CS145 IOT SuperPot Documentation
GROUP 0xA - PotDogs

ATIENZA, CARMELO ELLEZANDRO (2021-08090) - LAB1
CASTAÑEDA, KRISTINA CASSANDRA (2021-13300) - LAB1
DISPO, VINCENT ANGELO (2021-11482) - LAB4
DUCAY, JAMES DANIEL (2021-09010) - LAB1
GACHO, LORIDGE ANNE (2021-11236) - LAB1
GREGORIO, HERMINIO (2021-11715) - LAB2

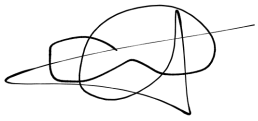
MEMBER	CONTRIBUTIONS	SIGNATURE
ATIENZA, CARMELO ELLEZANDRO	Front-end system Hardware assembly	
CASTAÑEDA, KRISTINA CASSANDRA	Hardware assembly & code Dragon's Den spiel	
DISPO, VINCENT ANGELO	Back-end system Hardware assembly & code	
DUCAY, JAMES DANIEL	Hardware assembly & code Slides and Poster	
GACHO, LORIDGE ANNE	Hardware assembly & code	
GREGORIO, HERMINIO	Front-end system Hardware assembly	

Table of Contents

1. Project Proposal and Updates

1.1. Rationale and User Story

Rationale

In tropical regions, factors like excessive heat, humidity, and sudden changes in weather conditions can pose significant hurdles to successful urban vegetable cultivation. Additionally, monitoring and maintaining optimal soil moisture levels, sunlight exposure, and temperature can be cumbersome tasks for growers, especially when they are not physically present to tend to their vegetables regularly.

The Super Pot addresses this challenge as a Smart Greenhouse Enclosure that provides optimal vegetable monitoring through smart technologies. It provides a controlled environment within its miniature greenhouse structure, shielding delicate sprouts from harsh sunlight and sudden weather changes. Additionally, the ability to regulate factors like soil moisture and temperature levels ensures optimal conditions for vegetable cultivation.

Vegetable cultivation, especially for home food production, offers numerous benefits, including reducing expenses on vegetables and greens, and providing potential profits from selling surplus produce. It promotes healthier food choices by avoiding pesticides and preservatives in commercial produce. Additionally, it fosters community bonding through the sharing of produce, seedlings, and gardening knowledge. On a personal level, gardening serves as a rewarding hobby and form of exercise, enhancing both physical and mental well-being.

By integrating IoT technology, we aim to create a self-sufficient system that automatically waters crops, adjusts shading based on weather conditions, and allows remote monitoring and control via the Internet.

User Story

a. As an office worker, I want automatic monitoring of the sprouts that I am cultivating so that it gives me peace of mind while I am outside the house.

b. As a beginner grower, I want to have an easier experience in cultivating vegetables that are sensitive to the slightest environmental changes.

c. As a sustainability advocate, I want to reduce my carbon footprint by growing my own vegetables. Though I have limited experience in growing my own food, the Super Pot can allow me to pursue this advocacy and contribute to a greener lifestyle.

1.2. Product Description

The Super Pot is an intelligent and self-sufficient Greenhouse Enclosure System for vegetable cultivation and seedling care. It incorporates sunlight and moisture sensors, automatic waterer, and weather-dependent shade mechanism that can be controlled and monitored remotely via the Internet, ensuring that plants receive ideal care regardless of their location.

Main Features

a. Moisture sensor-dependent automatic watering system to maintain ideal soil moisture levels.

b. Light sensor-dependent shade mechanism to protect plants from harsh sunlight.

c. Remote control watering and shade mechanism as well as monitoring via the Internet for convenience and accessibility.

1.3. Final List of Components

ITEM	DESCRIPTION	QUANTITY	UNIT PRICE	TOTAL
Unsoldered GY-30 BH1750FVI Module Digital Light Intensity Illumination Sensor	Sensor which detects the intensity of incoming sunlight in front of the greenhouse enclosure. Depending on the level of light intensity, the motorized hinge mechanism will trigger to open or close.	1	90	90
Soil Moisture Sensor Module for Arduino Raspberry Pi	Sensor which measures the current moisture of the soil for remote	1	60	60

	monitoring. Depending on the moisture levels, the automated watering system will be activated.			
SG90 Servo Motors	Actuator which motorizes the hinge mechanism of the device, enabling the opening and closing of the greenhouse enclosure.	4	109	436
1 Channel 5V Relay Module SPDT	This is used to control or switch devices which use a higher power than what most micro-controllers can handle. Specifically, this is used as a switch for the pump.	1	42	42
WeMosD1 - ESP8266	Microcontroller which provides processing power, memory, I/O capabilities, and Integrated Wi-Fi and dual-mode Bluetooth for our IoT device.	1	185	185
Breadboard		1	65	65

12V R385 Water Pump	Actuator which enables water delivery from the reservoir to the watering mechanism of the greenhouse enclosure	1	159	159
12V 5A AC/DC Power Supply Adapter	This adapter provides a stable 12 V output for electronic devices and equipment requiring up to 5 amps of current. This is specifically used to power our pump.	1	185	185
5V 3A DC Power Adapter	This adapter provides a stable 5 V output for electronic devices and equipment requiring up to 3 amps of current. This is specifically used to power our microcontroller.	1	120	120
Plastic Pot	Main container with cover for the greenhouse enclosure	1	580	580
Copper Wires	The wires are conductive materials that transmit electrical	15 meters	8	120

	power or signals from our microcontroller to the sensors, relay, servo motors, and pump.			
PVC pipe	The PVC pipe is connected to our servo motors as support in actuating the lid opening and closing mechanism. It is also used as the cover and pathway for the hose from the water source to the main pot.	1	130	130
Green hose (wire covering)	Protection for the wiring of servo motors and sensors.	1	30	30
Total				2202

1.4. System Architecture and Actuation FSMs

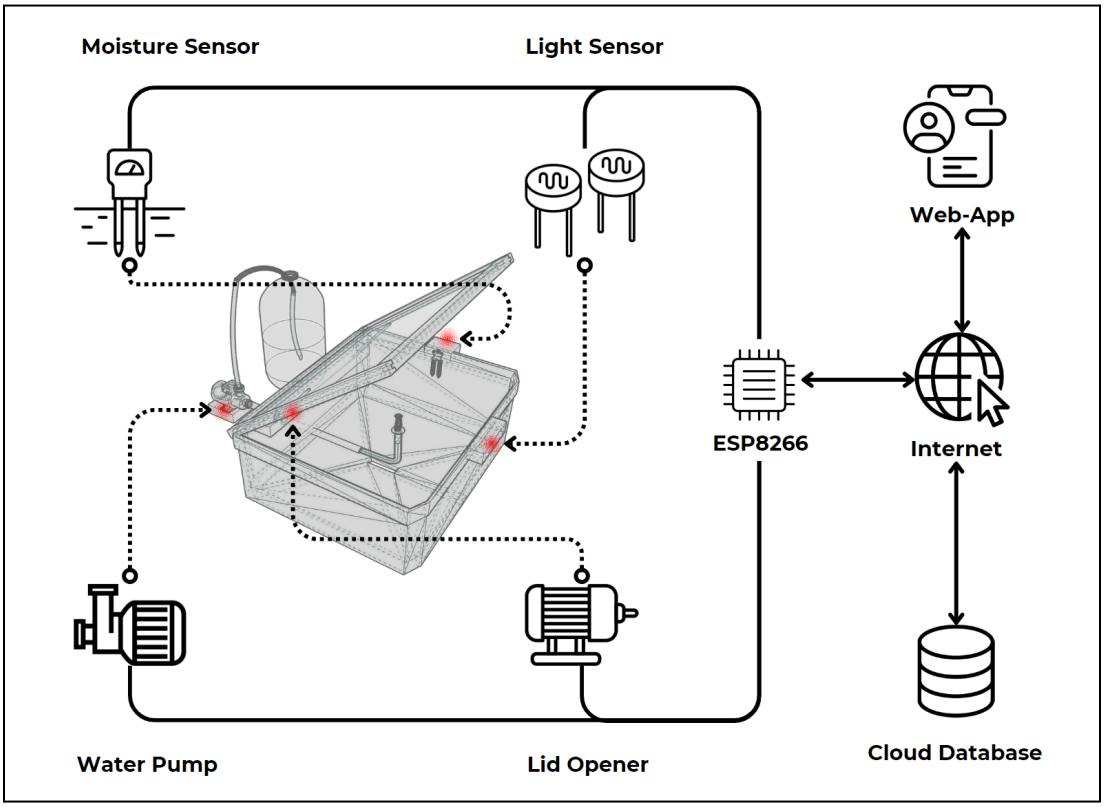


Figure 1. System Architecture

Main Hardware Components

- Unsoldered GY-30 BH1750FVI Module Digital Light Intensity Illumination Sensor
- Soil Moisture Sensor Module for Arduino Raspberry Pi
- 12V R385 Water Pump
- 4 SG90 Servo Motors
- WeMos D1 - ESP8266

Front-end (Website Link: https://mysuperpot.vercel.app/)

