COMP_SCI-5577-0001-44892-2025FS-IoTs

Group – 9

Mini-Project-4: Secure IOT Monitoring System with ESP32 and Web Dashboard

---

1. Introduction

- In this project we implemented a secure IoT-based temperature and humidity monitoring system using the ESP32 microcontroller, and it securely transmits encrypted real-time sensor readings through HTTP POST to a web server for further processing and visualization. In this we used DHT11 sensor for capturing temperature and humidity readings and integrates RESTful API endpoints for remote control.

- Here we include AES-128-CBC encryption for data transmission and storage, and used PostgreSQL database to maintain and manage data, and even included a web dashboard with real-time graphs and search features.

- Also this system include a secure login mechanism with hashed credentials and session-based authentication, ensuring only authorized users can access the dashboard, historical data and even any information from the end points.

- We added a weather API proxy with caching to show selected location weather details along with sensor data on the dashboard.

2. Objectives

- Here we built an IoT-based temperature and humidity monitoring system using an ESP32 and a DHT11 sensor, here the ESP32 reads data every 5 seconds and securely sends the encrypted readings to a web server via HTTP POST for every 10 seconds to display in dashboard to maintain a user friendly interface.

- Implemented AES-128-CBC encryption on the ESP32 to secure the sensor data before sending it, and this helps to prevent eavesdropping during transmission, and once the server receives the data, then it decrypts the data for safe storage and visualization.

- In PostgreSQL database the received data is stored with AES encryption for security, and web dashboard displays real-time temperature and humidity readings, along with a search feature that allows users to filter past data based on custom threshold values (above or below a specified limit).

- Also, included alert message to users, once the temperature or humidity reaches the threshold level the system will start an alarm alert.

- We implemented a secure authentication system that requires a username and password, and it stores only hashed passwords in the database to maintain login state using server-side sessions or JWTs, and also included a logout endpoint to terminate the user's session. Here only 'admin' user can able to create any user account, the 'forget password' option can directly hit the database for immediate password change.

- Also integrated the OpenWeatherMap API to fetch the real time weather information, here the weather column accepts latitude/longitude or city inputs and store the response in cache for 10 minutes to improve performance, and displays a weather card showing temperature, humidity, weather condition, icon, location, and the last update time.

- In ESP32 we implemented optimized network connection and disconnection feature without restarting the device, and also multiple user friendly option to control the ESP32 operations directly with IDE or with the RESTful API's like, to start, stop, enable or disable encryption, device information, sensor information and status of the device

3. Hardware and Software Used

- We used the ESP32 board as the main microcontroller, and it has built-in WiFi feature for data transmission and GPIO pins for connecting sensors. And DHT11 sensor is connected to GPIO 4 to get temperature and humidity readings.

- Used Arduino IDE for programming the ESP32 in C++, to enabling code compilation and uploading to web dashboard, and real-time debugging via serial monitor at 115200 baud rate for monitoring sensor data and network status.

- Flask Web Framework has used for server-side application to handle HTTP requests, decrypting the data, to maintain stable database interactions, and rendering the web dashboard with secure routes.

- In this project used PostgreSQL database to store encrypted sensor readings and user credentials, and it includes tables such as 'sensor_data' and 'users', and implemented indexes on 'timestamp' and 'team_number' columns in the 'sensor_data' table to improve query performance and also increase data retrieval speed.

4. Libraries Used:

ESP32- Libraries:

- **WiFi.h**: This helps ESP32 to enable WiFi connectivity and network scanning.
- **HTTPClient.h**: It is to handle the secure HTTP POST requests for data transmission from ESP32 to web server.
- **ArduinoJson.h**: It is to reformat the received sensor readings into JSON before encryption.

- **DHT.h:** It is used to read temperature and humidity from the DHT11 sensor, and it maintain a communication between ESP32 and DHT11 sensor.
- **WebServer.h:** It is to implement RESTful API endpoints on the ESP32 for remote configurations, such as adjusting upload intervals, server url, enabling or disabling encryption, fetching sensor and device information and also to start and stop the device.
- **mbedtls/aes.h** and **mbedtls/base64.h:** To provide AES-128-CBC encryption and Base64 encoding for securing sensor data before transmission.

mini-project-4_esp32.ino

```
#include <WiFi.h>                  // ESP32 WiFi library for network connectivity
#include <HTTPClient.h>            // HTTP client for sending data to cloud server
#include <ArduinoJson.h>           // JSON processing library for data formatting
#include <DHT.h>                   // DHT sensor library for temperature/humidity reading
#include <WebServer.h>             // HTTP server library for RESTful API endpoints
#include <time.h>                  // timestamp generation
#include <mbedtls/aes.h>           // AES encryption functions from mbedTLS library
#include <mbedtls/base64.h>        // Base64 encoding for secure data transmission

```

Server side Libraries:

- **psycopg2:** It is to Connect and interact with the PostgreSQL database.
- **Crypto.Cipher**: To performs AES decryption on received encrypted data.
- **requests:** It helps to integrate with the weather API to get the weather data and also to handle API response caching.

Dashboard Libraries:

- **Chart.js**: To create real-time interactive graphs for temperature and humidity.
- **JavaScript (AJAX)**: To enable periodic data polling and displays alarm notifications and also to play alarm notification sounds.

```
Debug: IV length: 16, Ciphertext length: 96
Debug: Decrypted padded length: 96
Debug: Successfully removed padding
Debug: Final decrypted string: {"temperature":"24.5","humidity":"34.0","timestamp":"1762127198","team_number":"9"}
Successfully decrypted data: Temp=24.5, Hum=34.0
SENSOR DATA RECEIVED (ESP32)
{
  "team_number": "9",
  "temperature": 24.5,
  "humidity": 34.0,
  "timestamp": 1762127198,
  "timestamp_iso": "2025-11-02 17:46:38",
  "encrypted": true,
  "source": "decrypted"
}
172.20.10.3 - - [02/Nov/2025 17:46:39] "POST /api/sensor-data HTTP/1.1" 200 -
Debug: Combined data length: 112 bytes
Debug: IV length: 16, Ciphertext length: 96
Debug: Decrypted padded length: 96
Debug: Successfully removed padding
Debug: Final decrypted string: {"temperature":"24.1","humidity":"34.0","timestamp":"1762127209","team_number":"9"}
Successfully decrypted data: Temp=24.1, Hum=34.0
SENSOR DATA RECEIVED (ESP32)
{
  "team_number": "9",
  "temperature": 24.1,
  "humidity": 34.0,
  "timestamp": 1762127209,
  "timestamp_iso": "2025-11-02 17:46:49",
  "encrypted": true,
  "source": "decrypted"
}
172.20.10.3 - - [02/Nov/2025 17:46:49] "POST /api/sensor-data HTTP/1.1" 200 -
```

5. PostgreSQL Database Setup

- **Database Creation and Configuration:** Firstly, install PostgreSQL and created 'iotdb' database using ‘pgAdmin’, and configure the created db with host 'localhost', port 5432,

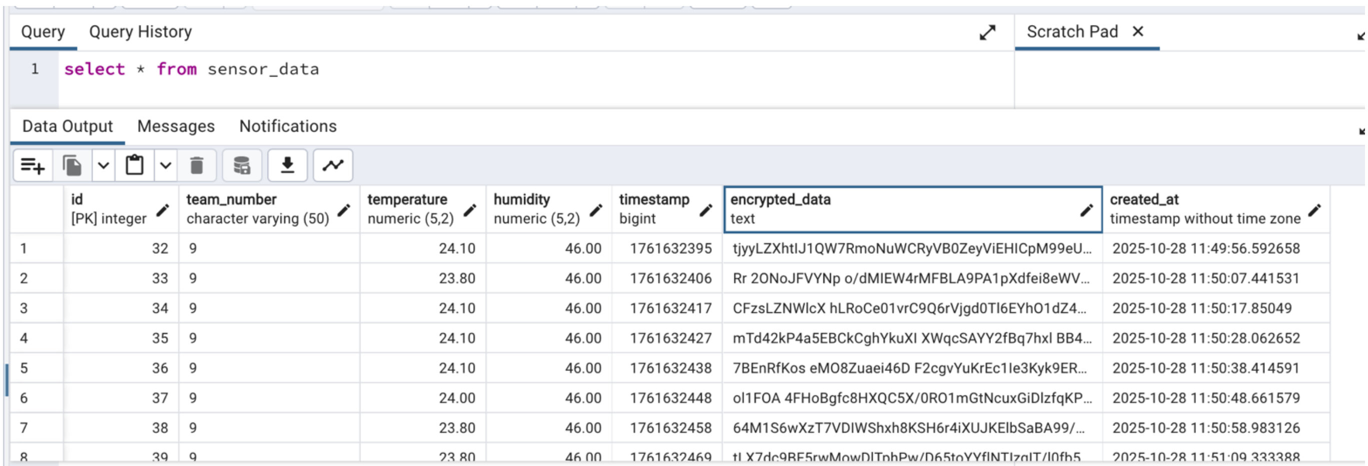
then create new user 'iotuser' and set a password for secure connections with server.py.
Note: whatever the user password is used in database use the same in server.py.

