Student name: Bilal El Barbir
ID: 2179917
Course name: “IoT : Intro to python”
Project name: "Smart IoT Light Monitoring and Control System Using ESP32, BBB, MQTT, SQLite, and Flask"
Date: Thursday August 24, 2023

Scope: This project is a smart light monitoring and control system using MQTT, Flask, uWSGI, SQLite and IoT devices. It involves a Mosquitto broker on a BeagleBone Black, an ESP32 MCU with a simple photocell light measuring circuit, and a Python-Flask web application for visualization of light sensor data and threshold setting.

DESCRIPTION OF EACH CODE:

1. ESP32 Code: The ESP32 reads light sensor data and publishes it to the topic "sensor/light". The ESP32 connects to my Wi-Fi and mosquitto broker, then reads data from a trim pot (simulating the photocell sensor) and sends the readings to the broker. If the ESP32 loses connection with the MQTT broker, it tries to reconnect.

2. MQTT Subscribe (light_subscribe.py): This script subscribes to the "sensor/light" topic to receive light sensor data from the ESP32. Each time it receives a message , it stores it in an SQLite database along with the current timestamp. The script also has callback functions to handle connection events and incoming messages.

3. Flask Application (my_flask_app.py): This Flask application serves a web page that visualizes light sensor data from the SQLite database and allows the user to set a desired light value through a web form. The form sends a POST request to the server, which updates the light value threshold in the database. If the form hasn't been used, the application sets a default threshold value. This Flask application listens on port 8082.

4. WSGI Entry Point (wsgi.py): This script is the entry point for the Flask application when served via a WSGI. It imports the Flask application instance from the application code and checks if the script is being run directly or imported as a module. In the case of running through a WSGI server, the Flask application is not run directly so the app.run() statement is not executed.

5. uWSGI Configuration (myapp.ini): This is the configuration file for uWSGI, which serves the Flask application. The file:
    - specifies the application module
    - enables the master process management mode
    - sets the number of worker processes
    - defines the socket for communication

The configuration also makes uWSGI remove generated files/sockets on exit and to shut down upon receiving the SIGKILL signal.

6. HTML Template (index.html): This HTML file serves as the template for the Flask application's main page. It displays the last 100 light sensor readings from the database in a table and provides a form for setting the light threshold. The page auto-refreshes every 30 seconds.

7. light_control.py: this script controls the operation of a light based on sensor readings. The script fetches the light sensor value from an SQLite database and compares it with the set threshold value. If the sensor value is below the threshold, it publishes a message to an MQTT topic to trigger the ESP32 program and turn on the light.

8. myproject.service: This file ensures that my Python script starts automatically whenever your BBB boots up, making them operate continuously and independently.

   Commands used:
   - sudo systemctl enable mqtt_subscribe
   - sudo systemctl start mqtt_subscribe

TROUBLESHOOTING, ANAYLSIS AND DISSCUSION:

Building this project was a great learning experience and like any other learning experience, I faced a couple of challenges.

In the beginning, setting up the SQLite database was a bit challenging, as I was unfamiliar with configuring Flask to connect to a database. Configuring the URI for the SQLite database in the Flask application took me some time to understand and execute.

I faced issues with converting the MQTT message payload to a format that could be stored in the SQLite database. However, with a lot of trial and error, I managed to implement the correct decoding of incoming MQTT messages.

Setting up and configuring the Flask app and uWSGI server took some time as well. It turned out that it was crucial to correctly define the WSGI entry point and ensure the Flask app was served through uWSGI. I had to adjust the uWSGI configuration several times to optimize performance.

The command is: uwsgi --socket 0.0.0.0:8082 --protocol=http -w wsgi:app

- uwsgi: This is the command to start the uWSGI server.
- -socket 0.0.0.0:8082: This specifies the address and port number on which the uWSGI server will listen for incoming connections.
--protocol=http: This tells uWSGI to use the HTTP protocol for communication.
-w wsgi:app: This points uWSGI to the Python that it should invoke to get the application object. The app object is an instance of the Flask class and serves as the entry point for handling incoming HTTP requests.

Another challenging aspect was understanding how to use Flask-SQLAlchemy's query API to retrieve data from the SQLite database. I wanted to display the latest 100 sensor readings for visualizing on the webpage. It was not immediately obvious to me how to sort the readings. After reading the Flask-SQLAlchemy documentation and some online resources, I was able to make the correct query to fetch the data as needed.

Lastly, creating the Flask application's user interface using HTML and Jinja2 templating was a new experience. The process involved learning how to pass variables from Flask to the template, how to handle form submissions and POST requests, and how to design a simple yet functional user interface.

Overall, despite the challenges, this project enabled me to gain new skills and knowledge like web development, database management, web server setup, and Flask web development.

Online resources used:

For flask:
- https://www.digitalocean.com/community/tutorials/how-to-create-your-first-web-application-using-flask-and-python-3

For HTML:
- https://www.youtube.com/watch?v=qz0aGYrrlhU