- NextJS
- TailwindCSS

Back-end

- Firebase Realtime Database

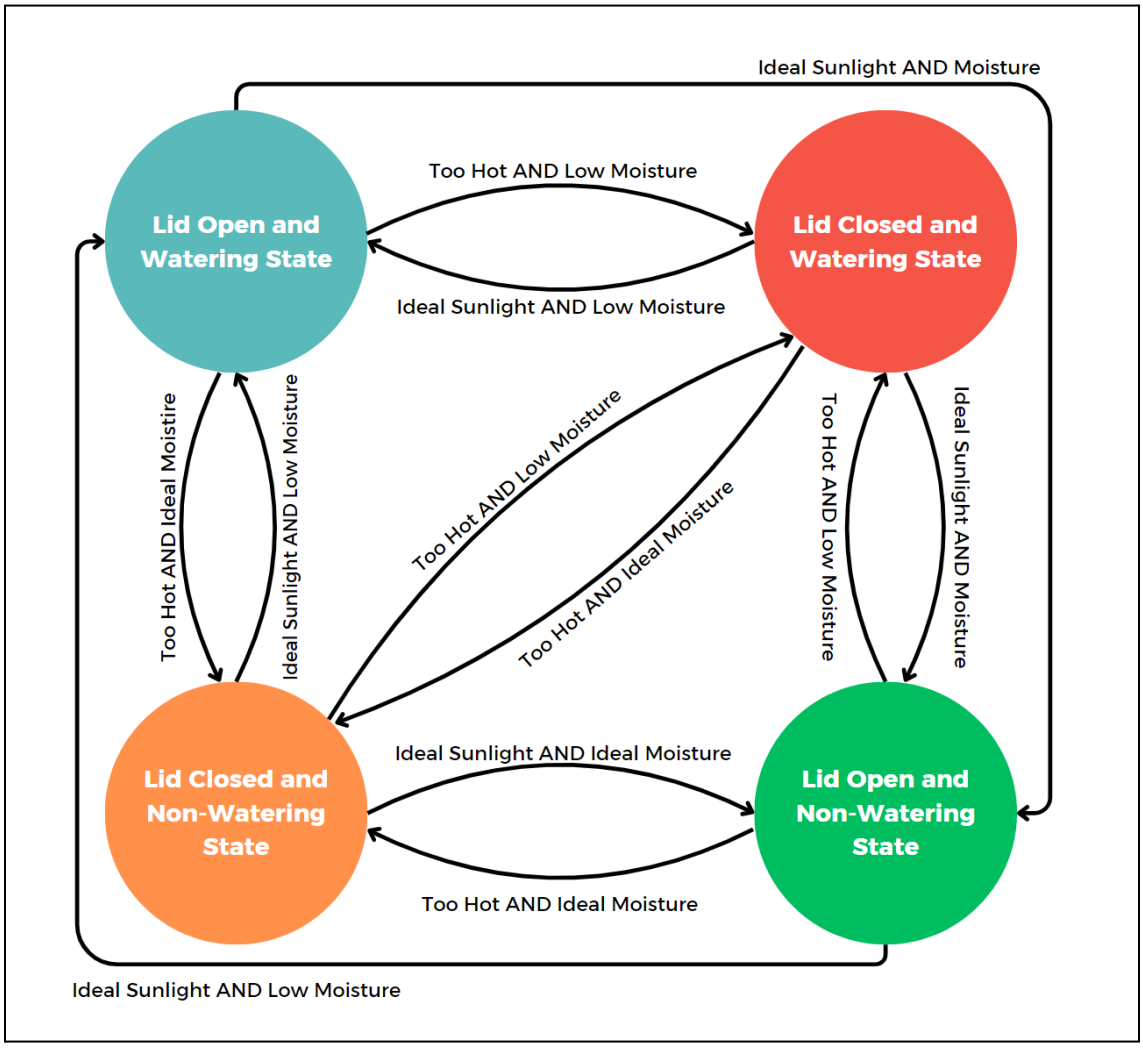


Figure 2. System Architecture FSM

2. Schematic Diagram

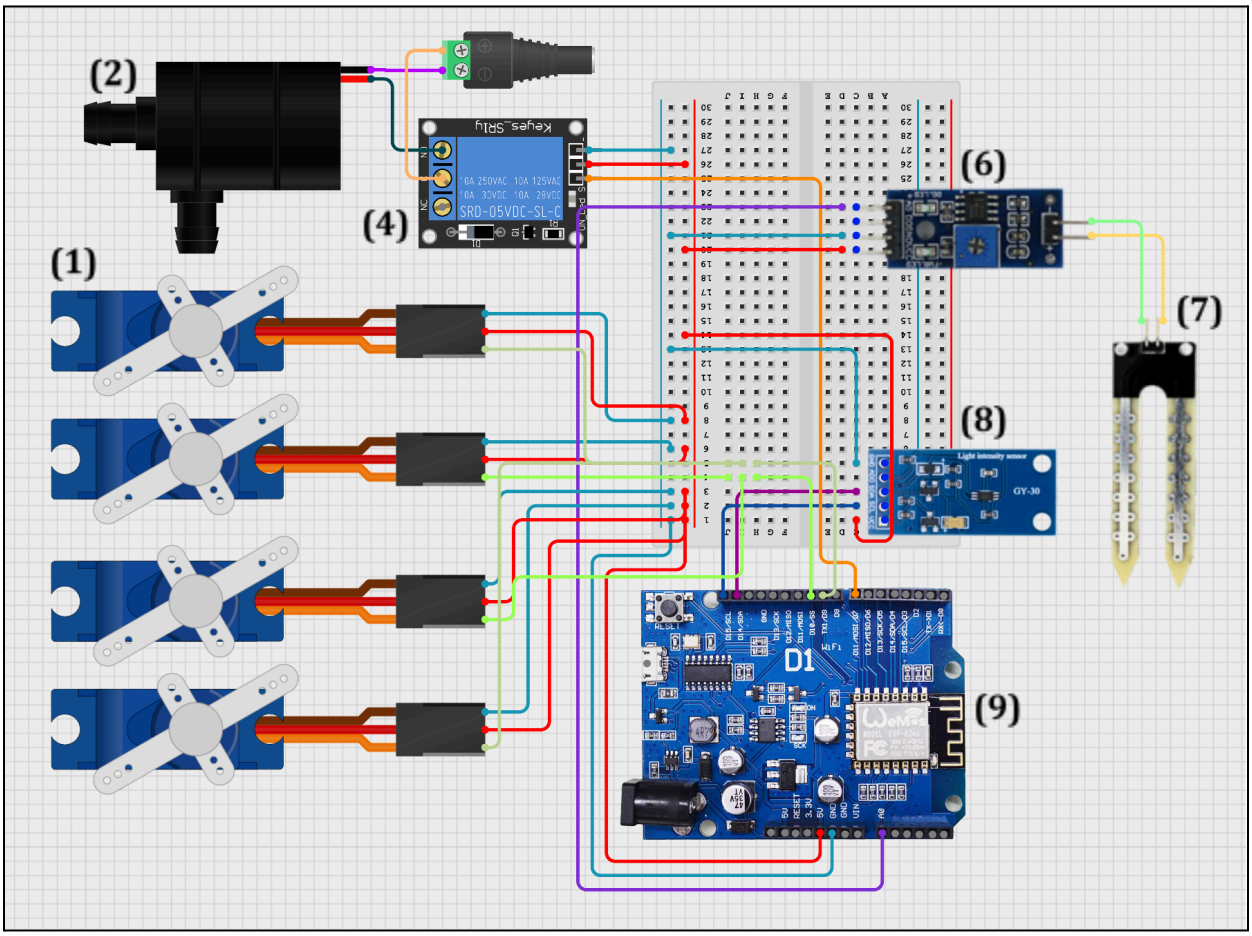


Figure 3. Schematic Diagram

Legend:

(1) Servo Motors
(2) Water Pump
(3) DC Barrel Jack Adapter
(4) Relay Module
(5) Breadboard
(6) Soil Hygrometer Detection Module
(7) Soil Moisture Sensor
(8) Light Intensity Sensor
(9) ESP8266 Microcontroller

1. **Setting up the Breadboard**
    a. The 5v pin on the ESP8266 microcontroller should be connected to the positive line on the breadboard using a wire.
    b. The GND pin on the ESP8266 microcontroller should be connected to the negative line on the breadboard using a wire.

2. **Servo Motors (for each)**
    a. The brown wire of each servo motor should be connected to the negative line of the breadboard. Recall that the negative line being referred to is where the GND pin is connected to.
    b. The red wire of each servo motor should be connected to the positive line of the breadboard. Recall that this is the positive line referred to is where the 5V pin is connected to.
    c. The D10 pin of the ESP8266 should be connected to one of the terminal strips, specifically row **4** using a wire.
    d. The signal pin (orange wire) of servo motor 1 should be connected on the **4th** row of the breadboard. Recall that this provides a connection between the servo motor and the D10 pin on the microcontroller.
    e. The signal pin (orange wire) of servo motor 2 should be connected on the **4th** row of the breadboard. Recall that this provides a connection between the servo motor and the D10 pin on the microcontroller.
    f. The D9 pin of the esp8266 should be connected to one of the terminal strips, specifically row **5** using a wire.
    g. The signal pin (orange wire) of servo motor 3 should be connected on the **5th** row of the breadboard. Recall that this provides a connection between the servo motor and the D10 pin on the microcontroller.
    h. The signal pin (orange wire) of the servo motor 4 should be connected on the **5th** row of the breadboard. Recall that this provides a connection between the servo motor and the D9 pin on the microcontroller.

3. **Light Intensity Sensor**
    a. A soldering iron should be used to solder the Unsoldered GY-30 BH1750FVI Module Digital Light Intensity Illumination Sensor.
    b. Connect the pins GND, VCC, SDA, and SCL of the soldered light sensor to the breadboard.
    c. A wire should be used to connect the SCL pin of the ESP8266 microcontroller to the SCL pin of the light sensor. This can be done by attaching the wire to the same row where the SCL pin of the light sensor is attached to on the breadboard.
    d. A wire should be used to connect the SDA pin of the ESP8266 microcontroller to the SDA pin of the light sensor. This can be done by attaching the wire to the same row where the SDA pin of the light sensor is attached to on the breadboard.
    e. The GND pin of the light sensor should be connected to the negative line on the breadboard using a wire. Recall that the negative line being referred to is where the GND pin is connected to when we set up the breadboard.
    f. The VCC pin of the light sensor should be connected to the positive line on the breadboard using a wire. Recall that the positive line being referred to is where the 5v pin is connected to when we set up the breadboard.

4. **Soil Moisture Sensor**
    a. Connect pins A0, GND, and VCC, of the moisture sensor to the breadboard.
    b. The GND pin of the moisture sensor should be connected to the negative line on the breadboard using a wire. Recall that the negative line being referred to is where the GND pin is connected to when we set up the breadboard.
    c. The VCC pin of the moisture sensor should be connected to the positive line on the breadboard using a wire. Recall that the positive line being referred to is where the 5v pin is connected to when we set up the breadboard.
    d. A wire should be used to connect the A0 pin of the ESP8266 microcontroller to the A0 pin of the moisture sensor. This can be done by attaching the wire to the same row where the A0 pin of the moisture sensor is attached to on the breadboard.

5. **Relay Module, Water Pump, and Power Supply**
    a. Connect pins GND, and VCC, of the 5V Relay Module to the breadboard.
    b. The GND pin of the 5V Relay Module should be connected to the negative line on the breadboard using a wire. Recall that the negative line being referred to is where the GND pin is connected to when we set up the breadboard.
    c. The VCC pin of the 5V Relay Module should be connected to the positive line on the breadboard using a wire. Recall that the positive line being referred to is where the 5v pin is connected to when we set up the breadboard.
    d. A wire should be used to connect the D7 pin of the ESP8266 microcontroller to the IN pin of the 5V Relay Module.
    e. A wire should be used to connect the Normally Open(NO) terminal of the 5V Relay Module to the positive terminal of the pump.
    f. A wire should be used to connect the Common(COM) terminal of the 5V Relay Module to the positive terminal of the power supply.
    g. A wire should be used to connect the negative terminal of the pump to the negative terminal of the power supply.

3. Source Code Explanations

3.1. Hardware Source Code

Setup

```
#include <Wire.h>               // Import wire library for communication I2C devices
#include <Servo.h>              // Import servo library
#include <BH1750.h>
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266mDNS.h>
#include <Firebase_ESP_Client.h>

#define GY30_ADDRESS 0x23       // address of GY30 light sensor

#ifndef STASSID
#define STASSID "Galaxy S23+ E481"
#define STAPSK "passpass"
#endif

#define API_KEY "AIzaSyDejxFnr9pcq7JhkMWtFo3vlylkSRNM414"
#define DATABASE_URL "https://superpot-9ef87-default-rtdb.asia-southeast1.firebasedatabase.app/"

#define USER_EMAIL "vdispo@up.edu.ph"
#define USER_PASSWORD "passpass"

FirebaseData fbdo;
FirebaseAuth auth;
FirebaseConfig config;
```

The code snippet above contains importing of required packages for the project. After the include statements, we define the address of the pins of the GY30 light intensity sensor. Next, the SSID and Passkey of the WiFi network the ESP will connect to is defined. Next, the API key and database URL for the Firebase realtime database is defined which is obtained from Firebase. Then we define the user email and password that is registered to the Firebase auth service which is registered using the web application. Lastly, instances of FirebaseData, FirebaseAuth, and FirebaseConfig are created and referenced with fbdo, auth, and config as variable names respectively.

```
26  const char * ssid = STASSID;
27  const char * password = STAPSK;
28
29  Servo servo1;
30  Servo servo2;
31
32  int relayControl = D7;
33  int moistPower = D8;
34
35  bool openLid = false;
36  bool openWater =  false;
37  bool isRaining = false;
38  bool manualCont = false;
39
40  int s1Angle = 0;
41  int s2Angle = 180;
42
43  // Consant variables or thresholds
44  const int LIGHTTHRESHOLD = 2000;
45  const int MOISTOPTIMAL = 500;
46  const int MOISTTOOWET = 400;
47  const int MOISTTOODRY = 900;
48
49  unsigned long openTime;                          // Variable to store open time
50  const unsigned long oneHour = 10000; // 3600000;     // oneHour = (60 * 60 * 1000) milliseconds
```

