VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

INTERNET OF THINGS APPLICATION DEVELOPMENT

PROJECT REPORT

SMART SCHEDULING IRRIGATION SYSTEM

INSTRUCTOR : **LÊ TRỌNG NHÂN**

STUDENTS : **ĐẶNG CÔNG KHANH - 2053105**
: **NGÔ VĂN TIÊN - 2053489**
: **BÙI VÕ CÔNG PHU - 2053326**

HO CHI MINH CITY, JUNE 2024

Introduction

This report details the development and implementation of an Internet of Things (IoT) project focused on creating a Smart Scheduling Irrigation System. The primary goal of this project is to optimize water usage and facilitate efficient irrigation through the integration of various technological components. The system is designed to include three mixers, an input pump, an area selector, and an output pump, which operate in a cyclic manner to ensure precise and controlled irrigation.

A mobile application was developed to serve as the user interface for the system, encompassing both front-end and back-end functionalities. This application allows users to monitor and control the irrigation process remotely, providing real-time updates and adjustments. The cloud server for this project is hosted on Adafruit, chosen for its powerful IoT support and reliability. For data management, a NoSQL database was employed to handle the unstructured data generated by the sensors and devices within the system.

Python programming was utilized to interface with the Raspberry Pi, which acts as the IoT gateway. The Raspberry Pi plays a crucial role in bridging the gap between the mobile application and the physical irrigation components, processing commands and transmitting data to and from the cloud server. This setup ensures that the irrigation system is responsive and adaptable to varying conditions and user inputs.

The integration of these technologies not only automates the irrigation pro-

cess but also enhances its efficiency and effectiveness. The system is capable of adjusting watering schedules and amounts based on real-time data, promoting sustainable agricultural practices. The mobile application provides a convenient and spontaneous platform for users to interact with the system, making it accessible and easy-to-use.

In conclusion, this IoT-based Smart Scheduling Irrigation System leverages modern technology to deliver a good solution for agricultural irrigation. The combination of mobile app development, cloud computing, and IoT hardware results in a comprehensive system that promises to revolutionize traditional irrigation methods.

Contents

List of Tables

List of Figures

1

Requirement Analysis

The first step in any project is to analyze the requirements to understand the goal we aim to achieve. In this chapter, we will discuss both functional and non-functional requirements, and present a use-case diagram of the entire system as well as a use-case diagram for a smaller component which is Customize Watering Schedule.

Contents

1.1 Functional requirements

These are the primal functional requirements of the user:

- **Observe real-time value of sensors**: the user can monitor the real-time values from temperature, humidity and light sensors.
- **Customize watering schedules**: the user can create and modify watering schedules. They can customize up to three watering schedules on their preferences with number of repetitions, area of watering, time needed for each mixer and start and end times.
- **Activate and Deactivate watering schedules**: the user can toggle a specific watering schedule.
- **Observe progress of watering schedules**: the user can monitor the ongoing status and progress of the watering schedules.
- **Change the Adafruit server key**: the user can update the security credentials for the Adafruit server.
- **Observe statistics**: the user can view statistical data related to the sensor readings and system performance, such as: average time to send and receive data, number of successful data uploads, and statistical table of humidity and temperature.

1.2 Non-functional requirements

These are the primal non-functional requirements of the system:

- **Real-time data processing**: the system should process and display real-time sensor data every 7 seconds.
- **Schedule activation/deactivation**: changes to watering schedules (activation or deactivation) should be reflected within 5 seconds, else the application will send a notification saying the request is failed and terminated.
- **User Interface**: the system's user interface should be intuitive and easy to navigate, allowing users to perform tasks with minimal effort and training.
- **Integration with Adafruit server**: the system should seamlessly integrate with the Adafruit server for data storage and retrieval.

1.3 Use-case diagram

1.3.1 Use-case diagram for the whole system

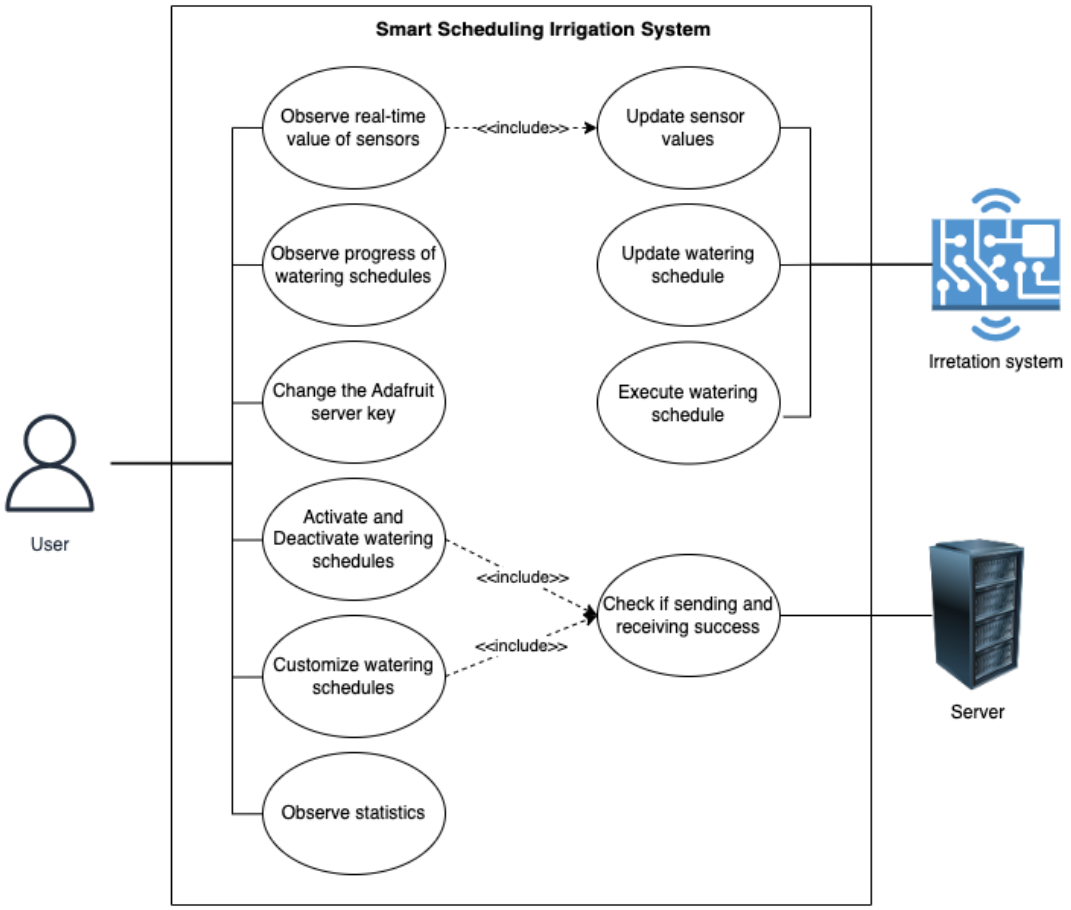

Figure 1.1: Use-case diagram of the whole system

The use-case diagram for the Smart Scheduling Irrigation System provides an overview of the user interactions with the system. Users can observe real-time sensor values, monitor the progress of watering schedules, change the Adafruit server key, activate or deactivate schedules, customize schedules, and observe

statistics. The system processes these interactions by updating sensor values, updating and executing watering schedules, and checking the success of sending and receiving data. The server plays a crucial role in ensuring data integrity and communication between the mobile application and the irrigation system. This comprehensive diagram highlights the functionalities and the flow of data within the system, ensuring efficient irrigation management.

1.3.2 Use-case diagram for Customize Watering Schedule

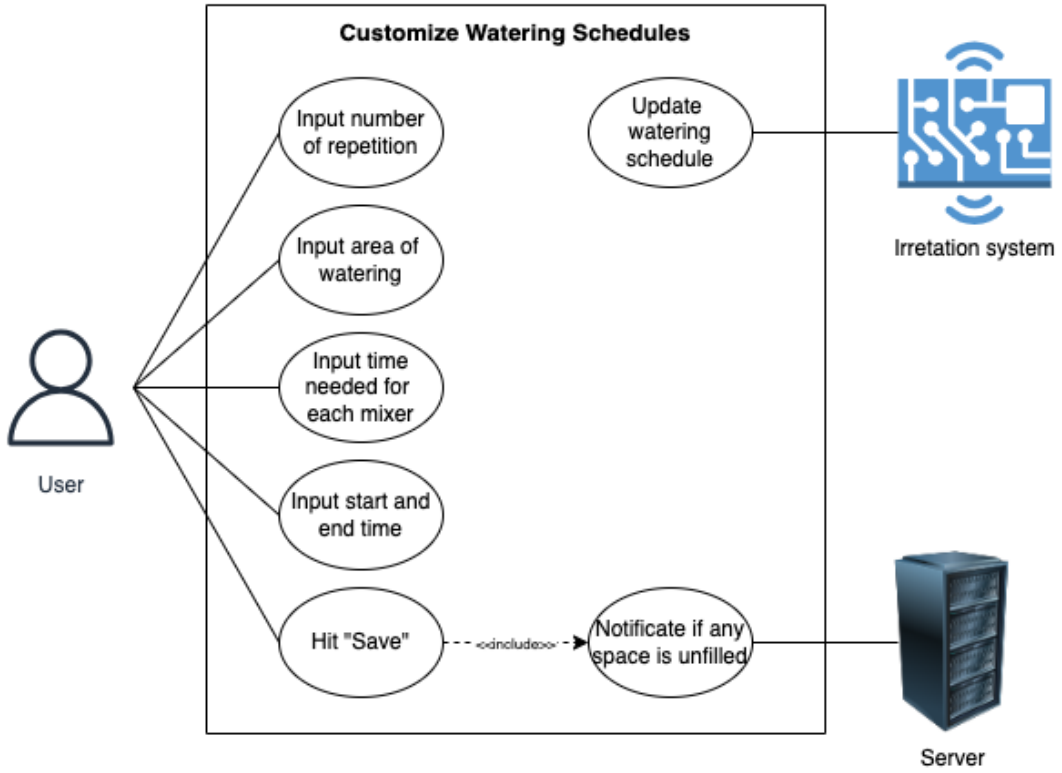

Figure 1.2: Use-case diagram of Customize Watering Schedule

The use-case diagram for customizing watering schedules illustrates the interactions between the user, the system, and the server. The user begins by inputting various parameters: the number of repetitions, the area to be watered, the time needed for each mixer, and the start and end times. After filling in these details, the user hits the "Save" button. If any required fields are left unfilled, the system will notify the user. Once the data is complete and submitted, the server processes the information, updates the watering schedules, and ensures that the irrigation system follows the user's customized settings. This process streamlines the management of irrigation schedules efficiently.