- **Sensor data table**: The ‘sensor_data’ table is created in database to store the sensor readings. Each record has a unique identifier (id) and includes the ‘team_number’. This table stores both decrypted values of temperature and humidity, and the original ‘encrypted_data’ for security and verification purposes. A ‘timestamp’ field records about when the sensor reading was taken, while the ‘created_at’ column has the logs when the data was inserted into the database. This table structure allows efficient tracking, retrieval, and analysis of sensor data while maintaining security.
- **Users table:** The ‘users’ table is created to manage authentication and track user sessions. Each user has a unique ‘id’ and ‘username’, with the encrypted password securely stored in ‘password_hash’ column. The ‘role’ column specifies the user’s permissions, while ‘active_sessions’ column keeps track of the number of ongoing sessions for that user. The ‘created_at’ field records when the account was created, enabling secure and organized management of user access and session history.

- **Indexes for Optimization:** To improve database performance, indexes were added on the ‘sensor_data’ table for ‘timestamp’ and ‘team_number’ columns, to enable faster retrieval of historical sensor readings. Similarly, an index on the ‘users’ table for ‘username’ column to allow quick lookups during authentication. These indexes are used to reduce query times, especially when working with large datasets.

- **Security Measures:** Passwords are securely hashed using SHA-256 for user credentials, and also to store encrypted sensor data using AES, this helps to prevent unauthorized access or modifications for 'iotuser' db users.

- **Database integration with Server:** The ‘server.py’ script automatically sets up the database by initializing database tables on startup using ‘init_database()’, and the server manages PostgreSQL connections through ‘psycopg2’ with proper error handling and automatically inserts a default admin user if one does not exist and it check the status of database for immediate use.




```
select * from sensor_data
```

	id [PK] integer	team_number character varying (50)	temperature numeric (5,2)	humidity numeric (5,2)	timestamp bigint	encrypted_data text	created_at timestamp without time zone
1	32	9	24.10	46.00	1761632395	tjyyLZXhtIJ1QW7RmoNuWCRyVB0ZeyViEHICpM99eU...	2025-10-28 11:49:56.592658
2	33	9	23.80	46.00	1761632406	Rr 2ONoJFVYNp o/dMIEW4rMFBLA9PA1pXdfei8eWV...	2025-10-28 11:50:07.441531
3	34	9	24.10	46.00	1761632417	CFzsLZNWlcX hLRoCe01vrC9Q6rVjgd0Tl6EYhO1dZ4...	2025-10-28 11:50:17.85049
4	35	9	24.10	46.00	1761632427	mTd42kP4a5EBCkCghYkuXI XWqcSAYY2fBq7hxl BB4...	2025-10-28 11:50:28.062652
5	36	9	24.10	46.00	1761632438	7BEnRfKos eMO8Zuaei46D F2cgvYuKrEc1Ie3Kyk9ER...	2025-10-28 11:50:38.414591
6	37	9	24.00	46.00	1761632448	ol1FOA 4FHoBgfc8HXQC5X/0RO1mGtNcuxGiDlzfqKP...	2025-10-28 11:50:48.661579
7	38	9	23.80	46.00	1761632458	64M1S6wXzT7VDIWShxh8KSH6r4iXUJKElbSaBA99/...	2025-10-28 11:50:58.983126
8	39	9	23.80	46.00	1761632469	tLX7dc9BF5rwMowDlTnhPw/D65toYYfINTIzqIT/I0fh5	2025-10-28 11:51:09.333388

Query | Query History | Scratch Pad

```sql
select * from users
```

Data Output | Messages | Notifications

	id [PK] integer	username character varying (100)	password_hash character varying (255)	role character varying (50)	active_sessions integer	created_at timestamp without time zone
1	3	pavan	912967c63ae45de519f389ee69446153148fc6a63d20acc1541c...	user	0	2025-10-28 11:29:36.41746
2	1	admin	240be518fabd2724ddb6f04eeb1da5967448d7e831c08c8fa822...	admin	2	2025-10-28 11:16:40.672287

6. Implementation Details

6.1 ESP32 Setup and functionality

- Firstly, DHT11 sensor is connected to GPIO 4 to capture the temperature and humidity readings for every 5 seconds, and here we used 'millis()' to prevent sensor delays.

- Then the sensor data converted to JSON object using 'ArduinoJson.h' library to ensure the structured payloads for encryption, and 'AES-128-CBC' encryption is performed with PKCS7 padding has applied to align data to 16 byte blocks and randomly generated 16 byte IV of ciphertext for security.

```c
// This is to return encrypted or plain data based on encryption setting
if (encryptEnabled) {
  // Encrypt and return Base64 encoded data
  size_t len = jsonStr.length();
  size_t padLen = ((len / 16) + 1) * 16;  // Round up to next 16-byte boundary

  unsigned char padded[padLen];
  memcpy(padded, jsonStr.c_str(), len); // To copy the JSON string to the padded array

  // PKCS7 padding to make the JSON string a multiple of 16 bytes
  unsigned char padValue = padLen - len;
  for (size_t i = len; i < padLen; i++) {
    padded[i] = padValue; // To set the padding value to the padded array
  }

  // To generate random IV
  unsigned char iv[16];
  unsigned char iv_copy[16];  // Store a copy of the IV before encryption
  for (int i = 0; i < 16; i++) {
    iv[i] = random(256);
  }

  // Save a copy of the IV before encryption (mbedtls modifies iv during encryption)
  memcpy(iv_copy, iv, 16);

  unsigned char ciphertext[padLen];
  mbedtls_aes_context aes;
  mbedtls_aes_init(&aes);
  mbedtls_aes_setkey_enc(&aes, encryptionKey, 128);
  mbedtls_aes_crypt_cbc(&aes, MBEDTLS_AES_ENCRYPT, padLen, iv, padded, ciphertext);
  mbedtls_aes_free(&aes);

  // Combine IV + ciphertext (use iv_copy instead of iv)
  unsigned char combined[16 + padLen];
  memcpy(combined, iv_copy, 16);  // Use iv_copy to ensure we have the original IV
  memcpy(combined + 16, ciphertext, padLen);
```

- After the encryption is completed, the encrypted Base64 string is uploaded to the server for every 10 seconds using HTTP POST command to '[http://[ESP32 IP]/api/sensor-data](http://[ESP32 IP]/api/sensor-data)' using 'HTTPClient.h' library.

```
17:46:38.818 -> Monitoring temperature & humidity started.
17:46:38.818 -> Temp: 24.50°C, Hum: 34.00%
17:46:38.818 -> === SENSOR READING & UPLOAD ===
17:46:38.849 -> Temperature: 24.5°C
17:46:38.849 -> Humidity: 34.0%
17:46:38.849 -> Timestamp:
17:46:38.849 -> WiFi Status: Connected
17:46:38.849 -> Encryption: Enabled
17:46:38.849 -> ==============================
17:46:38.849 -> Uploading data:
17:46:38.849 -> Team Number: 9
17:46:38.849 -> Server URL: http://172.20.10.2:8888/api/sensor-data
17:46:38.849 -> POST Data: team_number=9&temperature=24.5&humidity=34.0&timestamp=1762127198&is_encrypted=true&encrypted_data=Q8C546PARQrYjVbqJsFf3ow9tV/RuOAZOvsnqzajD0BJk2Otr
17:46:39.113 -> Server response: {"message":"Data received and stored","status":"success","timestamp":1762127198}
17:46:39.146 ->
17:46:43.896 -> Temp: 24.10°C, Hum: 34.00%
17:46:48.982 -> Temp: 24.10°C, Hum: 34.00%
17:46:49.174 -> === SENSOR READING & UPLOAD ===
17:46:49.174 -> Temperature: 24.1°C
17:46:49.174 -> Humidity: 34.0%
17:46:49.174 -> Timestamp: 1762127198
17:46:49.174 -> WiFi Status: Connected
17:46:49.174 -> Encryption: Enabled
17:46:49.174 -> ==============================
17:46:49.174 -> Uploading data:
17:46:49.174 -> Team Number: 9
17:46:49.205 -> Server URL: http://172.20.10.2:8888/api/sensor-data
17:46:49.205 -> POST Data: team_number=9&temperature=24.1&humidity=34.0&timestamp=1762127209&is_encrypted=true&encrypted_data=i8ioEpxCaShudO3kI2mIEcZAYHPsny5n9hZ9XUznoD2iihjn1
17:46:49.337 -> Server response: {"message":"Data received and stored","status":"success","timestamp":1762127209}
17:46:49.337 ->
```

- Also we included a retry mechanism which is to make 3 attempts for every 2 seconds if the upload is not success, this helps to overcome network interruptions

- Implemented RESTful API server on ESP32 to enable remote device management over the network without serial access and it runs on port 80, below are the example commands,

  1) curl -X POST http://[ESP32_IP]/start (this command to start the ESP32 operation.)

