SMART HOME MONITORING SYSTEM

GROUP 12

GROUP MEMBERS

Student id	Name	Studies
202206885	Mikail Ipek	SW
202003681	Emre Erbas	SW
202204580	Bassam Salou	SW

DATE

22. December 2024

DISCLAIMER

CHATGPT WAS USED IN THE PROJECT, FOR CODE AND REPORT

Table of Content

Work Area

All work is divided equally between group members and is done collaboratively, but the focus areas can be seen below:

Work Areas	Mikail Ipek	Emre Erbas	Bassam Salou
Idea	All	All	All
Architecture	All	All	All
Embedded Software and Hardware	Secondary	Primary	Secondary
MQTT Communication	Primary	Secondary	Primary
Database - Firebase Firestore	Secondary	Primary	Secondary
Frontend	Secondary	Secondary	Primary
Server - Azure VM	Primary	Secondary	Secondary
Results / Discussion / Conclusion	All	All	All

Table 1 - Work Areas

Introduction

This project focuses on creating an IoT-based monitoring system for tracking temperature, humidity, and motion. It uses MQTT for communication between devices, Firebase for storing data, and React.js for displaying the information on a web interface. The goal was to build a system that is real-time, easy to use, and scalable by connecting sensors, cloud services, and a web application.

The idea came from the need for remote environmental monitoring, which is useful in smart homes, industrial setups, and environmental research. To meet these demands, the project was designed with a modular structure, making it easy to expand with more sensors and devices in the future.

The system includes two ESP32 microcontrollers equipped with DHT11 sensors for temperature and humidity and PIR motion sensors for detecting movement. MQTT was chosen for its lightweight and reliable messaging capabilities. Firebase was selected because of its real-time database features, and React.js was used for building a dynamic and interactive web interface.

This report explains how the system was developed, covering its architecture, technology choices, and how it was implemented. It includes examples of important code and explanations of how different parts of the system work. Challenges encountered during the development process are also discussed, along with how they were solved.

Architecture

Overview

The architecture of the IoT-based monitoring system is designed to ensure efficient communication, data processing, and real-time monitoring. This section provides an outline of the system's high-level structure, and the updates made during the project phases. Component functionalities and implementation details are covered in the **Design and Implementation** section.

High-Level System Structure

The system's architecture consists of interconnected hardware and software components that facilitate seamless monitoring of environmental parameters. The major components are:

- **ESP32 Microcontrollers**: Responsible for data collection from temperature, humidity, and motion sensors.
- **MQTT Mosquitto Broker**: Manages data communication between ESP32 devices and the backend.
- **Backend (Python Script)**: Processes and enriches incoming data before storing it in the database.
- **Azure VM**: Responsible for housing the Broker and Backend with Microsoft's cloud service enriching IoT capabilities and access.
- **Firestore Database**: Stores sensor data and provides real-time updates to the frontend.
- **Frontend (React Application)**: Visualizes data through dashboards and graphs.

Architectural Diagrams

High-Level Architecture Diagram: Illustrates the overall data flow between hardware devices, cloud services, and the user interface.

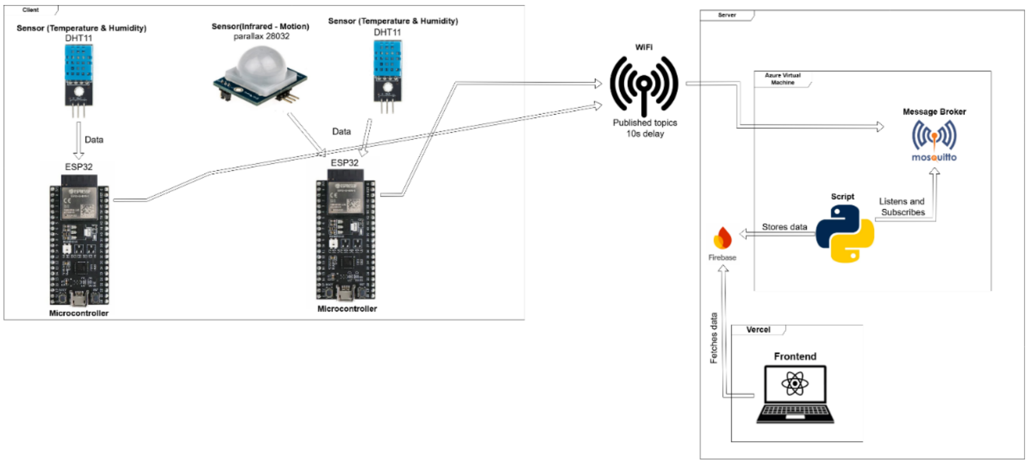


Figure 1 - High-level architecture diagram of the system

Detailed Architecture Diagram: Provides a C4-inspired view of module interactions, focusing on individual components and their interconnections. This diagram is also available in the zipped final delivery in case this is too pixilated.

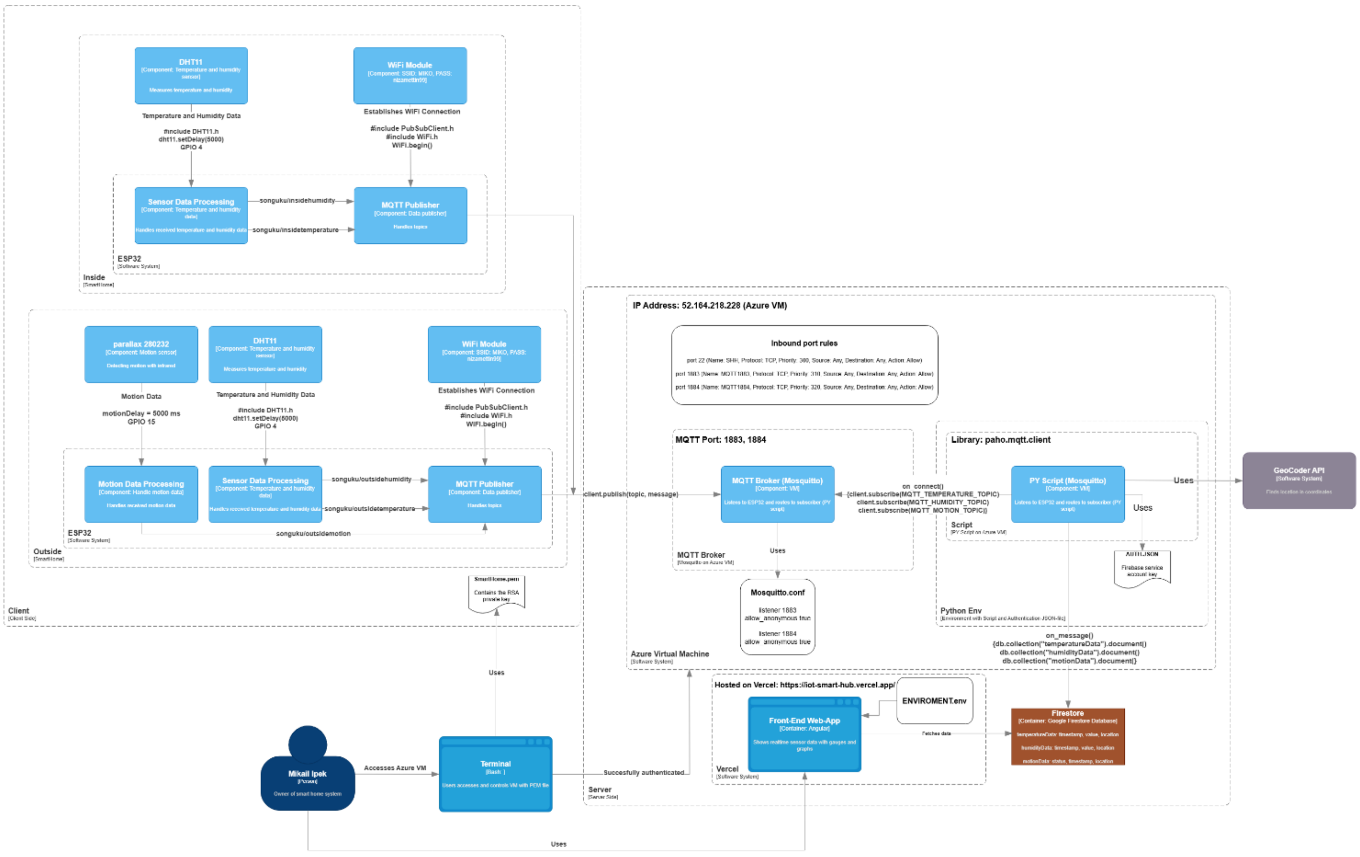

DHT11
[Component: Temperature and humidity sensor]
Measures temperature and humidity
WiFi Module
[Component: SSID: MIKO, PASS: nizamettin99]
Temperature and Humidity Data
#include DHT11.h
dht11.setDelay(5000)
GPIO 4
Establishes WiFi Connection
#include PubSubClient.h
#include WiFi.h
WiFi.begin()
Sensor Data Processing
[Component: Temperature and humidity data]
Handles received temperature and humidity data
songuku/insidehumidity
songuku/insidetemperature
MQTT Publisher
[Component: Data publisher]
Handles topics
ESP32
[Software System]
Inside
[SmartHome]
parallax 280232
[Component: Motion sensor]
Detecting motion with infrared
DHT11
[Component: Temperature and humidity sensor]
Measures temperature and humidity
WiFi Module
[Component: SSID: MIKO, PASS: nizamettin99]
Motion Data
motionDelay = 5000 ms
GPIO 15
Temperature and Humidity Data
#include DHT11.h
dht11.setDelay(5000)
GPIO 4
Establishes WiFi Connection
#include PubSubClient.h
#include WiFi.h
WiFi.begin()
Motion Data Processing
[Component: Handle motion data]
Handles received motion data
Sensor Data Processing
[Component: Temperature and humidity data]
Handles received temperature and humidity data
songuku/outsidehumidity
songuku/outsidetemperature
MQTT Publisher
[Component: Data publisher]
Handles topics
songuku/outsidemotion
ESP32
[Software System]
Outside
[SmartHome]
SmartHome.pem
Contains the RSA private key
Client
[Client Side]
Uses
client.publish(topic, message)
IP Address: 52.164.218.228 (Azure VM)
Inbound port rules
port 22 (Name: SHH, Protocol: TCP, Priority: 300, Source: Any, Destination: Any, Action: Allow)
port 1883 (Name: MQTT1883, Protocol: TCP, Priority: 310, Source: Any, Destination: Any, Action: Allow)
port 1884 (Name: MQTT1884, Protocol: TCP, Priority: 320, Source: Any, Destination: Any, Action: Allow)
MQTT Port: 1883, 1884
MQTT Broker (Mosquitto)
[Component: VM]
Listens to ESP32 and routes to subscriber (PY script)
on_connect()
{client.subscribe(MQTT_TEMPERATURE_TOPIC)
client.subscribe(MQTT_HUMIDITY_TOPIC)
client.subscribe(MQTT_MOTION_TOPIC)}
Library: paho.mqtt.client
PY Script (Mosquitto)
[Component: VM]
Listens to ESP32 and routes to subscriber (PY script)
Uses
GeoCoder API
[Software System]
Finds location in coordinates
Uses
Script
[PY Script on Azure VM]
AUTH.JSON
Firebase service account key
MQTT Broker
[Mosquitto on Azure VM]
Uses
Mosquitto.conf
listener 1883
allow_anonymous true
listener 1884
allow_anonymous true
Python Env
[Environment with Script and Authentication JSON-file]
on_message()
{db.collection("temperatureData").document()
db.collection("humidityData").document()
db.collection("motionData").document()}
Azure Virtual Machine
[Software System]
Hosted on Vercel: https://iot-smart-hub.vercel.app/
ENVIROMENT.env
Front-End Web-App
[Container: Angular]
Shows realtime sensor data with gauges and graphs
Fetches data
Firestore
[Container: Google Firestore Database]
temperatureData: timestamp, value, location
humidityData: timestamp, value, location
motionData: status, timestamp, location
Vercel
[Software System]
Mikail Ipek
[Person]
Owner of smart home system
Accesses Azure VM
Terminal
[Bash]
Users accesses and controls VM with PEM file
Succesfully authenticated
Server
[Server Side]
Uses

Figure 2 - Detailed architecture C4-inspired diagram

Architectural Updates from Previous Phases

Throughout the project, several updates were made to optimize performance and scalability:

- **ESP32 Modularization**: Added a second ESP32 microcontroller to separate indoor and outdoor monitoring, enhancing modularity and enhancing the scalability of the project.
- **Frontend Framework Update**: Switched from Angular to React for better performance and simplified Firebase integration.
- **MQTT Port Segmentation**: Configured distinct ports (1883 for outdoor data and 1884 for indoor data) to resolve communication conflicts.
- **Frontend Hosting Transition**: Migrated from local hosting to Vercel for improved accessibility and deployment simplicity.

Background and Technical Analysis

The development of this IoT monitoring system involved selecting specific technologies and designing an architecture that adheres to IoT principles. Each choice was informed by the system's functional requirements, performance expectations, and scalability needs.

Microcontroller: ESP32

The ESP32 microcontroller was chosen for its built-in Wi-Fi capability, low power consumption, and cost-effectiveness (not relevant), making it ideal for IoT applications. It provides sufficient computational power for data processing without requiring additional hardware. Alternatives like the Arduino Uno lack built-in Wi-Fi, and the Raspberry Pi offers more power than necessary for this project, making the ESP32 the most efficient choice.

