Build your own IoT Weather Node

By the Systems Community

Let’s begin the workshop by first interfacing the **LED.** The ESP32 is a microcontroller with multiple GPIO (General Purpose Input/Output) pins. These pins can be programmed to either read signals (input) or send signals (output). To control an LED, we need to configure a GPIO pin as an output and then control whether it sends power or not.

When we set a pin to HIGH, it outputs 3.3V, which provides power to the LED and makes it light up. When we set it to LOW, it outputs 0V, cutting off power and turning the LED off. By alternating between HIGH and LOW with delays in between, we create a blinking effect.

In C/C++, we use **#define** to create constants. This is a **preprocessor directive** that replaces every occurrence of the name with the value before compilation. It's a way to make code more readable - instead of seeing the number 2 everywhere, we see **LED_PIN**, which tells us exactly what that pin is used for.

The Arduino program structure has two main functions:

- **setup()** runs once when the board powers on or resets. We use it to configure pins.
- **loop()** runs repeatedly forever. We use it to create the blinking behavior.

We use **pinMode()** to tell the ESP32 that a specific pin should act as an output. Then **digitalWrite()** sets that pin to either **HIGH** or **LOW**. The **delay()** function pauses the program for a specified number of milliseconds.

```
#define LED_PIN 5

void setup() {
  pinMode(LED_PIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_PIN, HIGH);
  delay(1000);
  digitalWrite(LED_PIN, LOW);
  delay(1000);
}
```

The functions we're using here - pinMode(), digitalWrite(), and delay() - are part of the Arduino framework. The Arduino framework is a set of pre-written functions that make it easier to program microcontrollers. Without it, we would have to directly manipulate hardware registers, which is much more complex. The framework handles all that complexity for us and gives us simple, readable functions to work with.
Let's break down what the code does:
#define LED_PIN 2 creates a constant called LED_PIN with the value 2. Every time the compiler sees LED_PIN in the code, it replaces it with 2. Pin 2 is typically the built-in LED on most ESP32 development boards.
In setup(), we call pinMode(LED_PIN, OUTPUT). This configures GPIO pin 2 to act as an output pin, meaning it will send signals out rather than read them in.

In loop(), we repeatedly:

- Set the pin to **HIGH** with **digitalWrite(LED_PIN, HIGH)**, turning the LED on
- Wait for **1000 milliseconds (1 second)** with **delay(1000)**
- Set the pin to **LOW** with **digitalWrite(LED_PIN, LOW)**, turning the LED off
- Wait another 1000 milliseconds
- This creates a 1-second on, 1-second off blinking pattern that repeats forever.

Introducing Interrupts:

While **delay()** works for simple blinking, it has a major limitation - it completely blocks the program. During those 1000 milliseconds, the ESP32 cannot do anything else. It just sits there waiting. In a real application where we need to read sensors, respond to network messages, or handle multiple tasks, this blocking behavior becomes a problem.
This is where interrupts become important. An interrupt is a signal that temporarily stops the normal program flow to handle something urgent. Think of it like getting a phone call while working - you pause what you're doing, handle the call, then resume your work.

Hardware interrupts are triggered by external events, like a button press or a timer reaching zero. When an interrupt occurs, the microcontroller immediately stops what it's doing and runs a special function called an **Interrupt Service Routine (ISR)**. After the ISR completes, the program resumes exactly where it left off.

For our purposes, we'll use timer interrupts. A timer interrupt fires at regular intervals without blocking the main program. This lets us blink an LED while simultaneously reading sensors, handling network communication, and running other code. The timer runs in the background, and when it triggers, our **ISR** quickly toggles the LED state, then the main program continues running.

```
#define LED_PIN 5

hw_timer_t *timer = NULL;
volatile bool ledState = false;

void IRAM_ATTR onTimer() {
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState);
}

void setup() {
  pinMode(LED_PIN, OUTPUT);

  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &onTimer);
  timerAlarm(timer, 1000000, true, 0);
}

void loop() {
  // Main loop is free to do other things
}
```

Global Variables:

hw_timer_t *timer is a pointer to a hardware timer. The ESP32 has built-in hardware timers that count independently of the main program.

volatile bool ledState tracks the LED state. The volatile keyword tells the compiler this variable can be changed by an interrupt, preventing incorrect optimizations.

The Interrupt Service Routine (ISR):

IRAM_ATTR places the function in **RAM** for faster execution. ISRs need to run quickly and return control to the main program.

The function toggles ledState and updates the LED pin. This runs automatically every time the timer fires.

Setup Configuration:

timerBegin(1000000) initializes the timer with a 1MHz counting frequency.
timerAttachInterrupt(timer, &onTimer) connects our ISR function to the timer interrupt.
timerAlarm(timer, 1000000, true, 0) sets the timer to fire every **1000000 microseconds (1 second)**, with auto-reload enabled so it repeats continuously.

The LED now blinks automatically in the background while the **loop()** remains free to execute other code without any blocking delays. Interrupts allow the microcontroller to respond to events immediately without constantly checking for them. Instead of the program waiting or blocking with delays, the hardware timer counts in the background and triggers our ISR when it reaches the specified interval. This makes the system more efficient and responsive - the LED blinks on its own while the main program can handle sensors, network communication, or any other tasks simultaneously.

Now that we have seen interrupts, gone through the rite of passage of a blink sketch, got introduced to interrupts, and re-wrote this in such a way that is efficient by using the ISR instead of delay making us free to do whatever we want in loop, We will now learn about the DHT-22 sensor, how it works and integrating it into our code.

The DHT22 is a digital temperature and humidity sensor. Unlike analog sensors that output varying voltages, the DHT22 communicates digitally using a single data wire. It measures temperature from -40°C to 80°C with ±0.5°C accuracy and relative humidity from 0% to 100% with ±2% accuracy.

The sensor uses a single-wire protocol. When we request a reading, the DHT22 sends back 40 bits of data containing humidity, temperature, and a checksum for error detection. This communication happens very quickly, but it requires precise timing. Writing code to handle this timing would be complex, and we don't really have the time to do that in this workshop, so we use a library that does all the heavy lifting for us.

The Adafruit DHT library provides simple functions to read from DHT sensors. We create a DHT object by specifying which pin the sensor is connected to and which type of DHT sensor we're using (DHT11, DHT21, or DHT22). The library handles all the timing and bit manipulation internally.

