# **Chapter 9** Mastering IoT Connectivity Through Integrating Sensor Node with Message Queuing Telemetry Transport Protocol

IoT has changed the way we interact with our environment in a world where everything is connected. The various protocols that allow this connectivity include Message Queuing Telemetry Transport (MQTT), which is a light and powerful solution for real-time communication between sensor nodes and cloud platforms. This chapter aims to provide a comprehensive implementation of MQTT that includes both the hardware and software aspects. We'll first walk you through installing the MQTT broker, an important feature that enables efficient communication between devices. By following the instructions, you will gain a solid foundation to create a wireless local area network and cloud-based wireless sensor network. You will gain a comprehensive understanding of the design and implementation of MQTT-based, cloud-ready sensor nodes. With this knowledge, you will be able to design scalable IoT solutions that effectively collect, process, and send data from your sensor nodes to the cloud, enabling real-time control and monitoring. No matter how much experience you have in development or how familiar you are with IoT, the guides offered here will help you successfully integrate MQTT into your projects.

In this chapter, you will be able to

- Comprehend the foundational principles of MQTT, including its architecture, messaging model, and its role in IoT communication.
- Differentiate the communication models, connection management, data management, security features, and ideal use cases of MQTT from HTTP protocol.
- Install and configure an MQTT broker on operating systems such as Windows and Raspberry Pi as a local server.
- Execute Mosquitto command-line interface (CLI) commands and flags to publish messages, subscribe to topics, and manage QoS levels, retention, and authentication in both local and cloud-based MQTT broker.
- Connect to public or cloud MQTT brokers using software like MQTTX desktop client, to configure connections, subscribing to topics, and publish structured data in formats such as JSON.
- Implement the MQTT API to connect IoT devices to the cloud IoT platform such as Adafruit IO for secure publish a message with structured data and subscribe to a topic.
- Establish feeds and dashboards on Adafruit IO IoT platform and integrating data visualization widgets to monitor and analyze sensor data in real-time.

## 9.1 What is MQTT?

MQTT is a lightweight messaging protocol designed for low bandwidth, high latency, and systems connected to unreliable networks. It is ideal for connecting IoT devices such as sensors and actuators to the cloud or other systems in a simple, efficient, and reliable way [1], [2]. IBM invented and developed the MQTT protocol in the late 1990s. Its original application was to connect sensors on oil pipelines to satellite links [3], [4]. They needed a protocol to connect oil pipelines via satellite to minimize battery consumption, bandwidth, and reliable communications. This messaging protocol is scalable in unstable network environments because it allows asynchronous communication between publisher and subscriber that separates the message from sender and receiver in both space and time. After years of development, MQTT was recognized in 2014 by the Organization for the Advancement of Structured Information Standards (OASIS) as an open standard messaging protocol for machine-to-machine (M2M) communication and is now considered one of the leading

protocols for IoT [1]. Several implementations of MQTT are available using open-source software or programming language libraries like Python, Java, and C++.

## 9.2 Differences Between MQTT and HTTP

Choosing the right protocol for IoT applications requires understanding the differences between MQTT and HTTP. Selecting the appropriate communication protocol is crucial for effective data transfer when building and developing IoT systems, especially when devices have limited resources. Each has advantages and disadvantages, and which is better depends on the use case. Below is a detailed comparison of MQTT and HTTP considering several aspects.

**Communication Model**

MQTT uses a *publish-subscribe model* that allows flexible, two-way communication between devices. In this model, clients can publish messages with unique topics and other clients can subscribe to these topics to receive the message. This facilitates many-to-many communication, and the server can manage the scalability of the devices at a time. However, HTTP is based on the *request-response model* where the client's request establishes a new connection and the server then responds with the data for each request, resulting in significant latency, especially when processing multiple requests [5], [6], [7], [8], [9]. This protocol is one-to-one communication designed for hypertext data and file transfers on the web and is therefore heavier compared to MQTT.

**Connection Management**

MQTT maintains *persistent connection* meaning that devices remain connected to the broker and only publish or subscribe to messages when necessary. This approach reduces the need for frequent reconnections and saves power, which is important for battery-powered IoT devices. HTTP is a *short-lived session*, meaning the connection between client and server is terminated after each request-response cycle [9], [10], [11]. This requires a new connection for each request, which consumes more power and increases latency for IoT devices.

**Data Retention and Connection State**

MQTT enables message retention on the broker, allowing the client device to subscribe to the latest message for a topic, ensuring instant access to information. HTTP is a stateless protocol and doesn't retain data from the client. Additionally, MQTT supports *Last Will* messages, which notify other devices when a node goes offline unexpectedly [9], [12]. HTTP does not have built-in mechanisms for detecting device presence, making it more difficult to track the status of devices in real time.

**Data Packet Size**

MQTT is optimized for low bandwidth usage and uses two bytes of header message to establish a connection. Unlike HTTP, it uses more than eight bytes which includes detailed requests and response headers. This can become substantial in IoT applications where many small data messages are exchanged, leading to unnecessary bandwidth consumption [5], [13]. Its content relies on Base64 to encode and decode any binary code, hence creating more workload for the CPU. MQTT payload can be any type of data from plain text and binary data to formats like JSON or XML where encoding is not necessary. It can transfer the same amount of data 80 to 100 times faster than HTTP.

**Security**

Both MQTT and HTTP are TCP-based protocols that enable secure communication using SSL/TLS encryption. The main difference in their security measures lies in their authentication

and authorization verification methods. MQTT relies on the broker's capabilities such as username and password authentication or topic-based publishing and subscribing permissions. Conversely, HTTP offers a greater variety of options, including basic authentication, token authentication, and the use of certificates. Because of its extensive use in web applications, HTTP has an edge over MQTT in terms of security maturity.

**Ideal Use Cases**

MQTT is ideal for systems where low power, low bandwidth, low latency, and real-time updates are essential. These systems include constrained devices such as sensors and actuators used in IoT applications such as industrial automation, smart homes, biomedical devices, and more. When it comes to web applications, for example, cloud-based APIs built into IoT devices for firmware and data updates that need to be sent to a server that typically uses HTTP [4], [14].

## 9.3 MQTT Architecture

MQTT architecture is built around a publish-subscribe communication model, which significantly differs from traditional request-response models such as HTTP. The core components of the MQTT architecture include clients and a broker depicted in the Figure below. In the publish-subscribe model, devices cannot communicate directly. The broker acts as a central hub that receives, filters, and distributes messages based on specified topics [4].

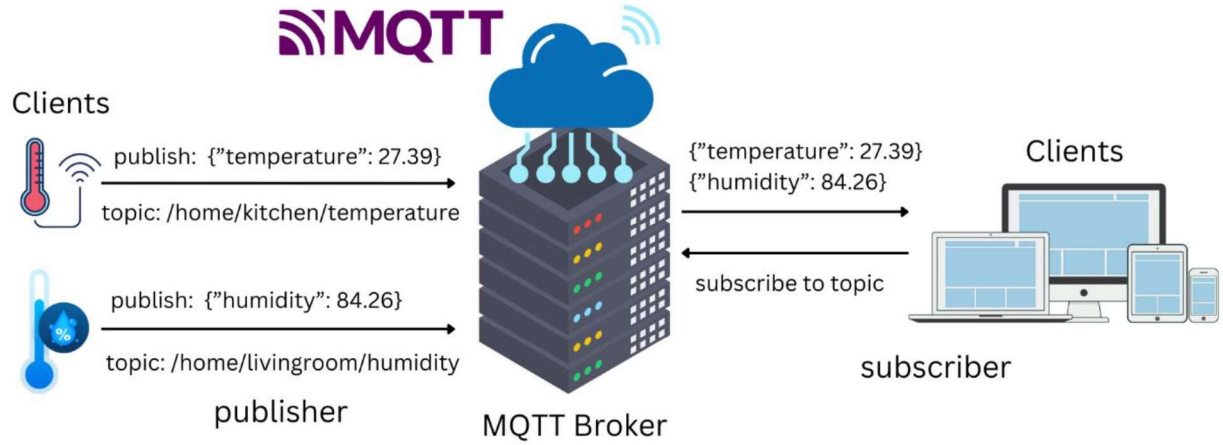



**Figure 1** *MQTT basic communication architecture.*

In the MQTT architecture shown in **Figure 1**, each message is tagged with a specific topic that categorizes the information being transmitted. Topics can be organized in a hierarchical structure using forward slashes "/" as delimiters and are organized similarly to URL paths. For instance, a topic structure for a smart home may look like /home/kitchen/temperature or /home/livingroom/humidity. Clients connect to the broker and publish messages with a topic while other clients subscribe to receive messages from that topic. Since MQTT messages are organized by topics, the application developer has the flexibility to specify that certain clients can only interact with certain messages [1]. The application can use any data format for the payload, such as text, JSON, XML, or other data encryption, as long as the subscriber can parse the payload.

