COMP-SCI 5577
Internet of Things
Mini-Project4 Report

Professor: Dr Qiuye He
Term members:
Hui Jin
Jiarui Zhu
Harshitha Bashyam
Apr 1, 2025
Github Link: https://github.com/nanxuanhui/Iot_miniproject.git
Demo Link:
https://drive.google.com/file/d/118sLmWfMxNjRDRaj91cCVT6VzHeMFQCr/view?usp=sharing

Content

Introduction

This project is a real-time temperature and humidity monitoring system based on ESP32, including the hardware side ESP32 and the software side front-end and back-end systems. The system can realize real-time collection, encrypted t ransmission, data storage, data analysis and front-end visualization display of temperature and humidity.

1. Key Features

- Real-time data collection based on ESP32 + DHT11
- AES-256 encrypted transmission to ensure data security
- Flask backend receives decrypted data and SQLite stores it
- React front-end visualization, including line charts and threshold search
- Data analysis and automatic processing (trend mining)
- User system authentication, high security

2. Design Goals

- Build a stable and reliable real-time environmental data monitoring system
- Ensure the security and scientific of data transmission and storage
- Provide efficient data analysis and front-end and user interaction components

System Design

1. System Structure and Module Division

- **Data acquisition module (ESP32 + DHT11):** Responsible for reading temperature and humidity data from the environment in real time. ESP32 uses the GPIO4 interface to read the data from DHT11, collects data every 10 seconds, encrypts it using the AES-256 algorithm, encodes it in Base64, and sends it to the server via HTTP POST.
- **Server module (Flask + SQLite):** Flask receives encrypted data from ESP32, decrypts and parses JSON, and writes valid data into the SQLite database. It also provides several RESTful API interfaces for the front-end to call, including obtaining real-time data, aggregated data, temperature search, etc.
- **Front-end display module (React + Chart.js):** By calling the API provided by the server to pull data in real time, the temperature and humidity data are displayed in the form of cards and line charts on the web page, and users are supported to set temperature thresholds for search and filtering. The UI design is responsive and adaptable to various device sizes.

2. Circuit Connections

The connection between ESP32 and DHT11 sensor is as follows:

ESP32	DHT11
3.3V	+
GND	-
GPIO4	out

- ESP32 reads the Data pin signal of DHT11 through GPIO4, and VCC and GND are connected to 3.3V and ground respectively.
- ESP32 connects to the local network through Wi-Fi and communicates with the server.

3. Data Flow

- **Data collection stage:** ESP32 obtains temperature and humidity data from DHT11.
- **Encryption and transmission stage:** ESP32 encrypts data using AES-256, and uploads it to the server via HTTP POST after encoding it with Base64.
- **Server processing stage:** After receiving the data, the Flask backend decrypts and parses the JSON, and then stores it in the SQLite database.
- **Data service stage:** The front end obtains real-time or historical data from the database by calling the RESTful API interface.
- **Front-end display stage:** The React page obtains data regularly and updates the chart, while supporting interactive operations such as user search and filtering.

Implementation

1. Key Steps

a. **Integration of ESP32 and DHT11:** Configure the GPIO interface and use the Arduino DHT library to read temperature and humidity.
b. **Encryption and Encoding:** JSON data is encrypted using AES-256 and Base64-encoded.
c. **HTTP communication interacts with the server:** Sends a POST request to the Flask backend every 10 seconds.
d. **Server decryption and storage:** Flask uses the Crypto library to decrypt data, parse JSON, and write to the SQLite database using SQLAlchemy.
e. **Front-end display and data interaction:** React periodically requests back-end data, Chart.js implements dynamic trend charts, and supports threshold search functions.

