callback into the reading structure shown below.



```
This struct represents the reading from one sensor,
*/
struct reading {
    uint16_t readingLabel; // e.g. 0, 1 etc.
    uint16_t value; // the value read by the sensor
};
```

Figure 16 in Embedded Implementation - reading.h file code snippet

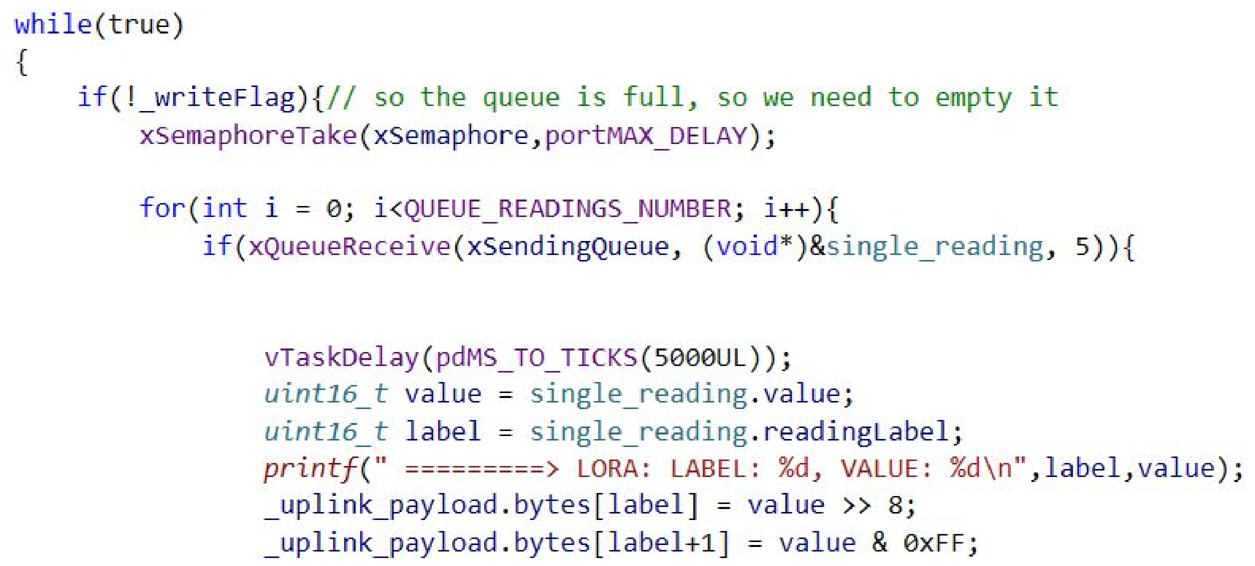
After the value is saved, the reading has to be sent into the SendingQueue.



```
if(_writeFlag){
    if(!xQueueSend(xSendingQueue, (void*)&measurmentCO2, 100)) {
        printf("Failed to send CO2 to the queue\n");
        vTaskDelay(200); // wait a little and try write data again
        } else {
        printf(" @@@@ ---- >>> Succeeded in writing CO2 to the queue <<< ---- @@@@\n");
        UBaseType_t itemsNumber = uxQueueMessagesWaiting( xSendingQueue );
        if(itemsNumber == QUEUE_READINGS_NUMBER){
            // queue is full
            printf("Queue is full, setting the flag to false\n");
            _writeFlag = false;
        }
        vTaskDelay(2000); // let other sensors write data to the queue
    };
} else {
    printf("Write flag is false, cannot write things in the queue\n");
    vTaskDelay(100);
}
```

Figure 17 in Embedded Implementation - CO2Handler.c file code snippet

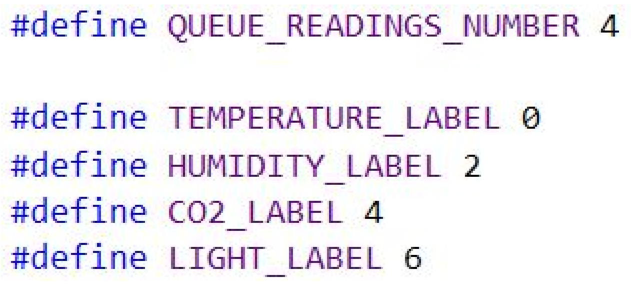
On the above snippet it can be seen that before the attempt to write something into the queue happens, the task first checks if the _writeFlag is set to true (so that it is possible to write something into because the fixed period elapsed as was discussed earlier). The pointer of the reading is then sent to the queue, after which if the queue became full the writeFlag is set to false to allow the LoRaHandler task send the data into the LoRa Server.



```
while(true)
{
    if(!_writeFlag){// so the queue is full, so we need to empty it
        xSemaphoreTake(xSemaphore,portMAX_DELAY);

        for(int i = 0; i<QUEUE_READINGS_NUMBER; i++){
            if(xQueueReceive(xSendingQueue, (void*)&single_reading, 5)){


                vTaskDelay(pdMS_TO_TICKS(5000UL));
                uint16_t value = single_reading.value;
                uint16_t label = single_reading.readingLabel;
                printf(" =========> LORA: LABEL: %d, VALUE: %d\n",label,value);
                _uplink_payload.bytes[label] = value >> 8;
                _uplink_payload.bytes[label+1] = value & 0xFF;
```

