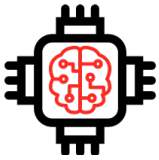
TUHH
Technische Universität Hamburg



GreenHouse Soil Parameter Monitoring IoT system: Server Building

Project Work

submitted by
Md. Sarfaraj Sikder
born in Dhaka, Bangladesh

Smart Sensors
Technische Universität Hamburg

April 2024

1. Reviewer : Prof. Dr.-Ing. Ulf Kulau
2. Reviewer : TBC

Affidavit

I hereby declare in lieu of an oath that I have produced this Project Thesis with the title of ,,GreenHouse Soil Parameter Monitoring IoT system: Server Building" independently and without unauthorized outside help. I have not used any sources or aids other than those indicated, nor have I marked any quotations, either verbatim or in spirit. The work has not been submitted in the same or a similar form to any examination authority.

Hamburg, June 16, 2024
Place, Date

Md. Sarfaraj Sikder

Abstract

Soil health is a crucial parameter when it comes to agriculture. Thus soil quality measurements are important research topics. The Institute of Wastewater Management of TUHH has an Arduino Platform for soil quality measurement system. It uses a CO2 sensor to monitor the CO2 emission rate, pressure sensors with Tensiometers to monitor the water content, temperature sensors to monitor the soil and ambient temperature, a humidity sensor to measure ambient air humidity, and an LCD monitor to display a few results. It is a single unit and contains a data logging shield which is used to log data for later research. However, the limitations of the system include no scalability, no real-time data monitoring, flexibility. Thus an IoT-based smart soil monitoring system has been developed in this project. It facilitates the scaling of the project and reduces unnecessary wiring on the field. Moreover, the user can monitor soil parameters in real time from his laptop when connected to the network over SSH or VNC. A solar power bank or a PV panel can be used to make the system totally off the grid making the system an off-grid standalone IoT device.

Contents

List of Figures

List of Tables

Acronyms

IP Internet Protocol

SSH Secure Shell

sftp Secure Shell File Transfer Protocol

DNS Domain Name System

DHCP Dynamic Host Configuration Protocol

AP Access Point

CSV Comma Separated Value

I2C Inter-Integrated Circuit

MQTT Message Queuing Telemetry Transport

SPI Serial Peripheral Interface

IoT Internet of Things

JSON JavaScript Object Notation

IDE Integrated Development Environment

TCP Transport Layer Protocol

RTC Real Time Clock

NTP Network Time Protocol

PCB Printed circuit board

EMI Electromagnetic Interference

1 Introduction

1.1 Motivation

Soil Health is very important when it comes to agriculture. It determines the growth and quality of the crops. There are many advanced tools for measuring soil parameters available in the market. However, these tools are usually for measuring a single property of the soil, which sums up in using several tools to determine a certain set of properties of soil as per the scope of research interest.

Using several tools makes the research very expensive to conduct. Thus an Arduino-based soil monitoring system was developed by Prashanth[1] in his master's thesis work at the Wastewater Management Institute of Hamburg University of Technology. The open-source Arduino platform has a data logger and sensors mounted on breadboards to monitor soil features. It was both cost-effective and compatible with a lot of sensors. However, the system crashed within 3 years due to a lack of maintenance and proper database. Moreover, it had no scalability option in the field, no flexibility with the sensors as all of the sensors were attached to one Arduino, no real-time data monitoring system, limitation of a good data visualization system, no system to change the sampling rate of the sensors without changing code, a lot of loose wire which caused malfunctioning of the sensors, no protection from rain or bad weather.

Considering all the above issues a step to develop the Arduino system was decided to solve all the issues mentioned. An IoT-based approach was considered to replace Arduino with NodeMCU ESP8266 board, which also uses an open-source platform with built-in wifi. It is lightweight and energy-efficient, which supports C++. Additionally, it works on the Arduino IDE environment with some prior configuration.

1.2 Project Approach

The initial idea of the project was to build an IoT system that is off the grid, energy-efficient, and scalable. Where data can be stored in a remote database and visualized from anywhere using the internet and a computer. Users should also have the flexibility of changing the sample rate of the sensors and be able to configure certain parameters of specific sensors. Moreover, the user will get an automated Comma Separated Value (CSV) output for later research purposes. The bulky wiring for the circuits from the Arduino setup is to be replaced by a PCB design and a 3D case for rain and protection. This approach solves all the issues arising from the current Arduino-based setup. However, firstly the old system was developed according to current needs and modified to run temporarily during the development phase of the IoT system.

As the work was too much for a single project work required by the university, it was divided into server building and sensor building parts. In this paper, only the server building part will be discussed in detail, i.e. the PCB design and 3D case will not be discussed. As shown in figure 1.1, the idea was to separate the database server from the main server handling the IoT system. All the components here are to be connected and hosted over the Eduroam network of the university and through port forwarding the data would be available to the internet. This would

allow the user to connect to the server from any remote location with an internet connection.

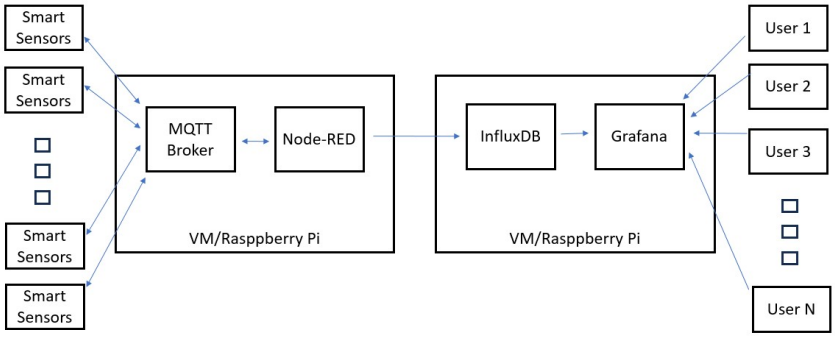


Figure 1.1: Block Diagram of the IoT Server

However, due to budget and security protocols followed by the university and the Eduroam network, a more feasible approach of hosting all the services on a single Raspberry Pi was decided along with hosting an AP from the Raspberry Pi for the sensors to connect. Instead of creating an AP, a gateway or router available in the market can also be used, which would have a higher range of coverage. The local AP of the Raspberry Pi limits the user to only connect remotely to the server when connected in the same local network of the Pi-AP as shown in the figure 1.2.

Initially, during the development phase, the server was hosted with a personal Tplink router. However, when trying with the university network, a lot of issues arose as the Eduroam doesn't approve Espressif devices due to its security protocols. Eduroam uses an individual data encryption level WPA3 which isn't supported by Raspberry Pi or ESP8266 yet. Nevertheless, Raspberry Pi can be connected to Eduroam by providing certain certifications and valid TUHH credentials [15], which again made the process complicated in the long run. There is documentation [11] related to using a Raspberry Pi connected to the internet via ethernet or WiFi and also hosting an AP which allows the connected devices to communicate to the internet. However, this approach is not used in this project.

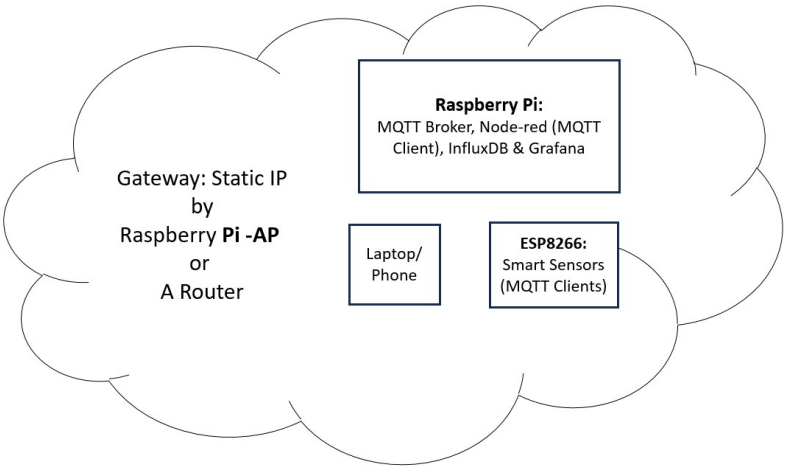


Figure 1.2: Local Network from Raspberry Pi Access Point

2 Old Arduino-based Setup

The previous soil monitoring setup produced by Prashanth [1] had a single unit based on the open-source Arduino platform, where all the sensors were connected along with a data logger shield, making the overall system a bulking and prone to sensor malfunction due to loose connections. Figure 2.1 shows an overview of the system.

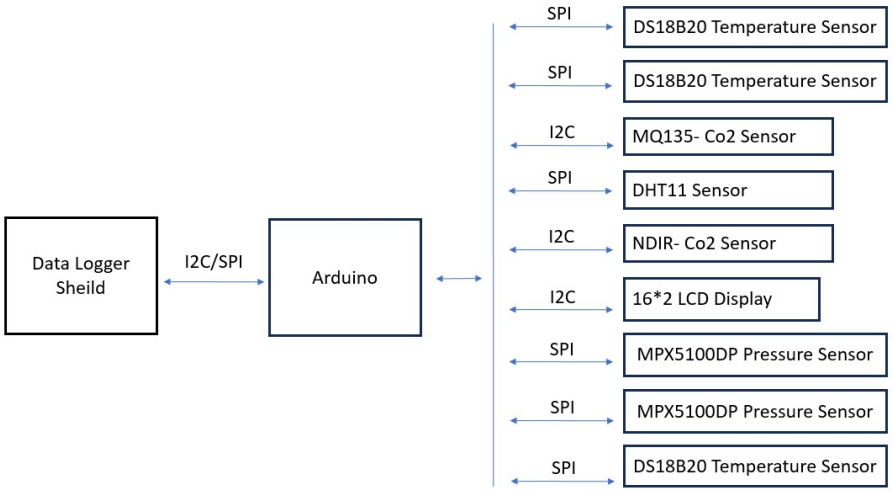


Figure 2.1: Block Diagram of the Old Arduino Setup

Although the system is very economical compared to the cost of the traditional tools used for soil monitoring available in the market, it has the limitations of scaling, no real-time data visualization, no remote connection, and limitation of storage from the perspective of a server. The thesis paper [1] had the sensor configuration code that helped to develop the new system with ESP8266 and debug the issue of system crashing of the old Arduino-based system. Two similar setups, one with Arduino Uno and another one with Arduino Mega were in use. Both of the setups consisted of DS18B20 Dallas Temperature sensors, an SD Card Shield (Data Logger), a 16*2 LCD Display, a DHT11 Humidity sensor, MPX5100 Pressure sensors for soil water pressure monitoring with the help of Tensiometers, and an NDIR gas sensor which used different types of serial communication protocols. Details about the sensors can be found in the thesis paper [1] and on their respective datasheets.

Figure 2.2 shows the overall sketch of how the Arduino system is connected to the soil in the field. Figure 2.3 shows the old Arduino arrangement in a box with the microcontroller, LCD, breadboard, data logger shield, etc. Holes were created on the box to facilitate the wire of the sensors to come out of the box. The figure also shows 3 Tensiometers connected with 3 pressure sensors for calibration purposes. More about the calibration parameters can be found in [16].