In the code snippet above, we put the defined WiFi access point name and passkey to the variables ssid and password respectively. Next, instances of Servo are defined and are referenced by servo1 and servo2. Next, the pins that are used to control the relay and power for the moisture sensor are defined in lines 32 - 33. Next, from lines 35 - 38, we define variables that describe the state of the actuators of the system. When `openLid` is set to true, the servos are powered to cause the lid to open, else it is powered to cause the lid to close. When `openWater` is set to true, the water pump of the pot is powered on, causing water to wet the soil. When the `isRaining` variable is set to true, the pot detects a state of too much moisture, thus commanding the servos to close the lid. When `manualCont` is set to true, it detects that a manual control of the lid is sent from the web app. The next section will discuss how these variables are checked and changed to change the state of the pot. Next, from lines 44 - 47, constant int variables are defined that are used as a basis when certain states change. `LIGHTTHRESHOLD` defines the light sensor value to which the pot will consider too bright causing `openLid` and `isRaining` to be set to false, `MOISTOPTIMAL` defines the moisture level that is just right for the soil and causes `openWater` to be set to false. `MOISTTOOWET` defines the moisture level that is too wet for the soil and causes isRaining to be set to true. `MOISTTOODRY` defines the moisture level too dry for the soil and causes `openWater` to be set to true. Lines 49 to 50 are used to keep track of the manual opening and closing of lid commands from the web app and ensures that manual commands from the web app do not persist permanently. openTime stores the time when the lid is opened automatically by the sensor oneHour is defined to be the time before a manual command is nullified.

We now move on to the void setup function which is executed once when the ESP8266 Microcontroller. Communication is initialized here.

Sensors

```
51  void setup() {
52    Serial.begin(115200);         // Initialize serial communication at 115200 baud (data rate per sec)
53
54    pinMode(A0, INPUT);             // For Moisture Sensor
55    pinMode(moistPower, OUTPUT);    // Power for Moisture Sensor
56    pinMode(relayControl, OUTPUT); // For Pump Relay
57
58    Wire.begin();                 // Initialize I2C communication
59    servo1.attach(D6);
60    servo2.attach(D5);
61    servo1.write(s1Angle);              // Initialize servo to closed state
62    servo2.write(s2Angle);
63    digitalWrite(moistPower, 0);  // Initialize moisture sensor power to off
```

This first part of the setup function sets up the pins of the sensors and actuators. After which, it sets up the moisture and light intensity sensors to read data and sets the servo motors and pump actuator's initial values. For the servo motors, it sets up the initial angles, `s1Angle`, and `s2Angle`, while for the moisture sensor, it sets it to off upon startup.

WiFi Connection

```
65    // Connect to Wi-Fi
66    WiFi.begin(ssid, password);
67    Serial.println("Connecting to Wi-Fi");
68
69    // Interrupt until connected to wifi
70    while (WiFi.status() != WL_CONNECTED) {
71      delay(500);
72      Serial.print(".");
73    }
74
75    Serial.println("");
76    Serial.print("Connected to ");
77    Serial.println(ssid);
78    Serial.print("IP address: ");
79    Serial.println(WiFi.localIP());
```

In the code snippet above, the connection to WiFi is initialized and the connection is attempted to be established in line 66. The while loop in lines 70 - 73 iterates until the WiFi connection is established. This effectively waits until the connection is established before proceeding to the next lines. The information of the connection is then printed in lines 75 - 79.

Firebase Database connection

```
81    // Firebase configurations
82    config.api_key = API_KEY;
83    config.database_url = DATABASE_URL;
84
85    auth.user.email = USER_EMAIL;
86    auth.user.password = USER_PASSWORD;
87
88    Firebase.reconnectNetwork(true);
89    fbdo.setBSSLBufferSize(4096, 1024);
90    fbdo.setResponseSize(2048);
91
92    Firebase.begin(&config, &auth);
93    Firebase.setDoubleDigits(5);
94
95    config.timeout.serverResponse = 10 * 1000;
96  }
```

This part of the function sets the API key and database URL for Firebase configuration. First, it assigns the API key, `API_KEY` to the Firebase configuration object config. This key is necessary to authenticate the microcontroller to the Firebase project. It then assigns the database URL, `DATABASE_URL` to the configuration object. This URL points to the Firebase Realtime Database endpoint.

Using an existing firebase account, it then assigns the user email, `USER_EMAIL` and the user password, `USER_PASSWORD` to the authentication object. After this, `Firebase.reconnectNetwork(true);` enables automatic reconnection to the network if the connection is lost. It helps maintain continuous connectivity to Firebase.

We then set a buffer size for managing memory usage and performance and also set a maximum response size that can be handled for Firebase operations to manage how much data can be processed in a single operation.

Void Loop Function

```
98  void loop() {
99    // Perform loop every 1 second
100   delay(1000);
101   if (Firebase.ready()) {
102     int light = readLight();
103     int moist = readMoist();
104
105     // Print moisture data
106     Serial.print("Moisture Level: ");
107     Serial.print(moist);
108     Serial.print(" Light Intensity: ");
109     Serial.print(light);
110     Serial.println(" lux");
111
112     // Get values for manual control from firebase
113     Firebase.RTDB.getBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/lidOpen", &openLid);
114     Firebase.RTDB.getBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/waterOpen", &openWater);
115     Firebase.RTDB.getBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/manualCont", &manualCont);
116
117     // Upload sensor values to firebase
118     Firebase.RTDB.setInt(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/sens/light", light);
119     Firebase.RTDB.setInt(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/sens/moist", moist);
```

In the void loop function, we create a loop that will get executed every second or 1000 milliseconds. But before doing so, there is a check if Firebase is ready for operations. The code inside the if statement will only execute if Firebase is properly initialized and ready to communicate.

We first call the functions `readLight()` and `readMoist()`, explained earlier, to read data from the light and moisture sensors, respectively. The values are stored in the variables light and moist. We then read the actuator values from Firebase. These values allow remote manual control of the lid and water through the internet.

The obtained current sensor values above are also written and uploaded to Firebase to display them in the web application.

Lid Logic

```
121   //// Light dependent actuation ////
122   if (isRaining) {          // Case for when it is currently raining
123     actuateServo(false);
124     if (light >= LIGHTTHRESHOLD) {
125       isRaining = false;
126     }
127   } else if (light >= LIGHTTHRESHOLD) {      // Threshold value to close the lid
128     openWater = true;       // Also open the water pump
129     if (!manualCont){          // Manual control of lid from web app
130       actuateServo(false);
131       openTime = millis();   // record time of opening
132       Serial.print("Time: ");
133       Serial.println(openTime);
134       Serial.print("");
135     } else {
136       actuateServo(openLid);
137       unsigned long currentTime = millis();
138       unsigned long elapsedTime = currentTime - openTime;
139       if (elapsedTime >= oneHour){
140         manualCont = false;
141         openLid = !openLid;    // Automatically reclose lid
142         Firebase.RTDB.setBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/lidOpen", openLid);
143         Firebase.RTDB.setBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/manualCont", manualCont);
144       }
145     }
146   } else {
147     actuateServo(true);
148   }
```

In this section, we discuss all the light dependent actuators for the pot. First, we scan if `isRaining` is set to `true` which indicates that the pot detects too much moisture which can be attributed to external factors such as rainfall. In this situation, we call the `actuateServo` function with the argument set to `false`. This command closes the lid to protect the pot from too much moisture. It also scans whether the light reaches a certain threshold to which it will consider it safe (it is not raining anymore) and sets the `isRaining` variable to false.

In the next else if block from line 127, we scan if the current light intensity is greater than the threshold that we set to indicate too much sunlight. In this situation, we turn on the pump by setting `openWater` to `true` in order to cool down the plants. Inside this else if block, we first check if `manualCont` is set to `false` which indicates that there is no manual control coming from the web app and total control is given to the sensors for actuation. In this case, we just close the lid by calling `actuateServo(false)` and recording the current time and storing it in the `openTime` variable to keep track of the last time the sensors took control of actuation of the lid. The next lines are just to print the recorded time. In the situation that `manualCont` is `true`, we enter the else block starting in line 135. In this situation, we set the status of our lid depending on the stored bool value in `openLid` which is modified by the web app as `manualCont` is `true`. Then, we record the current time and store it in the `currentTime` variable. We then check if the elapsed time from the last time the lid was controlled by the sensor is greater than the `oneHour` variable we defined

earlier. In this case, we now set the `manualCont` variable to `false` and the `openLid` variable to be its inverse. We then upload these values to Firebase as seen in lines 142 and 143.

After the else if block in line 127, if the light levels are not too harsh, then we just set the lid to open by calling `actuateServo(true)`.

Pump

The code snippet below is designed to control the pump for watering plants based on solid moisture level. It uses a relay to turn the pump on and off and interacts with a Firebase Realtime Database to update the pump's status.

The code initially verifies whether `openWater` is true (indicating that watering is currently active). If it is, the next step is to assess the soil moisture level (moist). If the moisture level is less than or equal to `MOISTOPTIMAL`, it indicates that the soil has enough moisture, so watering should stop. It sets openWater to false and turns the pump off using `digitalWrite(relayControl, HIGH)`. If the moisture level is higher than `MOISTOPTIMAL`, it continues watering by keeping the pump on using `digitalWrite(relayControl, LOW)`.

If `openWater` is false and the soil is too dry (`moist >= MOISTTOODRY`), it sets `openWater` to true to start watering. If the soil is too wet (`moist <= MOISTTOOWET`), it sets `isRaining` to true, likely indicating that it's raining and additional watering is not needed.

Finally, the code updates the Firebase Realtime Database with the current status of `openWater`. This allows the system to keep track of whether the pump is on or off.

```
150     ///// FOR PUMP /////
151     // If RelayControl1 is HIGH, pump is off
152     // If RelayControl1 is LOW, pump is on
153
154     // Catches case for both manual and automatic watering of plant
155     if (openWater) {
156       // Stopping of watering plant
157       if (moist <= MOISTOPTIMAL) {
158         openWater = false;
159         digitalWrite(relayControl, HIGH);
160       } else {
161         digitalWrite(relayControl, LOW);
162       }
163     } else if (moist >= MOISTTOODRY) {
164       openWater = true;
165     } else if (moist <= MOISTTOOWET) {
166       isRaining = true;
167     }
168     Firebase.RTDB.setBool(&fbdo, "/users/MDKPitgeP9SqreHsc1yw7RcS8yy1/actu/waterOpen", openWater);
169   }
170 }
```

Reading Light

The code below is designed to read light intensity from a GY-30 light sensor. The function `int readLight()` communicates with the sensor, initiates a high-resolution light measurement, waits for the measurement to complete, reads the data from the sensor, and then returns the light intensity in lux.

`int val = 0` initializes the variable `val` to 0. This variable will hold the light intensity value. `Wire.begintransmission(GY30_ADDRESS)` initiates communication with the GY-30 sensor at the address specified by the variable `GY30_ADDRESS`. `Wire.write(0x10)` sends a command to the sensor to select high-resolution mode. `Wire.endTransmission()` ends the communication with the sensor. Next, `delay(200)` pauses the program for 200 milliseconds to allow the sensor to take the measurement. This delay ensures that the sensor has enough time to capture the light intensity data.