Figure 3 - ESP32 microcontroller

It was also available through the university's **Embedded Stock ELab**[1], simplifying the setup process. While the project requirements influenced our decision, the ESP32's compatibility with IoT tasks made it the ideal choice.

[1] https://stock.ece.au.dk/Components

Sensors: DHT11 and PIR Motion Sensor

DHT11 was selected for temperature and humidity monitoring due to its affordability and ease of use. While the DHT22 offers higher precision and was thought of, the DHT11 sufficiently meets the project's requirements and was readily available.

Figure 4 - DHT-11 temperature and humidity sensor

Since the DHT22 was not available in the university's **Embedded Stock ELab**, the DHT11 became the practical and next best choice.

Table for pros and cons of the DHT11:

DHT11	
Pros	**Cons**
• Cheap • Easy to integrate and user-friendly • Perfect for indoor use, capable of measuring temperatures 0-50 degrees Celsius[2] • Long lasting	• Not as precise and fast measurements as DHT22 • Cannot measure freezing temperatures and extreme humidity values.

Table 2 - Pros and cons of DHT11

PIR Motion Sensor was chosen for its reliability and low power consumption in motion detection. Compared to ultrasonic sensors, it-is more cost-efficient and simpler to integrate, aligning with the project's focus on functionality over precision. In general, adding more distinct sensors to the system, is fun and challenging.

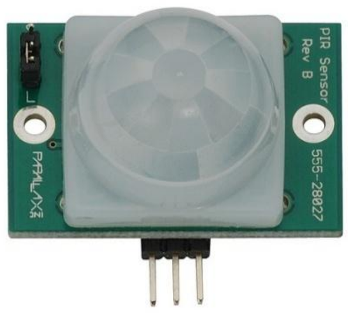


Figure 5 - Parallax PIR motion sensor

[2] https://ardustore.dk/produkt/dht-11-temperatur-fugtigheds-module

Messaging Protocol: MQTT

The MQTT protocol was selected for its lightweight design and suitability for resource-constrained IoT devices. It enables real-time communication with minimal latency, ensuring efficient data transmission.

Scalability and Real-World Applications:

- **Scalability**: MQTT's publish-subscribe model allows seamless integration of additional devices and sensors. For example, the system can scale to monitor multiple rooms or integrate new environmental parameters easily.
- **Real-World Use**: MQTT is widely adopted in industries like home automation (e.g., controlling smart lights) and logistics (e.g., tracking shipments with IoT devices). Its lightweight nature makes it a preferred choice for battery-powered microcontrollers, such as the ESP32.

Alignment with Project Goals: MQTT was also specified as part of the project requirements, making it a practical fit for the system.

Broker: Mosquitto

The **Mosquitto** broker was chosen to manage the MQTT communication. It is lightweight, open-source, and widely adopted in IoT projects, making it an excellent choice for the scale of this system. It also as a hugely available support on the internet on various topics.
Comparison to Alternatives:

Mosquitto	
Pros	**Cons (Pros for HiveMQ)**
• Lightweight and resource-efficient • Open source with no licensing costs. • Simple configuration, ideal for academic projects.	• Non-existing monitoring and analytics tools, not necessary for this project. • Not great for enterprise deployments due to missing advanced features, again not necessary.

Table 3 - Pros and cons of mosquitto

Database: Firebase Firestore

Firestore's real-time synchronization capabilities and schema-less design was game changing in enabling dynamic and flexible data management for this project. Unlike SQL databases, Firestore's NoSQL approach is ideal for handling IoT data streams and require minimal setup.

Firestore Advantages	**Compared to Alternatives**
Real-time data updates	SQL requires rigid schemas
Seamless integration with React	NoSQL offers better flexibility
Easy setup for rapid prototyping	Better for dynamic IoT project requirements

Frontend Framework: React

React.js was chosen for its modularity and simplicity, allowing rapid development of dynamic user interfaces. Compared to Angular, React offers a less steep learning curve and faster implementation for small to medium-sized projects.
Comparison to Alternatives:

Angular: More suited for complex, large-scale applications but has a steeper learning curve.

Go: While fast and efficient, Go lacks the frontend-specific libraries and frameworks available in React.

Azure VM:

Azure VM hosted the MQTT broker and backend services, ensuring reliability and scalability. The use of university-provided free credits further reduced costs.

Comparison to Local Hosting:

While local hosting offers lower latency, it restricts the system to a confined local network and lacks the flexibility of a cloud-based solution and therefore a worse option for IoT based systems that require easy accessibility.

Backend: Python

Python was selected for its simplicity and extensive library support, including paho-mqtt for MQTT integration and Firebase Admin SDK for database interaction. While alternatives like Node.js are a viable option, Python's readability and compatibility with the project's academic setting made it the preferred choice.

Justification as an IoT Project

The project exemplifies key IoT principles through interconnected devices, real-time data exchange, and cloud integration:

- **Interconnected Devices:**
  The ESP32 microcontrollers and sensors form a network that collects and transmits environmental data.
- **Real-Time Data Exchange:**
  MQTT enables efficient and instantaneous communication between devices, while Firebase ensures real-time data updates for the frontend.
- **Cloud Integration:**
  Hosting the MQTT broker on Azure and storing data in Firebase aligns with modern IoT architectures, emphasizing scalability and remote access.
- **User Interaction:**
  The React-based frontend allows users to monitor environmental data and receive updates seamlessly, completing the end-to-end IoT workflow.

By combining these elements, the project demonstrates a robust IoT system that integrates hardware, software, and cloud services efficiently.

Design and Implementation

The design and implementation of the Smart Monitoring System integrate embedded systems, communication protocols, cloud platforms, and a user-friendly interface to create a robust IoT solution. This section explains the design decisions, implementation steps, and justifications for each major component.

Embedded Software and Hardware (ESP32)

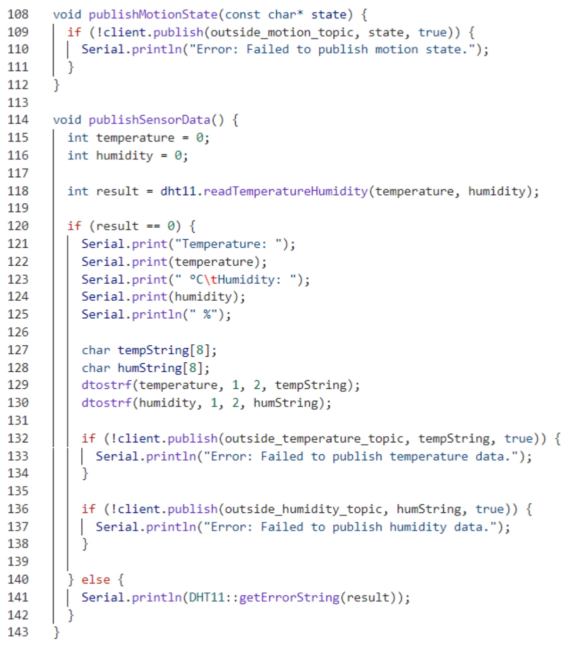
The embedded software and hardware components of the Smart Monitoring System were designed to collect, process, and transmit environmental data in real-time using an ESP32 microcontroller. This microcontroller was chosen for its built-in Wi-Fi capabilities, low power consumption, and cost-effectiveness, making it ideal for IoT applications. The embedded software was developed using C++ in the Arduino IDE environment.

The hardware consisted of an ESP32 microcontroller, a DHT11 sensor for temperature and humidity measurements, and a PIR motion sensor for motion detection. These sensors were interfaced with the ESP32 to ensure accurate data collection

Implementation Process

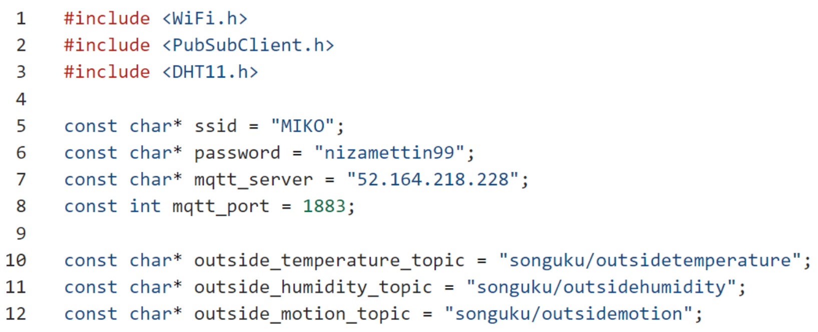
The implementation began with setting up the hardware components. The DHT11 sensor was connected to the ESP32 to provide periodic temperature and humidity readings, while the PIR sensor detected motion within its field of view. The ESP32 acted as the central processing unit, collecting data from these sensors and transmitting it to the MQTT broker as topics.

The software development involved configuring the ESP32 to establish a stable Wi-Fi connection using the **WiFi.h** library. The microcontroller was programmed to connect to the MQTT broker hosted on an Azure Virtual Machine via the **PubSubClient** library. This broker facilitated the communication between the ESP32 and the backend system.



```
void publishMotionState(const char* state) {
  if (!client.publish(outside_motion_topic, state, true)) {
    Serial.println("Error: Failed to publish motion state.");
  }
}

void publishSensorData() {
  int temperature = 0;
  int humidity = 0;

  int result = dht11.readTemperatureHumidity(temperature, humidity);

  if (result == 0) {
    Serial.print("Temperature: ");
    Serial.print(temperature);
    Serial.print(" °C\tHumidity: ");
    Serial.print(humidity);
    Serial.println(" %");

    char tempString[8];
    char humString[8];
    dtostrf(temperature, 1, 2, tempString);
    dtostrf(humidity, 1, 2, humString);

    if (!client.publish(outside_temperature_topic, tempString, true)) {
      Serial.println("Error: Failed to publish temperature data.");
    }

    if (!client.publish(outside_humidity_topic, humString, true)) {
      Serial.println("Error: Failed to publish humidity data.");
    }

  } else {
    Serial.println(DHT11::getErrorString(result));
  }
}
```

Figure 7 - esp32 (outside) publish logic snippet



```
#include <WiFi.h>
#include <PubSubClient.h>
#include <DHT11.h>

const char* ssid = "MIKO";
const char* password = "nizamettin99";
const char* mqtt_server = "52.164.218.228";
const int mqtt_port = 1883;

const char* outside_temperature_topic = "songuku/outsidetemperature";
const char* outside_humidity_topic = "songuku/outsidehumidity";
const char* outside_motion_topic = "songuku/outsidemotion";
```

Figure 6 - esp32 includes, data setup and topics

Wi-Fi Connectivity: The ESP32 was configured to connect to a specified mobile hotspot network using hardcoded credentials for testing purposes (figure 6). The connection status was continuously monitored, and the device attempted to reconnect automatically if the connection was lost.

MQTT Communication: The ESP32 was programmed to publish data to specific MQTT topics for temperature, humidity, and motion (figure 7). Each sensor's data was encapsulated in JSON format for structured transmission. For example, temperature and humidity data were sent to topics **songuku/outsidetemperature** and **songuku/outsidehumidity**, respectively.

Sensor Integration:

- The DHT11 sensor interfaced with the ESP32 using its digital pin to provide temperature and humidity readings. The dht11.readTemperatureHumidity function was employed to read and process sensor data.
- The PIR motion sensor monitored movement and triggered state changes. Motion detection was sent as events ("Motion detected" or "Motion ended") to the songuku/outsidemotion topic.

Data Publishing: The microcontroller maintained a timer to ensure periodic data updates. This interval was configured to 5 seconds, balancing real-time updates with network efficiency.

The hardware was built on a breadboard with necessary connections.

