EEE3097S IoT API Design Report

Implementation of a Python API in support of MCP9700 Temperature Sensor, and Light Dependent Resistor (formerly DHT22 2 Click)

Group Number: 24

Student Numbers: THMSTE021

Date: 1st November 2020

Abstract

This document aims to summarize the development process of a temperature and luminosity sensor Application Programming Interface (API) as well as the main parts of its composition. The API is the key highlight of this document; however, the demonstrator application is also of principle focus as it is used to deploy the API into a real-world situation.

Table of contents

Introduction

The aim of this project is to design a sensor Application Programming Interface (API) to aid in forest fire detection, and to test its capacity using a suitable demonstrator.
The initial sensor designated to be used was the CM2322 sensor (DHT22 2 click board) provided by MikroElektronika. I unfortunately encountered the following main system faults while carrying out my development and testing (illustrated further in *Appendix A*):

Power Fault: The default surface mount jumper is positioned at the 3.3 V logic level (default logic level), the option; however, does exist to position the logic level at the 5 V logic level (i.e., if development required use of manufacturer libraries such as the Adafruit DHT22 libraries). Interestingly, when the circuit board was powered via the 3.3 V port the POWER LED was OFF (the device was off) and contrary to expectation when the circuit board was powered on the 5V port the POWER LED turned ON (the device was on). This fault raised concern as to whether the sensor was wired differently than what the datasheet described as being the default "out of the box" state.

I^2C Wake-Up Requirement: The I^2C protocol in the DHT22 2 click board is modified by the manufacturer to include a sleep mode. My attempts to wake up the sensor to read its I^2C address and take sensor readings using the WiringPi library and low-level C code were futile.

It is for these reason that I carried out a U-turn in my development and opted to use a different line of sensors in my attempt to meet the use case.
I received the Raspberry Pi on the 2nd of October and was, therefore, limited to a very short time span to carry out development and fault control. I was, as a result, limited to choose sensors readily available to me at the time.
The resultant sensors selected include:

- MCP9700 Temperature Sensor
- Light Dependent Resistor

1 Requirements

1.1 API Requirements

1.1.1 Functional Requirements

1.1.1.1 Functionality

(1) The API is intended to provide a means of measuring environmental temperature.
(2) The API is intended to provide a means of measuring environmental luminosity.

1.1.1.2 Qualities or Capabilities

(1) The API is intended to be versatile and support multiple standards of measurement i.e., measure temperature in degrees Celsius or degrees Fahrenheit.

1.1.1.3 Constraints

(1) The API is intended to be portable among a wide range of operating systems i.e., Android, Linux.
(2) The API is intended to be developed in less than a month's time.
(3) The API is intended to require a maximum of 8 GB microSD card storage.

1.1.2 Non-Functional Requirements

1.1.2.1 Functionality
(1) The API is intended to support both a console output (print to screen) and a file output (which is used to update a database controlling the application).

1.1.2.2 Qualities or Capabilities
(1) The API is intended to adopt an object-oriented programming scheme – making it easier to manage the channel setup and sensor readings.

1.1.2.3 Constraints
(1) The API is intended to adhere to Python Enhancement Proposals (PEPs) standardizations.

1.2 Demonstrator Requirements

1.2.1 Functional Requirements

1.2.1.1 Functionality
(1) The demonstrator is intended to use a temperature sensor and light sensor (in the form of a Light-Dependent Resistor).
(2) The demonstrator is intended to alert the user if a condition indicating a high probability of a fire occurs.

1.2.1.2 Qualities or Capabilities
(1) The API is intended to interface the Raspberry Pi and the sensors through an Analog to Digital Converter.
(2) The demonstrator is intended to enable a large number of sensor readings to be monitored easily using a simple to read graph

1.2.1.3 Constraints
(1) The demonstrator is intended to operate in a -40 °C to 80 °C temperature range.
(2) The demonstrator is intended to be affordable (< 550 ZAR) and easily accessible in market.

1.2.2 Non-Functional Requirements

1.2.2.1 Functionality
(1) The demonstrator is intended to use a web application to relay the sensor information to the user.
(2) The demonstrator is intended to use a file sharing service to interface the data log (text file) from the Raspberry Pi's local server to the host computer's local server.

1.2.2.2 Qualities or Capabilities
(1) The demonstrator is intended to use the Flask framework and MariaDB database (interfaced though phpMyAdmin) to develop a web-based application.

1.2.2.3 Constraints
(1) The demonstrator is intended to be portable among a wide range of operating systems i.e., Android, Linux.

2 Functional Specification

2.1 API Specification

2.1.1 Source links

Resource	Source Link
README	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/API/README.md
CONTRIBUTING	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/API/CONTRIBUTING.md
LICENSE	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/API/LICENSE
API's documentation	https://eee3097s-project.readthedocs.io/en/latest/
API's PyPi page	https://pypi.org/project/API-StevenThomi/0.0.1/