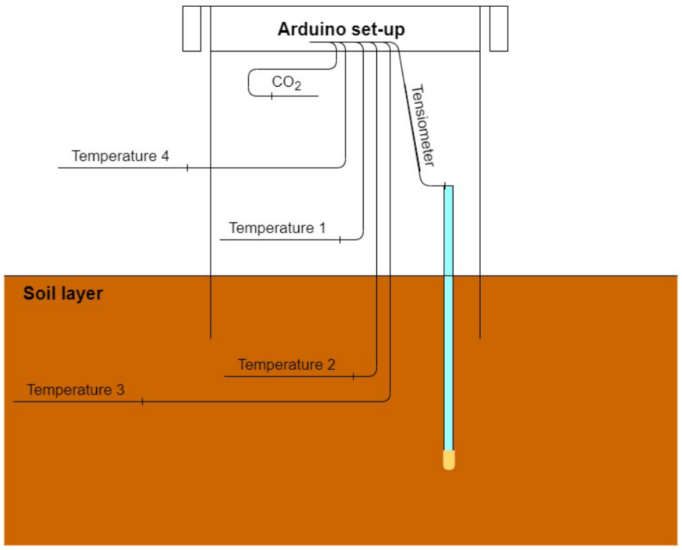


Figure 2.2: Typical arrangement for Soil Monitoring Arduino System[1]

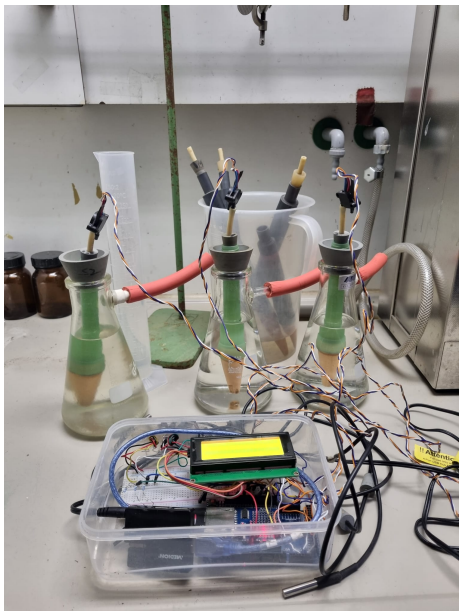


Figure 2.3: Bulky Arduino Setup with Tensiometers

3 New IoT-based setup

As an approach to the IoT system, ESP8266 has been chosen for the project. It is a small and affordable microcontroller that comes with built-in WiFi capabilities making it suitable for IoT projects. It features an energy-efficient powerful 32-bit microcontroller with low-power sleep mode. Moreover, it can be programmed using different development environments and languages including the Arduino IDE. The old setup was also built using the Arduino IDE which helped to keep the previous libraries and modify it accordingly to our needs. The ESP8266 includes several GPIO pins, but it's fewer compared to the Arduino board or its successor ESP32. However, it solves our purpose of creating individual smart sensors which require only one sensor to be connected to the board. It needs to be mentioned that it has good community support which is very useful for different runtime issues.

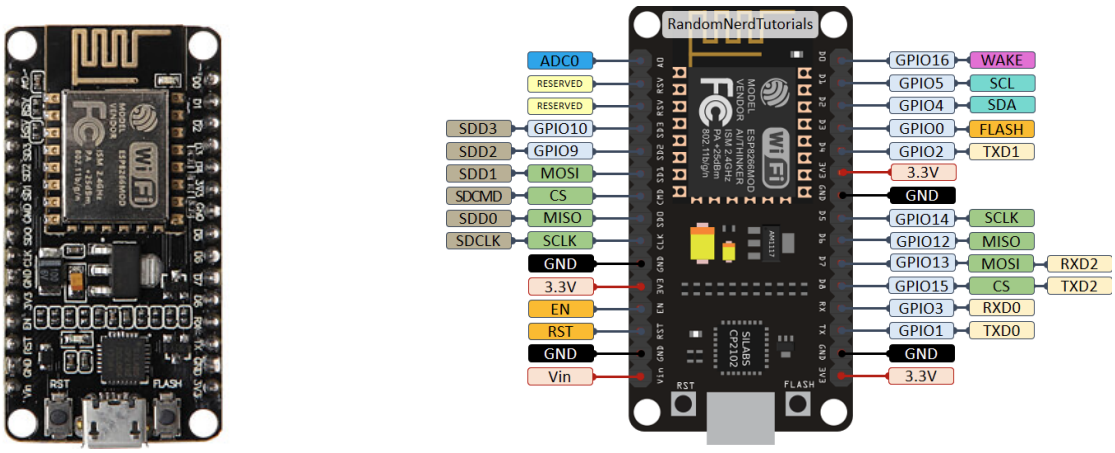


Figure 3.1: ESP8266 Microcontroller and Pinout Diagram [2],[3]

Figure 3.1 gives an overview of the GPIO pinout diagram and its physical appearance (Dimension: 49.3 x 26.3 x 13 mm). It can be powered by a micro-USB connecter from an adaptor/power bank or also from a laptop. The operating voltage is 3.0-3.6V with data memory of 96KB and internal memory of 64KB. The WiFi module operates at the frequency of 2.4 GHz which is compatible with every open-source platform such as Raspberry Pi or Arduino.

The smart IoT device is created using the lightweight Message Queuing Telemetry Transport (MQTT) Protocol to communicate with other devices over the network, while the connection between the ESP8266 board and the sensors uses different types of serial communication. As per discussion with the user, we decided to keep only 5 Pressure sensors, 6 Temperature sensors, and 1 Humidity sensor for the current demand and discard the other sensors used in the old setup.

Figure 3.2 demonstrates how devices are connected in the local network and communicate in real-time remotely. Raspberry Pi is chosen as the main server as it is small and yet very powerful. The physical dimensions are 85.6 mm x 56.5 mm x 20.32mm. It has very good documentation

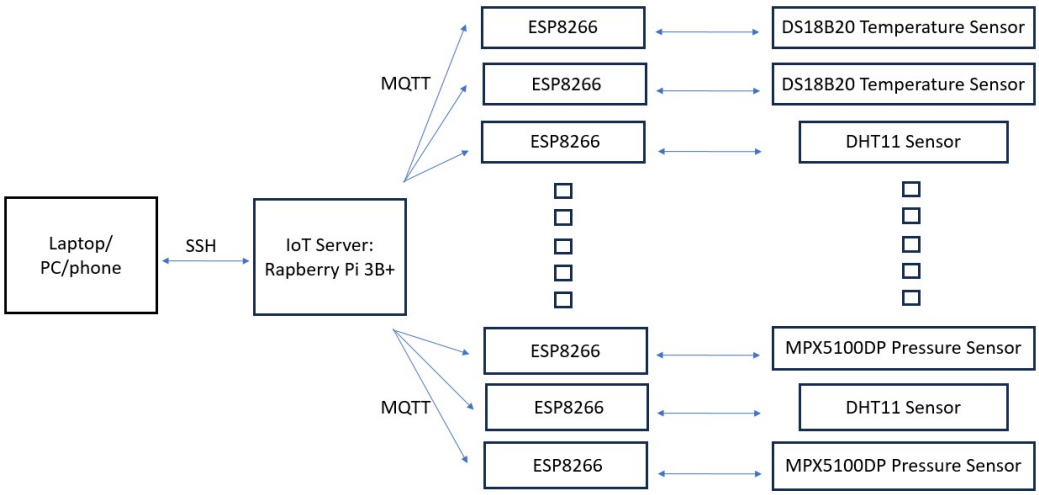


Figure 3.2: The Soil Monitoring IoT system

[11] and a user community which is of great help for doing project work. A laptop/computer or even a personal phone can be used to connect to the local AP and using any browser with the right Internet Protocol (IP) port and credentials, anyone can access or visualize the data. Raspberry Pi 3B+ has been used as it has all the new features including a ethernet port and built-in Wi-Fi. It also has the option for a user to connect a monitor, mouse, and keyboard directly if connection over Secure Shell (SSH) is not preferred. Raspberry Pi requires a Linux-based operating system based on the ARM processor that has to be installed and put in the SD slot to run the microcontroller. Figure 3.3 gives a visual overview of the different slots present on the front and back sides of the board.

Figure 3.3: Raspberry Pi 3B+ Front and Back side [4],[5]

Figure 3.4 gives a detailed pin-out diagram of all the GPIOs. We need the information later when setting up an Real Time Clock (RTC) module for time synchronization.

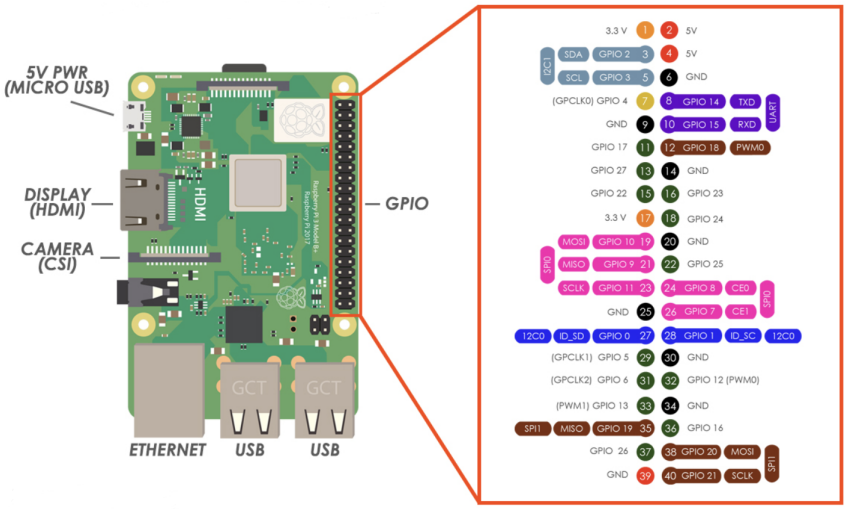


Figure 3.4: Raspberry Pi GPIO[6]

3.1 Embedded Sensors

The Project uses three different types of sensors which were used in the old Arduino setup. The pressure and temperature sensors are the main sensors used in the project along with one humidity sensor. There are varieties of pressure and temperature sensors available in the market. However, the DS18B20 Dallas Temperature sensor is widely used for the soil monitoring system as it comes with a cable and a good protective cover from soil moisture. The MPX5100 differential pressure sensor was chosen along with a tensiometer built by Prashanth in his project work [16]. Both the temperature sensor and the pressure sensor are well known for accurate results and resolution. A brief description of the sensors is given below.

3.1.1 DS18B20 Temperature Sensor

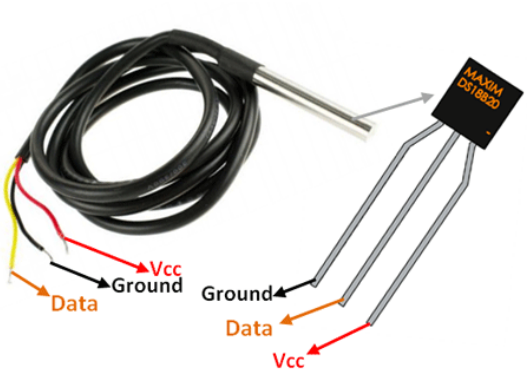


Figure 3.5: DS18B20 Temperature sensor [7]

This temperature sensor 3.5 works on the one-wire protocol developed by Dallas Semiconductor. More relevant information about the sensor can be found in the datasheet [17] as well.

Table 3.1: DS18B20 Temperature Sensor Pin Connection with ESP8266 Using BreadBroad

No.	DS18B20 Pin Name	Description
1	Ground	Connected with the ground pin of ESP8266
2	Data	Connected to the digital pin D2 of ESP8266
3	VCC	Connected with the 3.3V pin of ESP8266

Table 3.2: DS18B20 Temperature Sensor Pin Connection with ESP8266 Using PCB

No.	DS18B20 Pin Name	Description
1	Ground	Connected with J3 GND pin of PCB
2	Data	Connected with J3 D0 pin of PCB
3	VCC	Connected with J3 3.3V pin of PCB

These sensors require an external pull-up resistor to hold their value high to 1 to avoid undefined logic gates. It can be connected to the ESP8266 in two different ways, namely the normal mode and the parasitic mode. For the project, a 4.7K ohms pull-up resistor is used in the normal mode. Figure 3.6 shows the difference between the normal and parasitic connection.

A PCB was developed by a colleague, Anirudh, working on the sensor part of the project [14]. The PCB includes a pull-up resistor and thus another external pullup is not required. It's only required in case of using a breadboard. Table 3.2 describes how the sensors are connected to the PCB. Here the normal connection mode is used. For more information refer to the appendix chapter for the schematic diagram A.4 of the PCB design.

