Smart Garden

SEP4 Project Report

Supervisors:

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

Students:

Angel Iliyanov Petrov – 266489
Christian Schou Sørensen – 267142
Diyar Hussein Hussein – 266352
Eduard Nicolae Costea -266078
Erika Monica- Szasz- 280201
Ionel-Cristinel Putinica – 266123
Josipa Babic – 266757
Kenneth Ulrik Petersen – 269379
Mihai Tirtara - 266097
Remedios Pastor Molines – 266100

[66360 characters]

SEP4

4th Semester

15.05.2019

Table of content

Abstract

Owning plants requires a fair level of care and attention. For the purpose of simplifying plant tending, monitoring the external conditions of the plant such as the air humidity and temperature could play a key factor in providing it with the right care that includes tasks such as watering the plant and turning up the heat. The system that is the subject of this project has been designed from scratch in order to facilitate the growth and maintenance of plants. This project report has the purpose of describing all the methods, stages and iterations that went into implementation of this project, using Java, C, C# as the main programming languages.

The system has been fully designed with the end-user experience in mind, the purpose being to make the process of keeping track of the external conditions of plants as easy as possible and allow actions such as watering the plants remotely. The entire system was designed in three parts, by three different groups, with each one of them focusing on different tasks: Data Engineering, Embedded programming and the development of the Android Application.

1 Introduction

When deciding to own plants, people should first consider what kinds of plants work best with their lifestyle and environment and what is the maintenance level required to keep the plants in a good condition. Indoor plants can be a great addition to homes because they present a multitude of benefits such as improving air quality by removing carbon dioxide while providing extra oxygen and the exposure to them reduces stress levels, boosts mental health, calms anxiety and lowers blood pressure. Also, they improve productivity and concentration, which is great for people who work from home and need a boost to focus on the work at hand. Plus, they help regulate humidity and increase levels of positivity. It is equally important to mention that plants can help diversify bacteria in our bodies to fight infections and allergies which is very crucial for city dwellers who are less likely to be exposed to nature daily, therefore their immunity can be influenced negatively without the presence of indoor greenery (Tobebright, 2018).

There is a multitude of benefits and positive aspects to owning plants, but there are some downsides as well. The responsibility of keeping a plant alive might not be very easy. Monitoring the external conditions of a plant and controlling them remotely through a system of sensors and actuators that notify the user about the current status of the plant is the subject of this project.

The developed product can be defined as 3 systems, all of which communicate to each other to a certain extent. The team took the task of developing an application, with requirements related to the programming languages that have to be used, the sensors that the embedded tier must use. This set some boundaries and limitations over what the group came up with as far as design went, but still there was a good amount of opportunities for creativity, inventiveness and originality.

2 Requirements

2.1 Functional Requirements

ID	User Story Description	Priority
1	As a customer, I want to be able to access plant's data via mobile application.	High
2	As a customer, I want to be able to receive information about the plants' conditions.	High
3	As a customer, I should be able to define and/or modify the range of the conditions in which the plant(s) must be kept.	High
4	As a customer I should be able to remotely water my plants (from mobile application)	High
5	As a customer, I should be able to create profiles for plants.	Medium
6	As a customer, I should be able to choose a preset for the most common type of plant.	Low
7	As a customer I should be able to remove the profile of a plant.	Low
8	As a customer I should be able to access tips regarding growing a plant	Low
9	As a customer I should be able to remotely control the light in the room where my plants are.	Low
10	As a customer I should be able to control the heat remotely to provide the best conditions for growth of the plants.	Low
11	As a customer I should be able to see the progress of a plants' conditions over a period of time	Medium
12	As a customer I should have all my user data protected	High

Table 1: User Stories

ID	Requirement	Priority
1	• The user should be able to access his data in real time via mobile application	High
2	• The user should be able to receive information about their plant(s)	High
3	• The user should be able to define/modify the range of the conditions in which their plant(s) must be kept	High
4	• The user should be able to remotely water their plants	High
5	• The user should be able to create profiles for plants	Medium
6	• The user should be able to choose a preset for each type of plant	Low
7	• The user should be able to remove the profile of a plant	Low
8	• The user should be able to access tips regarding growing a plant	Low
9	• The user should be able to remotely control the light in the room where my plants are	Low
10	• The user should be able to control the heat remotely to provide the best conditions for growth of the plants	Low
11	• The user should be able to see the progress of a plants' conditions over a period of time	Medium
12	• The user should have all their user data protected	High

Table 2: Functional Requirements

ID	Requirement	Division
14	The application must retrieve, parse and display relevant data from a webservice	Interactive Media
15	The application must have a responsive user interface	Interactive Media
16	The application should use lists to display data	Interactive Media
17	The application should include various options for visualizing sensor data	Interactive Media
18	The application should have a settings menu	Interactive Media
19	The application should persist some data locally on the device	Interactive Media
20	The application could utilize authentication to sign in	Interactive Media
21	The application could utilize Google Maps and Location API to display sensor data	Interactive Media
22	The application could be able to send data to a webservice to interact with actuators	Interactive Media

Table 3: Functional Requirements – Android application

2.2 Non-Functional Requirements

ID	Requirement	Team
1	You must use at least five tasks.	IoT
2	Some data must be used by more than one task.	IoT
3	You must use semaphores, mutexes and queues.	IoT
4	You must unit test parts of your application.	IoT
5	You must use at least five tasks.	IoT

6	Some data must be used by more than one task.	IoT
7	You must use semaphores, mutexes and queues.	IoT
8	Apply knowledge of dimensional database modelling	Data Engineering
9	Design and implement a dimensional model	Data Engineering
10	Design and implement an Extract, Clean up, Transform, Load process for the data flow	Data Engineering
11	Design and implement web services	Data Engineering
12	Create paginated reports in Reporting services	Data Engineering
13	Create analyses in Power BI	Data Engineering
14	The application must be under version control for the entire development process	Interactive Media
15	The application must be developed using the official Android framework	Interactive Media
16	The application must be developed with Java	Interactive Media
17	The application should follow the Google Material Design guidelines	Interactive Media
18	The source code should be structured using an architectural pattern (MVVM is advised)	Interactive Media

Table 4: Non-functional Requirements

3 Analysis

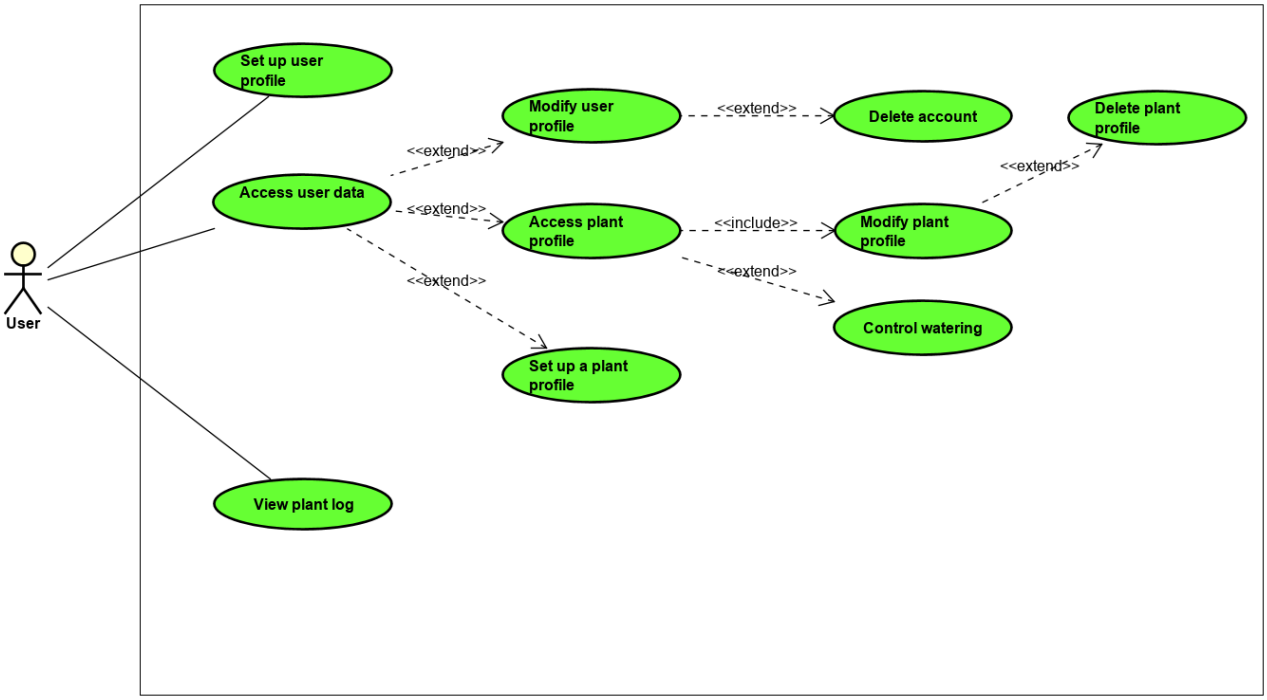


Figure 1: Use Case Diagram

3.1 Use Case Diagram

The above Use Case diagram represents the various operations that the user can perform in the system. The Time actor represents the displaying of a watering notification on the Android part. This is handled locally in the Android implementation whilst the rest of the system is interconnected.

3.2 Use Case Descriptions

The following Use Case descriptions explain how the functionality of the system was envisioned to function described thoroughly.

Table 5: Use Case - setting up a user profile

Use Case	Set up user profile
Actor	User
Pre-Condition	The user needs to be on the startup page.
Post-Condition	A user profile will be added to the system.
Base sequence	1. User inputs relevant information. 2. User presses the "Submit" button. 3. System verifies information. 4. System returns an OK status to the client.
Exception sequence	2A: User gets timeout error if the server does not respond. 3A. The system will not create a new user profile if the email of the user is already existent in the system.

Table 6: Use Case - Access user data

Use Case	Access user data
Actor	User
Pre-Condition	The user needs to be on the startup page
Post-Condition	The system will display the relevant main screen.
Base sequence	1. User inputs their username and password. 2. User presses the "Submit" button. 3. System verifies the inputted information. 4. System validates the inputted information. 5. System displays the dashboard page for the user.
Exception sequence	2A. The user enters either wrong cpr or password and will be presented with an appropriate error screen. 2B. The request to the server is not successful, therefore, the application will display a timeout message box.

Table 7: Use Case – View plant log

Use Case	View Plant Log
Actor	User
Pre-Condition	User must be logged in the system.
Post-Condition	User is able to see the plant log.
Base sequence	1. User clicks on Check Plant's log button. 2. System retrieves data from database. 3. System displays relevant data.
Exception sequence	2.1. The system cannot retrieve data from database. 2.1.1 System displays "Request failed" 2.1.2 Use Case restarts

Table 8: Use Case – Modify user profile

Use Case	Modify user profile
Actor	User
Pre-Condition	User should be logged into their profile.
Post-Condition	User profile will be updated in the system.
Base sequence	1. User accesses the "User Profile" section 2. User clicks "Modify Profile" button. 2.1 User modifies the fields they desire to modify. 3.1. User presses "Submit" button. 3.2 System saves any edits made by the user.
Exception sequence	2. The system will reject to save empty fields.

Table 9: Use Case – Access plant profile

Use Case	Modify user profile
Actor	User
Pre-Condition	A plant profile needs to be already existing in the system.
Post-Condition	User has accessed the data successfully and can now modify it and set the profile so it works on his/her plant.
Base sequence	1. User presses "Access plant profiles". 2. System returns a list of plant profiles. 3. User selects a certain profile that he/she wants to set for his/her plant.
Exception sequence	2. There are no profiles stored in the system.

Table 10: Use Case – Set up a plant profile

Use Case	Set up a plant profile
Actor	User
Pre-Condition	A plant profile needs to be already existing in the system.
Post-Condition	User has accessed the data successfully and can now modify it and set the profile so it works on his/her plant.
Base sequence	1. User presses "Access plant profiles". 2. System returns a list of plant profiles. 3. User selects a certain profile that he/she wants to set for his/her plant.
Exception sequence	2. There are no profiles stored in the system.

Table 11: Use Case – Access Plant Profile

Use Case	Modify user profile
Actor	User
Pre-Condition	A plant profile needs to be already existing in the system.
Post-Condition	User has accessed the data successfully and can now modify it and set the profile so it works on his/her plant.
Base sequence	1. User presses "Access plant profiles". 2. System returns a list of plant profiles. 3. User selects a certain profile that he/she wants to set for his/her plant.
Exception sequence	2. There are no profiles stored in the system.

Table 12: Use Case – Set up a plant profile

Use Case	Set up a plant profile
Actor	User
Pre-Condition	User is at relevant section in system.
Post-Condition	The plant profile will be added to the system.
Base sequence	1. User accesses the plant profile section. 2. User presses the "Create new plant profile" button. 3. User inputs relevant information. 4. User presses "Submit" button. 5. System saves information about profile.
Exception sequence	4A. The system will reject a profile without proper user input. See base sequence 3. 4B. The system will reject a profile name that already exists.

Table 13: Use Case – Delete account

Use Case	Set up a plant profile
Actor	User
Pre-Condition	User must be in the "Modify profile" section.
Post-Condition	User is returned to the log in screen.
Base sequence	1. User presses the "Delete my profile" button. 2. User presses the "Confirm" button. 2.1 System removes any information related to that user. 3. System returns user to login screen.
Exception sequence	2.1 The user may cancel the action of deleting their profile.

Table 14: Use Case – Modify plant profile

Use Case	Set up a plant profile
Actor	User
Pre-Condition	The plant profile must already exist in the system.
Post-Condition	A user has changed plant profile to fit the specific plant he/she owns.
Base sequence	1 .User modifies relevant profile fields. 2. User presses "Save" button. 3. System saves modified plant profile. 4. System returns status response to client.
Exception sequence	2A: User gets error message when invalid information is entered.

Table 15: Use Case – Control watering

Use Case	Control watering
Actor	User
Pre-Condition	A plant profile must already exist in the system.
Post-Condition	1. The updated information for the profile is stored in the system. 2. The system will display when the operation finishes.
Base sequence	1. User presses the "Set watering options" button 2. User sets the amount of water. 3. System triggers watering of plant.
Exception sequence	2A. Wrong input will cause an error dialog.

Table 16: Use Case – Delete plant profile

Use Case	Delete plant profile
Actor	User
Pre-Condition	A plant profile should be selected.
Post-Condition	Plant profile removed from the system.
Base sequence	1. User selects profile that they wish to delete. 2. User presses the "Delete profile" button. 3. System removes profile from database
Exception sequence	2. User may cancel the deletion of a profile on the confirmation dialog.

4 Design

The purpose of this part is to define the architecture, technologies, design patterns and UI choosing of the system. For a better explanation, this part contains several pictures of class diagrams and sequence diagram.

Regarding the requirements of SEP4, the project was divided in three main sections, each having different specifications.