Temperature and humidity sensors need a moment to stabilize after taking a reading. The DHT22 can only be read once every 2 seconds. If we try to read faster, we'll get the same values or errors. For our application, we'll read the sensor periodically and print the values to the Serial Monitor so we can verify everything is working.

```
#include <DHT.h>

#define LED_PIN 5
#define DHT_PIN 4
#define DHT_TYPE DHT22

DHT dht(DHT_PIN, DHT_TYPE);

hw_timer_t *timer = NULL;
```

```
volatile bool ledState = false;

void IRAM_ATTR onTimer() {
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState);
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);

  dht.begin();

  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &onTimer);
  timerAlarm(timer, 1000000, true, 0);
}

void loop() {
  float humidity = dht.readHumidity();
  float temperature = dht.readTemperature();

  if (isnan(humidity) || isnan(temperature)) {
    Serial.println("Failed to read from DHT sensor!");
  } else {
    Serial.print("Humidity: ");
    Serial.print(humidity);
    Serial.print("%  Temperature: ");
    Serial.print(temperature);
    Serial.println("°C");
  }

  delay(2000);
}
```

We start by including the DHT library and defining our pin configurations. We create a DHT object called dht by passing it the pin number and sensor type. In **setup()**, we initialize **Serial communication at 115200 baud** so we can see output in the Serial Monitor, and we call **dht.begin()** to initialize the sensor.
In **loop()**, we read **humidity** and **temperature** using dht.readHumidity() and dht.readTemperature(). These functions return floating-point numbers. If the sensor fails to read (maybe it's disconnected or not responding), the functions return **NaN (Not a Number)**. We check for this using **isnan()** and print an error message if something went wrong. Otherwise, we print the humidity and temperature values in a readable format.
The **delay(2000)** at the end pauses for 2 seconds between readings, respecting the DHT22's minimum sampling period. **Notice that the LED continues blinking during this delay because the timer interrupt handles it independently.** The main loop can block on the delay without affecting the LED, demonstrating the advantage of using interrupts for concurrent tasks.

The **OLED (Organic Light Emitting Diode) display** we're using is a **128x64 pixel monochrome screen** that communicates via the **I2C protocol**. Recall from our earlier discussion that **I2C is a two-wire communication protocol that uses a clock line (SCL) and a data line (SDA) to transfer information between the microcontroller and peripheral devices.** Multiple devices can share the same I2C bus, each identified by a unique address. Our OLED display typically uses address **0x3C**. We can also confirm this by running the I2C Scanner example present in the Examples directory of the repository.

The advantage of I2C is that it only requires two pins, regardless of how many devices you connect. The ESP32 has dedicated I2C pins, and the Wire library handles all the low-level communication details. We'll use the Adafruit SSD1306 library, which is designed specifically for these OLED displays and provides functions to draw text, shapes, and graphics.

The display works by maintaining a buffer in the ESP32's memory that represents every pixel on the screen. When we write text or draw shapes, we're actually modifying this buffer. Nothing appears on the physical display until we call the display function, which sends the entire buffer to the OLED over I2C. This approach gives us smooth updates because we can prepare a complete frame before showing it.

For our application, we'll display the temperature and humidity readings on the OLED instead of just printing them to Serial. This makes the device more standalone and user-friendly since you can see the data without needing a computer connected.

Steps for OLED Integration

- Include three new libraries: Wire, Adafruit_GFX, and Adafruit_SSD1306
- Define display constants: SCREEN_WIDTH (128), SCREEN_HEIGHT (64), OLED_RESET (-1), and SCREEN_ADDRESS (0x3C)
- Create the display object using the Adafruit_SSD1306 constructor
- In setup(), initialize the display with error checking, clear it, and set text properties
- In loop(), after reading sensor values, clear the display buffer, position the cursor, write formatted temperature and humidity text, and call display.display() to update the screen

Add these includes at the top:

```
     #include <Wire.h>
     #include <Adafruit_GFX.h>
     #include <Adafruit_SSD1306.h>
```

Add these defines after DHT_TYPE:

```
     #define I2C_SDA 21
     #define I2C_SCL 22
     #define SCREEN_WIDTH 128
     #define SCREEN_HEIGHT 64
     #define OLED_RESET -1
     #define SCREEN_ADDRESS 0x3C
```

Add this after the dht object:

```
     Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT,
&Wire, OLED_RESET);
```

Add this in setup() after dht.begin():

```
     // I2C init
     Serial.println("Initializing I2C bus...");
     Wire.begin(I2C_SDA, I2C_SCL);

     if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
       Serial.println("SSD1306 allocation failed");
       for(;;);
     }

     display.clearDisplay();
     display.setTextSize(1);
     display.setTextColor(SSD1306_WHITE);
     display.display();
```

Replace the entire loop() with:

```
   void loop() {
     float humidity = dht.readHumidity();
     float temperature = dht.readTemperature();

     if (isnan(humidity) || isnan(temperature)) {
       Serial.println("Failed to read from DHT sensor!");
       display.clearDisplay();
       display.setCursor(0, 0);
       display.setTextSize(2);
       display.println("Sensor");
       display.println("Error!");
       display.display();
     } else {
       Serial.print("Humidity: ");
       Serial.print(humidity);
       Serial.print("%  Temperature: ");
       Serial.print(temperature);
       Serial.println("°C");

       display.clearDisplay();
       display.setTextSize(1);
       display.setCursor(0, 0);
       display.println("Weather Node:");
       display.println();
       display.print("Temp: ");
       display.print(temperature, 1);
```

```
        display.println(" C");
        display.println();
        display.print("Humidity: ");
        display.print(humidity, 1);
        display.println(" %");
        display.display();
      }

      delay(2000);
    }
```

Now that we have integrated all our sensors, the biggest challenge comes along. How do we share the data that we are measuring and collecting? There are various ways to do this. One way is you write an API, make post requests to it, and send the data to a central server or some sort of Edge Computing Node. This way works, but it has its own set of limitations with scalability, security, etc. It's not that this way is flawed; every method has its advantages, disadvantages, and flaws. You just choose the method that suits your specific use case better, and there's no singular one shoe fits all solution that can be applicable for every situation, and that being the most efficient one. So, what other ways do we have? Some of them are as follows:

MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight publish-subscribe messaging protocol designed for constrained devices and low-bandwidth networks. It uses a broker-based architecture where devices (clients) connect to a central broker that handles message routing. Clients can publish messages to topics and subscribe to topics to receive messages. The broker manages all subscriptions and ensures messages reach the right subscribers. MQTT supports three quality of service levels for message delivery guarantees and includes features like retained messages and last will and testament for handling disconnections. Its small code footprint and minimal packet overhead make it ideal for IoT applications.