Name	Customize watering schedules
Primary actor	The user
Trigger	When the user want to change the settings of a watering schedule
Precondition	None
Postcondition	The watering schedule is updated
Normal flow	1. The user navigate to the "Watering schedule" on the application 2. The user choose a watering schedule that they want to customize 3. The user select number of repetitions 4. The user select the area of watering 5. The user select the time needed for each mixer 6. The user select start and end time for the watering schedule 7. The user hit the "Save" button
Alternative	Alternative 1, after step 7: The user can go back to step 2 and choose another watering schedule to customize
Exception	Exception 1, after step 7: If the user leave any space unfilled, the application will require the user to fill in the empty space before hitting "Save"

Table 1.1: Table description of Customize Watering Schedules

2

Design

In this chapter, we will show the design of the system by the workflow design (activity diagrams). We also outline the architecture design for the system and database. These designs provide a clear blueprint for implementation, ensuring efficient and effective system development.

Contents

2.1 Workflow design

In this section, our team want to utilize the power of activity diagram to represent the workflow design of the system, we will show the activity diagram for Observe real-time value of sensors and Customize watering schedules to show the role of the actors inside the system.

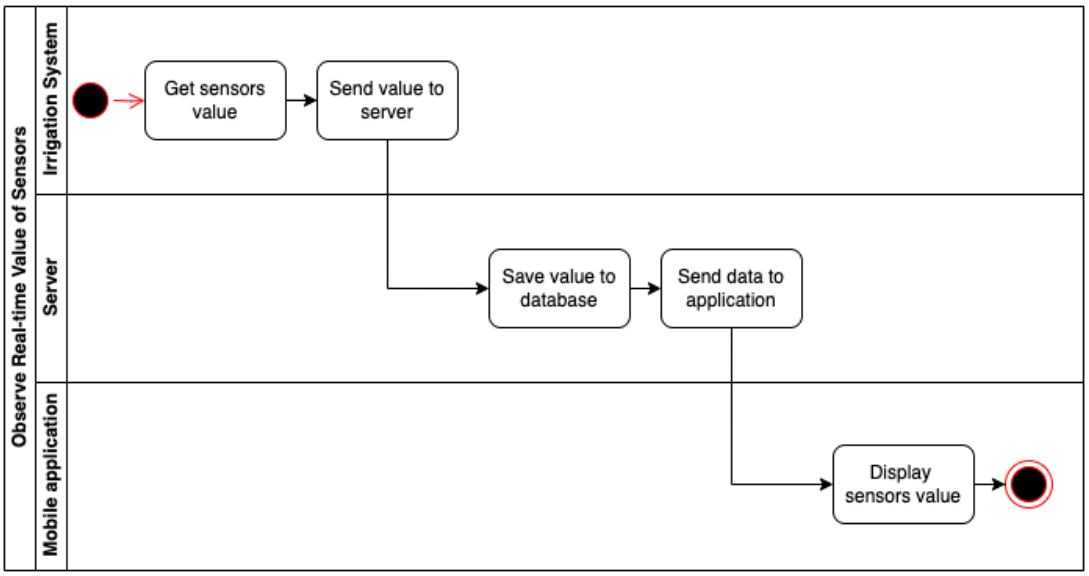

Figure 2.1: Activity diagram of Observe Real-time Value of Sensors

The figure illustrates the process of observing real-time sensor values within an irrigation system. The process begins with the irrigation system obtaining sensor values, which are then transmitted to a server. The server receives these values and stores them in a database. Following this, the server sends the data to a mobile application. The mobile application receives the data and displays the sensor values to the user. This diagram effectively shows the interaction between the ir-

rigation system, server, and mobile application, highlighting the sequential steps involved in real-time data monitoring and display for sensors.

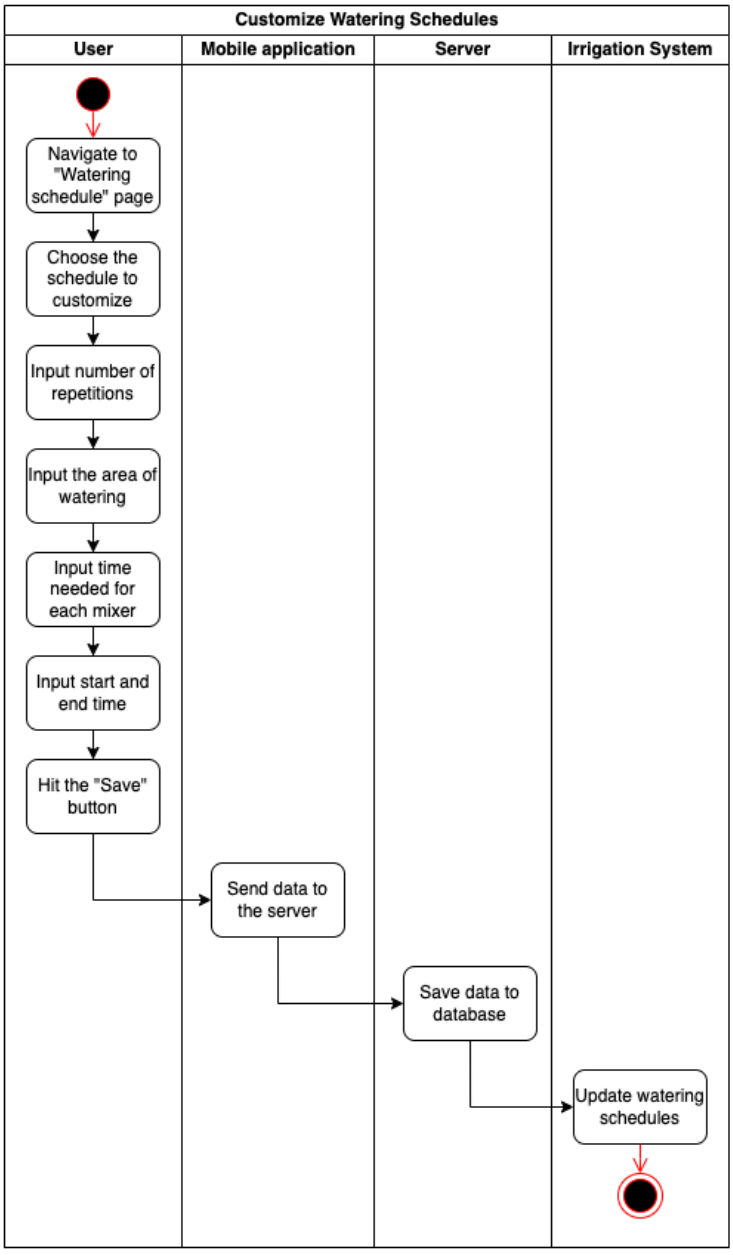

Figure 2.2: Activity diagram of Customize Watering Schedules

This figure illustrates the process of customizing watering schedules in the Smart Scheduling Irrigation System. The user navigates to the "Watering schedule" page on the mobile application, selects the schedule to customize, and inputs the number of repetitions, area of watering, time needed for each mixer, and start and end times. After hitting the "Save" button, the application sends the data to the server. The server saves this data to the database and updates the watering schedules accordingly. This process ensures that the user's customized watering preferences are effectively communicated and implemented in the irrigation system.

2.2 Architecture design

2.2.1 System design

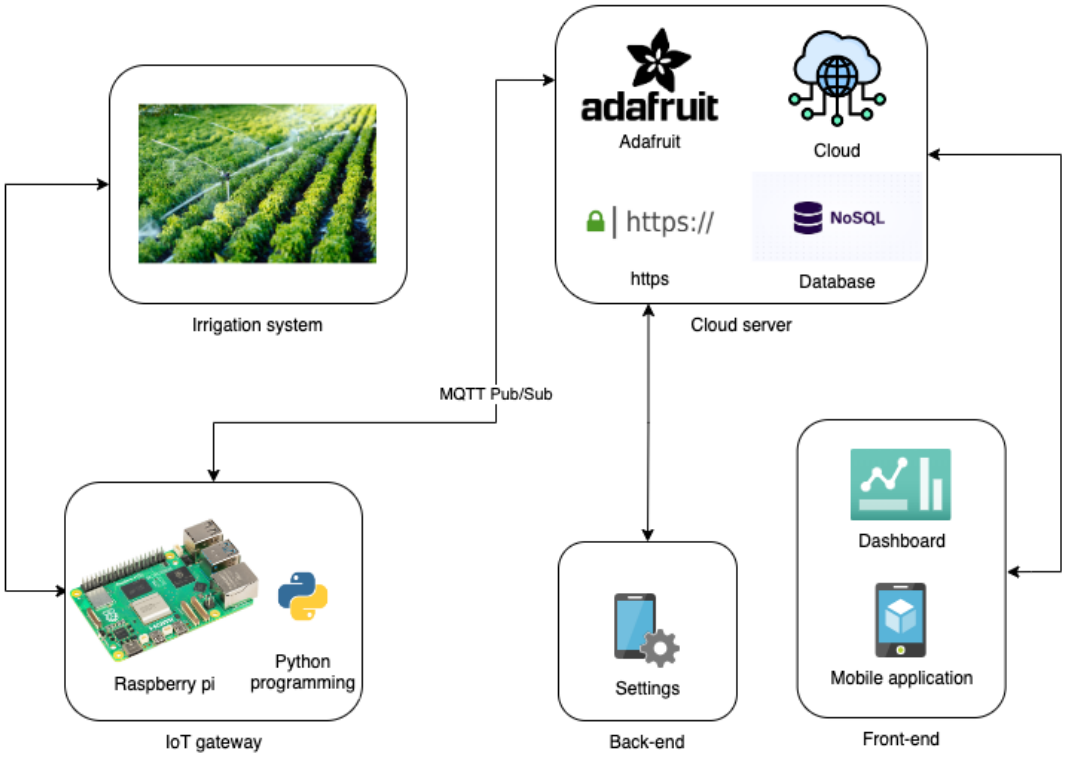

Figure 2.3: The overall design of the whole system

The figure depicts the overall system design architecture for an irrigation system. The design integrates multiple components: an irrigation system, a Raspberry Pi serving as an IoT gateway, and a cloud server powered by Adafruit. The IoT gateway, programmed with Python, facilitates communication between the irrigation system and the cloud server using the MQTT Pub/Sub protocol. Data from the irrigation system is sent to the Adafruit cloud, where it is stored in a

NoSQL database. The back-end server manages settings and communicates with the front-end, which includes a dashboard and a mobile application for real-time monitoring and management. This architecture ensures efficient data flow and remote accessibility.