## 9.3 Getting Started with MQTT

MQTT is a lightweight, low-bandwidth publish-subscribe messaging protocol optimized for M2M communications. MQTT provides a stable network to manage device communication, making it ideal for building connected systems, smart homes, or industrial IoT solutions. For each message to be published and subscribed to, an MQTT broker is required.

Although the term "broker" was first used in the MQTT documentation and is still commonly used, it is now also referred to as "server". MQTT brokers such as Mosquitto, which is free and open source, will be used for this guide [15], [16], [17]. This section covers installing the MQTT broker, learning common CLI commands and flags, and using public MQTT brokers to get started with MQTT development.

## 9.3.1 Install Mosquitto MQTT Broker on Windows

Eclipse Mosquitto is a widely used open-source MQTT broker that can be easily installed and configured on various platforms such as Windows and Raspberry Pi. To install Mosquitto on your computer, follow the guidelines listed below [18].

1. **Download and Install Mosquitto MQTT Broker**: Go to the official Mosquitto website https://mosquitto.org/download/. Download the newest Windows installation package. In this case, we have mosquitto-2.0.19-install-windows-x64.exe. Launch the Eclipse Mosquitto Setup Wizard as shown in **Figure 2** and click **Next >** to begin the installation process.


**Figure 2** *Eclipse Mosquitto setup installation window.*

**Figure 3** shows the **Visual Studio Runtime** and **Services** are required because Mosquitto relies on certain libraries and components it provides.

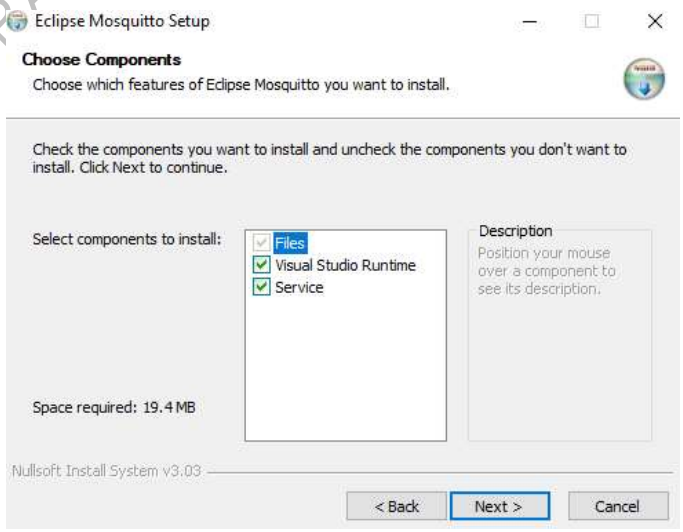



**Figure 3** *Install libraries or components required to install Eclipse Mosquitto MQTT broker.*

Click **Next >**, then select the Eclipse Mosquitto installation folder, or use the default installation directory path **C:\Program Files\mosquitto** for 64-bit or **C:\Program Files (x86)\mosquitto** for 32-bit machine as shown in **Figure 4**. Then click **Install**.

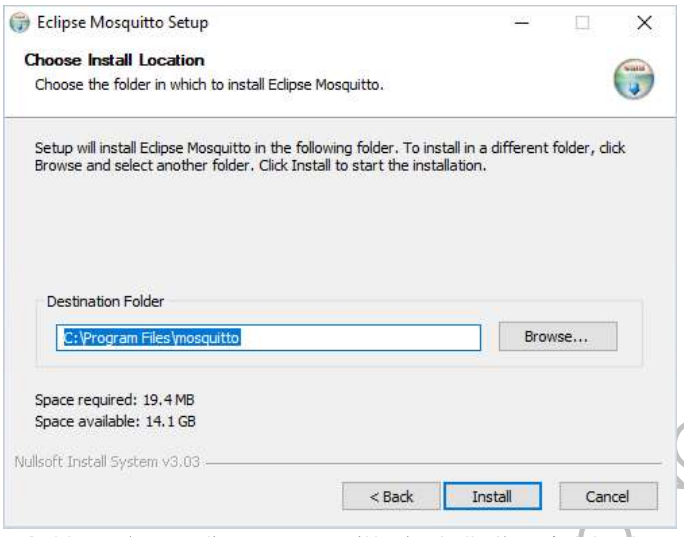



**Figure 4** *Choosing Eclipse Mosquitto installation folder location.*

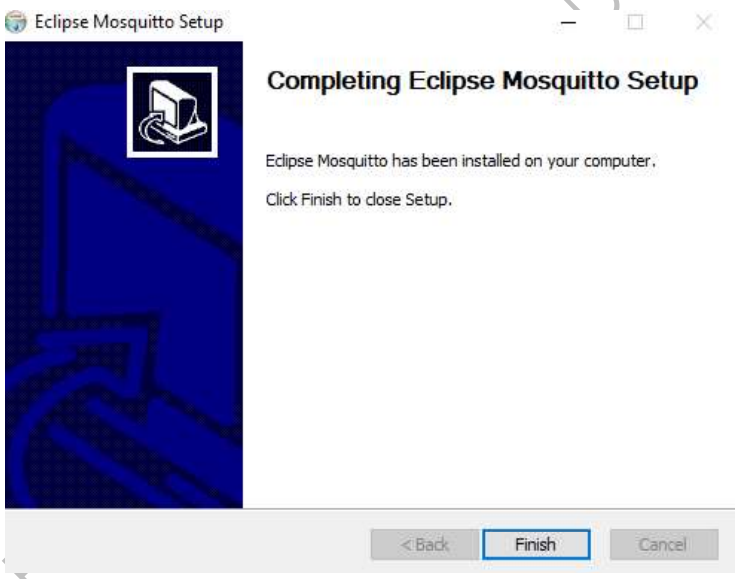



**Figure 5** *Completing Eclipse Mosquitto installation setup.*

Once the installation is complete, **Figure 5** shows a corresponding message will appear. To exit the setup wizard, click **Finish**.

2. **Adding Moquitto to System Path**: Mosquitto is not automatically globally available in the Command Prompt. To work with Mosquitto commands, you must either be inside the installation folder or add the installation folder to the system path environment variable. To add Mosquitto to the system path, locate the installation directory, which is typically found at **C:\Program Files\mosquitto** for 64-bit or **C:\Program Files (x86)\mosquitto** for 32-bit Windows machine. You can verify this by navigating to the folder where Mosquitto is installed and copy its path directory which contains mosquitto.exe. From your Windows Search Bar, type and search for Environment Variables, you should see a search result as shown in **Figure 6**.

