# Smart Campus

## An IoT Sensor Network for Classroom Monitoring

Design and Implementation Report

IEEE Student Branch, University of Western Macedonia, Kastoria
Smart Campus Team

# 1. Introduction

Smart Campus is an Internet of Things (IoT) system that monitors the environment and usage of university classrooms in real time. Each classroom is equipped with a small sensor node that measures temperature, humidity, light level, door state and human presence. The readings are sent over the campus Wi-Fi network to a central server, where they are processed and presented to facility administrators through a web dashboard.

The project was motivated by a set of everyday problems that are common across university buildings: lights and air conditioning left running in empty rooms, doors left open at the end of the day, and lecture halls that become uncomfortably warm or humid during busy periods. These problems waste energy and affect the comfort and safety of students and staff, yet they are hard to notice without someone physically walking through every building. Smart Campus makes this information available remotely and continuously.

This document describes the design decisions behind the system and how each part was implemented, from the sensor hardware and embedded firmware to the messaging layer, the data processing backend and the user interface.

# 2. Objectives and Requirements

The main objectives of the project are:

- Improve energy efficiency by revealing rooms where lights are on while nobody is present.
- Improve comfort by tracking temperature, humidity and the perceived (felt) temperature in each room.
- Improve safety and security by reporting doors that remain open.
- Provide occupancy information so that facility staff can see which rooms are in use.
- Offer a real-time, centralized view of all monitored classrooms.
- Build an open-source and scalable framework that other academic institutions can adopt.

From these objectives the following technical requirements were derived:

- **Low cost per room.** Nodes must use inexpensive, widely available components so that many rooms can be equipped.
- **Wireless operation.** Nodes must use the existing Wi-Fi infrastructure rather than dedicated cabling.
- **Lightweight messaging.** A publish and subscribe protocol must decouple the sensor nodes from the software that consumes their data.
- **Robustness.** A failing sensor or a temporary network outage must not stop a node from operating, and the node must recover automatically.
- **Configurability.** Pin assignments, network settings, timing and topic names must be easy to change per deployment without touching the application logic.

- **Simple deployment.** The server side must be reproducible on any machine with a single command.

# 3. System Architecture

The system follows a layered architecture in which every layer communicates with the next through a well defined protocol. This separation keeps each part simple and allows components to be replaced independently.

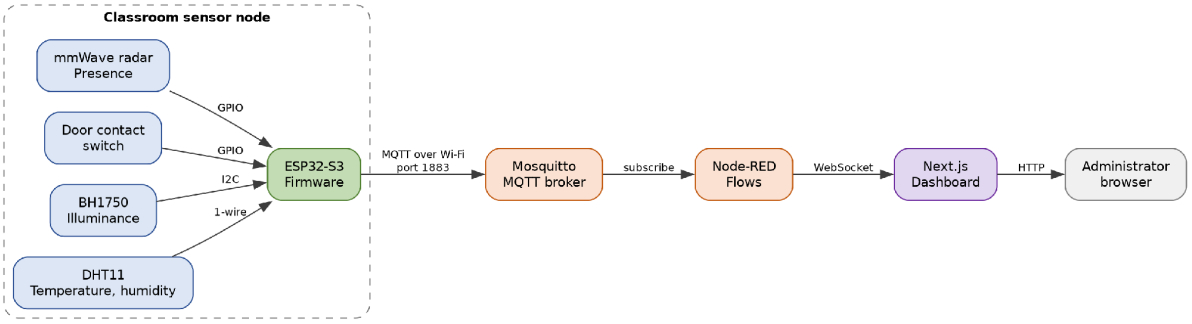



*Figure 1. High level architecture and data flow of the Smart Campus system*

The four layers are:

- **Sensing layer.** One ESP32-S3 node per classroom reads its attached sensors at a fixed interval.
- **Transport layer.** Nodes publish each reading as an MQTT message to a central Eclipse Mosquitto broker over Wi-Fi.
- **Processing layer.** Node-RED subscribes to the broker, associates incoming messages with the classroom they came from, and forwards the combined classroom state to clients over a WebSocket connection.
- **Presentation layer.** A web dashboard built with Next.js receives the live data and displays it to administrators, with one view per classroom.