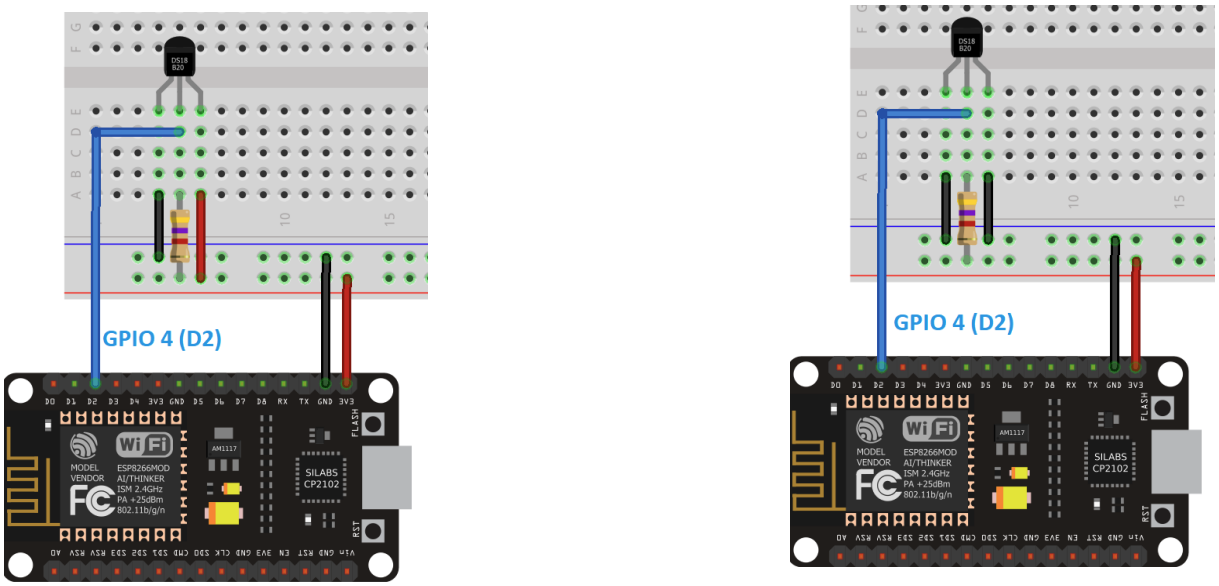


Figure 3.6: DS18B20 Connection Diagram: Normal Mode vs Parasitic Mode [8]

3.1.2 MPX5100DP Pressure Sensor

MPX5100DP 3.7 is a differential pressure sensor i.e. it measures the difference of pressure passed to the tube connections shown in the figure. There is a notch in the 1st pin for easy recognition.

This sensor is further connected to a tensiometer for monitoring the soil water pressure. Details about the sensor connection with the Tensiometer and its working principle can be found in the project work of Prashanth [16]. Information on the sensor can also be found in its datasheet [18].

Figure 3.7: MPX5100 Differential Pressure Sensor and Pinout Diagram [9]

An important thing to notice about the MPX5100DP sensor is that it requires a 5V power supply. Initially, it was powered by the Arduino board which had a 5V supply pin. However, ESP8266 doesn't have a supply pin higher than 3.3V but it gives close to 5V in the Vin pin when connected from a source of 5V. We can feed this to the pressure sensor but there is a risk of damaging the ESP8266 microcontroller. Despite this difficulty, we decided to keep this sensor instead of buying different ones with 3.3V input power. Thus a separate power supply is recommended for the sensor to function properly along with a voltage divider, as ESP8266 has a max ADC rating of 3.3V.

Alternatively, we can also run it from 3.3V with a lower resolution by changing and considering some parameters from the datasheet in the code. Refer to the appendix chapter for the code details. A few important parameters considered are described below in the table 3.4. Table 3.3 gives an outline for the sensor connection to an ESP8266 board with breadboard.

Table 3.3: MPX5100 Pin Connection with ESP8266 Using Breadboard

No.	MPX5100 Pin Name	Description
1	Ground	Connected with the ground pin of ESP8266
2	Analogue Output	Connected to the analogue pin A0 of ESP8266
3	VCC	Connected with the 3.3V pin of ESP8266

Table 3.4: MPX5100 Sensor supplied from different voltages

No.	Operating Characteristics	5V Supply	3.3V Supply
1	Offset Voltage	0.2V	0.132V
2	ADC Voltage	(Analogue value * 5)/1023	(Analogue value * 3.3)/1023

The PCB developed by Anirudh [14] has both of the provisions of powering the sensor with 3.3V and 5V as requested. If it's powered by 5V it needs to be passed by a voltage divider to the

ADC PIN. Refer to the schematic diagram A.4 in the appendix chapter for better understanding. J1 and J2 Jumper connections need to be adjusted so that the right supply voltage is provided to the sensor depending on the external or internal power supply.

Table 3.5: MPX5100 Pin Connection with ESP8266 Using PCB

No.	MPX5100 Pin Name	Description
1	Ground	Connected with the J4 ground pin of PCB
2	Analogue Output	Connected with J4 Vout pin of PCB
3	VCC	Connected with J4 Vs pin of PCB

Table 3.5 describes how the sensor is connected to the PCB. Moreover, to connect with the 3.3V as supplied to the sensor by the ESP, J1 requires pins 1 and 2 to be shorted and J2 requires also pins 1 and 2 to be shorted on the PCB.

3.1.3 DHT11 Humidity Sensor

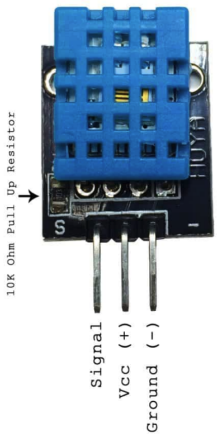


Figure 3.8: DHT11 Sensor with Pin Diagram[10]

A digital humidity and temperature sensor is considered as well for monitoring the humidity and the ambient temperature of the greenhouse. This is a small sensor that also uses a one-wire protocol like our temperature sensor. Details about the sensor can be found in its datasheet[19]. Table 3.6 gives an overview of the sensor connection to an ESP8266 board using the breadboard.

Table 3.6: DHT11 Pin Connection with ESP8266 using Breadboard

No.	Pin Name	Description
1	Ground	Connected with the ground pin of ESP8266
2	Data	Connected to the digital pin D4 of ESP8266
3	VCC	Connected with the 3.3V pin of ESP8266

Table 3.7 describes how DHT11 is connected to the PCB. Although with the breadboard, we were flexible to use any digital pin of the ESP instead of D4 as mentioned in table 3.6, with the PCB we must use D1 to connect to the data pin of the sensor. Because D1 is connected directly to the D5(GPIO14) of the ESP8266 and D0 is connected to D6(GPIO12) of the ESP8266

Table 3.7: DHT11 Pin Connection with ESP8266 Using PCB

No.	Pin Name	Description
1	Ground	Connected with J3 GND pin of PCB
2	Data	Connected with J3 D1 pin of PCB
3	VCC	Connected with J3 3.3V pin of PCB

attached to a pull-up resistor. The DHT11 we are using comes with a built-in pull-up resistor and thus it doesn't require any external resistor to function. Refer to figure A.4 for a better understanding of the circuit.

3.2 Setup Arduino IDE for ESP8266

ESP8266 is supported by the Arduino Integrated Development Environment (IDE). This is an open-source Arduino software that makes it easy to write code, compile, and upload it to the ESP board. To run ESP8266 on this IDE there are some steps to be followed [20].

First of all the Arduino IDE has to be downloaded and installed from the official site. After installation, the preference window needs to be opened from the options and the following URL needs to be entered in the additional board manager URLs field:

URL:"http://arduino.esp8266.com/stable/package_esp8266com_index.json". Then Esp8266 from the Boards manager needs to be installed. It's found inside the Board option in Tools. Then the right board has to be selected from the list. For this project, the NodeMCU 1.0 (ESP-12E Module) is selected.

Finally, different libraries need to be installed for the project to run. A list of used libraries with a brief description is provided in the appendix chapter. Also all the library files are uploaded to the TUHH GitLab repository.

3.3 Raspberry Pi Setup

The Raspberry Pi server is a Linux-based operating system. The following components mentioned in the table3.8 are needed for the project. A Raspberry Pi can be set headless or with a mouse, keyboard, and monitor. The recommended OS from the Raspberry Pi imager can be chosen to install and boot it on an SD card. One can also find all the required steps for installation in the good official documentation site[11]. For the project, Raspberry Pi OS (Legacy, 32-bit) has been chosen, which comes with a desktop environment that keeps the flexibility of using it with a screen if preferred. However, a lite OS version fits well with our project. As of writing the paper, a personal Raspberry Pi was used for setting up the server during the test phase.

The following credentials: "user: sikder, Password: Sarfaraj, hostname: raspberrypi.local" on the Raspberry Pi imager configuration window are given for the installation of the OS. Once the SD card is ready, eject it safely and insert it into the SD slot of the Raspberry Pi. The first boot usually takes a little bit of time to fire up the Pi. During the OS installation, preferences for internet connectivity and authentication credentials are needed. For the project, all of the next steps are done headless over an SSH connection. For establishing an SSH connection with keys, the user should have an already generated pair of private and public keys and the public key should be provided to the remote host (in our case Raspberry Pi) for connection. An SSH

connection is also possible via password credentials. Refer to online documentation for SSH connection steps.

Table 3.8: Raspberry Pi 3B+ Server Component List

No.	ITEMS	APPOX. UNIT PRICE (EURO)
1	Raspberry Pi 3b+	38.65
2	Memory card 32GB	5.95
3	5V/2.5A Power adaptor for RPi 3b+	12.4
4	Raspberry Pi 3b+ Aluminium Case	13.20
5	Ethernet Cable (3m) for direct connection over LAN	6.5
6	RPI RTC DS3231SN LAN	4.60

3.4 Setup Raspberry Pi Access Point

As mentioned before the initial approach of the project was to host a server and make it accessible over the Internet. However, due to the security protocols followed by the university Eduroam network, this wasn't possible. Thus we proceeded with the idea of creating our local network and connecting the smart sensors over the MQTT protocol.

This local network can be created by using a router or opening an AP from Raspberry Pi itself. The 2nd option was chosen for the project. There is a lot of documentation available on the internet for creating an access point in Raspberry Pi. Figure 3.9 describes how a Raspberry Pi acts as a WLAN AP and creates a local network with a new set of IP addresses. The Pi here is connected to the internet via ethernet cable. For our use case, we will not be connecting to the Eduroam at all, rather, we will only use the hotspot AP from the Pi and connect our devices to communicate via MQTT protocol without any internet.

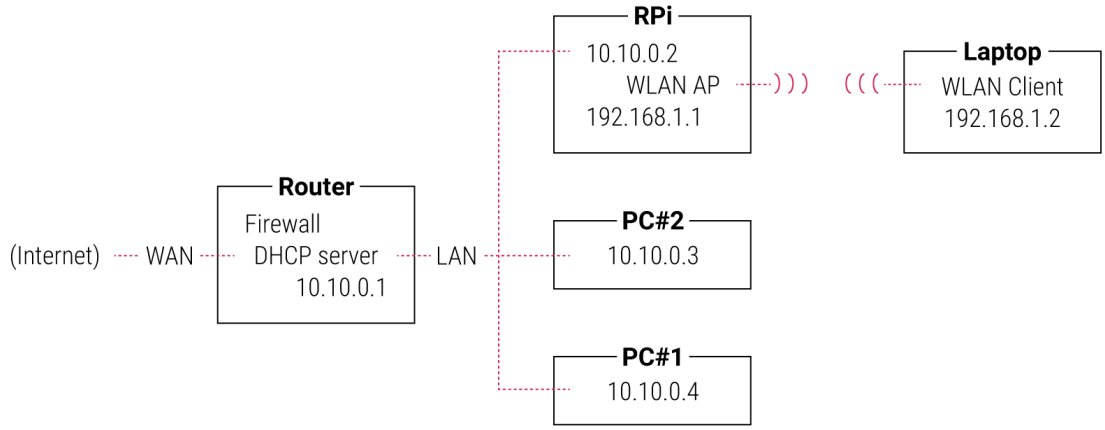


Figure 3.9: Raspberry Pi WLAN AP[11]

The Raspberry Pi is configured with the following details as in table 3.9. Hostapd daemon has to be configured for broadcasting our SSID and allowing WiFi connections on a certain channel. To set a static IP address for the WiFi interface we have to configure "interfaces" file in the root directory "etc". Dnsmasq daemon also needs to be configured to automatically assign mentioned IP addresses to the newly connected devices on the AP. Configuring these two

conf files will create our WiFi AP. Follow the steps in the Raspberry Pi official documentation page [11] to have a better understanding of it.