HTTP (Hypertext Transfer Protocol)

HTTP is a request-response protocol where a client sends a request to a server and waits for a response. Each transaction is independent and stateless, meaning the server doesn't maintain information about previous requests. While originally designed for web browsing, HTTP is commonly used in IoT through REST APIs. Devices send sensor data via POST requests or query information via GET requests. HTTP is easy to implement and works through most firewalls, but it has higher overhead than MQTT and requires the device to initiate all communication. HTTPS adds encryption for secure data transmission.

Zigbee:

Zigbee is a low-power wireless mesh networking protocol operating on 2.4 GHz and sub-GHz frequencies. It uses a mesh topology where devices can relay messages for each other, extending network range and providing redundancy. The network consists of coordinators, routers, and end devices. Coordinators manage the network, routers extend range and route messages, and end devices are typically battery-powered sensors. Zigbee is designed for short-range communication with low data rates but excellent power efficiency. It's commonly used in home automation, where many battery-powered sensors need to communicate reliably over extended periods.

LoRaWAN (Long Range Wide Area Network)

LoRaWAN is a protocol for long-range, low-power wireless communication designed for IoT applications that need to cover large areas. It uses a star topology where end devices communicate directly with gateways, which then forward messages to a network server via standard internet protocols. LoRaWAN can achieve ranges of several kilometers in rural areas and penetrate buildings well in urban environments. It operates in unlicensed spectrum bands and supports bidirectional communication, though with very low data rates. The protocol prioritizes battery life and range over bandwidth, making it suitable for applications like agricultural monitoring, smart city sensors, and asset tracking, where devices need to operate for years on a single battery.

CoAP (Constrained Application Protocol)

CoAP is a specialized web transfer protocol designed for constrained devices and networks. It follows a client-server model similar to HTTP but uses UDP instead of TCP, reducing overhead and connection establishment time. CoAP messages are much smaller than HTTP, making them suitable for low-power devices with limited processing capabilities. The protocol supports resource discovery, allowing devices to advertise their available services. CoAP also includes built-in support for observing resources, enabling a publish-subscribe pattern where clients can register to receive notifications when a resource changes. This makes it more efficient than repeatedly polling for updates.

For this project, we'll use MQTT to connect our ESP32 to AWS IoT Core. Before diving into the setup, let's understand some key MQTT concepts. An **MQTT broker** is a server that sits in the middle of all communication. Think of it like a post office where devices can drop off messages (publish) and pick up messages (subscribe). Devices don't talk directly to each other; they all talk to the broker, and the broker handles delivering messages to the right recipients. **AWS IoT Core** will be our broker, but other popular options include **Mosquitto (open source and free), HiveMQ, and EMQX.**

Topics are like **addresses** or **channels** in **MQTT**. They use a hierarchical structure with forward slashes, similar to file paths. For example, home/livingroom/temperature or factory/machine5/status. **When a device publishes a message, it sends that message to a specific topic. When a device subscribes to a topic, it tells the broker, "send me all messages that arrive on this topic."** Multiple devices can publish to the same topic, and multiple devices can subscribe to the same topic. The broker handles making sure all subscribers receive copies of published messages.

Publishing means sending a message to a topic. When our ESP32 reads temperature and humidity from the DHT22 sensor, it will publish that data to a topic. Subscribing means registering interest in a topic so you receive messages published to it. Our ESP32 will subscribe to a commands topic so it can receive instructions to control the LED.

In **AWS IoT Core**, every device is represented as a **Thing. A Thing is simply a virtual representation of your physical device in the AWS cloud.** Each Thing has a unique name that identifies it. This Thing name becomes part of our topic structure to keep different devices' messages separated. If we name our Thing weatherStation_01, then our topics become weatherStation_01/data for publishing sensor readings and weatherStation_01/commands for receiving control messages.

For our project, we'll operate with two topics based on the Thing name. The first topic, **thingName/data**, is where our device **publishes sensor readings. Every few seconds, the ESP32 will send temperature and humidity data as JSON-formatted messages to this topic. JSON (JavaScript Object Notation) is a text format that structures data as**

key-value pairs, making it easy for both humans and machines to read. A typical message might look like {"temperature": 23.5, "humidity": 45.2}.

The second topic, **thingName/commands**, is where our device listens for incoming control messages. We'll subscribe to this topic to receive JSON commands that can control the LED state. For example, sending **{"message": "led_on"}** to this topic will turn the **LED on** and vice versa. **This bidirectional communication pattern is common in IoT systems where devices need to both report their status and receive instructions from users or automated systems.**

AWS IoT Core handles all the message routing, security through certificates, and can scale to millions of devices. We only need to focus on publishing our sensor data and handling incoming commands. The publish-subscribe model means multiple systems or users can monitor our sensor data simultaneously without the device knowing or caring how many listeners there are. Similarly, any authorized MQTT client can send commands to our device without needing a direct connection.

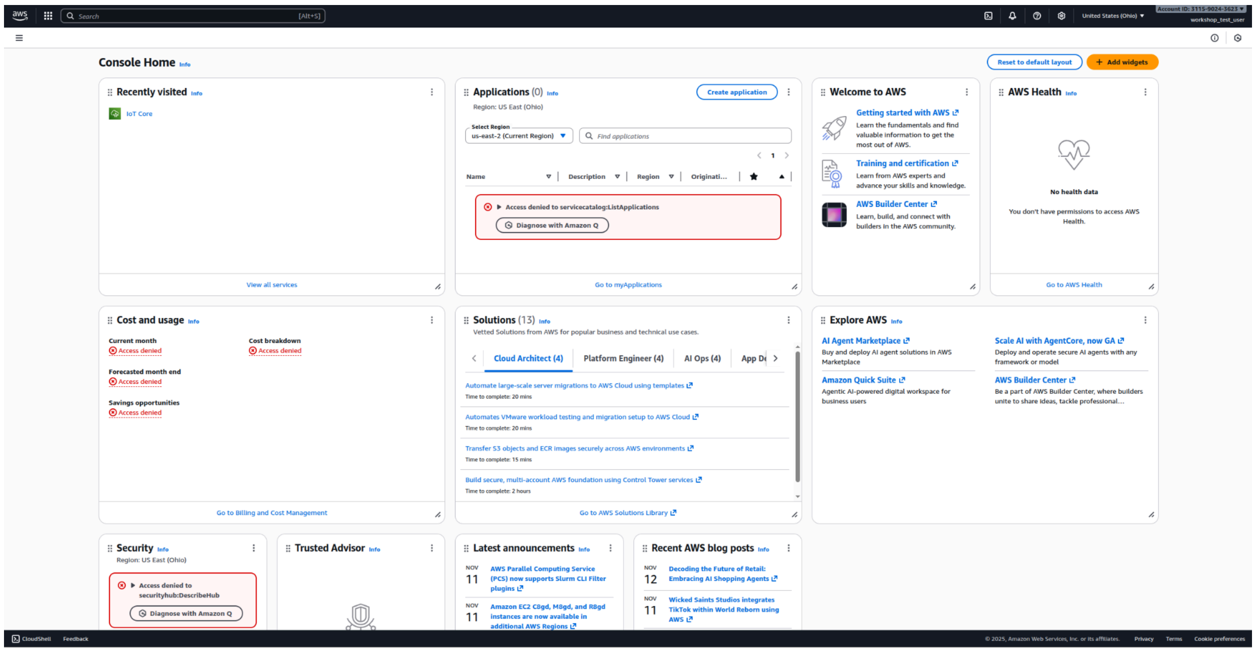
Getting Started with AWS IoT Core:

