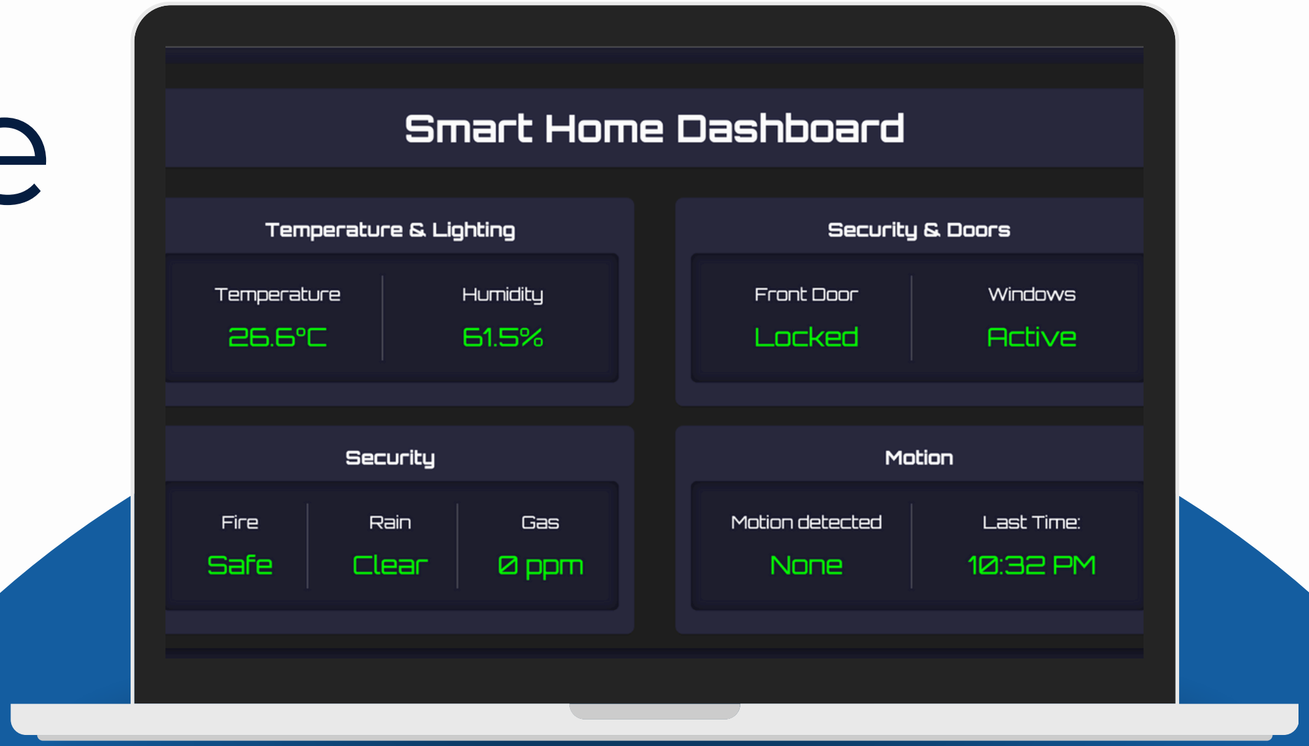
A.Y.N
Industries

Smart Home Monitoring System



Introduction

Our project is a Smart Home Monitoring and Control System that aims to provide safety, comfort, and energy efficiency.
We use various sensors (temperature, humidity, gas, flame, motion) connected to an ESP32 microcontroller to monitor real-time data.
All data is sent to a web application, where users can see sensor readings and control devices like lights, air conditioning, and doors.
Moreover, the system sends automatic alerts via Telegram Bot when an abnormal event is detected (like fire or gas leak).

About us

We are a group of 8 Information Technology students who collaborated to design and implement this Smart Home System as our graduation project.
Our team worked on both the software (web application, backend, Telegram alerts) and the hardware (ESP32, sensors, actuators).
Each member took a specific part of the project, ensuring a complete, integrated, and reliable system.

Our Values

- Innovation
- Teamwork
- Practical Application
- Data-driven Decisions

Hardware Components

Our smart home project includes key sensors like temperature, humidity, gas, flame, and motion sensors.
The ESP32 microcontroller acts as the central hub, collecting sensor data and sending it to the web interface.
This setup ensures reliable data tracking and quick alerts in case of critical conditions.

ESP32

The ESP32 serves as the brain of our system. It collects readings from all sensors, processes them, and sends data to the web server via WiFi. It supports multiple sensors simultaneously, making it perfect for real-time smart home monitoring.