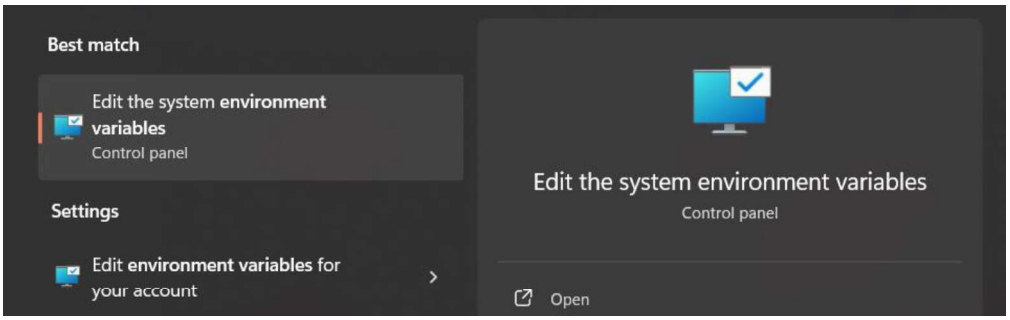



**Figure 6** *Search for Edit the system environment variables on Windows 11.*

Click Edit the system environment variables, this prompts you to a tab as shown in **Figure 7**.

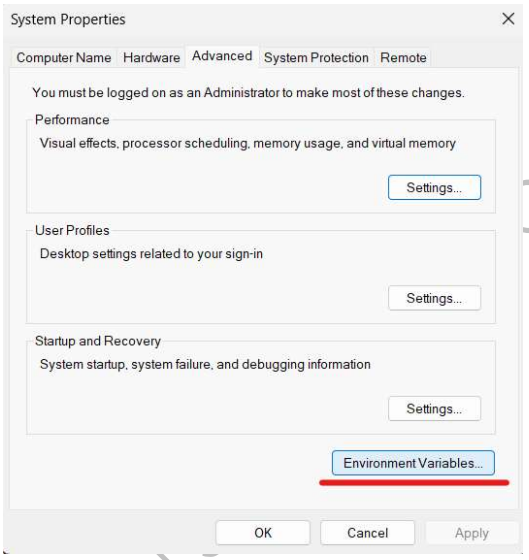



**Figure 7** *Editing environment variables on Windows 11 – System properties settings.*

Click **Environment Variables…** and under **System variables**, look for a variable named Path, then click Edit. The Edit environment variable window will appear. Click **New** and insert the path of your Mosquitto installation folder you copied earlier, and finally, click **OK**.

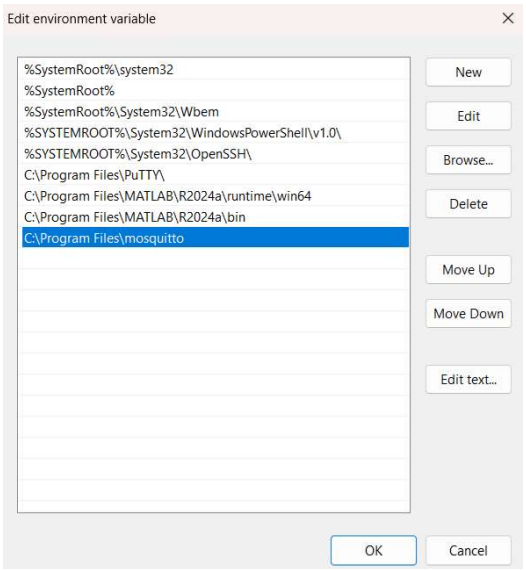



**Figure 8** *Adding Mosquitto installation directory path to system's environment variable.*

3. **Verify the Installation**: After installing the Mosquitto MQTT broker and adding it to your system variable, you may verify the installation using the Command Prompt. Open the Command Prompt and type command mosquito -h to check if Mosquitto commands work from any location. The prompt should respond with help information for the Mosquitto broker as shown below.

```
C:\Windows\System32>mosquito -h
mosquitto version 1.6.9

mosquitto is an MQTT v3.1.1 broker.

Usage: mosquitto [-c config_file] [-d] [-h] [-p port]

 -c : specify the broker config file.
 -d : put the broker into the background after starting.
 -h : display this help.
 -p : start the broker listening on the specified port.
      Not recommended in conjunction with the -c option.
 -v : verbose mode - enable all logging types. This overrides
      any logging options given in the config file.

See http://mosquitto.org/ for more information.
```

If the command executes successfully, the setup is complete.

4. **Install Mosquitto as a Windows Service**: After installation is completed, you can start the Mosquitto broker as a service. You can use Command Prompt or the Services application to do this. For ease of use, open Command Prompt as administrator and run the command.

```
C:\Windows\System32>net start mosquitto

The Mosquitto Broker service was started successfully.
```

Unless you change the listening setting for port 1883 in the mosquitto.conf file in the installation folder, the Mosquitto broker service listens on this port by default. Use the following command to see if Mosquitto is currently using port 1883.

```
C:\Windows\System32>netstat -an | findstr 1883
  TCP    0.0.0.0:1883           0.0.0.0:0              LISTENING
  TCP    [::]:1883              [::]:0                 LISTENING
```

The Mosquitto MQTT server has opened an IPv4 (0.0.0.0) and IPv6 ([::]) listening socket on port 1883, the output of this command is shown above. The Mosquitto broker is ready to accept both IPv4 and IPv6 connections on MQTT port 1883.

## 9.3.2 Install Mosquitto MQTT Broker on Raspberry Pi

The Raspberry Pi Foundation has developed a small, low-cost single-board computer called the Raspberry Pi. Originally designed to support the teaching of basic computer science in schools, it has since gained popularity for a variety of applications ranging from do-it-yourself electronics projects to industrial automation. If you are interested in electronics, programming, IoT, and more, Raspberry Pi has become a platform of choice due to its versatility, affordability, and large community of users.

One of the most powerful and feature-rich models in the Raspberry Pi family is Raspberry Pi 4 Model B+ as shown in **Figure 9** which was released in 2019. It brings a significant

upgrade in terms of performance and capabilities over its predecessors like Raspberry Pi 3 Model B, making it suitable for more demanding tasks, such as desktop computing, AI projects, and complex IoT applications.

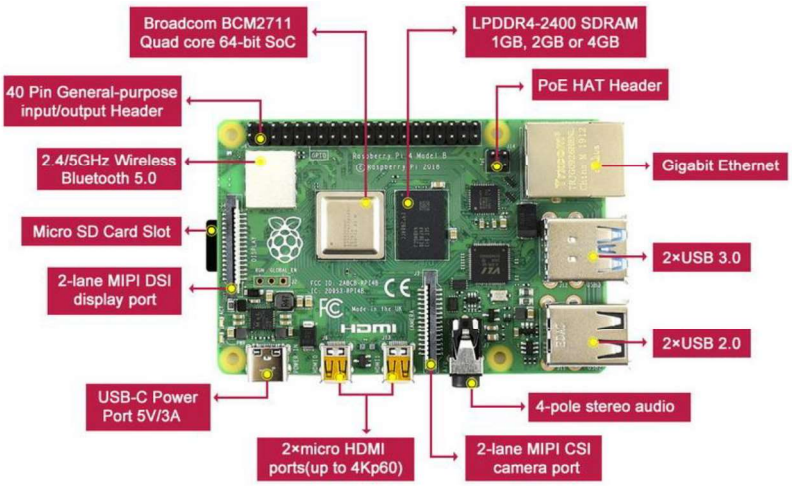



**Figure 9** *Raspberry Pi 4B+ board and its parts. Obtained from Hackatronic [19].*

It's powered by a Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC clocked at 1.5GHz, which is much faster and more powerful than the older Cortex-A53 cores found in previous Raspberry Pi models. Available in multiple variants with 1GB, 2GB, 4GB or 8GB LPDDR4 SDRAM, Raspberry Pi 4B+ offers more options for memory-intensive applications such as running databases, AI programs or multitasking environments. It can act as an IoT gateway, a server, collect data from devices and send it to the cloud, process it locally or control smart home devices. Increased storage and computing power, Raspberry Pi 4B+ can run image processing, and AI algorithms [20].