`Wire.requestFrom(GY30_ADDRESS, 2)` requests 2 bytes of data from the sensor. The light intensity measurement is stored here. `if(Wire.available() == 2)` checks if the 2 bytes of data are received from the sensor. If so, it reads the 2 bytes of data from the sensor and combines them into a single integer. The first byte is read, shifted left by 8 bites (`Wire.read() << 8`), and then combined with the second byte using a bitwise OR. (`| Wire.read()`). The final light intensity is stored in `val`. Finally, it returns the combined integer value representing the lux measured by the sensor.

```
172  int readLight() {
173    int val = 0;
174    Wire.beginTransmission(GY30_ADDRESS);     // Initiates communication with the GY30 sensor
175    Wire.write(0x10);                         // Command to select high resolution mode
176    Wire.endTransmission();                   // Ends the transmission to the sensor
177    delay(200);                               // Wait for measurement to be taken (arbitrary time)
178
179    Wire.requestFrom(GY30_ADDRESS, 2);        // Request 2 bytes of data
180    if(Wire.available() == 2){                // If 2 bytes of data is received
181      val = (Wire.read() << 8) | Wire.read(); // Combine the two bytes into a single integer
182    }
183    return val;                               // return the combine bytes (the lux value)
184  }
```

Read Moisture

The function `readMoist()` below is designed to read the soil moisture level using a moisture sensor connected to an analog pin (`A0`) on a microcontroller. The sensor is powered on just before taking a reading and powered off immediately afterward to conserve power, prevent electrolysis, and prevent electrochemical reactions that can lead to inaccurate moisture readings from the sensor..

`int moist` declares an integer variable that will store the moisture level read from the sensor. `digitalWire(moistPower, HIGH)` sets the pin connected to the moisture sensor's power supply to HIGH, turning the sensor on. This ensures that the sensor is only powered when a reading is needed. `moist = analogRead(A0)` reads the analog value from pin `A0`. The analog value represents the moisture level and is stored in the `moist` variable. `digitalWrite(moistPower, LOW)` sets the power supply of the sensor to LOW, turning the sensor off. This ensures that the sensor does not consume power when not in use and electrochemical reactions that can make the readings of the sensors inaccurate are minimal. Finally, the function returns the integer value `moist`, representing the moisture level.

```
186  int readMoist() {
187    int moist;
188    digitalWrite(moistPower, HIGH);
189    moist = analogRead(A0);
190    digitalWrite(moistPower, LOW);
191    return moist;
192  }
```

Actuate Servo

The function `actuateServo()` is designed to control two servos (`servo1` and `servo2`) to open or close the lid in a coordinated manner. The function gradually moves the servos to the desired positions.

The code initially verifies if the lid is open and `s1Angle` is at 180 degrees and `s2Angle` is at 0 degrees. These conditions likely represent that the lid is closed. If the conditions are met (meaning the servos are in the closed position and we want to open them), a `for` loop runs 18 times to gradually change the angles of the servos. Within each iteration, `s1Angle` is decreased by 10 degrees and `s2Angle` is increased by 10 degrees. The new angles are written to `servo1` and `servo2` using `servo1.write(s1Angle)` and `servo2.write(s2Angle)`. A delay of 100 milliseconds is introduced to make the movement smooth.

The `else if` statement checks if the lid is closed, if `s1Angle` is at 0 degrees, and `s2angle` is at 180 degrees. These conditions likely represent that the lid is open. If the conditions are met (meaning the servos are in the open position and we want to close them), A `for` loop runs 18 times to gradually change the angles of the servos. Within each iteration, `s1Angle` is increased by 10 degrees and `s2Angle` is decreased by 10 degrees. The new angles are written to `servo1` and `servo2` using `servo1.write(s1Angle)` and `servo2.write(s2Angle)`. A delay of 100 milliseconds is introduced to make the movement smooth.

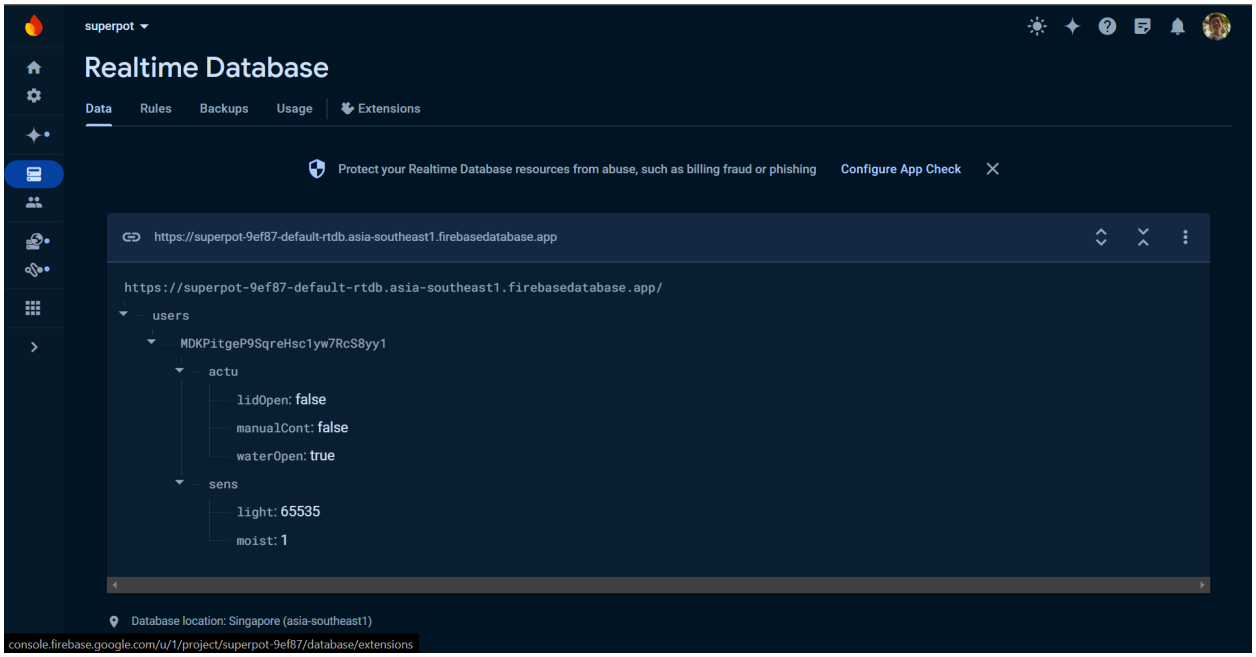
Finally, the function returns 0, which is used as a status code to indicate successful execution.

```
194  int actuateServo(bool open) {
195    if (open && s1Angle == 180 && s2Angle == 0) {
196      for (int i = 0; i < 18; i++) {
197        s1Angle = s1Angle - 10;
198        s2Angle = s2Angle + 10;
199        servo1.write(s1Angle);
200        servo2.write(s2Angle);
201        delay(100);
202      }
203    } else if (!open && s1Angle == 0 && s2Angle == 180) {
204      for (int i = 0; i < 18; i++) {
205        s1Angle = s1Angle + 10;
206        s2Angle = s2Angle - 10;
207        servo1.write(s1Angle);
208        servo2.write(s2Angle);
209        delay(100);
210      }
211    }
212    return 0;
213  }
```

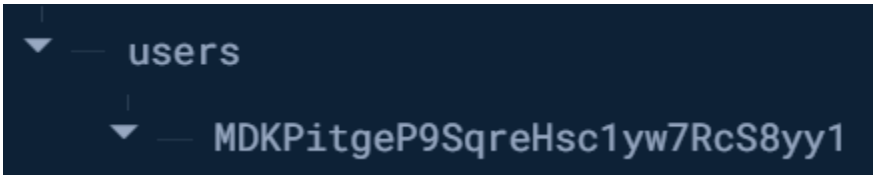
3.2. Backend Implementation

The backend used for SuperPot is Firebase, a backend-as-a-service platform which provides helpful modules that allows for an easier implementation of data sending and fetching. In particular, the Realtime Database is used, where it can handle continuous data readings from our ESP 8266 microcontroller, and update it realtime based on a user-defined polling rate.

The figure attached below is a snapshot of our Realtime Database. It does not feature a relational structuring of data; rather, it hierarchically represents the realtime data. Given the scale of our project, this should still be feasible.



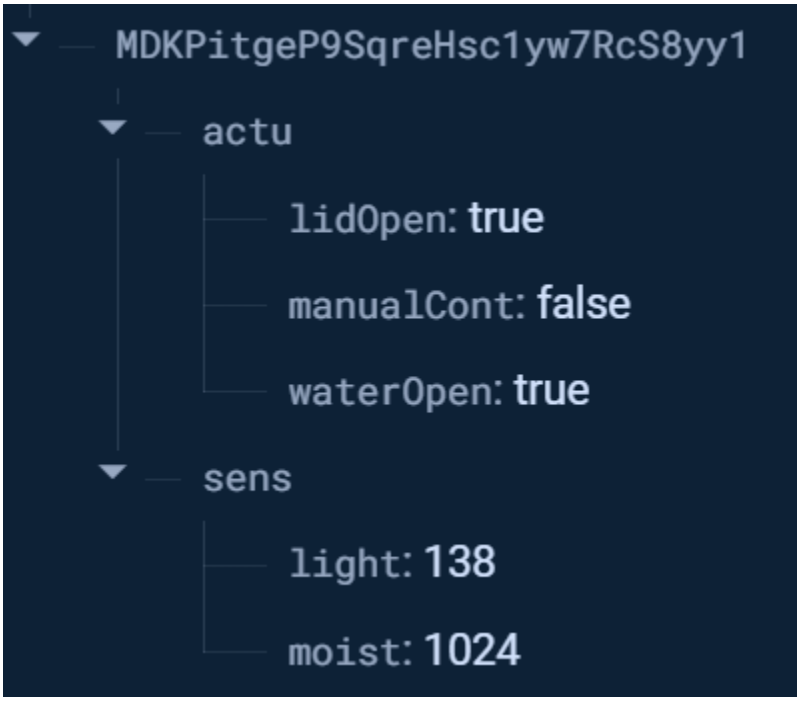
We can observe that under the `users` field are the `uid` of each user within the database. In our project, we have only set up **one user** for the database. This implies that every registered user will see the data for the `uid` of `MDKPitgeP9SqreHsc1yw7RcS8yy1`. **This is a current limitation of our project.** Of course, if the product is commercialized, we would fix this so that each registered account would have its own independent reading of their SuperPot.



We have two types of data stored for a given `uid`. First, we have the `actu` field. This field stores the current state of the different actuators of our hardware. This includes `lidOpen`, `manualCont`, and

`waterOpen`. The `lidOpen` and `waterOpen` fields contain data whether the lid of the pot is open, and the water pump is open, respectively. The `manualCont` signifies whether the user activates manual control of the lid from the web application.

Second, we have the `sens` field. This field stores the most recent readings from our different sensors, as transmitted by the ESP 8266 microcontroller. The `light` field contains data pertaining to the lux reading of our light sensors. The `moist` field contains data readings from the moisture sensor.