Sensors

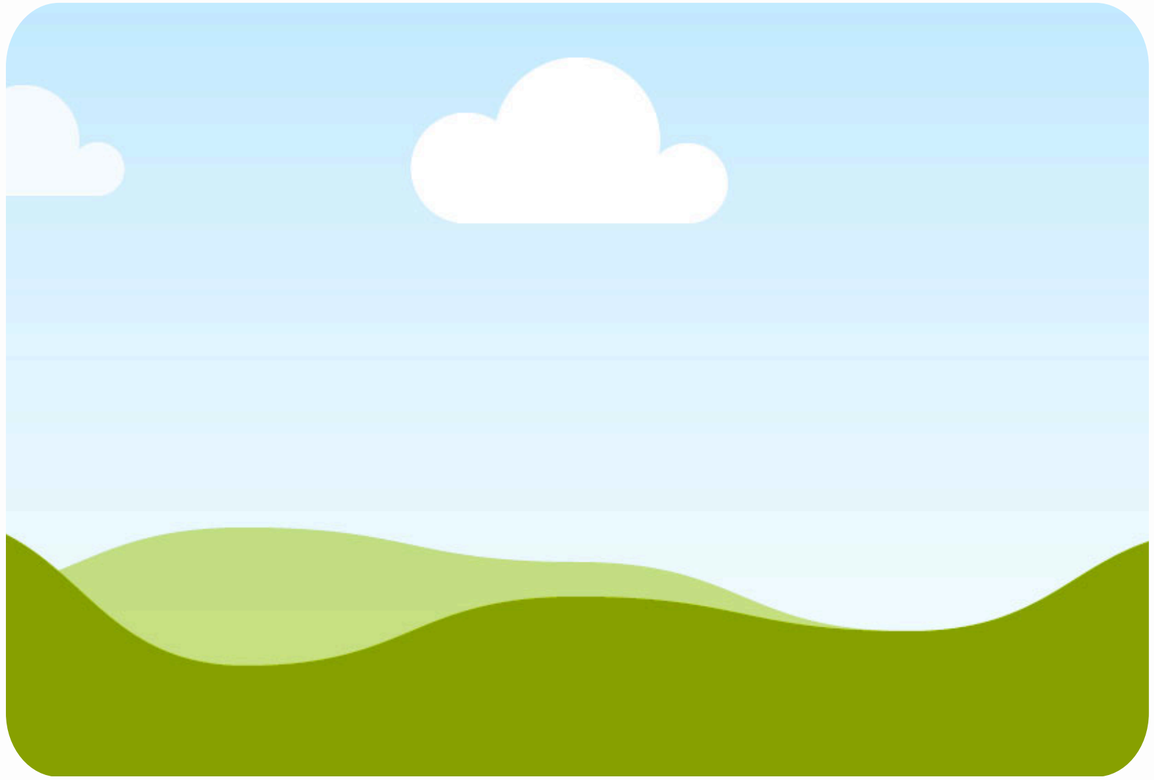
We use a set of reliable sensors:

- DHT22 for temperature & humidity.
- MQ-2 for gas leakage detection.
- Flame sensor for fire detection.
- PIR for motion tracking.