2.2.2 Database design

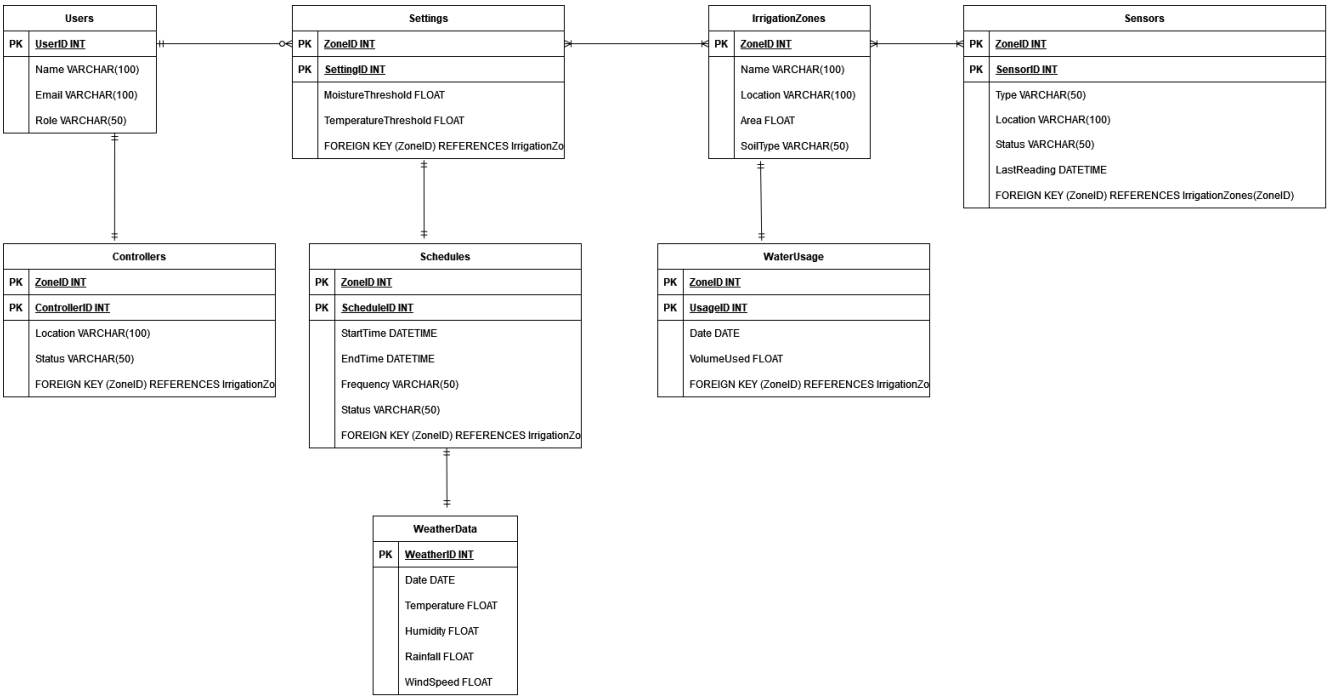

Figure 2.4: Database design of the whole system

The entity-relationship diagram represents the database schema for an irrigation management system. The schema comprises several tables:

- **Users**: This table stores user information, including user ID, name, email, and role. Settings: This table holds various settings for each irrigation zone,

such as moisture and temperature thresholds.

- **IrrigationZones**: This table contains information about different zones, including their names, locations, areas, and soil types.
- **Sensors**: This table records sensor data, including sensor ID, type, location, status, and the last reading timestamp.
- **Controllers**: This table maintains details about controllers assigned to each zone, capturing location and status.
- **Schedules**: This table schedules irrigation activities, specifying start and end times, frequency, and status for each zone.
- **WaterUsage**: This table logs water usage per zone, including usage date and volume used.
- **WeatherData**: This table stores weather-related data such as temperature, humidity, rainfall, and wind speed, correlating with zones.

3

Development

In this third chapter, we will detail the implementation of the system, divided into key components: the mobile application (including both front-end and back-end), the database, the Adafruit cloud server, and the IoT gateway on the Raspberry Pi.