MQTT was chosen as the transport protocol because it was designed for constrained devices and unreliable networks. Its publish and subscribe model means that sensor nodes do not need to know who consumes their data. New consumers, such as a logging database or an alerting service, can be added later simply by subscribing to the same topics, without any change to the firmware.

# 4. Hardware Design

## 4.1 Microcontroller

Each node is built around an Espressif ESP32-S3 development board (ESP32-S3-DevKitC-1). The ESP32-S3 provides a dual core processor, integrated 2.4 GHz Wi-Fi, a hardware I2C controller and plenty of general purpose input and output pins, which makes it possible to connect all sensors directly without any additional networking hardware.

The board is powered over USB, so a node can be installed next to any wall socket in a classroom.

### 4.2 Sensors

Four sensors are attached to every node. They were selected for low cost, low power consumption and simple interfacing with the ESP32.

| Sensor | Measured quantity | Interface | ESP32 pin |
|---|---|---|---|
| DHT11 | Temperature and relative humidity | Single wire digital | GPIO 4 |
| BH1750 | Ambient light (illuminance in lux) | I2C, address 0x23 | SDA GPIO 12, SCL GPIO 11 |
| Magnetic door contact | Door open or closed | Digital input with internal pull-up | GPIO 13 |
| 24 GHz mmWave radar | Human presence and motion | Digital output | Dedicated GPIO |

*Table 1. Sensors used in each classroom node*

**DHT11.** A low cost combined temperature and humidity sensor that communicates over a single data line. It offers an accuracy of about 2 degrees Celsius and 5 percent relative humidity, which is sufficient for judging classroom comfort.

**BH1750.** A digital ambient light sensor from ROHM with a 16 bit output, connected over the I2C bus. Unlike a simple photoresistor it reports calibrated values directly in lux, which makes it possible to tell reliably whether the room lights are switched on.

**Door contact.** A reed switch mounted on the door frame with a magnet on the door. The ESP32 internal pull-up resistor is enabled on the input pin, so no external resistor is needed and the pin reads a stable level for each door position.

**mmWave radar.** A 24 GHz millimeter wave radar presence sensor. Traditional passive infrared (PIR) sensors only react to movement, so a class of students sitting quietly during an exam can appear to be an empty room. A mmWave radar detects the very small movements caused by breathing and posture changes, which allows it to report presence even when people are sitting still. This makes it much better suited to occupancy detection in classrooms. The sensor exposes a digital output that goes high whenever a person is detected, which the ESP32 reads on a dedicated GPIO pin.

## 5. Firmware Design and Implementation

### 5.1 Tooling and Libraries

The firmware is written in C++ on top of the Arduino framework for the ESP32 and is built with PlatformIO. PlatformIO manages the toolchain, board definition and library dependencies declaratively through the platformio.ini file, so every developer builds the firmware with exactly the same versions. The main external libraries are:

- **PubSubClient** for the MQTT client.
- **Adafruit DHT sensor library** and **Adafruit Unified Sensor** for communication with the DHT11.
- The built-in **WiFi** and **Wire** (I2C) libraries of the ESP32 Arduino core.

## 5.2 Code Organization

The source code is split into small modules, each with a header in the include directory and an implementation in the src directory. Every module is responsible for exactly one sensor, while the main program only coordinates them. This keeps the drivers independent of the networking code, makes them easy to test in isolation, and allows a sensor to be replaced without affecting the rest of the system.

| File | Responsibility |
|---|---|
| config.h | Central configuration: GPIO pins, MQTT broker address and port, timing intervals and MQTT topic names |
| secrets.h | Wi-Fi network name and password, kept out of version control |
| dht11.hpp / dht11.cpp | DHT11Interface class for temperature, humidity and heat index |
| bh1750.hpp / bh1750.cpp | I2C driver for the BH1750 light sensor |
| DoorSensor.hpp / DoorSensor.cpp | Initialization and reading of the door contact |
| Motion sensor module | Initialization and reading of the mmWave radar presence output |
| main.cpp | Wi-Fi and MQTT connection handling, the main loop and publishing of readings |

*Table 2. Firmware modules*

## 5.3 Configuration and Credentials

All per deployment settings are collected in a single header, config.h. It defines the pin assignments, the broker address and port, the publishing interval, the MQTT reconnection delay and the names of all MQTT topics. Installing a node in a new room or with different wiring therefore only requires editing this file.