Connections

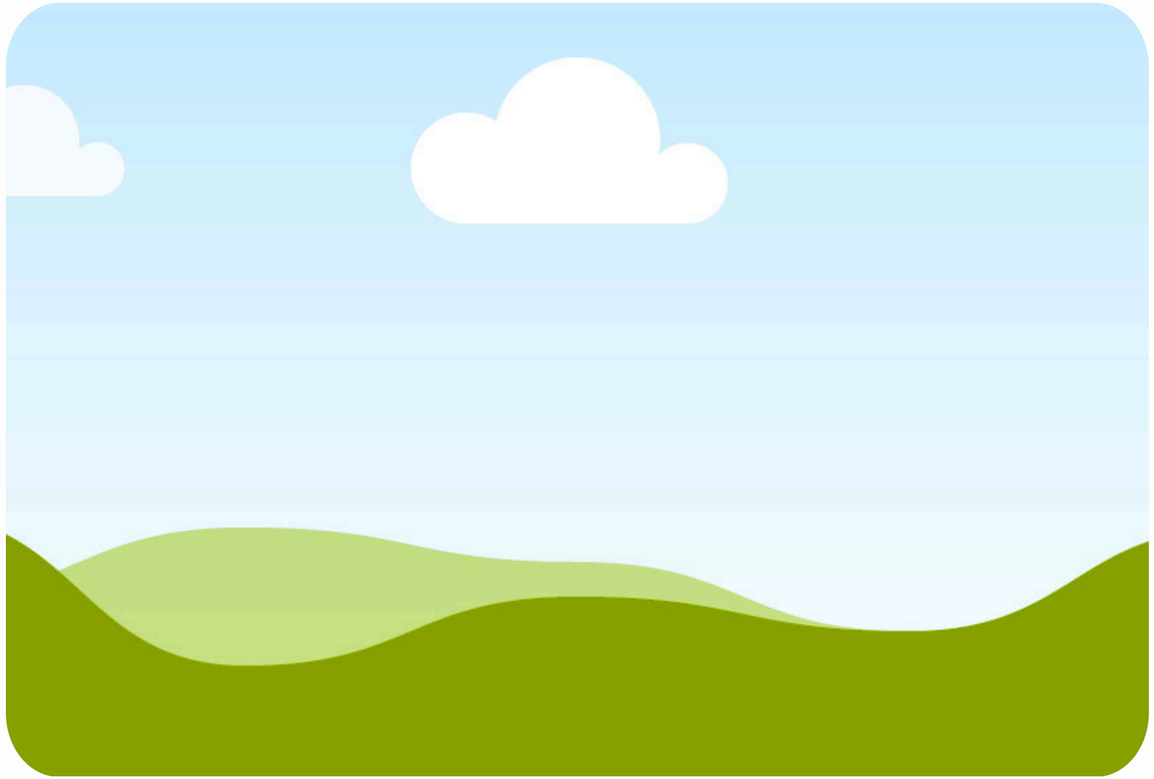
We used jumper wires and a breadboard to connect all sensors to the ESP32 securely. Proper wiring ensures that all components work together effectively, ensuring accurate readings and seamless communication with the web interface.

Hardware Overview



1 - ESP32 Microcontroller
2 - Gas Sensor
3 - Flame Sensor
4 - DHT22
5 - Reed Switch
6 - PIR
7 - LDR
8 - Breadboard & Wiring

Wiring & Connections



1 - Power To VCC & GND
2 - Sensors wired to specific GPIO pins
3 - Shared Ground For All Components
4 - Checked Connections To Avoid Short-Circuits
5 - Ready for firmware upload & testing

Initialization & Configuration

```
// WiFi and WebSocket
const char* ssid = "Adham Nemr";
const char* password = "Adham123";
WebSocketsClient webSocket;

// Sensors and Devices
#define DHTPIN 14
#define DHTTYPE DHT22
DHT dht(DHTPIN, DHTTYPE);
#define GAS_SENSOR_PIN 35
#define FLAME_SENSOR_PIN 34
#define PIR_SENSOR_PIN 33
#define LED_PIN 2
```

Initialization

Here we set up the WiFi credentials and declare all the sensors and device pins used in the project. These constants help the ESP32 recognize which pins are connected to which sensor or actuator so we can read and control them later.

sketch_jun20a.ino

```
// Connect to WiFi
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
 { delay(500); Serial.print("."); }
Serial.println("\n✅ WiFi connected!");

// Setup WebSocket
webSocket.begin("192.168.1.15", 8080, "/");
webSocket.onEvent(webSocketEvent);
```

WiFi & WebSocket

In the setup section, we establish the WiFi connection to the local network and wait until the ESP32 is successfully connected. Once connected, we initialize the WebSocket client and register its event handler so we can communicate with the server in real-time.

```
// Read sensors
float temperature = dht.readTemperature();
float humidity = dht.readHumidity();
int gasValue = analogRead(GAS_SENSOR_PIN);
int flameValue = analogRead(FLAME_SENSOR_PIN);

// Prepare JSON
StaticJsonDocument<256> doc;
doc["device_id"] = 1;
JsonObject readings = doc.createNestedObject("readings");
readings["temperature"] = temperature;
readings["humidity"] = humidity;
readings["gas"] = gasValue;
readings["flame"] = flameValue;
```

Sensor Read & Send

In the main loop, we read the current values from the sensors. These readings are placed into a JSON object and sent over WebSocket every 2 seconds. This keeps the server up to date with the real-time sensor data for further processing and alerting.