4.1 Architecture Diagram

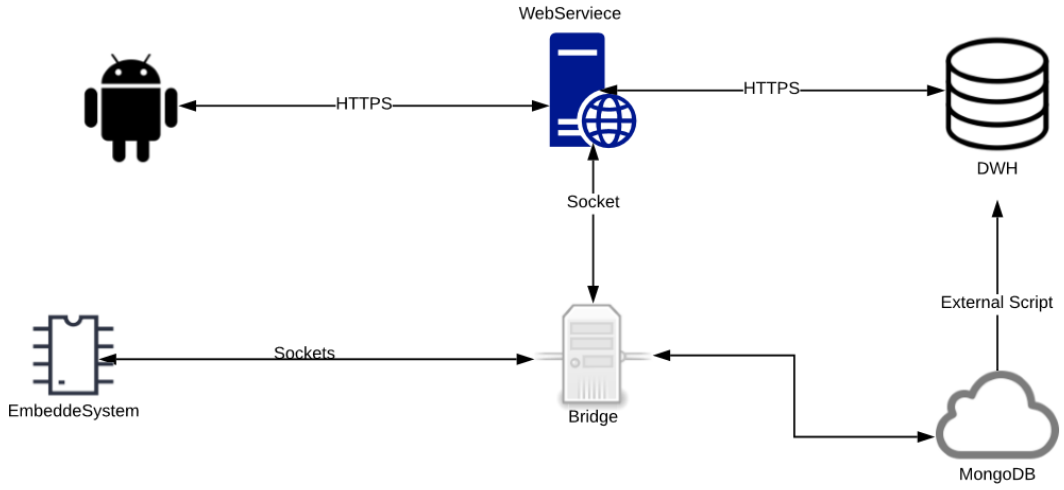


Figure 2: Architecture diagram model

The Architecture diagram shows an overall view of how the data that will be passed through the System. The microcontroller is responsible for sending data to the bridge application utilizing the LoRaWAN networking standard. In turn, the bridge application handles communication between the embedded system, the database and WebAPI (see section 4.3 IoT implementation design). The data from the MongoDB is put into the data warehouse. Following that, the WebService can invoke queries onto the data warehouse over HTTP. The WebService communicates with the Android application over HTTP/S where both the sending and receiving sides must have the same object structure for passing data onto one-another.

4.2 Sequence Diagram

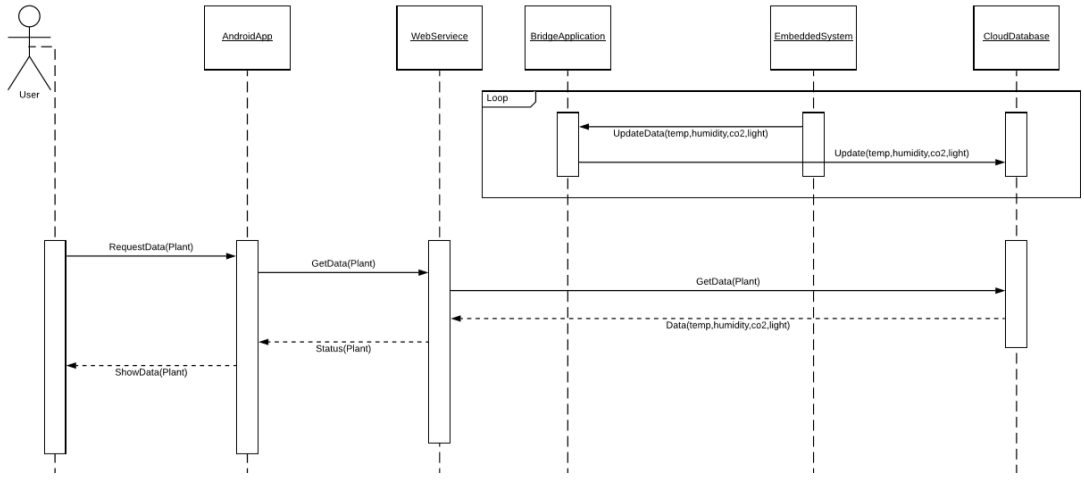


Figure 3: Sequence diagram – Get Status

The sequence diagram above shows the sequence of getting the status of a plant, including the Temperature, Humidity, Light , and CO2. The embedded system gets the measurements from the sensores and sends it to the bridge application, which sends it in turn to the database. This happens continuously. When the users want to check those data, they can specify the plant they want on the android app, the android app then sends a get request to the web service, carrying a message containing the plant to be checked. The web service looks up for that plant in the database, gets the data and sends it back to the android app.

4.3 IoT Implementation Design

Bridge Application

The bridge application consists of three handlers which will deal with the communication in each of the three parts of the system: the embedded system, the database and the webservice.

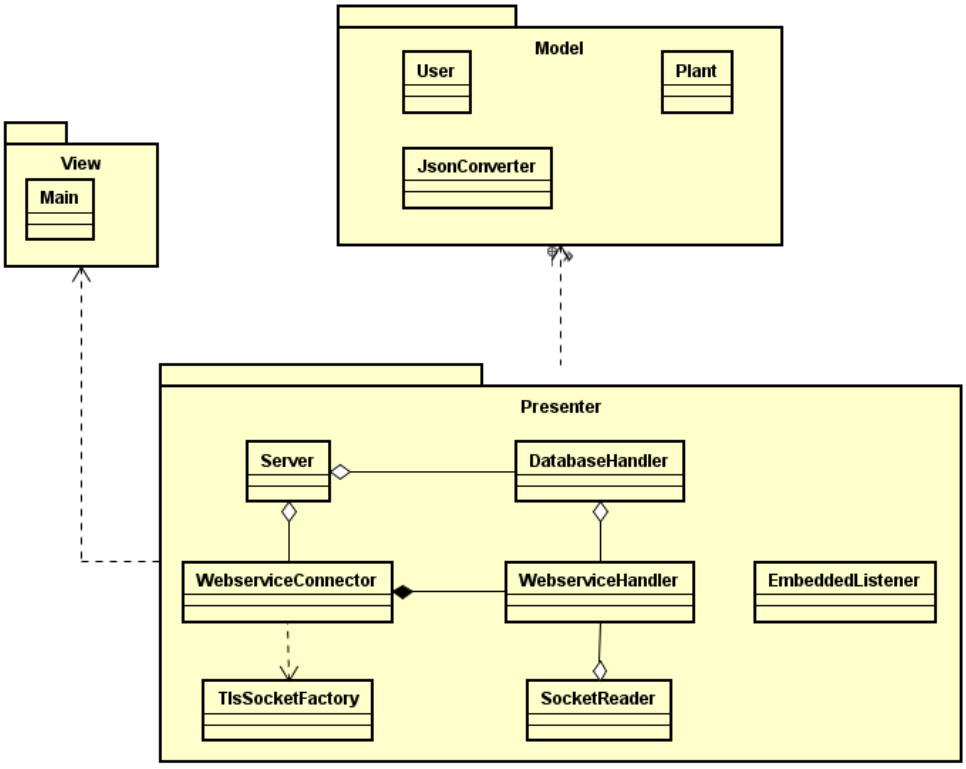


Figure 4: Analysis Diagram Bridge Application

The communication with the database uses the MongoDB driver.
The communication with the embedded system is through a Socket protocol as well as the webservice. The model classes are used to encapsulate data for users, plant profiles and plant sensors respectively. The WebserviceConnector maintains the socket connection to the web service and uses the IWebServiceHandler to perform actions matching the protocol commands and writes a response to the socket.
The socket communication is based on json and the model objects are serialized to json and sent over the socket connection. The protocol for communication is as follows:

Message	**Response**
getplantprofile:id	json/null
getaccount:json	json/null
removeaccount:json	accept/reject
removeplantprofile:id	accept/reject
addaccount:json	accept/reject
modifyaccount:json	accept/reject
all other cases	Error

Zero byte is used for delimiting the individual messages and responses. All messages and responses are UTF-8 text.

Embedded System

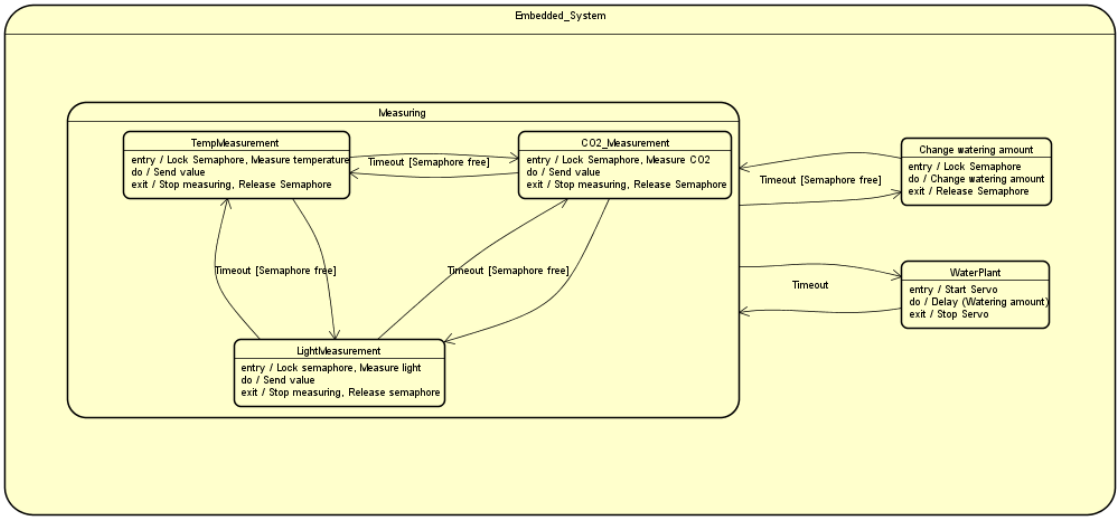


Figure 5: Analysis Diagram Bridge Application

The embedded system depends on Semaphores to change from a state to another. Each task in the system checks the semaphore status every specific number of milliseconds. If the semaphore is free, it locks it and sends the value that it took from the sensor to the bridge application, then releases the semaphore so another task could take place. However, the watering task starts the servo for a specific amount of time “depending on the specified watering amount" without checking the semaphore. This task starts every specific amount of time depending on how often the plant needs to be watered.

4.4 Data

Cloud Based Data:

IoT system needs to exchange data with android application. To make it possible cloud-based database were necessary. MongoDB is chosen in this case and it fulfills all requirements.

MongoDB is NoSql database program, it stores the information in JSON format into a document. The list of documents it is named collection, it will be caped. Caped collections support fixed-size collections. This type of collection maintains insertion order and, once the specified size has been reached, behaves like a circular queue. Capped collections will help with limited available space in the cloud. The data

contained in the document will include the readings taken from embedded system such as the temperature, CO2 and will also hold the "PlantID" and "PlantName" used for identification combined with the autogenerated ID.


```
_id: ObjectId("5cc97bec1c9d440000423d03")
PlantID: 1
PlantName: "Roses"
Temperature: 32
CO2: 430
Humidity: 54
AmountOfWater: 100
HoursSinceWatering: 2
Light: 1001
DateTime: 2019-04-05T23:10:05.000+00:00
```


Figure 6: MongoDb document

SQL Server:

Data from MongoDB it is transferred into Microsoft SQL Server for data warehousing purposed. SQL server it is used for ETL Process and it has the following phases: source database, staging area and final dimensional database. In the picture below it can be observed the EER diagram for the source database, which contain informations from all sensors: temperature, light, CO2, humidity, amount of water, hours since watering and reading date, as well as id plant and name.

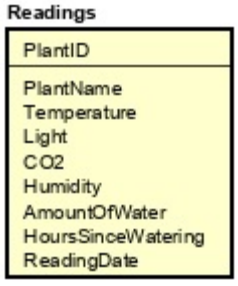


Figure 7: *Source database design*

In the staging area we begin separating the data through dimensional modelling. We create two dimension for date and time which which create a more detailed representation of the date by being able to query for example by the day of the week, or to look for a specific time of the day such as "Lunch", "Afternoon". Another dimension is the "D_staging_plant" which contains the primary key, business key, plant name and the validity consisted of two values "validFrom", "validTo", these are used for handling slow changes such as new addition of readings. Staging database has five fact tables.

These represent the information retrieved from the microcontroller alike light, humidity, temperature. The tables are only for the temporary storage of informations so the y do not contain a primary key and they hold the value of the measurement and the date and time at the moment of reading.

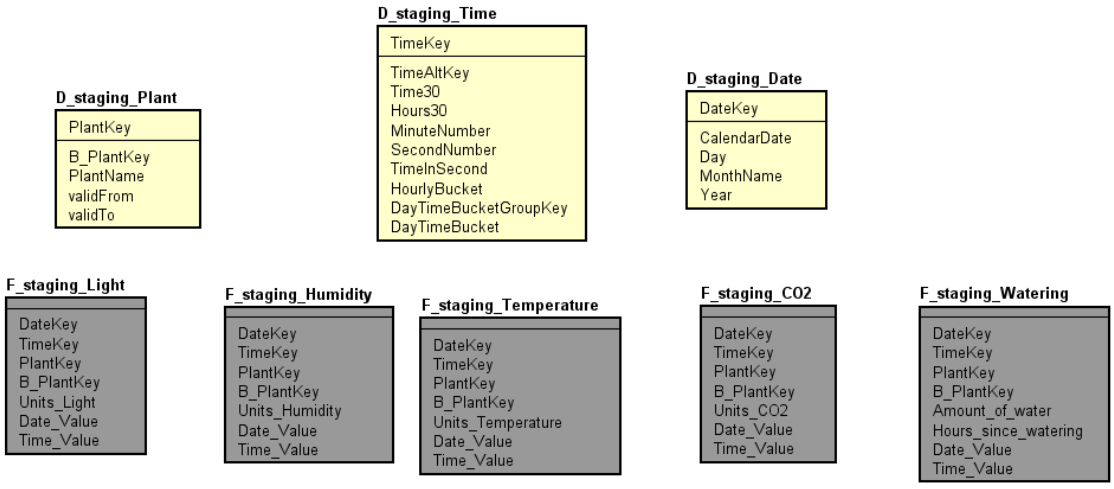


Figure 8: Staging database design

The final dimensional model is presented in the picture below. It contains the same tables as in staging but in here it is presented the relationship between them, each fact table contains the unit of measurement and the foreign key from the dimensional tables without business keys. The data from the plant, date and time dimension are shared between fact tables.

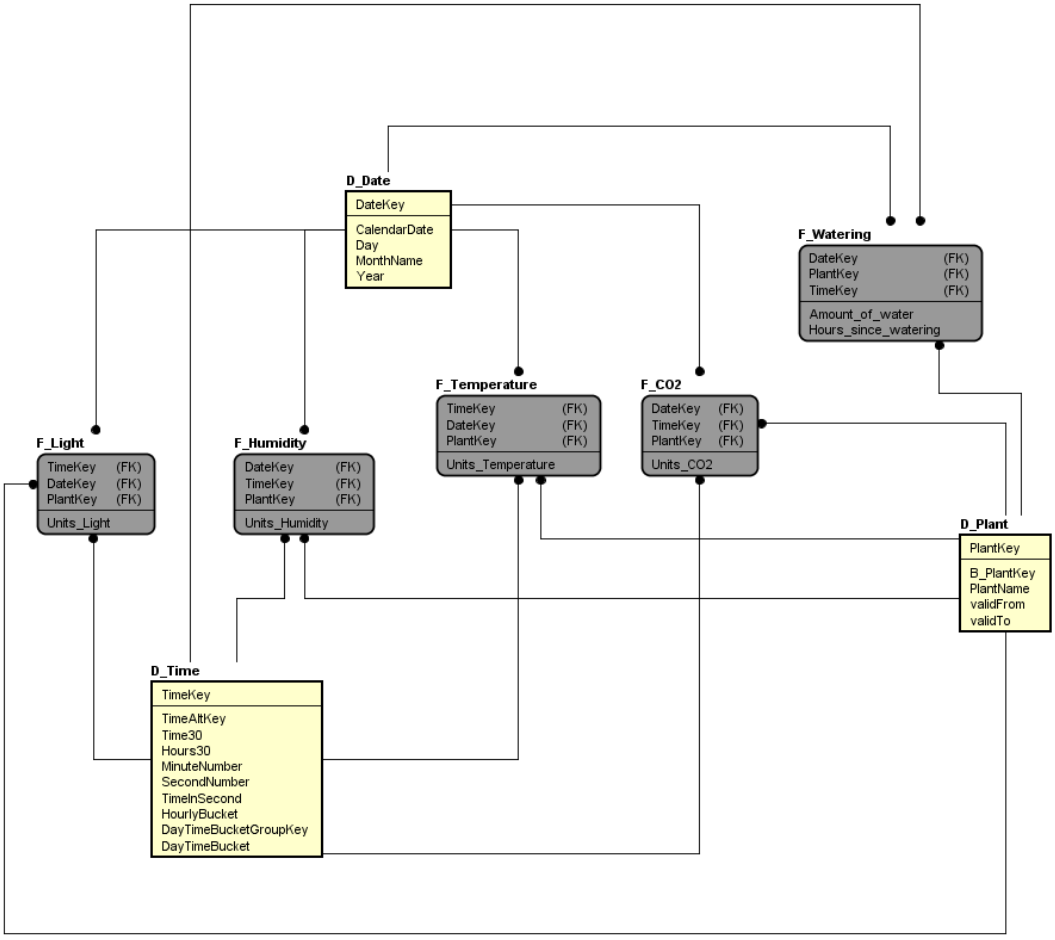


Figure 9: Final dimensional database design

To have a clearer view over the ETL process, an activity diagram was created. In the picture below it can be observed the procedures that were performed. The first step is to extract the data from the source table into the staging area, then creating the necessary date and time dimension. Afterwards we populate the temporary fact table and update the surrogate keys before inserting into the final dimensional model.