2.1.2 API Design Description

2.1.2.1 Class diagram

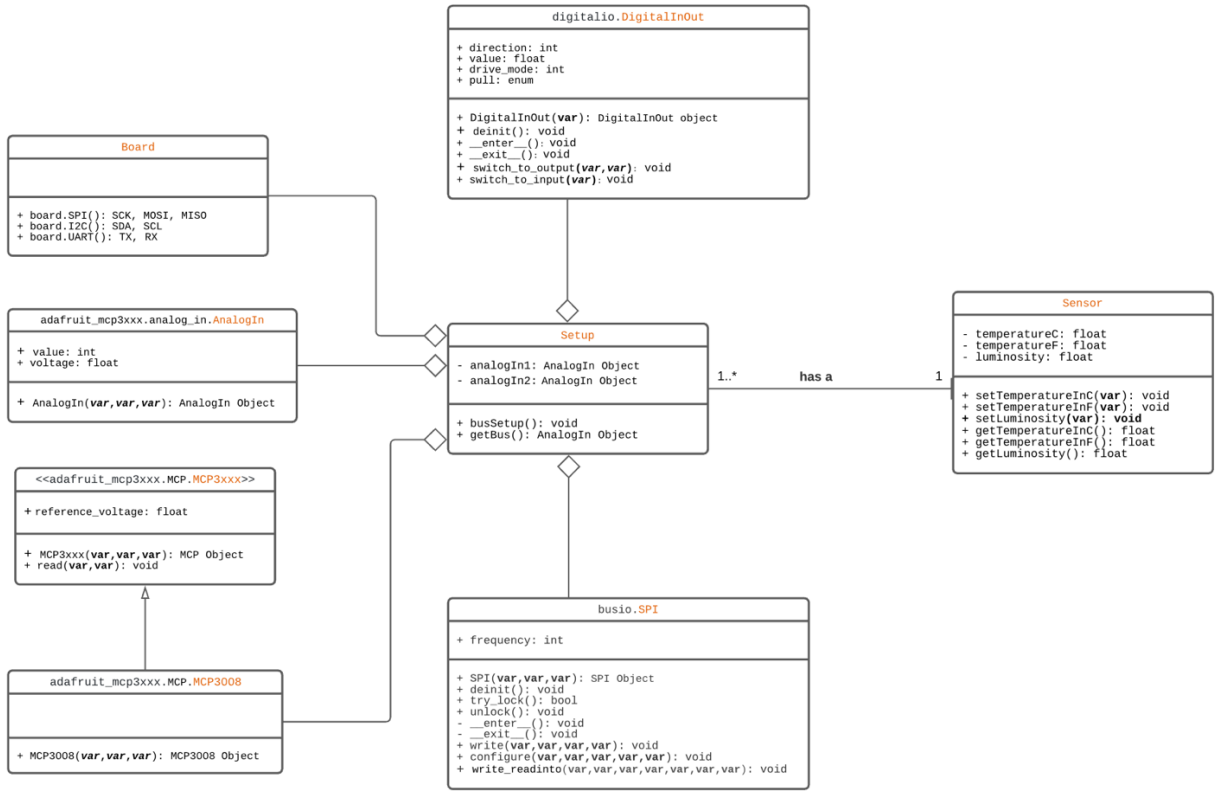

Figure 1: API Design Description; Class Diagram

2.1.2.2 Sequence diagram

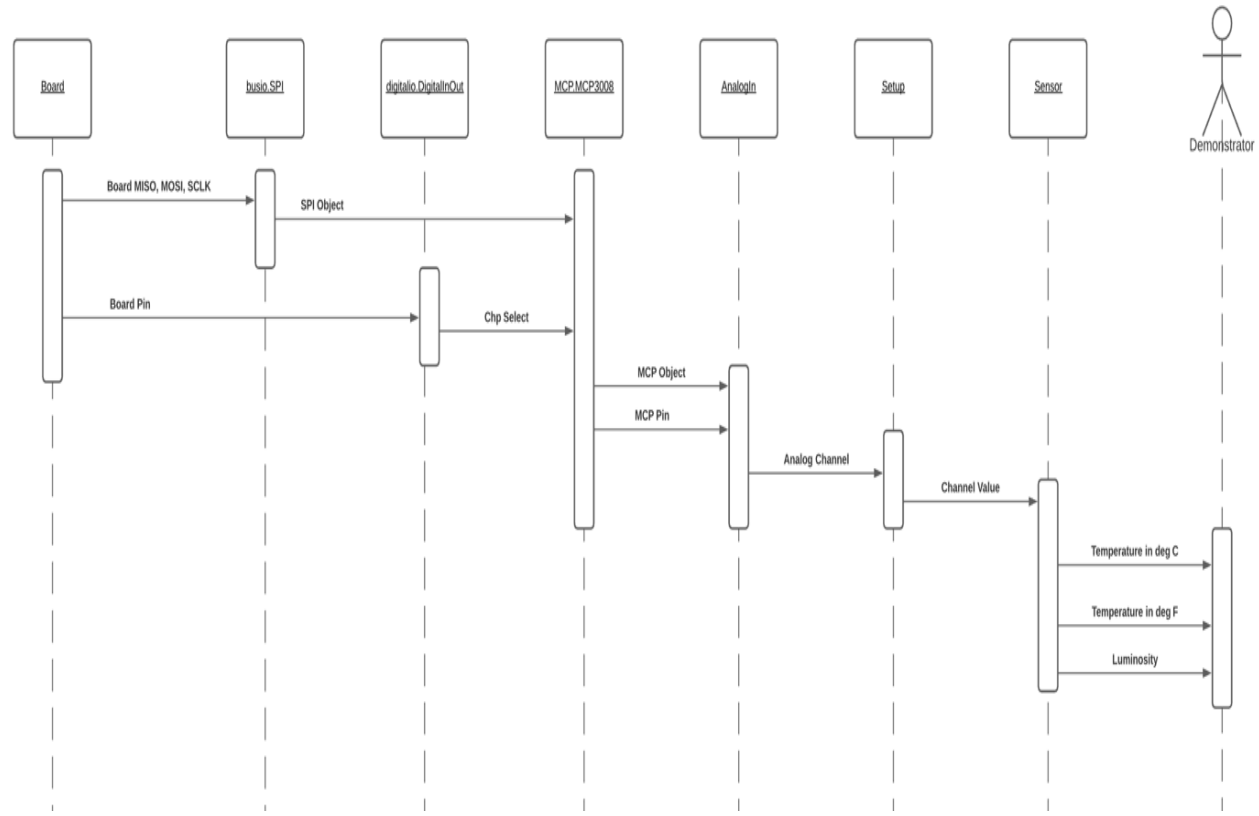

Figure 2: API Design Description; Sequence Diagram

1.1.1.1.(1): The API Sensor class contains getTemperatureInC() and getTemperatureInF() methods used to read temperature in degrees Celsius and degrees Fahrenheit respectively.

1.1.1.1.(2): The API Sensor class contains a getLuminosity() method used to read luminosity values.

1.1.1.2.(1): The API provides versatility in measurement by measuring temperature in degrees Celsius and degrees Fahrenheit.

1.1.1.3.(1): The API is portable among a wide range of operating systems. The classes used in the development of the API can all be obtained using either pip or pip3 commands.

1.1.1.3.(2): The API was developed in less than a month's time – as indicated on my GitHub activity tracker.

1.1.2.1.(1): The API returns a value using its Sensor class accessor methods. These values can easily be passed onto a text file or onto the console without altering the structure of the API as indicated in the application_file.py and application_console.py in the API repo.

1.1.2.2.(1): The API adopts an object-oriented programming scheme as indicated in the Class diagram.

1.1.2.3.(1): The API adheres to PEP standards.

2.2 Demonstrator Specification

2.2.1 Source links

Resource	Source Link
README	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/Web%20Development/README.md
CONTRIBUTING	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/Web%20Development/CONTRIBUTING.md
LICENSE	https://github.com/StevenThomi/eee3097s_project/blob/master/eee3097s_project/Web%20Development/LICENSE