Handling Commands

Sensor Readings

```
==========================================
============== Sensor Readings ==============
🌡️ Temp: 26.50 °C | 💧 Humidity: 58.30 %
🔥 MQ-2 Gas Value: 39
==========================================
============== Sensor Readings ==============
🌡️ Temp: 26.50 °C | 💧 Humidity: 58.40 %
🔥 MQ-2 Gas Value: 788
==========================================
============== Sensor Readings ==============
🌡️ Temp: 26.50 °C | 💧 Humidity: 58.50 %
🔥 MQ-2 Gas Value: 1378
==========================================
```

Responsible for reading all sensor values — such as temperature, humidity, gas, and flame — and sending them as a JSON payload to the server at regular intervals for real-time monitoring.

Restart LED & blink

```
void restartLED() {
  for (int i = 0; i < 3; i++) {
    digitalWrite(LED_PIN, LOW);
    delay(300);
    digitalWrite(LED_PIN, HIGH);
    delay(300);
  }
  digitalWrite(LED_PIN, LOW);  // Leave LED OFF
}

void blinkTwice() {
  for (int i = 0; i < 2; i++) {
    digitalWrite(LED_PIN, LOW);
    delay(300);
    digitalWrite(LED_PIN, HIGH);
    delay(300);
  }
  digitalWrite(LED_PIN, LOW); // OFF at the end
}
```

Provides visual feedback by flashing the LED. The restartLED function blinks three times to indicate a reset, while blinkTwice blinks twice to confirm an action was triggered.

Handling Commands

handleCommand

_jun20a.ino

```ino
void handleCommand(const String& cmd, const String& action) {
  if (action == "open" || action == "true") {
    digitalWrite(LED_PIN, HIGH);
    Serial.println("💡 Command ON");
  } else if (action == "close" || action == "false") {
    digitalWrite(LED_PIN, LOW);
    Serial.println("💡 Command OFF");
  } else if (action == "restart") {
    Serial.println("🔄 Command RESTART");
    restartLED();
  } else if (action == "closeAll") {
    Serial.println("🔒 Close All - Blinking twice!");
    blinkTwice();
  }
}
```

Checks incoming commands and executes the corresponding action, like turning the LED on/off or restarting the system. It simplifies the control logic so the device responds to remote commands.

webSocketEvent

sketch_jun20a.ino

```ino
void webSocketEvent(WStype_t type, uint8_t * payload, size_t length) {
  if (type == WStype_DISCONNECTED) {
    Serial.println("❌ WebSocket Disconnected!");
  } else if (type == WStype_CONNECTED) {
    Serial.println("✅ WebSocket Connected!");
  } else if (type == WStype_TEXT) {
    Serial.printf("📩 Received: %s\n", payload);

    StaticJsonDocument<200> doc;
    if (deserializeJson(doc, payload)) {
      Serial.println("❌ Error parsing JSON");
      return;
    }

    const char* keys[] = {"light","ac","fire","rain",
    "gas","door","window","motion","last_time"};
    for (auto &key : keys) {
      if (doc.containsKey(key)) {
        handleCommand(key, doc[key].as<String>());
      }
    }
  }
}

```

Listens for incoming WebSocket messages, parses the JSON data, and passes any recognized commands to the handleCommand() function to take action in real time.

Backend API Setup

This part of the system is built using Node.js and Express to receive data from the ESP32, process it, and prepare it for secure storage and real-time alerts.

Data Processing

Each reading is checked for completeness and validity. Any unusual values — like high gas or flame — trigger instant alerts before the data is safely stored for further use.

API Endpoints

We built dedicated routes (e.g. /api/sensorData, /api/alerts) to receive incoming readings. These endpoints accept JSON payloads and return simple status messages for smooth communication.

Error Handling

We added basic validation to catch missing or incorrect data and return clear error responses. This keeps the API stable and easier to debug during testing and deployment.

Database Integration

Our database is designed to efficiently manage and store all sensor readings and event data. By using a structured schema with proper relations, we ensure data consistency, easy access, and reliable reporting for future analysis.

01

Structured Tables

Each type of data — like sensor readings and alerts — is saved in its own table to keep the database organized and easy to maintain.

02

Data Integrity

Constraints and validation rules ensure that every record meets the required format and prevents errors from being saved in the database.

03

Efficient Queries

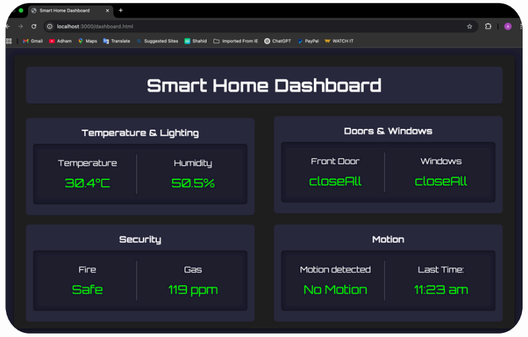
With proper indexing and relational design, we can quickly retrieve data for dashboards and reports without slowing down the system.