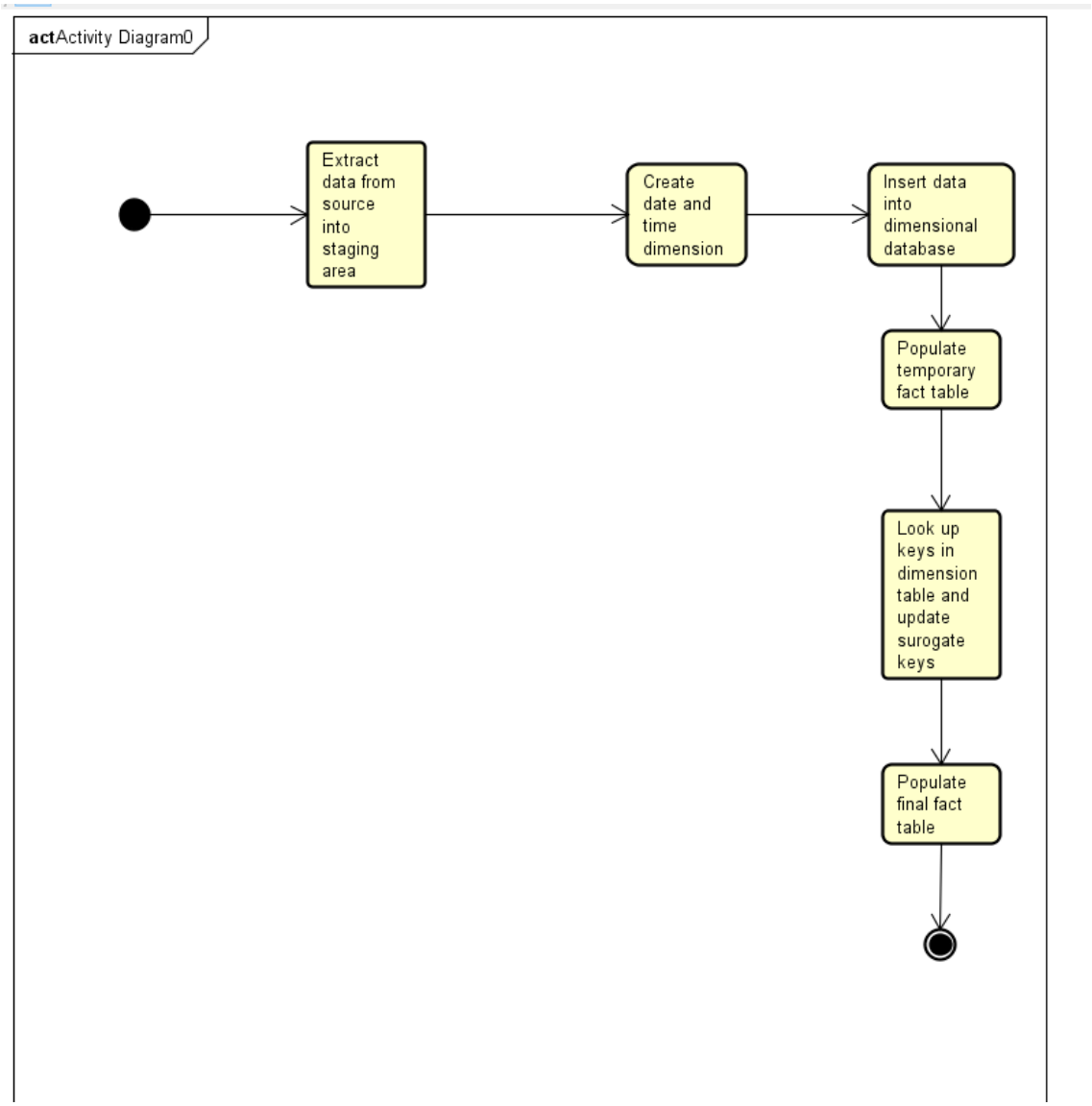


Figure 10: ETL Diagram

Web API

Connection establishment between Android app and Sql server

ID	Commands	Param1	Response
1	GETPLANTPROFILE	None	List of plants
2	GETPLANTID	JSON OrderData	PlantID assigned by database

Figure 11: Get Request

Description of web related operations

GETPLANTPROFILE:
The android app sends “GETPLANTPROFILE” and the server sends an array of plant profiles.

GETPLANTID:
The android app sends a command “GETPLANTID” and in turn, the server returns plant with signed ID.

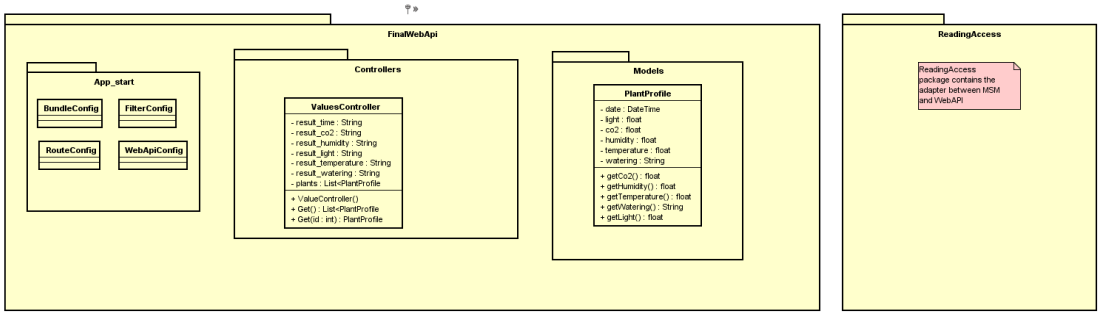


Figure 11: WebApi Class Diagram

The most important classes in the WebApi are the ones representing controller (ValuesController) and Model (PlantProfile) and they the configuration classes are present also but they don’t need special attention as they are not modified at all and they were created from a template.

Socket Communication with bridge application:

Bridge application stands for communication between IoT and data part of system. Socket communication is used to transfer data. Server of bridge application is implemented in Java and Client is implemented in C# programming language. Communication is safe, the server request password from client to established communication. Client is able to request information from Server.

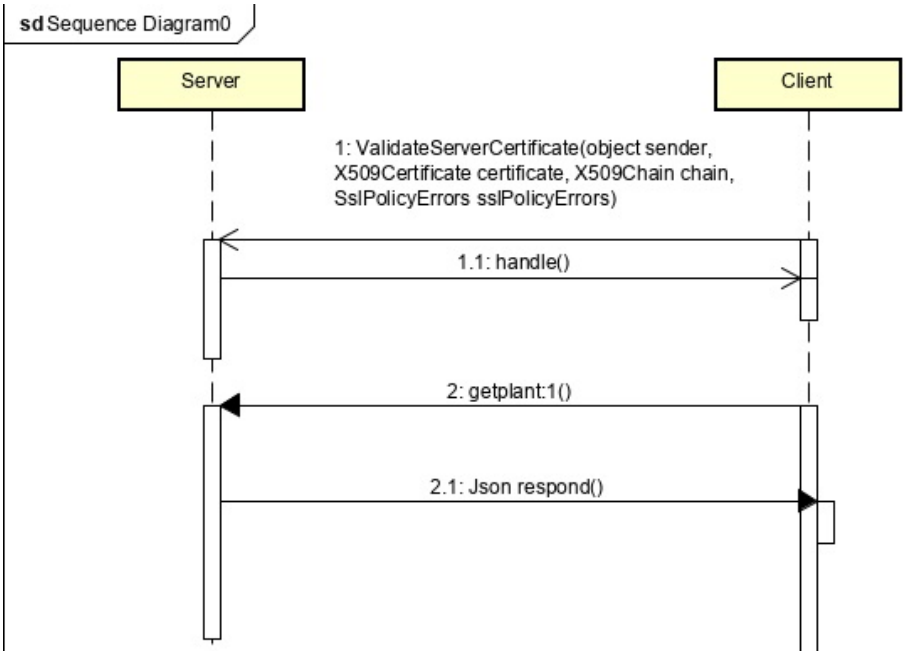


Figure 12: Sequence diagram socket connection

4.5 Android Application

The responsibility of the Android application is to grant the end-user the possibility to control a plants' environmental conditions. A user has the possibility to register and thus login afterwards. This gives them access to sections where they are able to see the list of their associated plants, they can register new plants and modify their account and profile settings.

The implementation of the application follows the MVVM – Model-View – ViewModel system architecture. The following diagram represents the use of MVVM:

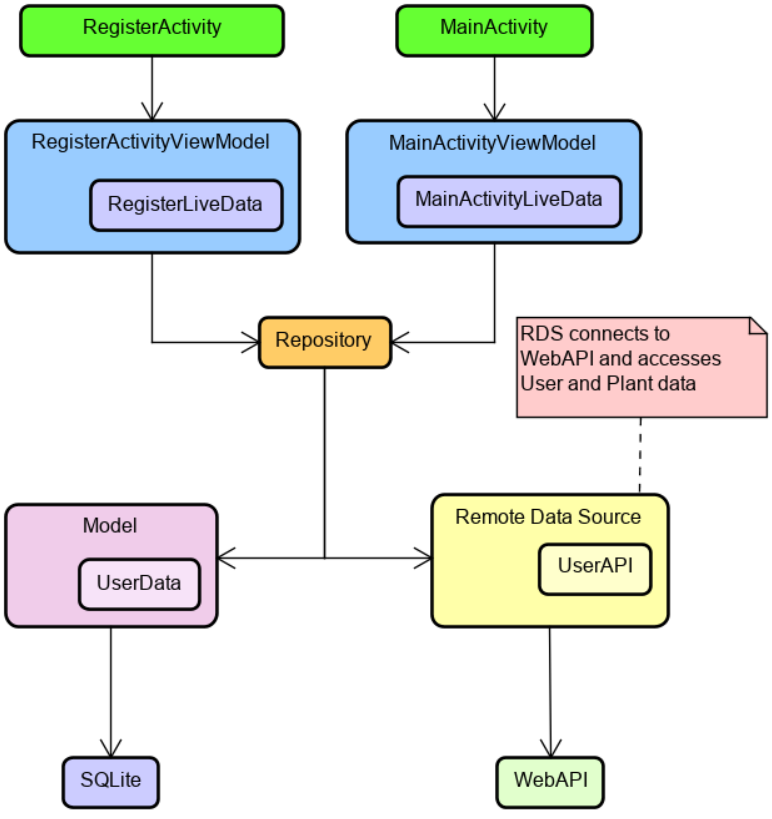


Figure 13: MVVM Architecture Diagram (Simplified Architecture Class Diagram)

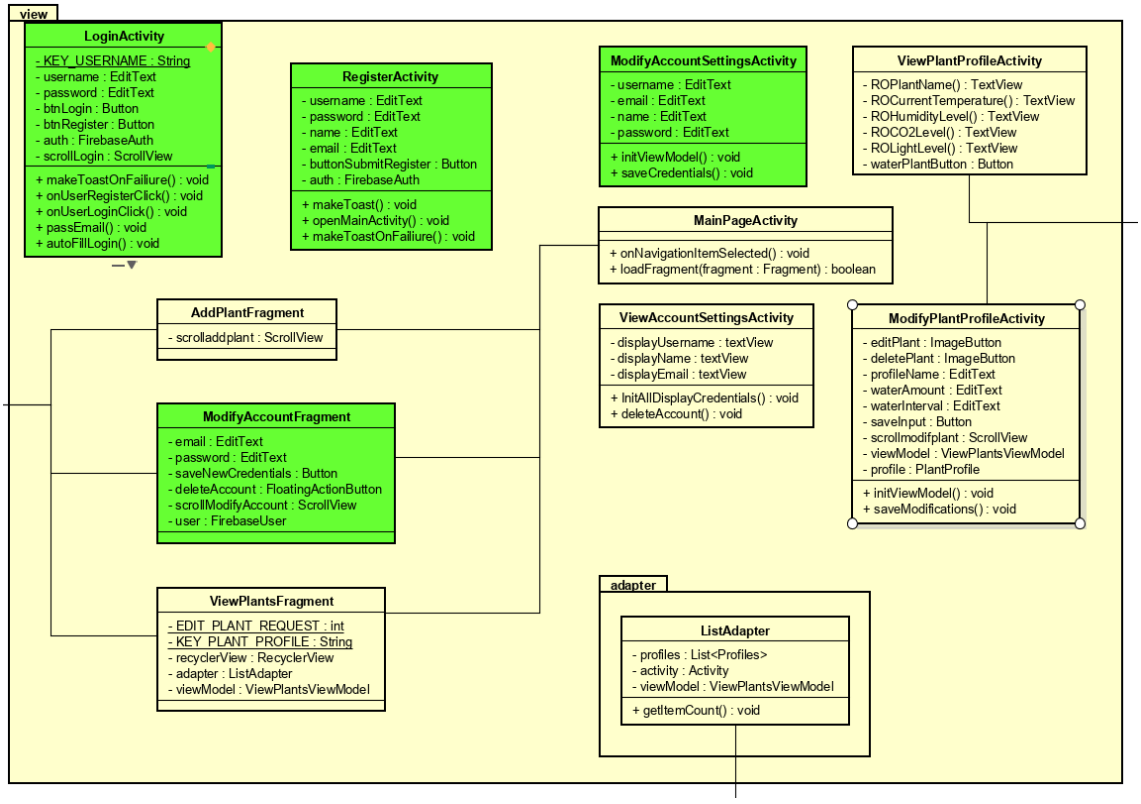


Figure 14: Design of MVVM Architecture Diagram – View

In the View, there are the view controllers and fragments that are responsible for handling the View elements and updating them. The view is using ViewModels that are in the view-model package to perform actions in the system. ViewAccountSettingsActivity and ModifyAccountSettingsActivity are responsible for presenting the users' details and making changes to them. ViewPlantProfileActivity is responsible for creating the plant profile by sending a request to the ViewModel. ViewPlantActivity has the purpose of displaying plant data, whilst ModifyPlantProfileActivity changes only the waterAmount and waterInterval attributes.