  2) curl -X POST http://ESP32_IP/stop (to stop ESP32)

  3) curl -X POST http://192.168.1.213/toggle-encryption (to enable or disable the encryption)

  4) curl -X POST http://ESP32_IP/push-now (to immediate push data to server)

  5) curl http://ESP32_IP/status (to check the status of ESP32)

  6) curl http://ESP32_IP/sensor (to check sensor information)

  7) curl http://ESP32_IP/config (to check the current configuration)

  8) curl http://ESP32_IP/health (to check the health status of ESP32)

  Below command to set or update the configuration,

  9)
```
curl -X POST http://ESP32_IP/config \
  -H "Content-Type: application/json" \
  -d '{
     "upload_interval": 10000,
     "encryption_enabled": true,
     "team_number": "9",
     "server_url": "http://192.168.1.100:8888/api/sensor-data"
    }'
```

- All the above commands respond in JSON format.

```
PROBLEMS 5    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/health
{"ok": true, "uptime_s": 1237}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/stop
Not found: /stop%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl -X POST http://172.20.10.3/stop
{"status":"success","message":"Monitoring stopped"}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl -X POST http://172.20.10.3/toggle-encryption
{"status":"success","encryption_enabled":false,"message":"Encryption toggled"}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/status
{"monitoring":false,"wifi_connected":true,"ip_address":"172.20.10.3","encryption_enabled":false,"team_number":"9","uploa
d_interval":10000}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl -X POST http://172.20.10.3/start
{"status":"success","message":"Monitoring started"}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/status
{"monitoring":true,"wifi_connected":true,"ip_address":"172.20.10.3","encryption_enabled":false,"team_number":"9","upload
_interval":10000}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/sensor
{"temperature":"24.1","humidity":"34.0","timestamp":"1762127900","team_number":"9"}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl -X POST http://172.20.10.3/toggle-encryption
{"status":"success","encryption_enabled":true,"message":"Encryption toggled"}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/sensor
+8FcWgzmv9AeJ18NOsQ0f5Q6A3z8l59eYL/nkriwTi5mDhByxAxL2AhqV73TvbQ0kjxJunME9JKKcfxQ/JMsDmhLQbjrjCL9xy9cGeFWUwJ2WNUicjsmr2m0
31px/Lyuy7qgbt5NgVbQSzVzHsLZOA==%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 % curl http://172.20.10.3/config
{"upload_interval":10000,"encryption_enabled":true,"team_number":"9","server_url":"http://172.20.10.2:8888/api/sensor-da
ta","monitoring":true}%
(base) pavankalam@PAVANs-MacBook-Pro mini-project-4 %
```

- During network connection the ESP32 wait for 30 seconds to get the input from the user, if the responses from the user is incorrect, then the system will auto restart and show all the available networks to connect, along with the SSID and RSSI to help users to connect to strong signal network.

```
// This is to prepare HTTP POST and send data to server (initialize with server URL)
HTTPClient http;
http.begin(serverUrl); // To initialize the HTTP client with the server URL
http.addHeader("Content-Type", "application/x-www-form-urlencoded");

// Logic to validate team number for data integrity
if (teamNumber.length() == 0) {
  Serial.println("ERROR: Invalid team number: '" + teamNumber + "'");
  Serial.println("Please use 'w' command to set proper team number");
  http.end();
  return;
}

// This is to build encoded data string for HTTP POST
String postData = "team_number=" + teamNumber +
                  "&temperature=" + String(temperature, 1) +
                  "&humidity=" + String(humidity, 1) +
                  "&timestamp=" + lastTimestamp +
                  "&is_encrypted=" + (encryptEnabled ? "true" : "false");
if (encryptEnabled) {
  postData += "&encrypted_data=" + encryptedData; // To add the encrypted data to the POST data
}
```

```
ESP32-WROOM-DA M...
mini-project-4_esp32.ino
#include <mbedtls/aes.h>          // AES encryption functions from mbedTLS library
#include <mbedtls/base64.h>       // Base64 encoding for secure data transmission

// To setup the pin and sensor
#define DHT_PIN 4
#define LED_PIN 2
#define SENSOR_TYPE DHT11

// To setup the default threshold ranges for both temperature and humidity
float low_threshold = 15.0;
```

```
Output    Serial Monitor
Message (Enter to send message to 'ESP32-WROOM-DA Module' on '/dev/cu.usbserial-1120')    New Line    115200 baud
17:35:54.717 -> ESP32 Mini-Project #4 Starting...
17:35:54.717 -> Waiting for serial input...
17:35:54.717 -> Scanning for available networks...
17:35:57.581 -> Found 21 networks:
17:35:57.581 -> 1: Pavan (RSSI: -57)
17:35:57.613 -> 2: UMKC Guest (RSSI: -68)
17:35:57.613 -> 3: UMKCWPA (RSSI: -68)
17:35:57.613 -> 4: UMKC-Media (RSSI: -68)
17:35:57.613 -> 5: UMKC Guest (RSSI: -73)
17:35:57.613 -> 6: UMKC-Media (RSSI: -73)
17:35:57.613 -> 7: UMKCWPA (RSSI: -76)
17:35:57.613 -> 8: UMKC-Media (RSSI: -76)
17:35:57.613 -> 9: UMKC Guest (RSSI: -76)
17:35:57.613 -> 10: DIRECT-5202405C (RSSI: -82)
17:35:57.613 -> 11: UMKC-Media (RSSI: -83)
17:35:57.613 -> 12: UMKCWPA (RSSI: -83)
17:35:57.613 -> 13: UMKC Guest (RSSI: -84)
17:35:57.613 -> 14: UMKCWPA (RSSI: -84)
17:35:57.613 -> 15: UMKCWPA (RSSI: -84)
17:35:57.646 -> 16: UMKC-Media (RSSI: -85)
17:35:57.646 -> 17: UMKC Guest (RSSI: -85)
17:35:57.646 -> 18: UMKC Guest (RSSI: -85)
17:35:57.646 -> 19: UMKCWPA (RSSI: -86)
17:35:57.646 -> 20: UMKC-Media (RSSI: -86)
17:35:57.646 -> 21: UMKC-Media (RSSI: -92)
17:35:57.646 ->
17:35:57.646 -> Select WiFi network:
17:35:57.646 -> Option 1: Enter network number (1-21)
17:35:57.646 -> Option 2: Enter full SSID manually
17:35:57.646 -> Your choice:
ESP32-WROOM-DA Module on /dev/cu.usbserial-1120
```

```
36:27.652 -> Enter Server URL (e.g., http://192.168.1.100:8888/api/sensor-data): ....Received: http://172.20.10.2:8888/api/sensor-data
36:46.434 -> Enter Team Number: .Received: 9
36:50.230 -> Connecting to WiFi.....
36:52.238 -> Connected to WiFi
36:52.238 -> IP Address: 172.20.10.3
36:52.238 -> NTP time synchronized
36:52.238 -> RESTful API server started
36:52.272 ->
36:52.272 -> --- Menu ---
36:52.272 -> s - Start monitoring
36:52.272 -> t - Stop monitoring
36:52.272 -> c - Calibrate sensor
36:52.272 -> h <value> - Humidity high threshold
36:52.272 -> L <value> - Humidity low threshold
36:52.272 -> l <value> - Temperature low threshold
36:52.272 -> u <value> - Temperature high threshold
36:52.272 -> e - Toggle encryption
36:52.272 -> w - Change WiFi settings
36:52.272 -> i - Show current info
36:52.272 -> n <number> - Set team number
36:52.272 -> m - Show menu
36:52.272 -> Thresholds - Temp: 15.00-30.00°C, Hum: 30.00-70.00%
36:52.304 -> Calibrated: No
36:52.304 -> Monitoring: Stopped
36:52.304 -> WiFi: Connected
36:52.304 -> Encryption: Enabled
36:52.304 -> RESTful API: http://172.20.10.3
36:52.304 -> ---
36:52.304 ->
36:52.304 -> Environment monitor is ready. Please Enter 'm' for options.
36:52.304 -> Setup complete. Starting loop...
```

6.2 Server Setup

- Here we implemented the Flask web server using ‘app = Flask(__name__)’ and also configured with Flask's built-in session management using cookies and it was secured by a secret key.

- Then we set the hardcoded values directly in the code for ‘PostgreSQL’ credentials like “host: 'localhost', database: 'iotdb', user: 'iotuser', password: 'iotpassword', port: 5432”, and ‘AES encryption’ key as (ENCRYPTION_KEY = b'MySecretKey12345'), and ‘OpenWeatherMap’ API key, the server directly uses these configurations without loading from external files.

```python
server.py > ...

# Setting up the Flask application
app = Flask(__name__)
app.secret_key = 'your-secret-key-change-in-production'  # Change this in production! (this is the secret key for the Flask application)

# PostgreSQL Database configuration and setting for the database connection
POSTGRES_CONFIG = {
    'host': 'localhost',
    'database': 'iotdb',
    'user': 'iotuser',
    'password': 'iotpassword',  # Use the same password as the one in the database or change this to your PostgreSQL password
    'port': 5432
}

# Below is the function to get the PostgreSQL database connection
def get_db_connection():
    """Get PostgreSQL database connection"""
    try:
        conn = psycopg2.connect(**POSTGRES_CONFIG)  # This is the connection to the database as per the given configuration
        return conn
    except psycopg2.OperationalError as e:
        print(f"❌ Database connection error: {e}")
        sys.exit(1)

# Below is the AES encryption configuration (must match ESP32)
ENCRYPTION_KEY = b'MySecretKey12345'  # 16 bytes key for AES-128
BLOCK_SIZE = 16
```

- After configuration, the server defines all the API routes directly using ‘@app.route()’, and includes ‘/api/sensor-data’ for data receiving from ESP32, ‘/login’ for authentication , and ‘/dashboard’ for visualizing the complete webpage with all the information, and once the login is success in one platform then all the API endpoints like ‘/api/current-data’, ‘/api/historical-data’ etc will proceed with no CORS configuration as the application assumes same-origin access, but the same endpoints won’t work in different browser unless the user login again.