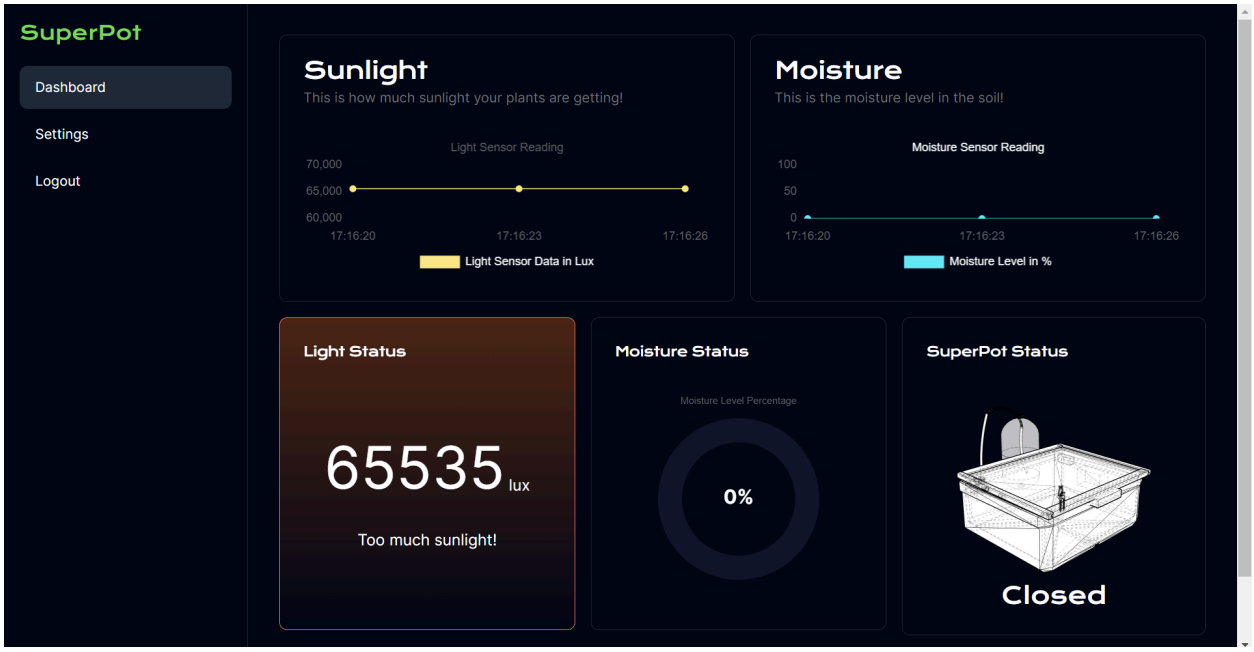


3.3. Frontend Source Code

The web application for SuperPot is implemented via the NextJS framework, and is bundled and deployed on Vercel. As previously mentioned, you may find our deployed website on https://mysuperpot.vercel.app/; you may also browse the GitHub repository of the SuperPot website. You may login to the dashboard with the following credentials:

Email: *vdispo@up.edu.ph*
Password: *passpass*

The main focus of the source code discussion will be on the **Dashboard Page** of the website, as it primarily deals with the polling of real-time data from the ESP 8266 microcontroller, intermediated by our Firebase Realtime Database backend.



Since NextJS is used as the framework, some terminologies used may require some knowledge of React; though, some discussed points here should be framework-agnostic. Located in the image below is our Dashboard page, along with some initial declarations of the [state variables] that we will

be using. First, we hard code the `uid,` which is set to `MDKPitgeP9SqreHsc1yw7RcS8yy1` . This was the default user id as specified within our Database Implementation.

Then, we declare our state variables for the sensors as a JSON object, named `sensors`. This object has two fields, the `light` and `moist`, where we will be storing the read data from our database.

```
export default function Dashboard() {
  const uid = "MDKPitgeP9SqreHsc1yw7RcS8yy1";
  const [sensors, setSensors] = useState<sensorData>({
    light: 0,
    moist: 0,
  });
```

3.3.1. Fetching Data from Database

The `fetchData` function is an asynchronous utility responsible for fetching and updating the sensor and actuator data for the dashboard in real-time. It plays a crucial role in ensuring that the user interface reflects the current state of the sensors and actuators.

```
const fetchData = async () => {
  const sensData = await ReadData(uid, "sens");
  const actuData = await ReadData(uid, "actu");
  const moistPercent = 100 - (sensData.moist / 1024) * 100;
  setSensors({
    light: sensData.light,
    moist: Math.round(moistPercent * 100) / 100,
  });
  console.log(sensors);
  setLidStat(actuData.lidOpen);
  setManualCont(actuData.manualCont);

  if (!copy) {
    setManualLid(actuData.lidOpen);
    setCopy(true);
  }
  setLightData((prevData) => [
    ...prevData.slice(-9),
    { timestamp: new Date(), light: sensData.light },
  ]);

  setMoistData((prevData) => [
    ...prevData.slice(-9),
    { timestamp: new Date(), moist: moistPercent },
  ]);
};
```

The function retrieves sensor data (`sensData`) and actuator data (`actuData`) from the real-time database using the `ReadData` function. It accesses data specific to the user identified by `uid`.

The moisture data is converted into a percentage by subtracting the raw value from 1024, normalizing it to a range of 0-100, and then rounding it to two decimal places.

It also performs the following updates to our state variables:

- Updates the `sensors` state with the latest light and moisture sensor readings.
- Sets the `lidStat` state to reflect whether the lid is currently open.
- Updates the `manualCont` state to indicate if manual control mode is active.

It also handles the manual override controls:

- Initializes the `manualLid` state to the current lid status during the first data fetch to avoid overwriting user settings on subsequent fetches.
- Utilizes a `copy` state flag to ensure that the manual lid status is only copied once.

It also manages our Chart Data for easier representation of our data:

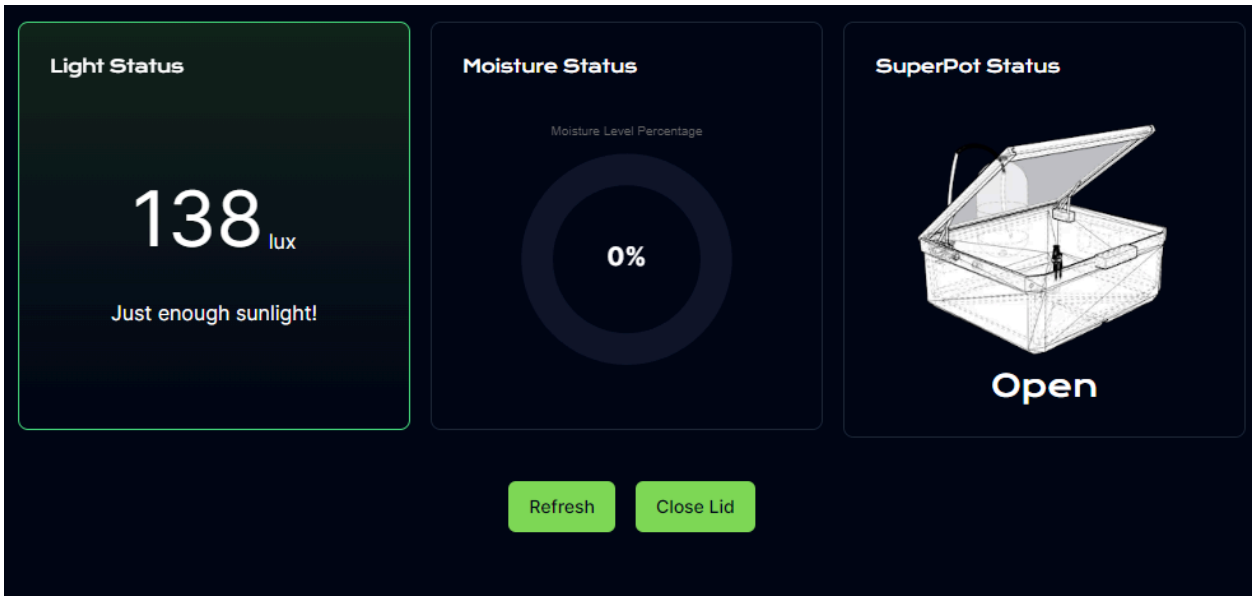
- Appends the latest light and moisture readings to their respective state arrays (`lightData` and `moistData`), ensuring the chart data arrays retain only the most recent 10 readings for performance and clarity.

This function is called at a regular interval of **3000 milliseconds (3 seconds)** within the `useEffect` hook to provide real-time updates to the dashboard.

Thus, using the `fetchData` effectively retrieves all necessary data for it to be displayed on the user interface of our Dashboard. The implementation of the UI is not covered in this documentation, though you may refer to the GitHub repository as to how it stylized the data for presentation.

3.3.2 Overriding Lid Status on Manual Control

Within our Dashboard, we can toggle the manual control of the lid via the **Open/Close Lid button.** Meaning, when the lid is open, we can choose to close the lid manually, overriding our sensor readings, and vice versa. To perform this, we need to overwrite the realtime database with inputs from the user.



The `handleLidButton` function is an asynchronous utility responsible for toggling the state of the lid actuator in the system. This function manages both the lid's open/close state and the manual control mode, ensuring that user interactions with the lid button are accurately reflected in the system's state and the real-time database.

```
const handleLidButton = async () => {
  await SetData(uid, !manualLid, "lidOpen");
  if (manualCont) {
    await SetData(uid, false, "manualCont");
  } else {
    await SetData(uid, true, "manualCont");
  }
  setLidStat(!manualLid);
  setManualLid(!manualLid);
};
```

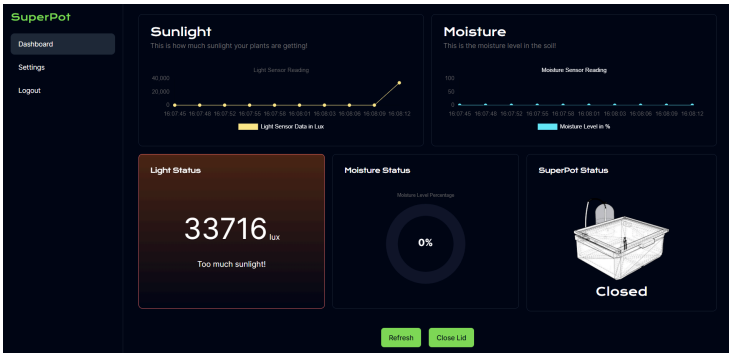
The function toggles the `manualLid` state, which represents the manual control status of the lid (open or closed). It calls the `SetData` function to update the `lidOpen` state in the real-time database. This action ensures that the database reflects the new lid state.

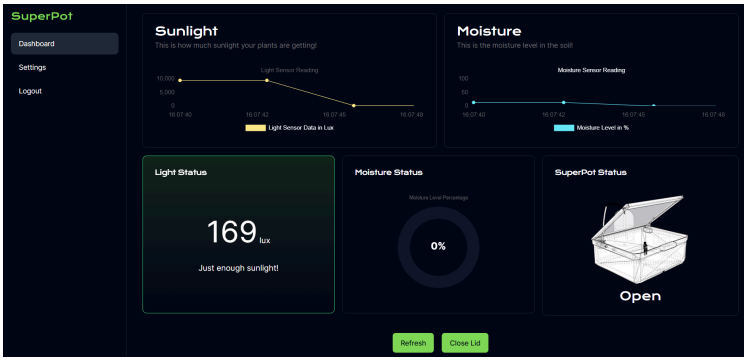
It also checks the current `manualCont` state to determine if manual control mode is active. If it is active, the function sets `manualCont` to `false` in the database, indicating that the system should no longer be in manual control mode. If manual control mode is not active, the function sets `manualCont` to `true` in the database, enabling manual control mode.

After, it updates the local `lidStat` state to match the new state of the `manualLid`, ensuring the user interface reflects the correct lid status. It also updates the `manualLid` state to the new toggled state, synchronizing the local state with the database.