- **Red:** VCC 5V power supply from ESP32 to PIR sensor and DHT11
- **Blue:** Data line between PIR sensor and ESP32
- **Black:** Common ground between all components to ensure stability and eliminate noise.
- **Green:** Data line between DHT11 and ESP32
- **White:** Micro USB 5V power supply for ESP32

The system functions without a pull-up resistor. However, a 5kohm resistor is recommended by the makers of the DHT11 sensor as mentioned in their datasheet.[3]

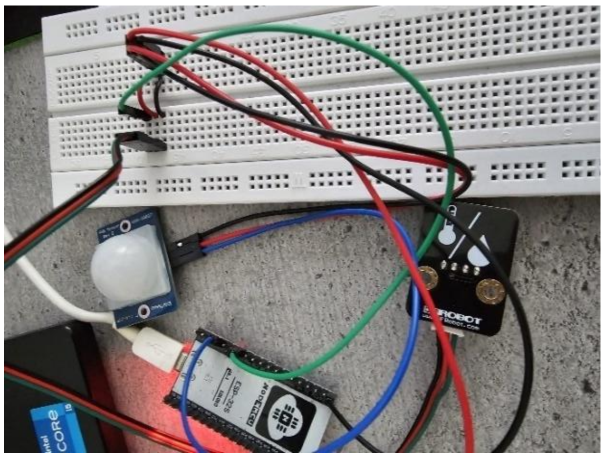
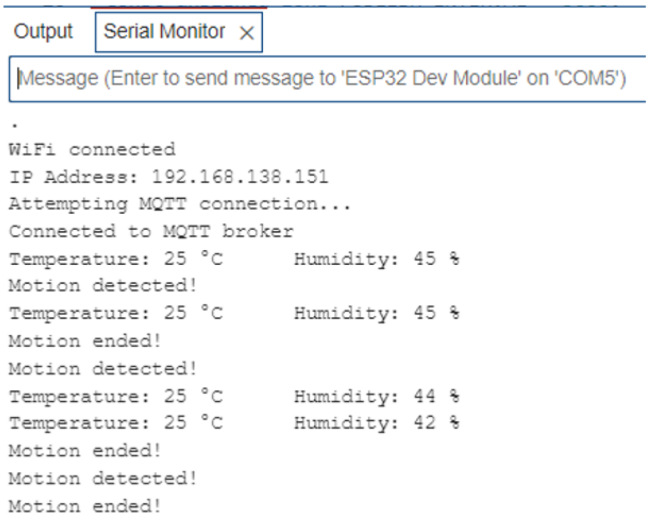


Figure 8 - Setup of esp32 (outside)

[3] https://ardustore.dk/produkt/dht-11-temperatur-fugtigheds-module

Before implementing anything else the sensors along with the MQTT connection was tested thoroughly and successfully worked:


Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM5')

```
.
WiFi connected
IP Address: 192.168.138.151
Attempting MQTT connection...
Connected to MQTT broker
Temperature: 25 °C      Humidity: 45 %
Motion detected!
Temperature: 25 °C      Humidity: 45 %
Motion ended!
Motion detected!
Temperature: 25 °C      Humidity: 44 %
Temperature: 25 °C      Humidity: 42 %
Motion ended!
Motion detected!
Motion ended!
```

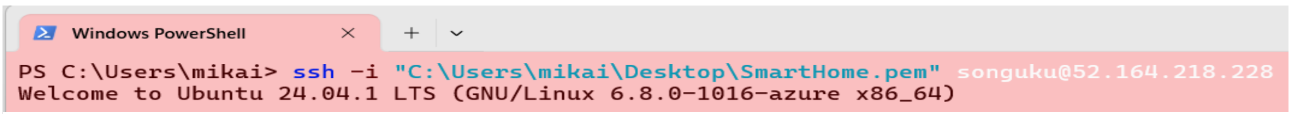
Figure 9 - Arduino IDE output

Cloud Infrastructure (Azure VM)

The cloud infrastructure for the Smart Monitoring System was built using a Microsoft Azure Virtual Machine (VM). This VM acted as the central hub for hosting the Mosquitto MQTT broker and the Python backend script. The decision to use Azure VM was driven by the need for scalability, remote accessibility, and the availability of free university credits, which made the deployment cost-effective. By taking advantage of this, the system achieved seamless integration between IoT devices and the cloud.

Azure VM Setup and Configuration

The virtual machine was set up using the Ubuntu 24.04 LTS operating system, a lightweight and widely supported Linux distribution. SSH key-based authentication was used to ensure secure and access to the VM. A private key file, SmartHome.pem, was generated and stored securely on the local machine. The VM was accessed using the following command:



```
PS C:\Users\mikai> ssh -i "C:\Users\mikai\Desktop\SmartHome.pem" songuku@52.164.218.228
Welcome to Ubuntu 24.04.1 LTS (GNU/Linux 6.8.0-1016-azure x86_64)
```

Figure 10 - How to access the Azure VM command

This method ensured that only authorized users could connect to the server.

MQTT Broker Deployment

The Mosquitto MQTT broker was chosen for its efficiency and ease of use in IoT environments. It was installed on the Azure VM to manage communication between the ESP32 microcontrollers and the backend. The setup process included:

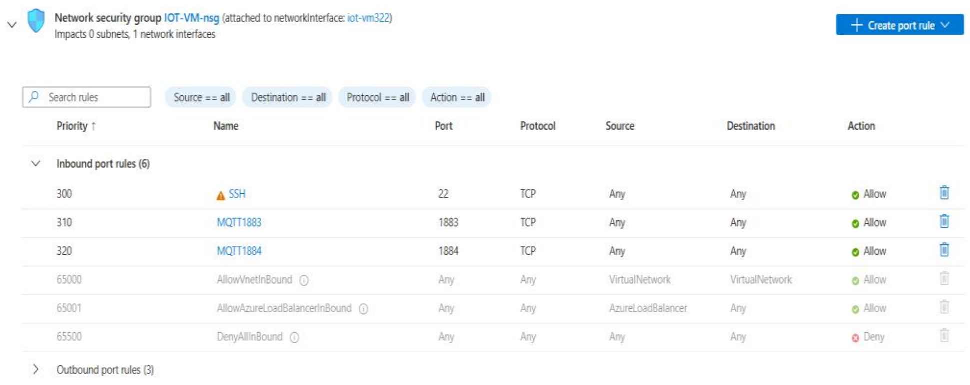
Installation: Mosquitto and its client tools were installed using the following command:

```
sudo apt install mosquitto mosquitto-clients
```

Configuration: The broker was configured to accept connections on two separate ports:

- Port 1883 for outdoor data
- Port 1884 for indoor data

These ports were defined in Azure's network security group settings to ensure distinct communication channels for multiple devices. The rules for these ports were visible in the Azure portal as shown in the network security group diagram.



Network security group IOT-VM-nsg (attached to networkinterface: iot-vm322)
Impacts 0 subnets, 1 network interfaces

Priority ↑	Name	Port	Protocol	Source	Destination	Action
Inbound port rules (6)						
300	SSH	22	TCP	Any	Any	Allow
310	MQTT1883	1883	TCP	Any	Any	Allow
320	MQTT1884	1884	TCP	Any	Any	Allow
65000	AllowVnetInBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowAzureLoadBalancerInBound	Any	Any	AzureLoadBalancer	Any	Allow
65500	DenyAllInBound	Any	Any	Any	Any	Deny
Outbound port rules (3)						

Figure 11 - MQTT port rules

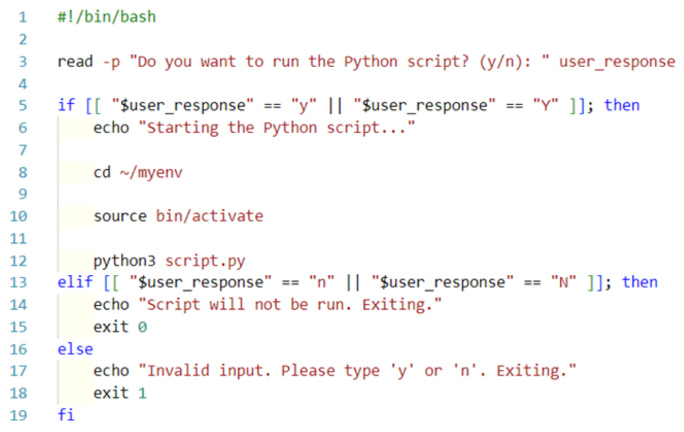
Startup Configuration: The Mosquitto service was enabled to start automatically on boot:

```
sudo systemctl enable mosquitto
```

This added a level of automation to the Azure VM. Further development details of the MQTT broker will be dwelled into in upcoming section **MQTT Communication.**

Backend Automation

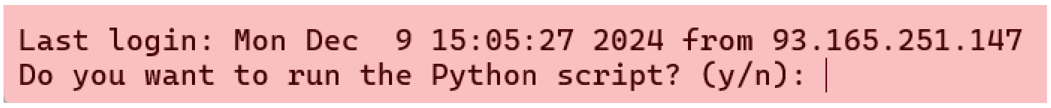
A Python script hosted on the VM processed sensor data received from the MQTT broker and stored it in Firebase Firestore. To streamline the deployment process, a shell script named **auto_run.sh** was created. This script automated the activation of the Python virtual environment and the execution of the backend script. Below in the if-statement is where the magic happens:



```bash
#!/bin/bash

read -p "Do you want to run the Python script? (y/n): " user_response

if [[ "$user_response" == "y" || "$user_response" == "Y" ]]; then
    echo "Starting the Python script..."

    cd ~/myenv

    source bin/activate

    python3 script.py
elif [[ "$user_response" == "n" || "$user_response" == "N" ]]; then
    echo "Script will not be run. Exiting."
    exit 0
else
    echo "Invalid input. Please type 'y' or 'n'. Exiting."
    exit 1
fi
```

Figure 12 - Automation script

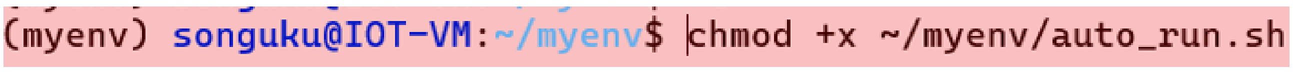
The script also prompted the user for confirmation before executing these commands, enhancing user experience.



```
Last login: Mon Dec  9 15:05:27 2024 from 93.165.251.147
Do you want to run the Python script? (y/n): 
```

Figure 13 - Automation script user prompt

For **auto_run.sh** to work it needed administrative access from the VM which was done by using the following command:



```
(myenv) songuku@IOT-VM:~/myenv$ chmod +x ~/myenv/auto_run.sh
```

Figure 14 - Automation script admin access

Transferring Files to the VM

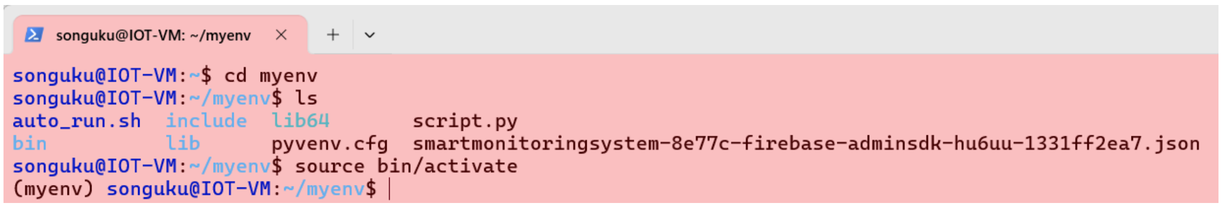
The necessary project files, including the Python script, Firebase configuration JSON file, and the automation script, were transferred to the VM using scp. For example, this is how the backend script was transferred:

```
scp -i "C:\Users\mikai\Desktop\IOT\SmartHome.pem" C:\Users\mikai\Desktop\IOT\script.py
songuku@52.164.218.228:/home/songuku/myenv/
```

Lastly to monitor active ports the following command was used to debug:

```
sudo netstat -tuln | grep 1883 || sudo netstat -tuln | grep 1884
```

For the python backend script to work it needs a set of dependencies and an environment to run on. This was done in a folder named **myenv**, created manually.:


songuku@IOT-VM: ~/myenv

```
songuku@IOT-VM:~$ cd myenv
songuku@IOT-VM:~/myenv$ ls
auto_run.sh  include  lib64      script.py
bin          lib      pyvenv.cfg  smartmonitoringsystem-8e77c-firebase-adminsdk-hu6uu-1331ff2ea7.json
songuku@IOT-VM:~/myenv$ source bin/activate
(myenv) songuku@IOT-VM:~/myenv$ |
```

Figure 15 - Python environment

Housing the script itself together with the firebase admin access Json file, the automation script **auto_run.sh** and python dependencies necessary for everything to work. This will be dwelled into with more details in the **Python** subsection.

MQTT Communication

The MQTT protocol was a key component of the Smart Monitoring System, providing efficient and lightweight communication between the ESP32 microcontrollers and the backend hosted on the Azure Virtual Machine (VM). MQTT

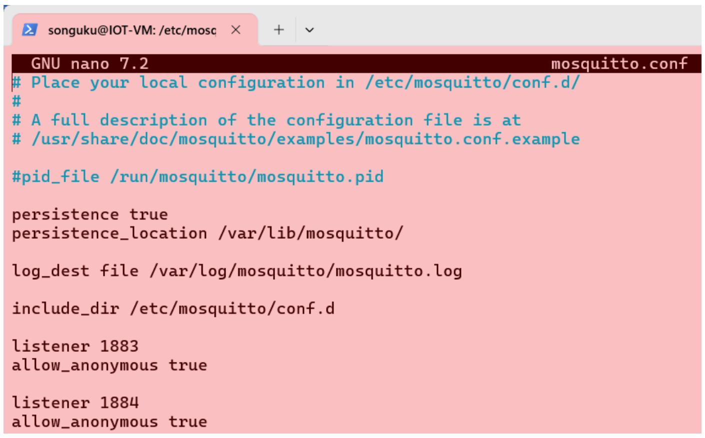
Configuration of Mosquitto Broker

The Mosquitto MQTT broker was installed and configured on the Azure VM to manage the communication between the IoT devices and the backend system. Its lightweight design and open-source nature made it an ideal choice for this project. Below is the configuration and setup process:

Configuration: The configuration file for Mosquitto was located at /etc/mosquitto/mosquitto.conf. The broker was set up to listen on two separate ports:

- Port 1883 for outdoor data
- Port 1884 for indoor data

This separation allowed for seamless integration of multiple devices without data collisions. The configuration file included the following settings:



```
  GNU nano 7.2                                               mosquitto.conf
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

#pid_file /run/mosquitto/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d

listener 1883
allow_anonymous true

listener 1884
allow_anonymous true
```

Figure 16 - Mosquitto conf settings

These settings enabled the broker to store persistent data, log events for debugging, and allow anonymous connections during the testing phase.

Testing MQTT Communication

The functionality of the Mosquitto broker was verified using its client tools (mosquitto_pub and mosquitto_sub) to simulate data transmission and reception:

1. **Publishing Test Data**: A test message was sent to the broker using the following command:

   ```
   mosquitto_pub -h 52.164.218.228 -t "test/topic" -m "MEOW" -p 1883
   ```

   Here, 52.164.218.228 is the broker's IP address, and 1883 is the port used for outdoor device communication.