- The server includes error handling via try-except blocks within individual routes, ‘using print()’ statements for logging errors and debug information to the server running console itself.

```python
# Below is the route to receive the sensor data from the ESP32
# Also to receive the sensor data from the ESP32 and store it in the database and also to decrypt the data
@app.route('/api/sensor-data', methods=['POST'])
def receive_sensor_data():
    """Receive encrypted sensor data from ESP32"""
    try:
        data = request.form

        # Extract raw values
        team_number = data.get('team_number')
        temperature = float(data.get('temperature', 0))
        humidity = float(data.get('humidity', 0))
        timestamp = int(data.get('timestamp', 0))
        is_encrypted = data.get('is_encrypted', 'false').lower() == 'true'
        encrypted_data = data.get('encrypted_data', '')

```

6.3 Authentication and Authorization

- The user credentials are stored in database under ‘users’ table with ‘SHA-256’ hashed passwords’, and every user has a role field set to 'user' by default or 'admin' for selected accounts and this can be done by the admin user in the dashboard.

- Once all this is complete, during the account login the user input password will be verified with the hashed password stored in the database using ‘verify_password(password, password_hash)’.

```python
# Below is to verify the password with the hashed password
if user and verify_password(password, user['password_hash']):
    # To update the active sessions for the user in the database for every login
    conn = get_db_connection()
    iotdb = conn.cursor()
    iotdb.execute('UPDATE users SET active_sessions = active_sessions + 1 WHERE id = %s', (user['id'],))
    conn.commit()
    conn.close()

    session['user_id'] = user['id']
    session['username'] = username
    session['role'] = user['role']
    flash('Login successful!', 'success')
    return redirect(url_for('dashboard'))
else:
    flash('Invalid username or password!', 'error')
```

- After successful verification, Flask's built-in session management is used to store session[‘user_id’], session['username'], and session['role'], through server-side session storage secured by the app's secret key.

- Also we implemented role-based access control by checking session.get('role') != 'admin' directly inside admin-only route functions, returning JSON error {'error': 'Unauthorized'} with 403 status if not admin, without a separate @admin_required decorator.

```python
# Below is the route to the admin page
# This is to display the admin page and handle the admin requests and also to check if the user is an admin
@app.route('/admin')
@login_required
def admin():
    """Admin page for user management and analytics"""
    if session.get('role') != 'admin':
        flash('Access denied: Admin privileges required', 'error')
        return redirect(url_for('dashboard'))

    return render_template('admin.html',
                           username=session.get('username'),
                           role=session.get('role'))
```

- Unauthorized access attempts are handled by redirecting the request to ‘/login’ with message like "Access denied: Admin privileges required", and errors are printed to console.

6.4 Database Initialization and Access

- For database we used ‘PostgreSQL’ with the ‘iotdb’ as the database name, and it was accessed by the server using ‘psycopg2.connect(**POSTGRES_CONFIG)’, and all the database connection parameters are configured in the server, ‘init_database()’ function runs on server startup and create tables if they don't exist.

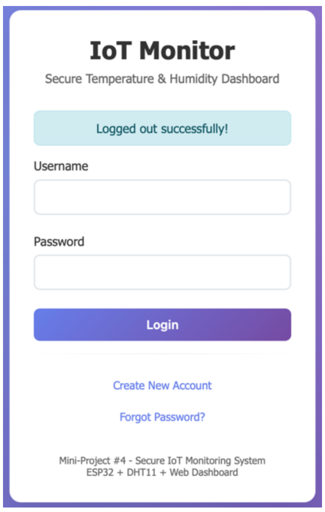
- After table are created, indexes are added using ‘CREATE INDEX IF NOT EXISTS’ on ‘idx_sensor_timestamp’(sensor_data.timestamp), ‘idx_sensor_team’ (sensor_data.team_number), and ‘idx_users_username’ (users.username) to optimize queries and speed up the database communication when the we have huge data.

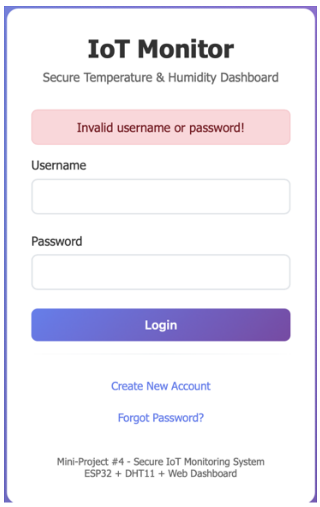
```
# Below is to create indexes for better performance and for faster retrieval of data
iotdb.execute('CREATE INDEX IF NOT EXISTS idx_sensor_timestamp ON sensor_data(timestamp)')
iotdb.execute('CREATE INDEX IF NOT EXISTS idx_sensor_team ON sensor_data(team_number)')
iotdb.execute('CREATE INDEX IF NOT EXISTS idx_users_username ON users(username)')
```

- Here the database connections are created fresh for each operation using ‘get_db_connection()’, with try-except blocks catching ‘psycopg2.OperationalError’ and printing errors on the console.

6.5 Login and Forgot Password

- Once open the server hosting url, the ‘/login’ route directly open the ‘login.html’ with a form of user inputs with the fields ‘username’ and ‘password’ and then the user inputs will be verified in the database for ‘password hash’ using ‘verify_password()’ function, and sets session variables session['user_id'], session['username'], session['role'] only if the user details are valid, and redirects to ‘/dashboard’ with message saying 'success' or 'error' for "Invalid username or password".



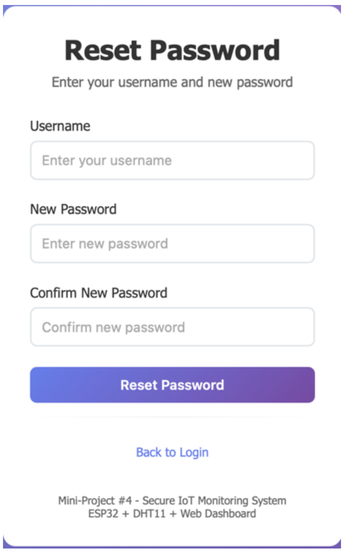


```python
# This function is to hash the password using SHA-256
def hash_password(password):
    """Hash password using SHA-256 (in production, use bcrypt or Argon2)"""
    return hashlib.sha256(password.encode()).hexdigest()

# This function is to verify the password with the hashed password
def verify_password(password, password_hash):
    """Verify password against hash"""
    return hash_password(password) == password_hash
```

- After login, the active_sessions count is incremented using:

```python
# Below is the route to the login page
# This is to display the login page and handle the login form submission
@app.route('/login', methods=['GET', 'POST'])
def login():
    """User login page"""
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        # Below is to commit the database connection and also to execute the query to get the user information from the database
        conn = get_db_connection()
        iotdb = conn.cursor(cursor_factory=RealDictCursor)
        iotdb.execute('SELECT id, password_hash, role, active_sessions FROM users WHERE username = %s', (username,))
        user = iotdb.fetchone()
        conn.close()
```

- Also we implemented 'forgot password' option the main login page for immediate change of user password, this option will redirect to forgot_password.html, and it open a form with the required details, 'username, new_password, and confirm_password', then the server validates the existing of the username in the database and update and save the new password with 'hash_password()' in the db, and also update the user 'active_sessions' to '0', then immediately redirect to 'login' page with the message 'Password reset successfully".




```python
# This will also hit the direct change in the database to update the password
@app.route('/forgot-password', methods=['GET', 'POST'])
def forgot_password():
    """Forgot password page to reset user password"""
    if request.method == 'POST':
        username = request.form.get('username')
        new_password = request.form.get('new_password')
        confirm_password = request.form.get('confirm_password')

        if not username or not new_password or not confirm_password:
            flash('All fields are required!', 'error')
            return render_template('forgot_password.html')

        if new_password != confirm_password:
            flash('Passwords do not match!', 'error')
            return render_template('forgot_password.html')

        # Below is to update the password in the database
        conn = get_db_connection()
        iotdb = conn.cursor()
        password_hash = hash_password(new_password)
        iotdb.execute('UPDATE users SET password_hash = %s WHERE username = %s', (password_hash, username))
        conn.commit()
        affected_rows = iotdb.rowcount
        conn.close()

        # Logic to check if the password was updated successfully
        if affected_rows > 0:
            flash('Password updated successfully! Please login with your new password.', 'success')
            return redirect(url_for('login'))
        else:
            flash('Username not found!', 'error')
            return render_template('forgot_password.html')

    return render_template('forgot_password.html')
```

6.6 RESTful API's (Server side)

- **/api/search-data:** The below url is to search data in the database with either above or below the threshold limit and with either 'temperature or humidity' metric and the team number is optional.
  http://Server_IP:8888/api/search-data?threshold=23&comparison=above&metric=temperature&team=7

192.168.1.222

[{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632910},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632899},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632889},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632879},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632869},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632858},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632848},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632838},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632828},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632817},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632807},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632797},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632787},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632777},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632766},{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632756},
{"humidity":46.0,"team_number":"7","temperature":23.8,"timestamp":1761632745}]

- **/api/current-data:** The below url is for fetching the current data, this will update for every new upload by the ESP32 to server and this can be updated for every 10 seconds.
  http://server_IP:8888/api/current-data

192.168.1.222

{"humidity":20.0,"team_number":"9","temperature":25.3,"timestamp":1761943119}

- **/api/historical-data:** The below is the url for fetching all the historical data stored in the database,
  http://server_IP:8888/api/historical-data