2.2.2 Demonstrator System Design

2.2.2.1 Usecase Diagram

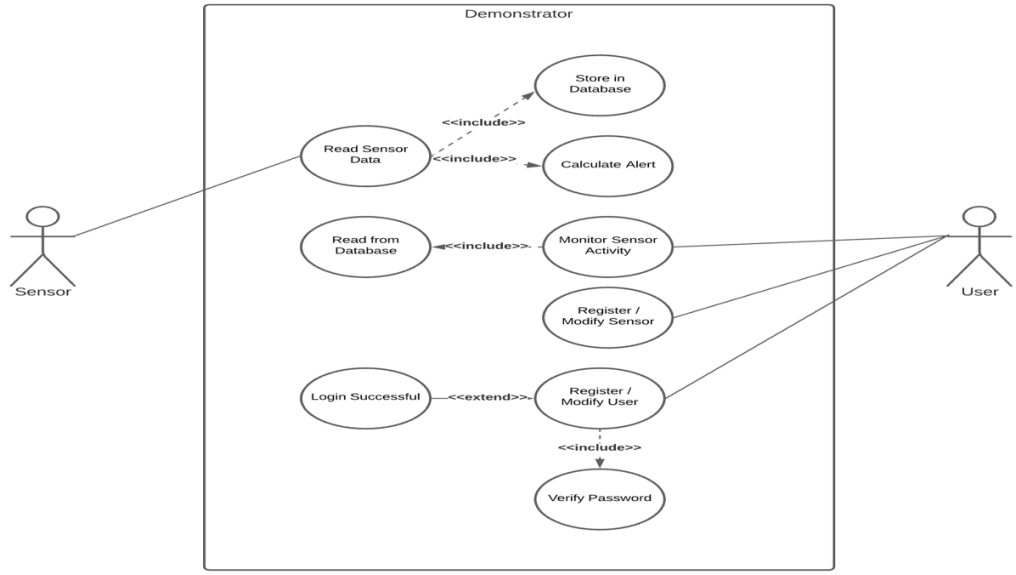

Figure 3: Demonstrator System Design ; Usecase Diagram

2.2.2.2 System Block Diagram

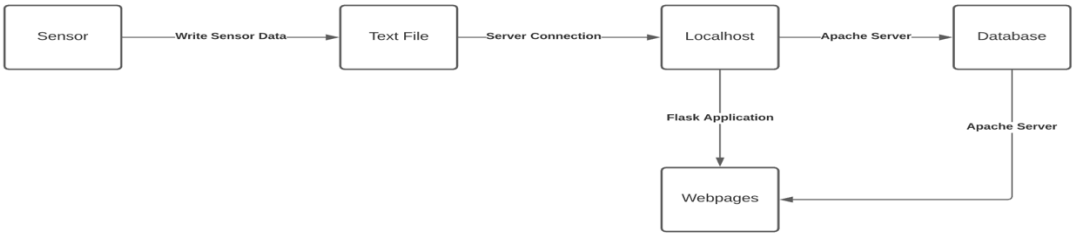

Figure 4: Demonstrator System Design ; System Block Diagram

2.2.2.3 Sequence Diagram

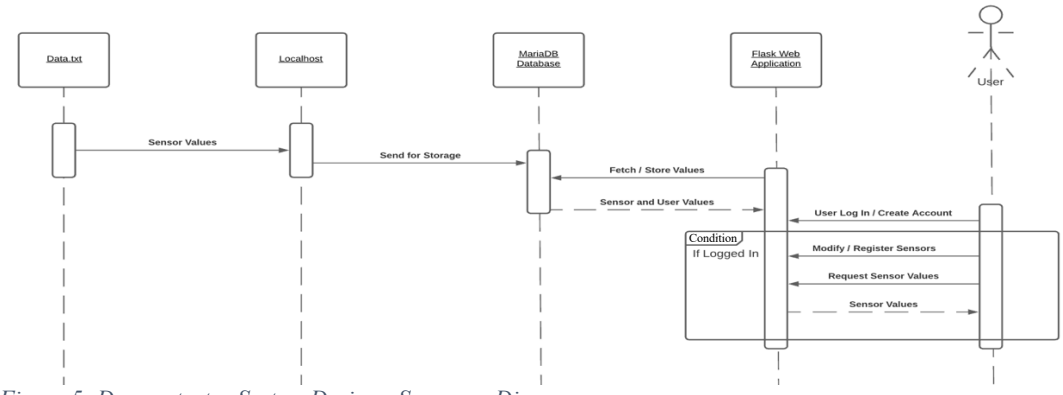

Figure 5: Demonstrator System Design ; Sequence Diagram

1.2.2.1.(2): The demonstrator uses samba file sharing to interface data.txt from the Raspberry Pi's local server to the host computer's local server.

1.2.2.2.(1): The demonstrator uses both a Flask application and a MariaDB database to develop the web-based application.

1.2.2.3.(1): The demonstrator uses services and libraries which are portable among a wide range of operating systems i.e., Android, Linux.

2.2.3 Demonstrator Electrical Design

2.2.3.1 Block Diagram

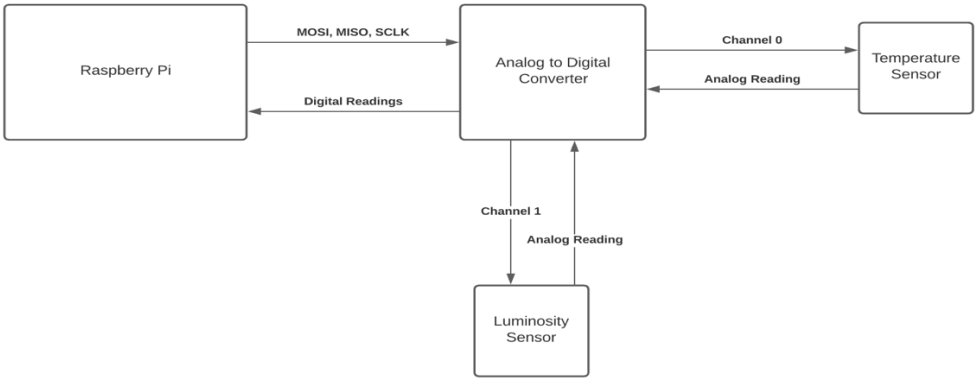

Figure 6: Demonstrator Electrical Design; Block Diagram

2.2.3.2 Schematic

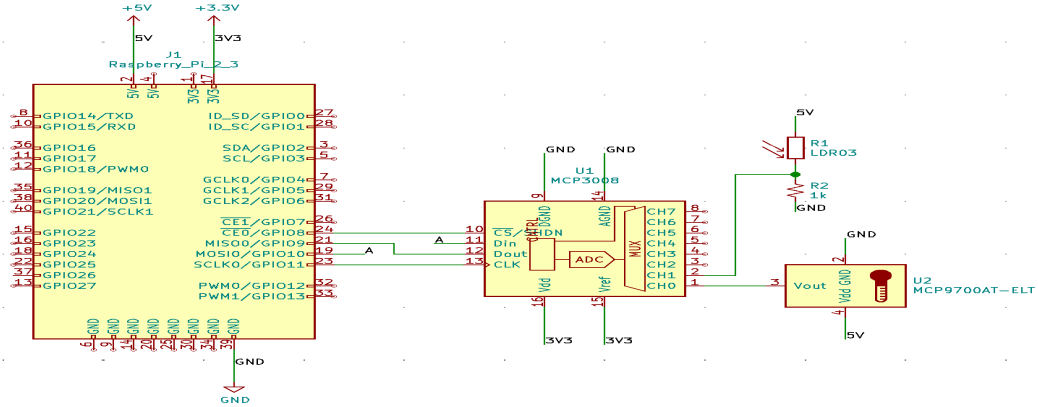

Figure 7: Demonstrator Electrical Design; Schematic

1.2.1.1.(1): The demonstrator uses a temperature sensor and luminosity sensor to meet the use case.