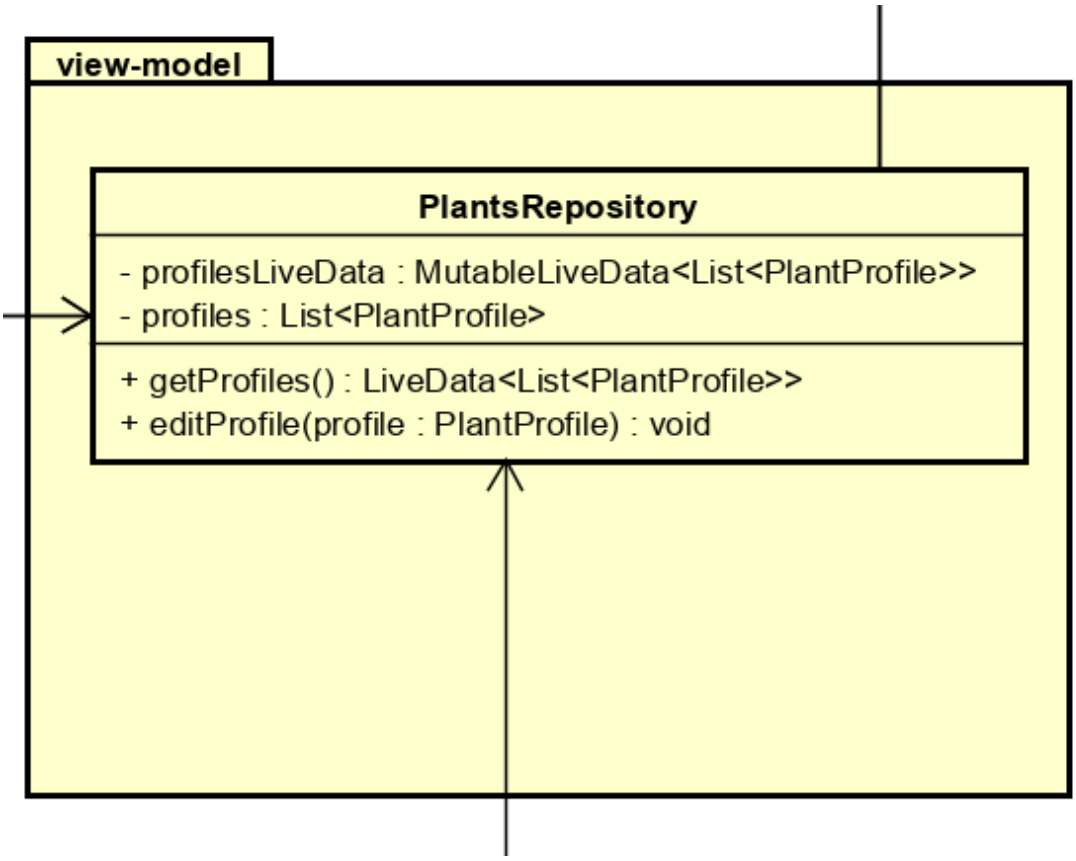


Figure 15: Design of MVVM Architecture Diagram – View-Model

The view-model's purpose is to handle actions from the view and to serve as a bridge between the view and the model. All view-models pass data to their respective associated repositories. UserViewModel handles operations for user-related activities, PlantViewModel from plant related activities and finally, Login/RegisterViewModel – user specific credentials.

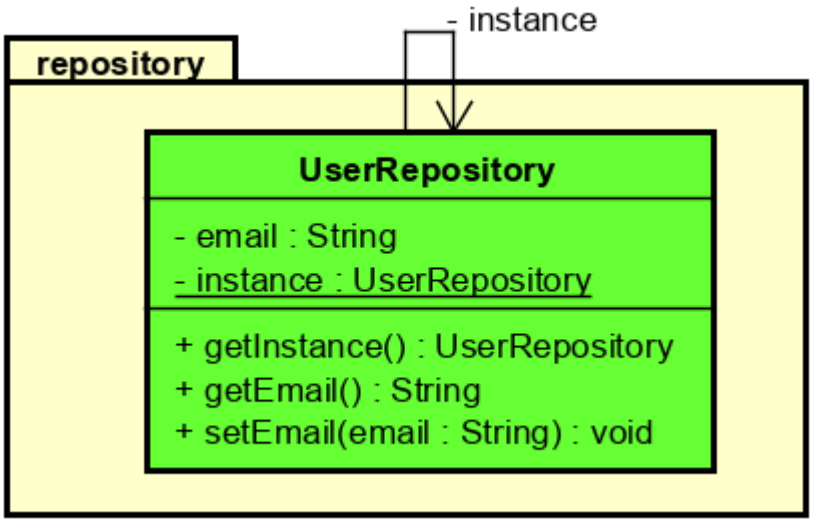
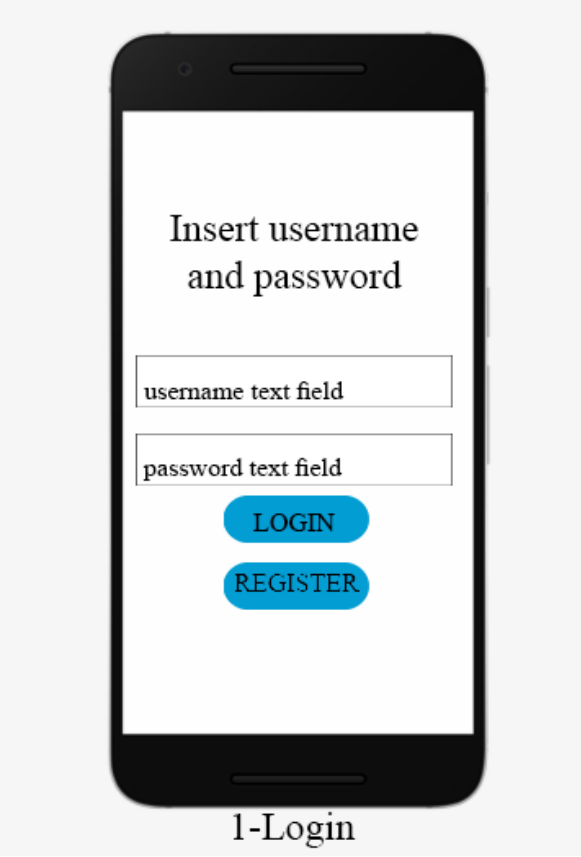


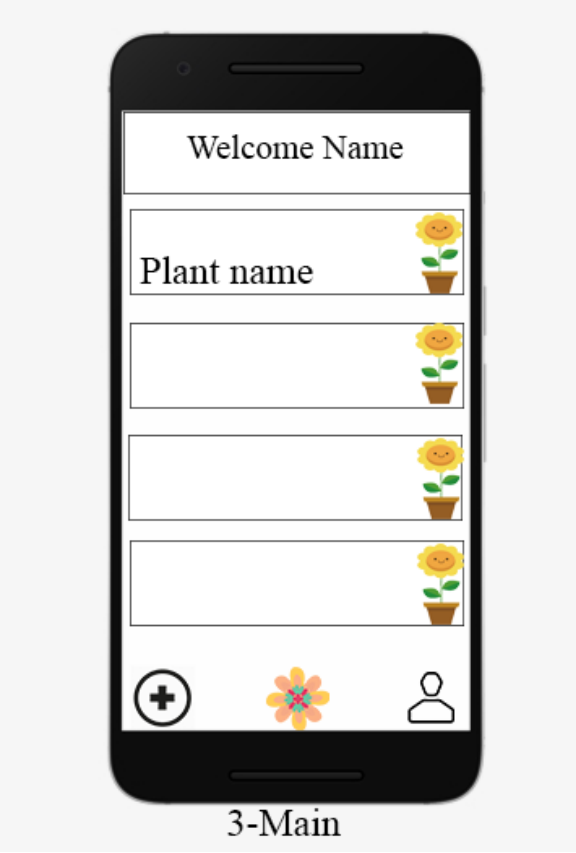
Figure 16:Design of MVVM Architecture Diagram – View-Model

The repository in the design implementation serves as a facade to the database and WebAPI. The Plants repository is handling CRUD operations on model entities. They implement Singleton design pattern to ensure that there is only one instance of the mentioned repositories in the system. The repositories are using DTOs from the model to exchange data with the WebAPI.

UI Design and Navigation

From the early stages of the project, the main focuses regarding the User Interface were to make every layout of every single activity in the Android Application simple, intuitive and eye-catching. In the Design phase, the Android group designed sketches of the layouts.

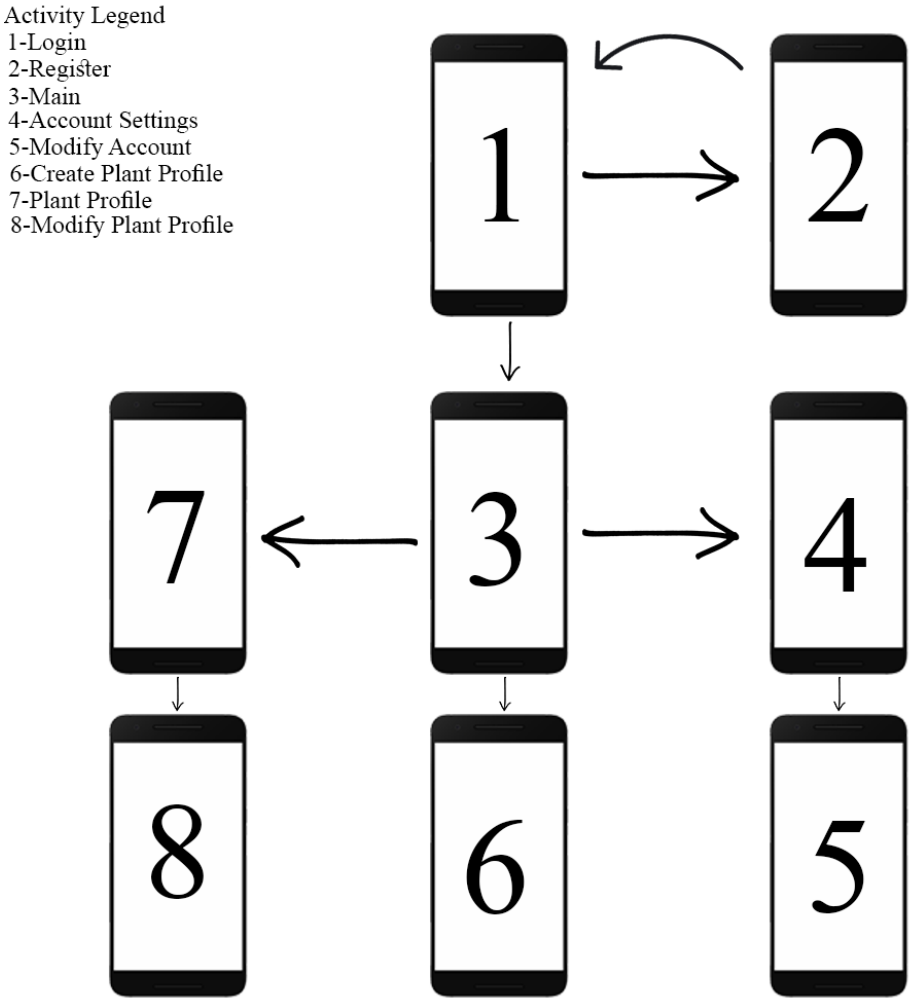




As it can be seen from the above represented sketches, the layouts have been designed to be as simple and straightforward as possible. Along with the layouts sketches, the group designed an interaction diagram for all the layouts, which has the

sole purpose of giving a better understanding of how the flow of events in the application works.

As it can be seen from the interaction diagram, there are 8 activities that the application uses: **Login**, **Register**, the **Main activity**, **Account Settings**, **Modify Account**, **Create Plant Profile**, **View Plant Profile** and **Modify Plant Profile**.



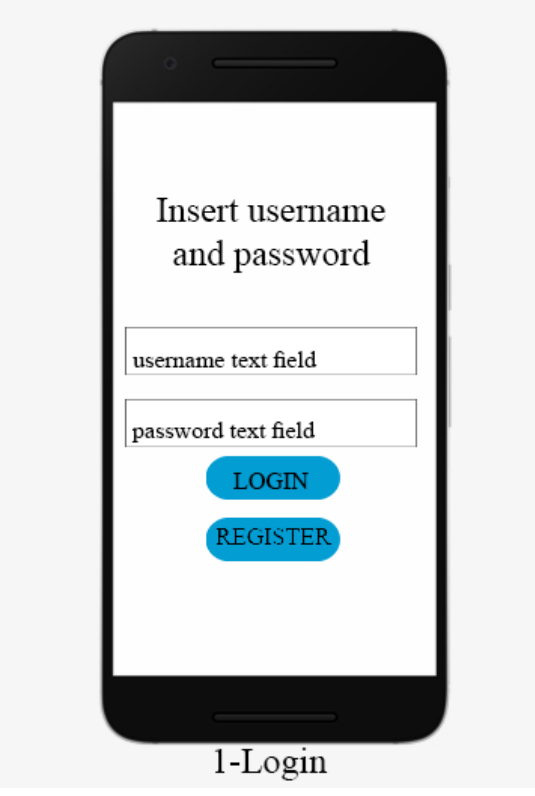
Login Activity

The **Login activity** is the one that the user is prompted with when first opening the application. From this activity, the user has two options:

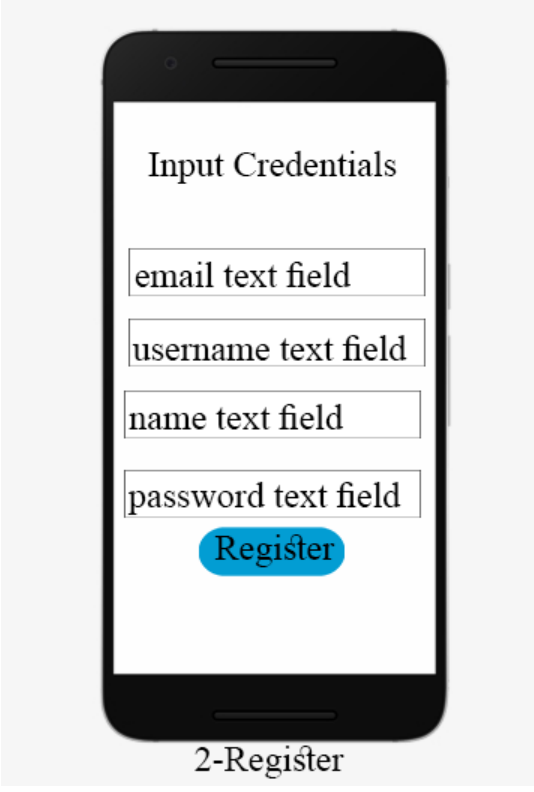
1. Input his login details of a previously created account and then press the “Login” button, from which he will be sent to the **Main Activity.**
2. Press on the “Register” button, getting sent to the **Register activity** where he is able to create a new account.