192.168.1.222

[{"humidity":20.0,"team_number":"9","temperature":25.3,"timestamp":1761943119},{"humidity":20.0,"team_number":"9","temperature":25.3,"timestamp":1761943109},
{"humidity":20.0,"team_number":"9","temperature":25.3,"timestamp":1761943099},{"humidity":20.0,"team_number":"9","temperature":25.8,"timestamp":1761943089},
{"humidity":19.0,"team_number":"9","temperature":25.8,"timestamp":1761943079},{"humidity":20.0,"team_number":"9","temperature":25.8,"timestamp":1761943065},
{"humidity":20.0,"team_number":"9","temperature":26.1,"timestamp":1761943055},{"humidity":20.0,"team_number":"9","temperature":26.2,"timestamp":1761943044},
{"humidity":21.0,"team_number":"9","temperature":26.7,"timestamp":1761943034},{"humidity":-4.9,"team_number":"9","temperature":26.7,"timestamp":1761942972},
{"humidity":-4.9,"team_number":"9","temperature":26.7,"timestamp":1761942961},{"humidity":-4.9,"team_number":"9","temperature":26.7,"timestamp":1761942951},
{"humidity":-4.9,"team_number":"9","temperature":26.7,"timestamp":1761942941},{"humidity":-4.9,"team_number":"9","temperature":26.7,"timestamp":1761942931},
{"humidity":-6.9,"team_number":"9","temperature":26.7,"timestamp":1761942921},{"humidity":-6.9,"team_number":"9","temperature":26.7,"timestamp":1761942910},
{"humidity":23.0,"team_number":"9","temperature":26.2,"timestamp":1761942879},{"humidity":23.0,"team_number":"9","temperature":26.2,"timestamp":1761942869},
{"humidity":23.0,"team_number":"9","temperature":26.2,"timestamp":1761942859},{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942849},
{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942744},{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942733},
{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942723},{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942713},
{"humidity":23.0,"team_number":"9","temperature":25.8,"timestamp":1761942703},{"humidity":23.0,"team_number":"9","temperature":25.3,"timestamp":1761942692},

- **/api/users:** The below url shows all the users information in the database like users role, created time, current active_sessions as shown the below screenshot,
  http://server_IP:8888/api/users

192.168.1.222

[{"active_sessions":0,"created_at":"Tue, 28 Oct 2025 11:29:36 GMT","id":3,"role":"user","username":"pavan"},{"active_sessions":2,"created_at":"Tue, 28 Oct 2025 11:16:40 GMT","id":1,"role":"admin","username":"admin"}]

- **/api/dashboard-stats:** This api will show the current login user details like, the current user active_sessions, and the efficiency of user connection to database and total number of records.
  http://server_IP:8888/api/users

192.168.1.222

{"active_connections":2,"query_efficiency":100,"throughput":0,"total_records":298}

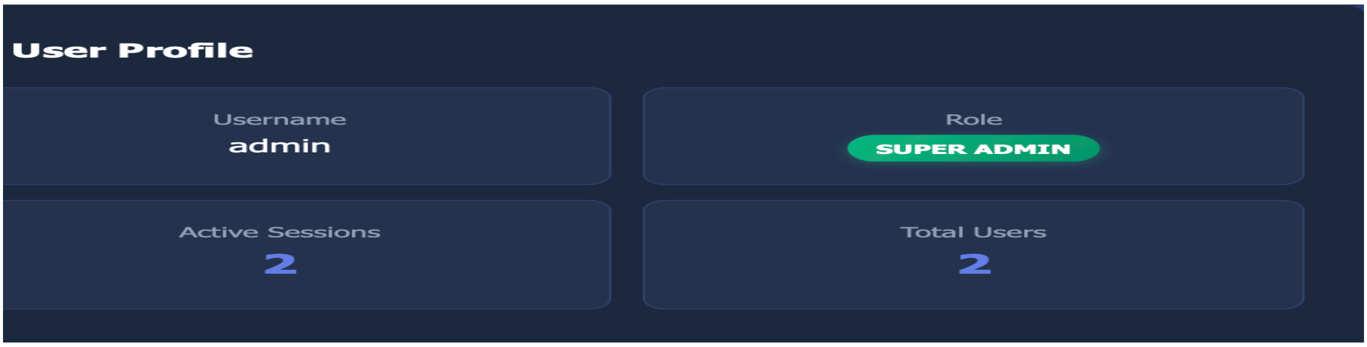
6.7 User Active Session

- First every use authentication request is protected by ‘@login_required’ decorator to check if the user is already login or not.
- After every successful login of the defined user, the ‘active_sessions’ column in users table tracks concurrent logins by incrementing the active_sessions count by 1, and similarly for logout it will decrement the count by 1, this can be done directly through SQL queries in the server.

```
# To update the active sessions for the user in the database for every login
conn = get_db_connection()
iotdb = conn.cursor()
iotdb.execute('UPDATE users SET active_sessions = active_sessions + 1 WHERE id = %s', (user['id'],))
conn.commit()
conn.close()
```

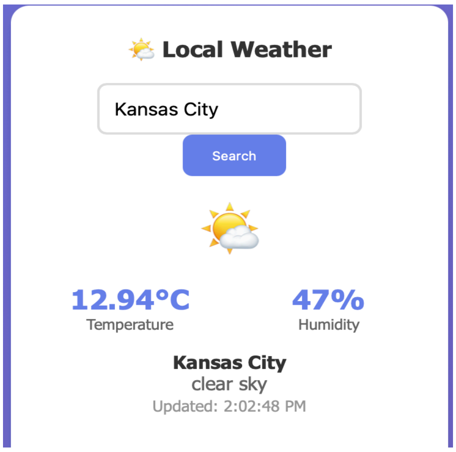
```
# Below is to decrement the active sessions for the user in the database for every logout
if 'user_id' in session:
    try:
        conn = get_db_connection()
        iotdb = conn.cursor()
        iotdb.execute('UPDATE users SET active_sessions = GREATEST(active_sessions - 1, 0) WHERE id = %s', (session['user_id'],))
        conn.commit()
        conn.close()
    except Exception as e:
        print(f"Error updating active sessions on logout: {e}")
```

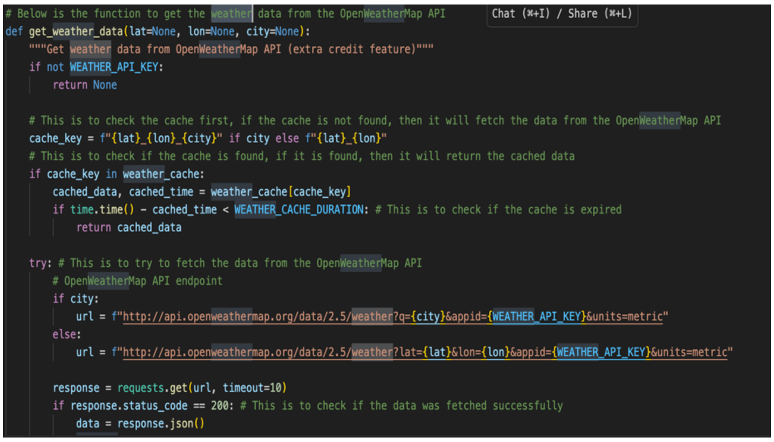
- Also, we included ‘Admin Panel’ for admin users to check the total ‘active_sessions’ of the current user and also every users active logins.



6.8 Weather Condition

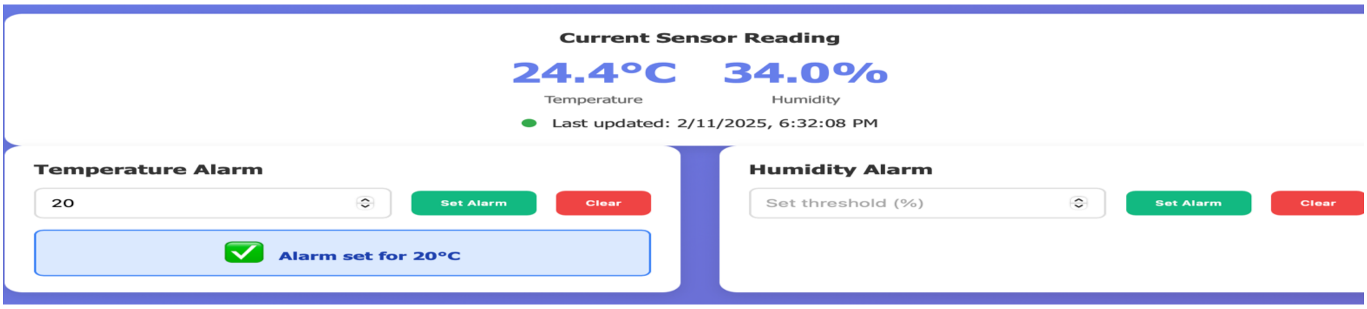
- To get the real-time weather details of any place in the world we use ‘OpenWeatherMap’ at ‘/api/weather’ route.
- To use this feature we hardcoded the ‘weather_api_key’ in the server code to utilize the application services, also we included that, the user can search the city name or any geographical ‘longitude and latitude’ directly in the dashboard.
- After receiving the response from the user, the ‘weather_api’ will fetch and display the requested location weather details like, temperature, humidity along with the weather icon, and this information will be stored in cache for 10 minutes.