1.2.1.2.(1): The Raspberry Pi is interfaced with the sensors using an Analog to Digital Converter – to convert the analog reading to a digital level relayed to the Pi's digital input pins.

1.2.1.3.(1): The MCP9700 temperature sensor is capable of recording temperature values in the range of -40 °C to 80 °C.

1.2.1.3.(2): The total demonstrator cost (i.e., sensors, ADC, Raspberry Pi Zero) meets the budget requirement (< 550 ZAR) and are easily accessible in market.

2.2.4 Demonstrator Software Design

Screen-captures have been taken of the demonstrator to document this stage in a more robust manner. These screen-captures can be found in *Appendix B*.

2.2.4.1 Block Diagram

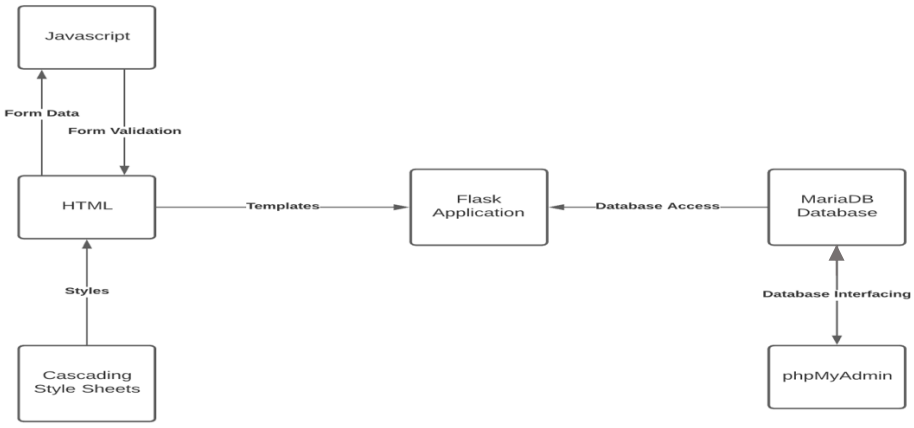

Figure 8: Demonstrator Software Design; Block Diagram

2.2.4.2 Sequence Diagrams

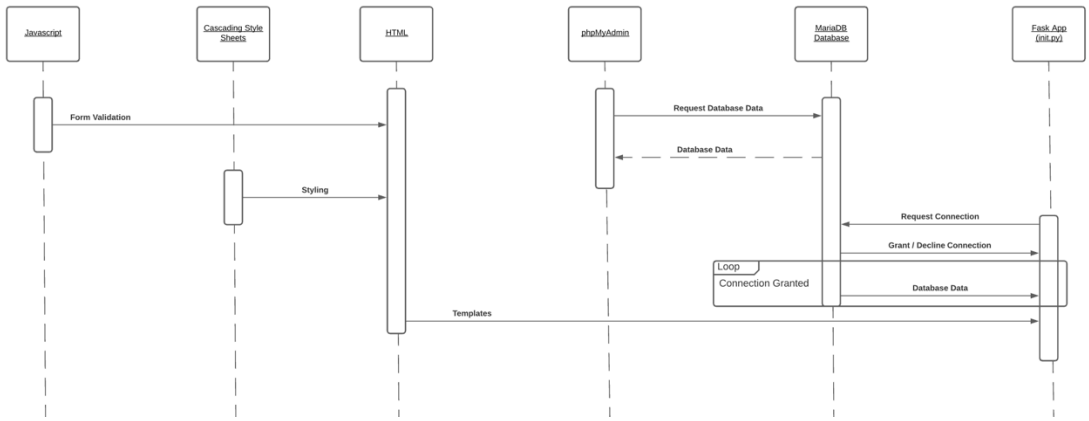

Figure 9: Demonstrator Software Design; Sequence Diagram

1.2.1.1.(2) The demonstrator (through the Flask Application) processes the database query results and alerts the user if a condition indicating a higher probability of a fire occurs using a Boolean value (0 for False, and 1 for True) as indicated in *Appendix B*.

1.2.1.2.(2) The demonstrator uses a marked line graph to document the ten most recent entries to the database using graphing libraries (implemented in the Flask init.py application in the Demonstrator repo) as indicated in *Appendix B*.

1.2.2.1.(1) The demonstrator uses a web application to relay the sensor information to the user.

1.2.2.2.(1) The demonstrator uses the Flask framework and MariaDB database to develop a web-based application.

Summary and Conclusions

API Summary and Conclusions

The API adds value to the IoT space by establishing an object-oriented, high-efficiency technique of reading and analyzing analog sensor values using the Raspberry Pi.
The developed API extends the functionality of the selected hardware module by catering for a wide range of sensor output modes i.e., Celsius and Fahrenheit temperature readings can be obtained.
I would wish to optimize the Analog to Digital Converter by increasing the number of sensors supported (i.e., adding a Geiger tube (radiation sensor) to report on radiation changes and chemical activity) and using a GPS module instead of keying in the latitude and longitude co-ordinates when registering a sensor package (as illustrated in the Demonstrator API repo and *Appendix B*).

Demonstrator Summary and Conclusions

The Demonstrator fetches the sensor values through the shared folder's data.txt file, performs calculations and writes the results to a database - which can be accessed using a web application.
The demonstrator can be used in the Agroforestry IoT application space as illustrated in *Appendix B* - a select number of officials are granted access to the website from where they can closely monitor activity in the forest ecosystem. An increase in the temperature reading (heat from fire) and decrease in the luminosity reading (smoky flame) would be indicators that a forest fire is probable.
I intended to use Sessions in my web development, but I run out of time; I would wish to add this feature to the demonstrator design in future. I would also like to use a Piezoelectric buzzer and LED configuration in my demonstrator to create a more finely tuned hardware interface. In addition to this, I would want to implement an aspect of control in my demonstrator i.e., after a fire alert is detected a possible mechanical action (i.e., water jet) should be triggered to control the fire.

Design process Summary and Conclusions

1. Github:

I worked on the project in a group of one. I; however, found it helpful to use GitHub as a resource as I was able to give the project supervisor an opportunity to closely monitor my progress. Moreover, the activity tracker kept me motivated to keep my GitHub account busy.
I am quite satisfied with how I carried out my development. The first thing I did was to form conceptual drawings (i.e., flowcharts, class diagrams) of my system from which I was able to form Command Line scripts (using random number generators to prototype sensor output) to test my system's capabilities. I also used logfiles to keep track of my UML and database changes through my development, keenly noting when significant changes were made.