Register Activity

The **Register activity** is oversimplified and it allows the user to input four text fields with his account details: username, email, name and password, after which the user presses the “Register” button which prompts to the **Main activity.**



1-Login



2-Register

Main Activity

The **Main activity** displays a custom welcome message for the user, based on his name that he inputted when he created his account, and below that it displays a list with plants. This activity also features a bottom navigation bar, which is also present in all the other activities, apart from the **Login** and the **Register** ones, with 3 options:

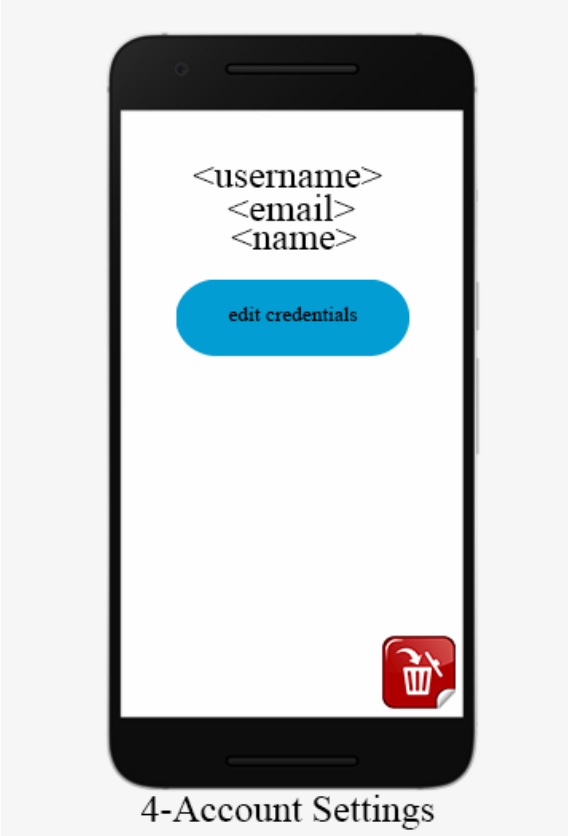
- The button on the right sends the user to the **Account Settings activity**
- The option in the middle has the purpose of sending the user to the **Main activity**
- The option in the left redirects the user to the **Create Plant Profile activity**

One last thing the user can do from the **Main activity** is to click on one of the existing plant profiles, which sends the user to the **View Plant Profile activity**

Account Settings Activity

In the **Account Settings activity**, the username, email and name of the user are displayed on top, along with an “Edit credentials” button, which, if pressed, sends the user to the **Modify Account activity**. This activity encloses a “Delete Profile” button, which will delete all the data of the user from the database, including the login details.

3-Main



4-Account Settings

Modify Account Activity

In the **Modify Account activity**, the user has 4 text fields where he can input his new details, in the case he wishes to change any, and then a save button, that, if pressed, saves and updates all the new data in the database.

Create Plant Profile

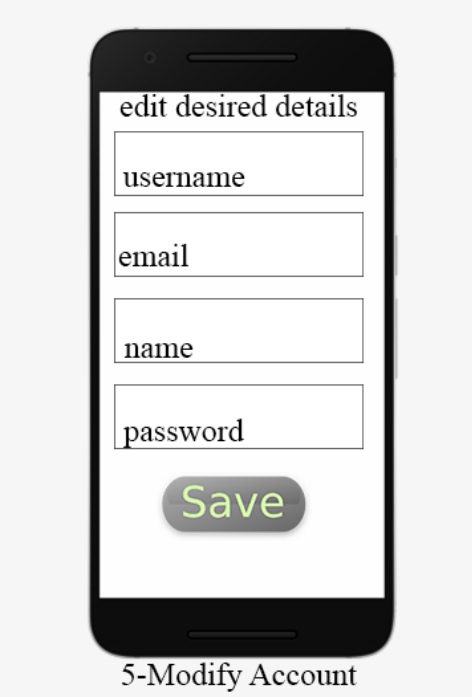
In the **Create Plant Profile activity**, the user is given to option to create a plant. He has to input a name for the Plant, and, in addition to this, he can select a predefined plant template (which defines the ranges of the temperature, humidity and carbon dioxide) from a drop down list.

View Plant Profile

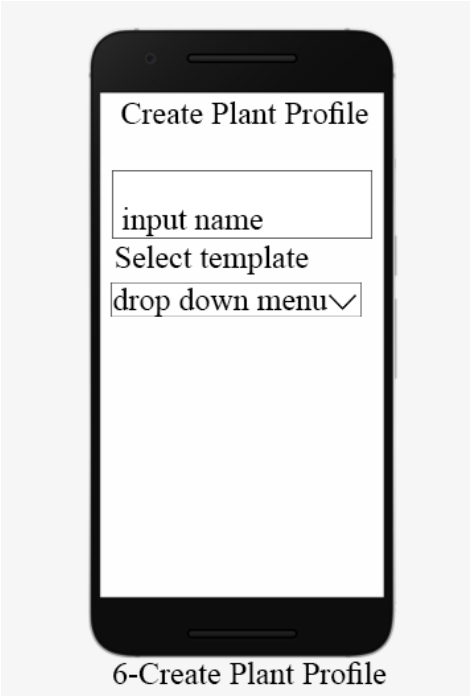
In the **View Plant Profile activity**, the application displays the name of the plant, the current external temperature, the humidity level and also the CO2 level. There is also a "Modify Plant Profile" button, that the user can press, which sends him to the **Modify Plant profile Activity**. This activity also features a "Water Plant" button that can be pressed to remotely water the plant. Lastly, a delete button also appears in the bottom of the activity, which can be pressed to delete all the data regarding that certain plant from the database and remove it from the system.

Modify Plant Profile

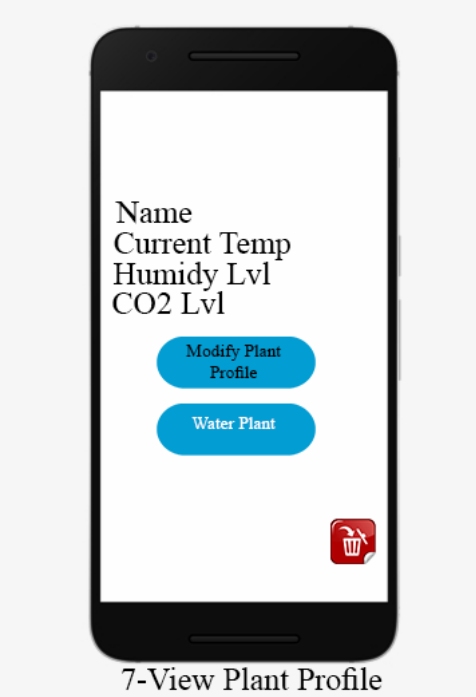
In the **Modify Plant Profile activity**, there are multiple text fields, where the user is able to change the temperature range, the CO2 level, the humidity level, the amount of water and the watering interval. For the changes to apply to the database, the user has to press the “Save Button”.



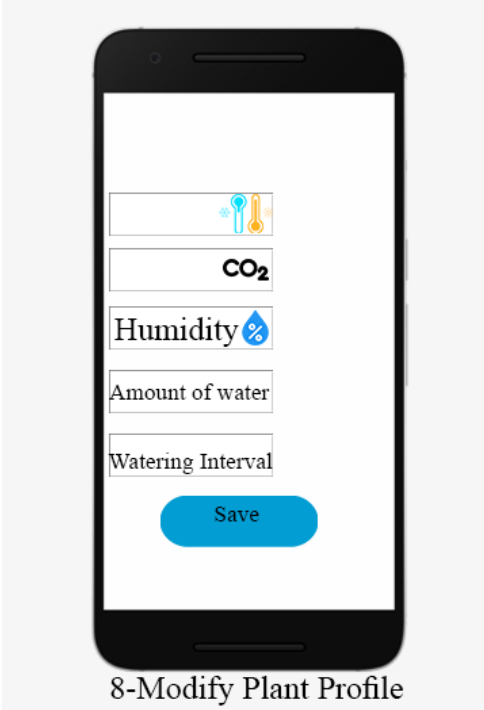
5-Modify Account



6-Create Plant Profile

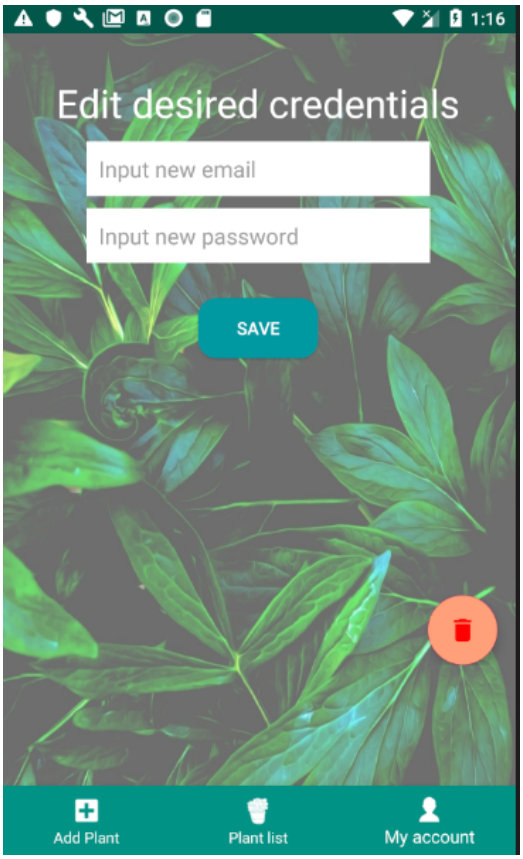


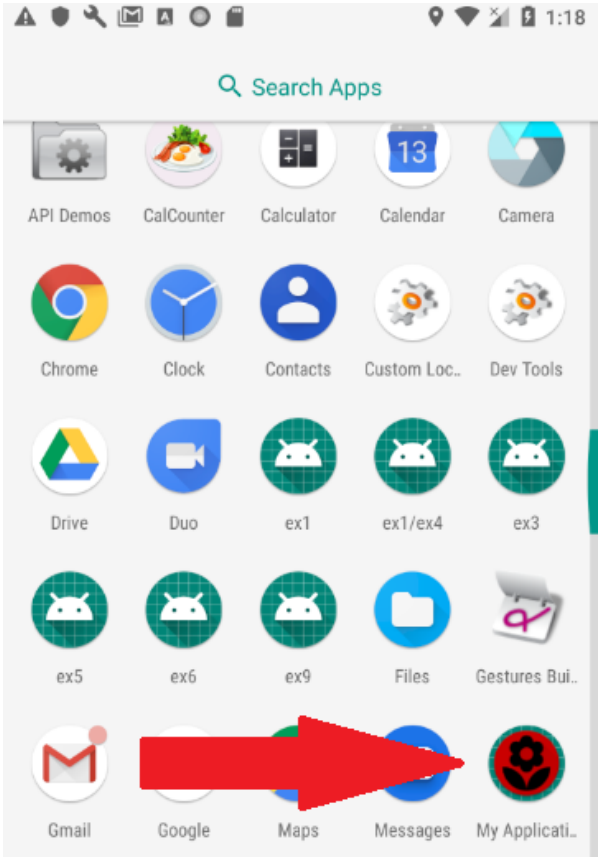
7-View Plant Profile



8-Modify Plant Profile

In regards of the actual user interface, the main characteristic of the layout is a thematic green-themed background, the group also deciding to include a custom icon for the application.





5 Implementation

5.1 IoT - Bridge Application

The presentation layer consists of three separate parts, each responsible for running one of the connectivities that the bridge application is needs to maintain.
The IDatabaseHandler interface allows other parts of the system to perform actions on the MongoDB database.
The WebserviceConnector maintains communication with the webservice, running externally from the bridge application, through a TLS-encrypted socket. The IWebservice interface abstracts the set of actions, which the input from the socket connection should be able to invoke on the bridge application.
In the system's current state, all of these actions are translated directly to database operations, and the interface's implementation has no logic in itself, simply parsing data on from the IDatabaseHandler. However, since the socket connection, in future, is likely to need to perform more non-database operations, the layer is necessary to ensure that the webservice can continue to interact through a single interface.
The IEmbeddedConnector abstracts the connection to the loriot server and allows the EmbeddedListener to receive the payload from the sensors as a hexadecimal string by calling getMessage().

Initializing the system

To start the presentation layer, the Server class is instantiated. The constructor of this class creates instances of DatabaseHandler, WebserviceHandler, WebserviceConnector, and EmbeddedListener. WebserviceConnector and EmbeddedListener implements the Runnable interface and are submitted to an ExecutorService. Thus, with the DatabaseHandler running in the main thread, the system now has threads for each of the connectivities.

Socket Connection

The socket connectivity with the webservice is built to allow only one connection at a time. The system should have only one bridge application and one webservice, so there is no reason to allow multiple connections.
The WebserviceConnector is constantly ready to accept a new connection from the webservice and when it does, it kills the old connection, to avoid having idle socket connections hanging in the system. When accepting a connection, the socket is passed to a new instance of the Runnable SocketReader, which is then submitted to the ExecutorService.

```
Socket socket;
SocketReader sr = null;
while (true) {
    System.out.println("Listening for new connection");
    socket = ss.accept();
    STATUS = "Client is connected";

    if(sr != null) {
        sr.kill();
    }

    sr = new SocketReader(socket, webserviceHandler);
    Server.executorService.submit(sr);
}
```

Figure 17:Server socket loop

The SocketReader reads every byte from the socket and puts in a ByteArrayOutputStream for buffering. It continues until a 0x00 byte is received, after which it will create a new string from the contents of the buffer and reset the buffer. The string is parsed to the handle() method, where the string is split into a command and parameter, delimited by the first occurrence of a comma. A switch on the command will process the parameter accordingly.

```
ir = new InputStreamReader(socket.getInputStream());
bw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));

ByteArrayOutputStream baos = new ByteArrayOutputStream();

while(!socket.isClosed()) {
    int c;
    while((c = ir.read()) != 0) {
        baos.write(c);
    }

    String str = new String(baos.toByteArray(), "UTF-8");
    baos.reset();
    handle(str, bw);
```

Figure 18: Socket connection

Figure 19: Authentication check

When the connection is established, the webservice must first send and authentication string. Until the boolean 'auth' is true, the handle method will refuse to perform any actions.
The authtoken is hardcoded into system. Since it is never used client-side and always sent over an encrypted socket, this is not a threat to security.

```
if(!auth) {
    if(input.equals(authToken)) {
        auth = true;
        respond("accept", bw);
    }
    else {
        respond("reject", bw);
    }
    return;
}
```

Embedded Connection

Data from the sensor system is received from a websocket, listened to by a java.net.http.Websocket.Listener implementation, which also implements the IembeddedConnector interface. It places each received message in a queue.
The EmbeddedListener will call getMessage() on the interface once every second.
If the internal queue in the websocket listener is empty it getMessage() returns null and EmbeddedListener will wait another second before calling again.
If the queue is not empty, a JSON string from the loriot server is returned.
EmbeddedListener will then extract the data element as a string, and convert it to a

char[], so that the individual hexadecimal characters can be used for converting each value contained in the data string.

```
String update = wsl.getMessage();
if(update != null) {
    char[] hex = Document.parse(update).getString("data").toCharArray();
    int hum = Integer.parseInt("" + hex[0] + hex[1], 16);
    int temp = Integer.parseInt("" + hex[2] + hex[3], 16);
    int co2 = Integer.parseInt("" + hex[4] + hex[5] + hex[6] + hex[7], 16);
    int light = Integer.parseInt("" + hex[8] + hex[9] + hex[10] + hex[11], 16);
    int water = Integer.parseInt("" + hex[12] + hex[13], 16);
```

Figure 20: Data extraction

The model class 'PlantProfile' encapsulates the data in the structural layout expected by the database. A new instance of this class is constructed and the values is set on it. Finally the PlantProfile object is parsed to the setPlantProfile() method of the IDatabaseHandler object referenced in the EmbeddedListener.

```
PlantProfile pp = new PlantProfile();
pp.AmountOfWater = 0;
pp.CO2 = co2;
pp.Humidity = hum;
pp.HoursSinceWatering = water;
pp.Light = light;
pp.Temperature = temp;
pp.PlantName = name;
pp.PlantID = id;

databaseHandler.setPlantProfile(pp);
```

Figure 21: Data object construction

Database Connection

The DatabaseHandler, implementing the IDatabaseHandler, uses the libraries provided by MongoDB to connect to and manipulate the document database. The constructor takes a connection string needed to create the connection to the database. After connecting, the collections used by the DatabaseHandler are retrieved as MongoCollection<Document> objects. The methods in DatabaseHandler can then use the collection objects to get and set data.

```
public DatabaseHandler(String hostString) {
    mongo = MongoClients.create(hostString);
    database = mongo.getDatabase("Project");
    if(database != null) {
        STATUS = "Connected to mongoDB";
    }

    plantCollection = database.getCollection("PlantProfiles");
    accountCollection = database.getCollection("Accounts");
}
```

Figure 22: Establishing database connection

Apart from interacting with the database, the class also caches the data received from the embedded system, so that this data can be returned to the webservice without performing a database query. The caching is handled by storing the last received PlantProfile data object into a HashMap in the DatabaseHandler. If the cache is empty for a requested ID, the DatabaseHandler instead queries the database.