Figure 18 in Embedded Implementation - LoraHandler.c file code snippet

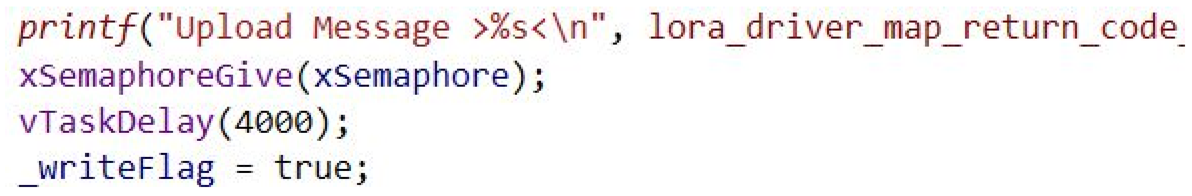
The above code snippets represents a part of the LoraHandler task. First it checks if the queue is full, then it takes the semaphore to prevent any reading in the time of emptying the queue. After that it start the loop that will write data in a fixed order to the payload that is said to be sent to the LoRa Server. The order can be seen in the following code snippet.



```
#define QUEUE_READINGS_NUMBER 4

#define TEMPERATURE_LABEL 0
#define HUMIDITY_LABEL 2
#define CO2_LABEL 4
#define LIGHT_LABEL 6
```

Figure 19 in Embedded Implementation - ios_io.h file code snippet

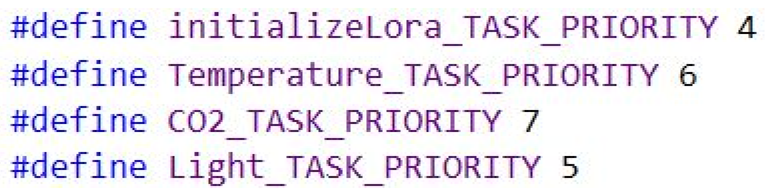
The labels define the bytes of the final payload. The data always contain two bytes, this is why they are even numbers starting from 0. THe QUEUE_READINGS_NUMBER defines the number of the sensor data sent to the queue.
After the above is called and the payload saved, the data is sent to the LoRa Server and the task is suspended for a fixed period, after which the writeFlag is set back to true to enable new readings to be written into the queue. This can be seen on the following code snippet.



```
printf("Upload Message >%s<\n", lora_driver_map_return_code_
xSemaphoreGive(xSemaphore);
vTaskDelay(4000);
_writeFlag = true;
```

Figure 20 in Embedded Implementation - LoraHandler.c file code snippet

What is worth mentioning is also the priority stack used in the system, which looks like on the following snippet.



```
#define initializeLora_TASK_PRIORITY 4
#define Temperature_TASK_PRIORITY 6
#define CO2_TASK_PRIORITY 7
#define Light_TASK_PRIORITY 5
```

Figure 21 in Embedded Implementation - main. file code snippet

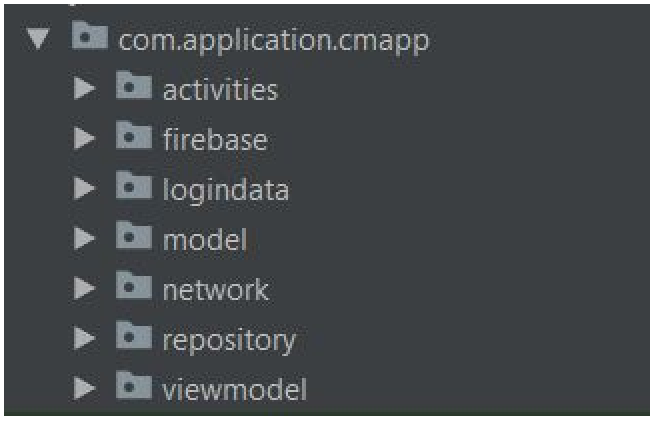
This ensures the sensors' drivers will initialize before the lora would send the data to the server.

Android Implementation

(Authors: Amahdya Delkescamp, Andrei Cioanca, Nikita Roskovs)

System Architecture

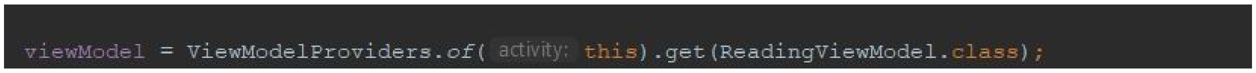
As mentioned above, the system is built using MVVM architecture. In that sense, the system is structured in the following packages:



```
▼ com.application.cmapp
    ▶ activities
    ▶ firebase
    ▶ logindata
    ▶ model
    ▶ network
    ▶ repository
    ▶ viewmodel
```

Figure 1 in Android Implementation - Android App Packages

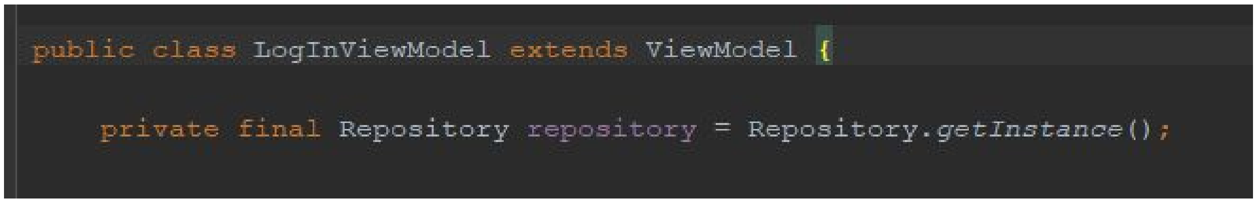
The app starts on the MainActivity and goes through the creation process. While the Firebase part of the project and the networking part of the project are deemed separate, both viewmodels are used in the Main Activity sometimes due to it having to observe changes on the login process as well as the readings. Once MainActivity is started it gets its viewModel assigned and the “initialization chain” of the architecture moves further down.



```
viewModel = ViewModelProviders.of( activity: this).get(ReadingViewModel.class);
```

Figure 2 in Android Implementation - MainActivity Code Snippet 1

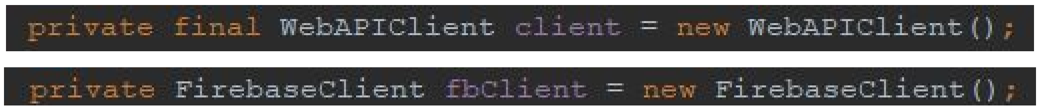
Once the viewmodel has been instantiated we also get an instance of a singleton repository that takes care of all data coming in and out of both firebase and the API. This is done from inside the viewmodel in order to respect the architecture.



```
public class LogInViewModel extends ViewModel {

    private final Repository repository = Repository.getInstance();
```

Figure 3 in Android Implementation - ViewModel Code Snippet 1

The same thing is done for the ReadingViewModel, both of them using the same repository. After this, the repository holds one instance of FirebaseClient, which handles the firebase connections and one instance of WebAPIClient, which handles connections and requests to the api.