Contents

3.1 Mobile application

3.1.1 Front-end

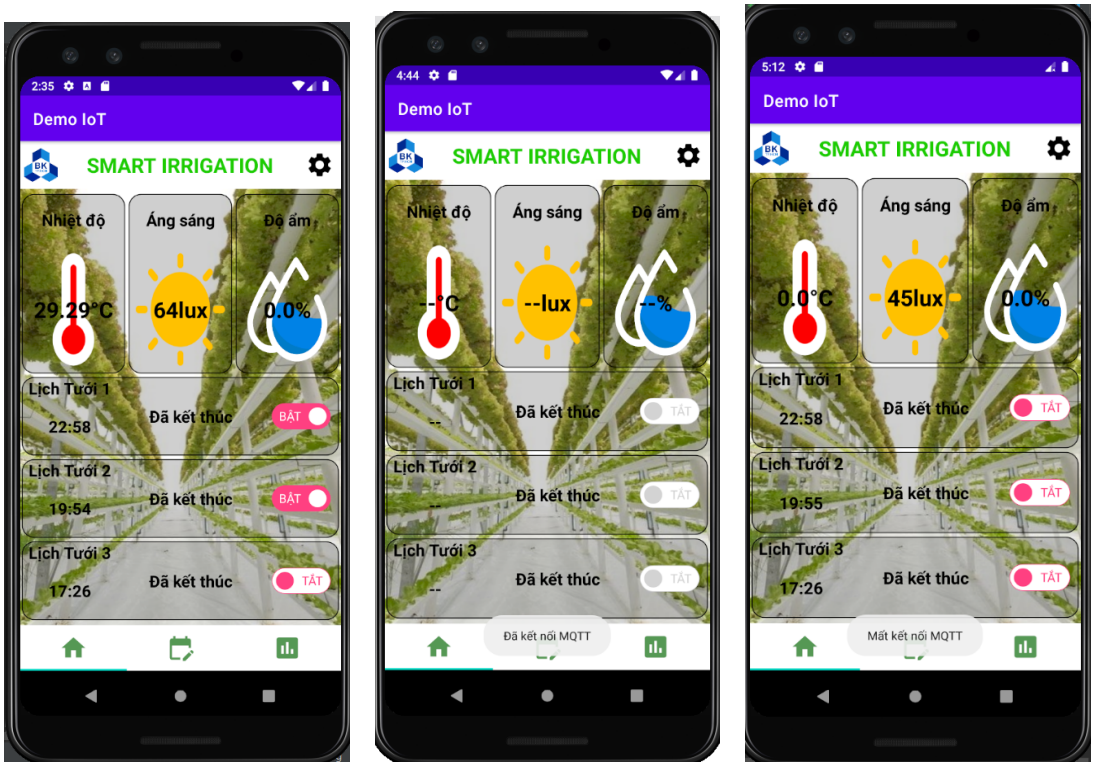

Figure 3.1: Home layout

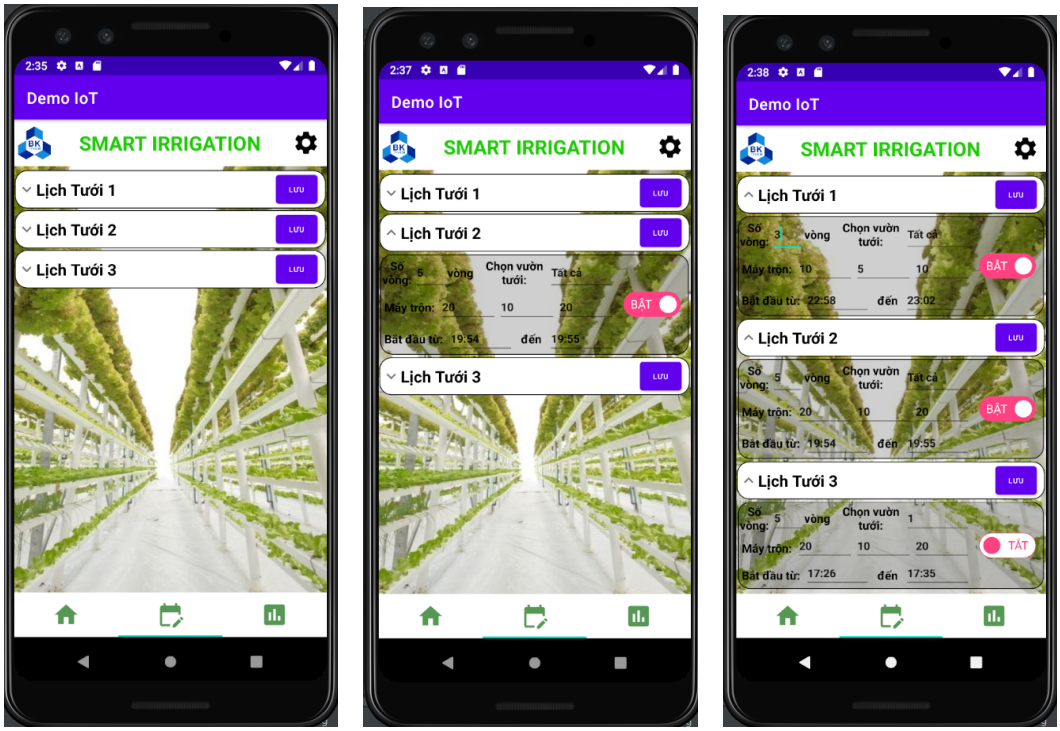

Figure 3.2: Irrigation schedule layout

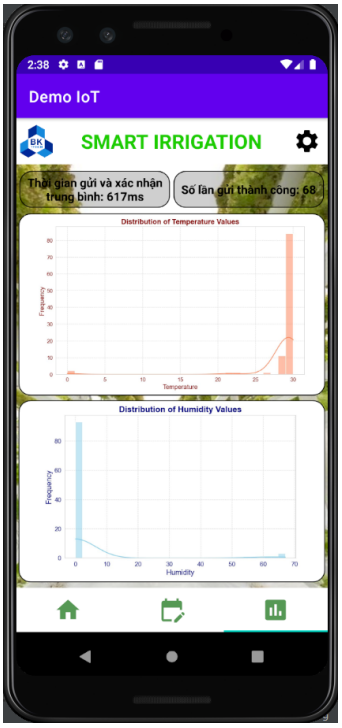

Figure 3.3: Statistics layout

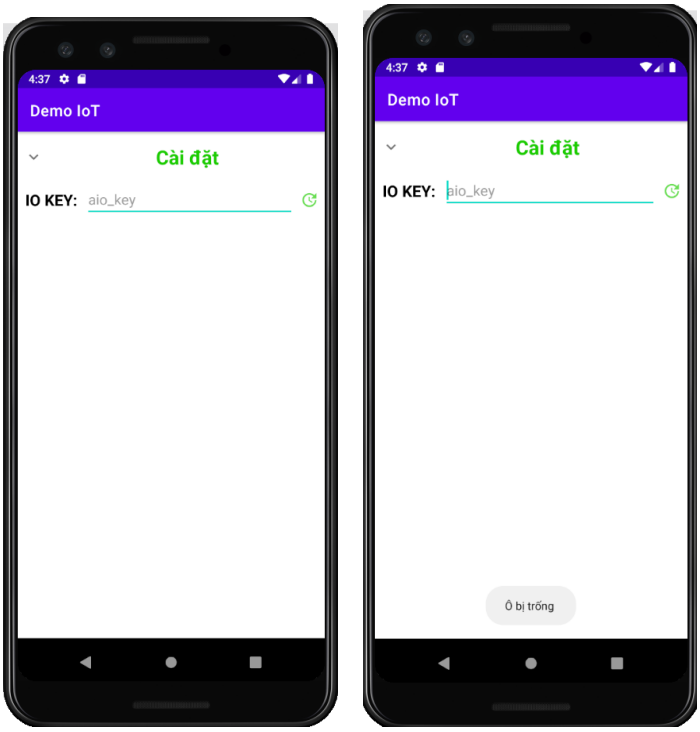

Figure 3.4: Settings layout

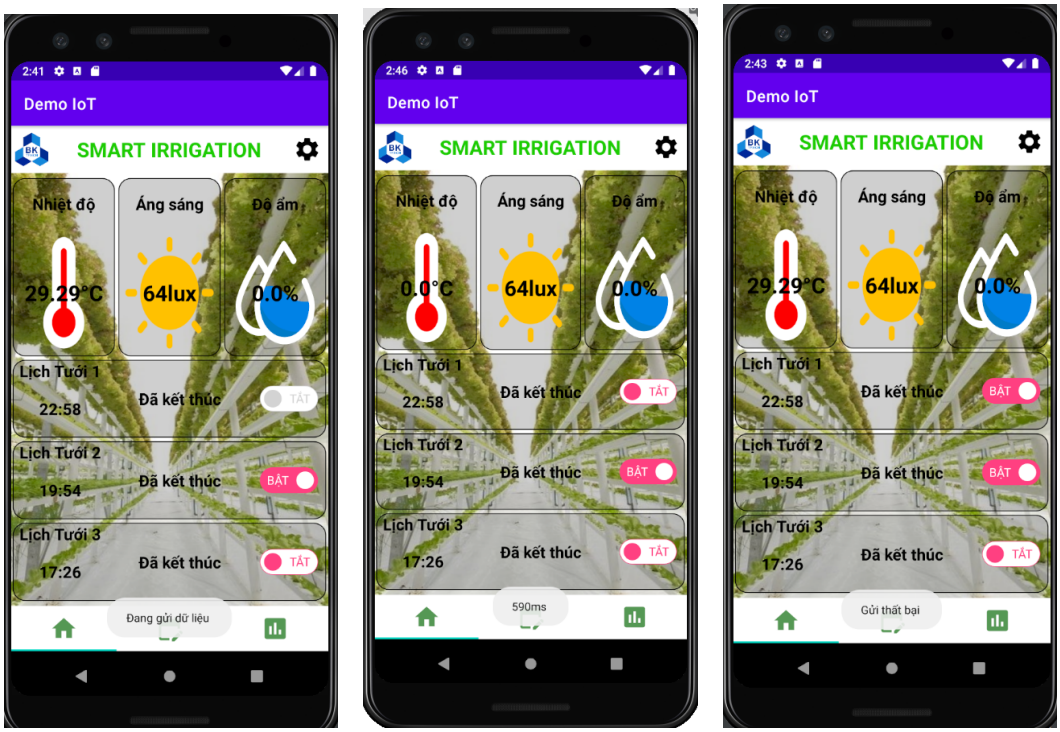

Figure 3.5: Activate/deactivate schedule

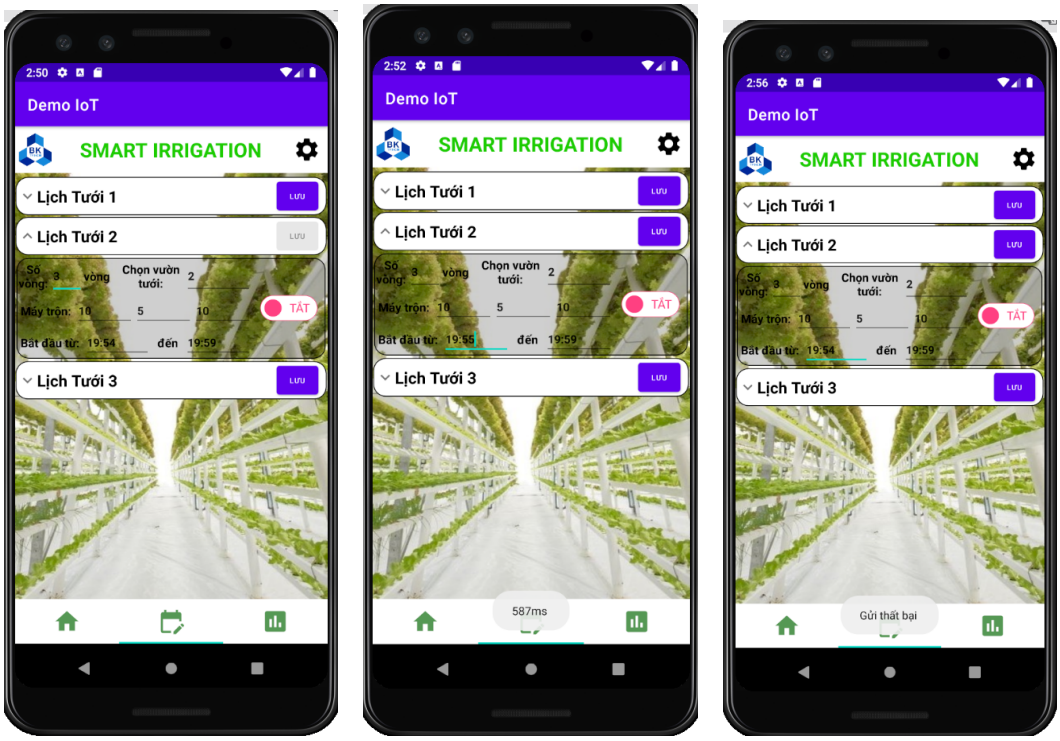

Figure 3.6: Customize schedule

3.1.2 Back-end

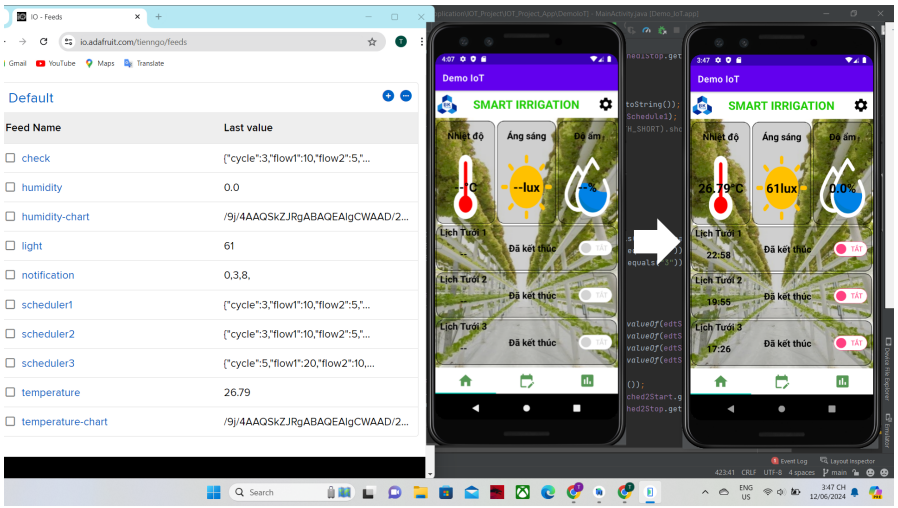

Figure 3.7: Start mobile app

When starting app, the app will synchronize the data(sensors, schedules, chart) with the Adafruit Server by using and calling API from each feeds (HTTP request).

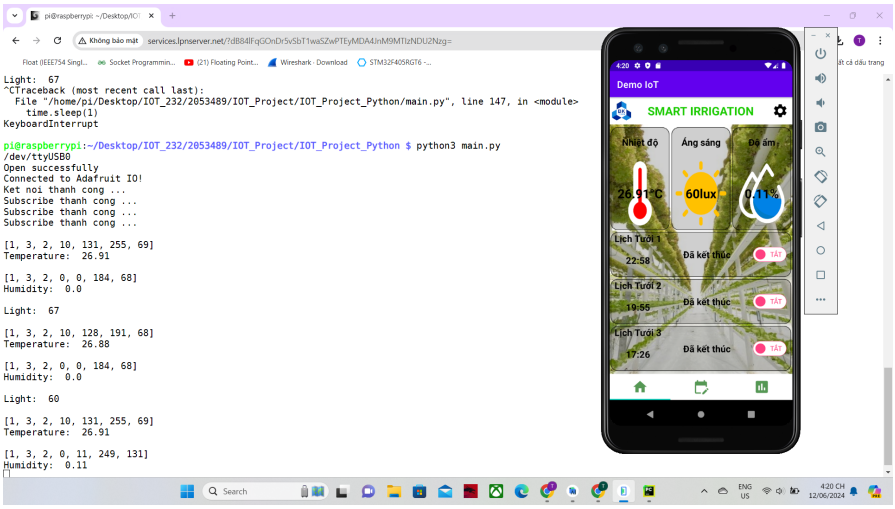

Figure 3.8: Sensor value from Raspberry pi via RS485

After getting the sensor values via RS485, Raspberry pi will send it to adafruit and the app use it to display for user.