04

Scalability

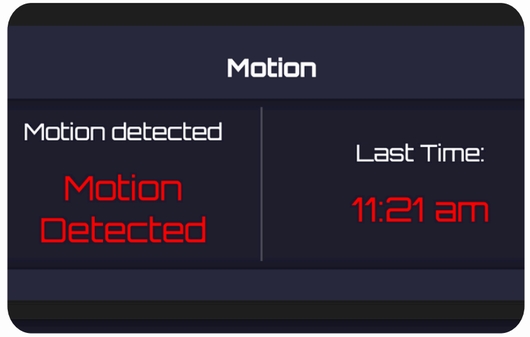
The schema is built with future growth in mind, allowing new sensors or features to be added without redesigning the entire database.

User Interface Overview



Dashboard Overview

Provides a clear, unified view of all the system's readings and device statuses. With one glance at the dashboard, you can quickly see the current temperature, gas level, flame and the status of all connected devices, making it easier to keep track of everything happening in your smart home.



Real-Time Data

Transforms raw sensor data into clear, color-coded indicators and intuitive graphics. Whether it's high gas levels or an unexpected flame detection, the dashboard highlights abnormal readings instantly, so you can take action before small problems become serious.

User Controls

Built to adapt smoothly to any screen — desktop, tablet, or phone — so you can monitor and control the smart home wherever you are. The layout stays easy to read and navigate, providing a seamless experience no matter what device you use.

Smart Home Control

Temperature Control

Here, we manage the temperature and humidity settings for the AC, as well as control lighting devices. The interface lets you easily adjust room conditions and turn lights on or off, creating a comfortable and energy-efficient environment.

Safety Controls

This section allows you to monitor gas and flame sensors and quickly respond to safety alerts. You can enable or disable sensor monitoring and receive instant warnings if any unsafe levels are detected.

Real-Time Communication

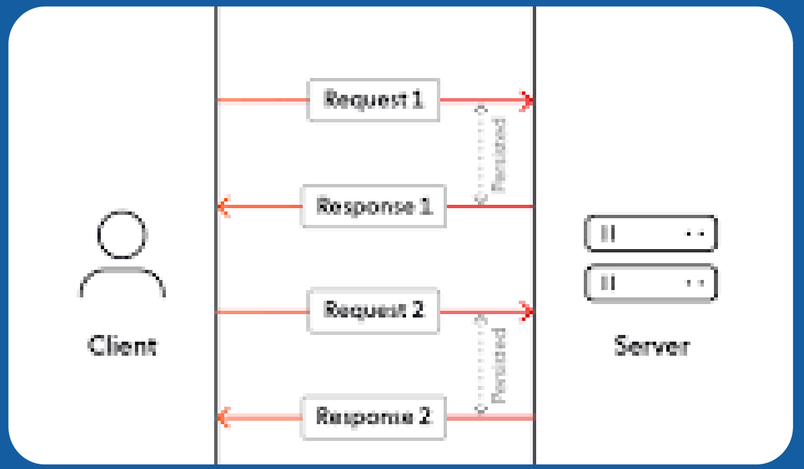
Persistent WebSocket Connection

We used a persistent WebSocket connection between the ESP32 and the backend server. This allows the system to maintain an open, continuous link — making it possible to send and receive data in real time without needing repeated HTTP requests.
Persistent WebSocket Connection

```
// WebSocket
webSocket.begin
("192.168.1.15", 8080, "/");
webSocket.onEvent
(webSocketEvent);
```

Reliable Two-Way Data Flow

The WebSocket connection enables two-way data transmission: the ESP32 sends real-time sensor readings such as temperature, gas, or flame levels, while the backend can send control commands instantly, such as turning lights on or restarting a device. This immediate interaction between devices and the server enhances the responsiveness of the system and ensures that user actions — or system alerts — are reflected without delay on the dashboard and in the environment itself.