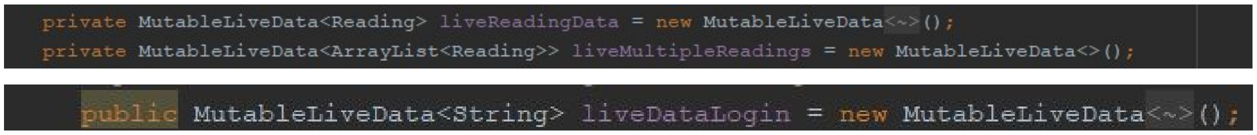


```
private final WebAPIClient client = new WebAPIClient();
private FirebaseClient fbClient = new FirebaseClient();
```

Figure 4 in Android Implementation - Repository Code Snippet 1

The app isn’t using a realtime database such as Firebase to hold its data so instead of normal LiveData it uses MutableLiveData. The difference between those being the postValue() and setValue() which are made public on MutableLiveData thus enabling the background thread to notify the UI thread with postValue().

There are three MutableLiveData objects in the app, one for the most recent reading in the database, one for the list of readings by date and one for the login information through firebase. All three of them are instantiated in the repository and then retrieved by the activities through getters.



```
private MutableLiveData<Reading> liveReadingData = new MutableLiveData<~>();
private MutableLiveData<ArrayList<Reading>> liveMultipleReadings = new MutableLiveData<>();
```

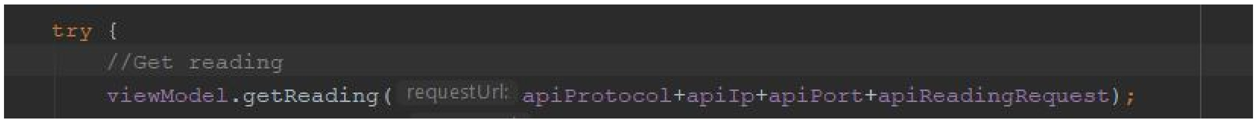
```
public MutableLiveData<String> liveDataLogin = new MutableLiveData<~>();
```

Figure 5 in Android Implementation - Repository Code Snippet 2

The app uses the aforementioned live data to pass information back up the “chain” without disrupting the MvvM architecture thus ensuring the low-coupling of the different parts of the app. So if at some point the API server changes then only the client in the app will change, leaving the other code intact.

Handling a GET request

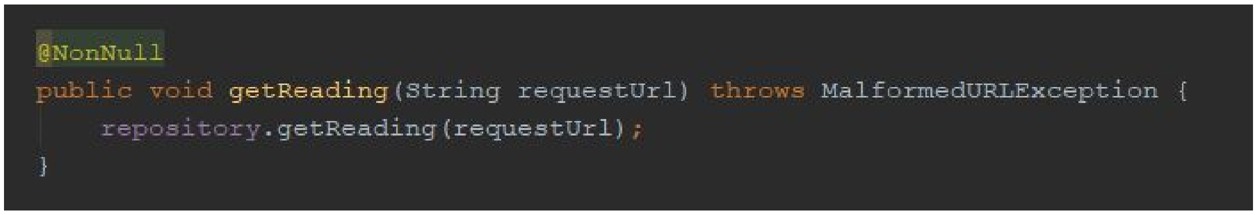
The live data ultimately serves the goal of handling API requests and Firebase notifications. With this in mind here is how the system retrieves the latest reading from the database.



```
try {
    //Get reading
    viewModel.getReading( requestUrl: apiProtocol+apiIp+apiPort+apiReadingRequest);
```

Figure 6 in Android Implementation - Main Activity Code Snippet 2

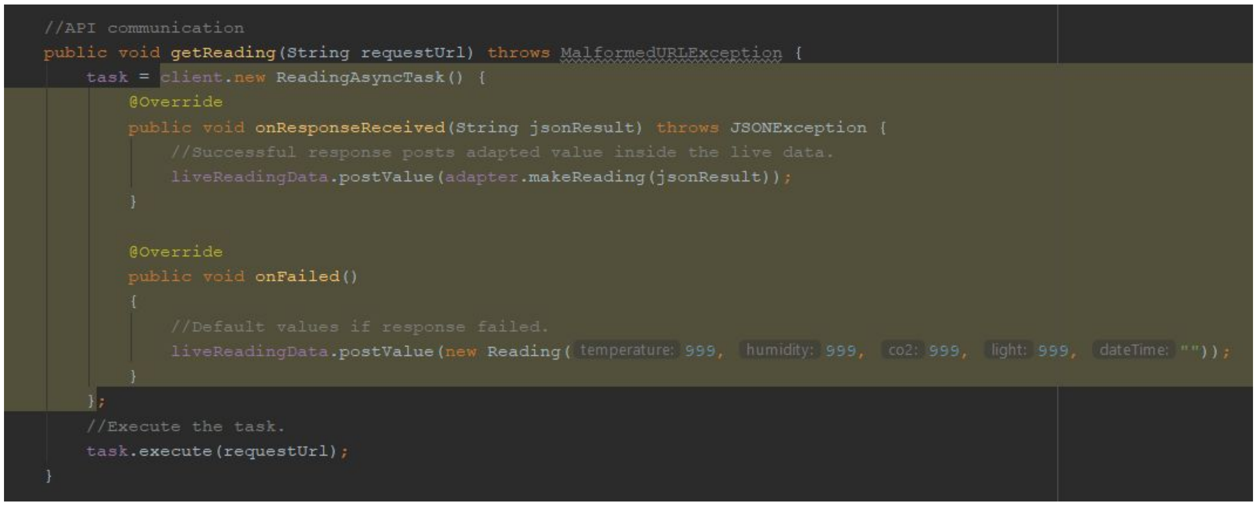
It all starts in the MainActivity where a void method is called on the viewmodel to retrieve the latest reading and the connection string is passed. Inside the viewmodel, the called method passess the responsibility to the repository where the getReading method is called.



```
@NonNull
public void getReading(String requestUrl) throws MalformedURLException {
    repository.getReading(requestUrl);
}
```

