ECE LABORATORY

DREXEL UNIVERSITY

To: **Dr. Peters**

From: **Ehi Simon**

Re: **ECE 304 Lab 5 - Web Server Creation Using Flask**

PURPOSE:

The purpose of this week's lab is to develop basic web server using Python and Flask, develop basic HTML templates and place my local computer and ESP32 on my Local Area Network.

Discussion:

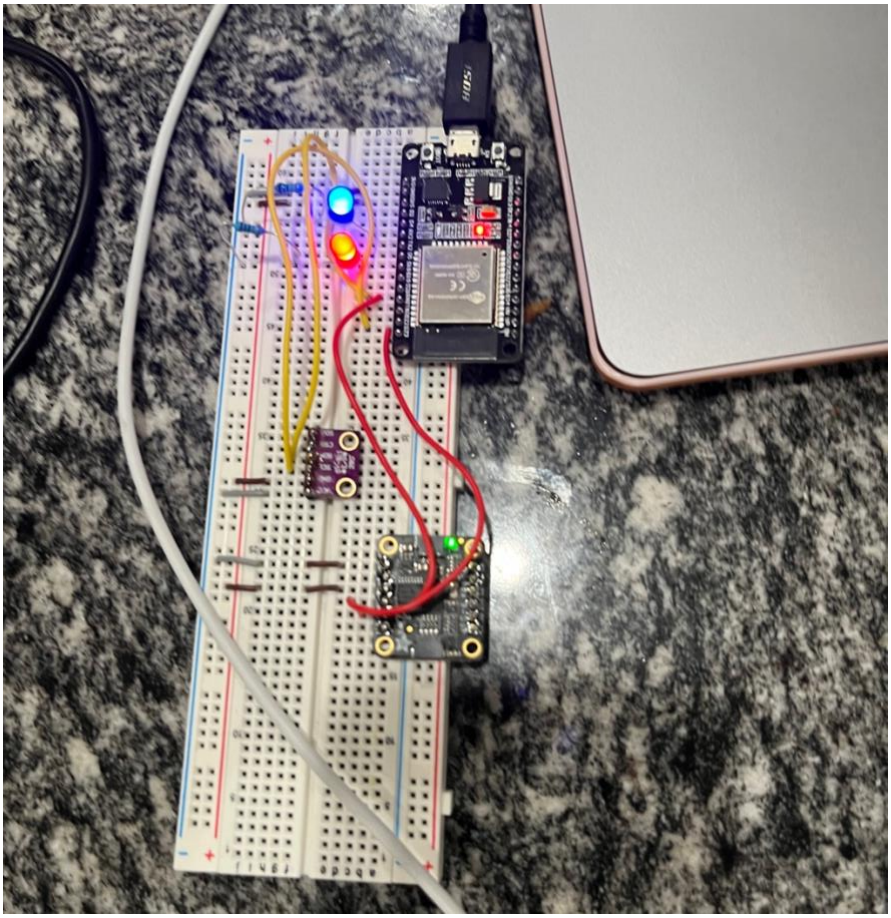

Fig. 1. Circuit Connection for Project 2

The circuit for the lab was built like the one above. It consists of 2 330Ω resistors, a red LED, a blue LED, a BME280 Environmental Sensor, an Adafruit BNO085 IMU, and an ESP32S microcontroller.

The setup of the project was very tasking. I downloaded "requirements.txt" to install all the required libraries and created a virtual environment in my project directory that I copied it into. From this directory, using the virtual environment I was able to connect to http://127.0.0.1:5000/. This page had my climate information. I was also able to connect to http://127.0.0.1:5000/led. This page had my LED command. To connect to these pages, I first had to run a python file named In_Class_Experiment.py with the command "$ python3 In_Class_Experiment.py". For all this to work, I still had to build and upload my ESP32 program. Figures showing this process can be found below.

```
(venv) ehisimon@Simons-MBP Week 5 % cd myproject
(venv) ehisimon@Simons-MBP myproject % deactivate
ehisimon@Simons-MBP myproject % python3 -m venv venv
ehisimon@Simons-MBP myproject % . venv/bin/activate
(venv) ehisimon@Simons-MBP myproject % pip3 install -r requirements.txt
ERROR: Could not open requirements file: [Errno 2] No such file or directory: 'requirements.txt'

[notice] A new release of pip available: 22.3.1 -> 23.1.2
[notice] To update, run: pip install --upgrade pip
(venv) ehisimon@Simons-MBP myproject % pip3 install -r requirements.txt
ERROR: Could not open requirements file: [Errno 2] No such file or directory: 'requirements.txt'

[notice] A new release of pip available: 22.3.1 -> 23.1.2
[notice] To update, run: pip install --upgrade pip
(venv) ehisimon@Simons-MBP myproject % pip3 install -r requirements.txt
Collecting certifi==2022.12.7
  Using cached certifi-2022.12.7-py3-none-any.whl (155 kB)
Collecting charset-normalizer==3.0.1
  Using cached charset_normalizer-3.0.1-cp311-cp311-macosx_11_0_arm64.whl (121 kB)
Collecting click==8.1.3
  Using cached click-8.1.3-py3-none-any.whl (96 kB)
Collecting colorama==0.4.6
  Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Collecting Flask==2.2.3
  Using cached Flask-2.2.3-py3-none-any.whl (101 kB)
Collecting idna==3.4
  Using cached idna-3.4-py3-none-any.whl (61 kB)
Collecting itsdangerous==2.1.2
  Using cached itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting Jinja2==3.1.2
  Using cached Jinja2-3.1.2-py3-none-any.whl (133 kB)
Collecting MarkupSafe==2.1.2
  Using cached MarkupSafe-2.1.2-cp311-cp311-macosx_10_9_universal2.whl (17 kB)
```