1. **Subscribing to a Topic**: The published message was received by subscribing to the same topic:

   ```
   mosquitto_sub -h 52.164.218.228 -t "test/topic" -p 1883 -d
   ```

   The -d flag enabled debug mode, displaying detailed information about the MQTT messages.

This testing confirmed that the broker was functioning correctly, allowing data to flow seamlessly between the publisher (ESP32) and the subscriber (backend).

Backend (Python Script)

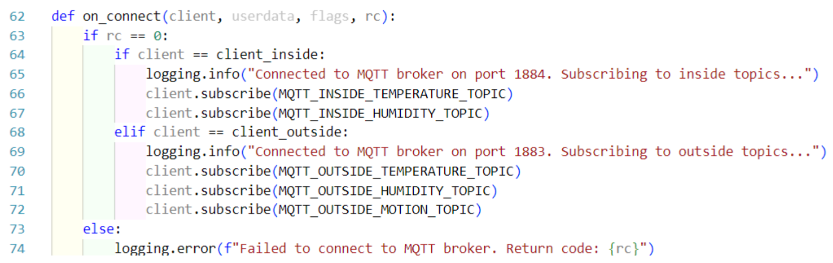
The backend of the Smart Monitoring System was implemented in Python to act as the intermediary layer between the MQTT broker and the Firebase Firestore database. This script played a critical role in processing, storing, and managing the sensor data collected from the ESP32 devices.

Key Functionalities

The backend script was designed with the following functionalities:

Connection to MQTT Broker: The script utilized the paho-mqtt library to connect to the Mosquitto MQTT broker. It subscribed to predefined topics to receive sensor data, including temperature, humidity, and motion events. Two clients were configured to handle separate ports for indoor and outdoor devices (1883 and 1884).

Snippet: MQTT Connection Logic

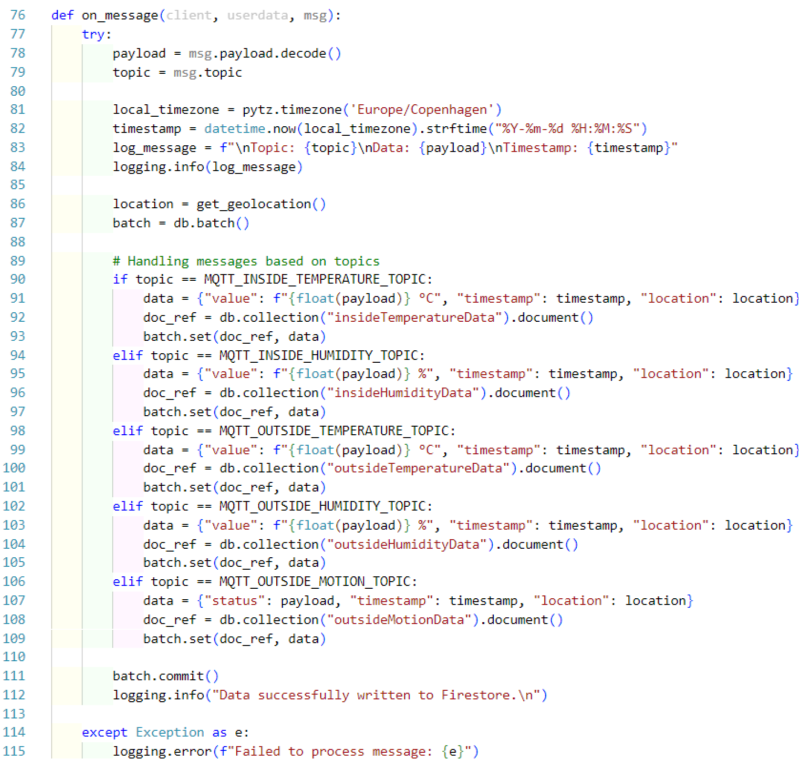


```python
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        if client == client_inside:
            logging.info("Connected to MQTT broker on port 1884. Subscribing to inside topics...")
            client.subscribe(MQTT_INSIDE_TEMPERATURE_TOPIC)
            client.subscribe(MQTT_INSIDE_HUMIDITY_TOPIC)
        elif client == client_outside:
            logging.info("Connected to MQTT broker on port 1883. Subscribing to outside topics...")
            client.subscribe(MQTT_OUTSIDE_TEMPERATURE_TOPIC)
            client.subscribe(MQTT_OUTSIDE_HUMIDITY_TOPIC)
            client.subscribe(MQTT_OUTSIDE_MOTION_TOPIC)
    else:
        logging.error(f"Failed to connect to MQTT broker. Return code: {rc}")
```

Figure 17 - Python - MQTT Connection Logic

Message Handling and Processing: The on_message callback function processed incoming messages from the MQTT broker. Each message's payload was parsed, enriched with metadata (timestamp, location), and organized into a structured JSON format before being stored in Firebase.

Snippet: Message Processing Logic

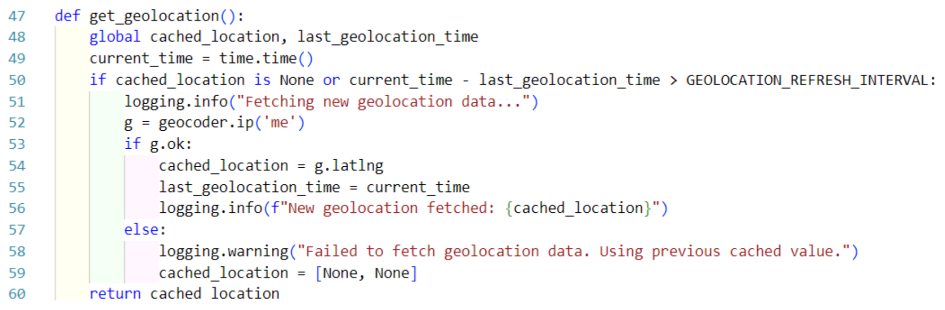


```python
def on_message(client, userdata, msg):
    try:
        payload = msg.payload.decode()
        topic = msg.topic

        local_timezone = pytz.timezone('Europe/Copenhagen')
        timestamp = datetime.now(local_timezone).strftime("%Y-%m-%d %H:%M:%S")
        log_message = f"\nTopic: {topic}\nData: {payload}\nTimestamp: {timestamp}"
        logging.info(log_message)

        location = get_geolocation()
        batch = db.batch()

        # Handling messages based on topics
        if topic == MQTT_INSIDE_TEMPERATURE_TOPIC:
            data = {"value": f"{float(payload)} °C", "timestamp": timestamp, "location": location}
            doc_ref = db.collection("insideTemperatureData").document()
            batch.set(doc_ref, data)
        elif topic == MQTT_INSIDE_HUMIDITY_TOPIC:
            data = {"value": f"{float(payload)} %", "timestamp": timestamp, "location": location}
            doc_ref = db.collection("insideHumidityData").document()
            batch.set(doc_ref, data)
        elif topic == MQTT_OUTSIDE_TEMPERATURE_TOPIC:
            data = {"value": f"{float(payload)} °C", "timestamp": timestamp, "location": location}
            doc_ref = db.collection("outsideTemperatureData").document()
            batch.set(doc_ref, data)
        elif topic == MQTT_OUTSIDE_HUMIDITY_TOPIC:
            data = {"value": f"{float(payload)} %", "timestamp": timestamp, "location": location}
            doc_ref = db.collection("outsideHumidityData").document()
            batch.set(doc_ref, data)
        elif topic == MQTT_OUTSIDE_MOTION_TOPIC:
            data = {"status": payload, "timestamp": timestamp, "location": location}
            doc_ref = db.collection("outsideMotionData").document()
            batch.set(doc_ref, data)

        batch.commit()
        logging.info("Data successfully written to Firestore.\n")

    except Exception as e:
        logging.error(f"Failed to process message: {e}")
```

Figure 18 - Data Processing Logic

Geolocation Integration: To add spatial context to the sensor data, the script fetched geolocation information using a geocoding API[4]. The location data was cached to minimize API calls and ensure efficiency. This API added extra useful data for the system to where the data comes from.

Snippet: Geolocation Retrieval

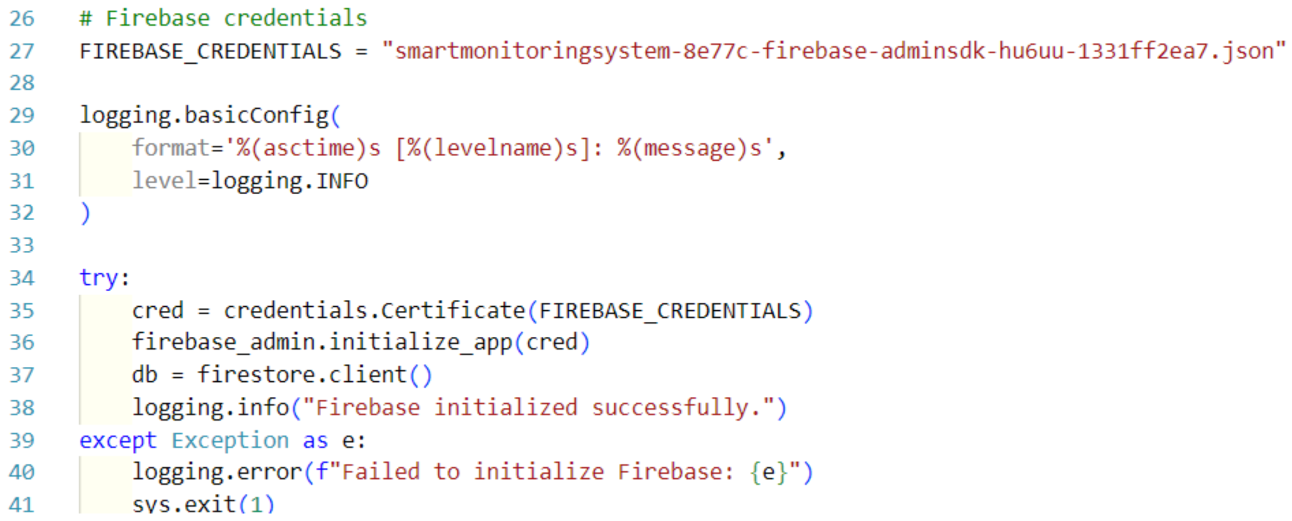


```python
def get_geolocation():
    global cached_location, last_geolocation_time
    current_time = time.time()
    if cached_location is None or current_time - last_geolocation_time > GEOLOCATION_REFRESH_INTERVAL:
        logging.info("Fetching new geolocation data...")
        g = geocoder.ip('me')
        if g.ok:
            cached_location = g.latlng
            last_geolocation_time = current_time
            logging.info(f"New geolocation fetched: {cached_location}")
        else:
            logging.warning("Failed to fetch geolocation data. Using previous cached value.")
            cached_location = [None, None]
    return cached_location
```

Figure 19 - Geolocation Retrieval

[4] https://pypi.org/project/geocoder/

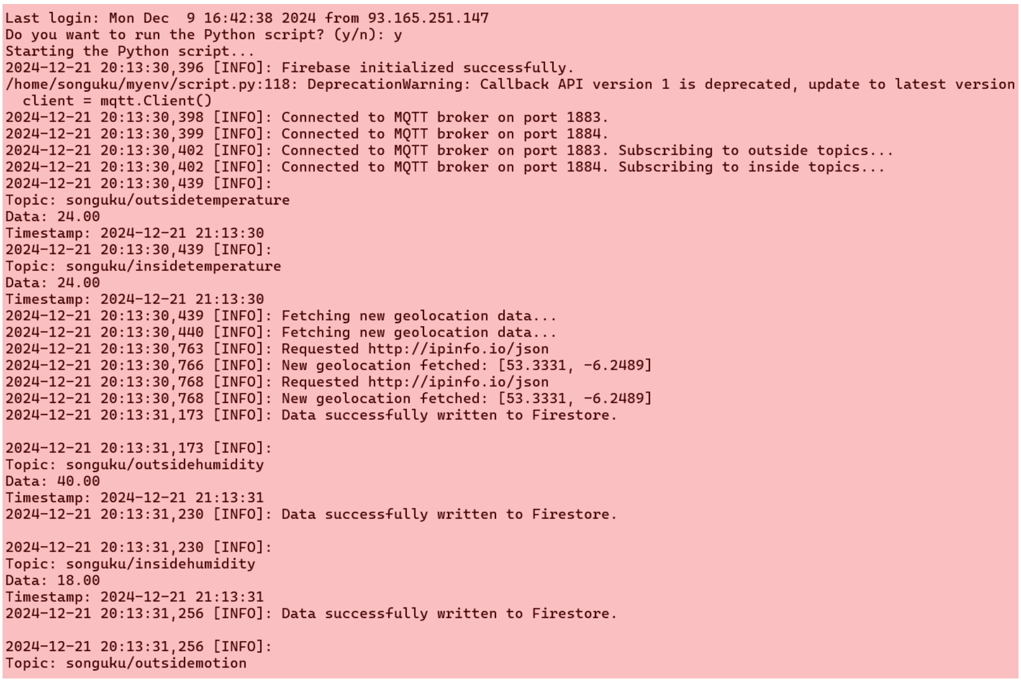
Most importantly the firebase admin sdk json file was used to write to the database:



```python
# Firebase credentials
FIREBASE_CREDENTIALS = "smartmonitoringsystem-8e77c-firebase-adminsdk-hu6uu-1331ff2ea7.json"

logging.basicConfig(
    format='%(asctime)s [%(levelname)s]: %(message)s',
    level=logging.INFO
)

try:
    cred = credentials.Certificate(FIREBASE_CREDENTIALS)
    firebase_admin.initialize_app(cred)
    db = firestore.client()
    logging.info("Firebase initialized successfully.")
except Exception as e:
    logging.error(f"Failed to initialize Firebase: {e}")
    sys.exit(1)
```