All your hardware kits should have come with a note that has the login URL and the credentials you will need to use to log in to the AWS console. Once you access it per the setup, you will need to set up your own password, and you will need to note it down or remember it safely, as that new password will be used to access your account. All the AWS accounts you are given only have access to the AWS IoT Core for now. If you want to explore other services, please approach us after the workshop via email, Teams, or in person, and we will create a new IAM account with the services you want to explore.

Once you have logged in and reset your password, and relogged in, you should see a home page similar to this. Usually, you would see a whole bunch of other services, but for security reasons, all the accounts are restricted to only AWS IoT Core full access for the duration of the workshop.

Also, please note that the Accounts will be deleted on 16th November, 2025 11:59 PM EST

So if you need extended access, please contact us, and we can either give you access to an AWS IAM instance or help you create your own account.



Search [Alt+S] | United States (Ohio) | Account ID: 3115-9024-3623 | workshop_test_user

Console Home Info

Reset to default layout | + Add widgets

Recently visited Info
IoT Core
View all services

Applications (0) Info
Create application
Region: US East (Ohio)
Select Region: us-east-2 (Current Region)
Find applications

Name	Description	Region	Originati...

Access denied to servicecatalog:ListApplications
Diagnose with Amazon Q
Go to myApplications

Welcome to AWS
Getting started with AWS
Learn the fundamentals and find valuable information to get the most out of AWS.
Training and certification
Learn from AWS experts and advance your skills and knowledge.
AWS Builder Center
Learn, build, and connect with builders in the AWS community.

AWS Health Info
No health data
You don't have permissions to access AWS Health.
Go to AWS Health

Cost and usage Info
Current month: Access denied
Cost breakdown: Access denied
Forecasted month end: Access denied
Savings opportunities: Access denied
Go to Billing and Cost Management

Solutions (13) Info
Vetted Solutions from AWS for popular business and technical use cases.
Cloud Architect (4) | Platform Engineer (4) | AI Ops (4) | App De
Automate large-scale server migrations to AWS Cloud using templates
Time to complete: 20 mins
Automates VMware workload testing and migration setup to AWS Cloud
Time to complete: 20 mins
Transfer S3 objects and ECR images securely across AWS environments
Time to complete: 15 mins
Build secure, multi-account AWS foundation using Control Tower services
Time to complete: 2 hours
Go to AWS Solutions Library

Explore AWS Info
AI Agent Marketplace
Buy and deploy AI agent solutions in AWS Marketplace
Scale AI with AgentCore, now GA
Deploy and operate secure AI agents with any framework or model
Amazon Quick Suite
Agentic AI-powered digital workspace for business users
AWS Builder Center
Be a part of AWS Builder Center, where builders unite to share ideas, tackle professional...

Security Info
Region: US East (Ohio)
Access denied to securityhub:DescribeHub
Diagnose with Amazon Q

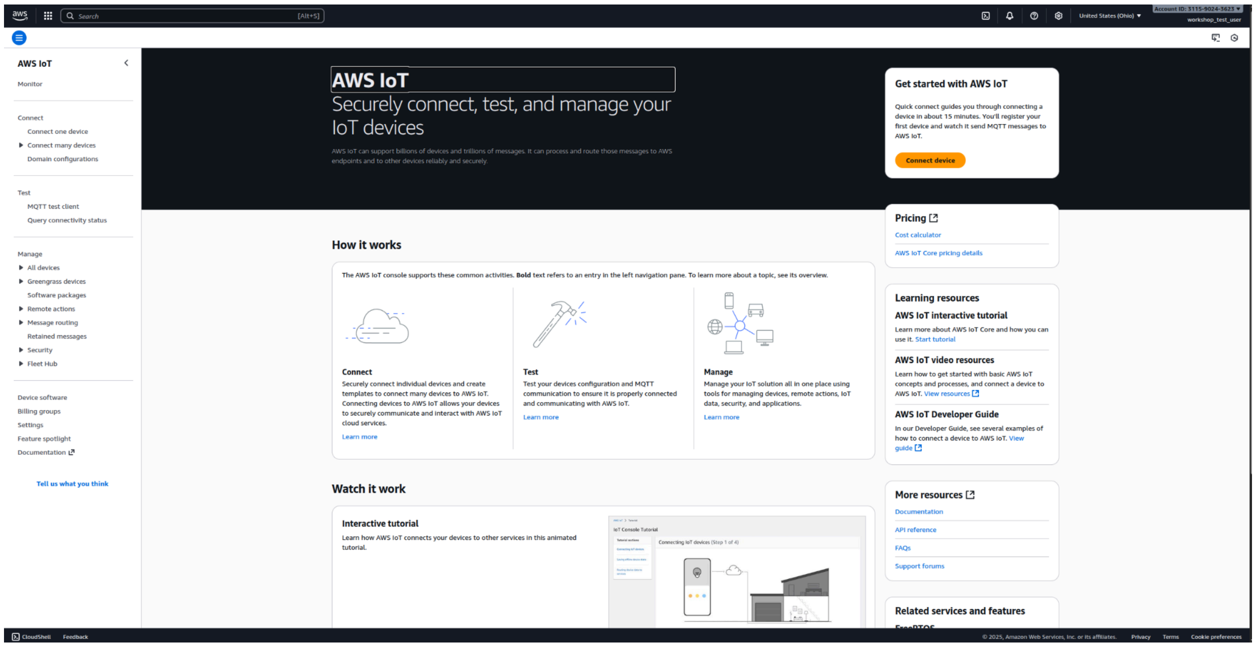
Trusted Advisor Info

Latest announcements Info
NOV 11 AWS Parallel Computing Service (PCS) now supports Slurm CLI Filter plugins
NOV 11 Amazon EC2 C8gd, M8gd, and R8gd instances are now available in additional AWS Regions