2. Agile:

Agile development methodologies helped my team make progress. The team was more motivated to complete tasks with the knowledge that a progress report was due in the week.
Next time I need to collaborate on an embedded system project I would alternate a week to develop hardware and a week to develop software. The hardware and software are both crucially important in the overall API and demonstrator design and as a result both require a

fair share of time. I believe this alternate week method of development would result in a more robust product.

3. Tests:

Creating tests enhanced the API formation process by working as a means of debugging the software, resulting in cleaner code.
I am satisfied with my approach to the software development process i.e., using command line scripts and tests as a means of prototyping before finally coding a Flask Application.

References

1. MCP3008 Datasheet [Online]. Available at: https://ww1.microchip.com/downloads/en/DeviceDoc /21295d.pdf (Accessed: 22nd October 2020)

2. MCP9700 Datasheet [Online]. Available at: http://ww1.microchip.com/downloads/en/DeviceDoc/20001942G.pdf (Accessed: 22nd October 2020)

Appendices

Appendix A: CM2322 Faults

Power Fault

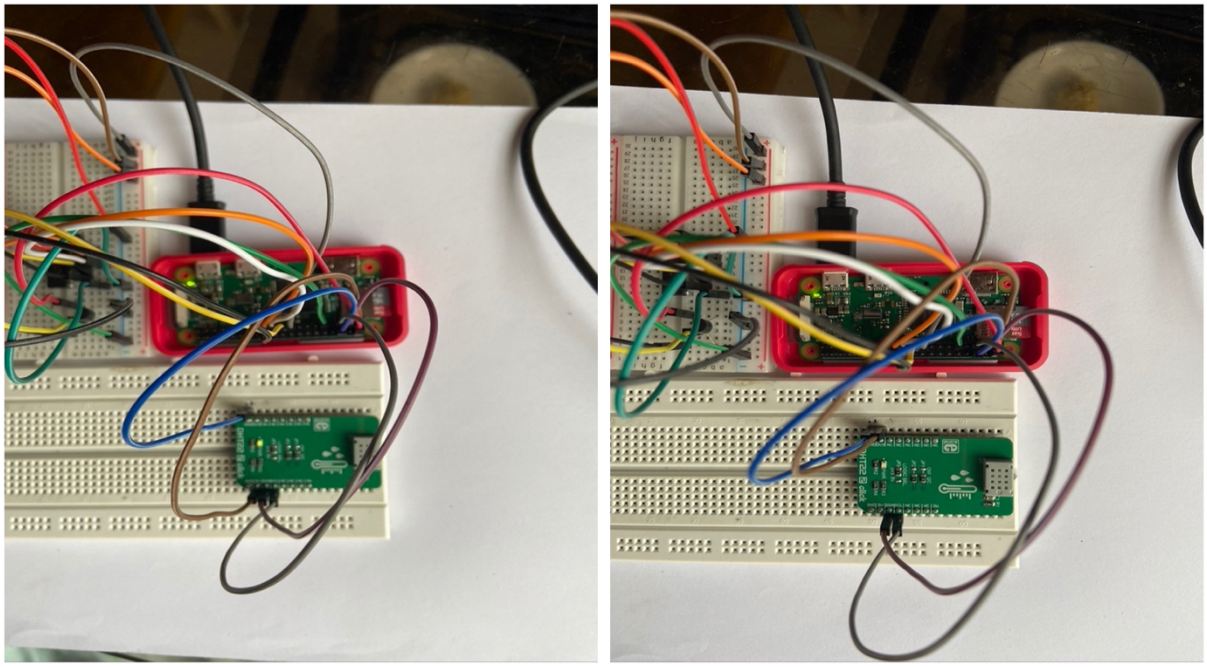

Figure 10: Power Fault

The POWER LED is ON when powered on a 5V logic level (left image) and OFF when powered on a 3.3V logic level (right image).
This goes against manufacturer guidelines and should be probed using an oscilloscope.
Further reading on this can be found here: https://www.mikroe.com/dht22-2-click

I²C Wake-Up Requirement

```
[pi@pibot:~ $ ls /dev/*i2c*
 /dev/i2c-1
[pi@pibot:~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@pibot:~ $
```

Figure 11: I2C Wake-Up Requirement

The I²C setting is first tested using the ls /dev/*i2c* to confirm that indeed the setting is activated. The I²C detect command is called thereafter.
The command does not detect any I²C devices. This can be attributed to the sleep mode state described in the Introduction.

Appendix B: Demonstrator Software Design

Flask Web Application

- **Monitor Page** *(red line indicates luminosity; blue line indicates temperature)*