Fig. 2. Figure Showing Creation of Virtual Environment and Installation of Required Libraries

```
(venv) ehisimon@Simons-MBP Week 5 % python3 In_Class_Experiment.py
 * Serving Flask app 'In_Class_Experiment'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
 server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.213:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 145-398-783
127.0.0.1 - - [11/Jun/2023 00:50:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:50:04] "GET /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:50:12] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:50:16] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:51:38] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:51:43] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:51:46] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:55:16] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:59:21] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 00:59:26] "POST /led HTTP/1.1" 200 -
127.0.0.1 - - [11/Jun/2023 01:03:12] "GET / HTTP/1.1" 500 -
```

Fig. 3. Figure Showing Opening and Interacting with Webpage

Main.cpp

In my main.cpp file, I initialized the multiple libraries that were needed for the sensors to work and provide readings. I also included libraries to get the ESP32 to connect to STA, and libraries needed to interact with a server in JSON. I had to install the Adafruit JSON library prior. The LEDs are defined, the sea level reference pressure is defined and the BNO08X chip is reset. The BME object is created and the network SSID and password. The web server is then opened on port 80. This can all be found in the figure below.

```
// Import Libraries
#include <Arduino.h>
#include <WiFi.h>
#include <WebServer.h>
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Arduino.h>
#include <Adafruit_BNO08x.h>


// Define constants
const int blue_LED_pin = 4;
const int red_LED_pin=5;
const int freq = 5000; // LED PWM frequency
const int ledChannel = 0; // LED Channel
const int resolution = 8; // LED PWM resolution

// Define Sea level reference pressure (hPa)
#define SEALEVELPRESSURE_HPA (1013.25)
#define BNO08X_RESET -1

struct euler_t {
  float yaw;
  float pitch;
  float roll;
} ypr;

// Pin Request Status
bool LED_request = LOW;

// Create object for DHT
Adafruit_BME280 bme; // DHT object

/* Put your SSID & Password */
const char* ssid = "Verizon_C4ZKVV";  // Enter SSID here
const char* password = "dna-blear7-bow";  //Enter Password here

WebServer server(80); // Web Server open on port 80
```

Fig. 4. Figure Showing Initialization of Libraries and Variable Definitions

```
/* Put your SSID & Password */
const char* ssid = "Verizon_C4ZKVV";  // Enter SSID here
const char* password = "dna-blear7-bow";  //Enter Password here

WebServer server(80); // Web Server open on port 80

Adafruit_BNO08x  bno08x(BNO08X_RESET);
sh2_SensorValue_t sensorValue;
sh2_SensorId_t reportType = SH2_ARVR_STABILIZED_RV;
long reportIntervalUs = 5000;

void setReports(sh2_SensorId_t reportType, long report_interval) {
  Serial.println("Setting desired reports");
  if (! bno08x.enableReport(reportType, report_interval)) {
    Serial.println("Could not enable stabilized remote vector");
  }
}

void quaternionToEuler(float qr, float qi, float qj, float qk, euler_t* ypr, bool degrees = false) {

    float sqr = sq(qr);
    float sqi = sq(qi);
    float sqj = sq(qj);
    float sqk = sq(qk);

    ypr->yaw = atan2(2.0 * (qi * qj + qk * qr), (sqi - sqj - sqk + sqr));
    ypr->pitch = asin(-2.0 * (qi * qk - qj * qr) / (sqi + sqj + sqk + sqr));
    ypr->roll = atan2(2.0 * (qj * qk + qi * qr), (-sqi - sqj + sqk + sqr));

    if (degrees) {
      ypr->yaw *= RAD_TO_DEG;
      ypr->pitch *= RAD_TO_DEG;
      ypr->roll *= RAD_TO_DEG;
    }
}
```

Fig. 5. Figure Showing Setup for Yaw, Pitch, and Roll Values

The figure above shows the setup to obtain the yaw, pitch, and roll. The BNO08X chip is reset, and all the necessary initializations and calculations are performed to obtain the values.