2. Key Code Snippets

a. **ESP32: Data collection and encrypted upload**

```
// AES encryption key (must match server)
const uint8_t aes_key[32] = {
    0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38,
    0x39, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36,
    0x37, 0x38, 0x39, 0x30, 0x31, 0x32, 0x33, 0x34,
    0x35, 0x36, 0x37, 0x38, 0x39, 0x30, 0x31, 0x32
};
void sendData(float temperature, float humidity) {
    // Create JSON data
    String jsonData = "{\"temperature\":" + String(temperature) +
                      ",\"humidity\":" + String(humidity) +
```

```
                ",\"timestamp\":" + String(millis()) + "}";
    // AES encryption
    AES256 aes;
    aes.setKey(aes_key, 32);
    String encrypted = aes.encrypt(jsonData);
    // Base64 encoding
    String base64Data = base64::encode(encrypted);
    // Send to server
    HTTPClient http;
    http.begin(serverName + "/api/post-data");
    http.addHeader("Content-Type", "text/plain");
    int httpResponseCode = http.POST(base64Data);
}
```

b. Flask: receiving, decrypting, storing

```
# AES decryption function
def decrypt_aes(cipher_text):
    try:
        cipher_text = base64.b64decode(cipher_text)
        cipher = AES.new(aes_key, AES.MODE_ECB)
        decrypted = cipher.decrypt(cipher_text).decode()
        return decrypted.strip().replace("\x00", "")
    except Exception as e:
        logger.error(f"AES decryption failed: {str(e)}")
        return None
# Data reception route
@app.route('/api/post-data', methods=['POST'])
def receive_data():
    try:
        # Get encrypted data
        raw_data = request.get_data().decode('utf-8')
        # Decrypt data
        decrypted_data = decrypt_aes(raw_data)
        # Parse JSON
        data = json.loads(decrypted_data)
        # Save to database
        save_data(data)
        return jsonify({"message": "Data received successfully"})
    except Exception as e:
        logger.error(f"Data processing failed: {str(e)}")
        return jsonify({"error": "Data processing failed"}), 500
# Data storage function
def save_data(data):
    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()
    cursor.execute("""
        INSERT INTO sensor_data (temperature, humidity, timestamp)
```

```
        VALUES (?, ?, ?)
    """, (data["temperature"], data["humidity"], data["timestamp"]))
    conn.commit()
    conn.close()
```

c. React: real-time data fetching and display

```
const Dashboard = () => {
const [sensorData, setSensorData] = useState([]);
// Real-time data fetching
useEffect(() => {
    const fetchData = async () => {
        try {
            const response = await fetch('/api/get-data');
            const data = await response.json();
            setSensorData(data);
        } catch (error) {
            console.error('Data fetching failed:', error);
        }
    };
    // Initial fetch
    fetchData();
    // Periodic updates
    const interval = setInterval(fetchData, 5000);
    return () => clearInterval(interval);
}, []);
// Chart configuration
const chartData = {
    labels: sensorData.map(d => new Date(d.timestamp * 1000).toLocaleTimeString()),
    datasets: [
        {
            label: 'Temperature',
            data: sensorData.map(d => d.temperature),
            borderColor: 'rgb(255, 99, 132)',
            tension: 0.1
        },
        {
            label: 'Humidity',
            data: sensorData.map(d => d.humidity),
            borderColor: 'rgb(53, 162, 235)',
            tension: 0.1
        }
    ]
};
return (
    <div className="dashboard">
        <h2>Real-time Temperature & Humidity Monitoring</h2>
        <div className="chart-container">
```

```
            <Line data={chartData} />
          </div>
          <div className="current-values">
            <div className="temperature">
              Current Temperature: {sensorData[0]?.temperature}°C
            </div>
            <div className="humidity">
              Current Humidity: {sensorData[0]?.humidity}%
            </div>
          </div>
        </div>
    );
};
```