Wi-Fi credentials are deliberately kept in a separate file, secrets.h, which is listed in .gitignore so that passwords are never committed to the public repository. A template named secrets.example.h is provided instead. The main program uses the __has_include preprocessor check to detect whether secrets.h exists, and if it is missing the build stops with a clear error message explaining how to create it. This turns a common configuration mistake into an immediate, self explanatory compile time error rather than a node that silently fails to connect.

## 5.4 Sensor Drivers

**DHT11 interface.** The DHT11 is wrapped in a DHT11Interface class that hides the underlying Adafruit library. A call to read() fetches the temperature and humidity, verifies that neither value is invalid (the library returns NaN on a timeout or checksum error), computes

the heat index and records a timestamp. Getter methods then expose the temperature, humidity and heat index, while isValid() reports whether the last read succeeded. The heat index is the apparent, or felt, temperature, which combines temperature and humidity into a value that better reflects human comfort.

**BH1750 driver.** The light sensor driver is written directly against the I2C bus rather than using a third party library, which keeps it small and fully understood by the team. initBH1750() starts the I2C bus on the configured pins and sends the Continuous High Resolution mode command (0x10), in which the sensor measures continuously with a resolution of 1 lux. getSensorData() requests two bytes from the sensor, and computeLx() combines them into a 16 bit value and divides it by 1.2, the conversion factor given in the ROHM datasheet, to obtain lux. computeLx() reports failure if fewer than two bytes are received, so that a disconnected sensor never produces a misleading value.

**Door sensor.** initDoor() configures the door pin as an input with the internal pull-up resistor enabled, and readDoor() returns the current state of the pin, where a high level means the door is open and a low level means it is closed.

**Motion sensor.** The mmWave radar module follows the same pattern as the door sensor. An initialization function configures its GPIO pin as a digital input during start up, and a read function returns whether presence is currently detected. Because the presence decision is made inside the radar module itself, the firmware only needs to sample a single digital level, which keeps the main loop fast and simple.

## 5.5 Main Program Flow

The setup() function runs once at power on. It starts the serial console for diagnostics, connects to the Wi-Fi network, configures the MQTT client with the broker address and initializes all sensors.

The loop() function then runs continuously and performs the following steps:

1. If the MQTT connection has been lost, reconnect to the broker, retrying every 5 seconds until it succeeds. After every successful connection the node publishes a status message announcing that it is online.
2. Call the MQTT client loop so that the connection is kept alive.
3. Check whether the publishing interval (5 seconds by default) has passed since the last publication. This check uses the millis() timer instead of a blocking delay, so the node remains responsive to the network between measurements.
4. When the interval has passed, read every sensor and publish each value to its own MQTT topic.
5. Wait 10 milliseconds to give the background tasks of the ESP32, such as the Wi-Fi stack and the watchdog timer, time to run.

## 5.6 Error Handling and Recovery

Because nodes are installed in classrooms and are expected to run unattended for long periods, the firmware is designed to tolerate faults and recover from them automatically:

- **Independent sensors.** Each sensor is read and published separately. If one sensor fails, the values of the others are still published.
- **No invalid data.** A failed reading is never published. For the DHT11 the values are only sent when read() succeeds, and for the BH1750 only when computeLx() returns a complete measurement. Failures are reported on the serial console instead.
- **Light sensor recovery.** The node keeps track of whether the BH1750 is ready. If the sensor was not available at start up, or if a reading fails later, the driver is re-initialized on the next cycle, so the sensor recovers on its own after a loose connection is fixed.
- **Network recovery.** A lost MQTT connection is detected at the start of every loop and re-established automatically.
- **Publish verification.** All messages are sent through a helper function, publishWithCheck(), which logs the topic of any message that could not be delivered.