In this section, you will learn how to install the Mosquitto MQTT broker on a Raspberry Pi. This guide assumes that you already know how to set up and configure your Raspberry Pi, including installing operating system, network connectivity configuration, and enabling SSH for remote access of the board [21]. If you need help setting up Raspberry Pi, check out the official Raspberry Pi documentation (https://www.raspberrypi.com/documentation/).

If you have Raspberry Pi with the Raspberry Pi operating system installed (i.e., 64-bit Debian Bookworm is recommended for Raspberry Pi 4), you can continue with this guide, using different operating systems, the following guide may be different [20].

Before installing any new software, it's essential to ensure that your Raspberry Pi's package list and installed packages are up to date. Open a new Raspberry Pi terminal window run the update and upgrade commands as shown:

```
sudo apt update && sudo apt upgrade
```

We can proceed with the actual installation of Mosquitto, after the update has finished. The Mosquitto broker is available from the official Raspberry Pi package repository. To install Mosquitto and its client tools, execute the following command:

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

The mosquitto installs the MQTT broker software, which allows the Raspberry Pi to act as a broker and route messages between clients. The mosquito-clients include useful command line tools such as mosquitto_pub for publishing messages and mosquitto_sub for subscribing to topics.

Once installed, Mosquitto is not enabled by default, to ensure that the Mosquitto service starts automatically when your Raspberry Pi boots up, and start the service manually for the first time, run the following commands:

```
sudo systemctl start mosquitto
sudo systemctl enable mosquitto
```

This command returns a prompt that says that synchronize mosquitto as a service with the system as shown below.

```
Synchronizing state of mosquitto.service with SysV service script with
/lib/syst            emd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable mosquitto-v
```

To ensure that Mosquitto is running properly, you can check its status with the following command:

```
mosquitto -v
```

This returns the Mosquitto version that is currently running on your Raspberry Pi. In this case, the software version is 2.0.11, and the broker opens the port socket 1883 to listen.

```
1729762729: mosquitto version 2.0.11 starting
1729762729: Using default config.
1729762729: Starting in local only mode. Connections will only be possible
from clients running on this machine.
1729762729: Create a configuration file which defines a listener to allow
remote access.
1729762729: For more details see
https://mosquitto.org/documentation/authentication-methods/
1729762729: Opening ipv4 listen socket on port 1883.
1729762729: Error: Address already in use
1729762729: Opening ipv6 listen socket on port 1883.
1729762729: Error: Address already in use
```

If Mosquitto is running correctly, you can also verify that it's listening for connections on the default MQTT port 1883 by executing this command:

```
sudo netstat -tlnp | grep mosquitto
```

You should see the output as shown below.

```
tcp        0      0 127.0.0.1:1883          0.0.0.0:*               LISTEN      744/mosquitto
tcp6       0      0 ::1:1883                :::*                    LISTEN      744/mosquitto
```

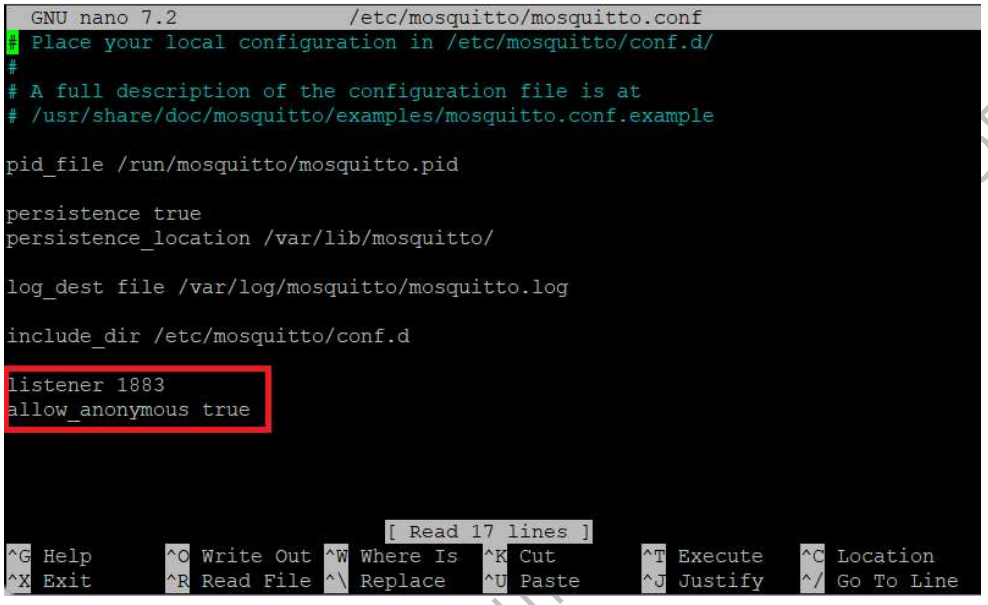
This confirms that the Mosquitto broker is listening on port 1883, ready to accept connections. However, as you see, the previous prompt message says "Starting in local only mode. Connections will only be possible from clients running on this machine. Create a configuration file which defines a listener to allow remote access." This means that by default, you can't communicate with the Mosquitto broker from another device. If you want other devices on your network to connect to the Raspberry Pi as an MQTT broker, you need to modify the configuration file to enable remote access. Open the Mosquitto configuration file in a text editor by executing the command:

```
sudo nano /etc/mosquitto/mosquitto.conf
```

Move to the end of the file by pressing down arrow key, then add the following line to allow Mosquitto to listen for connections from any IP address without authentication.

```
listener 1883
allow_anonymous true
```

Editing the configuration file by adding the two lines above should be the same as shown in configuration file below.




Press the shortcut key CTRL+X to exit then press Y and ENTER to save changes in the file. After changes are made to the configuration file, restart the Mosquitto service to apply the changes in the system by running the command:

```
sudo systemctl restart mosquitto
```

To verify that if Mosquitto is up and running after system service restart, run the command:

```
sudo systemctl status mosquitto
```

You should see output indicating that the Mosquitto service is active and enable as shown in the response prompt below.

```
● mosquitto.service - Mosquitto MQTT Broker
     Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled;
preset: en>
     Active: active (running) since Thu 2024-10-24 18:33:59 PST; 12min ago
       Docs: man:mosquitto.conf(5)
             man:mosquitto(8)
    Process: 2936 ExecStartPre=/bin/mkdir -m 740 -p /var/log/mosquitto
(code=ex>
    Process: 2937 ExecStartPre=/bin/chown mosquitto /var/log/mosquitto
(code=ex>
    Process: 2938 ExecStartPre=/bin/mkdir -m 740 -p /run/mosquitto
(code=exited>
...
```

Now, the Mosquitto broker will be accessible from any device on the network. To test remote access, use another device on the same network to publish or subscribe to topics using the IP address of your Raspberry Pi as the MQTT broker. To display the IP address of your Raspberry Pi, run the command:

```
hostname -I
```

It may display more than one IP address, use the one that corresponds to the network interface that you are using to connect the Raspberry Pi to the network. For example, if we have a prompt response:

```
192.168.1.40 2001:4450:4f95:e400:44a:53e7:127d:aca7
```

In this example, the IP address of the Raspberry Pi is 192.168.1.40. Note your Raspberry Pi IP address, and it will be used to test the MQTT connection and ready to handle communications between your IoT devices.

## 9.3.3 Mosquitto MQTT CLI Commands and Flags

Using a set of command line interface (CLI) tools offered by Mosquitto, you can publish, subscribe to, and manage MQTT brokers. The main Mosquitto CLI commands and their corresponding flags are listed below. These will help you manage MQTT communications effectively [22], [23], [24].

1. **`mosquitto`** - used to start the Mosquitto MQTT broker from the command line.