4. Demonstration

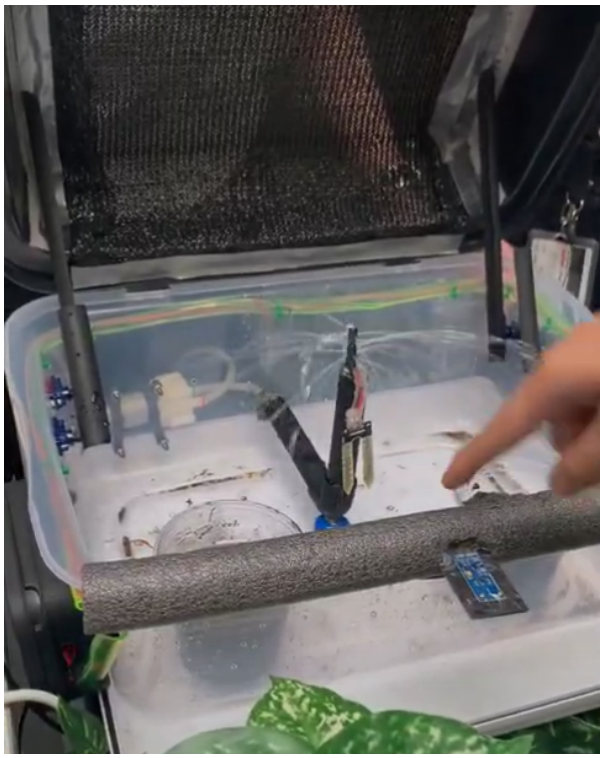
4.1. Minimum and Additional Functionalities

MINIMUM FUNCTIONALITIES	
1. Pot covers the pot with a transparent mesh when sunlight reaches a threshold level.	
	 As depicted in the figure, intense sunlight is simulated using a flashlight. The lid automatically shuts when a high level of sunlight is detected (given the placeholder value of greater than or equal to 5000 lux). This sunlight intensity can also be monitored through the web app's dashboard.

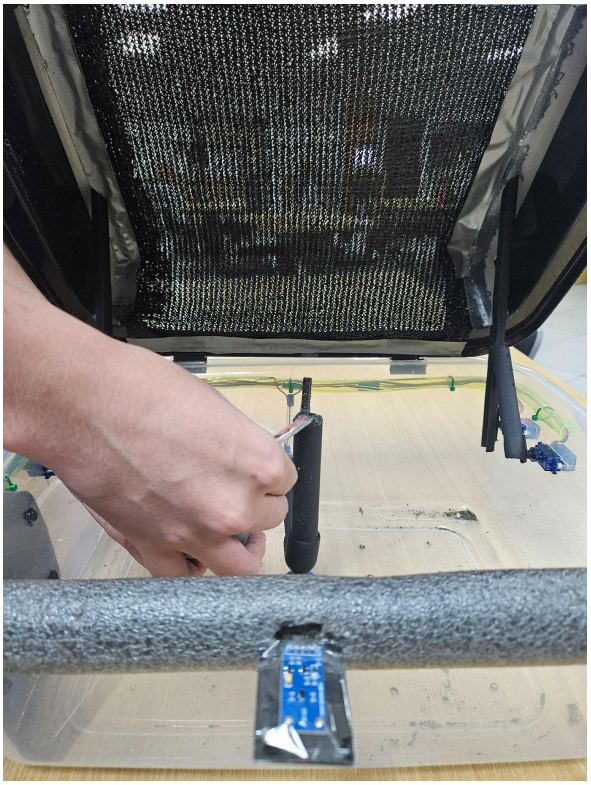


The lid reopens when a sufficient or healthy amount of sunlight is present (given the placeholder value less than 5000 lux) as simulated by removing the flashlight.

2. Stop watering the plants at a certain moisture level.

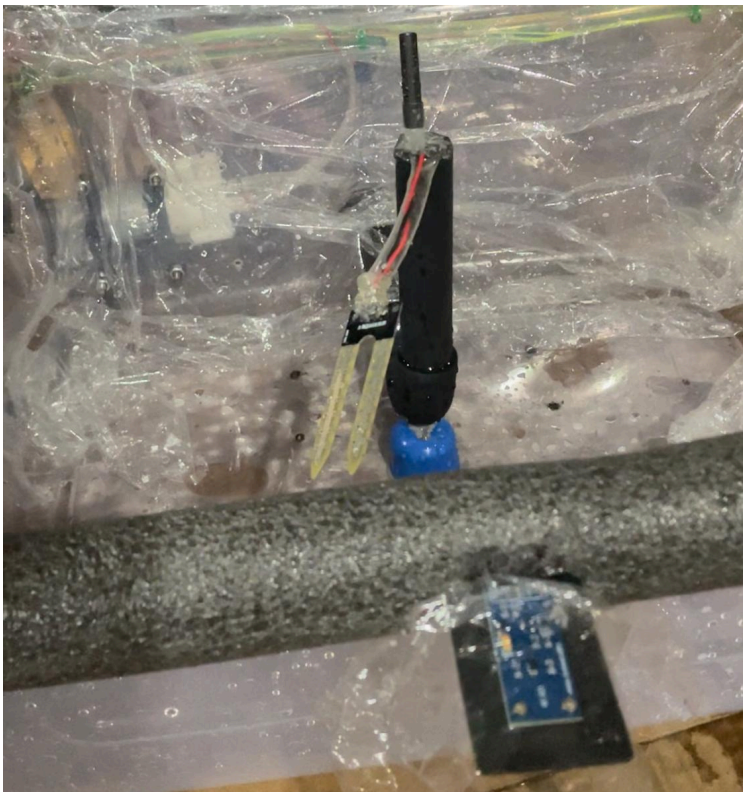


As observed in the figure, the low moisture level, (equating to 22.46% moisture level), is simulated by removing the moisture sensor from the damp soil. This activates the water pump connected to a water source behind.



On the other hand, when the soil moisture is inserted in the damp soil (equivalent to 83.2% moisture level), the water pump will turn off.

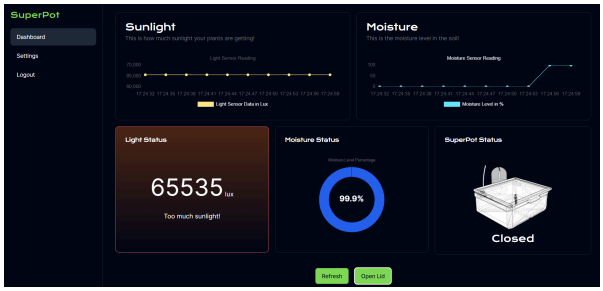
3. Pot automatically waters the plants at certain levels of sunlight.



As observed in the image, harsh sunlight is simulated by a flashlight.

The water pump is automatically turned on with the given sunlight level while a low level of moisture or dry soil is detected (the soil moisture sensor is not inserted into moist soil).

4. Control shade mechanism and monitor moisture/sunlight status remotely via the app.	
	Through our Vercel app, we manually close the lid of our SuperPot through the “Close Lid” button. As seen in the screen display, the lid status is still open before using the button.
	After pressing the “Close Lid” button, it can be observed that the lid status and the actual lid of the SuperPot is closed.

ADDITIONAL FUNCTIONALITIES	
1. Pot covers the plants from rain when a threshold moisture level is reached (for rainy situations)	
	 As depicted in the images, the servo motor arms are in a closed position, signifying that the lid is down. For demonstration purposes, the lid has been temporarily removed. Damp soil is simulated by placing the moisture sensor in the soil. When a high moisture level is detected, as shown in the dashboard image, indicating rainy weather, the lid closes.

Link to a sample demonstration taken at our booth: [Booth_demo]

4.2. Wireshark Packet Analysis

The packet analysis detailed below applies to all the functionalities mentioned.

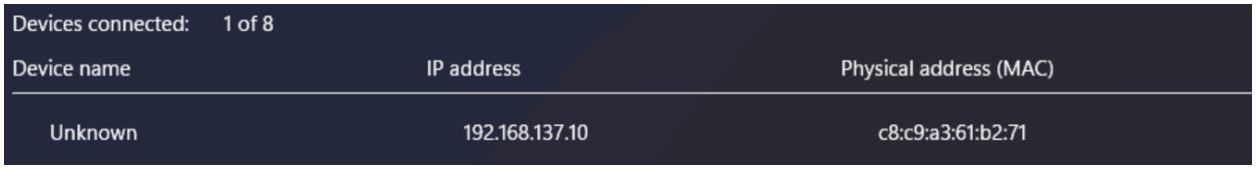


Figure 4. Hotspot settings menu

From the hotspot settings menu, we can observe that the Physical address (MAC) is c8:c9:a3:61:b2:71. From packet 443 in the screenshot below, we can conclude that the IP address of the ESP 8266 from the tracefile is 192.168.137.120. This is shown in the Source field in Ethernet II where its value is Espressif_61:b2:71 (c8:c9:a3:61:b2:71) in Figure 5 (Packet 443).

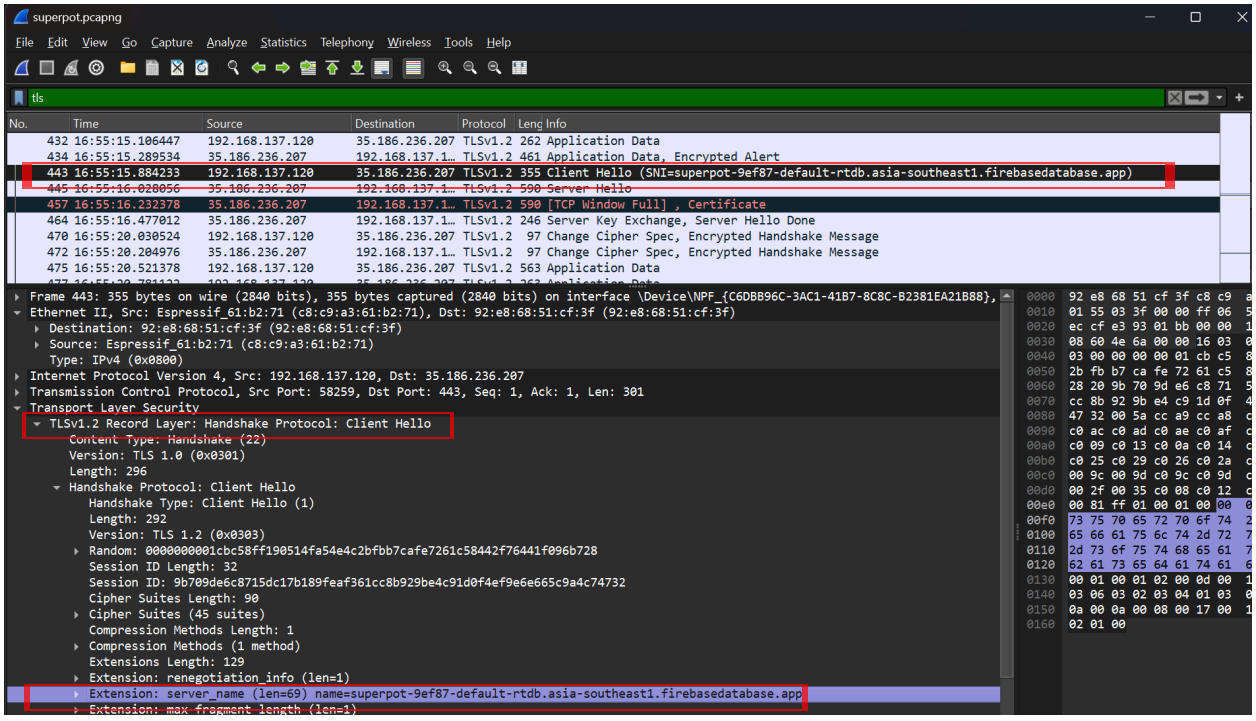


Figure 5. Packet 443 (Client Hello Packet)

Observe in Figure 5 that it is a Client Hello Packet. The client at 192.168.137.120 (IP Adress of the ESP 8266) initiated a TLS handshake with the server at 35.186.236.207, sending a Client Hello message.