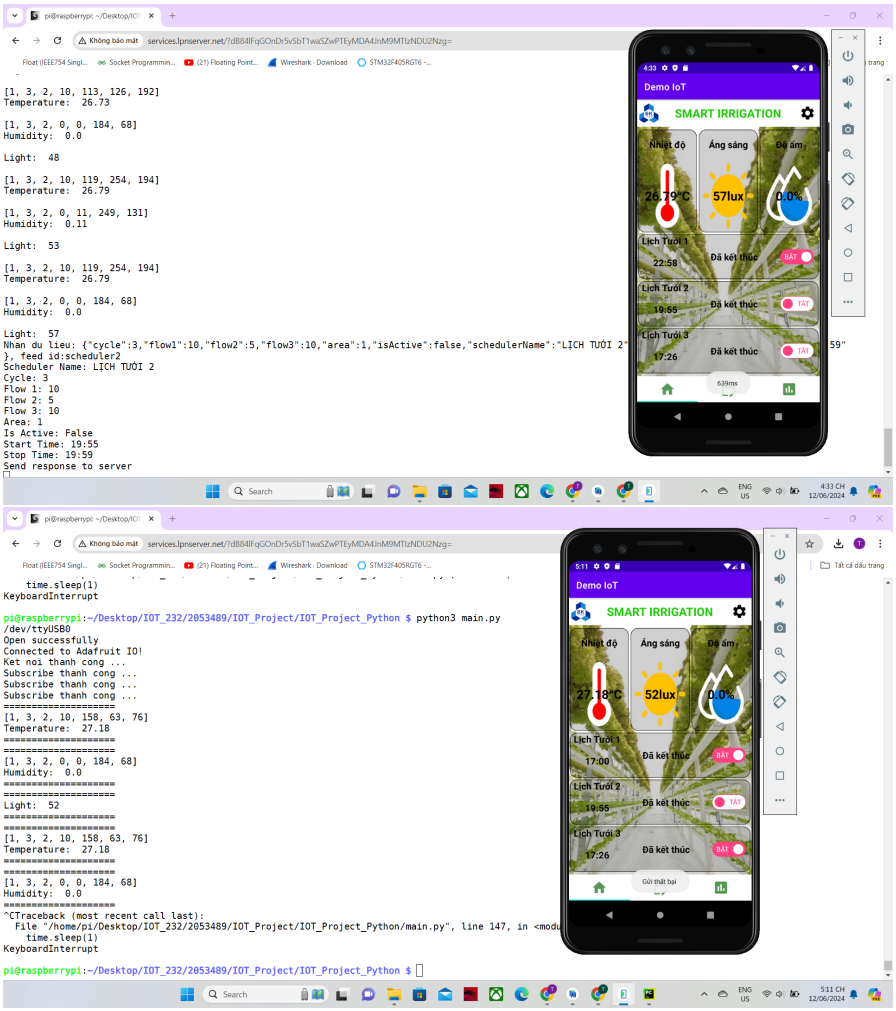

Figure 3.9: Active/deactive schedule

When user active or deactive an schedule, the app will prepare an new schedule (json format) with "isActive" variable changed and send it to server. When the Raspberry pi receive successfully, it will send to "check" feed a response for app to confirm the success. The timing of this whole process will be displayed and saved for statistic in app. In another case, after 5 seconds, the app do not receive exact the response, it notify an error and return the active button to its old position.

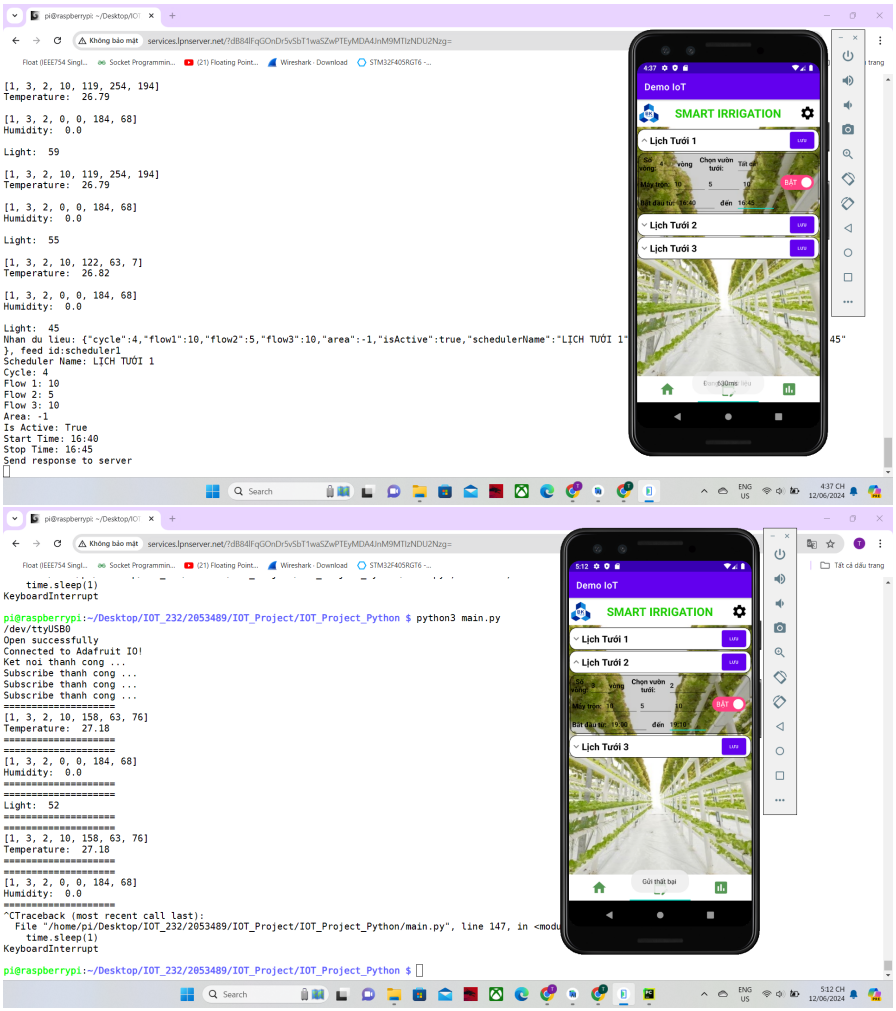

Figure 3.10: Send new schedule to Raspberry pi

When user customize a new schedule and click save button, this also send a new schedule(json format) with input from user to Adafruit server and Raspberry pi. Checking the sending successfull is also the same with when Active/deactive a schedule.

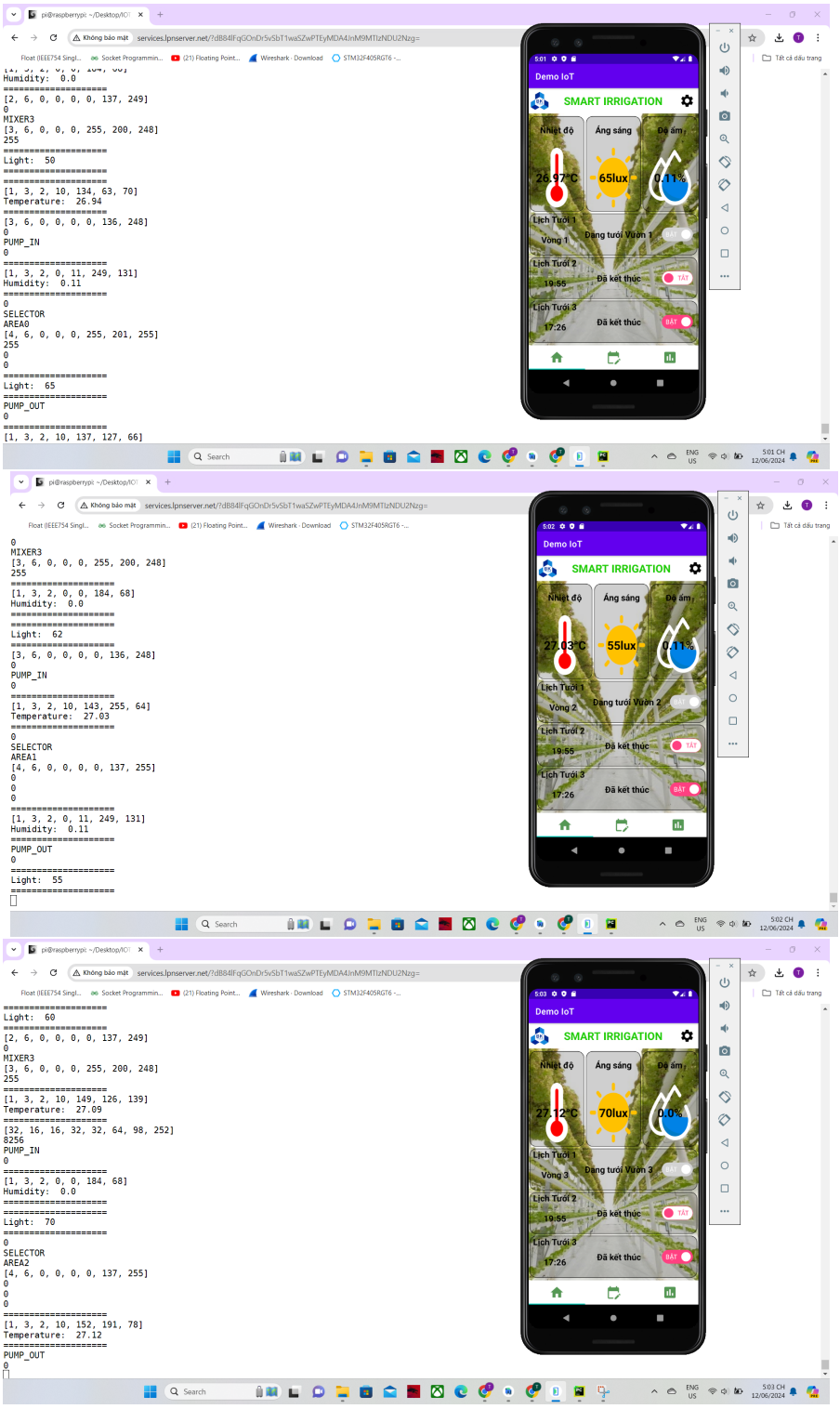

Figure 3.11: Process of finite state machine

When it's time for the schedule to execute, IOT gateway will send its information include schedule id, cycle, status of schedule and area for irrigation(optional) to the "notification" feed. The app will use these information to display on the home page. In this case, the irrigation area of schedule is "Tất cả" so it will irrigate each area in turn according to each cycle.