Table 3.9: AP configuration details

SSID	MyPiAP
AP Password	raspberry
Static IP for Pi	192.168.5.1
DHCP range for the smart sensors	192.168.5.100 - 192.168.5.200

3.5 Setup RTC Module

In any IoT system, accurate timestamps are crucial. Raspberry Pi doesn't have a real-time clock within itself. It uses the Network Time Protocol (NTP) to sync to the real-time when connected to a NTP server in the local network or over the internet. We have decided not to use the internet due to the complicacy with the Eduroam network, so we need an external RTC module. RPI RTC DS3231 has been chosen for the project. Any other RTC would suffice for our use case.

Figure 3.10: RPI RTC DS3231SN[12]

This module works on the Inter-Integrated Circuit (I2C) protocol. The connection with the Raspberry Pi is described in the table 3.10. Refer to figure 3.4 for a better understanding of the pin connection. Note that there is an extra slot in the RTC which has no contact with the Rpi GPIO.

Table 3.10: Raspberry Pi RTC DS3231SN Pin connection

No.	Raspberry Pi PIN	RTC PIN
1	PIN 1	Power supply 3.3V
2	GPIO 2 or PIN 3	I2C SDA
3	GPIO 3 or PIN 5	I2C SCL
4	PIN 9	Ground

The installation process is quite simple. A detailed guide can be found in [21]. So first, we need to connect the RTC module and install it so that our Pi server knows how to collect time

and date from the RTC instead of NTP. Figure 3.10 shows the time and date of the RTC module after it is synchronized with the real-time collected from NTP in the Raspberry Pi.

```
sikder@raspberrypi:~ $ date; sudo hwclock -r
Thu 30 May 09:32:40 CEST 2024
2024-05-30 09:32:40.155855+02:00
```

Figure 3.11: Sync Internet Time to RTC

3.6 Setup Crontab

Although we are getting all the data in the influxDB and monitoring them in real-time, a daily export of data was necessary for later analysis by the user. An automatically generated CSV formatted output file is required for this case.

Linux offers a built-in task scheduler called Crontab. A cron job is scheduled to run a bash script 'collect_data.sh' regularly at midnight which produces the output file as 'Output_current-date.csv' in the home directory inside the directory 'CSV_files' of the Raspberry Pi. The executable 'collect_data.sh' script is set to query our database mentioning the name of the database and measurement for the last 24 hours and creating the mentioned Output file. The script needs to be made executable to run with the "chmod" command. Refer to the appendix chapter to find the collect_data.sh script and crontab details. If there is no existing cronjob, open it with the command "crontab -e" and set the file as shown in the appendix chapter.

4 IoT Docker Containers Configuration

4.1 Install git

The first approach was to install all four services on the host machine: Mosquito, Node-red, InfluxDB, and Grafana on the Raspberry Pi to explore how it works and if the setup will be effective for our use case. As it has been determined that we will proceed with these services and they serve our purpose, we decided to run them in docker containers. Running them as a docker container gives us more flexibility in deploying the project in any other environment later without additional configuration.

While doing some research, an IoT stack tutorial[22] repository in GitHub was found. This repository has exactly the 4 containers needed for the project. So, to access it in the Raspberry Pi, first, install git from the terminal and then git clone the repository. Git is a distributed version control system (DVCS) that allows developers to track and change code during software development. The GitHub repository has a docker-compose yaml file with a basic configuration of the four containers. Yaml is a human-readable data serialization language simply used to configure services. It has been changed according to our need for the project.

4.2 Install Docker and Docker-compose

Now to run, configure, and manage containers; Docker and Docker-Compose need to be installed. Docker is a platform for developers that allows them to develop, transfer, and run applications in containers; as if the services are running in their physical infrastructure. Containers help to package any application along with dependencies into a single unit, which is isolated from the underlying infrastructure. Exposing ports and mounting volumes to the host allows the services to communicate within themselves.

Figure 4.1 shows how the server looks from a higher level. The containers are well-secured and separated from each other. The AP service and the Cron service will be running outside docker containers i.e. directly on the host machine. These are kept outside of the docker containers as it is specifically needed for the server when it is a Raspberry Pi. When the system is run in a different environment like Windows or any other Linux-based computer, the AP can be created from the router. Also, the Cronjob is a configuration file to schedule a job for a Linux operating system. Here, it is to only fetch data and store it in a CSV file automatically. The same thing can be done in different ways in different environments. Refer to the appendix chapter to learn more about the docker-compose.yaml file and dockerfile, which is used for the project configuration. The RTC module is required to sync time and provide the correct time to the docker services.

With the help of the public GitHub repository from IoT-stack-tutorial and configuring the docker-compose.yaml file according to the need, the containers are ready under the home directory /GreenHouse. Now using the command "docker-compose up -d", the containers will be set up and running as a daemon in the background. It will download all the required images and set the dependencies mentioned in the yaml file. To check if the containers are up and running

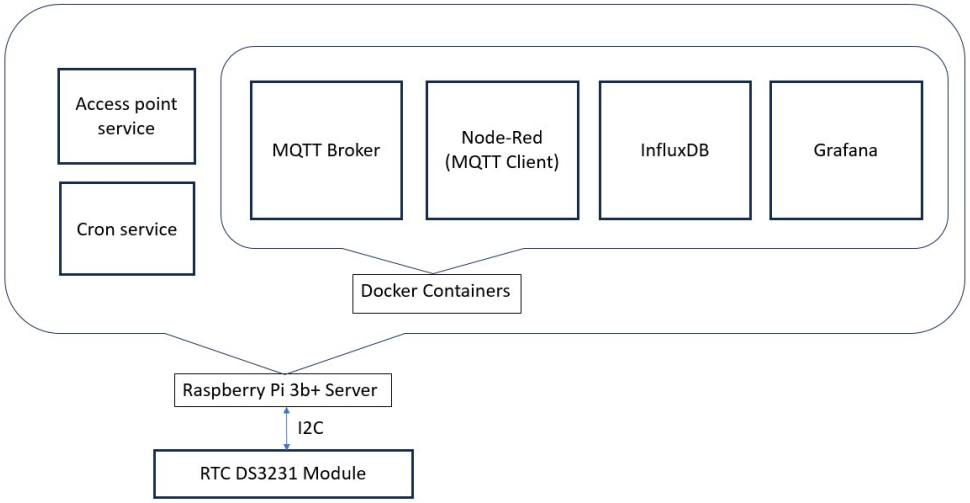


Figure 4.1: Block Diagram of Server Infrustructure

use the command "docker ps" externally or "docker-compose ps" internally from the directory /GreenHouse.

```
sikder@raspberrypi:~/GreenHouse $ docker-compose ps
  Name                Command                  State                          Ports
-------------------------------------------------------------------------------------------------------
grafana     /run.sh                          Up             0.0.0.0:3000->3000/tcp
influxdb    /entrypoint.sh influxd           Up             0.0.0.0:8086->8086/tcp
mosquitto   /docker-entrypoint.sh /usr ...   Up             0.0.0.0:1883->1883/tcp, 0.0.0.0:9001->9001/tcp
nodered     ./entrypoint.sh                  Up (healthy)   0.0.0.0:1880->1880/tcp
```

Figure 4.2: Containers Running on the Pi Server

Figure 4.2 gives a detailed list of the containers including name, initial command, state, and the exposed ports with the used communication protocol.

4.3 Setup Mosquitto Broker with Authentication

Eclipse Mosquitto is an open-source message broker that implements the MQTT protocol. It allows devices and applications to use the publish/subscribe mechanism and communicate with each other. It acts as a middleman to collect messages from the publishers and deliver them to its subscribers. It's a two-way communication system where messages can be sent and received in topics.

The broker is the central point of all the data that ESP8266 and Node-RED are transferring. The broker here is running inside the Pi in a container. Node-RED is also subscribing to topics for changing the sample rate. Thus both Node-RED and ESP8266 are publishing and subscribing to certain mentioned topics for the IoT system. For this reason, the broker has to be secured so that no other device can send data to our topics without permission.

Thus the devices have to connect to the broker with password authentication and the file needs to be also hashed so that it's not human-readable. The official documentation[23] page

from Eclipse Mosquitto has been used as a guide to creating a password file. Running and authenticating Mosquitto in the host machine is easy. However, doing the same inside a container is a bit tricky. We have to create the password file inside the container and mount the directory to the host machine. Otherwise, the devices will run into an error while trying to connect to the broker with the credentials set. Table 4.1 shows the credentials required for the devices to connect to the broker. The latest version of the Eclipse Mosquitto image is taken from the docker hub as a base. Check the appendix chapter for the mosquito configuration file.

Table 4.1: Mosquitto Broker Authentication Credentials

User	Password
ESP	IoTsensor

4.4 Setup Node-RED Flow

Node-RED is a visual tool for wiring or connecting the IoT components as nodes. It has a user interface to check the outputs of data in different formats and also interactive switches can be used to pass commands to the nodes. It also provides a web browser-based flow editor, allowing users to create JavaScript functions. More information about Node-RED can be found in [24].

Node-RED image 3.1.3 from the docker hub is used as the base image for the project. Node-RED is only used for connecting the IoT components and sending data to the database. Although it provides the facilities to visualize the data in a graphical user interface, Grafana has been chosen for the data visualization due to more flexibility provided there.

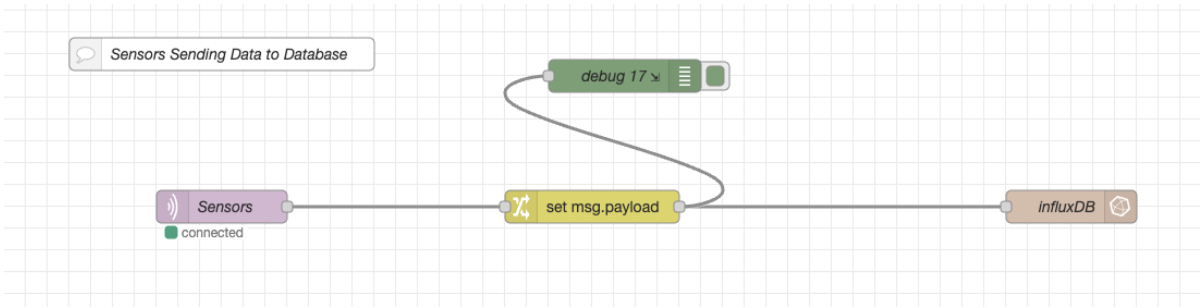


Figure 4.3: Node-RED Flow for the IoT system

Figure 4.3 shows the connecting node flow diagram for the IoT system. By default, the platform doesn't have the palettes installed. The required palettes can be installed to use the nodes from the setting option. For the project, a few palettes mentioned in the table 4.2 have been installed. Refer to the appendix chapter for the flows.json file. In the figure 4.3, the MQTT-in node namely "Sensors" is subscribed to the topic "device/Sensors" to receive sensor data from the ESP8266 sensors. All the ESP8266 sensors are publishing data on the same topic in a JavaScript Object Notation (JSON) format. The change node namely "set msg.payload" is responsible for sorting it out, naming it properly, and passing it to the influxDB out node. The data is stored in the database with this node.

Figure 4.4 shows the edit panel of the change node where the key-value pair is set as a javascript expression. It takes the values from the 12 sensors and sets the appropriate name before sending it to the database. Users can connect to the Node-RED dashboard by visiting the

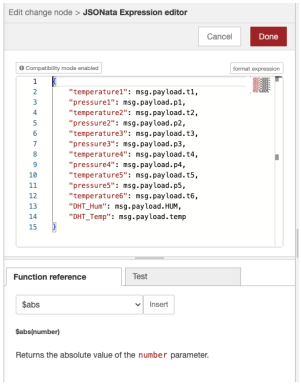


Figure 4.4: Edit Panel of the "set msg.payload"

site http://192.168.5.1:1880/ when connected to the AP. 1880 is the default port of Node-RED. When connected directly to the Raspberry Pi with a monitor, the user can replace the IP with "localhost". Users can set the desired sensor configuration from this site, which will be discussed later in detail.