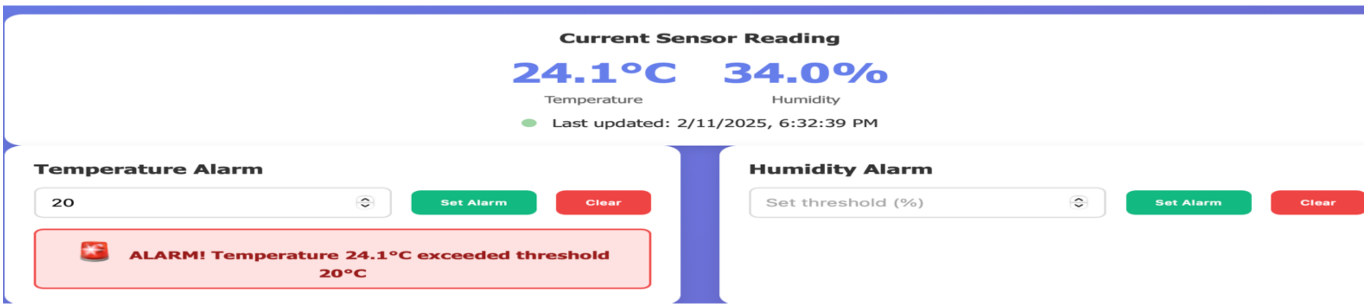




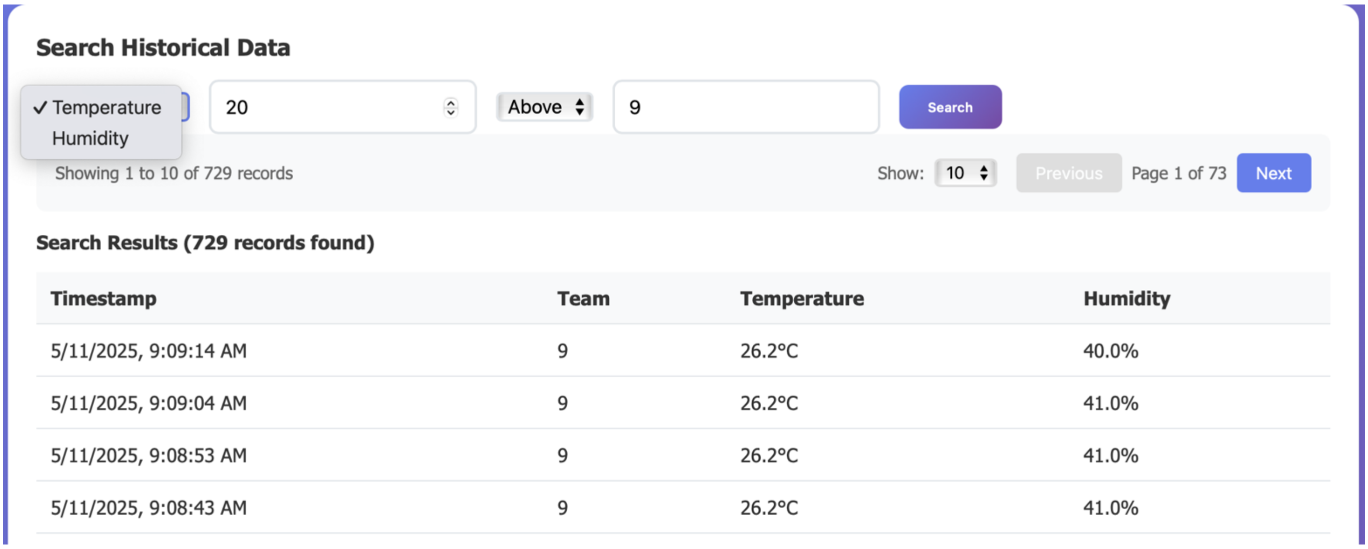
6.9 Alert, Search and Historical data Identification:

- For ‘alerts’ the user has to set the temperature or humidity thresholds as input in the dashboard, and then the system will save the details in the browser local storage.
- For every data retrieval from the sensor the system will check for the alert threshold limit, once the temperature or humidity reaches the threshold limit then immediately the browser will display the alert message and also play the alert sound. Also, the user can clear the alert limit in the dashboard.





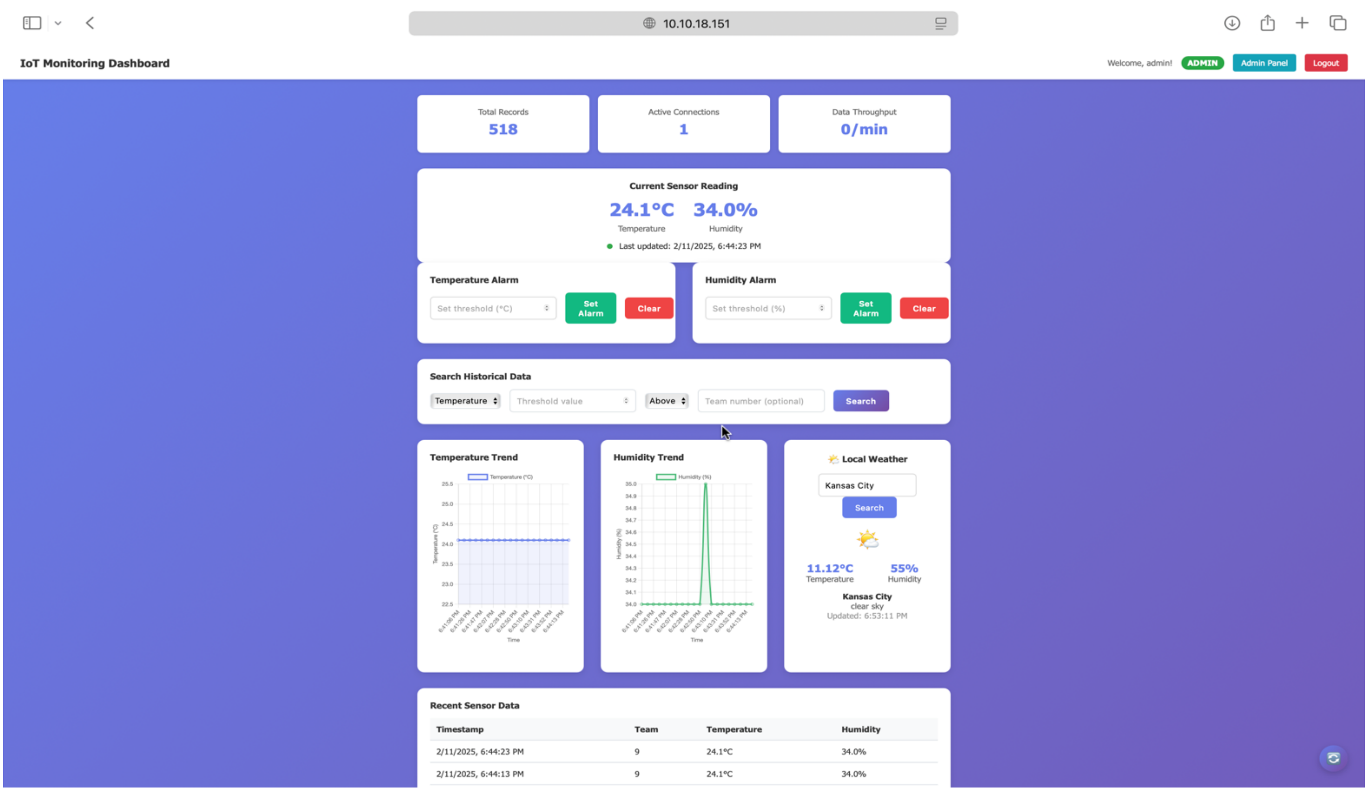
- In the dashboard, the user can directly search the historical data through by selecting temperature or humidity range, as shown in the below screenshot.