**`-c <config file>`:** Loads the broker configuration from the config file named mosquitto.conf.

**`-p <port>`**: Specifies the port for the broker to listen on, the default is 1883.

**`-d`**: Run the broker in the background refers to as a *daemon*.

**`-v`**: Run the broker in *verbose* mode, providing detailed logs.

2. **`mosquitto_pub`** - used to publish messages to a specified MQTT topic.

**`-h <hostname>`**: Specifies the hostname of the broker to connect to. It can be an IP address or a domain. The default hostname is localhost.

**`-t <topic>`**: The MQTT topic to publish a message.

**`-m <message>`**: The message or payload to be sent.

**`-q <QoS>`**: Specify the quality of service (QoS) level, from 0, 1, and 2. The default value is 0.

**`-u <username>`**: Specifies a username for authentication (optional).

**`-P <password>`**: Provide a password for authentication (optional). Using this argument without specifying a username is invalid.

**`-r`**: Makes the last message sent to be *retained* on the broker.

**`-d`**: Enables debugging messages to print detailed output.

3. **`mosquitto_sub`** – used to subscribe to topics and print the received messages.

**`-h <hostname>`**: Specifies the hostname of the broker to connect to. It can be an IP address or a domain. The default hostname is localhost.

**`-t <topic>`**: The MQTT topic to publish a message.

**`-m <message>`**: The message or payload to be sent.

**`-q <QoS>`**: Specify the quality of service (QoS) level, from 0, 1, and 2. The default value is 0.

**`-u <username>`**: Specifies a username for authentication (optional).

**`-P <password>`**: Provide a password for authentication (optional). Using this argument without specifying a username is invalid.

**`-r`**: Makes the last message sent to be *retained* on the broker.

**`-d`**: Enables debugging messages to print detailed output.

**`-v`**: Print both topic and payload upon message receipt. This is referred to as *verbose*.

## 9.3.4 Publishing a Message and Subscribing to a Topic

To test the installed Mosquitto MQTT Broker on Windows or IoT devices such as Raspberry Pi, publish and subscribe to messages using the mosquitto_pub and mosquitto_sub command lines as shown below [8].

- **`mosquitto_pub`**: Publishes messages to an MQTT topic.
- **`mosquitto_sub`**: Subscribes to an MQTT topic to receive messages.

This gives you peace of mind that the broker is set up correctly and working properly. To ensure the testing process is completed, follow the steps listed below.

1. **Subscriber**: Open the Command Prompt runs as administrator to subscribe to a topic. The Mosquitto CLI command and flags to subscribe to a topic are shown below.

```
mosquitto_sub -h <broker_address> -t <topic>
```

   - **`-h`**: Specifies the hostname or IP address of the broker.
   - **`-t`**: Defines the topic to subscribe to.

   For example, subscribe to the topic home/kitchen/temperature

```
mosquitto_sub -h localhost -t home/kitchen/temperature
```

   This command subscribes to the home/kitchen/temperature topic on the broker running on localhost and listens for available messages. You cannot receive data because there is no publisher at this time. To receive a message, perform the following publisher steps in another Command Prompt.

2. **Publisher**: The syntax to publish a message is shown below.

```
mosquitto_pub -h <broker_address> -t <topic> -m <message>
```

   - **`-h`**: Specifies the hostname or IP address of the broker.

- `-t`: Defines the topic to which the message will be published.
- `-m`: The message content to be published.

In the example, you should publish a message temperature data 27.39°C encoded in JSON data format with the topic home/kitchen/temperature, you would have.

```
mosquitto_pub -h localhost -t home/kitchen/temperature -m
"{"temperature": 27.39, "unit": "°C"}"
```

Now check your subscriber Command Prompt, you should get data as shown.

```
C:\Windows\System32>mosquitto_sub -h localhost -t
home/kitchen/temperature
{temperature: 27.39, unit: °C}
```

The same publishing and subscribing command syntax can be run on the Mosquitto MQTT broker installed in Raspberry Pi.

### 9.3.5 Using Cloud MQTT Broker with MQTTX Desktop Client

A local computer cannot be used as a broker in a production system. One option is to use a cloud MQTT broker. This is an MQTT broker housed in a cloud infrastructure and providing global access to communications between web apps, IoT devices, and other connected systems. Using public MQTT brokers, anyone can connect to, publish, and subscribe to MQTT topics without having to install a local server. The Eclipse Foundation hosts the popular MQTT broker Mosquito Eclipse, which is an example of a public broker that can be used for free.

To connect to the public broker, we chose MQTTX, an open-source desktop MQTT client developed by EMQ, instead of using a command prompt. Compatible with Windows, macOS, and Linux, this tool provides a user-friendly graphical interface for managing MQTT connections and messages. This simplifies the testing and debugging process of MQTT communication and eliminates the need to remember complex command line syntax. With MQTTX, you can easily visualize incoming and outgoing messages, manage multiple connections, and even simulate IoT devices by running multiple clients at once. To get started with MQTTX, follow the instructions listed below [25], [26], [27], [28].

1. **MQTTX Installation**: You can download MQTTX from the MQTTX official website download page (https://mqttx.app/downloads) as shown in **Figure 10**. Choose an installer compatible with your machine's operating system. For Windows with 64-bit architecture, look for vX.XX.X.win64.exe where X.XX.X is the version number.

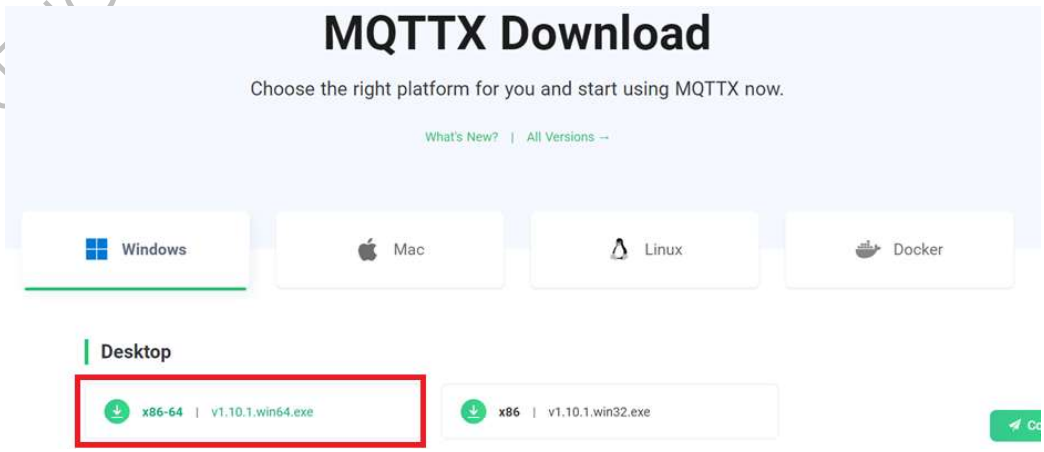



**Figure 10** *Download MQTTX installation setup from its main web page.*

After the download is complete, launch the installation file and follow the simple installation instructions. After installation, you can find MQTTX in the start menu or on your desktop. Double-click the icon to launch the application.

2. **Manage MQTT Connection**: When your MQTTX desktop client is ready, click the **+** icon in the left menu bar to add and configure a new MQTT connection or click on the **+ New Connection** button at the center of the main application page as shown in **Figure 11**.