In the figure below, I modified the handle_OnConnect function to include the yaw, pitch, and roll values in the doc under the JSONVar class. This class also contains the temperature, humidity, altitude, and pressure taken from the BME 280 sensor. I also have a set_LEDs function that interacts with the server and uses the client's response to update the status of the LEDs.

```
void handle_OnConnect() {
  LED_request = LOW;
  if (bno08x.getSensorEvent(&sensorValue)) {
  quaternionToEulerRV(&sensorValue.un.arvrStabilizedRV, &ypr, true);

  static long last = 0;
  long now = micros();
  long time = now - last;
  }
  JSONVar doc;
  doc["yaw"]= ypr.yaw;
  doc["pitch"] = ypr.pitch;
  doc["roll"] = ypr.roll;
  doc["id"]="Circuit 1";
  doc["Temperature"] = bme.readTemperature();
  doc["Humidity"] = bme.readHumidity();
  doc["Pressure"] = bme.readPressure()/1000;
  doc["Altitude"] = bme.readAltitude(SEALEVELPRESSURE_HPA);
  delay(1000);
  server.send(200, "application/json", JSON.stringify(doc));
}

void set_LEDs(){
  String body = server.arg("plain");
  JSONVar myObject = JSON.parse(body);
  digitalWrite(blue_LED_pin, (int) myObject["blue_led"]);
  ledcWrite(ledChannel,(int) myObject["red_led"]); // Set LED brightness
  server.send(200,"application/json",JSON.stringify(myObject));
}
```

Fig. 6. Figure Showing Handle_OnConnect Function and set_LEDs Function

The figure below shows the setup and loop functions. This setup function does more than those in previous projects. It sets the LED pin modes. It also initializes the BME280 sensor and the BNO08X chip. It then connects to the local Wi-Fi network and turns on the server. The setup function makes it possible to open the webpage and GET or POST from the page. The loop function simply keeps handling the client.

```
void setup() {
  Serial.begin(115200);
  delay(100);
  pinMode(blue_LED_pin, OUTPUT);
  bool status;
    status = bme.begin(0x76);
    if (!status) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
    }
  while (!Serial) delay(10);     // will pause Zero, Leonardo, etc until serial console opens

  Serial.println("Adafruit BNO08x test!");

  // Try to initialize!
  if (!bno08x.begin_I2C()) {
    Serial.println("Failed to find BNO08x chip");
    while (1) { delay(10); }
  }
  Serial.println("BNO08x Found!");


  setReports(reportType, reportIntervalUs);

  Serial.println("Reading events");
  // configure LED PWM functionalitites
  ledcSetup(ledChannel, freq, resolution);
  // attach the channel to the GPIO to be controlled
  ledcAttachPin(red_LED_pin, ledChannel);

  Serial.println("Connecting to ");
  Serial.println(ssid);

  //connect to your local wifi network
  WiFi.begin(ssid, password);
  delay(100);

  //check wifi is connected to wifi network
  while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
  Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected..!");
  Serial.print("ESP32 Address on Local Network: ");
  Serial.println(WiFi.localIP());

  server.on("/", handle_OnConnect);
  server.on("/led",HTTP_POST,set_LEDs);
  server.onNotFound(handle_NotFound);
  server.begin();
  Serial.println("HTTP server started");
}

void loop() {
  server.handleClient();
}
```

Fig. 7. Figure Showing Setup and Loop Functions

The final result consists of multiple webpages and changes from GET/POST requests. They can be found in the figures below.

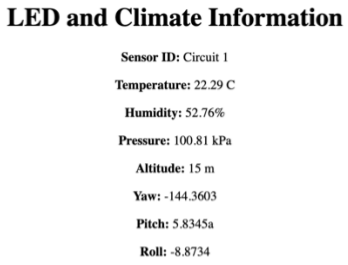

Fig. 8. Figure Showing Climate Information Webpage

Fig. 9. Figure Showing LED Commander Webpage

Conclusion

In this lecture, I learned about Flask, which is a lightweight web application framework in Python. I discovered the basic components of a Flask implementation, such as initializing an application instance, mapping URLs to Python functions, and starting the server. I also set up my computer by obtaining the IP address and creating a "templates" subfolder. The lecture demonstrated how to create dynamic web pages using Flask and interact with the ESP32 device. Overall, it was an informative session that provided hands-on experience with web server creation using Flask.