Figure 7 in Android Implementation - ViewModel Code Snippet 2

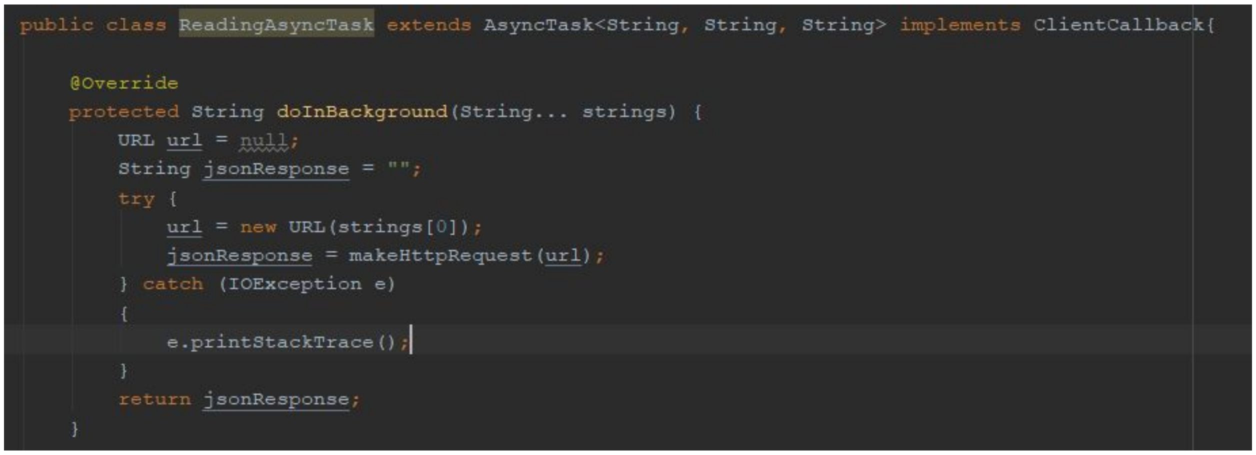
Inside the repository, the called method creates a new AsyncTask for the client that the system automatically instantiates on the creation of the repository. Here the system employs a clever use of Interface Callbacks that enables the data to “travel” one level up the chain without breaking MvvM.



```
//API communication
public void getReading(String requestUrl) throws MalformedURLException {
    task = client.new ReadingAsyncTask() {
        @Override
        public void onResponseReceived(String jsonResult) throws JSONException {
            //Successful response posts adapted value inside the live data.
            liveReadingData.postValue(adapter.makeReading(jsonResult));
        }

        @Override
        public void onFailed()
        {
            //Default values if response failed.
            liveReadingData.postValue(new Reading( temperature: 999,  humidity: 999,  co2: 999,  light: 999,  dateTime: ""));
        }
    };
    //Execute the task.
    task.execute(requestUrl);
}
```