# 6. Communication Layer

## 6.1 MQTT Topics and Payloads

Each measured quantity is published to its own topic as a plain text value. Using one topic per quantity keeps the payloads tiny, allows consumers to subscribe only to what they need, and makes the messages easy to inspect with standard MQTT tools during development.

| Topic | Content | Payload example |
|---|---|---|
| esp32/status | Node connection announcement | ESP32 Connected |
| Temperature | Air temperature in degrees Celsius | 23.40 |
| FeltTemperature | Heat index (felt temperature) in degrees Celsius | 23.85 |
| Humidity | Relative humidity in percent | 47.00 |
| Lx | Illuminance in lux | 412 |
| Door | Door state | Open or Closed |
| Motion | Presence detected by the mmWave radar | Detected or Clear |

*Table 3. MQTT topics published by each node*

Numeric values from the DHT11 are formatted with two decimal places, while the illuminance is sent as an integer, since the BH1750 resolution in the selected mode is 1 lux.

## 6.2 MQTT Broker

The central broker is Eclipse Mosquitto, a lightweight and widely used open-source MQTT broker. It listens on the standard MQTT port 1883 on all network interfaces. Message persistence is enabled and stored in a dedicated data volume, so retained state survives a restart of the broker. Errors, warnings, notices and informational messages are written to a log file in a separate volume, which helps with diagnosing connection problems with individual nodes.

# 7. Data Processing with Node-RED

Node-RED is a flow based programming tool that is well suited to IoT integration work. Processing logic is built visually by wiring together nodes in the browser based editor, which is reachable on port 1880. This lowers the barrier for team members who are not full time software developers and makes the data flow easy to understand and modify.

In Smart Campus, the Node-RED flows perform three tasks:

- **Collection.** MQTT input nodes subscribe to the topics published by the classroom nodes.
- **Aggregation.** Function nodes convert the individual text payloads into typed values and combine them into a single state object per classroom containing the temperature, felt temperature, humidity, illuminance, door state and presence.
- **Distribution.** A WebSocket output node pushes the updated classroom state to every connected dashboard as soon as a new value arrives, so the user interface never has to poll the server.

Placing this logic in Node-RED rather than in the firmware or the dashboard keeps both of those components simple. Additional behaviour, such as storing a long term history in a database or sending an alert when a door is left open after hours, can be added as new flows without changing any other part of the system.

# 8. Web Dashboard

## 8.1 Technology Stack

The dashboard is a single page web application built with Next.js 16 and React 19, written in TypeScript. Styling is done with Tailwind CSS 4, charts are drawn with Recharts, icons come from the Lucide icon set and accessible tab components are provided by Base UI. The address of the Node-RED WebSocket endpoint is supplied through an environment variable, NODERED_WS_URL, so the same build can be pointed at different servers without code changes.

## 8.2 Layout and Navigation

The page is organized around the classrooms being monitored. A sticky header shows the project name and a System Online indicator. Below it, a row of tabs lists the configured classrooms, for example B1, B11, Lab A3 and the main lecture hall. Each classroom is described by a small configuration record containing its identifier, display name, building and floor, so adding a new room to the dashboard only requires adding one entry to this list.

Selecting a tab opens the view for that classroom. The view shows the room name, its building and floor, the current time and a pulsing Live data indicator, followed by a responsive grid of widgets. The grid shows four columns on large screens, two on tablets and a single column on phones, so the dashboard can also be used from a mobile device while walking around campus. The interface uses a dark theme that remains readable on wall mounted displays.

### 8.3 Widgets

Every sensor quantity is represented by a dedicated widget. Besides the raw value, each widget translates the reading into a short, human readable status and a colour, so that an administrator can understand the state of a room at a glance.