Recent AWS blog posts Info
NOV 12 Decoding the Future of Retail: Embracing AI Shopping Agents
NOV 11 Wicked Saints Studios integrates TikTok within World Reborn using AWS

CloudShell | Feedback | © 2025, Amazon Web Services, Inc. or its affiliates. | Privacy | Terms | Cookie preferences

From here, using the search menu, please open the AWS IoT Core console. Here you will be able to access the IoT Core dashboard. It will look something like the following



AWS IoT

Securely connect, test, and manage your IoT devices

AWS IoT can support billions of devices and trillions of messages. It can process and route those messages to AWS endpoints and to other devices reliably and securely.

Get started with AWS IoT

Quick connect guides you through connecting a device in about 15 minutes. You'll register your first device and watch it send MQTT messages to AWS IoT.

Connect device

Pricing

Cost calculator

AWS IoT Core pricing details

How it works

The AWS IoT console supports these common activities. **Bold** text refers to an entry in the left navigation pane. To learn more about a topic, see its overview.

Connect
Securely connect individual devices and create templates to connect many devices to AWS IoT. Connecting devices to AWS IoT allows your devices to securely communicate and interact with AWS IoT cloud services.
Learn more

Test
Test your devices configuration and MQTT communication to ensure it is properly connected and communicating with AWS IoT.
Learn more

Manage
Manage your IoT solution all in one place using tools for managing devices, remote actions, IoT data, security, and applications.
Learn more

Learning resources

AWS IoT interactive tutorial

Learn more about AWS IoT Core and how you can use it. Start tutorial

AWS IoT video resources

Learn how to get started with basic AWS IoT concepts and processes, and connect a device to AWS IoT. View resources

AWS IoT Developer Guide

In our Developer Guide, see several examples of how to connect a device to AWS IoT. View guide

Watch it work

Interactive tutorial

Learn how AWS IoT connects your devices to other services in this animated tutorial.

IoT Console Tutorial

Connecting IoT devices (Step 1 of 4)

More resources

Documentation

API reference

FAQs

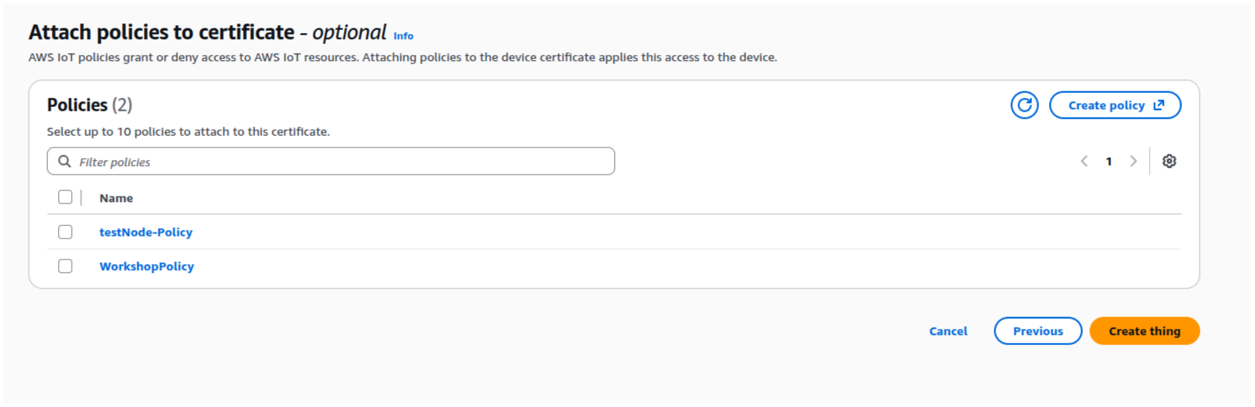
Support forums

Related services and features

From here, you can click on **All Things** on the left panel under **Manage** and select **Things** from the sub menu. You should be seeing a button to **Create Things.** From here, click the button, and it should ask you if you want to **create a single thing** or **create multiple things**. Usually, you would select either of the options depending on your use case, but for now, you can click on the option that says **Create Single Thing.**

Here, you can name your thing and leave the other options as is and move to the next page. For the device certificate, you will want to select **Auto-Generate New Certificate.** Please note that **the device certificates, private keys, and all are very sensitive information so please never push them to any public place or share confidential keys. It is considered a very bad practice.**

From here, you can move to the final page and select the policy that says **WorkshopPolicy.** What that is and how all of it works, we will discuss shortly.



Attach policies to certificate - *optional* Info

AWS IoT policies grant or deny access to AWS IoT resources. Attaching policies to the device certificate applies this access to the device.

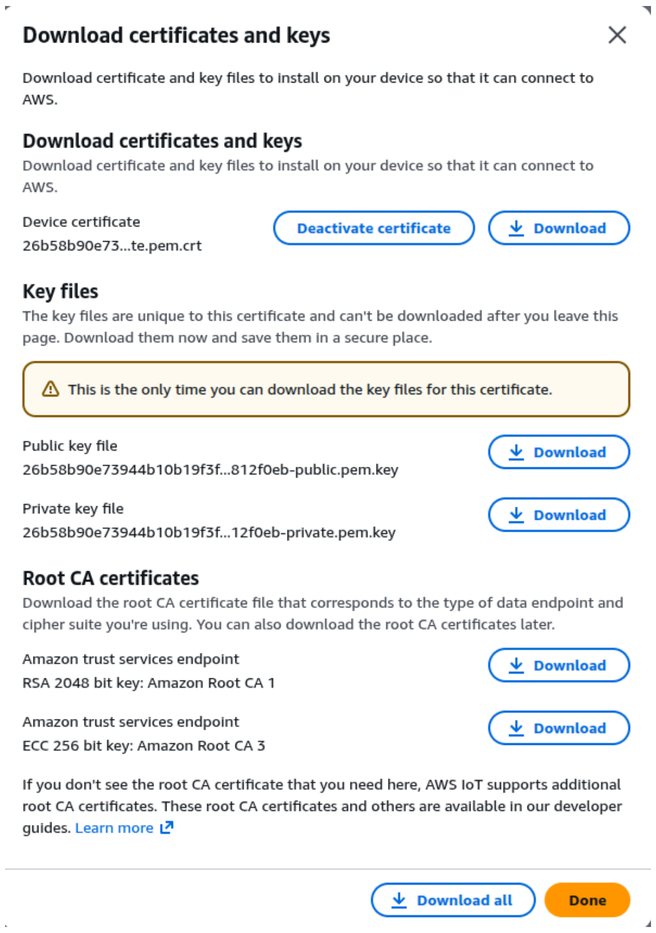
Policies (2)

Select up to 10 policies to attach to this certificate.