Table 4.2: Required Palettes for Node-RED

No	Palettes
1	dashboard-evi
2	node-red
3	node-red-contrib-influxdb

4.5 Setup InfluxDB

InfluxDB is an open-source time-series database. It can handle high-volume of automatically time-stamped data. It is moreover optimized for storing, visualizing, and querying time series data. Usually, it is a great choice for IoT devices as they generate time-series data as output. It has a SQL-like query language and also has retention policies for better database management.

For the use case of our project, first, an InfluxDB database is created without any authentication from inside the container and later is linked to the function node in Node-RED to store all the data that is received from the MQTT-in nodes. More detailed information about InfluxDb and its query commands can be found in [25]. InfluxDB version 1.8.10 is used for the project. By default, influxDB uses the server's local nanosecond timestamp in UTC. To see the time stamp in a better human-readable manner it can be changed to RFC3339 format. The external RTC plays an important role in setting the correct timestamp on the system.

4.6 Setup Graphana Dashboard

Grafana is also an open-source analytics and visualization platform used for monitoring real-time data. It has a lot of options to optimize and arrange data. It allows us to add a database and query from it, visualize the data, and put thresholds and alerts depending on the use case. It also allows to choose the time frame for the data to be displayed. It has a built-in authentication

system and the default user and password are "admin" and "admin". The password was later changed to "iotsensor".

Once logged in to the Grafana page from a browser, one can design dashboards according to one's needs. The best thing is that it allows a great scope of flexibility and allows users to visualize the data in whatever form they want. The latest Grafana image is used from the docker hub as our base image for the project. Grafana also has good documentation [26] which helps in all the setting up and preparing the dashboard from scratch. Users can connect to the Grafana dashboard by visiting the site http://192.168.5.1:3000/ when connected to the AP. 3000 is the default port of Grafana.

Figure 4.5: Data Visualization with Grafana

Figure 4.5 shows a snippet of data visualization quality and effects provided by Grafana, during the testing phase of a few sensors sending data through two ESP8266 microcontrollers.

4.7 Data Sampling

Data Sampling is an extra facility for the user to choose the desirable rate of data. This is again done by the subscribe/publish feature of MQTT. It helps to communicate back to the smart sensors and ask them to change the rate of sampling without even changing the code from the microcontrollers. In figure 4.3, the MQTT-out node namely "config Temperature Sensor", "config Pressure Sensor" and "config DHT11 Sensor" is publishing data to the topic "config/sampleRate/Temperature", "config/sampleRate/Presuure" and "config/sampleRate/DHT11" respectively which is received by the ESP8266 sensors. Here by default, at boot, the ESP8266 is set to a default sample rate of 10secs. With the EEPROM of the ESP8266, the changed sample rate by the user can be stored in non-volatile memory and retained when power is lost. Every ESP board has a limited number of write cycles after which it will start to lose its efficiency and wear off. However, EEPROM facilities have not been configured in the smart sensors here.

Figure 4.6, 4.7 and 4.8 shows the inject node with the "sampleRate" information for all the sensors and with pressure sensor it includes the configuration for the slope and the intercept to be set dynamically by the user. The slope and intercept need to be determined by calibrating the Tensiometers with the pressure sensors. The inject node is sending the payload as a JSON object in a key-value pair. To change the sample rate to the desired value, first, the user needs to edit the inject node and put the value in ms. Then once the node is changed, it has to be deployed. After successfully deploying the node, the inject node needs to be triggered for the

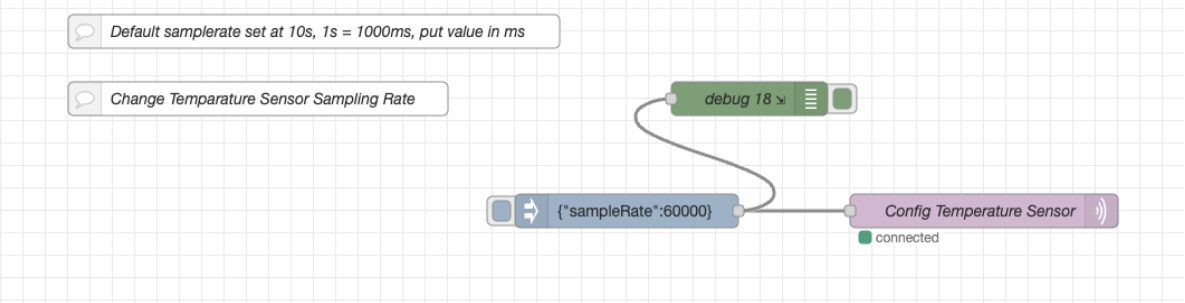


Figure 4.6: Configuring Sample Rate of Temperature Sensor

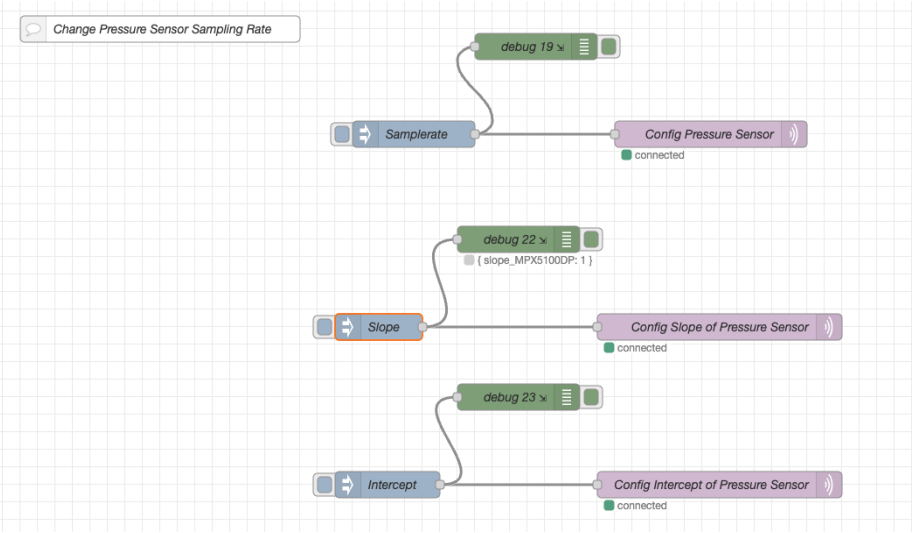


Figure 4.7: Configuring Sample Rate, Intercept, and Slope of Pressure Sensor

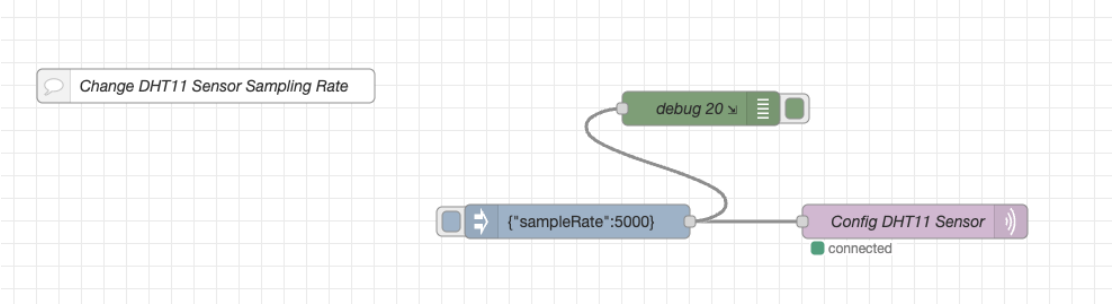


Figure 4.8: Configuring Sample Rate of DHT11 Sensor

change to effect.

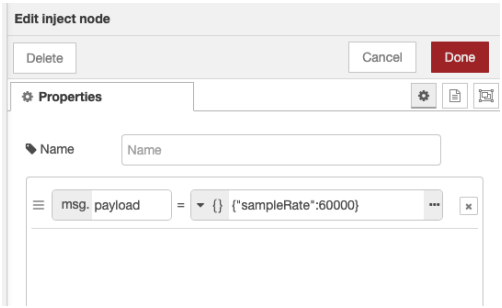


Figure 4.9: Edit Inject Node for Sample Rate

Figure 4.9 gives a visual aid and a description of configuring the inject node edit panel. By double-clicking on the node to get the edit panel, the value from the rectangular box can be directly changed or the values can be edited by clicking on the three dots to go inside. Finally, save the value by clicking "Done". Following the same procedure the configuration of the intercept and slope can be achieved for the pressure sensor.

5 Network Protocols

Some of the important network protocol that is used in the project is briefly described below. There is plenty of information available on the net to learn about them in detail.

5.1 SSH Protocol

SSH stands for Secure Shell. It is a secure method of connecting remotely and sending commands or data over an unsecured network. It uses public and private encrypted keys to connect to a device making it secure. It also supports password credentials. SSH usually uses Transport Layer Protocol (TCP). The default port for SSH is port 22. It's a very common trait to use SSH Tunnel to host a server over the internet. Our project can be further developed by using an SSH Tunnel to host it over the internet.

5.2 IP

IP stands for Internet Protocol. An IP address is a unique numerical mapping assigned to each device on a network. It is the physical address of the device and can be compared to the house numbers on a street. IP addresses are used by devices for communication over the internet. There are two types of IP addresses, namely static and dynamic.

A static IP is a fixed IP for a device, which needs to be set manually. Static IP addresses are often used for servers or network devices that require permanent addresses. On the other hand, dynamic IP is set randomly by the Dynamic Host Configuration Protocol (DHCP). In order to host a server over the internet, it is important to have a static IP. In our project, the Raspberry Pi is hosting an AP with a static IP. Then it assigns dynamic IP in the mentioned range to the newly connected devices. Routers and network devices use the IP address to determine the most efficient path for the packets to reach their destination.

Ip addresses can be also categorized into IPv4 and IPv6. These are version 4 and version 6. IPv4 is a 32-bit numerical address expressed in dotted-decimals notation. It is the most common notation found in daily use. IPv6 is more advanced with a 128-bit hexadecimal address expressed in colon-separated notation.

5.3 Network Time Protocol

The network time protocol is a networking protocol used for synchronizing the clocks to give real-time. It works with the server-client architecture. When a client is connected to a network, it fetches time from the server and syncs with it to give the exact time. In the physical world, there are different NTP servers across the world, and all the clients are connected to them to sync time when it boots and is connected to the internet. By default, Raspberry Pi also uses this protocol.

5.4 DNS Protocol

DNS stands for Domain Name System. It is a human-readable naming system used to label websites and other internet resources. Domain Name System (DNS) is used to translate complex IP addresses to names. For instance, instead of remembering and writing the IP addresses of different websites, we use their domain names to connect to them. It is more convenient and user-friendly. For example, google.com is the domain name. As we write the domain name in the web browser DNS translates it to a numerical IP address using an IP database.

5.5 DHCP Protocol

DHCP stands for Dynamic Host Configuration Protocol. This network protocol is used to dynamically assign IP addresses and other network configuration parameters to a connected device on the network. It is an automatic process that makes life easier for a network engineer. DHCP operates based on the message exchange process between DHCP clients and servers.

In the local AP hosted from the Raspberry Pi, Dnsmasq is used. It is a lightweight DNS forwarder and DHCP server designed for small networks. In short, it provides DNS resolution and DHCP services for the AP.

6 Communication Protocols

Some of the important communication protocol that is used in the project and in the Arduino system is briefly described below.

6.1 I2C Protocol

I2C stands for Inter-Integrated Circuit. It is a serial communication protocol used to connect ICs and peripherals in embedded systems, microcontrollers, etc. It is a half-duplex communication and supports clock stretching. This communication takes place using a data line (SDA) and a clock line (SCL). In the thesis work by Prashanth [1], this protocol has been explained very nicely and in detail. Communication synchronously takes place between a master and a slave device. Data is always transmitted bit by bit along a single wire controlled by the clock signal from the master device. The key features and characteristics of the I2C protocol are Two-wire communication, Master-slave Architecture, Multi-master support, addressing, clock speed, Acknowledgment, and start-stop conditions. The external RTC used, is also using I2C protocol to communicate with Raspberry Pi.