Live Data Transmission

Real-Time Sensor Readings

The ESP32 reads sensor values (temperature, humidity, gas, flame, motion) every few seconds and sends them to the server using WebSocket. This ensures that the dashboard always displays the latest readings without needing manual refresh or delay. The data is structured as a compact JSON object that includes both sensor type and value.

```
void loop() {
  float temp = dht.readTemperature();
  float hum = dht.readHumidity();
  int gas = analogRead(GAS_SENSOR_PIN);
  int flame = analogRead(FLAME_SENSOR_PIN);

  StaticJsonDocument<256> doc;
  doc["device_id"] = 1;
  JsonObject readings = doc.createNestedObject("readings");
  readings["temperature"] = temp;
  readings["humidity"] = hum;
  readings["gas"] = gas;
  readings["flame"] = flame;

  String jsonString;
  serializeJson(doc, jsonString);
  webSocket.sendTXT(jsonString);
}
```

Efficient JSON Packaging

By packaging all readings into a single JSON message, we reduce the communication overhead and make the transmission efficient. The server receives this data instantly and updates the dashboard in real time, allowing users to monitor the system continuously and respond quickly to any abnormal conditions.

```
{
  "device_id": 1,
  "readings": {
    "temperature": 28.5,
    "humidity": 61,
    "gas": 320,
    "flame": 3550,
    "motion": 1
  }
}
```

Command Handling

Receiving Commands from Server

The ESP32 listens for incoming commands through the WebSocket connection. When a message is received, it is parsed as JSON, and specific actions are triggered — such as turning on the light or restarting a device. This ensures fast, remote control without any delay.

```
void webSocketEvent
  (WStype_t type, uint8_t * payload, size_t length) {
  if (type == WStype_TEXT) {
    StaticJsonDocument<200> doc;
    if (deserializeJson(doc, payload)) return;

    const char* keys[] =
      {"light", "ac", "gas", "flame"};
    for (auto &key : keys) {
      if (doc.containsKey(key)) {
        handleCommand(key, doc[key].as<String>());
      }
    }
  }
}
```

Connection Stability

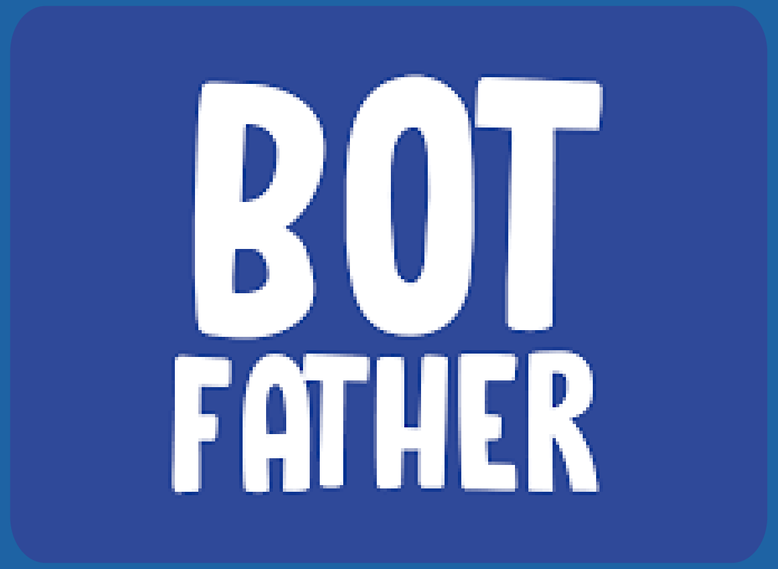
Maintaining a stable connection is critical for real-time systems. The ESP32 continuously monitors the WebSocket connection status. If the connection is lost — due to network issues or power fluctuations — the system detects the disconnection and automatically attempts to reconnect to the server. This auto-reconnect mechanism ensures that communication is restored without requiring user intervention, keeping the system reliable and responsive at all times.

```
void webSocketEvent
(WStype_t type, uint8_t * payload, size_t length) {
  switch(type) {
    case WStype_DISCONNECTED:
      Serial.println("❌ WebSocket Disconnected!");
      break;

    case WStype_CONNECTED:
      Serial.println("✅ WebSocket Connected!");
      break;
  }
}
```

Telegram Bot

Telegram Bot



We started by creating the bot using Telegram's BotFather, which provided us with a unique token. This token allows our backend system to send messages directly to the user's chat, forming the base for real-time alerts through Telegram.