Name
testNode-Policy
WorkshopPolicy

Now you can click on **Create Thing** and you will be presented with the generated Device Certificates. You will want to download the following files:

- Private key file
- Public key file (Optional, we don't have a use for this at the moment)
- Device certificate
- Amazon trust services endpoint RSA 2048-bit key: Amazon Root CA 1



Download certificates and keys

Download certificate and key files to install on your device so that it can connect to AWS.

Download certificates and keys

Download certificate and key files to install on your device so that it can connect to AWS.

Device certificate
26b58b90e73...te.pem.crt

Key files

The key files are unique to this certificate and can't be downloaded after you leave this page. Download them now and save them in a secure place.

This is the only time you can download the key files for this certificate.

Public key file
26b58b90e73944b10b19f3f...812f0eb-public.pem.key

Private key file
26b58b90e73944b10b19f3f...12f0eb-private.pem.key

Root CA certificates

Download the root CA certificate file that corresponds to the type of data endpoint and cipher suite you're using. You can also download the root CA certificates later.

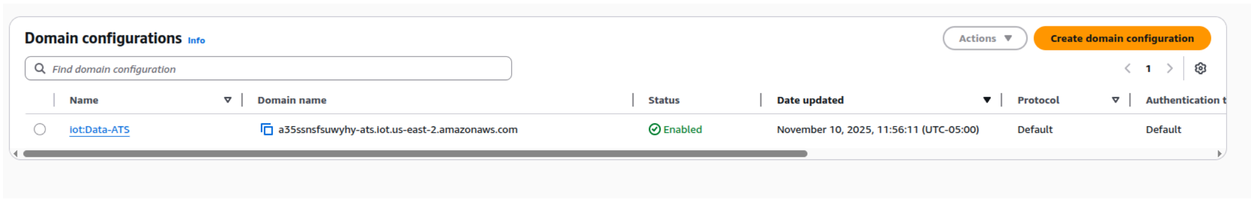
Amazon trust services endpoint
RSA 2048 bit key: Amazon Root CA 1

Amazon trust services endpoint
ECC 256 bit key: Amazon Root CA 3

If you don't see the root CA certificate that you need here, AWS IoT supports additional root CA certificates. These root CA certificates and others are available in our developer guides. Learn more

Keep these files safe and treat them as confidential. If the keys are leaked, anyone with them can access the Thing's endpoint and perform malicious actions, as they can pose as the device.

Once completed, you need to select the option **Domain Management** on the left-hand side panel. From here, please note the domain name unique to you. This will be the IoT endpoint where we will be sending all our data.


Domain configurations Info

Name	Domain name	Status	Date updated	Protocol	Authentication t
iot:Data-ATS	a35ssnsfsuwyhy-ats.iot.us-east-2.amazonaws.com	Enabled	November 10, 2025, 11:56:11 (UTC-05:00)	Default	Default

Now that we have all the configurations set up, let's work on having the wifi working on the node. The ESP-32 platform has both Bluetooth BLE and Wifi built into it. It is advisable to connect to the 2.4 GHz network for better compatibility. Due to campus limitations on the TLS communication for unregistered hardware, we are using a kind of janky setup. You will need to connect to the wifi network specified on the whiteboard for the ESP-32. **Please note this is for the ESP-32 only due to the network restrictions. Please connect to the 2.4 GHz network.**

Add these includes at the top:

```
#include <WiFi.h>
#include "secrets.h"
```

Add WiFi connection function before setup():

```
void connectToWiFi() {
  Serial.print("Connecting to WiFi");
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("\nConnected to WiFi");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());
}
```

Add this at the beginning of setup(), right after Serial.begin():

```
connectToWiFi();
```

Create a new file called secrets.h in the same folder as:

```
#define WIFI_SSID "your_wifi_name"
#define WIFI_PASSWORD "your_wifi_password"
```

Once this is run, you can see in the serial monitor the IP address of the node along with the sensor readouts. We've now connected our ESP32 to WiFi, which is the first step toward cloud communication. The ESP32 has built-in WiFi capabilities, and we use the WiFi library that comes with the ESP32 board package to access them. We created a separate file called secrets.h to store our WiFi credentials. This is a common practice in development because it keeps sensitive information separate from the main code. If you share your code or upload it to GitHub, you can exclude the secrets file to protect your credentials.

The connectToWiFi() function handles the WiFi connection process. It calls WiFi.begin() with the SSID (network name) and password, then enters a while loop that checks the connection status. The loop prints dots to the Serial Monitor while waiting, giving visual feedback that the connection is in progress. Once connected, it prints the assigned IP address. This IP address is how other devices on the network can identify our ESP32. Now, our next step is to connect this node to AWS IoT, and we can start publishing to the topic named **data.**

First, we add in a few fields to the secrets.h

Add in the define statement for the Thing Name you specified in the IoT console

```
#define THINGNAME "your_thing_name_here"
```

Next, add in the following:

```
const char AWS_IOT_ENDPOINT[] =
"your-endpoint.iot.us-east-2.amazonaws.com";

// Amazon Root CA 1
static const char AWS_CERT_CA[] PROGMEM = R"EOF(
-----BEGIN CERTIFICATE-----
paste your root CA certificate here
-----END CERTIFICATE-----
)EOF";

// Device Certificate
static const char AWS_CERT_CRT[] PROGMEM = R"EOF(
-----BEGIN CERTIFICATE-----
paste your device certificate here
-----END CERTIFICATE-----
)EOF";

// Device Private Key
static const char AWS_CERT_PRIVATE[] PROGMEM = R"EOF(
-----BEGIN RSA PRIVATE KEY-----
paste your private key here
-----END RSA PRIVATE KEY-----
)EOF";
```

Here, we just defined the configuration we need for the connection to IoT Core. It uses TLS encryption for the HTTPS communication with the MQTT broker, and the certificates we downloaded and paste in here will be the basis for the TLS authentication. Without these, your device will be treated as an unauthorized node and won't be able to communicate with IoT Core. Now, let's go back to our file and start implementing it.