6.2 SPI Protocol

SPI stands for Serial Peripheral Interface. It is another standard serial communication protocol used in embedded systems, microcontrollers, etc. It is a full-duplex communication and doesn't support clock stretching. It is a synchronous communication protocol based on the master-slave configuration. Serial Peripheral Interface (SPI) has four signal lines namely SCLK (serial clock), MOSI (Master Out Slave In), MISO (Master In Slave Out), and SS/CS (Slave Select/Chip Select). In the thesis work by Prashanth [1], this protocol has been also explained very nicely and in detail.

6.3 MQTT Protocol

MQTT stands for Message Queuing Telemetry Transport. It is a lightweight messaging protocol which is a standard method of communication for IoT devices. Typically there has to be one MQTT broker with which several clients can connect and send messages using certain topics.

Figure 6.1 shows how an MQTT client can Publish messages when connected to the MQTT broker. It is a two-way communication and works on the publish-subscribe architecture. MQTT broker can be used with or without authentication. It is best practice to authenticate it with credentials to make use of no client without permission can publish or subscribe to any topic. This way our data is secured. There are different brokers and clients in the market. For the project, the open-source Eclipse Mosquitto Broker is chosen. More information can be found in [23].

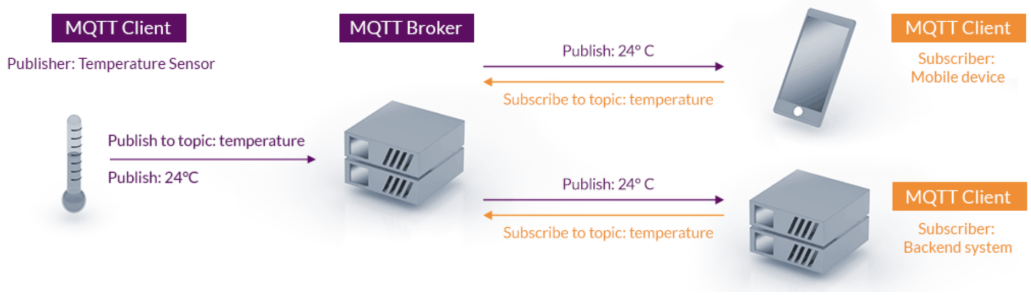


Figure 6.1: MQTT Publish/Subscribe Architecture [13]

6.4 One-wire Protocol

One-wire temperature sensors like DS18B20 or DHT11 work on a proprietary one-wire protocol developed by Dallas Semiconductor. The protocol uses the same line to transmit the clock and as well as data signals hence the name 1-wire. The protocol is similar to I2C but instead of separate Data and CLK lines, they are mixed into 1. This provides power over the data line with unique device addresses. This feature allows the user to connect, multiple one-wire sensors with unique addresses, to only one data pin on the ESP8266. It is very efficient for applications where minimizing wiring is a goal. The overall architecture is a Master-Slave architecture similar to SPI or I2C. A pull-up resistor is typically connected between the data line and the power supply to ensure proper voltage levels during communication. The value of the pull-up resistor affects the bus speed and signal integrity.

6.5 TCP

TCP (Transmission Control Protocol) is a core protocol of the Internet Protocol Suite [27]. It operates at the transport layer. It is well known for its reliable, ordered, and error-checked delivery of data between applications running on hosts on a TCP/IP network. TCP is widely used in various applications such as web browsing, email, file transfer, and real-time communication protocols like HTTP, FTP, SMTP, and VoIP. It is comparatively slower than UDP but it ensures reliable data transfer. It incurs higher overhead and latency due to its connection-oriented nature.

6.6 UDP

UDP (User Datagram Protocol) is a transport layer protocol in the Internet Protocol Suite [28]. It's a connectionless protocol, meaning it does not establish a connection before sending data. UDP provides a simple, minimalistic, and low-overhead mechanism for transmitting datagrams between hosts on a network. It is lightweight, connectionless, and offers lower latency but does not guarantee the reliability or ordering of data. It is widely used in real-time applications where timely delivery of data is more important than guaranteed delivery.

7 Results

In this chapter, the results in the form of data visualization, final product development, and CSV output will be discussed.

7.1 Final Product Development

The final product is developed to be a small, compact smart sensor with fewer wires and scalability. The output is visualized in real-time with the Grafana dashboard when connected to the local network. It allows changing certain sensor configurations according to the user's needs.

Figure 7.1: Raspberry Pi Server with Protective Aluminium Case

Figure 7.1 shows the aluminum case used as a protective case for our server. It has a good cooling system as it has a net-like structure to allow air to follow. The side rail was not well insulated and it was getting in contact with the power supply causing an Electromagnetic Interference (EMI) issue. It has been solved by using good electrical wire tape for insulating the holes in the side rail.

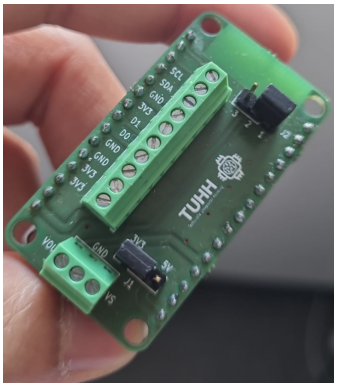


Figure 7.2: PCB Developed by Anirudh for the Project [14]

Figure 7.2 shows the Printed circuit board (PCB) which is replacing the external additional wires and the breadboard. It makes the smart sensor more compact and easy to handle.

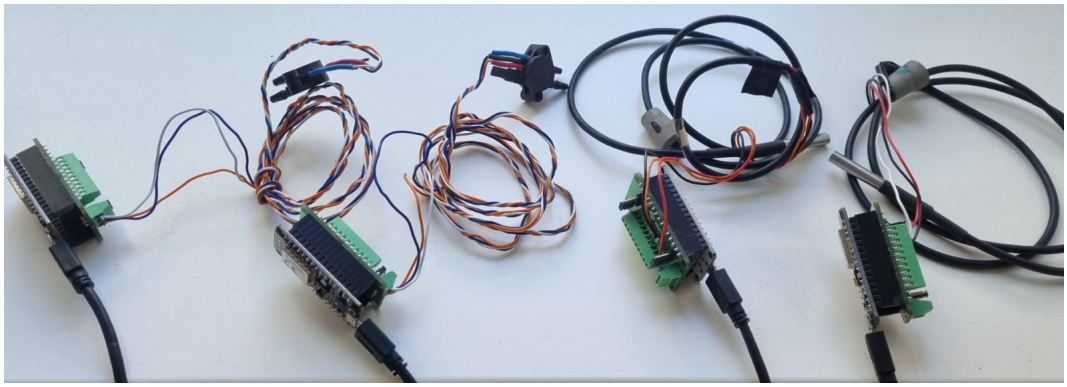


Figure 7.3: Smart Sensors Sending Data to Server

Figure 7.3 shows 4 smart sensors connected to the micro USB supply during the test phase. Here the ESP8266 and the PCB are filled and connected with the sensors and sending data to the server over MQTT.

7.2 Grafana dashboard

The Grafana dashboard shows the output of 12 sensors collected in a short period. It can be arranged in any way the user finds it suitable. The data can be visualized for user-preferred time and user-preferred visualization forms. That is the user can select a graph, chart, pie chart, table, etc. as needed.

Figure 7.4: Realtime Data Visualization of 12 Sensor in Grafana Dashboard During Test.

Figure 7.4 shows the data values of 12 sensors for the last 3 hours time frame. It only shows the data for a few hours as the sensors were removed shortly after testing and configuring. At the time of testing, there were only 5 sensors at hand thus it was configured in a way that at a time only 5 were connected and sending data to the server. The dashboard includes 6 temperature sensor panels and 5 pressure sensor panels including 2 panels for the DHT11 sensor. The dashboard is set to auto-refresh every 1m to fetch new data from the database. The temperature unit is displayed in degrees Celsius, pressure in hectopascal (hPa), and humidity in percentage.

Figure 7.5: Data visualization of 9 Connected Sensors

Figure 7.5 shows the output of nine sensors over the last 2 hours, configured a bit differently with different visualization parameters. It just gives an example of how easy and flexible it is for the user to rearrange the dashboard according to his needs. An edit option on the panels lets the user go inside it and change the data visualization effects as needed. Visualization effects include labeling, marking, setting threshold values, adding colors, adding unit values, legends, etc.

7.3 CSV output file

The CSV output file is generated automatically every day once at 12 AM with the help of a cronjob and a bash script. The output file is named Output_Today's-Date.csv. Figure 7.6 shows the list of files being saved in the CSV_files directory every day. Note that if the server is running and no smart sensor is connected, still cronjob will create an empty file with no data stored.

```
sikder@raspberrypi:~/CSV_files $ ls
Output_2024-03-15.csv  Output_2024-03-18.csv  Output_2024-03-23.csv  Output_2024-03-30.csv
Output_2024-03-16.csv  Output_2024-03-19.csv  Output_2024-03-24.csv  Output_2024-04-01.csv
Output_2024-03-17.csv  Output_2024-03-21.csv  Output_2024-03-25.csv
```

Figure 7.6: List of Files in CSV_files Directory

```
sikder@raspberrypi:~/CSV_files $ head Output_2024-03-24.csv
name,time,pressure1,pressure2,pressure3,pressure4,temperature1,temperature2,temperature3,temperature4,temperature5
test12,2024-03-23T10:48:50.482951135Z,NA,0.115268826,NA,NA,NA,NA,NA,NA,NA
test12,2024-03-23T10:48:51.053478073Z,NA,NA,NA,NA,NA,19.5,NA,NA,NA
test12,2024-03-23T10:48:52.250071652Z,0.165448025,NA,NA,NA,NA,NA,NA,NA,NA
test12,2024-03-23T10:48:53.401354668Z,NA,NA,NA,NA,18.5,NA,NA,NA,NA
test12,2024-03-23T10:49:00.4852659ZZ,NA,0.122437276,NA,NA,NA,NA,NA,NA,NA
test12,2024-03-23T10:49:01.268599396Z,NA,NA,NA,NA,NA,19.5,NA,NA,NA
test12,2024-03-23T10:49:02.251917072Z,0.165448025,NA,NA,NA,NA,NA,NA,NA,NA
test12,2024-03-23T10:49:03.447025153Z,NA,NA,NA,NA,18.5,NA,NA,NA,NA
test12,2024-03-23T10:49:10.485470444Z,NA,0.122437276,NA,NA,NA,NA,NA,NA,NA
```

Figure 7.7: CSV Output

Figure 7.7 shows how the data is arranged in the CSV file. The header file shows the name of the database, time, and connected sensors with commas in between. Then the next lines follow the same pattern. If a certain sensor is not connected on that day or only connected for a certain period on that day, the CSV file will give 'NA' as its output value when there is no value. Here

'test12' is the name of the database and the names of the sensors are self-explanatory. Later the database name was changed to 'SmartSensors'. The time format is chosen in rfc3339 format.

7.4 Managing CSV output

To collect the output file from a Mac and Linux device, the user's device must first be connected to the Raspberry Pi AP, and a terminal has to be opened. Then "sftp sikder@raspberrypi.local" should be typed, followed by navigating to the home directory to the folder "CSV_files" and then "ls" should be typed to view the content of the folder. The get command can be used along with the name of the desired file to copy it to the user's laptop. The exit command can then be used to exit out of the Sftp server once the copying is done.

The same can be done with the rsync command. Rsync follows the source and destination pattern, where the user has to mention the source as the Raspberry Pi file location and the destination as the location where the user wants to transfer the Output file.

For a Windows user, a file transfer application like puTTY needs to be installed to use Secure Shell File Transfer Protocol (sftp) functionality. Alternatively, the user can use a USB stick to copy files directly from the Raspberry Pi when connected to a screen, keyboard, and mouse.

8 Conclusion and Outlook

The initial idea of the project was to make an off-grid smart sensor that could be accessed over the internet to see real-time data. It would have a database server separate from the IoT server. The purpose of the development of the old system was to eliminate the problems of scalability and manual data collection. It was also prone to loose wire connection as the system was not compact or protected by any box. Moreover, it was bulky and hard to manage.

However, during the timeline of the project, several obstacles redirected the project plan. For instance, the university security protocol restricted hosting servers in their network. Furthermore, Espressif networking system could not be supported by the university IT and security system. These issues caused the project to be driven in a slightly different direction which was more suitable considering the use-case and cost.