Figure 20 - Reading firebase admin sdk json file

Upon running the python backend, the following output is successfully shown in the terminal, how the data is read and published:



```
Last login: Mon Dec  9 16:42:38 2024 from 93.165.251.147
Do you want to run the Python script? (y/n): y
Starting the Python script...
2024-12-21 20:13:30,396 [INFO]: Firebase initialized successfully.
/home/songuku/myenv/script.py:118: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
2024-12-21 20:13:30,398 [INFO]: Connected to MQTT broker on port 1883.
2024-12-21 20:13:30,399 [INFO]: Connected to MQTT broker on port 1884.
2024-12-21 20:13:30,402 [INFO]: Connected to MQTT broker on port 1883. Subscribing to outside topics...
2024-12-21 20:13:30,402 [INFO]: Connected to MQTT broker on port 1884. Subscribing to inside topics...
2024-12-21 20:13:30,439 [INFO]:
Topic: songuku/outsidetemperature
Data: 24.00
Timestamp: 2024-12-21 21:13:30
2024-12-21 20:13:30,439 [INFO]:
Topic: songuku/insidetemperature
Data: 24.00
Timestamp: 2024-12-21 21:13:30
2024-12-21 20:13:30,439 [INFO]: Fetching new geolocation data...
2024-12-21 20:13:30,440 [INFO]: Fetching new geolocation data...
2024-12-21 20:13:30,763 [INFO]: Requested http://ipinfo.io/json
2024-12-21 20:13:30,766 [INFO]: New geolocation fetched: [53.3331, -6.2489]
2024-12-21 20:13:30,768 [INFO]: Requested http://ipinfo.io/json
2024-12-21 20:13:30,768 [INFO]: New geolocation fetched: [53.3331, -6.2489]
2024-12-21 20:13:31,173 [INFO]: Data successfully written to Firestore.

2024-12-21 20:13:31,173 [INFO]:
Topic: songuku/outsidehumidity
Data: 40.00
Timestamp: 2024-12-21 21:13:31
2024-12-21 20:13:31,230 [INFO]: Data successfully written to Firestore.

2024-12-21 20:13:31,230 [INFO]:
Topic: songuku/insidehumidity
Data: 18.00
Timestamp: 2024-12-21 21:13:31
2024-12-21 20:13:31,256 [INFO]: Data successfully written to Firestore.

2024-12-21 20:13:31,256 [INFO]:
Topic: songuku/outsidemotion
```

Figure 21 - Running python script output snippet

Database (Firestore)

The database layer of the Smart Monitoring System was implemented using Google Firebase Firestore, a NoSQL cloud database known for its real-time synchronization capabilities and seamless integration with IoT systems. Firestore served as the central repository for storing and managing sensor data transmitted by the ESP32 devices through the MQTT broker and processed by the Python backend.

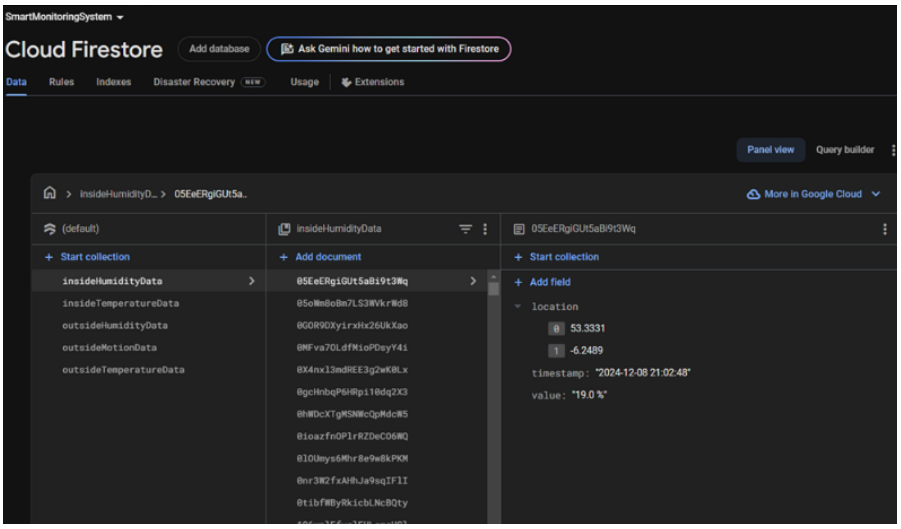
Design and Setup

Firestore was structured to handle different types of IoT data, such as temperature, humidity, and motion events, ensuring efficient organization and easy retrieval. The database schema was divided into distinct collections for each data type:

- insideTemperatureData
- insideHumidityData
- outsideTemperatureData
- outsideHumidityData
- outsideMotionData

Each collection stored documents containing sensor readings enriched with metadata, including timestamps and geolocation.

Snippet: Example Document Structure



insideHumidityD... > 05EeERgiGUt5a..

(default)
- insideHumidityData
- insideTemperatureData
- outsideHumidityData
- outsideMotionData
- outsideTemperatureData

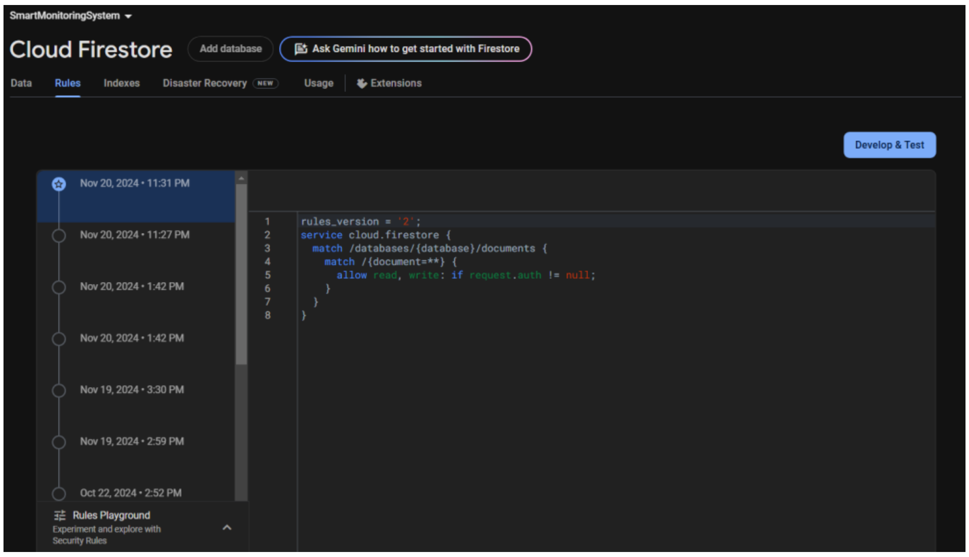
insideHumidityData
- 05EeERgiGUt5aBi9t3Wq
- 05oWm8oBm7LS3WVkrWd8
- 0GOR9DXyirxHx26UkXao
- 0MFva7OLdfMioPDsyY4i
- 0X4nx13mdREE3g2wK0Lx
- 0gcHnbqP6HRpi10dq2X3
- 0hWDcXTgMSNWcQpMdcW5
- 0ioazfnOPlrRZDeCO6WQ
- 0lOUmys6Mhr8e9w8kPKM
- 0nr3W2fxAHhJa9sqIFlI
- 0tibfWByRkicbLNcBQty

05EeERgiGUt5aBi9t3Wq
```
location
    0  53.3331
    1  -6.2489
timestamp: "2024-12-08 21:02:48"
value: "19.0 %"
```

Figure 22 - Document structure snippet

Database Rules

Firestore's security relies on **rules** defined to manage access. The rules implemented for this project ensure that only authenticated users can read or write to the database:



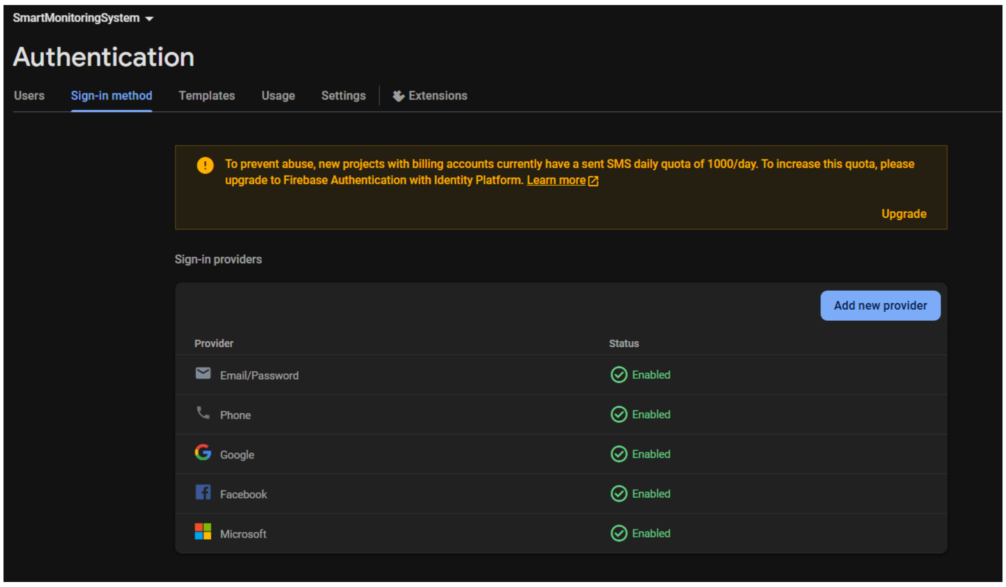
```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Figure 23 - Firestore rules

These rules require that all database requests are authenticated, safeguarding data from unauthorized access while supporting real-time interactions.

Authentication

To allow users to access the Firestore database securely, Firebase Authentication was employed. Multiple sign-in methods were enabled, providing flexibility and ease of access:



To prevent abuse, new projects with billing accounts currently have a sent SMS daily quota of 1000/day. To increase this quota, please upgrade to Firebase Authentication with Identity Platform. Learn more

Sign-in providers

Provider	Status
Email/Password	Enabled
Phone	Enabled
Google	Enabled
Facebook	Enabled
Microsoft	Enabled

Figure 24 - Authentication setup

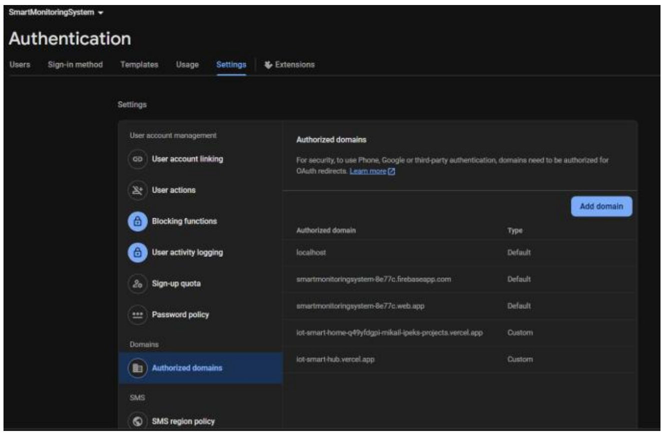
- **Google Authentication**: Simplifies the sign-in process for users with Google accounts.
- **Email/Password Authentication**: Offers a traditional sign-in method for users without social media accounts.

This multi-method authentication setup ensures inclusivity while maintaining secure data access. Firebase Authentication integrates seamlessly with Firestore, automatically validating users during database interactions. Only the above mentioned are functioning, while Facebook, microsoft and phone required extra steps, why it was ditched eventually, as google and email/password is sufficient.

Domain Authentication

To enable seamless communication between the React frontend and Firestore, **authorized domains** were manually configured. The following trusted domains were added:

Configuring these domains ensured that the web application could securely communicate with Firestore, preventing unauthorized sources from accessing the database.



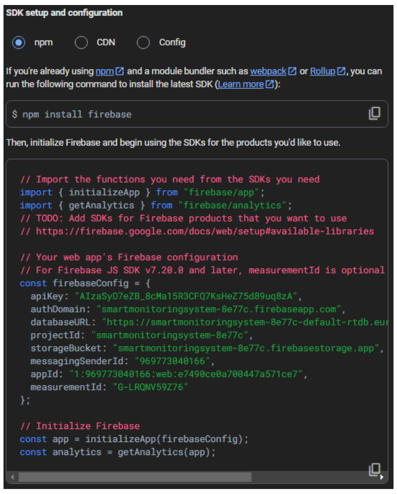
Authorized domains

For security, to use Phone, Google or third-party authentication, domains need to be authorized for OAuth redirects. Learn more

Authorized domain	Type
localhost	Default
smartmonitoringsystem-8e77c.firebaseapp.com	Default
smartmonitoringsystem-8e77c.web.app	Default
iot-smart-home-q49yfdgpi-mikail-ipeks-projects.vercel.app	Custom
iot-smart-hub.vercel.app	Custom

Figure 25 - Trusted domains

Firestore Initialization: The Firebase Admin SDK was used to connect the backend Python script with Firestore (mentioned earlier) and Firestore with the frontend. The following JSON service account key file facilitated secure communication can be seen. Here the npm package manager was chosen as it is easy to use and maintain:


SDK setup and configuration

[✓ npm] CDN Config

If you're already using npm and a module bundler such as webpack or Rollup, you can run the following command to install the latest SDK (Learn more):

```
$ npm install firebase
```

Then, initialize Firebase and begin using the SDKs for the products you'd like to use.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
import { getAnalytics } from "firebase/analytics";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyD7eZB_8cMa15R3CFQ7KsHeZ75d89uq8zA",
  authDomain: "smartmonitoringsystem-8e77c.firebaseapp.com",
  databaseURL: "https://smartmonitoringsystem-8e77c-default-rtdb.eu
  projectId: "smartmonitoringsystem-8e77c",
  storageBucket: "smartmonitoringsystem-8e77c.firebasestorage.app",
  messagingSenderId: "969773040166",
  appId: "1:969773040166:web:e7490ce0a700447a571ce7",
  measurementId: "G-LRQNV59Z76"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const analytics = getAnalytics(app);
```

Figure 26 - JSON service account key file

Frontend (React)

Design

The frontend of the Smart Monitoring System was designed to create an intuitive, responsive, and interactive user interface. The goal was to ensure seamless visualization and interaction with real-time IoT data. Using **React**, the system leverages its component-based architecture to handle dynamic updates efficiently. **Material-UI (MUI)** was chosen for its extensive library of pre-styled components, enabling a modern and professional design with minimal custom CSS. To visualize historical data trends, **Chart.js** was integrated, providing flexible and interactive charts.