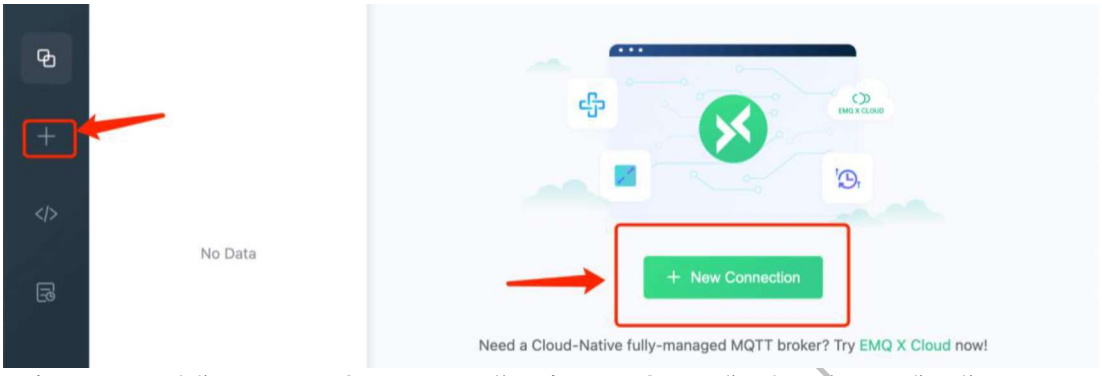



**Figure 11** *Adding new MQTT connection from MQTTX client main application page.*

Once you've navigated the new connection page, you need to configure MQTT broker information to connect a client. For Eclipse Mosquitto public MQTT broker, the connection details are shown below.

**Name**: MosquittoEclipse (or name you want)

**Host**: test.mosquitto.org

**Port**: 1883

**MQTT Version**: 3.1

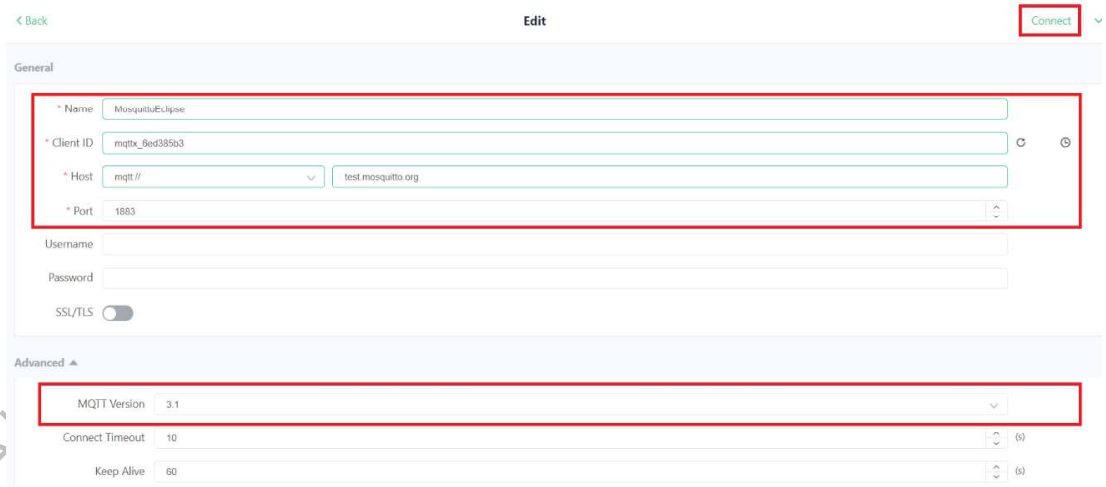
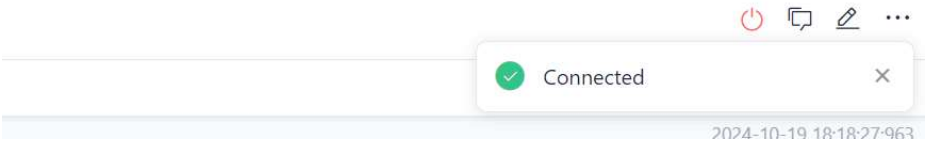



**Figure 12** *Configuring MQTTX connection with Eclipse Mosquitto cloud MQTT broker.*

After completing the configuration, click the Connect button in the upper right corner of the page. If the connection is successful, you should see a pop-up connection status as shown below.




3. **Subscribe and Publish**: Upon successful connection, you'll access the main interface of the connection. Click the **+ New Subscription** button in the upper left corner of the page to add a topic to subscribe to as shown in **Figure 13**.

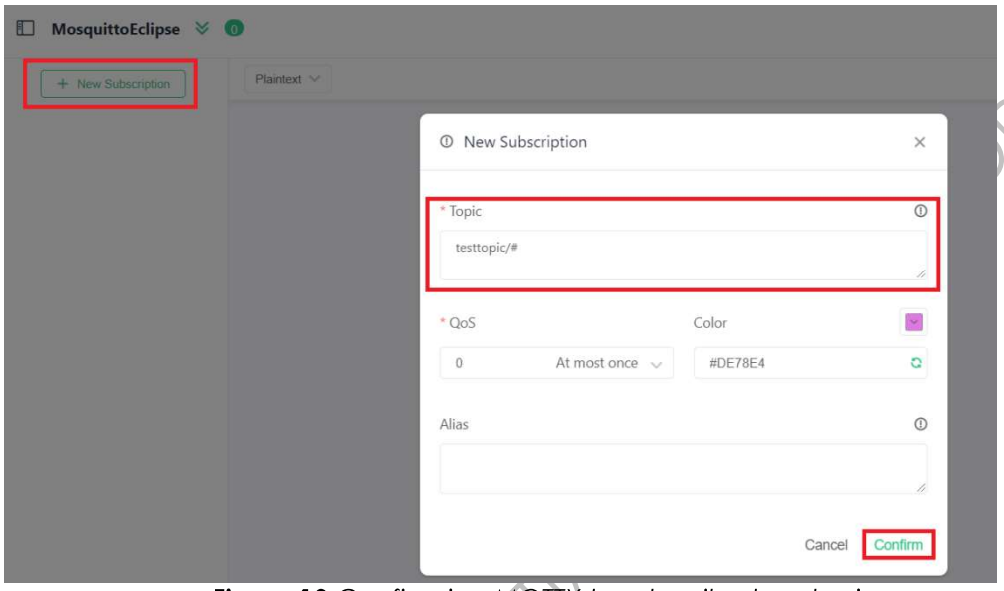



**Figure 13** *Configuring MQTTX to subscribe to a topic.*

By default, the Topic field value is **testtopic/#**, we can change the topic to **home/kitchen/temperature**. When adding a Topic, you can set an Alias for each Topic. This option is optional. Click Confirm to add the subscription.

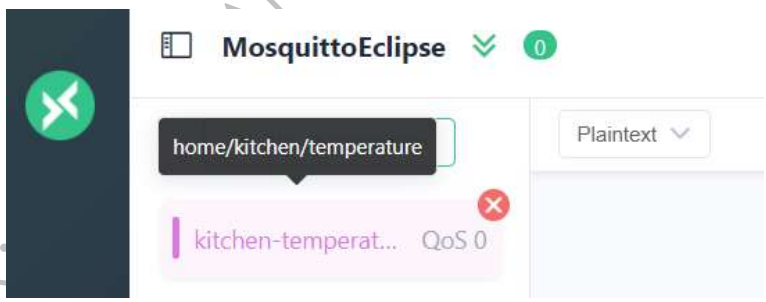



**Figure 14** *A topic can be seen in MQTTX besides text field interface.*

When the subscription is set and added, the Topic in the subscription list will be displayed as an Alias, and hovering over the Topic item will also display the original value of the Topic as shown in **Figure 14**. After the Topic is successfully subscribed, you can test the sending and receiving of messages. Fill in the Topic information you just subscribed to in the lower left corner of the page.