The Server Name Indication (SNI) Extension field in the Transport Layer Security Protocol in Packet 443 is used to indicate the hostname that the client is attempting to connect to during the TLS handshake. The “name=superpot-9ef87-default-rtdb.asia-southeast1.firebasedatabase.app” here is the actual server name that the client is trying to connect to, indicating a request to connect to a

Firebase Realtime Database. From this, we can also conclude that the Destination IP Address 35.186.236.207 is associated with the Firebase Server.

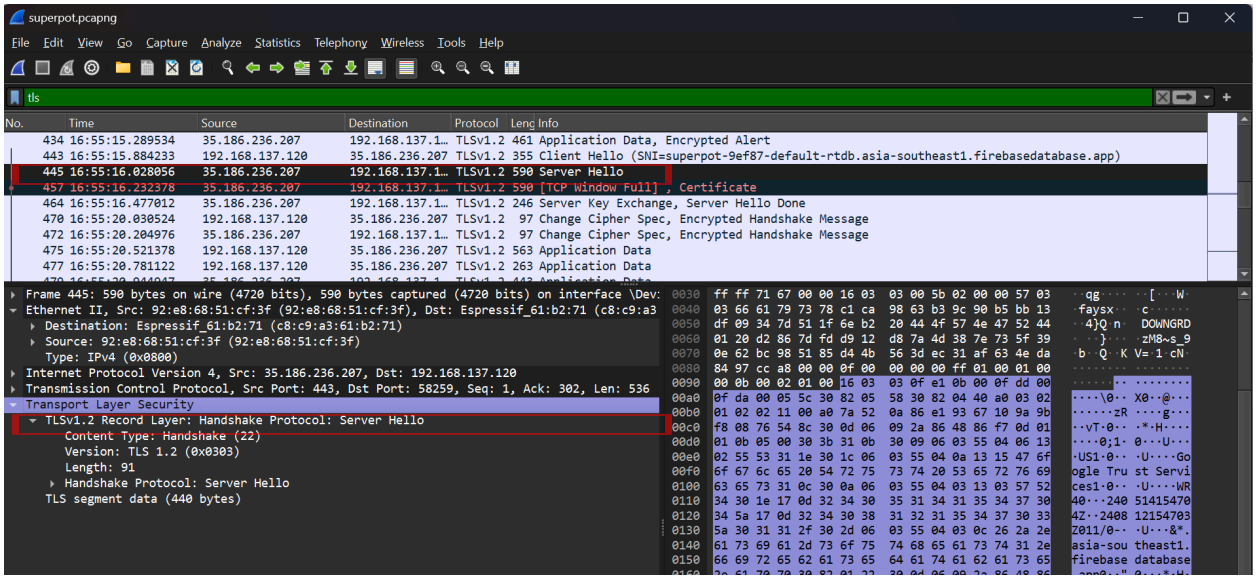


Figure 6. Packet 445 (Server Hello Packet)

The server or Firebase responds with a Server Hello message. This message confirms that the server is willing to proceed with the connection.

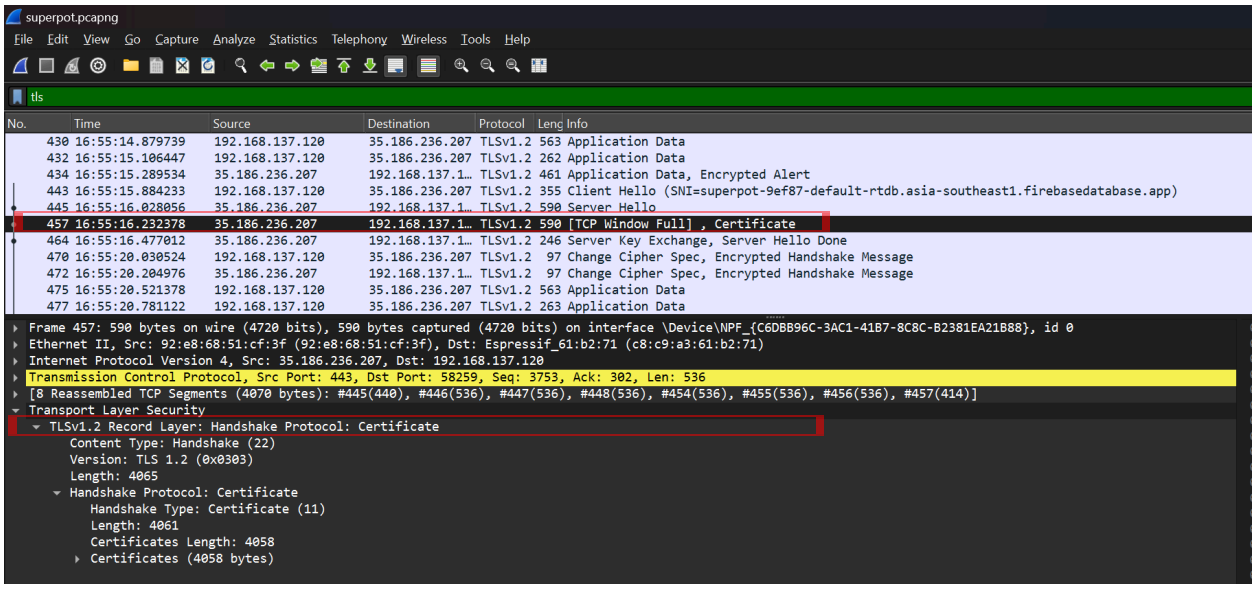


Figure 7. Packet 457 (Server Certificate Packet)

The server sends its digital certificate, which the client uses to authenticate the server.

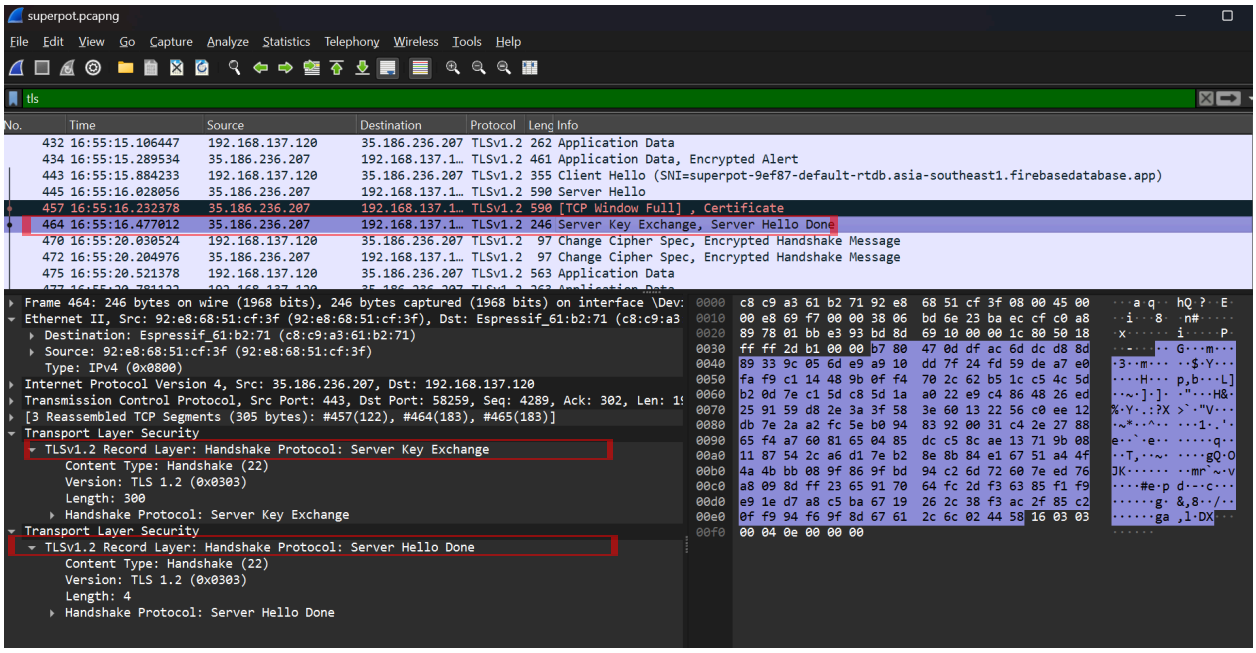


Figure 8. Packet 464 (Server Key Exchange and Server Hello Done Packet)

The "Server Key Exchange" packet (Packet 464) provides necessary key exchange parameters to establish the shared secret key, while the "Server Hello Done" signals the completion of the server's initial handshake messages.

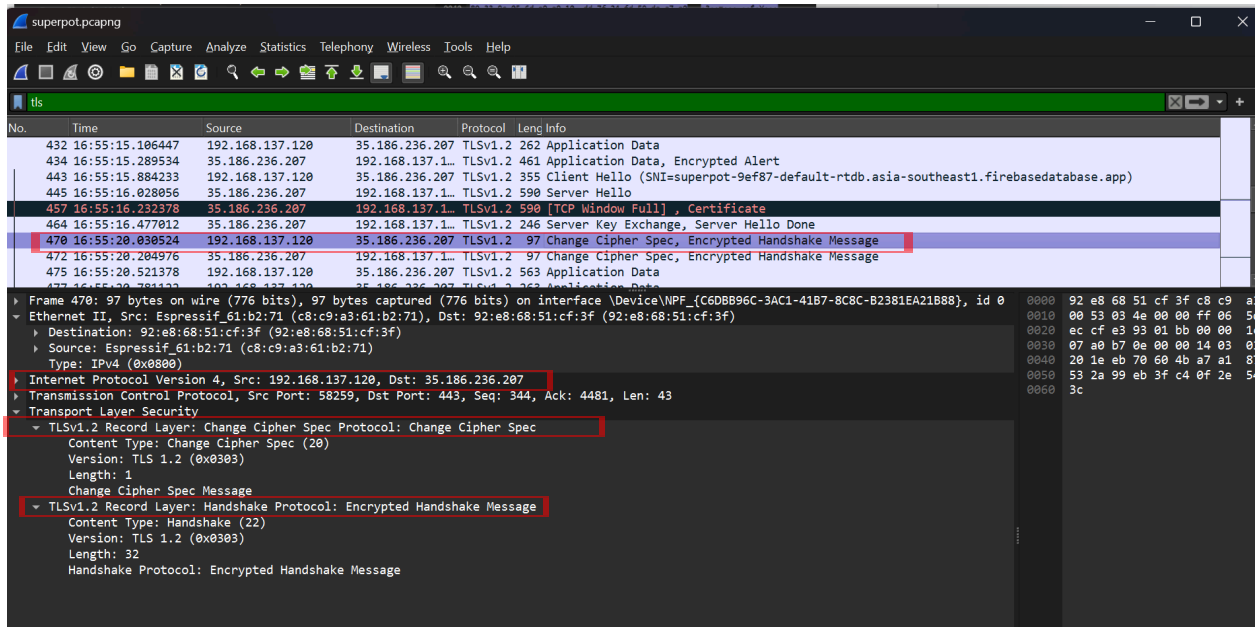


Figure 9. Packet 470 (Change Cipher Spec Packet)