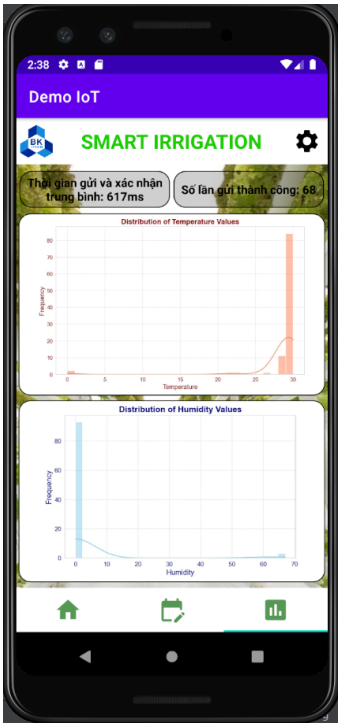

Figure 3.12: Statistics page

For statistic, the app will save the times and timing of all successfull sending information to calculate the average. Besides, Statistics page also haves the chart of temperature and humidity from the database system via the "temperature-chart" and "humidity-chart" feeds.

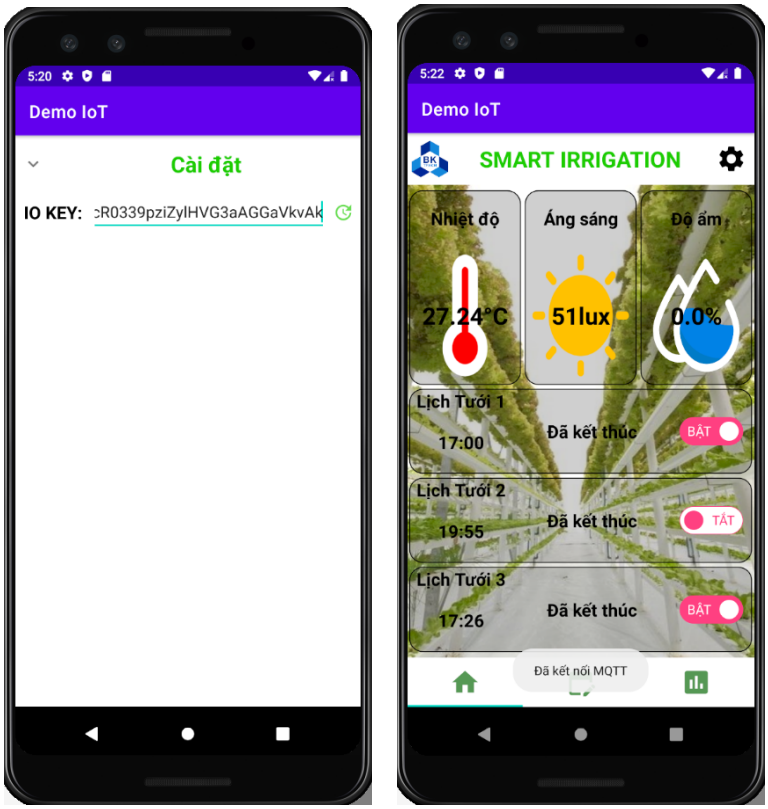

Figure 3.13: Configure Adafruit key

Sometimes the key of Adafruit server can be changed so the app must have an function for user to configure the right key and save this right key to use in the future.

3.2 Database

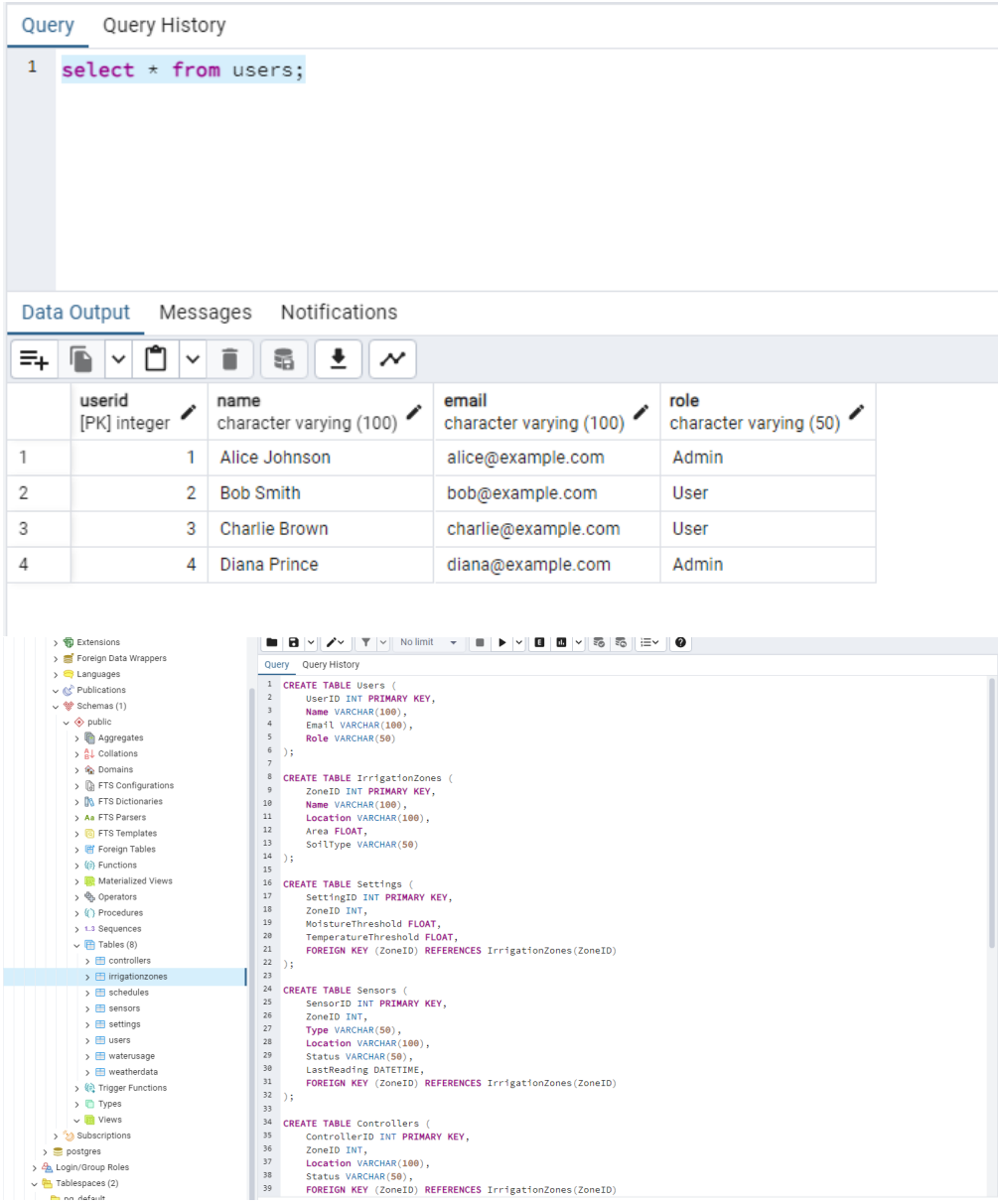

Figure 3.14: Implement Database by Postgres

Here is the SQL code to create the tables for the irrigation system database:

```
1 CREATE TABLE Users (
```

```
    UserID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Role VARCHAR(50)
);

CREATE TABLE IrrigationZones (
    ZoneID INT PRIMARY KEY,
    Name VARCHAR(100),
    Location VARCHAR(100),
    Area FLOAT,
    SoilType VARCHAR(50)
);

CREATE TABLE Settings (
    SettingID INT PRIMARY KEY,
    ZoneID INT,
    MoistureThreshold FLOAT,
    TemperatureThreshold FLOAT,
    FOREIGN KEY (ZoneID) REFERENCES IrrigationZones(ZoneID)
);

CREATE TABLE Sensors (
    SensorID INT PRIMARY KEY,
    ZoneID INT,
    Type VARCHAR(50),
    Location VARCHAR(100),
    Status VARCHAR(50),
    LastReading DATETIME,
    FOREIGN KEY (ZoneID) REFERENCES IrrigationZones(ZoneID)
);

CREATE TABLE Controllers (
    ControllerID INT PRIMARY KEY,
    ZoneID INT,
    Location VARCHAR(100),
    Status VARCHAR(50),
    FOREIGN KEY (ZoneID) REFERENCES IrrigationZones(ZoneID)
);

CREATE TABLE Schedules (
    ScheduleID INT PRIMARY KEY,
    ZoneID INT,
    StartTime DATETIME,
    EndTime DATETIME,
```

```
    Frequency VARCHAR(50),
    Status VARCHAR(50),
    FOREIGN KEY (ZoneID) REFERENCES IrrigationZones(ZoneID)
);

CREATE TABLE WaterUsage (
    UsageID INT PRIMARY KEY,
    ZoneID INT,
    Date DATE,
    VolumeUsed FLOAT,
    FOREIGN KEY (ZoneID) REFERENCES IrrigationZones(ZoneID)
);

CREATE TABLE WeatherData (
    WeatherID INT PRIMARY KEY,
    Date DATE,
    Temperature FLOAT,
    Humidity FLOAT,
    Rainfall FLOAT,
    WindSpeed FLOAT
);
```

Listing 3.1: SQL Code to Create Database Tables

The database schema for the irrigation management system serves as a comprehensive tool for optimizing water usage and ensuring efficient operation. It consists of interconnected tables designed to store and manage various aspects of the irrigation process:

1. **Users**: This table stores user information, including unique identifiers, names, email addresses, and roles (Admin or User). It enables secure access control and user management within the system.