Figure 12: Monitor Page

- **Documentation Page**

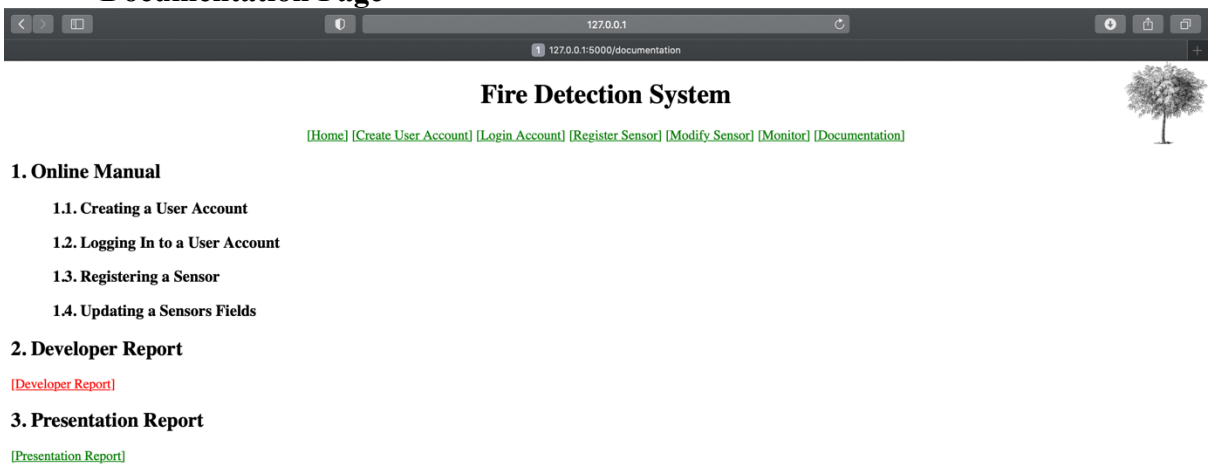

Figure 13: Documentation Page

- **Modify Sensor Page**

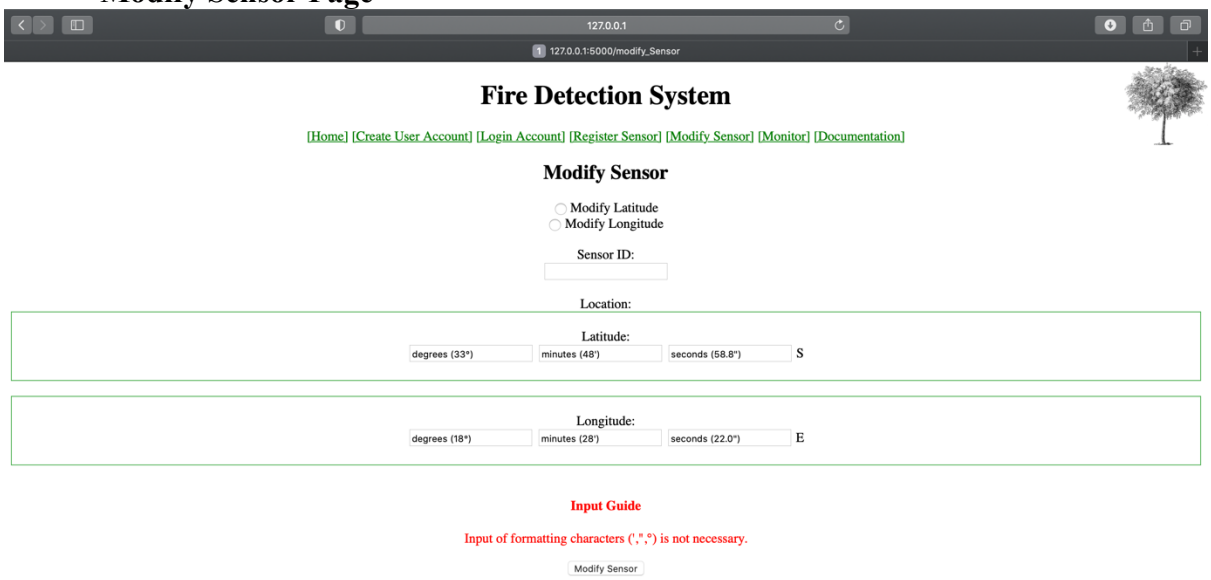

Figure 14: Modify Sensor Page

- **Register Sensor Page**

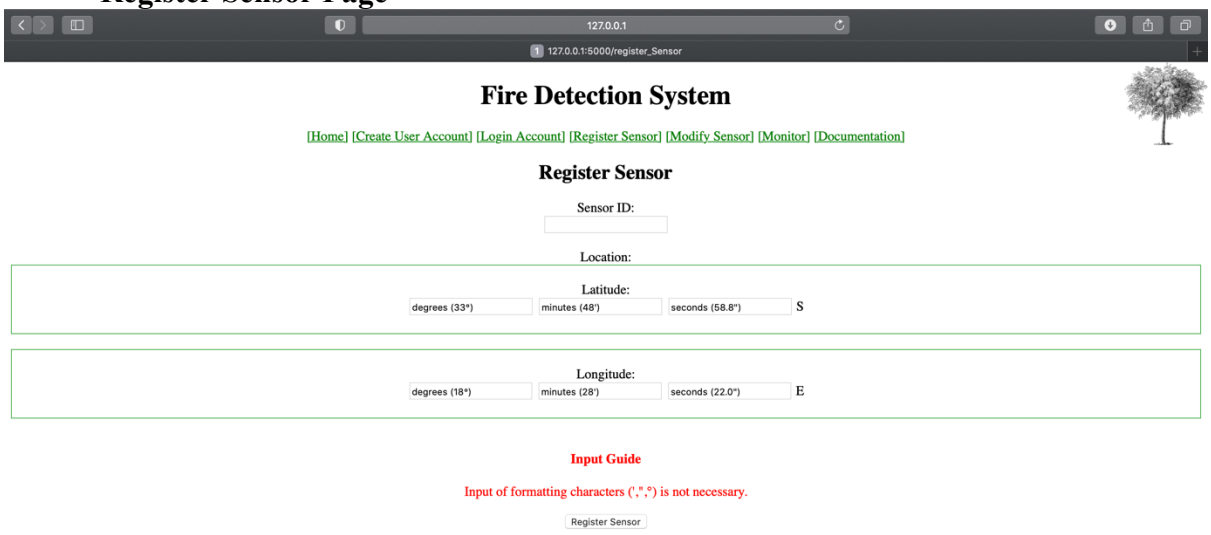