Figure 8 in Android Implementation - Repository Code Snippet 3

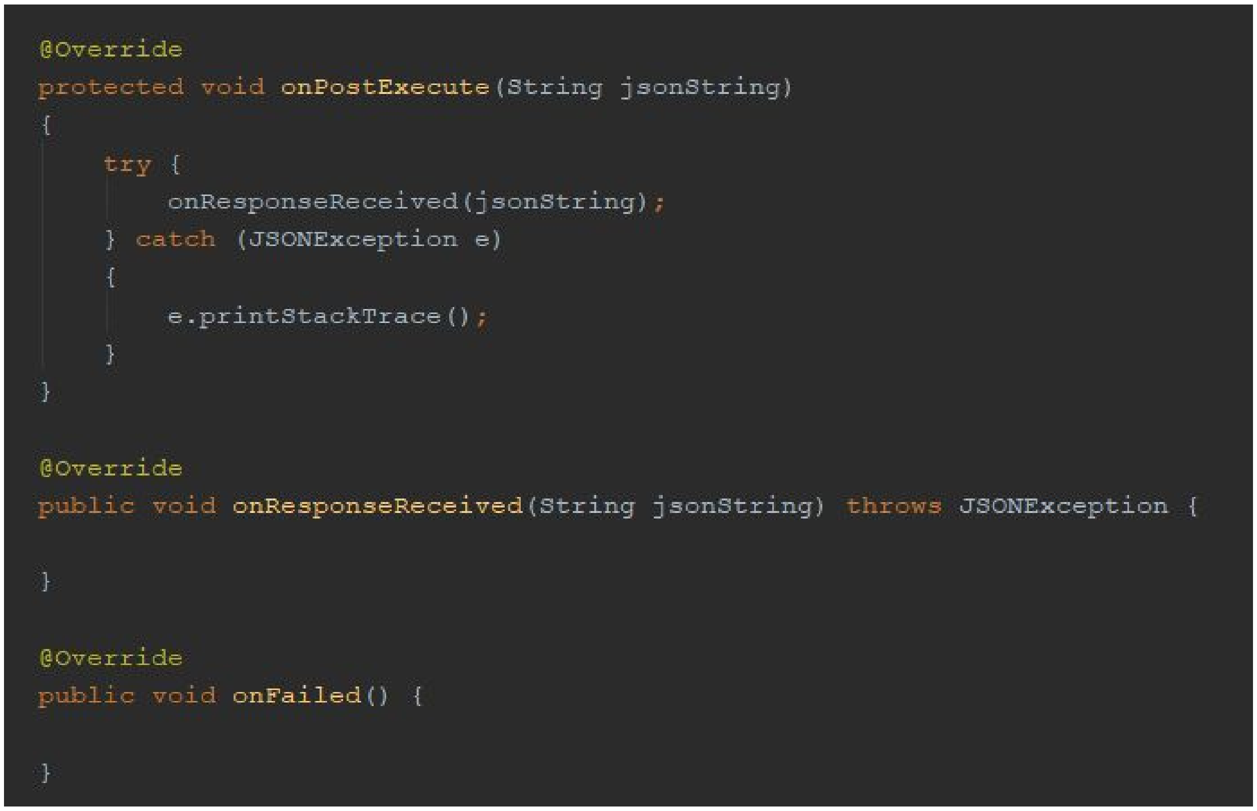
The AsyncTask class is an inner class of WebAPIClient that implements the ClientCallback interface, adding two more overriden methods to the existing methods of an AsyncTask.



```
public class ReadingAsyncTask extends AsyncTask<String, String, String> implements ClientCallback{

    @Override
    protected String doInBackground(String... strings) {
        URL url = null;
        String jsonResponse = "";
        try {
            url = new URL(strings[0]);
            jsonResponse = makeHttpRequest(url);
        } catch (IOException e)
        {
            e.printStackTrace();
        }
        return jsonResponse;
    }
```

Figure 9 in Android Implementation - WebAPIClient Code Snippet 3

Here the WebAPIClient connects to the API and retrieves the response from the GET request based on the URL that was passed down from Repository. After the doInBackground() method is finished the AsyncTask calls onPostExecute which in turn calls the implemented interface's (ClientCallback) own method onResponseReceived() and passes the received json string from the API.

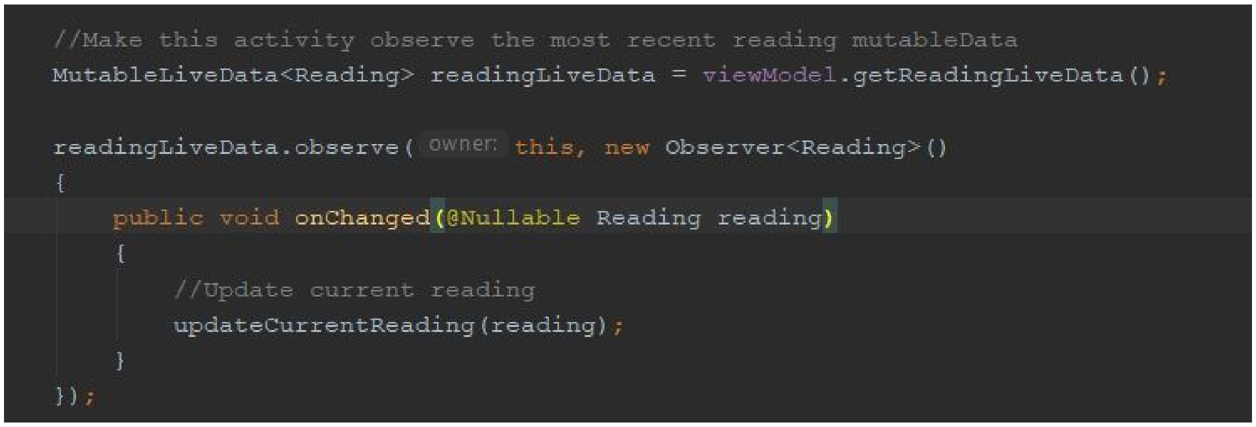


```
@Override
protected void onPostExecute(String jsonString)
{
    try {
        onResponseReceived(jsonString);
    } catch (JSONException e)
    {
        e.printStackTrace();
    }
}

@Override
public void onResponseReceived(String jsonString) throws JSONException {

}

@Override
public void onFailed() {

}
```

Figure 10 in Android Implementation - WebAPIClient Code Snippet 2

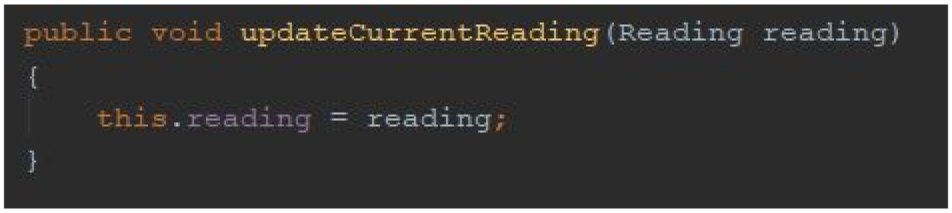
Then the system calls the same overriden method but form one "level" above the WebAPIClient which is the actual repository. Inside the repository, if the call is successful, the system passes the received value through an Adapter that translates the received data into something that the system can read and then posts the value to the MutableLiveData, thus notifying any active observers of the data's update as seen in Figure 27 above.