Add these includes after WiFi.h:

```
#include <WiFiClientSecure.h>
#include <MQTT.h>
```

Add these global objects after the display object:

```
WiFiClientSecure net = WiFiClientSecure();
MQTTClient client = MQTTClient(256);
```

WiFiClientSecure net = WiFiClientSecure(); creates a secure WiFi client object that handles encrypted communication. The "Secure" part means it supports TLS/SSL encryption, which is required for AWS IoT Core. This object manages the actual network connection and handles the certificate-based authentication we set up with **setCACert(), setCertificate(),** and **setPrivateKey().**

MQTTClient client = MQTTClient(256); creates an **MQTT client object** with a **256-byte buffer size.** This is the object that handles all MQTT operations like connecting to the broker, publishing messages, and subscribing to topics. **The 256 specifies the maximum size of MQTT messages it can handle. The MQTT client uses the net object underneath for the actual network communication, which is why we pass net to client.begin() in the connectAWS() function.**

Add the ensureMqtt() function

```
void ensureMqtt() {
  while (!client.connected()) {
    Serial.print("Connecting MQTT to AWS as clientId=");
    Serial.println(THINGNAME);

    if (client.connect(THINGNAME)) {
      Serial.println("MQTT connected");
    }
    else {
               Serial.printf("MQTT  connect  failed,  status=%d.
Retrying...\n",
                    client.lastError());
      delay(1500);
    }
  }
}
```

The ensureMqtt() function ensures that the MQTT connection to AWS IoT Core is established and maintained. It checks if the client is connected, and if not, attempts to connect using the Thing name as the client ID. If the connection fails, it displays the error code and retries every 1.5 seconds. This function can be safely called multiple times - it only attempts a connection if not already connected.

Now we define the PUB_TOPIC in the file secrets.h, which will be the topic to which we will be sending the sensor readouts.

```
#define PUB_TOPIC "yourNodeName/data"
```

Now we add in the publishSensor() method, which posts data to AWS using the topic we defined in the secrets file.

```
void publishSensor(float t, float h) {
  char payload[128];
  snprintf(payload, sizeof(payload),
           "{\"temperature\":%.2f,\"humidity\":%.2f}", t, h);
  bool ok = client.publish(PUB_TOPIC, payload);
  Serial.printf("Publish %s: %s\n", ok ? "OK" : "FAIL",
payload);
}
```

This function takes temperature and humidity readings, formats them into a JSON payload, and publishes them to AWS IoT Core. The snprintf() function safely formats the data with 2 decimal places, and client.publish() sends it to the PUB_TOPIC defined in your secrets.h file.

Update your setup() function to include AWS IoT configuration:

```
void setup() {
  Serial.begin(115200);

  connectToWiFi();

  // Configure TLS certificates (one-time setup)
  net.setCACert(AWS_CERT_CA);
  net.setCertificate(AWS_CERT_CRT);
  net.setPrivateKey(AWS_CERT_PRIVATE);

  // Initialize MQTT client
  client.begin(AWS_IOT_ENDPOINT, 8883, net);

  // Connect to AWS IoT
  ensureMqtt();

  pinMode(LED_PIN, OUTPUT);
  dht.begin();
  // I2C init
  Serial.println("Initializing I2C bus...");
  Wire.begin(I2C_SDA, I2C_SCL);
  if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println("SSD1306 allocation failed");
    for(;;);
  }

  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.display();

  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &onTimer);
  timerAlarm(timer, 1000000, true, 0);
}
```

The **setCACert()**, **setCertificate()**, and **setPrivateKey()** methods load the AWS certificates into the secure WiFi client for authentication. This is done once during setup.

client.begin() initializes the MQTT client with the **AWS endpoint, port 8883 (MQTT over TLS)**, and the secure network client.

ensureMqtt() establishes the initial connection to AWS IoT Core with proper error handling.

Add connection maintenance in loop()

```
// Maintain connections
  if (WiFi.status() != WL_CONNECTED)
    connectToWiFi();
  if (!client.connected())
    ensureMqtt();

  client.loop();
```

The **WiFi check** ensures you reconnect if the WiFi drops

The **ensureMqtt()** call automatically reconnects to AWS if the MQTT connection drops

client.loop() processes MQTT keep-alive packets to maintain the connection

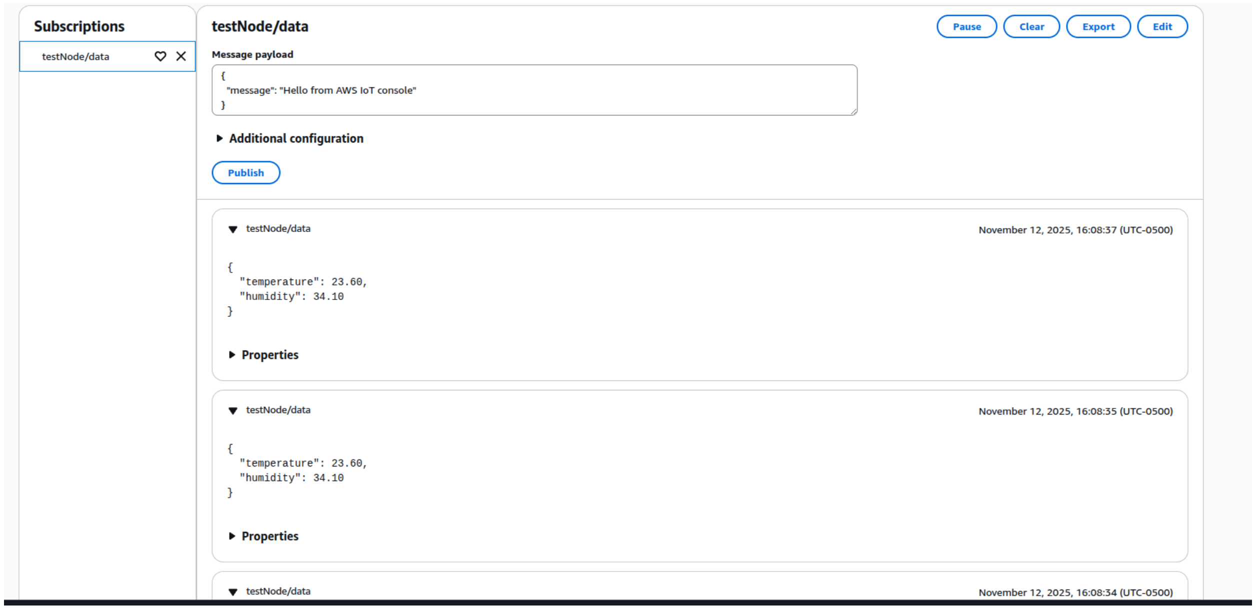
Add this inside the else block where you display sensor data:

```
// Publish sensor data to AWS
publishSensor(temperature, humidity);
```