View

The view package contains a single class intended to provide an administrative command-line interface on the machine running the bridge application. It accepts a few simple text commands that allows an administrator to perform a simple status check on the system, query the database, as well as to perform a proper shutdown of the application. This class contains the applications main() method, in which the instance of the Server class is constructed. The connectivity objects used in the Server object are declared public, as to allow the main() method to reference them through the Server object reference, whenever status retrieval or database queries need to be performed.

5.2 IoT - Embedded System

For the project five different tasks have been used including the LoRaWAN transmitter for sending the data from the board. The other sensors are:

- CO2 sensor
- Combined temperature and humidity sensor
- Light sensor
- Servo motor

All the sensors are connected to an Arduino mega 2560 board running the FreeRTOS operating system.

FreeRTOS

FreeRTOS is a Real time operating system which provides multitasking capabilities. It also provides method and functionality for concurrency.
The multitasking functionality is used throughout the system to ensure measurements with each sensor are done frequently. Each sensor has a defined task which is executed in the main method of the program.
Most of the handling of tasks are taken care of in the sourcefiles of the FreeRTOS library, but are accessed by our system through methods.

Task

Each of the sensors in system have their own task method. The tasks contains the code and method necessary to get the measurements. To make the tasks work as tasks, each task method has fulfill some requirement for the task to work. These requirements are necessary for the FreeRTOS taskManager to work correctly.
First of all the task method has to take a void pointer (pvParameters) in the definition. This is used to pass a value into the task. The parameter is a void which allow the instance to be any type of data, this means that the data has to be cast back to the original format to be used.[1]
The pvParameters pointers are not used in this system, but because FreeRTOS require them, has it been necessary for the group to defined and cast them anyway.

The task's functionality that has to be executed multiple times, must be defined inside of an infinite loop. This is done to ensure that the method keeps running forever. The purpose is that the FreeRTOS operating system manages the timing of executing each task. By putting the functionality in an infinite loops, allows the FreeRTOS taskmanager to pause and resume tasks as intended, without the method ever ending.
All of the task has the FreeRTOS method vTaskDelete in the end of the task. The method is used to catch the taskmanager if it accidentally exits the infinite loop.

Delays

Delays are used throughout the system in the attempt of controlling method execution frequency and to force wait time for measuring. The group decided to use the method vTaskDelay from the FreeRTOS library for this purpose. The vTaskDelay method moves the task into the Blocked state for a defined number of ticks. This prevents the task from using any processing time while waiting to proceed.[2]

Main method

The main method is used for setting up the drivers and creating the tasks. FreeRTOS contains two different methods, xTaskCreate and xTaskCreateStatic. The non static version of the method automatically manages the use of memory form the FreeRTOS memory heap, where the static version depends the user to specify the location.[3] The group decided on using the xTaskCreate method because manual memory management is not needed in this system.

The xTaskCreate method takes each of these variables to create a task:

- Task method.
- Char string as a name for the method.
- Unsigned short to tell the kernel the depth of the stack.
- Void pointer to pass the task a value.

- UBaseType_t priority.
- TaskHandle_t pointer.[4]

The method returns a pdPASS or a pdFAIL, which is internally used in the system by the FreeRTOS. pdPASS indicates that the task has been created successfully. pdFAIL indicates that the task has not been created because of problems with FreeRTOS memory heap.[5]
The other purpose of the main method is to setup the driver. Each of the drivers has a create method, which is called through the driver's header file.
An important function of the main method is to call the FreeRTOS method vTaskStartScheduler, which manages the executions of the tasks.

PlantData

Plantdata is a struct which is used for storing the values from the measurements. The struct contains a value of temperature, humidity, CO2, light and a value for keeping track of the time since last watering. The plantdata is structured in the header file so all classes in the system can access and share the data.

Sensors

The functionality of the sensors is done through external drivers. The drivers and documentation have been provided by VIA University College. All of the drivers have built in error codes which are used throughout the system for status and error check.
Here is a short explanation of the how the drivers have been used for each sensor in the system.

CO2

The CO2 sensor uses the "mh_z19" driver. The system has two method for the sensor in the CO2_sensor file. One for the methods is the task method which does the

measuring. If the measuring goes wrong, it will print the error code. The other method is a callback method. This method is used to set the CO2 value in plant data.

Light

The light uses the “TSL2591” driver. The light sensor driver has multiple options for measuring various types of light. The group decided to use the method “tsl2591GetLux” which gets the measuring of visible light.

The system has two method in the light_sensor file. One method is a task method which activates the light sensor and prints out an error code if it does not work. The other method is a CallBack method which is used for setting the light value in plant data.

Temperature and Humidity

The temperature and humidity is combined into one sensor and uses the “HIH8210” driver. The system has only one method in the temp_sensor file, which used to invoke the sensor and measure both temperature and humidity. Both values are then used to set the temperature and humidity value inside of plantdata.

Servo motor

The servo motor uses the “rcServo” driver. The system has only one method in the servo_motor file. The method uses the driver to turn the position of the motor from the left to the right. This is done to simulate the watering mechanism. The driver method for this is called rcServoSet which takes a number between 100 and -100, for turning left to right. It also takes a port number for output on the board.

The data stored is an integer that is used to calculate the time since last watering. The method xTaskGetTickCount from FreeRTOS is used to get the number of tick since program start. This number is then stored in plantdata. The FreeRTOS has been configured to return a 32-bit unsigned int as the count. The number’s difference from

the current xTaskGetTickCount is then calculated to hours right before the value is sent in the loRaWAN task.

LoRaWAN

The system has two methods inside the LoRaWAN file. The first method is used to setup and connect the LoRaWAN. It used functions from the lora_driver to establish the connection. The method ends by trying to establish connection to the receiver.
When the connection has been established it can then proceed to the task method. The task method is responsible for formatting and sending the data. This is done by creating a lora_payload_t instance. The length of the payload and the port number has to be defined before it can be used. The length of the payload is an integer which defines how many bytes should be sent. Each byte is then loaded into each container of the payload. After this the payload can be converted into text and sent. This is also done by using methods from the drivers.

---

[1] FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide – page 51
[2] FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide – page 67
[3] FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide – page 39
[4] FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide – page 34 - 36
[5] Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf – page 52

5.3 Data Engineering

The implementation of the system has been done using upper mentioned technologies and solutions to make the process simpler and quicker. We used as technologies such as C#, SQL. Some of the most interesting parts are the connection between Bridge Application and WebService and also between the Bridge Application and MongoDB.

1.WEB-SERVICE

The web-service was implemented in C# and it's based on the .NET Framework Template. When it comes to the implementation part it does contain two small projects, the API itself and also another project that's an adapter between the database made in Microsoft SQL Server Management and the web service that being the reason why there exists a dependency of the web API on the adapter.

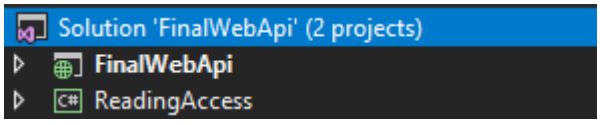


Figure 23: Web API solution explorer

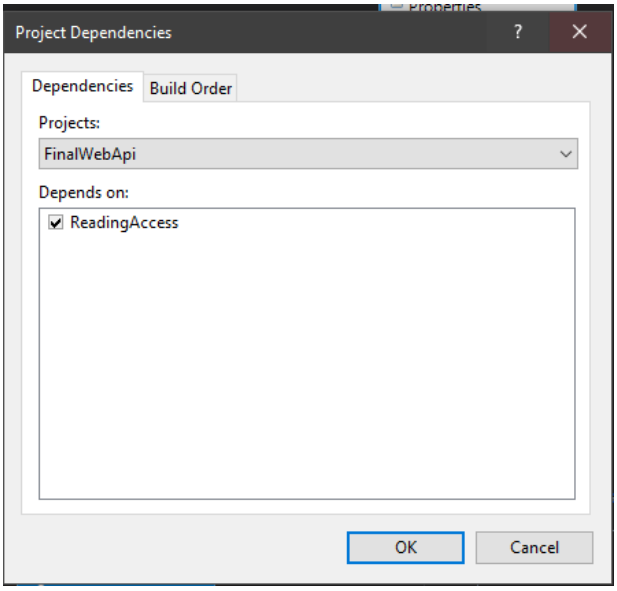


Figure 24: Projects dependencies

The implementation of the web API is a standard one with a model class and a controller. The model Plant Profile is the one that has the data from the SQL Server imported but it does not contain the DateTime field as we couldn't find a way around to import it, so in consequence, only GET and GET by id requests being available for the users that are trying to consume the web service .

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace FinalWebApi.Models
{
    6 references
    public class PlantProfile
    {

        private DateTime date;
        private float co2;
        private float humidity;
        private float light;
        private float temperature;
        private String watering;



        1 reference
        public void setCo2(float co2) { this.co2 = co2; }

        1 reference
        public void setHumidity(float humidity) { this.humidity = humidity; }

        1 reference
        public void setLight(float Light) { this.light = Light; }

        1 reference
        public void setTemperature(float temperature) { this.temperature = temperature; }

        1 reference
        public void setWatering(String value) { this.watering = watering; }


        0 references
        public float getCo2() { return co2; }

        0 references
        public float getHumidity() {return humidity; }

        0 references
        public float getLight() { return light; }

        0 references
        public float getTemperature() { return temperature; }

        0 references
        public String getWatering() { return watering; }

    }
}
```

Figure 25: PlantProfile Model class

```
espace FinalWebApi.Controllers

1 reference
public class ValuesController : ApiController
{

   List<PlantProfile> plants = new List<PlantProfile>();
    public String result_time = "";
    public String result_co2 = "";
    public String result_humidity = "";
    public String result_light = "";
    public String result_temperature = "";
    public String result_watering = "";

   0 references
   public ValuesController()
    {
        SEPDimensionEntities entities = new SEPDimensionEntities();



        /*
        foreach (D_Time f in entities.D_Time.ToList())
        {
            result_time = result_time + f.ToString();
        }

        */

        foreach (F_CO2 f in entities.F_CO2.ToList())
        {
            result_co2 = result_co2 + f.ToString();
        }

        foreach (F_Humidity f in entities.F_Humidity.ToList())
        {
            result_humidity = result_humidity + f.ToString();
        }

        foreach (F_Light f in entities.F_Light.ToList())
        {
            result_light = result_light + f.ToString();
        }
```

Figure 26: PlantProfile Model class

```
foreach (F_Temperature f in entities.F_Temperature.ToList())
{
    result_temperature = result_temperature + f.ToString();
}

foreach (F_Watering f in entities.F_Watering.ToList())
{
    result_watering = result_watering + f.ToString();
}

int counter = 0;
int x = 0;



var c_co2 = result_co2.Split(',');
var c_humidty = result_humidity.Split(',');
var c_light = result_light.Split(',');
var c_temperature = result_temperature.Split(',');
var c_watering = result_watering.Split(',');

for (int i = 0;i< c_co2.Length;i++)
{
    PlantProfile plant = new PlantProfile();

    int.TryParse(c_co2[i], out x);
    plant.setCo2(x);

    int.TryParse(c_humidty[i], out x);
    plant.setHumidity(x);

    int.TryParse(c_light[i], out x);
    plant.setLight(x);

    int.TryParse(c_temperature[i], out x);
    plant.setTemperature(x);

    plant.setWatering(c_watering[i]);

    plants.Add(plant);

}
```

Figure 25: WebAPI Controller

A special implementation was created as when we retrieve one attribute (CO2,Temperature etc.) we receive one string with all the correspondent values. There is a specific String formatter in the constructor. It's supposed to retrieve all the values split them by a comma and a space string that have the role of separators of the values. After all the values are split the program proceeds to collect each value for each field and creating objects that will end up in a list of Plant Profile objects. When the GET request is performed then the whole list is returned or a specific object is retrieved when the id is known by the web API consumer.
We decided to leave the routing as it is standard.

```
// GET
0 references
public List<PlantProfile> Get()
{

    return plants;

}

// GET api/values/5
0 references
public PlantProfile Get(int id)
{
    return plants[id];
}
```

Figure 26: WEB API requests

2.Client Server Socket Communication:

Communication between Server and Client was handle with socket protocol communication. The server side was implemented in java and the client side was implemented in C#. Server requests the secure password from the client. Client needs to insert valid password to be able to continue communication with server. In the Validate Server Certificate method we are validating token for secure communication.

```
class Client
{
    private TcpClient client;
    private NetworkStream ns;
    private SslStream sl;
    private byte[] data;
    private bool connected = false;
    public BinaryReader read;
    public Client()
    {

        client = new TcpClient("0.tcp.ngrok.io", 15978);
        ns = client.GetStream();
        connected = true;
        var ssl = new X509Certificate(File.ReadAllBytes(System.Environment.GetEnvironmentVariable("USERPROFILE")+ "\\pu

        sl = new SslStream(ns, false, new RemoteCertificateValidationCallback(ValidateServerCertificate), null);
        sl.AuthenticateAsClient("bridge");
        read = new BinaryReader(sl);

    }


    public static bool ValidateServerCertificate(
        object sender,
          X509Certificate certificate,
          X509Chain chain,
          SslPolicyErrors sslPolicyErrors
        )
    {
        return true;
    }
```

Figure 27: Client Server Socket Communication

In the method send message we are writing message to memory stream and sending it to server.

```
public void SendMsg(string msg)
{
    if (connected)
    {
        byte[] response = new byte[1024];
        response = Encoding.UTF8.GetBytes(msg + '\0');
        sl.Write(response, 0, response.Length);

        byte c;
        MemoryStream memoryStream = new MemoryStream();
        while((c = read.ReadByte()) != 0)
        {
            memoryStream.WriteByte(c);
        }
        string console;
        console = Encoding.UTF8.GetString(memoryStream.ToArray());
        Console.WriteLine(console);
    }
}
```

Figure 28: Response

3. Transfer to SQL

For transferring the data from the NoSQL database, MongoDB to the Microsoft SQL Server it was used an external script written in C#.The first part is to create a client that will retrieve the information from the cloud database and adding the objects into a list of “”PlantProfiles”.