2. **IrrigationZones**: Represents different zones within the irrigation system. Each zone is characterized by a unique ID, name, location, area, and soil type. This information provides context for effective irrigation management tailored to specific zones.

3. **Settings**: Contains specific configuration parameters for each irrigation zone, such as moisture and temperature thresholds. These settings allow the system to adapt irrigation practices based on environmental conditions and plant requirements.

4. **Sensors**: Tracks sensors deployed in each zone to monitor conditions such as moisture levels and temperature. This table records sensor types, locations, statuses, and latest readings, enabling data-driven decision-making for irrigation scheduling.

5. **Controllers**: Manages information about controllers responsible for executing irrigation tasks within each zone. It includes details such as controller locations and statuses, ensuring timely and efficient water delivery.

6. **Schedules**: Defines irrigation schedules for each zone, specifying start and end times, frequency, and schedule status. By adhering to predefined schedules, the system optimizes water usage while maintaining optimal soil moisture levels.

7. **WaterUsage**: Monitors water usage in each irrigation zone by logging the date and volume of water used. This information helps assess the efficiency of the irrigation system and identify areas for improvement.

8. **WeatherData**: Stores weather-related data such as temperature, humidity, rainfall, and wind speed. By integrating weather forecasts and historical data, the system can dynamically adjust irrigation plans to account for changing environmental conditions.

Overall, the database schema facilitates comprehensive management and monitoring of irrigation processes, enabling stakeholders to optimize water resources, conserve energy, and promote sustainable agricultural practices.

3.3 Cloud server (Adafruit)

Figure 3.15: Adafruit logo

Our feed on Adafruit is used for managing and monitoring various feeds related to an irrigation system. The interface lists several feeds, each with a specific function, key, last recorded value, and the time of the last update. For instance, the "light" feed has a last value of 45 and was updated about three hours ago. This cloud server facilitates continuous data transmission and reception, acting as an intermediary between the application and the IoT gateway. This setup enables efficient control and monitoring of the irrigation system by processing data from sensors and executing commands to manage irrigation schedules and environmental conditions. The feeds include parameters like humidity, temperature, light, and scheduler configurations, ensuring comprehensive control over the irrigation processes.

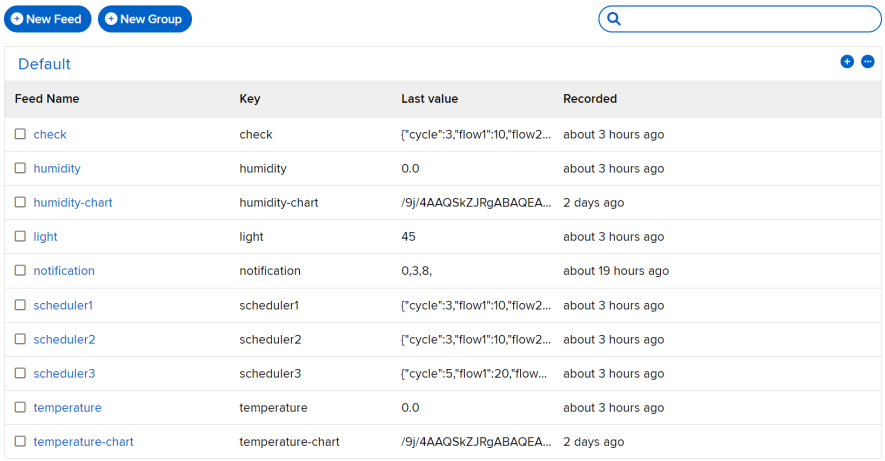

Figure 3.16: Adafruit feeds

These are the feeds that our team used in this project:

- **check**: When the application send a new scheduler to the Raspberry pi, the board sends back a response to the feed "check" to say that the command is received.
- **humidity**: This feed tracks the humidity levels measured by humidity sensor.
- **humidity-chart**: Receiving the chart from database, this feed helps the application to draw the statistic chart of humidity on the Statistic page.
- **light**: This feed tracks the light levels measured by light sensor.
- **notification**: This feed tracks the current progress of the running watering schedule to know that the irrigation system is mixing, or it is pumping in or out.

- **scheduler1, scheduler2, scheduler3**: These feeds contains scheduling information for irrigation cycles. The value is a JSON object detailing cycle and other parameters.
- **temperature**: This feed tracks the temperature levels measured by temperature sensor.
- **temperature-chart**: Receiving the chart from database, this feed helps the application to draw the statistic chart of temperature on the Statistic page.

3.4 IoT Gateway

3.4.1 Raspberry pi

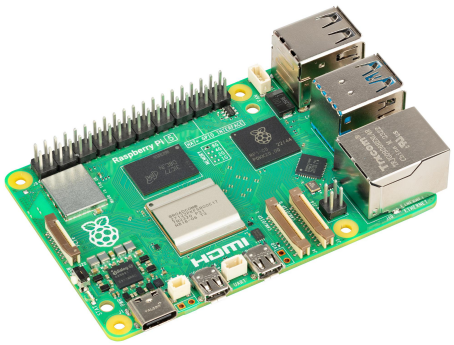

Figure 3.17: Raspberry pi

The Raspberry Pi, functioning as an IoT gateway in this project, plays a critical role in managing the irrigation system. Leveraging Python programming, the Raspberry Pi receives sensor data, which it then relays to a cloud-based application. This enables real-time updates of sensor values on the application, allowing users to monitor environmental conditions such as humidity, temperature, and

light levels effectively.

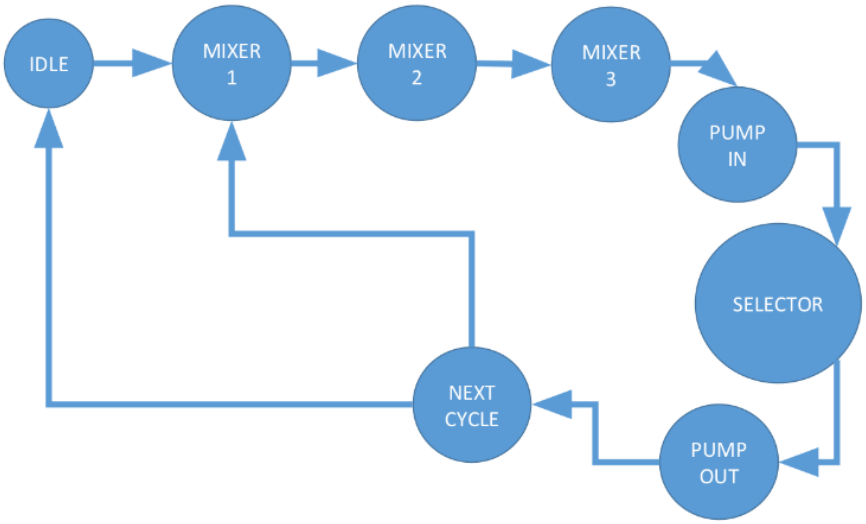

Figure 3.18: Finite state machine design

The Raspberry Pi also handles incoming watering schedules from the application. It transmits these schedules to the irrigation system, ensuring precise activation and deactivation of watering cycles. The finite state machine design illustrated in the above figure outlines the process, where the system transitions through various states like "IDLE," "MIXER 1," "MIXER 2," "MIXER 3," "PUMP IN," "SELECTOR," and "PUMP OUT." These states ensure the system operates correctly, mixing and distributing according to the user-defined schedule.

3.4.2 Python programming

Python programming on the Raspberry Pi facilitates seamless integration and communication between sensors, the cloud application, and the irrigation hardware. Python's versatility and extensive library support enable efficient data pro-

cessing, control logic implementation, and network communication. Consequently, the Raspberry Pi acts as a robust bridge, empowering users to control the irrigation system remotely via a mobile application.

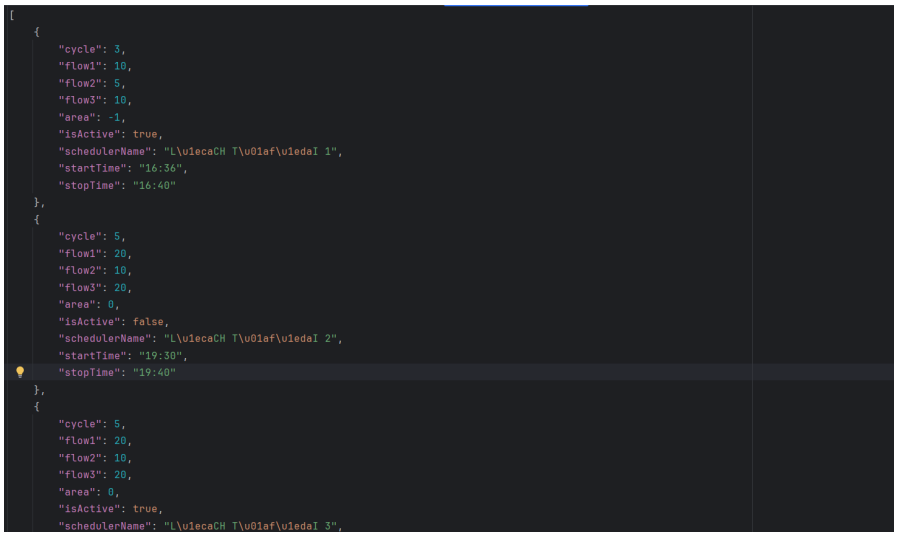

Figure 3.19: Data format of 3 schedules(scheduler_data.json)

```
def message(client, feed_id, payload):
    print("Nhan du lieu: " + payload + ", feed id:" + feed_id)
    if feed_id == "scheduler1":
        write_JSON_file( id: 0, payload)
        scheduler_list[0].set_schedule()
        scheduler_list[0].print_data()
    if feed_id == "scheduler2":
        write_JSON_file( id: 1, payload)
        scheduler_list[1].set_schedule()
        scheduler_list[1].print_data()
    if feed_id == "scheduler3":
        write_JSON_file( id: 2, payload)
        scheduler_list[2].set_schedule()
        scheduler_list[2].print_data()

    client.publish("check", payload)
    print("Send response to server")
```