Figure 15: Register Sensor Page

- **Login Page**

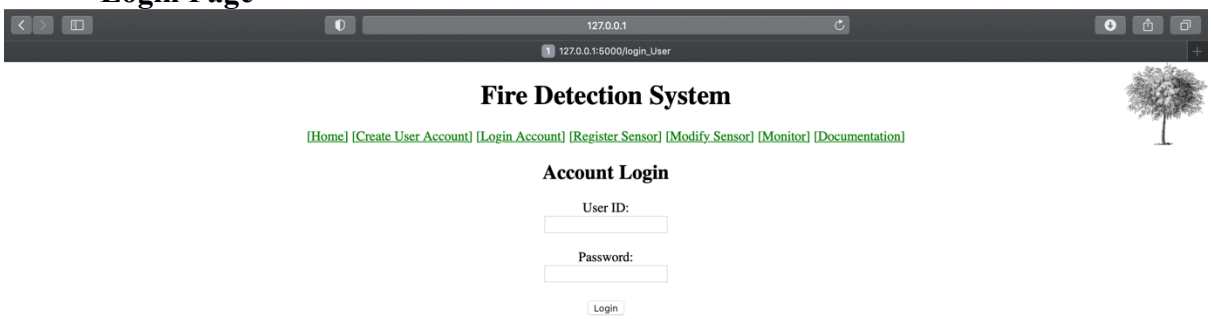

Figure 16: Login Page

- **Create User Account Page**

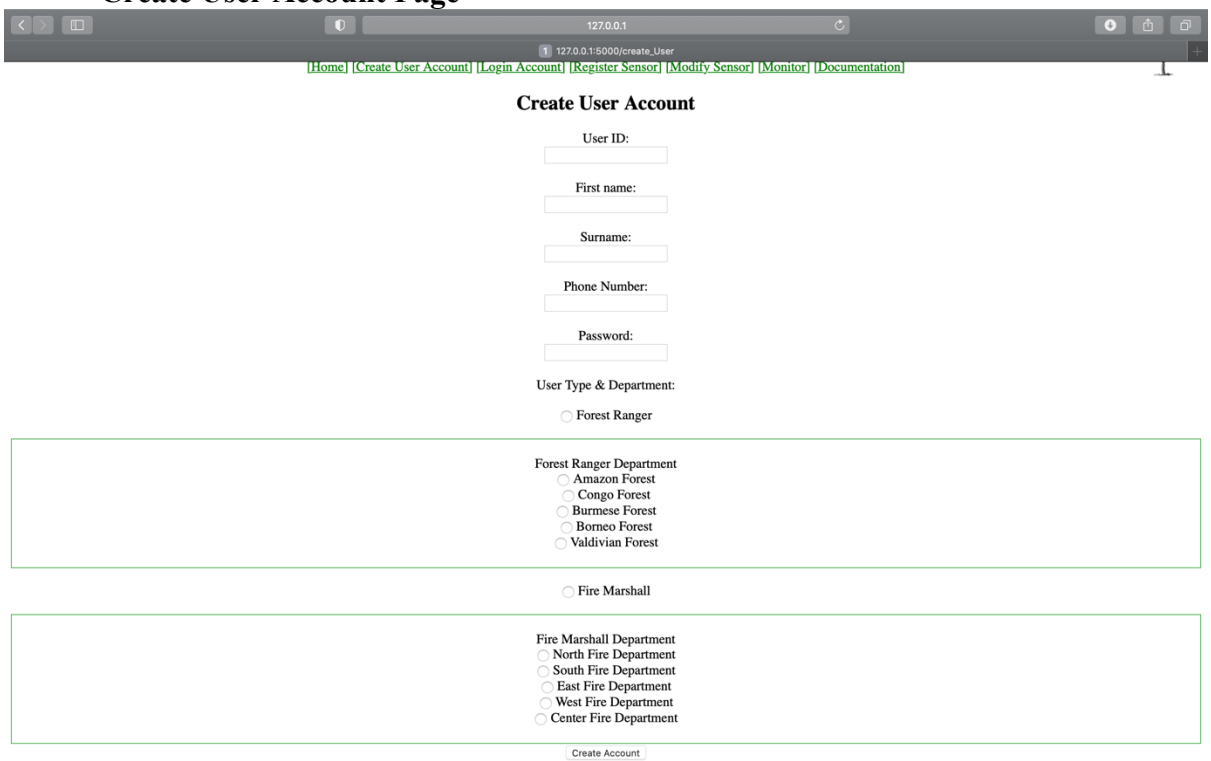

Figure 17: Create User Account Page

- **Home Page**

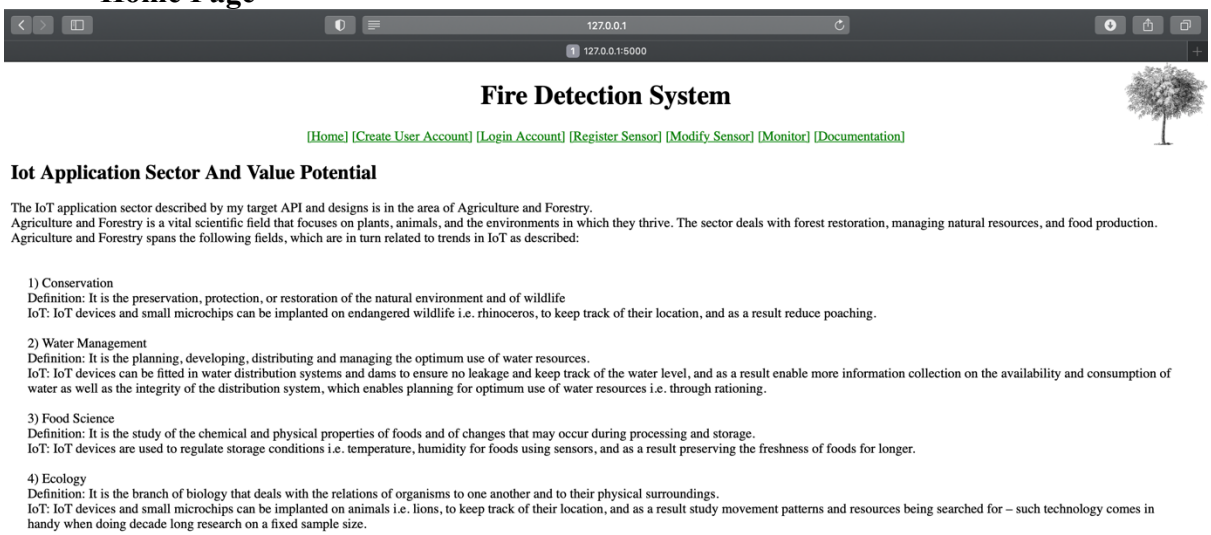