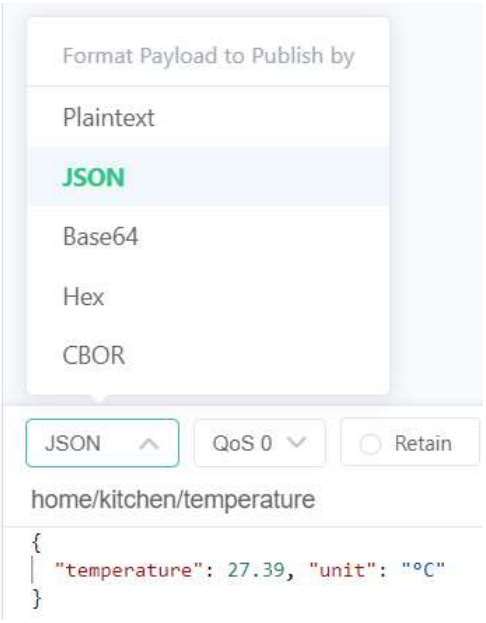



**Figure 15** *MQTTX configuring message payload format as JSON string.*

Change the payload format to JSON and add a message to the text field below the Topic as shown in **Figure 16**.

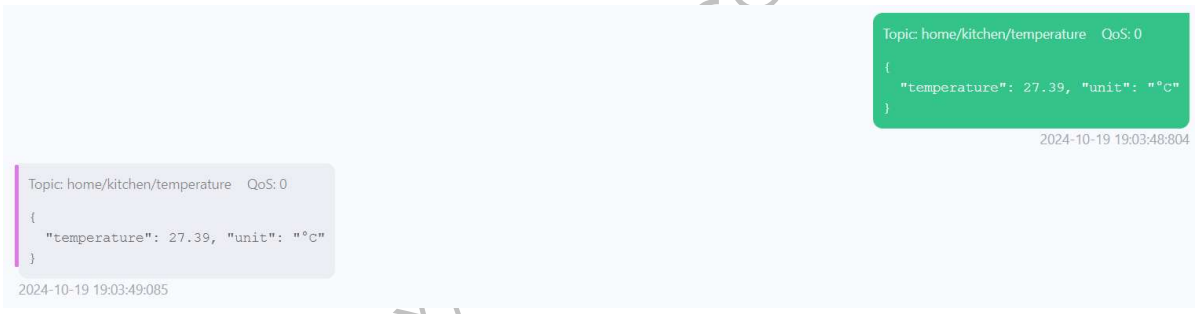



**Figure 16** *MQTTX client text filed showing published and subscribed topic and its payload as JSON string.*

After sending successfully, you can immediately receive the message just sent. Note that in the message box, the right column is the sent message, and the left column is the received message.

## 9.4 Set Up IoT Dashboard with MQTT API

Adafruit IO is an intuitive IoT cloud platform that allows you to store, visualize, and interact with data from connected devices, simplifying the development of IoT applications using existing backend infrastructure. Adafruit IO supports MQTT, an effective solution for connecting IoT devices that enables real-time data communication and facilitates data visualization through customizable dashboards [29].

This guide will walk you through the essential steps to get started with Adafruit IO, creating your first dashboard, and connecting an IoT device using the MQTT API [30], [31].

1. **Create an Adafruit IO Account**: Go to the Adafruit IO homepage (https://io.adafruit.com/). Click the "**Get Started for Free**" and fill in the necessary details to create your Adafruit IO account.

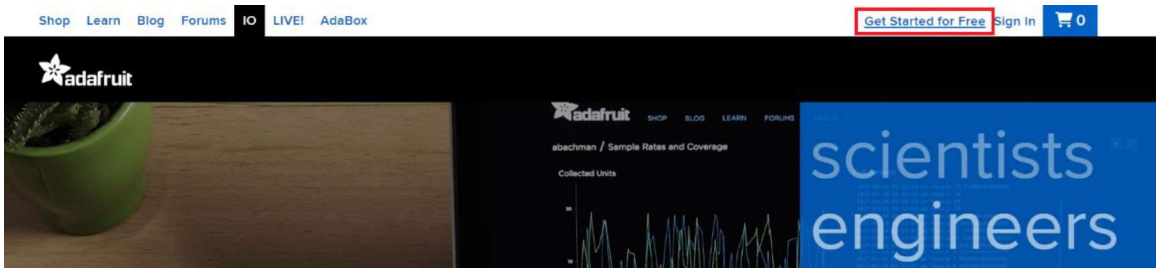



**Figure 17** *Adafruit IO main page [32].*

After signing up, check your email for a verification link from Adafruit. Click the link to verify your account and complete the registration process.

2. **Explore the Adafruit IO Page**: Once registered, navigate to your Adafruit IO page by logging in as shown in **Figure 18**. Familiarize yourself with the Adafruit IO interface such as feeds, dashboards, actions, API key and account settings [33].

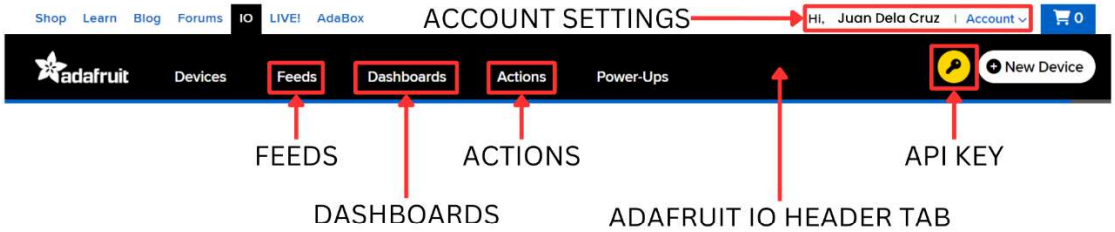



**Figure 18** *Adafruit IO dashboard page and its main interface tab.*

- **Feeds**: serves as containers for storing data and its metadata. Each feed represents a unique data stream, allowing you to manage diverse types of information from various devices.
- **Dashboard**: offer a user-friendly interface to visualize and interact with the data collected from feeds. They are accessible from any modern web browser, enabling real-time monitoring and control of your IoT devices.
- **Actions**: enable you to automate responses based on feed data conditions. They can be triggered by events, allowing for interactive IoT applications.
- **API Key**: Each Adafruit IO account is associated with a unique API key, which is used to authenticate requests made to the Adafruit IO API. This key is essential for connecting your devices to Adafruit IO.
- **Account Settings**: an area that manages your profile information, Adafruit IO account status, plan subscription, dashboard data and its activity.

Understanding the basics of feeds, dashboards, actions, and API keys, you can effectively manage your IoT projects and create applications that respond to real-time data.

3. **Create Your First Feed**: From Adafruit IO main page, click on the **Feeds** on the header to navigate to the feed management page. On the Feeds page, click the "**+ New Feed**" button. A pop-up window will appear to enter details for your new feed as shown in **Figure 19** [34].

    - Name: Enter a unique name for your feed (e.g., home-kitchen-temperature). This name should clearly indicate the data the feed will store, as it will appear in dashboard.

- Description: Optional, a brief description to provide context about the feed will store (e.g., "Temperature readings from the kitchen").

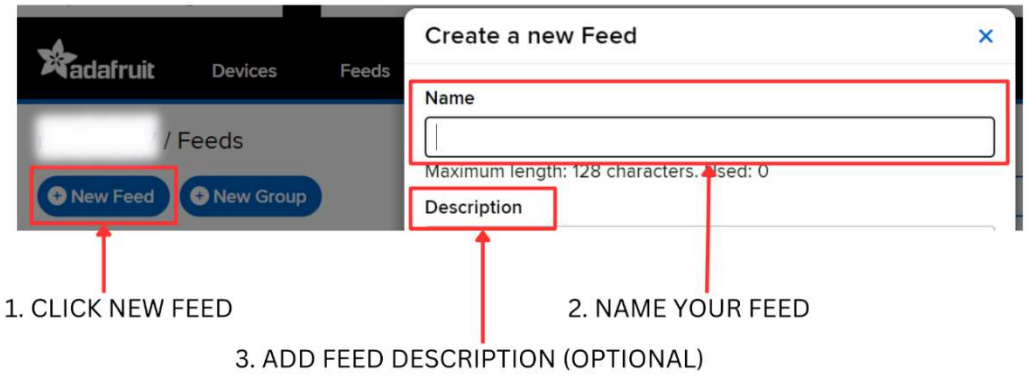



**Figure 19** *Adafruit IO creating new Feed.*

After entering the necessary information, click the "**Create**" button to finalize to add feed creation. Once the feed is created successfully, you will be redirected to the Feeds page, where you can see your newly created feed as shown in **Figure 20**.