Testing & Results

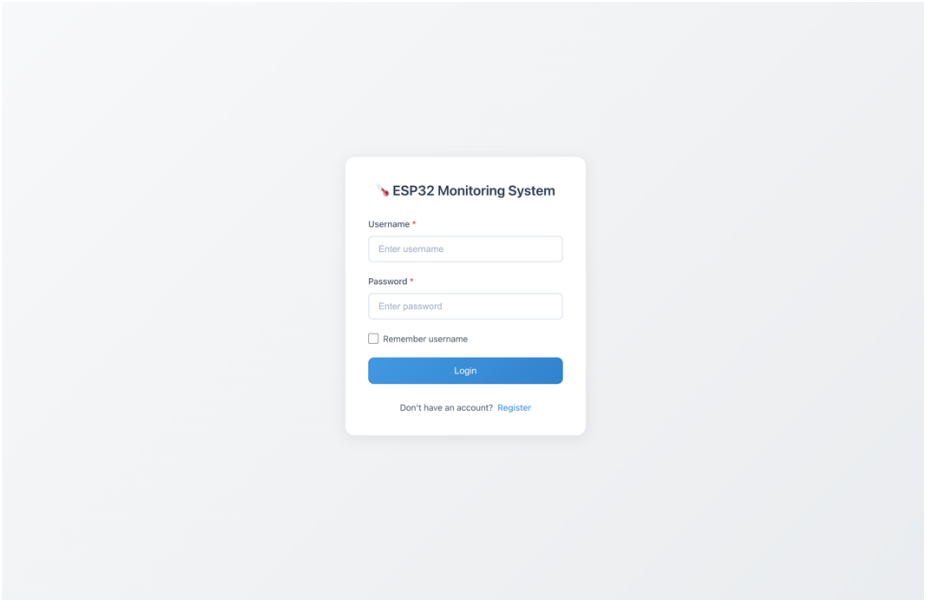
1. Testing Steps

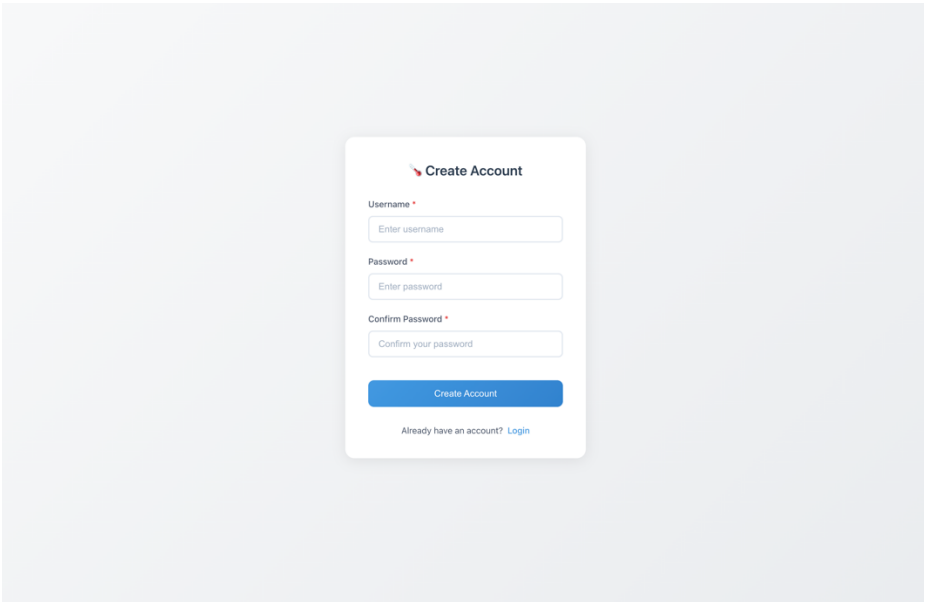
a. **Hardware wiring test:** Make sure the DHT11 and ESP32 are wired correctly, and use the serial monitor to observe whether the sensor is reading normally.
b. **Wi-Fi connection test:** Verify whether ESP32 can connect to the specified Wi-Fi network and successfully obtain an IP.
c. **Encryption and POST data test:** Use the serial port output to observe whether the encrypted Base64 data is successfully sent to the Flask backend.
d. **Backend data reception and decryption test:** Check whether the Flask console successfully receives the data, decrypts it successfully, and writes it to the database.
e. **Database write test:** Use the SQLite tool to view the database content and confirm that the data is written accurately.
f. **Front-end display test:** Run the React application to check whether the front-end successfully pulls data and renders charts and real-time data normally.
g. **Abnormal situation testing:** disconnect Wi-Fi, input illegal data, etc., to test system stability and error handling mechanism.