Back in the MainActivity, during the onCreate cycle, the system retrieves the instances of liveData from the repository through the viewModel and then places an active listener on them. When the data is inevitably changed from the repository with postValue(), then the onChanged() method in the MainActivity gets called where the new data gets saved and then later processed and displayed accordingly.



```
//Make this activity observe the most recent reading mutableData
MutableLiveData<Reading> readingLiveData = viewModel.getReadingLiveData();

readingLiveData.observe( owner: this, new Observer<Reading>()
{
    public void onChanged(@Nullable Reading reading)
    {
        //Update current reading
        updateCurrentReading(reading);
    }
});
```

Figure 11 in Android Implementation - MainActivity Code Snippet 3



```
public void updateCurrentReading(Reading reading)
{
    this.reading = reading;
}
```

Figure 12 in Android Implementation - MainActivity Code Snippet 4

The above mentioned process ensures each piece of the architecture is loosely coupled with one another and can easily be expanded, changed or scaled to accommodate extra functionality while still ensuring the architecture remains intact.

The other live data objects mentioned serve the same purpose within the app. They're used in the Firebase authentication process and in the other API requests and function in a similar way as the GET request explained above.

BI Reports

(Authors: Claudiu Rediu, Stefan Harabagiu)

In considering what is relevant to the environment in which the system would work, the following representations were made:

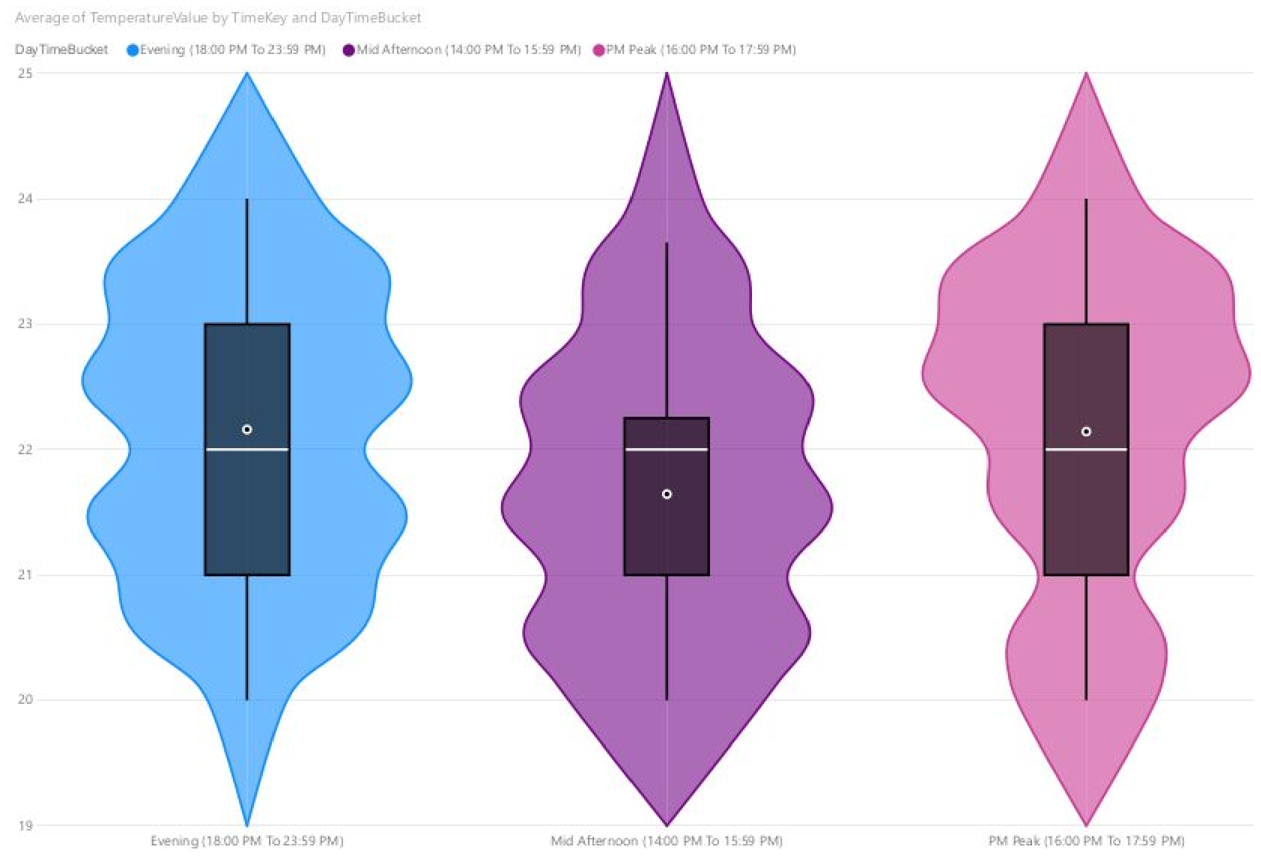


Figure 1 in BI Reports - Violin Plot of Average Temperature in the Afternoon

The afternoon is usually when the students would be present in the library, after the courses. The temperatures recorded showcase how the median of multiple readings is at the same value. The average stays close to the median, so there isn't too much variation that could harm the people in the library or the books. If any of this would change, actions would be recommended to adjust the environment.