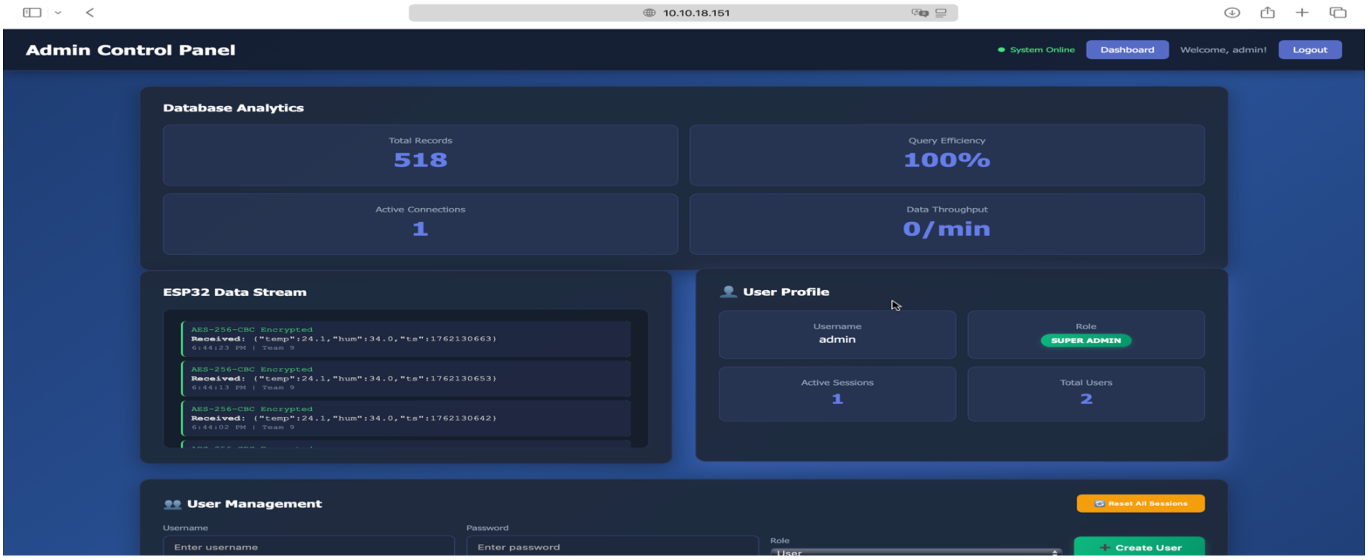


6.10 Admin and User Control Access

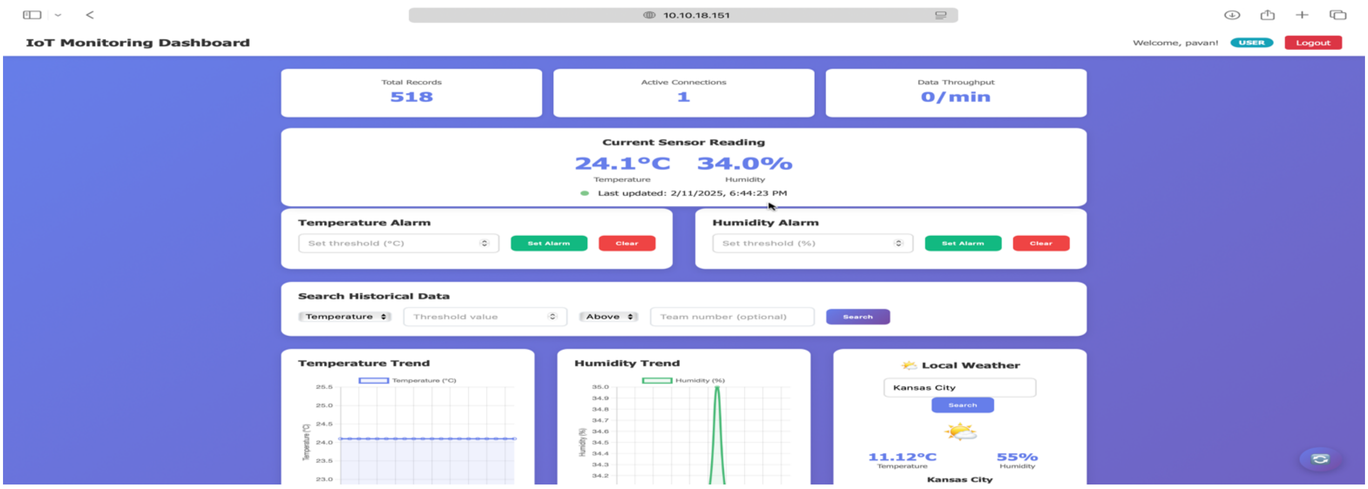
- We had implemented two types of roles 'admin and user', if the user login as admin then the system will load the 'admin.html' for extra features, like listing all the users information from the database like their username, role, active sessions and also user creation time, along with it admin user can able to reset all the user sessions.

  Below is the screenshot of admin and admin panel pages,

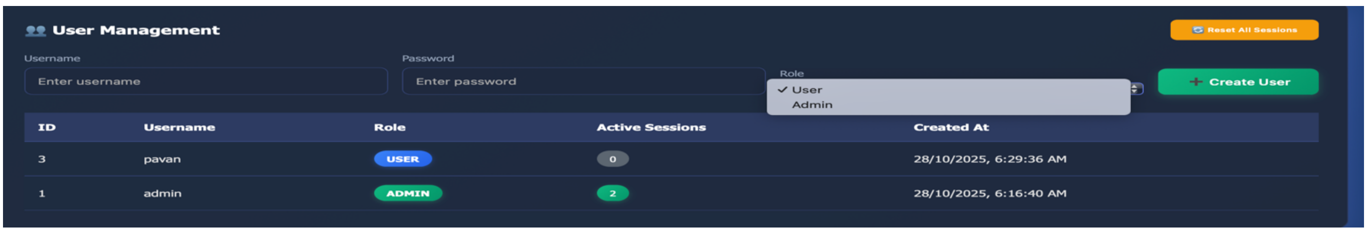




Below is the screenshot of non-admin just ‘user’ page,



- Only the admins can create new users via a POST form in ‘admin.html’ with username, password, role fields, submitting to ‘/api/users’ which hashes password and inserts into DB.

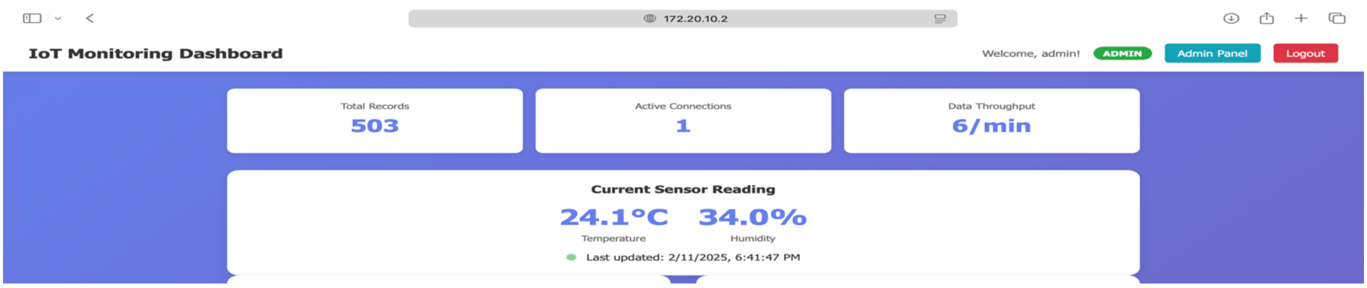


- Normal users cannot access /admin, if the normal user hit any admin accessible url’s, then the route checks “session.get('role') != 'admin' ” and returns 403 error, and auto redirect to dashboard with message "Access denied".

```python
# Below is the route to get all the users
# This is to get all the users from the database and return it as a JSON object
# This will return the user id, username, role, active sessions, and created at
@app.route('/api/users')
@login_required
def get_users():
    """Get all users (admin only)"""
    if session.get('role') != 'admin':
        return jsonify({'error': 'Unauthorized'}), 403

    # This is to get the database connection and also to get all the users from the database
    conn = get_db_connection()
    iotdb = conn.cursor(cursor_factory=RealDictCursor)
    iotdb.execute('SELECT id, username, role, active_sessions, created_at FROM users')
    users = iotdb.fetchall()
    conn.close()

    return jsonify([dict(user) for user in users])
```

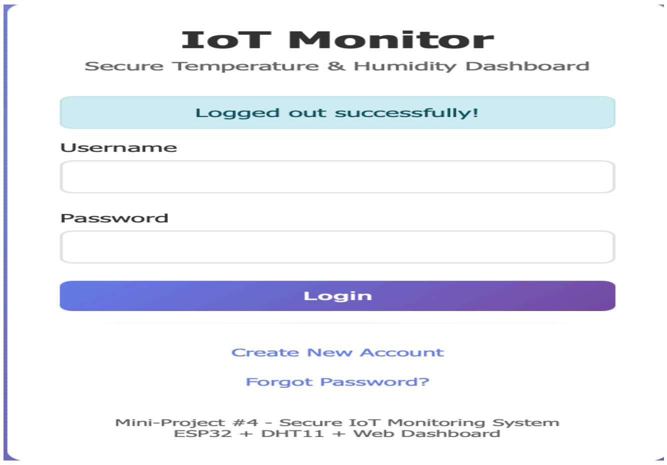
6.11 Logout Setup

- We implemented this 'logout' feature in the dashboard to logout any user from current session, and immediately update user session in the database, then immediately the system will redirect to 'login page'



- Also we included session validation in '@login_required' decorator to check session.get('user_id') on every api route, the system will redirect to login page if the particular user is not active.

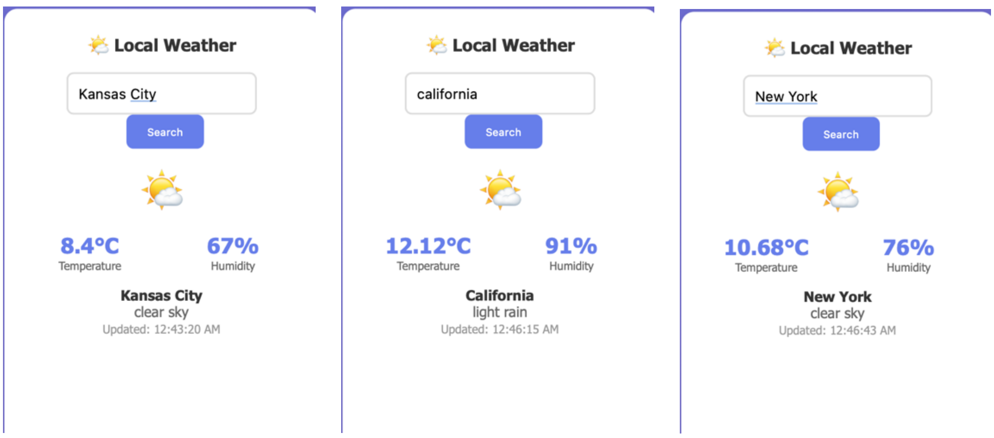
```python
# Below is the route to the logout page        Chat (⌘+I) / Share (⌘+L)
# This is to handle the logout form submission and clear the session
@app.route('/logout')
def logout():
    """User logout"""
    # Below is to decrement the active sessions for the user in the database for every logout
    if 'user_id' in session:
        try:
            conn = get_db_connection()
            iotdb = conn.cursor()
            iotdb.execute('UPDATE users SET active_sessions = GREATEST(active_sessions - 1, 0) WHERE id = %s', (session['user_id'],))
            conn.commit()
            conn.close()
        except Exception as e:
            print(f"Error updating active sessions on logout: {e}")

    session.clear()
    flash('Logged out successfully!', 'info')
    return redirect(url_for('login')) # This is to redirect to the login page after logout
```



- All protected routes like /dashboard, /admin, and APIs apply '@login_required', ensuring no access without valid session.