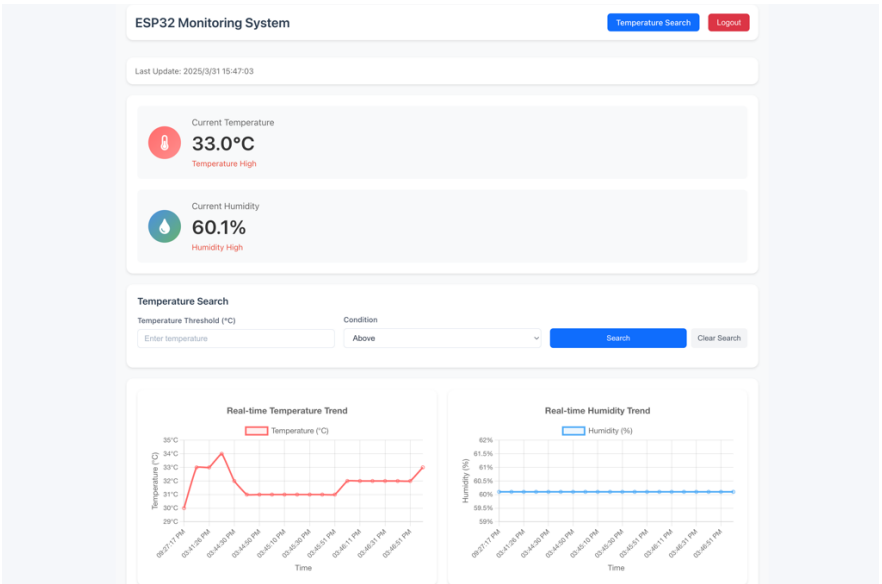
2. Testing Results

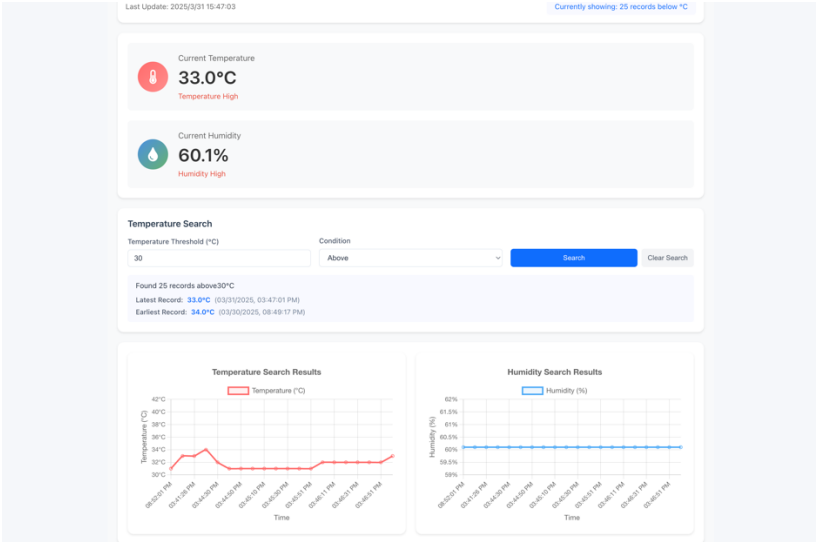
Test Item	Expected Result	Actual Result
Read Temperature & Humidity	Data is correctly retrieved	✅
Threshold Filter Function	Filters historical data on frontend based on threshold	✅
Encrypted Upload & Decrypt	ESP32 uploads encrypted data, Flask decrypts successfully	✅
Database Storage	Data is saved in SQLite and can be queried	✅
Real-Time Chart Display	React updates data every 10s and refreshes chart	✅