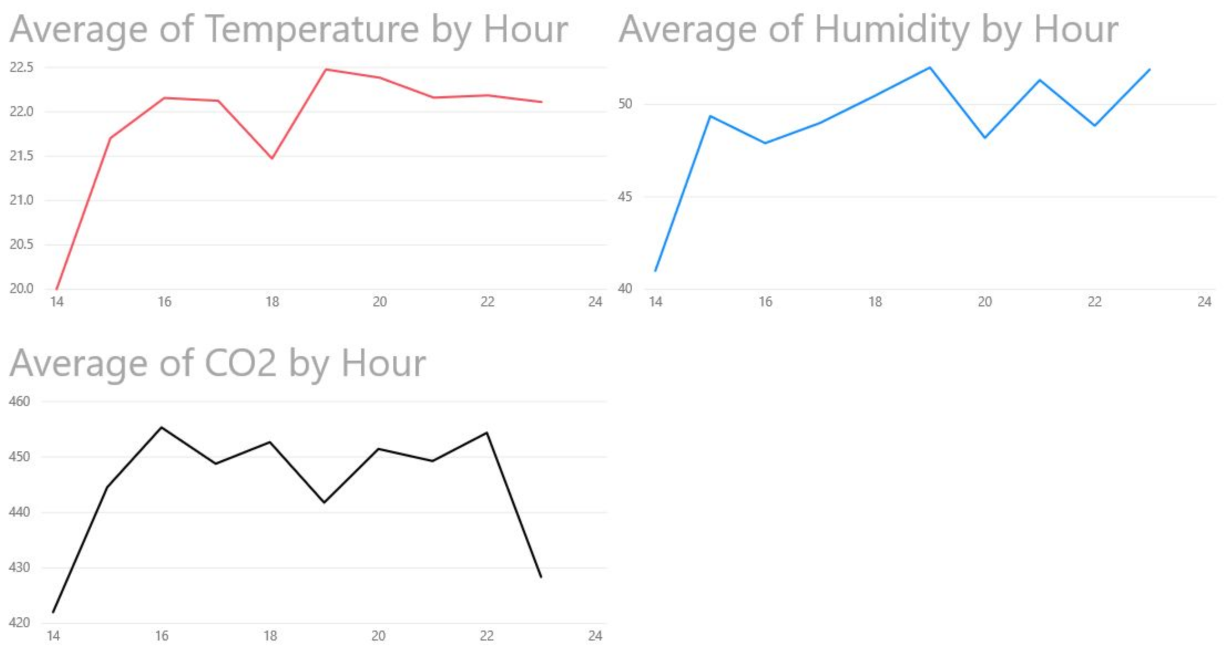


Figure 2 in BI Reports - Line Plot of Average Temperature, Humidity and CO2 by Hour

Signs of multiple students in the library can be seen in the CO2 graph, as multiple people breathing in the same rooms would help in rising the CO2 level. When getting close to midnight, there is a sudden dip that goes hand in hand with the students leaving for home. This information can aid in managing the air flow better during peak hours in the library.

Testing

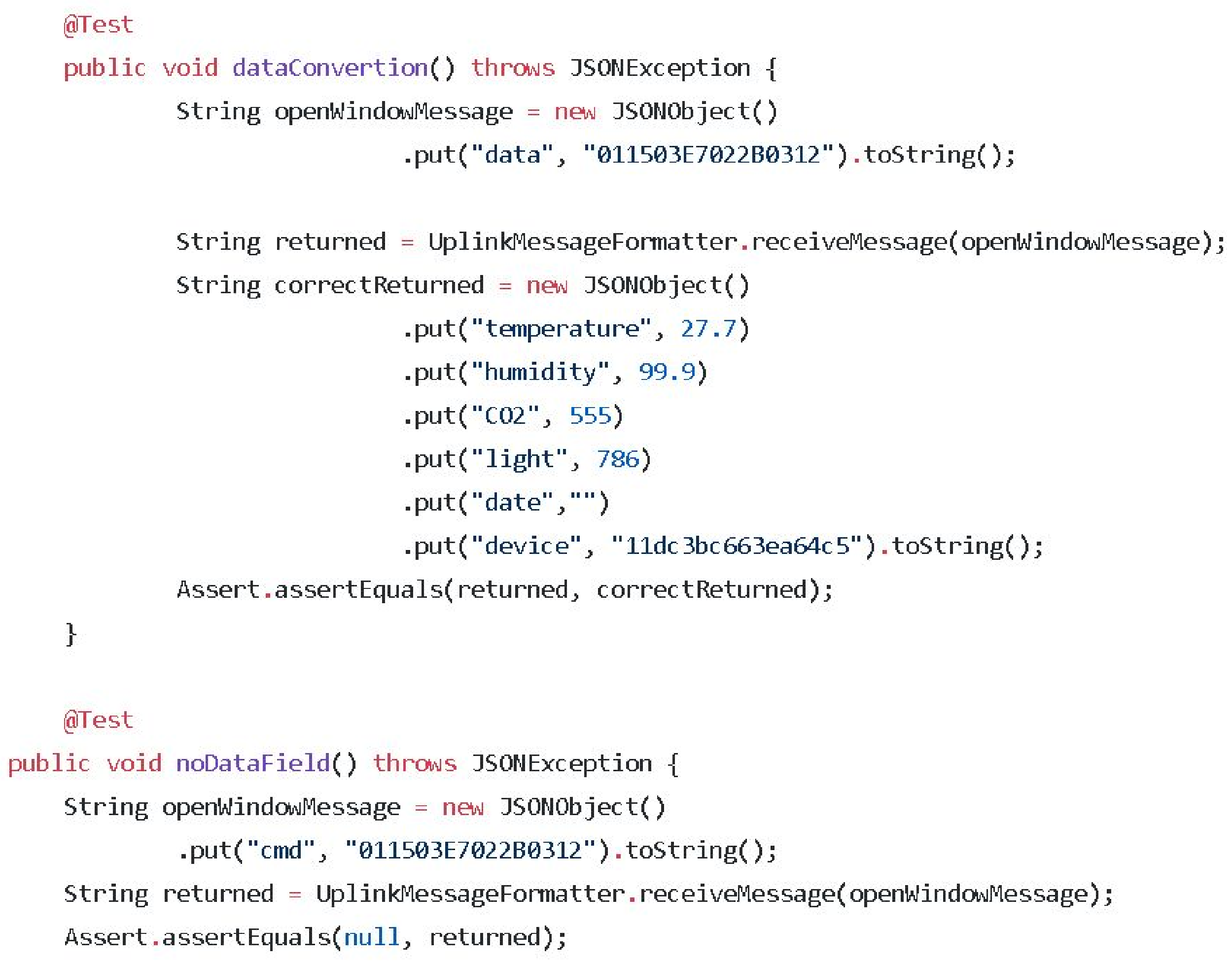
(author: Mihail Kanchev)