7. Testing complete system performance

- We tested by sending encrypted packets from ESP32 to server and verified decryption success rates, and tested HTTP POST in different WiFi networks.
- Also, we tested the authentication by attempting unauthorized dashboard access and checked web page redirects for different users.
- Performed brute-force logins to ensure hashing resistance, and validated session clearance on logout across multiple browsers.
- In dashboard we tested on filtering historical data, and also weather api caching and even tested weather conditions on different cities.



- Tested the alert notifications on different threshold limits, and also checked the updating of user active sessions in the database.
- Conducted the tests on all the RESTful api's of both ESP32 server and the dashboard server, and all returned the expected JSON format results.

8. Challenges Faced

- Initially we got encryption key mismatch between ESP32 mbedtls and server PyCrypto and they cause decryption errors and it got resolved by standardizing PKCS7 padding and IV handling, with Base64 cleaning for transmission artifacts.

- Also we implements a feature for dynamic selection of WiFi network on ESP32 with a connection waiting time of 30 seconds, this cause trouble to connect due to signal strength, and it got resolved by RSSI based network selection.

- We got PostgreSQL connection permission issues and this got resolved by adjusting the user privileges and configurations in db settings and also by adding error handling in server.py to handle connection failures.

- We got trouble on getting real time updates on dashboard and it was rectified by optimizing the requests to 10 seconds intervals and also this ensure smooth graph renders without server overload.

- While testing, we found concurrent user logins caused inconsistent session count, to resolve this we implemented automatic SQL operations on 'active_sessions' column to track the multiple user session with multiple logins.

9. Results and Conclusion

- Finally the system efficiently collects the temperature and humidity data from ESP32 and encrypts it before transmission, and decrypting the encrypted data on server and stored securely in PostgreSQL.

- The data transmission was occurred for every 10 seconds, and performed real time monitoring and real time graphs updates in the dashboard with effective performance of search filters on historic data, and even working alarm notifications on reaching effective user threshold limits.

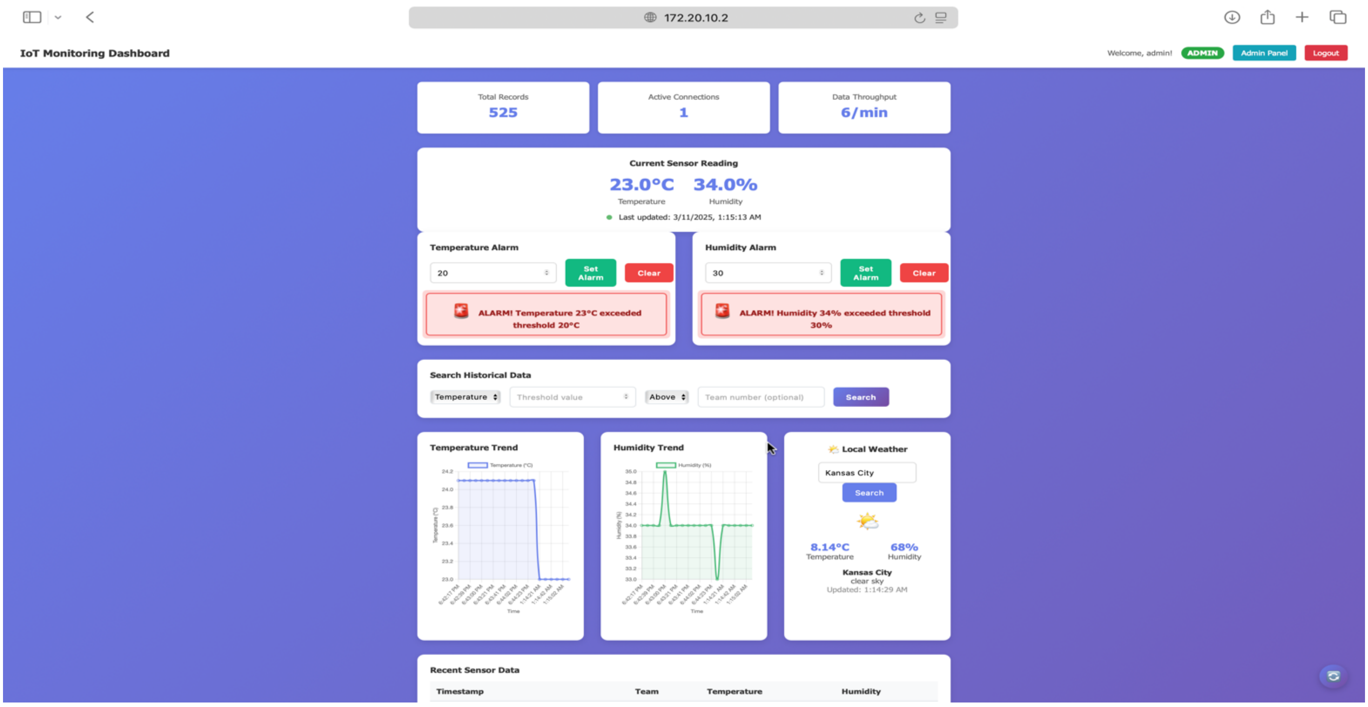
- AES-128-CBC encryption, SHA-256 password hashing, and session-based authentication effectively protects data, while admin panel handled user creation and resets efficiently.

- The system perfectly shows the weather information using cached API data, and the dashboard allow users to compare the weather api data with the sensor readings.

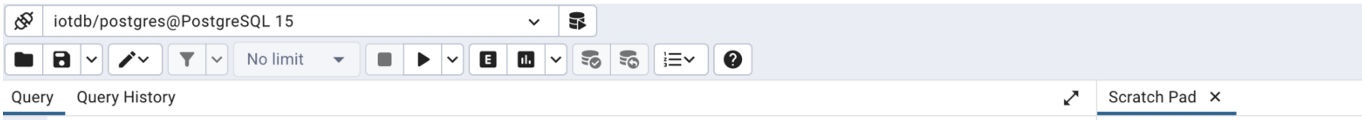
- Finally the system has successfully integrated IoT, web, database and security techniques to maintain the continuous system functionality without break.

Below is the screenshot showing the 'admin' user dashboard, with direct reflection of database holding total sensor reading records, along with current active sessions, and also displaying number of records server receiving per minute from ESP32. In next column, showing current sensor reading and user declared temperature and humidity thresholds levels and it's alert message. Next historical data search feature and real time graphical representation of the sensor readings and

OpenWeatherMap API integration. Also, providing ‘logout’ and ‘Admin Panel’ option for monitoring and controlling every user in the system.



Below screenshot shows the new sensor readings with the encrypted_data and it’s created time.




```sql
select * from sensor_data
order by created_at desc
limit 10
```

	id [PK] integer	team_number character varying (50)	temperature numeric (5,2)	humidity numeric (5,2)	timestamp bigint	encrypted_data text	created_at timestamp without time zone
1	560	9	23.00	34.00	1762154154	5Cw5fBLrhcotBJBzXApGcArfltUDffTz2vtwjmVsAc7W...	2025-11-03 12:45:54.525124
2	559	9	23.00	34.00	1762154144	ttzkbXirAARDYGL6Bgfynd wOSeN9TXa4B25QPNgUG...	2025-11-03 12:45:44.21458
3	558	9	23.00	34.00	1762154133	elv56nGRwENi 3vYqbclrfvVps7A8olDjpzO5E9U80ZO6...	2025-11-03 12:45:34.056529
4	557	9	23.00	34.00	1762154123	LgPn5Oin/QyPMvc1lsSYYUPkMUljWEsazcHmvOYxLq...	2025-11-03 12:45:23.901766
5	556	9	23.00	34.00	1762154113	vJVonxPPRDjDBzlp1GdY/MDItEnhS5BVhSiNY9PNFV...	2025-11-03 12:45:13.569451
6	555	9	23.00	34.00	1762154102	nK6jtTLQxSkqMhg/r7pB6qP7sV2KfHrbtBOD vPTn8uJ...	2025-11-03 12:45:03.133357
7	554	9	23.00	34.00	1762154092	bLnu4oF7gjIeMnvz67Qt8xIBOuKl2Qw0qF24xiESSZas...	2025-11-03 12:44:52.769919
8	553	9	23.00	34.00	1762154082	HUA1etiJCj/GmH2GoaF2Ifn lolzm2pyHq2RifcKVDYC...	2025-11-03 12:44:42.457105
9	552	9	23.00	34.00	1762154071	l1uOwOZFmW0G3EMHMC43LzGO79RZb1GtKmIobiv...	2025-11-03 12:44:32.079631