```
static void Main(string[] args)
{


    var mongoClient = new MongoClient(@"mongodb+srv://Llamaxy:865feeBA@test-i10mg.mongodb.net/test?retryWrite

    var mongoDatabase = mongoClient.GetDatabase("Project");
    var PlantsCollection = mongoDatabase.GetCollection<PlantProfile>("PlantProfiles");
    var allPlants = PlantsCollection.AsQueryable<PlantProfile>().ToList();

    var plants = new List<PlantProfile>();

    allPlants.ForEach(plant => plants.Add(plant));

    Console.WriteLine(plants[0].CO2);

    SqlConnection SqlConnection;
    SqlCommand SqlCommand;
```

Figure 29: Client for MongoDB

The second step is creating a connection with the SQL server and then inserting the information for data warehousing. It is done through a SQL command which has the connection and the necessary query.

```
try
{

    for (int i = 0; i < plants.Count; i++)
    {


        string sql = "Insert into [SEP].[dbo].[Readings] (PlantID, PlantName,Temperature,Light,CO2,Humidity,Amo
            " values (@PlantID,@PlantName,@Temperature,@Light,@CO2,@Humidity,@AmountOfWater,@HoursSinceWatering
        SqlConnection.Open();
        SqlCommand = new SqlCommand(sql, SqlConnection);
        SqlCommand.Parameters.Add("@PlantID", SqlDbType.Int).Value = plants[i].PlantID;
        SqlCommand.Parameters.Add("@PlantName", SqlDbType.NChar).Value = plants[i].PlantName;
        SqlCommand.Parameters.Add("@Temperature", SqlDbType.Float).Value = plants[i].Temperature;
        SqlCommand.Parameters.Add("@Light", SqlDbType.Float).Value = plants[i].Light;
        SqlCommand.Parameters.Add("@CO2", SqlDbType.Float).Value = plants[i].CO2;
        SqlCommand.Parameters.Add("@Humidity", SqlDbType.Float).Value = plants[i].Humidity;
        SqlCommand.Parameters.Add("@AmountOfWater", SqlDbType.Float).Value = plants[i].AmountOfWater;
        SqlCommand.Parameters.Add("@HoursSinceWatering", SqlDbType.Float).Value = plants[i].HoursSinceWatering;
        SqlCommand.Parameters.Add("@ReadingDate", SqlDbType.DateTime).Value = plants[i].DateTime;
        SqlCommand.ExecuteNonQuery();
```

Figure 30: Insert into SQL Server

4. Data Warehousing

In the ETL process at first we have the source table with the data extracted from the MongoDB.

Results | Messages

	PlantID	PlantName	Temperature	Light	CO2	Humidity	AmountOfWater	HoursSinceWatering	ReadingDate
1	1	Roses	32	1001	430	54	100	2	2019-04-05 23:10:05.000
2	2	Apple Trees	20	999	400	34	90	1	2019-03-06 12:20:10.000
3	3	Black alder	22	900	367	44	70	3	2019-04-06 22:11:14.000
4	4	Aconitum	22	800	431	50	50	1	2019-04-07 23:10:30.000
5	5	African Daisy	33	850	331	20	10	1	2019-04-01 21:44:22.000
6	6	Alyssum	22	888	350	22	89	1	2019-04-08 23:22:22.000
7	7	Anthurium	20	887	377	43	86	7	2019-04-09 03:19:00.000
8	8	Erigeron	28	1001	356	53	120	8	2019-05-01 16:22:42.000

Figure 31: SQL Server Source Table

The next part is to create the staging area where we have our dimensional table and the temporary fact tables for every measurement that we have from the sensors.

```
USE [SEP_Staging]
GO

INSERT INTO [dbo].[F_Temperature_Staging]
           (
           [B_PlantKey],
           [Units_Temperature]
           ,[Date_Value]
           ,[Time_Value])
     Select
     PlantID,
     Temperature,
convert(date, [ReadingDate]) as [Date],
 convert(time, convert(time, [ReadingDate])) as [Time]
  from [SEP].[dbo].Readings


GO
```

Figure 32: Insert into staging fact table

In the picture above we have the SQL query for inserting data in the into the temperature fact table where we insert the business key the measurement and the date and time value which are separated using the convert function. Afterwards we do the lookup for updating the surrogate keys.

```
USE [SEP_Staging]
GO

UPDATE [dbo].[F_Temperature_Staging]
   SET [PlantKey] = (Select PlantKey from Plant_Dimension_Staging
   where Plant_Dimension_Staging.B_PlantKey = F_Temperature_Staging.B_PlantKey)

GO
```

Figure 33: Update surrogate key

There were create two dimensions for date and time with fields that are used for queries or reports such as the day of the week or time of the day. In the picture below it is presented the query for creating the time dimension.

```
DROP TABLE [dbo].[D_Time]
GO
CREATE TABLE [dbo].[D_Time](
[TimeKey] int IDENTITY(1,1) NOT NULL,
[TimeAltKey] [int] NOT NULL,
[Time30] [varchar](8) NOT NULL,
[Hour30] [tinyint] NOT NULL,
[MinuteNumber] [tinyint] NOT NULL,
[SecondNumber] [tinyint] NOT NULL,
[TimeInSecond] [int] NOT NULL,
[HourlyBucket] varchar(15)not null,
[DayTimeBucketGroupKey] int not null,
[DayTimeBucket] varchar(100) not null
CONSTRAINT PK_D_Time PRIMARY KEY (TimeKey)
)
GO
SET ANSI_PADDING OFF
GO
/***** Create Stored procedure In Test_DW and Run SP To Fill Time Dimension with Values****/
SET ANSI_NULLS ON
GO
```

Figure 33: Create time dimension

There was also used a procedure for setting the fields, for example we set the for different time of the day specific names as between “00:00 AM ” and “02:59 AM” is “Late Night”.

```
SELECT @DayTimeBucket =
CASE
WHEN (@TimeAltKey >= 00000 AND @TimeAltKey <= 25959)
THEN 'Late Night (00:00 AM To 02:59 AM)'
WHEN (@TimeAltKey >= 30000 AND @TimeAltKey <= 65959)
THEN 'Early Morning(03:00 AM To 6:59 AM)'
WHEN (@TimeAltKey >= 70000 AND @TimeAltKey <= 85959)
THEN 'AM Peak (7:00 AM To 8:59 AM)'
WHEN (@TimeAltKey >= 90000 AND @TimeAltKey <= 115959)
THEN 'Mid Morning (9:00 AM To 11:59 AM)'
WHEN (@TimeAltKey >= 120000 AND @TimeAltKey <= 135959)
THEN 'Lunch (12:00 PM To 13:59 PM)'
WHEN (@TimeAltKey >= 140000 AND @TimeAltKey <= 155959)
THEN 'Mid Afternoon (14:00 PM To 15:59 PM)'
WHEN (@TimeAltKey >= 50000 AND @TimeAltKey <= 175959)
THEN 'PM Peak (16:00 PM To 17:59 PM)'
WHEN (@TimeAltKey >= 180000 AND @TimeAltKey <= 235959)
THEN 'Evening (18:00 PM To 23:59 PM)'
WHEN (@TimeAltKey >= 240000) THEN 'Previous Day Late Night _
(24:00 PM to '+cast( @Size as varchar(10)) +':00 PM )'
END
```

Figure 34: Procedure for setting moments of the day

For the plant dimension we have the business and surrogate keys as well as the plant name and the “”validFrom” and “”validTo” for handling slow changes such as the incremental load of new readings.

```
USE [SEP_Staging]
GO

INSERT INTO [dbo].[Plant_Dimension_Staging]
           ([B_PlantKey]
           ,[PlantName]
           ,validFrom
           ,validTo)

           Select
           PlantID,
           PlantName,
           '2019-01-01'
           ,'2099-01-01'
           from [SEP].[dbo].[Readings]

GO
```

Figure 34: Insert into dimension staging

After the staging area we insert the data into the final dimensional model. The date and time dimension are the same and the final fact tables contain only the surrogate keys and the unit of measurement populated with the data taken from the staging area. In the image underneath it is presented the insertion of data into the CO2 fact table.

```
USE [SEPDimension]
GO

INSERT INTO [dbo].[F_CO2]
           ([DateKey]
           ,[TimeKey]
           ,[PlantKey]
           ,[Units_CO2])
     Select
     DateKey,
     TimeKey,
     PlantKey,
     Units_CO2
     from SEP_Staging.dbo.F_CO2_staging
GO
```

Figure 35: Insert into final fact table

The information from the data warehouse can be used for reports. The reports were made with PowerBi and they were published on the reporting server.

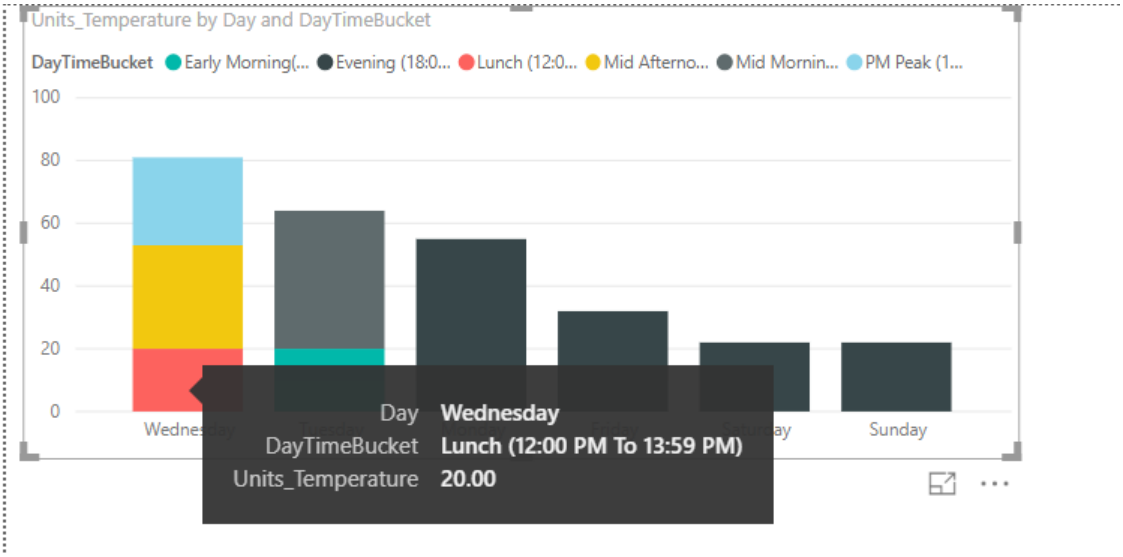


Figure 36:Temperature Power BI report

In the picture above it can be observed the value of the temperature for example throughout different moments of the day also on multiple days, in this approach we can create statistics and the user can see when the temperature raise or drop below a certain threshold and it can take action.

5.4 Interactive Media

Application architecture

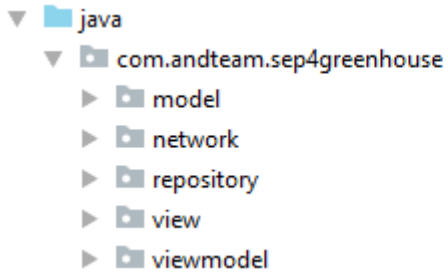
Architecture was a main concern when the application was designed, as it represents the foundation for an app that is well maintainable, scalable and reliable. The aspects taken into account when deciding on an architectural pattern where the separation of concerns, the code reusability, testability and independence. Even though the term "good architecture" may sound slightly abstract, having these concerns in mind, the MVVM architecture was chosen for this project. It maintains a clear separation between application logic and the UI, therefore addressing numerous development issues and making the application easier to test and maintain. The emphasis is put on dividing the responsibilities, so the UI components are separated from the business logic and the business logic is separated from the data access.

There are three core components in the MVVM pattern: the Model, the View and the ViewModel, each serving a distinct purpose. The figure below illustrates the relation between the components:

Figure 37: Data flow in MVVM

The ViewModel isolates the View from the Model, so it receives its data from the Model and exposes the data and command objects that the View requests. The Model is unaware of the ViewModel and the ViewModel is unaware of the View.

In the figure below, how the MVVM pattern was related to the application can be observed:



The Model represents the actual data that is dealt with in the application but it does not hold behaviours or services that manipulate the information. It does not have anything to do with the UI or with fetching any data. Business logic is kept separate from the Model as it belongs to other classes that act on the model.

The View is responsible for the structure and appearance of what the user sees on the screen. Each view is defined in an XML file containing code without any business logic. The View retrieves its data from the ViewModel through the use of binding.

The ViewModel is the component that connects the View to the Model by accessing the methods and properties of the Model that are then made available to the View.

Firebase authentication

Firebase is a platform that allows the development of web and mobile applications without server-side programming and it provides a series of built-in services. Firebase Authentication is very easy and quick to implement, has autoscaling built-in and

provides real-time updates. It automatically stores users' credentials securely by using bcrypt and it separates this sensitive user information from the application's data.
Setting up Firebase is a very intuitive, fast and easy process. Everything was set up by using a google account, creating a new project and adding a new app to it. For the authentication, the email/password sign-in method was enabled.
The main activity opens the app in the login screen where existing users can log in and new users can register. The logic behind the login activity and its connection to Firebase can be observed in the screenshot below.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.login_activity);
    auth = FirebaseAuth.getInstance();
    username = findViewById(R.id.username_login_input);
    password = findViewById(R.id.password_login_input);
    btnRegister = findViewById(R.id.button_register);
    btnRegister.setOnClickListener((v) → { onUserRegisterClick(); });

    //firebase part
    btnRegister = (Button) findViewById(R.id.button_register);
    btnLogin = (Button) findViewById(R.id.button_login);
    btnLogin.setOnClickListener((v) → {
            auth.signInWithEmailAndPassword(username.getText().toString(), password.getText().toString()).addOnCompleteListener((task)
                    if (task.isSuccessful()) {
                        onUserLoginClick();
                    } else {
                        makeToastOnFailiure();
                    }
            });
    });
    btnRegister.setOnClickListener((v) → { onUserRegisterClick(); });
}
private void makeToastOnFailiure() {
    Toast.makeText( context: this,  text: "Operation unsuccessful! Error.", Toast.LENGTH_LONG).show();
}
private void onUserRegisterClick() {
    Intent register = new Intent( packageContext: this, RegisterActivity.class);
    startActivity(register);
}
private void onUserLoginClick() {
    Intent login = new Intent( packageContext: this, MainActivity.class);
    startActivity(login);
}
```

Figure 38: Data flow in MVVM

The layout is set for this activity and the input fields and the buttons are identified. For the Register button and on click listener is attached which runs the method that opens the Register activity.

When the login button is pressed, the user data from the input fields is compared to the Firebase database and if the user was registered, the Main Activity opens up. Otherwise, an error message will be displayed on the screen.

The logic behind the Register activity is presented in the image below. Once the Firebase connection has been established and the layout was set, on the press of the "submit register" button, the user credentials are sent to the Firebase database and the Main activity starts. In the eventuality of a failed registration, an error message is displayed on the screen.

```java
public class RegisterActivity extends AppCompatActivity {

    EditText username;
    EditText password;
    Button buttonSubmitRegister;
    FirebaseAuth auth;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.register_activity);
        auth = FirebaseAuth.getInstance();
        username = findViewById(R.id.email_register_input);
        password = findViewById(R.id.password_register_input);
        buttonSubmitRegister = findViewById(R.id.button_submit_register);
        buttonSubmitRegister.setOnClickListener((v) -> {
                auth.createUserWithEmailAndPassword(username.getText().toString(), password.getText().toString()).addOnCompleteListener((task) -> {
                        if (task.isSuccessful()) {
                            openMainActivity();
                        } else {
                            makeToastOnFailiure();
                        }
                });
        });
    }
    private void makeToast() {
        Toast.makeText( context: this,  text: username.getText().toString() + " " + password.getText().toString(), Toast.LENGTH_LONG).show();
    }

    private void openMainActivity() {
        Intent mainActivity = new Intent( packageContext: this, MainActivity.class);
        startActivity(mainActivity);
    }

    private void makeToastOnFailiure() {
        Toast.makeText( context: this,  text: "Operation unsuccessful! Error.", Toast.LENGTH_LONG).show();
    }
}
```

Figure 39: Register listener

The bottom navigation bar

The bottom navigation bar makes it easy to explore and switch between views. It is convenient to use in cases where there are three to five top-level navigation items of alike importance as it will be omnipresent no matter which view was selected. For the app's bottom navigation bar there are three main navigation items that correspond to Adding a new plant, Viewing the current list of plants and Modifying the user's account settings. The smooth transition between views where the bottom navigation bar stays in place is done using fragments and each view has its own layout resource file and fragment class. The initial (default) fragment that is loaded is View list of plants.