The project can be further developed by replacing Raspberry Pi with a more powerful server. The use of an SD card in the Raspberry Pi system is a drawback due to its poor lifespan and durability. The SD card has a limitation of read-write cycles. Every time data is stored or a simple log file is written to the SD card, there is a risk of SD card partition crash, as SD cards are not suitable for frequent read-write cycles such as HDD or SSD. Moreover, if the project is switched to a network outside the university's Eduroam network without the security protocol, the IoT system can be freely accessible over the internet from different parts of the world.

In conclusion, the IoT smart sensor developed for greenhouse soil monitoring is a very cost-effective, compact, and scalable system that uses all open-source materials to make it affordable. The whole server and configuration details along with the codes have been uploaded to a GitLab repository and will be made available for general use.

A Appendix

A.1 ESP8266 Code

The following gives a skeleton structure of the code used in the smart sensor. For the different sensors, the code will be changing in certain sections.

```
1
2  /*Set credentials for AP and MQTT*/
3
4  const char* ssid = "MyPiAP";
5  const char* password = "#####";
6  const char* mqtt_server = "192.168.5.1";
7  const char* mqtt_username = "#####";
8  const char* mqtt_password = "#####";
9
10 /*Function for setting up wifi connection*/
11
12 void setup_wifi() {
13   delay(10);
14
15 //setting ESP in station mode
16   WiFi.mode(WIFI_STA);
17   WiFi.begin(ssid, password);
18
19   while (WiFi.status() != WL_CONNECTED) {
20     delay(500);
21
22   }
23
24   randomSeed(micros());
25
26
27 }
28
29 /*Function for handling the configurations
30 set by user by subscribing to topics*/
31
32 void handleConfig(char* topic, byte* payload, unsigned int length) {}
33
34
35 /*Function for reconnecting to the MQTT Broker
36 until it establishes a connection*/
37
38 Void reconnect() {}
39
40
41 /*Function to run once at the beginning of the execution of the code*/
```

```
42
43
44 void setup() {
45   Serial.begin(115200); //for debugging
46   setup_wifi();
47   client.setServer(mqtt_server, 1883); //1883 -->  MQTT Port
48   client.setCallback(handleConfig);
49   }
50
51 /* This part of the code runs infinitely */
52
53 void loop() {
54   if (!client.connected()) {
55     reconnect();
56   }
57   client.loop();
58
59     /*code of the different sensors goes here */
60
61 // send json data over MQTT
62   StaticJsonDocument<32> doc;
63   doc["value"] = measurement_value;
64
65   char output[55];
66   serializeJson(doc, output);
67
68   client.publish("device/Sensors", output);
69
70 //Update lastSampleTime for next sensor reading
71
72   lastSampleTime = millis();
73
74 }
```

Listing A.1: Sample Arduino C++ Code

A.2 ESP8266 And Arduino-Mega libraries

For the code to run in Arduino IDE, different libraries need to be installed. Navigate to the tools option and then choose "Manage libraries" to install the following A.1. It includes the libraries for the modified old Arduino setup along with the new ESP8266 setup with a short description.

A.3 nodered flow.json

The following JSON file gives the GreenHouse node-RED flow, seen previously in the chapter for Node-RED setup, in JSON format. In a fresh Node-RED window, one can import the flows with the JSON file below which will automatically create the flows.

```
[
    {
```

Table A.1: Required Libraries for Arduino IDE

No	Name of the Libraries	Description
1	ArduinoJson	Used for the formating of the mqtt payload in JSON fromat
2	Adafruit_BusIO	This is a helper library to abstract away I2C and SPI transactions and registers for the old Arduino setup.
3	Adafruit_unified_sensor	Data Processiong and logging in the old Arduino system
4	DallasTemperature	Needed for the temperature sensor in both the setups
5	DHT_sensor_library	Needed for the DHT11 sensor in both the setups
6	OneWire	Needed for the onwwire sensors like DHT11 or DS18B20
7	RTClib	Needed for the RTC module in the old setup
8	SD	Needed for SD card for data logging in the old setup
9	PubSubCllient	Needed for MQTT connection in the ESP8266 setup

```
    "id": "6768569826a46141",
    "type": "tab",
    "label": "GreenHouse",
    "disabled": false,
    "info": "",
    "env": []
},
{
    "id": "9d7e4c8207cf02a3",
    "type": "debug",
    "z": "6768569826a46141",
    "name": "debug 17",
    "active": true,
    "tosidebar": true,
    "console": true,
    "tostatus": true,
    "complete": "payload",
    "targetType": "msg",
    "statusVal": "payload",
    "statusType": "auto",
    "x": 630,
    "y": 100,
    "wires": []
},
{
    "id": "0a7b81e8c2b41c97",
    "type": "change",
    "z": "6768569826a46141",
    "name": "",
    "rules": [
       {
           "t": "set",
           "p": "payload",
           "pt": "msg",
           "to": "{\t \"temperature1\": msg.payload.t1,

           \t \"pressure1\": msg.payload.p1,
```

```
            \t \"temperature2\": msg.payload.t2,

            \t \"pressure2\": msg.payload.p2,\t \"temperature3\": msg.payload.t3,

            \t \"pressure3\": msg.payload.p3,\t \"temperature4\": msg.payload.t4,

            \t \"pressure4\": msg.payload.p4,\t \"temperature5\": msg.payload.t5,

            \t \"pressure5\": msg.payload.p5,\t \"temperature6\": msg.payload.t6,

            \t \"DHT_Hum\": msg.payload.HUM,\t \"DHT_Temp\": msg.payload.temp\t}",
            "tot": "jsonata"
        }
    ],
    "action": "",
    "property": "",
    "from": "",
    "to": "",
    "reg": false,
    "x": 600,
    "y": 220,
    "wires": [
        [
            "19cce3b52c447ffb",
            "9d7e4c8207cf02a3"
        ]
    ]
},
{
    "id": "98968a35f684931e",
    "type": "mqtt out",
    "z": "6768569826a46141",
    "name": "Config Temperature Sensor",
    "topic": "config/sampleRate/Temperature",
    "qos": "",
    "retain": "false",
    "respTopic": "",
    "contentType": "",
    "userProps": "",
    "correl": "",
    "expiry": "",
    "broker": "5e5b410c36415b7f",
    "x": 940,
    "y": 580,
    "wires": []
},
{
    "id": "8f81369b1d0a1333",
    "type": "debug",
    "z": "6768569826a46141",
    "name": "debug 18",
    "active": true,
    "tosidebar": true,
```

```
    "console": true,
    "tostatus": true,
    "complete": "payload",
    "targetType": "msg",
    "statusVal": "payload",
    "statusType": "auto",
    "x": 730,
    "y": 480,
    "wires": []
},
{
    "id": "33971ed98333ae22",
    "type": "inject",
    "z": "6768569826a46141",
    "name": "Samplerate",
    "props": [
        {
            "p": "payload"
        }
    ],
    "repeat": "",
    "crontab": "",
    "once": false,
    "onceDelay": 0.1,
    "topic": "",
    "payload": "{\"sampleRate\":60000}",
    "payloadType": "json",
    "x": 590,
    "y": 580,
    "wires": [
        [
            "98968a35f684931e",
            "8f81369b1d0a1333"
        ]
    ]
},
{
    "id": "19cce3b52c447ffb",
    "type": "influxdb out",
    "z": "6768569826a46141",
    "influxdb": "7e061297b0432b4c",
    "name": "influxDB",
    "measurement": "test12",
    "precision": "",
    "retentionPolicy": "",
    "database": "database",
    "precisionV18FluxV20": "ms",
    "retentionPolicyV18Flux": "",
    "org": "organisation",
    "bucket": "bucket",
    "x": 1040,
    "y": 220,
    "wires": []
```

```
    },
    {
        "id": "896f587f769c2fe7",
        "type": "mqtt in",
        "z": "6768569826a46141",
        "name": "Sensors",
        "topic": "device/Sensors",
        "qos": "0",
        "datatype": "json",
        "broker": "5e5b410c36415b7f",
        "nl": false,
        "rap": true,
        "rh": 0,
        "inputs": 0,
        "x": 260,
        "y": 220,
        "wires": [
            [
                "0a7b81e8c2b41c97"
            ]
        ]
    },
    {
        "id": "a6e71ac385b3e666",
        "type": "mqtt out",
        "z": "6768569826a46141",
        "name": "Config Pressure Sensor",
        "topic": "config/sampleRate/Pressure",
        "qos": "",
        "retain": "false",
        "respTopic": "",
        "contentType": "",
        "userProps": "",
        "correl": "",
        "expiry": "",
        "broker": "5e5b410c36415b7f",
        "x": 910,
        "y": 840,
        "wires": []
    },
    {
        "id": "3cc7ff1b2dad96e4",
        "type": "debug",
        "z": "6768569826a46141",
        "name": "debug 19",
        "active": true,
        "tosidebar": true,
        "console": true,
        "tostatus": true,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "payload",
        "statusType": "auto",
```

```
    "x": 710,
    "y": 740,
    "wires": []
},
{
    "id": "af43c9e431c5f5e0",
    "type": "inject",
    "z": "6768569826a46141",
    "name": "Samplerate",
    "props": [
       {
          "p": "payload"
       }
    ],
    "repeat": "",
    "crontab": "",
    "once": false,
    "onceDelay": 0.1,
    "topic": "",
    "payload": "{\"sampleRate\":20000}",
    "payloadType": "json",
    "x": 570,
    "y": 840,
    "wires": [
       [
          "a6e71ac385b3e666",
          "3cc7ff1b2dad96e4"
       ]
    ]
},
{
    "id": "ce98dcd63bb70ba2",
    "type": "mqtt out",
    "z": "6768569826a46141",
    "name": "Config DHT11 Sensor",
    "topic": "config/sampleRate/DHT11",
    "qos": "",
    "retain": "false",
    "respTopic": "",
    "contentType": "",
    "userProps": "",
    "correl": "",
    "expiry": "",
    "broker": "5e5b410c36415b7f",
    "x": 920,
    "y": 1560,
    "wires": []
},
{
    "id": "fe883ab463a7b32f",
    "type": "debug",
    "z": "6768569826a46141",
    "name": "debug 20",
```

```
        "active": true,
        "tosidebar": true,
        "console": true,
        "tostatus": true,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "payload",
        "statusType": "auto",
        "x": 730,
        "y": 1460,
        "wires": []
    },
    {
        "id": "8fa3d10a27925fee",
        "type": "inject",
        "z": "6768569826a46141",
        "name": "Samplerate",
        "props": [
            {
                "p": "payload"
            }
        ],
        "repeat": "",
        "crontab": "",
        "once": false,
        "onceDelay": 0.1,
        "topic": "",
        "payload": "{\"sampleRate\":5000}",
        "payloadType": "json",
        "x": 590,
        "y": 1560,
        "wires": [
            [
                "ce98dcd63bb70ba2",
                "fe883ab463a7b32f"
            ]
        ]
    },
    {
        "id": "d16f475518783a7b",
        "type": "comment",
        "z": "6768569826a46141",
        "name": "Change Temparature Sensor Sampling Rate",
        "info": "",
        "x": 290,
        "y": 480,
        "wires": []
    },
    {
        "id": "6ffcb9b30d5085d6",
        "type": "comment",
        "z": "6768569826a46141",
        "name": "Default samplerate set at 10s, 1s = 1000ms, put value in ms",
```

```
    "info": "",
    "x": 340,
    "y": 420,
    "wires": []
},
{
    "id": "11ef2ed00d1901c4",
    "type": "comment",
    "z": "6768569826a46141",
    "name": "Change Pressure Sensor Sampling Rate",
    "info": "",
    "x": 280,
    "y": 720,
    "wires": []
},
{
    "id": "f944428c1abd5a91",
    "type": "comment",
    "z": "6768569826a46141",
    "name": "Change DHT11 Sensor Sampling Rate",
    "info": "",
    "x": 290,
    "y": 1440,
    "wires": []
},
{
    "id": "f0e276a65000d798",
    "type": "comment",
    "z": "6768569826a46141",
    "name": "Sensors Sending Data to Database",
    "info": "",
    "x": 260,
    "y": 80,
    "wires": []
},
{
    "id": "4dbfd5100fc60b77",
    "type": "mqtt out",
    "z": "6768569826a46141",
    "name": "Config Slope of Pressure Sensor",
    "topic": "config/slope_MPX5100DP",
    "qos": "",
    "retain": "false",
    "respTopic": "",
    "contentType": "",
    "userProps": "",
    "correl": "",
    "expiry": "",
    "broker": "5e5b410c36415b7f",
    "x": 920,
    "y": 1060,
    "wires": []
},
```

```
{
    "id": "ce8b4d0eed1ee988",
    "type": "debug",
    "z": "6768569826a46141",
    "name": "debug 22",
    "active": true,
    "tosidebar": true,
    "console": true,
    "tostatus": true,
    "complete": "payload",
    "targetType": "msg",
    "statusVal": "payload",
    "statusType": "auto",
    "x": 690,
    "y": 960,
    "wires": []
},
{
    "id": "2d645cfb3869d2e6",
    "type": "inject",
    "z": "6768569826a46141",
    "name": "Slope",
    "props": [
        {
            "p": "payload"
        }
    ],
    "repeat": "",
    "crontab": "",
    "once": false,
    "onceDelay": 0.1,
    "topic": "",
    "payload": "{\"slope_MPX5100DP\":1}",
    "payloadType": "json",
    "x": 530,
    "y": 1060,
    "wires": [
        [
            "4dbfd5100fc60b77",
            "ce8b4d0eed1ee988"
        ]
    ]
},
{
    "id": "d66b80a7c00ab3c8",
    "type": "mqtt out",
    "z": "6768569826a46141",
    "name": "Config Intercept of Pressure Sensor",
    "topic": "config/intercept_MPX5100DP",
    "qos": "",
    "retain": "false",
    "respTopic": "",
    "contentType": "",
```

```
    "userProps": "",
    "correl": "",
    "expiry": "",
    "broker": "5e5b410c36415b7f",
    "x": 920,
    "y": 1240,
    "wires": []
},
{
    "id": "9629418d7beb1e6e",
    "type": "debug",
    "z": "6768569826a46141",
    "name": "debug 23",
    "active": true,
    "tosidebar": true,
    "console": true,
    "tostatus": true,
    "complete": "payload",
    "targetType": "msg",
    "statusVal": "payload",
    "statusType": "auto",
    "x": 690,
    "y": 1140,
    "wires": []
},
{
    "id": "13de09d68ef848a3",
    "type": "inject",
    "z": "6768569826a46141",
    "name": "Intercept",
    "props": [
        {
            "p": "payload"
        }
    ],
    "repeat": "",
    "crontab": "",
    "once": false,
    "onceDelay": 0.1,
    "topic": "",
    "payload": "{\"intercept_MPX5100DP\":6}",
    "payloadType": "json",
    "x": 540,
    "y": 1240,
    "wires": [
        [
            "d66b80a7c00ab3c8",
            "9629418d7beb1e6e"
        ]
    ]
},
{
    "id": "5e5b410c36415b7f",
```

```
        "type": "mqtt-broker",
        "name": "",
        "broker": "192.168.5.1",
        "port": "1883",
        "clientid": "",
        "autoConnect": true,
        "usetls": false,
        "protocolVersion": "4",
        "keepalive": "60",
        "cleansession": true,
        "autoUnsubscribe": true,
        "birthTopic": "",
        "birthQos": "0",
        "birthRetain": "false",
        "birthPayload": "",
        "birthMsg": {},
        "closeTopic": "",
        "closeQos": "0",
        "closeRetain": "false",
        "closePayload": "",
        "closeMsg": {},
        "willTopic": "",
        "willQos": "0",
        "willRetain": "false",
        "willPayload": "",
        "willMsg": {},
        "userProps": "",
        "sessionExpiry": ""
    },
    {
        "id": "7e061297b0432b4c",
        "type": "influxdb",
        "hostname": "192.168.5.1",
        "port": "8086",
        "protocol": "http",
        "database": "SensorDB",
        "name": "",
        "usetls": false,
        "tls": "",
        "influxdbVersion": "1.x",
        "url": "http://localhost:8086",
        "timeout": "10",
        "rejectUnauthorized": true
    }
]
```