Figure 18: Home Page

Database Design

- **USER Table**

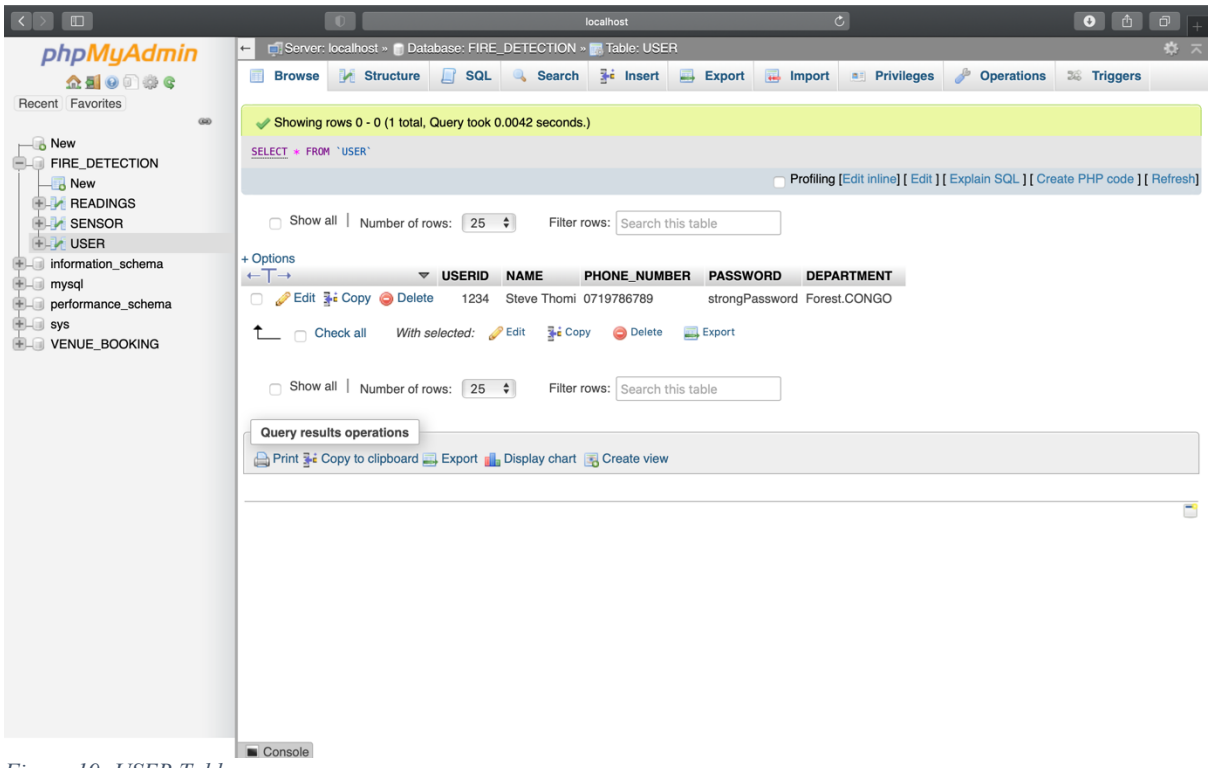

Figure 19: USER Table

- **SENSOR Table**

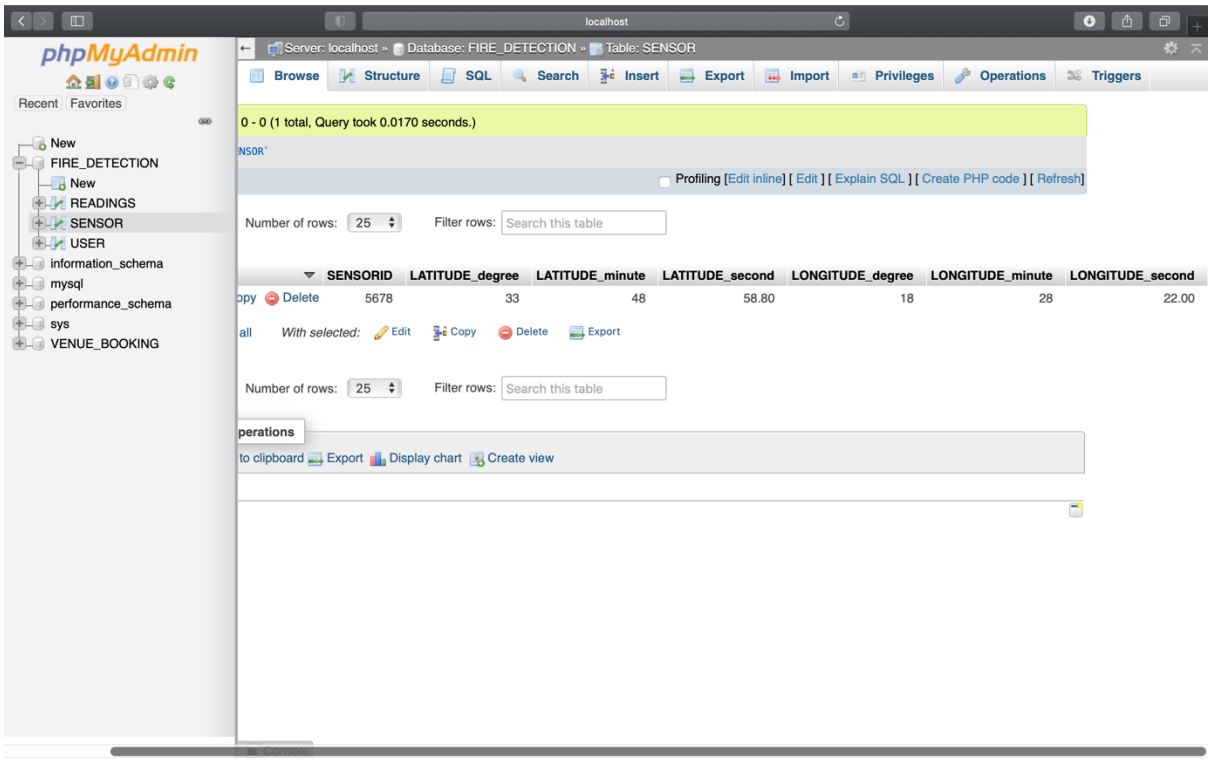

Figure 20: SENSOR Table

- **READINGS Table**

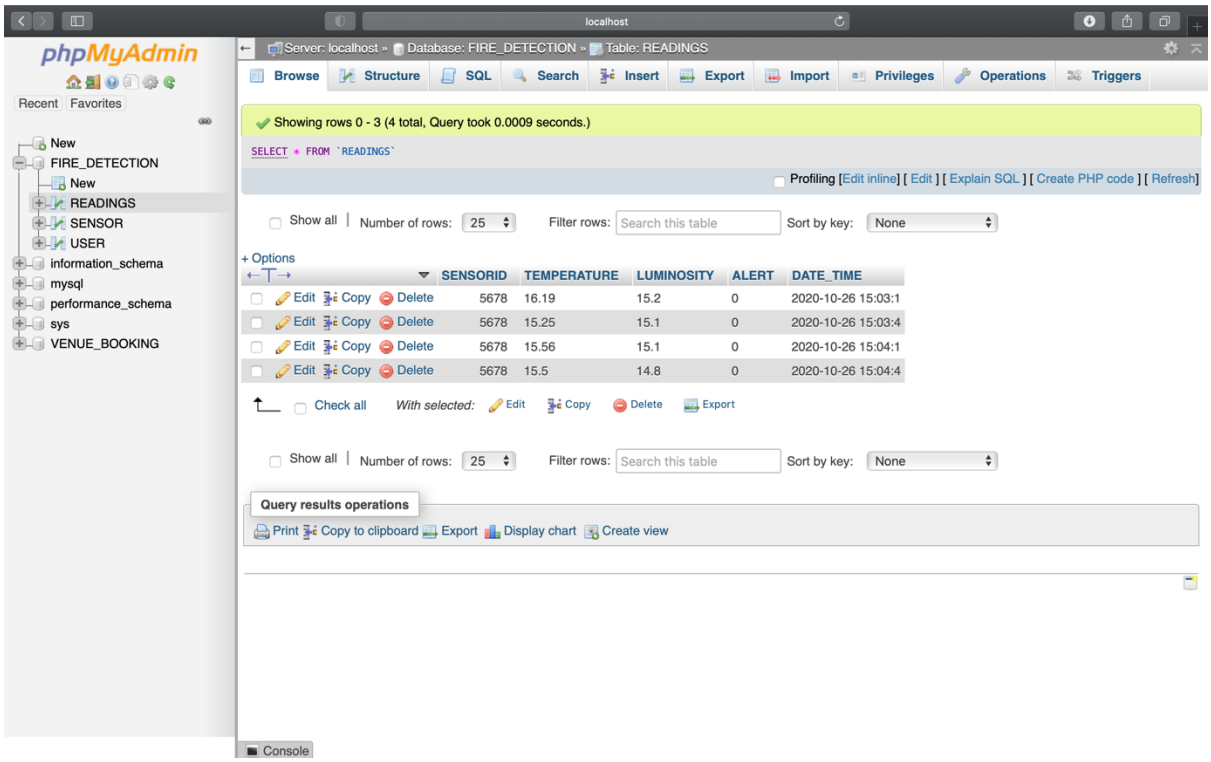

Figure 21: READINGS Table