In the onCreate() method in the main activity the BottomNavigationView object was defined as well as the listener for detecting the navigation item selection.

```
public class MainActivity extends AppCompatActivity implements BottomNavigationView.OnNavigationItemSelectedList

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Loading the default fragment
        loadFragment(new ViewPlantsFragment());

        // Getting bottom navigation view and attaching the listener
        BottomNavigationView navigation = findViewById(R.id.navigation);
        navigation.setOnNavigationItemSelectedListener(this);
    }
```

Figure 40: Load fragment

The loadFragment() method deals with switching between fragments and it is called inside the onCreate() method to load the default fragment on start, as seen in the screenshot below.

```
@Override
public boolean onNavigationItemSelected(@NonNull MenuItem item) {
    Fragment fragment = null;

    switch (item.getItemId()) {
        case R.id.navigation_add_plant:
            fragment = new AddPlantFragment();
            break;

        case R.id.navigation_recycler_plants:
            fragment = new ViewPlantsFragment();
            break;

        case R.id.navigation_user_profile:
            fragment = new ModifyAccountFragment();
            break;
    }
    return loadFragment(fragment);
}

private boolean loadFragment(Fragment fragment) {
    //switching fragment
    if (fragment != null) {
        getSupportFragmentManager() FragmentManager
                .beginTransaction() FragmentTransaction
                .replace(R.id.frame_layout, fragment) FragmentTransaction
                .commit();
        return true;
    }
    return false;
}
```

Figure 41: Navigation item selected listener and loadFragment method

Each fragment class for the three main views extends the Fragment class and inflates the respective layout for each fragment.

```
public class AddPlantFragment extends Fragment {

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        //just change the fragment_dashboard
        //with the fragment you want to inflate
        //like if the class is HomeFragment it should have R.layout.home_fragment
        //if it is DashboardFragment it should have R.layout.fragment_dashboard
        return inflater.inflate(R.layout.fragment_addplant,  root: null);
    }
}
```

Figure 42: The Add Plant fragment class

Retrofit HTTP Client Implementation

The Retrofit API serves as a data bridge for the WebAPI and the Android application. JSON object passing is established via HTTP and routed into a designated URI. The Retrofit declaration requires the type of request made to the WebAPI (GET, PUT, POST, DELETE). In the body of the request, the type of object is declared followed by a name declaration for calling this request.

```
// 2
@GET("/plantprofile")
Call<List<PlantProfile>> getAllPlantProfiles();

// 3
@POST("/plantprofile")
Call<Void> addPlantProfile(@Body PlantProfile profile);

// 4
@DELETE("/plantprofile/{id}")
Call<Void> deletePlantProfile(@Path("pId") int id);
```

Figure 43: Retrofit request type & parameters declaration

An example use case is calling the deletePlantProfile (number 4 in figure 43) in the DeletePlantProfile class. A Retrofit object is declared on which the baseUrl method is called. This method takes the target location of the WebAPI to be consumed. Following that, the PlantProfile data object is passed into JSON and the request is made to the WebAPI (figure 44).

```
public void start(PlantProfile plant, VoidCallBack requestCallback) {
    this.callback = requestCallback;

    // Build Retrofit Connection
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build();

    // Reference to RetrofitAPI class
    RetrofitAPI api = retrofit.create(RetrofitAPI.class);
    Call<Void> call = api.modifyPlantProfile(plant);
    call.enqueue( callback: this);
}
```

Figure 44: Retrofit connection build

UI Implementation

The user interface has been implemented using XML - the group has made use of group views to properly place the contents of the layouts in the desired manner.

Through the application, it has been tried to keep a consistency regarding the formatting of the user interface, thus the big majority of the items in the layouts have similar formatting: the buttons, the text and also the input text fields.

In order to clear up and optimize the code, the strings file is made use of, in order to better organize the text used throughout the application.

```
<!--ModifyAccount-->
<string name="edit_credentials">Edit desired credentials</string>
<string name="new_username_hint">Input new username</string>
<string name="new_email_hint">Input new email</string>
<string name="new_name_hint">Input new name</string>
<string name="new_password_hint">Input new password</string>
```

Figure 45: String example from the string file

```
<Button
    android:id="@+id/button_login"
    android:layout_width="96dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="269dp"
    android:layout_marginLeft="269dp"
    android:layout_marginTop="137dp"
    android:layout_marginEnd="46dp"
    android:layout_marginRight="46dp"
    android:layout_marginBottom="1dp"
    android:text="Login"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:background="@drawable/btn"
    android:textColor="@color/white"/>
```

Figure 46: Button styling in the Android Application

```
    android:orientation="vertical"
    android:background="@drawable/backgroundlogin"
     >
    <ScrollView
        android:id="@+id/scrolllogin"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical">
    <ImageView
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_marginTop="30dp"
        android:src="@drawable/plantlogo"
        android:layout_gravity="center_horizontal"/>
    <TextView
        android:id="@+id/text_view_id"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Insert username and password"
```

Figure 47: XML implementation of the login layout

6. Testing

Black box testing

The system test checks if the system works correctly from the user's perspective. Does each function do what it promises and are the UI easy and understandable to read? The general goal for the system test is to check whether the use cases work.

Description	Precondition	Expected result	Steps	Results
A user wants to create an account.	The person has installed the app and has an email address.	The user creates a profile which is added to the database, and should automatically get logged in.	1. The user launches the app. 2. The user clicks the "Register" button. 3. The user inputs the email and password. 4. The uses presses the "Submit" button.	Success, feature works as expected.
A user wants to login.	The person has previously created an account.	The user logs in and is taken to the main activity.	1. The user inputs the credentials. 2. The user clicks the "Login" button.	Success, feature works as expected.

A user wants to add a new plant to his list.	The user must be logged in.	The user fills in the form and the new plant is added to the list in the main activity.	1. The user clicks the “Add plant” button. 2. The user fills in the form. 3. The user clicks the “Create” button.	Failure
A user wants to edit their account.	The user must be logged in.	The user edits the desired credentials which are saved to the database.	1. The user clicks the “My account” button. 2. The user edits their email and/or password. 3. The user clicks the “Save” button.	Success, feature works as expected.
A user wants to delete their account.	The user must be logged in.	The user data is removed from the database.	1. The user clicks the “My account” button. 2. The user clicks the delete floating button. 3. The account is deleted.	Success, feature works as expected.

The user wants to view the plant's external conditions.	The user must be logged in and must have at least one plant in the list.	The system displays the relevant data.	1. The user clicks the plant name in the main activity. 2. The user is presented with the plant information.	Success, feature works as expected.
The user wants to modify a plant's profile.	The user must be logged in and must have at least one plant in the list.	The new data is saved in the database.	1. The user clicks the edit icon next to the plant's name. 2. The user edit's the desired fields. 3. The user clicks the "Save" button.	Failure
The user wants to delete a plant's profile.	The user must be logged in and must have at least one plant in the list.	The data is removed from the database.	1. The user clicks the delete icon next to the plant's name. 2. The plant profile is deleted.	Failure

Testing of database was performed in two ways: black box and white box testing. Inserting dummy data in MongoDB and later getting them in SQL server prove that communication between this two parts works. Communication between SQL server and Web-Api is also tested by sending GET request and getting data from MongoDB. White Box testing was made on source, staging and final database. Queries on specific parts of each database were executed to prove functionality of each database.

Test ID	Target test section	Test procedure	Comments / Remarks	Event type	HTTP status code	Outcome	System reacts to wrong input?
1	HTTPS - /api/values	Send a GET request from browser		Browser receives a JSON response	200 - OK	✓	✓
2	HTTPS - /api/values/1	Send a GET request from browser	Forwards to navigation help page	Browser receives a response	200 - OK	✓	✓

Table 17: HTTP request testing

Bridge Testing

The bridge application was tested primarily by doing black box testing using a test client that was capable of connecting to the socket connection, designed for the webservice, to test that sent strings resulted in the correct responses.To observe the data received from the embedded system and compare it to print outs of the sensor in the embedded systems own terminal, printouts in the WebSocketListener have been used.

The database connection was tested with unit tests, documented below.

Test Case	Result
Connect to Database	Passed
Add account	Passed
Get account	Passed
Modify account	Passed
Remove account	Passed
Set plant profile	Passed

Get plant profile	Passed
Remove plant profile	Passed

Table 18: Android Black Box Testing

7 Results and Discussion

The “Green House” project was developed to meet the requirements of an Internet of Things project that can retrieve, analyze and visualize sensor data. The retrieving of data is conveyed through the sensors , stored in a database and finally the data is visualized on an app that was developed in Android Studio. Users are able to register and login in the app and see live data sent from the sensors that analyse their plants’ environmental conditions.

However, in terms of functionality, one task that the group did not manage to implement was watering the plants. Furthermore, there are several points that can be improved such making.

The group decided from the very first stage of the implementation that the main focus will the put on the top-priority requirements, the goal being fulfilling those rather than starting the work on the secondary ones, that don’t affect the functionality of the system in a great manner.

The implementation of the system, as expected, turned out to be a complex task, the group having to work on three different “sub-systems”, an Android Application, an embedded system and also a Data Warehousing part.

The group realizes there are mistakes and there are a lot of improvements to be done, but since time was short, the final system was not a perfect one, but the group is pleased with the final result, and is prepared to do an improved job on the next project, with the experience gained from this one.

8 Conclusions

The ultimate goal of every semester project is to facilitate everything learned through the semester, work on the project having the goal to spark the interest on the targeted topics, in this particular project the learning goals being Android Programming, Embedded Programming and Data Warehousing. Apart from applying concepts learned during the courses, the group is also motivated to research the subjects more in-depth. The group gained valuable knowledge in all fields, analyzed and designed all the features from scratch, along with also spending a good amount of time on implementing and testing these features. Ultimately, the group is pleased with the final result with the project, once again, feeling they gained valuable insight about different frameworks, technologies and programming languages.

9 Project future

This project is a simplified result of the original idea. The system only passes information from the sensors, but in the future the system should be able to give orders from the android app (like watering plants which have not been implemented).
Another aspect to have in mind is the dimension of the system. Since it is a scalable system it will be easier to expand.
The final idea will be to have an app which will be able to get the information of an specific, closed and small place, as an incubator where the plant will be in. From this information, each plant will have adapted the conditions to their necessities.

10 Sources of information

(2018, June 21). Retrieved from Tobebright:

https://tobebright.com/the-importance-of-indoor-plants-in-an-apartment/

Tonelli, L., & Kelsey, K. (2018, August 21). Retrieved from Elledecor:

https://www.elledecor.com/life-culture/fun-at-home/news/g3284/best-indoor-plants-for-apartments/

Emma Loewe (December 14, 2018). Retrieved from mindbodygreen:

https://www.mindbodygreen.com/articles/how-to-make-sure-your-plants-dont-die-when-youre-away-for-holidays

FreeRTOS Official documentation - FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf

FreeRTOS Official documentation
-Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

11 Authors

Josipa Babic - Analysis, Requirements, Use Case Diagram and Use Case Descriptions, Data Design, Client Server Socket Communication

Eduard Nicolae Costea - Analysis, Requirements, Use Case Diagram and Use Case Descriptions, Data Design, Web-Service

Diyar Hussein Hussein - Analysis, Requirements, Use Case Diagram, IOT design and Use Case Descriptions

Remedios Pastor Molines - Analysis, Requirements, Use Case Diagram and Use Case Descriptions, Project Future

Kenneth Ulrik Petersen - Analysis, Requirements, Use Case Diagram and Use Case Descriptions

Angel Iliyanov Petrov - Android Application, Analysis, Retrofit HTTP Client Implementation, Requirements, Use Case Diagram and Use Case Descriptions

Ionel-Cristinel Putinica - Abstract, Introduction, Android Application, UI Design and Navigation, Discussion, Result, Conclusion, Analysis, Requirements, Black Box Testing, Use Case Diagram and Use Case Descriptions, UI Implementation

Erika Monica Szasz - Abstract, Bottom navigation bar, Android Application. Firebase Authentication, Discussion, Result, Analysis, Requirements, Application Architecture, Black Box Testing, Use Case Diagram and Use Case Descriptions

Christian Schou Sørensen - Analysis, Requirements, Use Case Diagram and Use Case Descriptions, IoT - Embedded System.

Mihai Tirtara - Analysis, Requirements, Use Case Diagram and Use Case Descriptions, Data Design, SQL Server, Transfer To SQL, Data WareHousing

12 Appendices

UserGuide.pdf

Architecture.pdf

Group_Contract.pdf

Analysis/SSD_Plant.pdf

ProjectDescription.pdf

AndroidLayouts/3main.png

AndroidLayouts/1login.png

AndroidLayouts/2register.png

API_Requests/API_Requests.txt

Reports(PowerBI)/Reports.pbix

SequenceDiagram_GetStatus.pdf

Analysis/DomainModel_Plants.pdf

WebAPIDiagram/WebApi-Class.asta

AndroidLayouts/5modifyAccount.png

ER_Diagram/ER_source_diagram.asta

Statemachine Diagram [12.05].asta

ER_Diagram/ER_Staging_diagram.asta

Analysis/UseCaseDiagram_Plants.asta

AndroidLayouts/4accountSettings.png

Bridge_ClassDiagram_Simplified.asta

AndroidLayouts/7ViewPlantProfile.png

ER_Diagram/ER_dimension_diagram.asta

AndroidLayouts/interactiondiagram.png

AndroidLayouts/6createPlantProfile.png

AndroidLayouts/8modifyPlantProfile.png

PlantClassDiagram/PlantClassDiagram.png

ArchitectureDiagramAndroid/[1.08]MVVM_AND.asta

SystemSequenceDiagram/SystemSequenceDiagram.png

PlantClassDiagram/[1.07]Plant_Class_Diagram.asta

SystemSequenceDiagram/[1.00]Sequence_Diagram.asta

Analysis/ActivityDiagrams/ModifyPlantProfileAD.asta

SystemSequenceDiagram/getPlant.SequenceDiagram.asta

BridgeApplicationDiagram/[1.03]BridgeApplication.asta

Analysis/ActivityDiagrams/AccessUserDataActivityD.asta

Analysis/ActivityDiagrams/Access_Plant_Profile_AD.asta

BridgeWebserviceProtocol/bridge-webservice-protocol.txt

Analysis/SequenceDiagrams/SequenceDiagram(ViewPlantLog).asta

Analysis/ActivityDiagrams/Modify_User_Profile_Diagram_AD.asta

Analysis/SequenceDiagrams/Control_wateringSEPsequenceDiagram.asta

Analysis/ActivityDiagrams/Control WateringActivity_DiagramSEP.asta