- **Temperature.** Shows the current temperature and the felt temperature, together with a line chart of the recent history of both. The actual temperature is drawn as a solid line and the felt temperature as a dashed line, making it easy to see when humidity makes a room feel warmer than the thermometer suggests.
- **Door status.** Shows a large Open or Closed label with a matching door icon. An open door is highlighted in amber and a closed door in green.
- **Occupancy.** Shows whether the mmWave radar currently detects people in the room, displayed as Occupied or Vacant with a distinct colour for each state.
- **Humidity.** Shows the relative humidity on a 0 to 100 percent bar divided into Dry, Comfortable and Humid zones. Values below 30 percent are labelled Low, values above 60 percent High, and values in between Optimal.
- **Luminosity.** Shows the illuminance in lux with a progress bar and a soft glow whose intensity follows the light level. Readings are classified as Dim (below 100 lux), Moderate (100 to 300 lux), Bright (300 to 600 lux) or Very Bright (600 lux and above).

### 8.4 Data Handling

Each classroom view obtains its data through a custom React hook, useClassroomData, which takes the classroom identifier and returns the latest state of that room. The hook keeps the current sensor values together with a rolling history of the last 30 temperature points used by the chart. When a new message arrives, the newest point is appended and the oldest is dropped, so memory use stays constant no matter how long the dashboard remains open. All widgets are pure presentation components that receive their values as properties, which keeps the data handling in one place and makes the widgets easy to reuse.

## 9. Deployment

The server side of the system is containerized with Docker. A single docker-compose.yml file defines two services:

- **mosquitto**, based on the official eclipse-mosquitto image, exposing port 1883 for MQTT and port 9001 for WebSocket connections. Its configuration file is mounted from the repository, and its data and logs are stored in named volumes.
- **node-red**, based on the official nodered/node-red image, exposing port 1880 and storing its flows and settings in a named volume. The container time zone is set to Europe/Athens so that timestamps match local time on campus.

Both services use the restart always policy, so they start automatically when the server boots and are restarted if they ever stop unexpectedly. The whole backend can therefore be brought up on any machine with Docker installed by running docker compose up.

To deploy a sensor node, the installer copies secrets.example.h to secrets.h and enters the Wi-Fi credentials, adjusts config.h if the wiring or broker address differ, and flashes the firmware over USB with the PlatformIO upload command. The node connects to the network and starts publishing immediately after power on. The dashboard is a standard Next.js application and can be run on the same server as the backend or hosted separately.

## 10. Development Practices

- **Version control.** The project is hosted on GitHub. New features, such as each sensor driver, are developed on separate feature branches and merged through pull requests.
- **Contribution templates.** The repository provides templates for bug reports, feature requests and pull requests, which keep contributions consistent and easy to review.
- **Code style.** C++ code is formatted automatically with clang-format using the LLVM style, so the code base has a uniform appearance regardless of who wrote it.
- **Documentation.** Every public function and class in the firmware is documented with Doxygen comments describing its purpose, parameters, return values, preconditions and limitations. Doxygen generates browsable HTML documentation from these comments, which is kept in the docs directory.
- **Licensing.** The project is released under the MIT License, allowing other universities and students to reuse and extend it freely.

## 11. Use Cases

Combining the readings from several sensors in the same room gives administrators information that no single sensor could provide on its own:

- **Forgotten lights.** A high illuminance reading while the radar reports no presence, especially in the evening, indicates that the lights were left on in an empty room.
- **Open doors.** A door reported as open outside teaching hours, or in a room with no presence, points to a room that was not secured.
- **Comfort monitoring.** Rising temperature, felt temperature and humidity in an occupied lecture hall show when ventilation or air conditioning should be adjusted.
- **Room usage.** Presence data shows which rooms are actually in use, which can inform room booking and cleaning schedules.

## 12. Conclusion

Smart Campus demonstrates how inexpensive, off the shelf components and open-source software can be combined into a practical monitoring system for university classrooms. ESP32-S3 nodes measure temperature, humidity, light, door state and human presence, MQTT and Mosquitto provide reliable and decoupled message delivery, Node-RED aggregates the data, and a modern web dashboard presents it in a form that administrators can understand at a glance. The modular firmware, central configuration, automatic fault recovery and containerized backend make the system straightforward to deploy, maintain

and extend, providing a solid foundation for improving energy efficiency, comfort and safety across the campus.