Listing A.2: flow.json file

A.4 Mosquitto Configuration file

Figure A.1 shows how to configure the mosquito broker so that it knows the exact location of the password file and it only allows clients to communicate with the required user and password.

```
sikder@raspberrypi:~/GreenHouse/mosquitto/config $ cat mosquitto.conf
listener 1883
allow_anonymous false
password_file /mosquitto/passwd_file/passwd
```

Figure A.1: Mosquitto Broker Configuration File

A.5 Docker compose file

The following yaml file contains the description for the version of the services to be run along with their set parameters, exposed ports, and environments. It also tells the system where to mount its config, data, log, or password files in the external host device. This allows the user to change according to their need directly from the host machine, without going inside the containers.

```
#Docker-compose.yaml file

version: '3'

services:
  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mosquitto
    restart: unless-stopped
    ports:
      - "1883:1883"
      - "9001:9001"
    volumes:
      - ./mosquitto/config:/mosquitto/config:rw
      - ./mosquitto/data:/mosquitto/data:rw
      - ./mosquitto/log:/mosquitto/log:rw
      - ./mosquitto/passwd_file:/mosquitto/passwd_file:rw
    environment:
      - TZ=${TZ:-Etc/UTC}

  influxdb:
    image: influxdb:1.8.10
    container_name: influxdb
    restart: unless-stopped
    ports:
      - "8086:8086"
    volumes:
      - ./influxdb:/var/lib/influxdb
    environment:
      - TZ=${TZ:-Etc/UTC}

  grafana:
```

```
    image: grafana/grafana:latest
    user: "0"
    container_name: grafana
    restart: unless-stopped
    ports:
      - "3000:3000"
    environment:
      - TZ=${TZ:-Etc/UTC}
    volumes:
      - ./grafana:/var/lib/grafana:rw

  nodered:
    image: nodered/node-red:3.1.3
    container_name: nodered
    restart: unless-stopped
    ports:
      - "1880:1880"
    volumes:
      - ./nodered:/data:rw
    environment:
      - TZ=${TZ:-Etc/UTC}
```

A.6 Dockerfile

This is a sample dockerfile to create an image from the Grafana service of the project. The Dockerfile is not important for the project as the project is running from the docker-compose file.

```
FROM grafana/grafana:latest

# Set the user to root
USER root

# Copy Grafana configuration and data
COPY ./grafana /var/lib/grafana

# Expose Grafana port
EXPOSE 3000

# Start Grafana
CMD ["grafana-server"]
```

A.7 collect_data.sh

Figure A.2 shows the bash script which queries the influxdb database and gets the data for the last 24hs from the database "SensorDB" and the measurement "SmartSensors" with the time

format rfc3339. Moreover, it replaces the empty fields in the timestamps with "NA" for better representation of the CSV file.

```
sikder@raspberrypi:~ $ cat collect_data.sh
#!/bin/bash


# Define output filename based on current date
output_filename="Output_$(date +'%Y-%m-%d').csv"


# Query data from InfluxDB and save to CSV file

docker exec influxdb influx -database 'SensorDB' -precision rfc3339 -execute "SELECT * FROM SmartSensors WHERE time >= now() -24h" -format csv | awk 'BEGIN{FS=OFS=","} {fo
r(i=1; i<=NF; i++) {if($i=="") $i="NA"}; print}' > ./CSV_files/"$output_filename"
```

Figure A.2: collect_data.sh Bash Script

A.8 Crontab

Figure A.3 shows the automation for the collection of the CSV output once every day at midnight.

```
sikder@raspberrypi:~ $ crontab -l


#grep 'mycmd' /var/log/syslog    gives the log output

0 0 * * * /home/sikder/collect_data.sh 2>&1 | logger -t mycmd
```

Figure A.3: Crontab Command to Generate Daily Output

A.9 Schemetic Diagram of the PCB

This Schematic diagram is developed by Anirudh for his part of the project on the Smart Sensors. The diagram is shared here to understand better the physical wiring connections and corresponding coding for the IoT system to run.

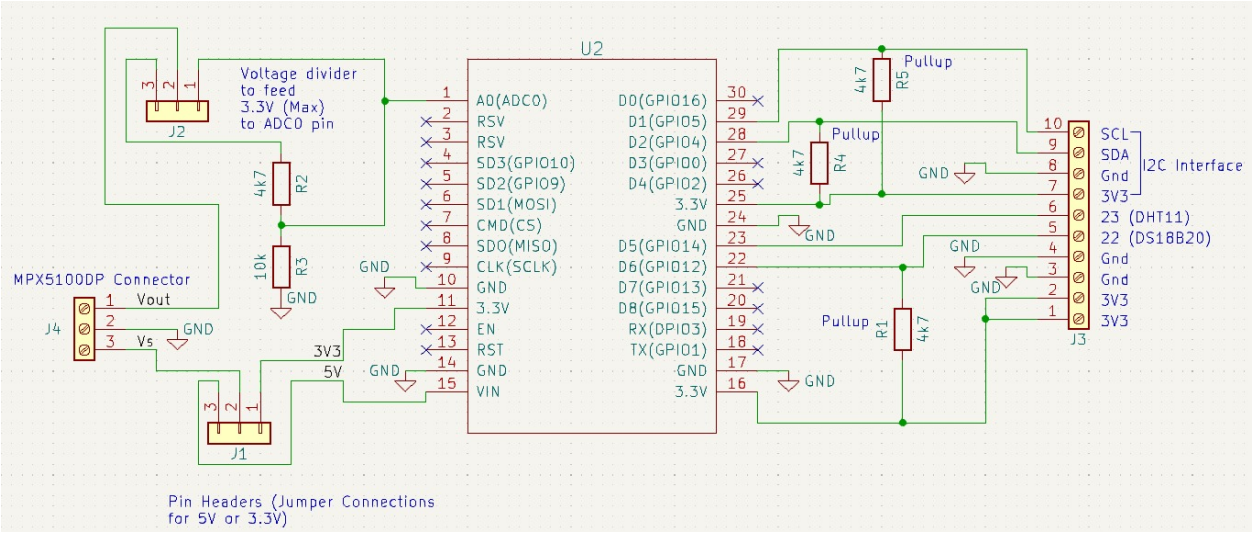


Figure A.4: Schemetic diagram of the PCB [14]

Bibliography

[1] Prashanth Reddy Ujjalli. *Prototyping an Arduino IoT Platform for Soil Quality Measurements*. PhD thesis, Hamburg University of Technology, Apr 2021.

[2] reichelt.de, 2024.

[3] Esp8266 pinout reference: Which gpio pins should you use?, Mar 2024.

[4] wikipedia.org, 2024.

[5] wikimedia.org, 2024.

[6] A look at the gpio pins on a raspberry pi.

[7] Temperature sensor.

[8] Esp8266 ds18b20 sensor web server arduino ide (single, multiple) — random nerd tutorials, Mar 2024.

[9] Differential pressure sensor 0-10kpa mpx5010dp original (made in usa) -mikroelectron, Mar 2024.

[10] How to set up the dht11 humidity sensor on the raspberry pi, Mar 2024.

[11] Official documentation for raspberry pi.

[12] Rpi rtc ds3231sn.

[13] Mqtt - the standard for iot messaging, Apr 2024.

[14] Anirudh Balasubramanian. *Soil Health Monitoring : Sensor Interface and Enclosure Design*. PhD thesis, Hamburg University of Technology, 2024.

[15] Raspberry pi, Apr 2024.

[16] Prashanth Reddy Ujjalli. Development of an arduino based field-tensiometer for soil health assessment. October 2019.

[17] Ds18b20 datasheet(pdf).

[18] Mpx5100 datasheet(pdf), Mar 2024.

[19] $\mathrm{Dht11}_{d}atasheet$.

[20] Installation · esp8266 arduino core, Mar 2024.

[21] pimylifeup.com, 2024.

[22] echtelerp/iot-stack-tutorial.

[23] Authentication methods, Mar 2024.

[24] Node-red, Mar 2024.

[25] Influxdb oss v1 documentation.

[26] Grafana open source documentation — grafana documentation, Mar 2024.

[27] What is tcp/ip in networking?

[28] Udp.