Code Function

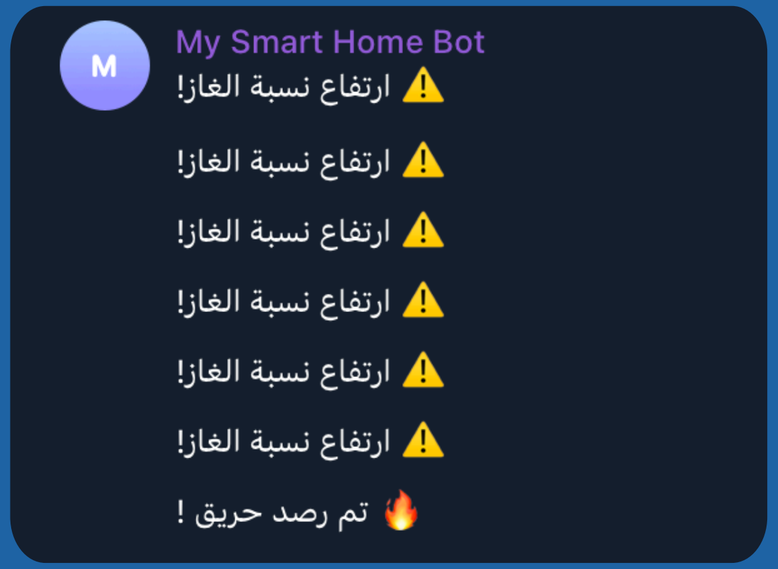
```
const sendTelegramAlert = async (message) => {
  const token =
    "7754093447:AAEvJ3oWSPMOJ8kRhHAE0A2naGEllV39tmo";
  const chatId = "1048932090";
  const url =
    `https://api.telegram.org/bot${token}/sendMessage`;

  try {
    await axios.post(url, {
      chat_id: chatId,
      text: message,
    });
    console.log("✅ Telegram alert sent.");
  } catch (err) {
    console.error("❌ Failed to send Telegram alert:",
    err.message);
  }
};
```

The bot is connected to the server using Telegram's official API. When triggered, it sends an HTTP request with the alert message. The process is fast and reliable, making sure the user receives safety notifications immediately.

From Sensor Data to Alert

Telegram Bot



These alerts appear directly on the user's phone, even if they're not using the dashboard. This ensures fast response and improves safety in emergency situations.

Code Function

```
router.post("/api/alerts", async (req, res) => {
  const { type, value } = req.body;
  if (!type || value === undefined) {
    return res.status(400).json({ error: "Invalid alert payload" });
  }
  try {
    let message = "";
    if (type === "gas" && value > 400) {
      message = `⚠️ ارتفاع نسبة الغاز!`;
    } else if (type === "fire" && value < 3000) {
      message = `🔥 تم رصد حريق !`;
    } else {
      return res.status(200).json({ message: "No alert needed." });
    }
    await sendTelegramAlert(message);
    res.status(200).json({ message: "Alert sent successfully." });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

The server receives data from ESP32 and checks for danger values like high gas levels or fire detection. If any value crosses the safety threshold, an alert message is sent instantly via Telegram.

Database Responsibilities

In this project, the database developer played a key role in building the system's data backbone. From designing the relational schema to storing and organizing sensor readings, their work ensured that all data flows smoothly and can be accessed reliably. The structure they built supports everything from live dashboard updates to historical analysis and alert generation.

01

Database Design

Created the database schema using MySQL Workbench with structured tables for readings and alerts.

02

Data Storage Implementation

Handled storing real-time sensor data with timestamps and device IDs into the database.

03

Querying and Data Retrieval

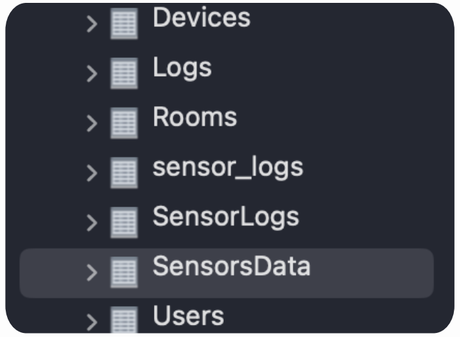
Built SQL queries to filter critical values and support dashboard and alert functions.

04

Supporting Analytics

Ensured the data is organized and ready for reports, trends, and future system expansion.

Database Management



Structured Schema

We created a relational database structure , with separate tables for sensor data, alerts, and devices. Each table is connected through foreign keys, allowing for organized storage and easy linking between readings and their sources. This schema helps keep the data consistent and scalable.

id	device_id	data_type	value
432	1	humidity	57.8
433	1	gas	169
434	1	flame	4095
435	1	motion	0
436	1	temperature	26.5
437	1	humidity	57.8
438	1	gas	161
439	1	flame	4095
440	1	motion	0
441	1	temperature	26.5
442	1	humidity	57.9

Real-Time Storage

As soon as the ESP32 sends sensor readings, the backend stores them directly into the database. Each record includes the reading type, the sensor value, the time it was received, and the device ID. This setup ensures that every reading is logged in real-time and can be retrieved whenever needed.