The application was structured into three main sections:

1. **Authentication**: A secure login system with multiple sign-in methods.
2. **Dashboard**: Real-time monitoring of sensor data (temperature, humidity, motion detection).
3. **Graphs Page**: Historical trend visualization using dynamic charts.

Implementation

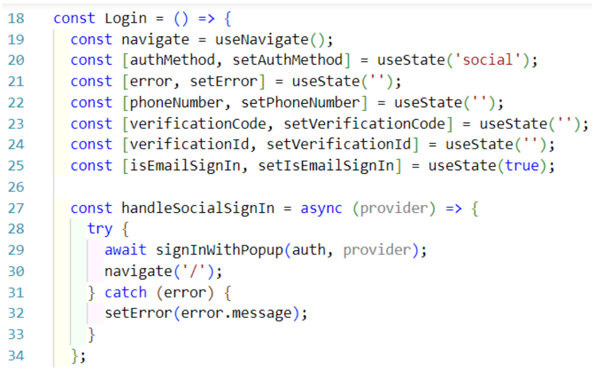
Authentication

The authentication system was implemented using **Firebase Authentication**, offering multiple login methods to enhance user convenience and security:

- **Social Login** with Google, Facebook, and Microsoft, achieved via Firebase's predefined providers.
- **Email/Password Login** for traditional user authentication.
- Although initially planned, **Phone Login** was excluded due to complexities in integrating ReCAPTCHA effectively.

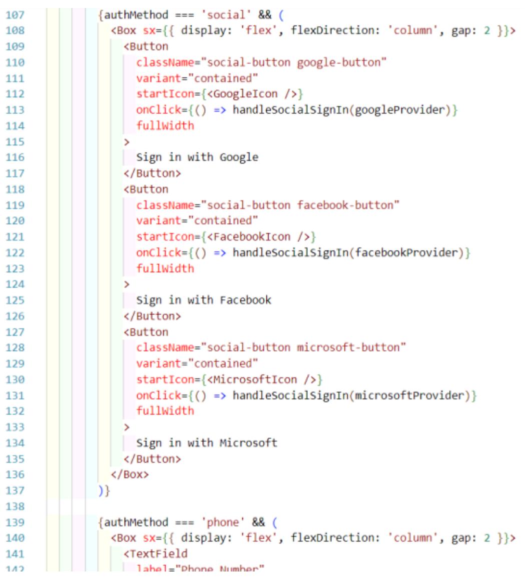
The **Login Component** was designed with a tabbed layout to switch between login methods. Firebase handles the backend logic, while React components manage the user interface.

Example Code:



```
const Login = () => {
  const navigate = useNavigate();
  const [authMethod, setAuthMethod] = useState('social');
  const [error, setError] = useState('');
  const [phoneNumber, setPhoneNumber] = useState('');
  const [verificationCode, setVerificationCode] = useState('');
  const [verificationId, setVerificationId] = useState('');
  const [isEmailSignIn, setIsEmailSignIn] = useState(true);

  const handleSocialSignIn = async (provider) => {
    try {
      await signInWithPopup(auth, provider);
      navigate('/');
    } catch (error) {
      setError(error.message);
    }
  };
```

Figure 27 - Social sign in form snippet



```
{authMethod === 'social' && (
  <Box sx={{ display: 'flex', flexDirection: 'column', gap: 2 }}>
    <Button
      className="social-button google-button"
      variant="contained"
      startIcon={<GoogleIcon />}
      onClick={() => handleSocialSignIn(googleProvider)}
      fullWidth
    >
      Sign in with Google
    </Button>
    <Button
      className="social-button facebook-button"
      variant="contained"
      startIcon={<FacebookIcon />}
      onClick={() => handleSocialSignIn(facebookProvider)}
      fullWidth
    >
      Sign in with Facebook
    </Button>
    <Button
      className="social-button microsoft-button"
      variant="contained"
      startIcon={<MicrosoftIcon />}
      onClick={() => handleSocialSignIn(microsoftProvider)}
      fullWidth
    >
      Sign in with Microsoft
    </Button>
  </Box>
)}

{authMethod === 'phone' && (
  <Box sx={{ display: 'flex', flexDirection: 'column', gap: 2 }}>
    <TextField
      label="Phone Number"
```

Figure 28 - Sign In handler method

This approach ensures that each provider's unique requirements (e.g., Google's account selection prompt) are properly managed.

The login page was properly designed with popping colours and great design, showing all sign in options beautifully:

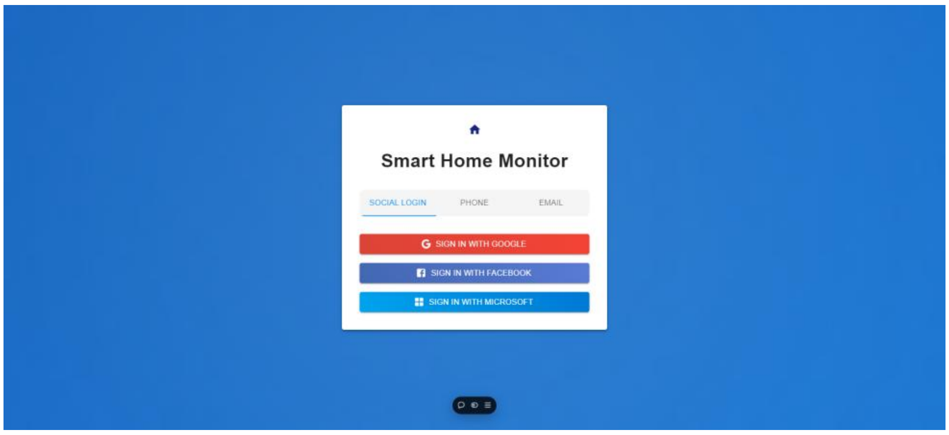


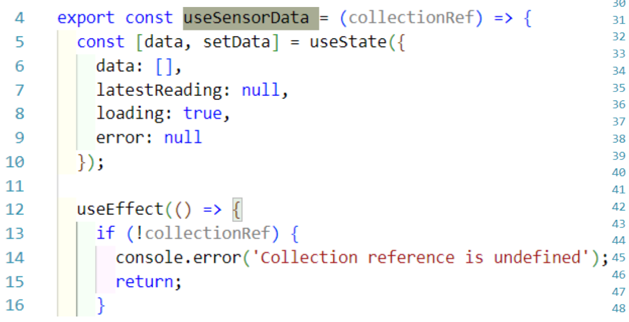
Figure 29 - Sign In page

Dashboard

The dashboard was designed to display real-time sensor data. Each data type (e.g., indoor/outdoor temperature and humidity, motion detection) is fetched dynamically from **Google Firestore**.

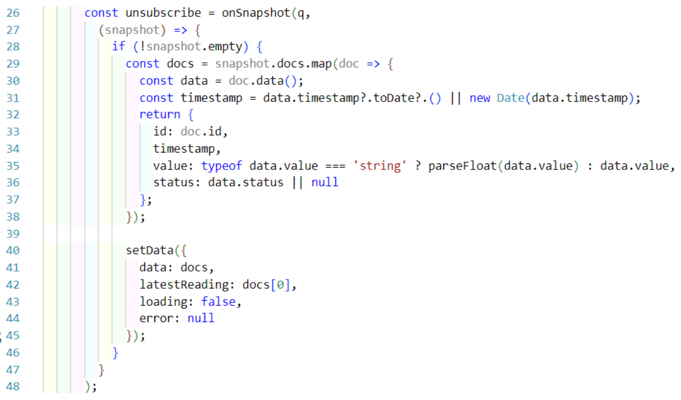
To achieve real-time updates, the **useSensorData** custom hook was created in **useSensorData.js**, utilizing Firestore's snapshot listener to fetch and update data whenever the database changes. This was to simplify the process of fetching and managing real-time sensor data from a Firebase Firestore database The real-time data is captured with the **onSnapshot** function.

This is a snippet of the hook is set up:



```javascript
export const useSensorData = (collectionRef) => {
  const [data, setData] = useState({
    data: [],
    latestReading: null,
    loading: true,
    error: null
  });

  useEffect(() => {
    if (!collectionRef) {
      console.error('Collection reference is undefined');
      return;
    }
```

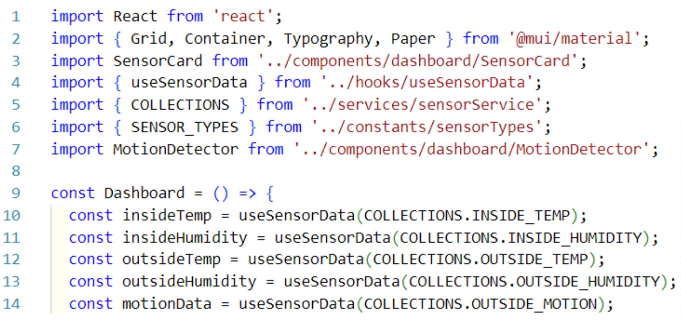
Figure 30 - onSnapshot function



```javascript
      const unsubscribe = onSnapshot(q,
        (snapshot) => {
          if (!snapshot.empty) {
            const docs = snapshot.docs.map(doc => {
              const data = doc.data();
              const timestamp = data.timestamp?.toDate?.() || new Date(data.timestamp);
              return {
                id: doc.id,
                timestamp,
                value: typeof data.value === 'string' ? parseFloat(data.value) : data.value,
                status: data.status || null
              };
            });

            setData({
              data: docs,
              latestReading: docs[0],
              loading: false,
              error: null
            });
          }
        }
      );
```

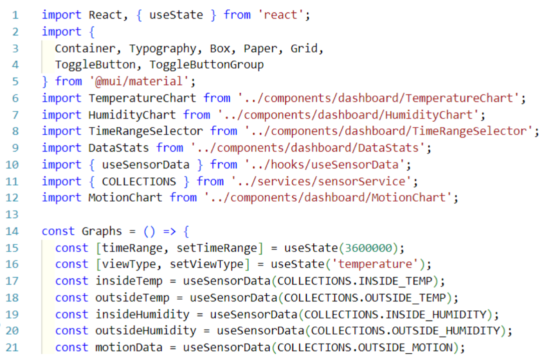
Figure 31 - useSensorData hook

Thanks to this hook, it can be utilized by the two most important files, **Dashboard.js** and **Graphs.js** respectively:



```javascript
import React from 'react';
import { Grid, Container, Typography, Paper } from '@mui/material';
import SensorCard from '../components/dashboard/SensorCard';
import { useSensorData } from '../hooks/useSensorData';
import { COLLECTIONS } from '../services/sensorService';
import { SENSOR_TYPES } from '../constants/sensorTypes';
import MotionDetector from '../components/dashboard/MotionDetector';

const Dashboard = () => {
  const insideTemp = useSensorData(COLLECTIONS.INSIDE_TEMP);
  const insideHumidity = useSensorData(COLLECTIONS.INSIDE_HUMIDITY);
  const outsideTemp = useSensorData(COLLECTIONS.OUTSIDE_TEMP);
  const outsideHumidity = useSensorData(COLLECTIONS.OUTSIDE_HUMIDITY);
  const motionData = useSensorData(COLLECTIONS.OUTSIDE_MOTION);
```