Our code should look like this after adding all these changes to incorporate the publish functionality. Take a look at the file **Publish_Checkpoint_Code.txt.**

https://drive.google.com/drive/folders/1FI86X0qgz52V4gqPdhDuHUduhmTBC4w4?usp=sharing

If you run the code and use the AWS MQTT Test Client and subscribe to the topic you defined, you should be able to see the JSON messages being posted.



testNode/data

Message payload

```
{
  "message": "Hello from AWS IoT console"
}
```

testNode/data November 12, 2025, 16:08:37 (UTC-0500)

```
{
  "temperature": 23.60,
  "humidity": 34.10
}
```

testNode/data November 12, 2025, 16:08:35 (UTC-0500)

```
{
  "temperature": 23.60,
  "humidity": 34.10
}
```

testNode/data November 12, 2025, 16:08:34 (UTC-0500)

So, with this, we have successfully published data to AWS IoT Core. But wait, we can make this bidirectional. To demonstrate this, we are gonna get rid of the blink functionality, and we will use the **nodeName/commands** topic to listen to, and we will listen for incoming commands to turn the LED on or off. So by the end of this, you will be able to control your LED as long as the node is connected to internet. This is in fact how most of our in-house IoT devices are controlled remotely. Let's get started.

First, we remove timer-related global variables. Delete these lines from the global variable section:

```
hw_timer_t *timer = NULL;
volatile bool ledState = false;
```

Now we remove the timer interrupt function

```
cppvoid IRAM_ATTR onTimer() {
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState);
}
```

Finally, we remove timer initialization from setup(). For this, delete these lines from the setup() function:

```
cpptimer = timerBegin(1000000);
timerAttachInterrupt(timer, &onTimer);
timerAlarm(timer, 1000000, true, 0);
```

Now, we need to parse the incoming stream of data from the topic we are gonna subscribe to. Before any of it, let's define the subscribe topic in the secrets file.

```
#define SUB_TOPIC "yourNodeName/commands"
```

Add the following message parser function

```
void messageHandler(String &topic, String &payload) {
  Serial.println(">>> Message Received");
  Serial.println("Topic: " + topic);
  Serial.println("Payload: " + payload);

  if (topic == SUB_TOPIC) {
    if (payload.indexOf("led_on") != -1) {
      digitalWrite(LED_PIN, HIGH);
      Serial.println("LED turned ON");
    }
    else if (payload.indexOf("led_off") != -1) {
      digitalWrite(LED_PIN, LOW);
      Serial.println("LED turned OFF");
    }
    else {
      Serial.println("Unknown command payload");
    }
  }
}
```

The **messageHandler()** function is a callback that gets triggered automatically whenever a message arrives on a subscribed topic. It receives two parameters: the topic name and the message payload. The function first logs the received message for debugging purposes. Then it checks if the message came from our **SUB_TOPIC**. If it did, it parses the payload looking for "led_on" or "led_off" strings (using indexOf(), which returns -1 if the string isn't found). Based on what command is found, it either turns the LED on by setting the pin HIGH, turns it off by setting it LOW, or reports an unknown command if neither string is found. This allows us to remotely control the LED from AWS IoT Core by publishing JSON messages like **{"message": "led_on"}** or **{"message": "led_off"}** to the command topic.

Update **ensureMqtt()** to Include **Subscription Logic**

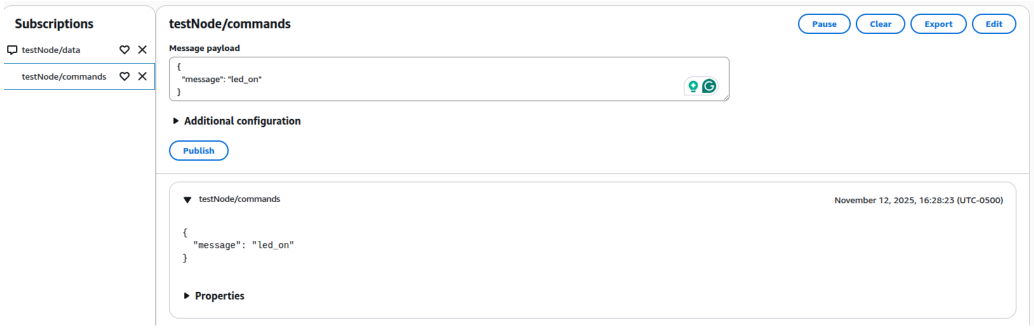
```
void ensureMqtt() {
  while (!client.connected()) {
    Serial.print("Connecting MQTT to AWS as clientId=");
    Serial.println(THINGNAME);

    if (client.connect(THINGNAME)) {
      Serial.println("MQTT connected");

      // Set up message handler for incoming messages
      client.onMessage(messageHandler);

      // Subscribe to the command topic
      if (client.subscribe(SUB_TOPIC)) {
        Serial.printf("Subscribed to topic: %s\n", SUB_TOPIC);
      }
      else {
        Serial.printf("Failed to subscribe to topic: %s\n",
SUB_TOPIC);
      }
    }
    else {
      Serial.printf("MQTT connect failed, status=%d.
Retrying...\n",
                    client.lastError());
      delay(1500);
    }
  }
}
```

If you are having trouble with the code, you can use the earlier link to the shared folder and use the **Final_Code.txt** file, which has the subscribe implemented, and check where it goes wrong, and also connect with us! On successful flashing, we can go ahead and open the MQTT Test Client from earlier and subscribe to the topic **yourNodeName/commands** so that it's available in the menu (Seriously, idk why it doesn't let me use publish directly). Then go to the publish section in the UI and select the new topic to publish to, and replace the message with led_on, and once it's published, the LED should be lit ON.



```
{
  "message": "led_on"
}
```

testNode/commands November 12, 2025, 16:28:23 (UTC-0500)

```
{
  "message": "led_on"
}
```