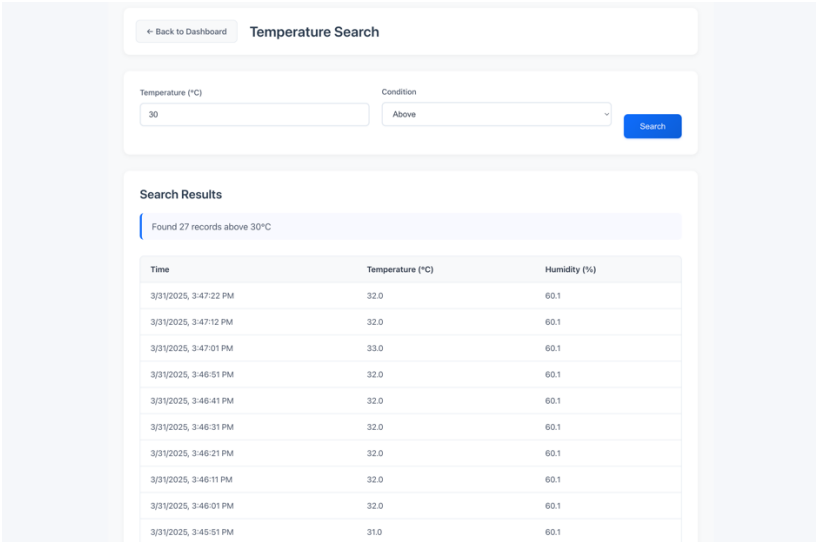
3. Screenshots


ESP32 Monitoring System
Username *
Enter username
Password *
Enter password
Remember username
Login
Don't have an account? Register


Create Account
Username *
Enter username
Password *
Enter password
Confirm Password *
Confirm your password
Create Account
Already have an account? Login


ESP32 Monitoring System
Temperature Search
Logout
Last Update: 2025/3/31 15:47:03
Current Temperature
33.0°C
Temperature High
Current Humidity
60.1%
Humidity High
Temperature Search
Temperature Threshold (°C)
Enter temperature
Condition
Above
Search
Clear Search
Real-time Temperature Trend
Temperature (°C)
Time
Real-time Humidity Trend
Humidity (%)
Time






```
19:31:44.089 -> Reading DHT sensor data...
19:31:44.121 -> Time synchronized, preparing JSON...
19:31:44.121 -> Unencrypted JSON Data: {"temperature":30,"humidity":60.1,"timestamp":1743467504}
19:31:44.121 -> Sending Encrypted JSON Data: Ppu5OUSg9nq9tHNO3nW+03AYRVnW+R02qXHR4jQXy/uVBN9IX+kylTNHeyUV9vM1p7uAGI0o3eZp5AtgeGq+3w==
19:31:44.219 -> Server Response Code: 200
19:31:44.252 -> Server Response Body: {
19:31:44.252 ->   "message": "\u6570\u636e\u63a5\u6536\u6210\u529f"
```

```
2025-03-31 19:31:54,292 - __main__ - DEBUG - 接收到的原始数据: Ppu5OUSg9nq9tHNO3
nW+03AYRVnW+R02qXHR4jQXy/uVBN9IX+kylTNHeyUV9vM1/VZwIA0Z8hVgdG6WypBsKg==
2025-03-31 19:31:54,292 - __main__ - DEBUG - 解密后的数据: {"temperature":30,"hu
midity":60.1,"timestamp":1743467514}
2025-03-31 19:31:54,292 - __main__ - DEBUG - 解析后的数据: {'temperature': 30, '
humidity': 60.1, 'timestamp': 1743467514}
2025-03-31 19:31:54,295 - __main__ - INFO - 数据保存成功: {'temperature': 30, 'h
umidity': 60.1, 'timestamp': 1743467514}
2025-03-31 19:31:54,295 - werkzeug - INFO - 172.31.99.27 - - [31/Mar/2025 19:31:
54] "POST /api/post-data HTTP/1.1" 200 -
2025-03-31 19:31:54,580 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:31
:54] "GET /api/get-data HTTP/1.1" 200 -
2025-03-31 19:31:54,614 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:31
:54] "GET /api/get-data HTTP/1.1" 200 -
2025-03-31 19:31:56,582 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:31
:56] "GET /api/get-data HTTP/1.1" 200 -
2025-03-31 19:31:58,578 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:31
:58] "GET /api/get-data HTTP/1.1" 200 -
2025-03-31 19:32:00,581 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:32
:00] "GET /api/get-data HTTP/1.1" 200 -
2025-03-31 19:32:02,582 - werkzeug - INFO - 172.31.99.212 - - [31/Mar/2025 19:32
:02] "GET /api/get-data HTTP/1.1" 200 -
```