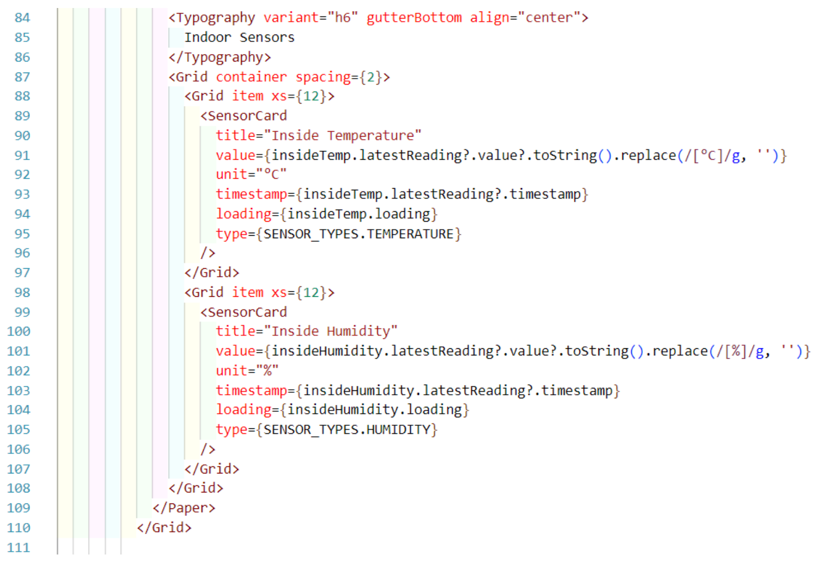
Figure 32 - Graphs.js utilizing hooks



```javascript
import React, { useState } from 'react';
import {
  Container, Typography, Box, Paper, Grid,
  ToggleButton, ToggleButtonGroup
} from '@mui/material';
import TemperatureChart from '../components/dashboard/TemperatureChart';
import HumidityChart from '../components/dashboard/HumidityChart';
import TimeRangeSelector from '../components/dashboard/TimeRangeSelector';
import DataStats from '../components/dashboard/DataStats';
import { useSensorData } from '../hooks/useSensorData';
import { COLLECTIONS } from '../services/sensorService';
import MotionChart from '../components/dashboard/MotionChart';

const Graphs = () => {
  const [timeRange, setTimeRange] = useState(3600000);
  const [viewType, setViewType] = useState('temperature');
  const insideTemp = useSensorData(COLLECTIONS.INSIDE_TEMP);
  const outsideTemp = useSensorData(COLLECTIONS.OUTSIDE_TEMP);
  const insideHumidity = useSensorData(COLLECTIONS.INSIDE_HUMIDITY);
  const outsideHumidity = useSensorData(COLLECTIONS.OUTSIDE_HUMIDITY);
  const motionData = useSensorData(COLLECTIONS.OUTSIDE_MOTION);
```

Figure 33 - Dashboard.js utilizing hooks

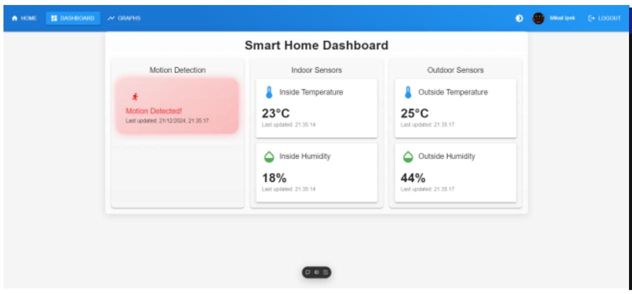
Snippet from **Dashboard.js** that defines the functionality of indoor sensors:



```
    <Typography variant="h6" gutterBottom align="center">
      Indoor Sensors
    </Typography>
    <Grid container spacing={2}>
      <Grid item xs={12}>
        <SensorCard
          title="Inside Temperature"
          value={insideTemp.latestReading?.value?.toString().replace(/[°C]/g, '')}
          unit="°C"
          timestamp={insideTemp.latestReading?.timestamp}
          loading={insideTemp.loading}
          type={SENSOR_TYPES.TEMPERATURE}
        />
      </Grid>
      <Grid item xs={12}>
        <SensorCard
          title="Inside Humidity"
          value={insideHumidity.latestReading?.value?.toString().replace(/[%]/g, '')}
          unit="%"
          timestamp={insideHumidity.latestReading?.timestamp}
          loading={insideHumidity.loading}
          type={SENSOR_TYPES.HUMIDITY}
        />
      </Grid>
    </Grid>
  </Paper>
</Grid>
```

Figure 34 - Dashboard.js indoor sensor snippet

The dashboard page with the motion data and indoor / outdoor sensor data looks like:



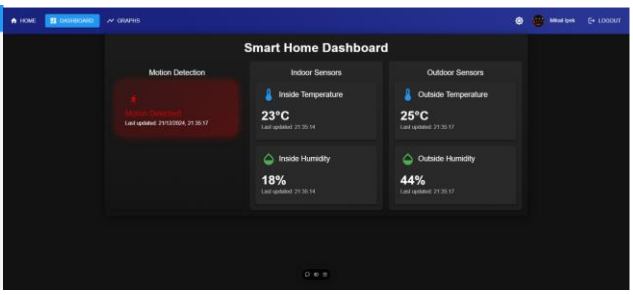


Figure 35 - dashboard page - white mode

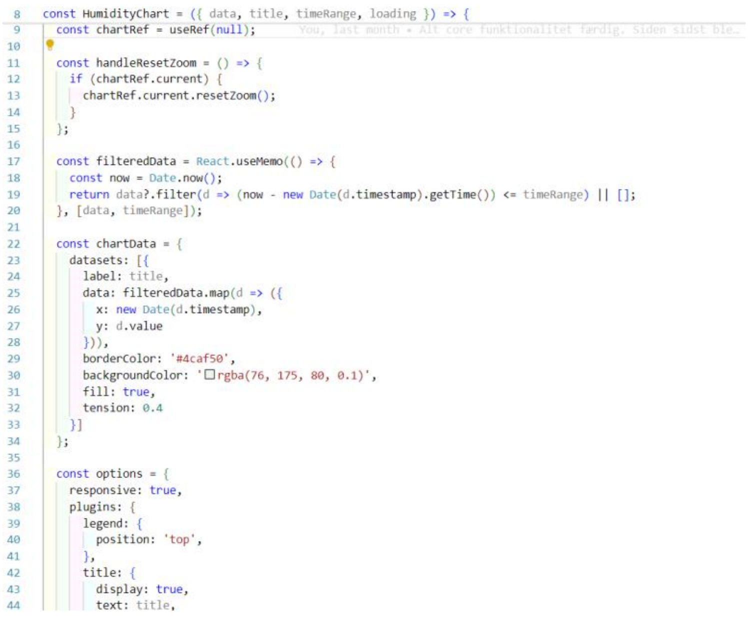
Figure 36 - dashboard page - dark mode

Dark mode was also implemented to further enhance user experience.

Graphs Page

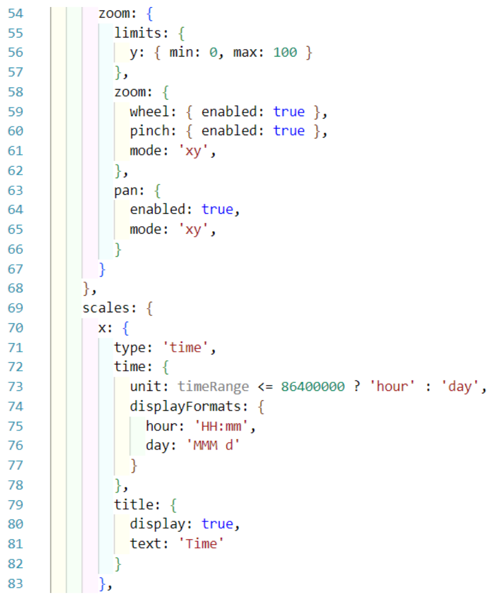
The graphs page visualizes historical sensor data trends using **Chart.js**. Charts are interactive, allowing users to zoom, pan, and filter data based on a selected time range. The page was built to support multiple data types, such as temperature, humidity, and motion events.

The **TemperatureChart** and **HumidityChart** components are used for rendering line charts. The data is pre-processed to group readings by timestamps, and each chart dynamically updates when the time range is modified by the user.



```javascript
const HumidityChart = ({ data, title, timeRange, loading }) => {
  const chartRef = useRef(null);        You, last month • Alt core funktionalitet færdig. Siden sidst ble…

  const handleResetZoom = () => {
    if (chartRef.current) {
      chartRef.current.resetZoom();
    }
  };

  const filteredData = React.useMemo(() => {
    const now = Date.now();
    return data?.filter(d => (now - new Date(d.timestamp).getTime()) <= timeRange) || [];
  }, [data, timeRange]);

  const chartData = {
    datasets: [{
      label: title,
      data: filteredData.map(d => ({
        x: new Date(d.timestamp),
        y: d.value
      })),
      borderColor: '#4caf50',
      backgroundColor: '□rgba(76, 175, 80, 0.1)',
      fill: true,
      tension: 0.4
    }]
  };

  const options = {
    responsive: true,
    plugins: {
      legend: {
        position: 'top',
      },
      title: {
        display: true,
        text: title,
```

Figure 37 - HumidityChart snippet 1



```javascript
      zoom: {
        limits: {
          y: { min: 0, max: 100 }
        },
        zoom: {
          wheel: { enabled: true },
          pinch: { enabled: true },
          mode: 'xy',
        },
        pan: {
          enabled: true,
          mode: 'xy',
        }
      }
    },
    scales: {
      x: {
        type: 'time',
        time: {
          unit: timeRange <= 86400000 ? 'hour' : 'day',
          displayFormats: {
            hour: 'HH:mm',
            day: 'MMM d'
          }
        },
        title: {
          display: true,
          text: 'Time'
        }
      },
```

Figure 38 - HumidityChart snippet 2

Above is an example from **HumidityChart.js** showing snippets of its filter functionality and looks like this:

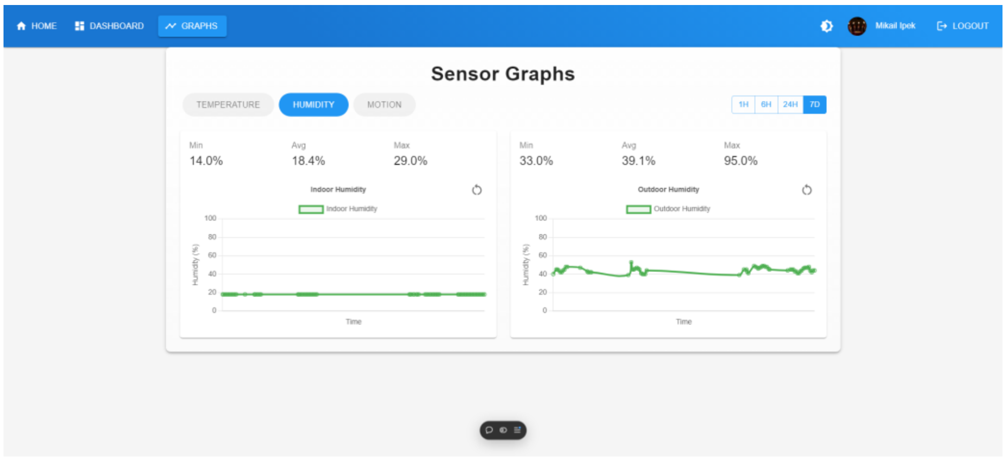
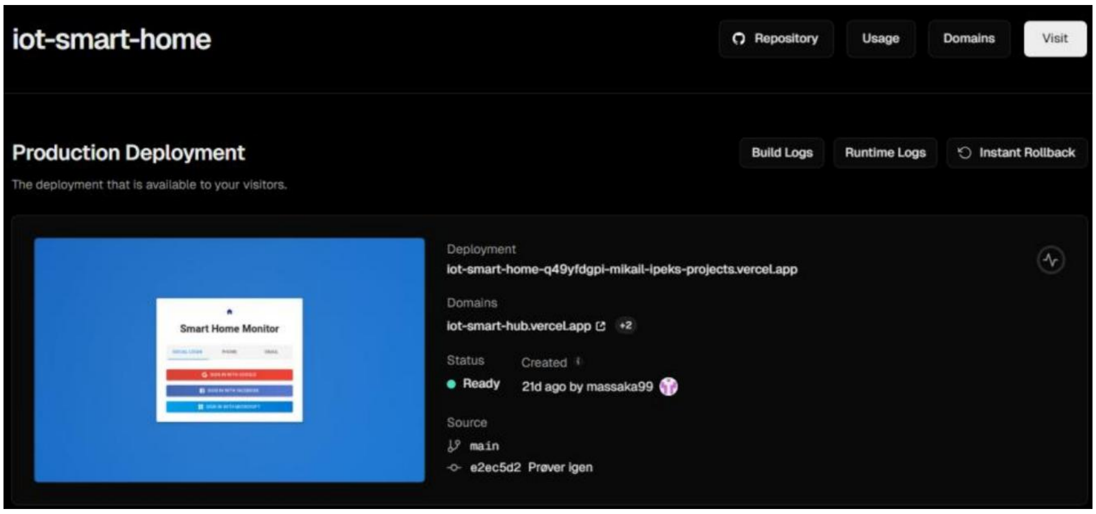


Figure 39 - Sensor Graphs page

Deployment

The application was deployed on Vercel, taking advantage of the seamless integration with GitHub and React-based projects. This was done to make the website accessible from wherever:


iot-smart-home

Repository | Usage | Domains | Visit

Production Deployment

The deployment that is available to your visitors.