Figure 3.20: Receive new schedule and respond to app(main.py)

This function updates a schedule by receiving new schedule data from sched-

ule feed, update this data to JSON file, and updating schedule object. After these updates, it sends to "check" feed a response for app to confirm the successful update.

```
def check_scheduler_time():
    if not scheduler_list:
        print("The schedule list is empty.")
        return

    current_time = time.localtime()
    current_hour = current_time.tm_hour
    current_minute = current_time.tm_min
    #Change current_hour to GMT+7
    current_hour = (current_hour + 6) % 24

    for idx in range(len(scheduler_list)):
        start_hour, start_minute = map(int, scheduler_list[idx].startTime.split(':'))
        stop_hour, stop_minute = map(int, scheduler_list[idx].stopTime.split(':'))
        if not is_running_list[idx]:
            if current_hour == start_hour and current_minute == start_minute:
                if scheduler_list[idx].isActive:
                    print(f"Scheduler '{scheduler_list[idx].schedulerName}' is active now.")
                    schedule.every(0).seconds.do(run_schedule_fsm, idx).tag(FSM_TASK_TAG)
                    is_running_list[idx] = True
                else:
                    print(f"Scheduler '{scheduler_list[idx].schedulerName}' is not active.")
                    is_running_list[idx] = False

        if current_hour == stop_hour and current_minute == stop_minute:
            if is_running_list[idx]:
                print(f"Scheduler '{scheduler_list[idx].schedulerName}' is stop now.")
                schedule.clear(FSM_TASK_TAG)
                is_running_list[idx] = False
```

Figure 3.21: check time to run and end FSM(main.py)

This function checks the time to run the FSM by adding it to the schedule (schedule library) and run it every second or end it by deleting it from schedule .

Figure 3.22: read sensor value(rs485.py)

This function to read the value of temperature, humidity sensor via RS485 and random data for virtual light sensor. Then send it to Adafruit feeds.

```
schedule.every(10).seconds.do(check_scheduler_time)
schedule.every(7).seconds.do(read_serial_sensor, client)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Figure 3.23: Run tasks periodically(main.py)

In this project, we use **schedule** library to run tasks periodically. Specifically, "check_scheduler_time" function is added to the scheduler and execute every 10 seconds and "read_serial_sensor" function is 7 seconds. In the super loop is checking every second the pending tasks to run.

4

Testing

In this chapter, we will show the testcases we implemented to unit and system test the system, including testing for functional requirements such as observe real-time value of sensors, customize watering schedules, activated and deactivate watering schedules, observe progress of watering schedules, change the Adafruit server key, and observe statistics. As well as the testcases for nonfunctional requirements like real-time data processing, schedule activation and deactivation, user interface and the integration with Adafruit server.

Contents

4.1 Testing for functional requirements

4.1.1 Observe real-time value of sensors

ID	Name	Test step	Expected result	Status
1. Observe real-time value of sensors				
1.1	Temperature sensor value display	1. User open Main page of the application	1. The real-time temperature sensor value is displayed	Passed
1.2	Humidity sensor value display	1. User open Main page of the application	1. The real-time humidity sensor value is displayed	Passed
1.3	Light sensor value display	1. User open Main page of the application	1. The real-time light sensor value is displayed	Passed

Table 4.1: Table description of testing for Observe real-time value of sensors

4.1.2 Customize watering schedules

ID	Name	Test step	Expected result	Status
2. Customize watering schedules				
2.1	Customize a watering schedule	1. User open the Watering schedule page 2. User customize a watering schedule 3. User hit the "Save" button	1. The new watering schedule is saved in the database and in the irrigation system	Passed
2.2	Catching unfilled blank	1. User left unfilled blank in the form 2. User hit the "Save" button	1. The data is sent to the server 2. The server catch the error and send it to the application 3. The application alert to user	Passed
2.3	Not hitting "Save" button	1. User customize a watering schedule 2. User does not hit the "Save" button, and navigate to another page or kill app	1. The application understand that User has canceled the action and do nothing	Passed

Table 4.2: Table description of testing for Customize watering schedules

4.1.3 Activate and Deactivate watering schedules

ID	Name	Test step	Expected result	Status
3. Activate and Deactivate watering schedules				
3.1	Activate a watering schedule	1. User toggle on a watering schedule	1. The watering schedule is activated	Passed
3.2	Deactivate a watering schedule	1. User toggle off a watering schedule	1. The watering schedule is deactivated	Passed

Table 4.3: Table description of testing for Activate and Deactivate watering schedules

4.1.4 Observe progress of watering schedules

ID	Name	Test step	Expected result	Status
4. Observe progress of watering schedules				
4.1	Observe the progress of a running watering schedule	1. User open Main page of the application	1. The real-time process of the running watering schedule is displayed	Passed

Table 4.4: Table description of testing for Observe progress of watering schedules

4.1.5 Change the Adafruit server key

ID	Name	Test step	Expected result	Status
5. Change the Adafruit server key				
5.1	Change the Adafruit server key	1. User open Settings button of the application 2. User type the new key 3. User hit the "Save" button	1. The application send the new key to the server 2. The server checks if the new key can be connected 3. The server sends confirmation back to the application 4. The application display successful or throw error if the key can not be connected	Passed

Table 4.5: Table description of testing for Change the Adafruit server key

4.1.6 Observe statistics

ID	Name	Test step	Expected result	Status
6. Observe statistics				
6.1	Observe statistics	1. User open Statistics page of the application	1. The server sends data to the application 2. The Average send and receive time, Total successful send and receive time, and the Distribution of temperature and humidity values are displayed in the application	Passed

Table 4.6: Table description of testing for Observe statistics

4.2 Testing for non-functional requirements

4.2.1 Real-time data processing

ID	Name	Description	Expected result	Status
1. Real-time data processing				
1.1	Temperature sensor value sent	The application receive the real-time temperature sensor value	The temperature value is updated every 7 seconds	Passed
1.2	Humidity sensor value sent	The application receive the real-time humidity sensor value	The humidity value is updated every 7 seconds	Passed
1.3	Light sensor value sent	The application receive the real-time light sensor value	The light value is updated every 7 seconds	Passed

Table 4.7: Table description of testing for Real-time data processing

4.2.2 Schedule activation/deactivation

ID	Name	Description	Expected result	Status
2. Schedule activation/deactivation				
2.1	Prevent deactivating a running watering schedule	Toggle off a running watering schedule	The toggle button is disabled, and the watering schedule can not be turned off	Passed
2.2	End-time deactivating a running watering schedule	The end time reached when the watering schedule is still running	The irrigation system is turned off, and the watering schedule is stopped	Passed

Table 4.8: Table description of testing for Schedule activation/deactivation

4.2.3 User interface

ID	Name	Expected result	Status
3. User interface			
3.1	Intuitive and easy to navigate User interface	User interface is friendly and easy to use	Passed
3.2	Application response time	The delay time when the user use the app is unnoticeable	Passed
3.3	Notification	When an action is completed or canceled, it is well announced so the user can understand	Passed

Table 4.9: Table description of testing for User interface

4.2.4 Integration with Adafruit server

ID	Name	Expected result	Status
4. Integration with Adafruit server			
4.1	Data storage and retrieval	The data is transferred from the application to the server and backward smoothly	Passed
4.2	Response time	The average response time when the data is transfered is less than 650ms	Passed

Table 4.10: Table description of testing for Integration with Adafruit server

5

Maintenance

In this chapter, we will talk about the conclusion of the project: what we have achieved, and evaluate our implemented methods. As well as documenting it by writing this report and posting a public repository on GitHub. Finally, we will talk about future plans - how this project can further be improved in the hope that one day some farm can bring our Smart Scheduling Irrigation System to practice.

Contents

5.1 Conclusion

Here are the things that our team achieved by completing this project:

- Developed a Smart Scheduling Irrigation System for agriculture purpose, including a mobile application, a cloud server based on Adafruit, an IoT gateway based on Raspberry pi, and the irrigation system itself.
- A mobile application based on Java, including front-end functionalities such as observing the sensors real-time values, the current progress of the running watering schedule, or the statistics of the application; from customizing watering schedules to activate or deactivate them; as well as changing the key to the Adafruit server inside the application itself. And the back-end functionalities like sending data forward and backward from the application to the server, saving and extracting data from the database, and allowing the user to know the current state of the application by pop up notifications telling them their action has been finished or canceled.
- A cloud server based on Adafruit to communicate with the application by sending data and receiving continuously, it also stands as the intermediate - connecting the application with the IoT gateway to further control the irrigation system in general.
- An IoT gateway based on Raspberry pi using Python programming, this component in the project receive the value of sensors and send them so that the application can update the real-time value of the sensors. It also receive the new watering schedules, sending it to the irrigation system to activate and deactivate them on the right time, letting the irrigation system knows

the number of repetitions, how long does the first to third mixer works, and in general acting as a bridge allowing the user to control the irrigation system on the mobile application.

In conclusion, our team has developed a robust Smart Scheduling Irrigation System, the processing time of the application is unnoticeable, the data transmission from the application to the server and vice versa is smoothly, the user interface is friendly, easy-to-understand and easy-to-use, and the application updates its states continuously so that the user can easily understand what is going on.

5.2 Future plans

While our Smart Scheduling Irrigation System has made significant strides, there are several places for future improvement.

- Firstly, developing the machine learning algorithms to predict optimal watering schedules based on weather forecasts and soil moisture data would greatly increase efficiency. Integrating additional sensors, such as those for nutrient levels and pH, could provide more comprehensive soil health monitoring. Expanding the mobile application to support multiple languages and regional settings would make it more accessible to a global audience.

- Additionally, exploring the use of solar power for the IoT gateway and irrigation system could promote sustainability. Finally, conducting field tests in various agricultural settings will be essential to refine the system and ensure its practical applicability. By continuously innovating and iterating on our design, we hope to bring our Smart Scheduling Irrigation System into

widespread use, benefiting farmers and contributing to sustainable agriculture practices.

Appendix: Source code

Our source code can be found at this GitHub link:

`https://github.com/tienngo02/IOT_Project`