Due to the scale of the project and the size of its components, testing was not emphasized to an extent the team would have appreciated. Testing was either done by console messages during development or as black box. The group concluded the solution by completely covering two out of three use cases. The third and ultimate use case was partially covered but a solution was not found to it due to complications in the communication between LoRaWAN and the arduino.

Test Specifications

Test Cases were used to ensure that the Use Cases and the requirements have been respected. The following table displays all the test specifications:

Nr	Description	Actor	Precondition	Expected Result	Steps	Result
1	Read Climate Conditions	admin/user	The application is opened The most recent reading is loaded up at the start	The admin/user has the data displayed	1. The admin/user presses the refresh button "Refresh" 2. The system loads up the most recent reading	The admin/user has the data displayed
2	Open/Close Window Remotely	admin	The application is opened The actuator is connected The admin is authenticated	The actuator starts opening/closing the window	1. The admin decides to open the window 2. The admin presses the button "use window" 3. The actuator starts.	The signal is sent but is not received by the arduino due to lack of a connection between the device and LoRaWAN
3	Present Readings Per Day	admin/user	The application is opened The most recent list of readings are loaded up	The admin/user has the data displayed	1. The admin/user selects a day 2. The admin/user presses "Refresh" 3. The system loads up the most recent reading in the selected day	The admin/user has the data displayed



```
    @Test
    public void dataConvertion() throws JSONException {
            String openWindowMessage = new JSONObject()
                            .put("data", "011503E7022B0312").toString();

            String returned = UplinkMessageFormatter.receiveMessage(openWindowMessage);
            String correctReturned = new JSONObject()
                            .put("temperature", 27.7)
                            .put("humidity", 99.9)
                            .put("CO2", 555)
                            .put("light", 786)
                            .put("date","")
                            .put("device", "11dc3bc663ea64c5").toString();
            Assert.assertEquals(returned, correctReturned);
    }

    @Test
public void noDataField() throws JSONException {
    String openWindowMessage = new JSONObject()
            .put("cmd", "011503E7022B0312").toString();
    String returned = UplinkMessageFormatter.receiveMessage(openWindowMessage);
    Assert.assertEquals(null, returned);
```

Figure 1 in Test - Unit Test

These two unit tests are trying to assert if the correct type of data is received from the LoRaWAN. The values from all sensors are received in the form of 4 digit hexadecimal numbers. Knowing the received values of "011503E7022B0312" we can deduce that splitting them by four (ex. 0115 03E7 022B 0312) we get the following - (27.7 99.9 555 786). Having that in mind, a new JSON object is created with the same values, device and date. After that both the received object and the newly created one are asserted.

In the second test a new JSON object is created, without a date value and is asserted to the received one.

Results and Discussion

The product owner received a product made considering the business requirements. The only thing that the system fails to do is to open the window using the phone application, as there were issues sending through LoRaWAN to the embedded implementation. The other requirements have been satisfied by building an IoT system using JAVA, SQL, C, Arduino atMega2560, MongoDB, Microsoft SQL Server and an Android Application that employs the MVVM Architecture.

Conclusion

The project followed SCRUM & AUP during the development process. The start was focused on preparing and understanding why the system should be created. As it progressed, functional, non-functional requirements and use cases have been defined considering the needs of the client.
A Domain Model was the result of analysing the objects and their relationships in the problem domain. The Design Class diagrams were used as blueprints for the implementation of the IoT System. In the last phases of the project, the application was tested based on the Use Cases. Once finalized, the results were subject to discussion and analysis to bring further improvements in the future.

Project Future

The system satisfies all the requirements, except one. Other improvements can also be brought.
Firstly, being able to open the window remotely should be the main feature that should be implemented in the future. This is the main issue with the current status of the system.
Secondly, improving the Android Application to be more explicit and make better use of the data. As it is now, it is enough to satisfy the customer, but it could use a more detailed view with graphs and reports.
Thirdly, setting up a system to easily add new customers to the system, so they could use admin accounts to open the window and more arduino boards.

List of Appendices

Sources of Information

Webography

Allergy, A., 2019. Achoo Allergy. [Online] Available at:
https://www.achooallergy.com/learning/the-effects-of-humidity-on-the-human-body/
[Accessed 18 2 2019].

ecoadmin, 2016. ECOthink Group. [Online] Available at:
http://www.ecothinkgroup.com/the-effects-of-elevated-carbon-dioxide-levels-in-schools/
[Accessed 18 2 2019].

educationcorner, 2019. educationcorner.com. [Online] Available at:
https://www.educationcorner.com/study-location.html [Accessed 18 2 2019].

Lawrence, D., 1997. Gateways to knowledge : the role of academic libraries in teaching, learning, and research. s.l.:s.n.

Merriam-Webster, 2019. Merriam-Webster. [Online] Available at:
https://www.merriam-webster.com/dictionary/library [Accessed 18 2 2019].

Ogden, S., 2019. NEDCC. [Online] Available at:
https://www.nedcc.org/free-resources/preservation-leaflets/2.-the-environment/2.1-temperature,-relative-humidity,-light,-and-air-quality-basic-guidelines-for-preservation [Accessed 18 2 2019].