Build Logs | Runtime Logs | Instant Rollback

Deployment
iot-smart-home-q49yfdgpi-mikail-ipeks-projects.vercel.app

Domains
iot-smart-hub.vercel.app +2

Status: Ready
Created: 21d ago by massaka99

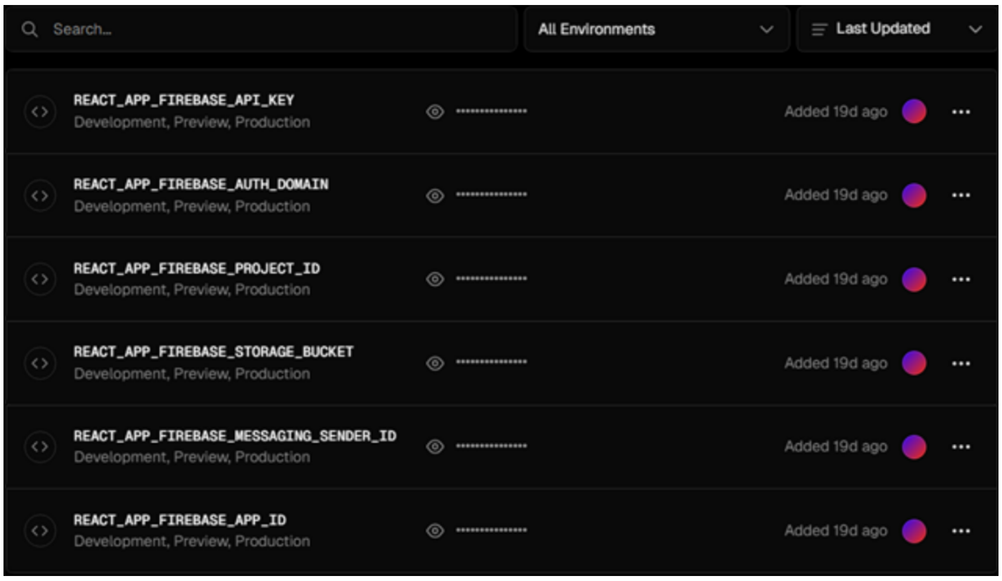
Source
main
e2ec5d2 Prøver igen

Figure 40 - Website hosted on Vercel

Link to webpage hosted on Vercel: https://iot-smart-hub.vercel.app/login

If one wants to run the frontend through a coding IDE, this can be done by simply running the command **npm start**, after running **npm install**.

Within Vercel, the firebase configuration details (API keys, project IDs) were stored in Vercel environment variables to enhance security. These were excluded from the repository in (.gitignore) for best practices.


Search... | All Environments | Last Updated

REACT_APP_FIREBASE_API_KEY Development, Preview, Production	•••••••••••••••	Added 19d ago
REACT_APP_FIREBASE_AUTH_DOMAIN Development, Preview, Production	•••••••••••••••	Added 19d ago
REACT_APP_FIREBASE_PROJECT_ID Development, Preview, Production	•••••••••••••••	Added 19d ago
REACT_APP_FIREBASE_STORAGE_BUCKET Development, Preview, Production	•••••••••••••••	Added 19d ago
REACT_APP_FIREBASE_MESSAGING_SENDER_ID Development, Preview, Production	•••••••••••••••	Added 19d ago
REACT_APP_FIREBASE_APP_ID Development, Preview, Production	•••••••••••••••	Added 19d ago

Figure 41 - Vercel environment variables

Results

The IoT-based monitoring system successfully met its main goals, providing real-time tracking of temperature, humidity, and motion using cloud-based services. The project brought together hardware devices, cloud platforms, and a web interface to ensure smooth data collection, processing, and display.

The two ESP32 microcontrollers worked well, sending sensor data to the Mosquitto MQTT broker hosted on an Azure Virtual Machine. A Python script processed this data, adding timestamps and geolocation details before storing it in Firebase Firestore. This setup allowed real-time updates on the web application built with React and deployed on Vercel.

The web app displayed data through a live dashboard with updates, trend graphs, and motion alerts. Users could monitor both indoor and outdoor conditions, showing the system's reliability and accuracy. The interface was clear and easy to use, supporting the project's goal of creating a practical monitoring tool.

While there were some initial problems, like setting up hardware drivers and configuring dual MQTT ports on the Azure VM, these issues were resolved through research and testing. The DHT11 sensor, though less precise than the DHT22, worked well enough for the project's needs.

To see the system in action and its results, including how sensor data is processed and displayed, refer to the simulation videos, which demonstrates the entire setup working as intended, from start to finish. The first video is filmed with phone and second is a screencast. Both the software and hardware are shown.

Discussions and Challenges

Port Configuration Issues with ESP32 Microcontrollers

One of the initial challenges faced during the project was managing the ports for the ESP32 microcontrollers when connecting them to the PC. Normally, the drivers required for ESP32 devices are installed automatically upon detection by windows. However, this process failed in our setup, resulting in significant delays. We spent nearly two hours troubleshooting the issue, only to discover that in our case, the drivers needed to be manually installed from the manufacturer's website, silicon labs. Once installed, the ESP32 devices were successfully recognized, allowing us to proceed with the implementation. This challenge underscored the importance of verifying hardware compatibility and ensuring necessary drivers are installed during the initial stages of the project.

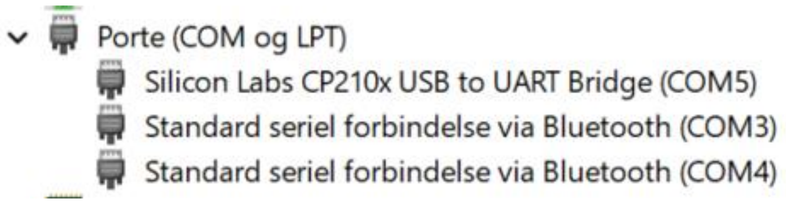

Porte (COM og LPT)
Silicon Labs CP210x USB to UART Bridge (COM5)
Standard seriel forbindelse via Bluetooth (COM3)
Standard seriel forbindelse via Bluetooth (COM4)

Figure 42 - COM ports in windows device manager

MQTT Port Issues

At the outset, when the system had only one ESP32 device, using a single MQTT port on the Azure VM worked. However, as the system expanded to include a second ESP32 device, conflicts arose because both devices attempted to communicate via the same port. This led to data collisions, where the devices interfered with each other's communication. After considerable research and a trial-and-error process, we resolved the issue by configuring a separate MQTT port for the second ESP32 device. Assigning unique ports (1883 for the outdoor device and 1884 for the indoor device) eliminated the interference and restored smooth communication.

Python Environment Setup

Setting up the Python environment on the Azure VM was another notable challenge. Installing the necessary libraries and configuring the environment to run the Python script proved more complex than anticipated. We had to carefully manage library dependencies and ensure compatibility between the script and the system environment. Despite these challenges, the use of a virtual environment helped isolate the project setup and ensure a clean runtime environment.

Frontend Deployment Challenges

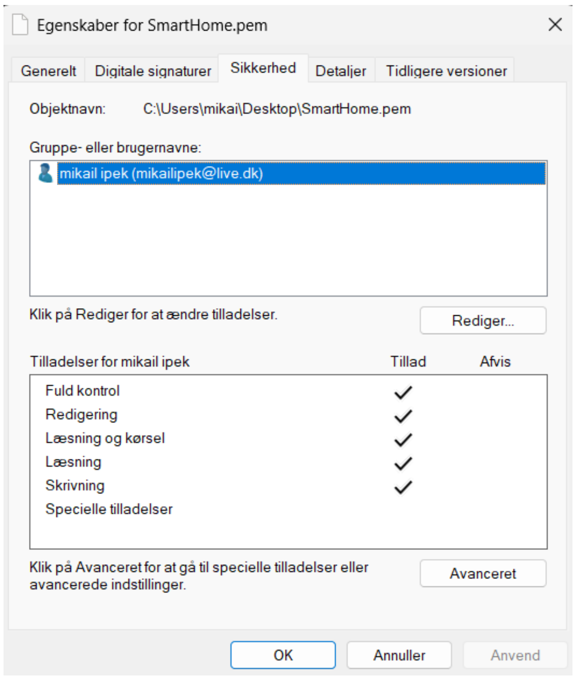
Initially, we attempted to deploy the frontend application using Firebase Hosting. However, the process was full of errors that were difficult to diagnose and resolve. Firebase Hosting's complexity and badly optimized deployment configuration requirements led to several deployment failures. Ultimately, we opted to use Vercel, which provided a more streamlined and intuitive deployment process. Vercel's direct integration with GitHub allowed us to fetch the latest commits and deploy the application with minimal effort. This decision saved significant time and provided a more reliable deployment platform.

Firebase Database Rules

Another challenge involved configuring Firebase's database rules to support both read and write operations from the frontend. At first, the restrictive rules caused errors when attempting to fetch or write data to the Firestore database. After experimenting with various configurations, we found a rule that balanced functionality and security. Although the current rules are not the most secure, they have authentication enabled to restrict access to authorized users.

SSH Key Permissions for Azure VM Access

Accessing the Azure VM using the **smarthome.pem** file presented initial difficulties. We discovered that the SSH key would not work if multiple users had administrative access to the file. By default, Windows assigns admin permissions to several system accounts, which caused the key to be rejected. To resolve this, we manually removed unnecessary administrative access and ensured that only our user account had control over the file. This adjustment enabled us to connect to the VM and proceed with the project setup.


Egenskaber for SmartHome.pem

Objektnavn: C:\Users\mikai\Desktop\SmartHome.pem

Gruppe- eller brugernavne:
mikail ipek (mikailipek@live.dk)

Klik på Rediger for at ændre tilladelser.

Tilladelser for mikail ipek	Tillad	Afvis
Fuld kontrol	✓	
Redigering	✓	
Læsning og kørsel	✓	
Læsning	✓	
Skrivning	✓	
Specielle tilladelser		

Klik på Avanceret for at gå til specielle tilladelser eller avancerede indstillinger.

Figure 43 - SSH Key Permissions

Conclusion

The project went well, and we successfully completed all the goals we set out to achieve. The system we built works as intended, providing real-time monitoring of temperature, humidity, and motion using a combination of hardware and cloud-based technologies. The chosen tools and platforms, like ESP32 microcontrollers, Firebase Firestore, React.js, and the Azure Virtual Machine, worked together effectively to deliver a reliable system.

One of the main highlights of this project was how much we learned about planning and solving problems. Challenges like setting up MQTT ports, fixing ESP32 driver issues, and deploying the frontend pushed us to find solutions through research and testing. These experiences made the system stronger and helped us understand how to tackle such problems in the future.

This project taught us a lot about IoT, especially how hardware and cloud systems need to work together smoothly. Using Firebase for real-time data synchronization and React for a dynamic and interactive dashboard made it clear how important it is to design systems that are easy for users to interact with.

Feedback on the IoT Project and Course

The IoT project was a great learning experience. It allowed us to try out different technologies and apply what we've learned in a practical way. The project structure, with its milestones and goals, helped keep us on track, and having the freedom to choose our tools encouraged creativity with certain constraints, such as MQTT as a must. It felt like a semester project all over again, which we appreciate.

The course was well-organized and provided good guidance overall. Especially being able to listen to other groups and what they made was interesting and a learning experience. The teaching was also good. One of the best, if not the structured course in AU.

Working as a team on this project also made it a good opportunity to learn from each other and solve problems together. This collaboration played a big part in our success.

General Comments on the Course and Project Activities

The course was well thought out, and the project activities were practical and relevant to what we were learning. The focus on combining hardware and software made the project interesting and gave us valuable experience in building a full IoT system.

Overall, the course and project were very useful and enjoyable. They helped us build a strong foundation in IoT and gave us the skills to handle similar projects in the future.

Future work

While the current IoT monitoring system works well, there are several ways it could be improved and expanded in the future to enhance its performance, security, and user experience.

Hardware Expansion
One way to improve the system would be to add more devices and sensors. For example, adding air quality or light sensors could make the system more useful in different environments. More ESP32 microcontrollers could also be added to cover larger areas or more rooms.

Mobile App Development
Creating a mobile app would make it easier for users to check data and get notifications. A mobile app could provide real-time alerts for important events like motion detection or sudden temperature changes. It would also let users manage their devices from anywhere, offering more convenience than a web-only platform.

Website Security Improvements
To make the system more secure, adding user roles would be helpful. For example, administrators could have full control, while regular users could only view data. This would prevent unauthorized changes and keep the system safer.

Data Backup System
Currently, the system stores data only in Firebase Firestore. Adding a second backup database would improve reliability in case Firestore has issues. The Python script could be updated to send data to both databases, ensuring no data is lost if one service goes down.

These improvements would make the system more reliable, user-friendly, and secure while preparing it for future needs and expansion.

References

Arduino IDE: https://www.arduino.cc/en/software

MQTT Broker Mosquitto: https://mosquitto.org/

Firebase google database: https://console.firebase.google.com/u/0/

Firebase how to: https://firebase.google.com/docs/web/setup

Vercel: https://vercel.com/

Microsoft Azure: https://portal.azure.com/#home

EmbeddedStock: https://stock.ece.au.dk/Components

ESP32 datasheet: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf

DHT-11 datasheet: https://ardustore.dk/produkt/dht-11-temperatur-fugtigheds-module

Parallax PIR Sensor datasheet: https://docs.rs-online.com/6a22/0900766b8125a3ed.pdf

Python geocoder API: https://pypi.org/project/geocoder/

Silicone labs CP210x driver for ESP32: https://www.silabs.com/developer-tools/usb-to-uart-bridge-vcp-drivers?tab=downloads

Chatgpt: https://chatgpt.com/