```
adham@adhams-MacBook-Pro ~ % SELECT
  device_id,
  data_type,
  value,
  timestamp
FROM
  sensor_data
WHERE(data_type = 'gas' AND value > 400)
OR(data_type = 'flame' AND value < 3000)
ORDER BY
  timestamp DESC
LIMIT 10;
```

Data Structure

Having a well-designed database is critical for system performance and future expansion. It allows us to retrieve and analyze historical data efficiently, generate useful reports, and feed accurate information to the dashboard for live monitoring and decision-making.

SYSTEM INTEGRATION & TESTING

01 Connecting Project

This step involved linking all parts of the project into one complete system. The frontend interface, backend server, ESP32 controller, and MySQL database were integrated to work together seamlessly and in real time.

02 Hardware-Software Sync

he integration ensured accurate and consistent communication between the hardware sensors and the software interface. Every reading from the ESP32 appeared instantly on the dashboard, reflecting true live monitoring.

03 Final Testing & Debugging

The system was tested thoroughly under real usage scenarios. All features were checked — from sensor alerts to device control — and bugs were fixed to guarantee stable, reliable performance.

04 Overall System Validation

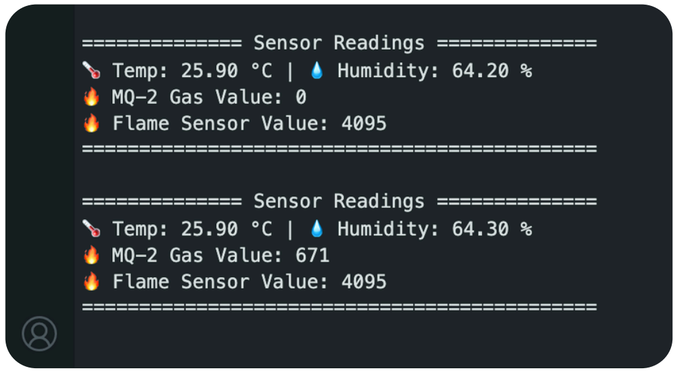
After integration and testing, the system was reviewed as a whole to ensure all components function correctly together. The final setup was validated in real conditions and met the project goals successfully.

INTEGRATION & FINAL VALIDATIN

This slide presents screenshots from the system during the final testing phase. Each component was verified in real conditions to ensure full functionality, reliability, and real-time response.

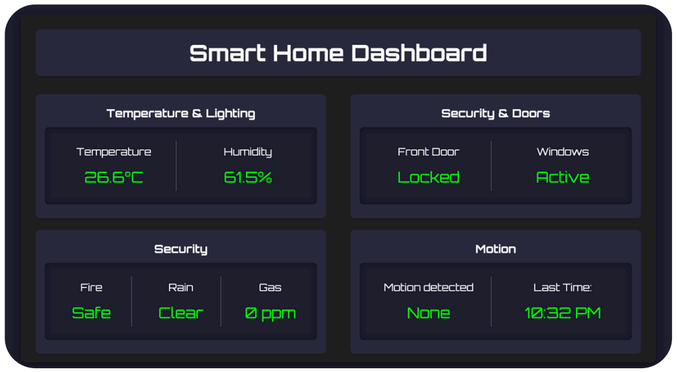
WebSocket Connection

We created a relational database structure , with separate tables for sensor data, alerts, and devices. Each table is connected through foreign keys, allowing for organized storage and easy linking between readings and their sources. This schema helps keep the data consistent and scalable.



Real-Time Sensor Readings

As soon as the ESP32 sends sensor readings, the backend
stores them directly into the database. Each record includes the reading type, the sensor value, the time it was received, and the device ID. This setup ensures that every reading is logged in real-time and can be retrieved whenever needed.



Dashboard Interaction

We created a relational database structure , with separate tables for sensor data, alerts, and devices. Each table is connected through foreign keys, allowing for organized storage and easy linking between readings and their sources. This schema helps keep the data consistent and scalable.

Our Team

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

Aaron Loeb
CEO & Founder

A.Y.N
Industries

THANK YOU!

Thank you for your attention and support.

We appreciate the opportunity to present our smart home project and look forward to your valuable feedback.