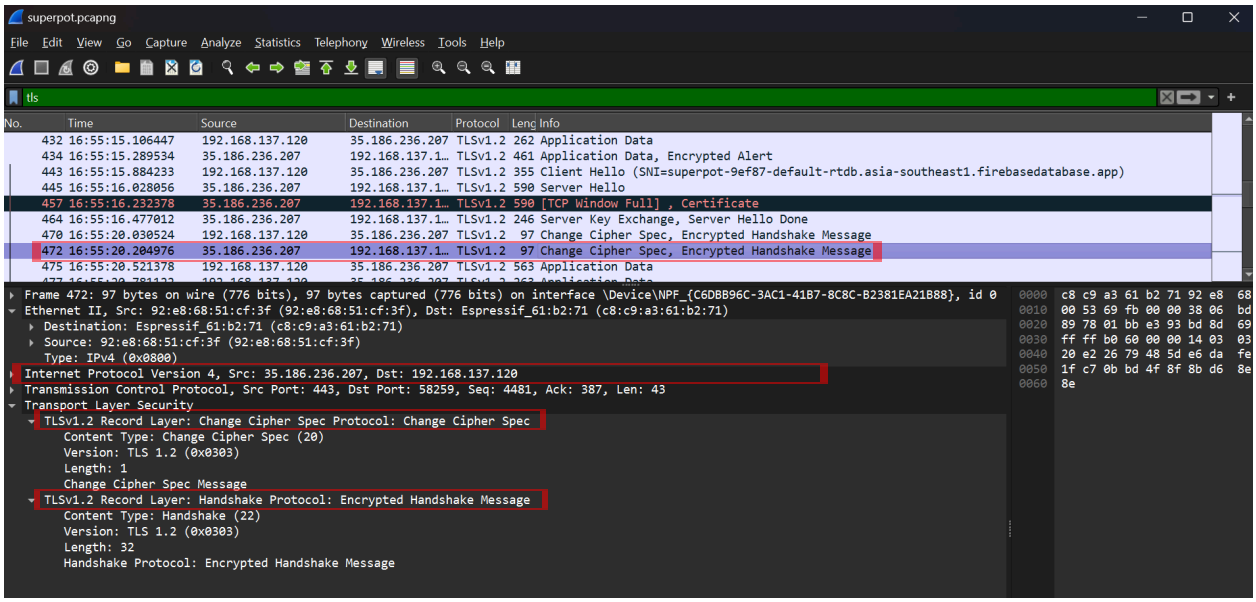


Figure 10. Packet 472 (Change Cipher Spec Packet)

Observe in Figures 9-10 that both the client(IP Address 192.168.137.120) and server(IP Address 35.186.236.207) send a Change Cipher Spec message to indicate they will now use the agreed-upon encryption parameters that they will use in the session.

Figures 5-10 indicate that there is an established connection between our SuperPot (i.e. the ESP 8266) with Firebase.

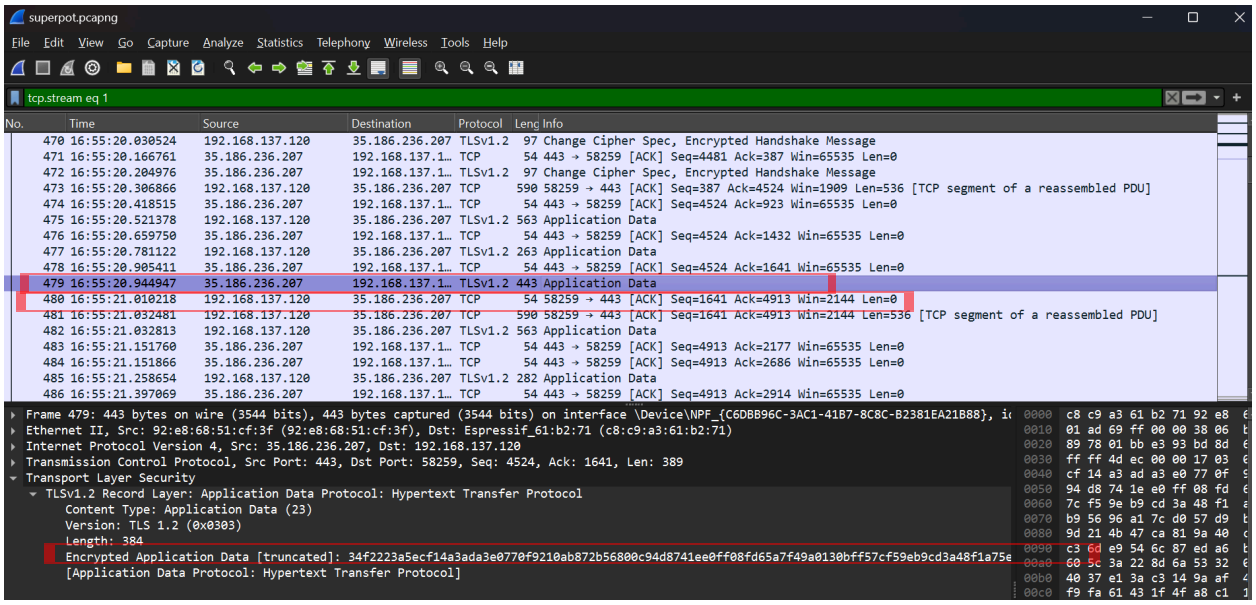
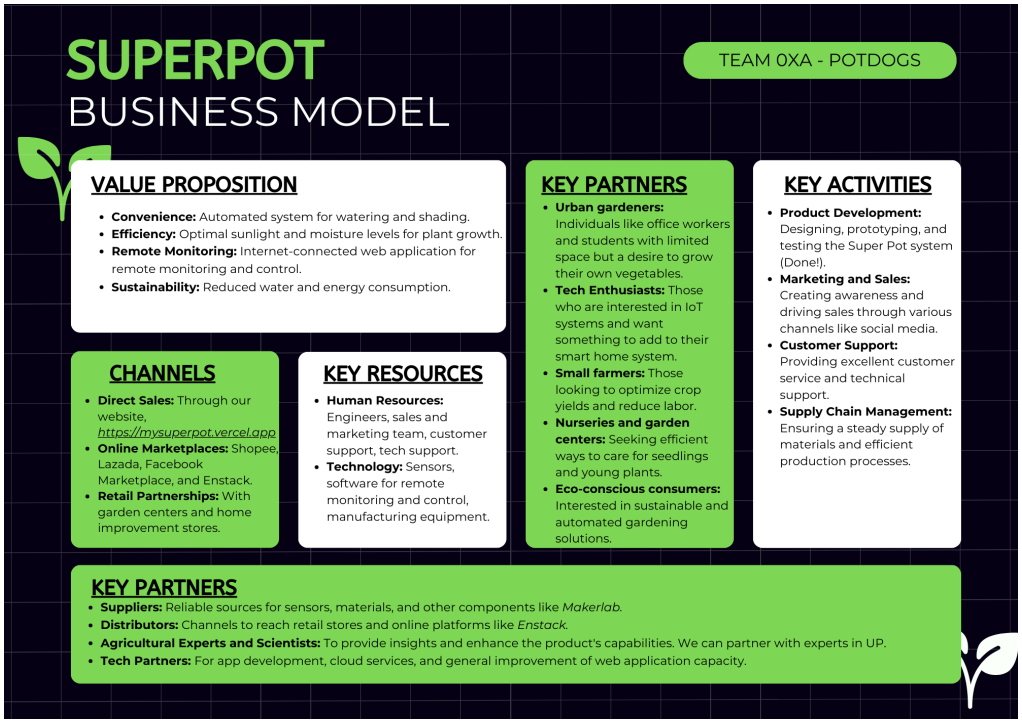


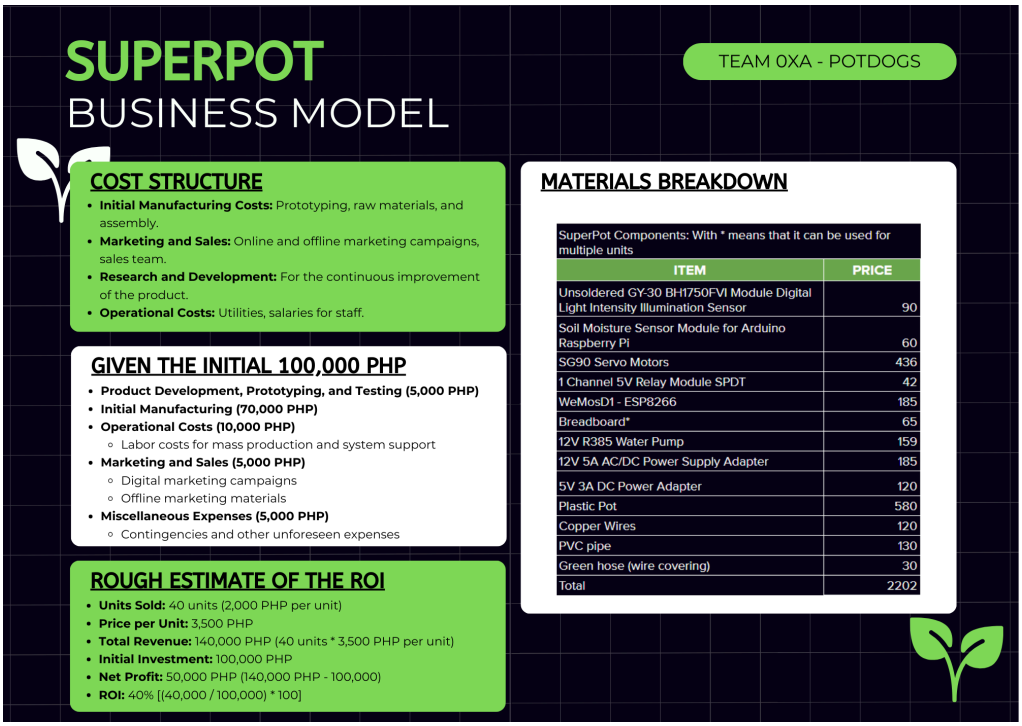
Figure 11. Packet 479 (Application Data) and Packet 480 (Acknowledgement)

As an example for checking whether data is being transmitted from the client to server or vice versa, we can check Figure 11(Packet 479). This packet shows that the server(Source IP: 35.186.236.207) is sending encrypted application data to the client(Destination IP: 192.168.137.120) over a TLS 1.2 connection, hence we cannot see the actual data being sent. The following packet(Packet 480) is an acknowledgment from the client(Source IP: 192.168.137.120) to the server(Destination IP: 35.186.236.207), indicating that the client successfully received the application data sent in packet 479.

5. Bonus

5.1. Business Model





SuperPot Components: With * means that it can be used for multiple units

ITEM	PRICE
Unsoldered GY-30 BH1750FVI Module Digital Light Intensity Illumination Sensor	90
Soil Moisture Sensor Module for Arduino Raspberry Pi	60
SG90 Servo Motors	436
1 Channel 5V Relay Module SPDT	42
WeMosD1 - ESP8266	185
Breadboard*	65
12V R385 Water Pump	159
12V 5A AC/DC Power Supply Adapter	185
5V 3A DC Power Adapter	120
Plastic Pot	580
Copper Wires	120
PVC pipe	130
Green hose (wire covering)	30
Total	2202

5.1. Enstack Bonus Track: Building Trust and Reliability in Logistics and E-commerce

You may visit our Interactive Product Visualization at https://mysuperpot.vercel.app/view.