	id	temperature	humidity	timestamp	created_at
1	1	30	60.1	1743381185	2025-03-31 00:33:05
2	2	30	60.1	1743381195	2025-03-31 00:33:15
3	3	30	60.1	1743381205	2025-03-31 00:33:25
4	4	29	60.1	1743381215	2025-03-31 00:33:36
5	5	30	60.1	1743381226	2025-03-31 00:33:46
6	6	30	60.1	1743381236	2025-03-31 00:33:56
7	7	30	60.1	1743381246	2025-03-31 00:34:06
8	8	30	60.1	1743381256	2025-03-31 00:34:16
9	9	29	60.1	1743381266	2025-03-31 00:34:26

	id	username	password_hash	created_at
1	1	admin	scrypt:32768:8:1$fwnaGDPGPCKMOw1E$87ae6d1…	2025-03-31 02:16:07.642808
2	2	user	scrypt:32768:8:1$aHe17pDg7b0zvkd6$d62563e…	2025-03-31 02:26:12.222166
3	3	hui	scrypt:32768:8:1$tF1mnKebDKb2LYW6$4b7cd19…	2025-03-31 02:34:03.148189
4	4	jiarui	scrypt:32768:8:1$156cjh1anct1b0TQ$8f9437a…	2025-03-31 20:32:37.203573
5	5	huijin	scrypt:32768:8:1$AdPSpMEbkNSKyqJn$b827e10…	2025-03-31 20:44:51.230788

Challenges Faced

1. Because AES encryption requires the input length to be an integer multiple of 16 bytes, incorrect length often occurs when JSON strings are used as plain text in the early stages, resulting in decryption failure.
2. When decrypting on the server side using Python, padding removal was initially not handled properly, causing the JSON string to not parse correctly.
3. After the system has been running for a long time, the SQLite database may store a large amount of historical data, affecting query efficiency.

Conclusion

This project successfully demonstrates the integration of an ESP32 microcontroller with a DHT11 sensor and a full-stack web architecture to enable real-time environmental monitoring, secure data transmission, and intuitive web-based visualization. The system reliably captures temperature and humidity data, encrypts and transmits them over HTTP to a Python Flask backend, and stores them in a lightweight SQLite database. A responsive React frontend presents real-time updates and historical insights using dynamic charts and user-defined filters.

Throughout the development process, thoughtful design choices were made to ensure the system's security, scalability, and stability. The use of AES-256 encryption, Base64 encoding, and well-defined RESTful APIs allowed seamless and secure communication between the hardware and software components. On the client side, features such as threshold filtering and chart auto-refresh offered users an intuitive and informative experience.

Numerous challenges, including encryption compatibility, stable Wi-Fi communication, and performance tuning of frontend updates, were encountered and

resolved. These experiences significantly improved our understanding of secure IoT communication, asynchronous frontend handling, and backend data persistence. The success of this project illustrates the viability of ESP32 as a low-cost, energy-efficient platform for wireless sensing applications. The modular architecture and clean integration between hardware and software enable this solution to be extended to other environmental metrics or industrial monitoring needs. The practical skills and design patterns learned here lay a strong foundation for future development of scalable and secure IoT systems.

Reference

[1] https://lastminuteengineers.com/esp32-pinout-reference/