**Figure 20** *Adafruit IO created Feed.*

Click on the name of your feed to view its details and settings. Here, you can see the current values stored in the feed, feed information, feed privacy settings, and access additional options like data logging features.

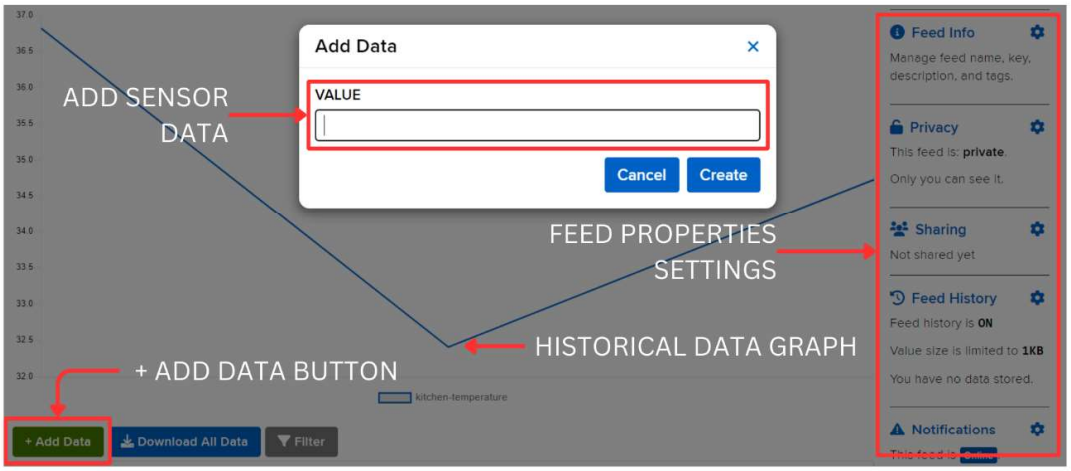



**Figure 21** *Adafruit IO Feed advance settings from dashboard widget's tab.*

Click "**+ Add Data**" button to simulate data storing on your feed. Enter any value of temperature data such as 34.73 and click "**Create**" button to add data. While there is no explicit feed data type selection during creation, Adafruit IO will determine the type based on the data you send. You can view real-time data that the feed collects. This may include historical values if you have already begun sending data via feed management page or to an IoT device.

4. **Create a Dashboard**: Click on the "**Dashboards**" section in the Adafruit IO header tab, then click "**+ New Dashboard**" to create your first dashboard.

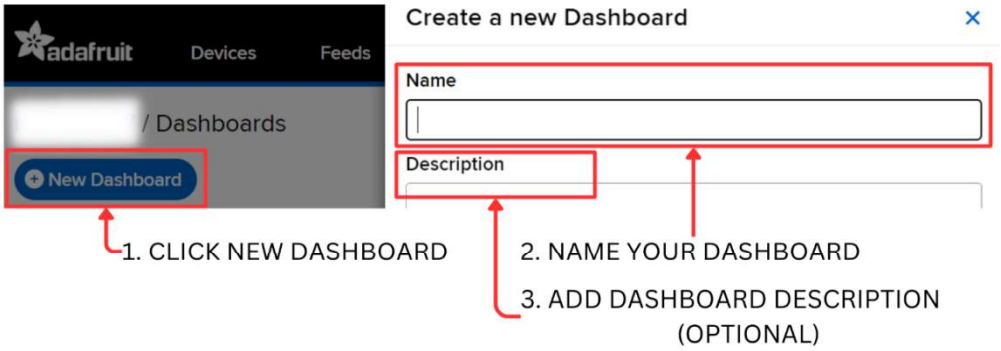



**Figure 22** *Adafruit IO creating a new dashboard.*

A window will pop up, prompting you to enter some basic information for your dashboard.

- **Name**: Enter a descriptive name that identifies your dashboard (e.g., "Home Monitoring").
- **Description**: Optional, add a brief description of the dashboard's purpose (e.g., "Dashboard to monitor home environment").

After entering the necessary information, click the "**Create**" button to finalize your new dashboard. Once the dashboard has been created successfully, click the dashboard you created, and you will be automatically redirected to a blank dashboard with the name you provided as shown below.

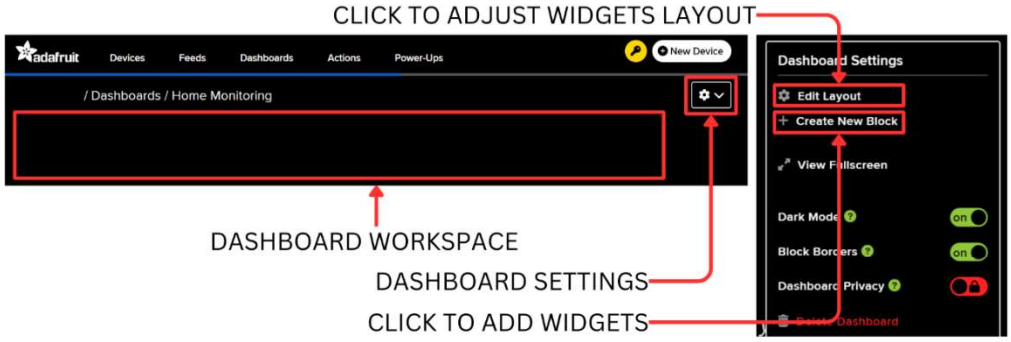



**Figure 23** *Adafruit IO creating a new dashboard.*

To start adding widgets, click the "**Dashboard Settings**" button or the gear icon located in the upper right corner of the dashboard. This will enable an edit mode, allowing you to modify the dashboard layout and add new widgets. While in edit mode, click the "**+ Create New Block**" button on the Dashboard Settings menu. This will open a list of available widgets or block types would like to add to your dashboard.

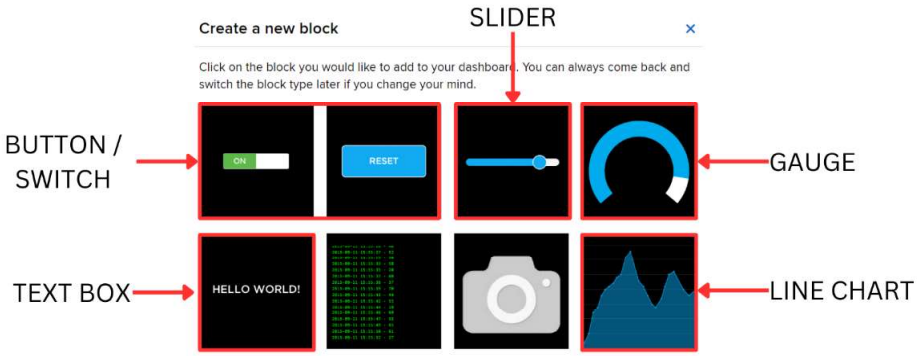



**Figure 24** *Adafruit IO dashboard widgets or data visualization component.*

Select the widget type that best matches the data you want to display or control. Common widget options include:

- **Line Chart**: Displays data trends over time, ideal for tracking temperature, humidity, or other variables.
- **Gauge**: Displays a single value, perfect for real-time measurements like speed or voltage.
- **Slider**: Allows you to set values within a range, useful for brightness, volume, or speed control.
- **Toggle Switch**: Acts as an on/off switch for devices (e.g., turning lights or motors on and off).
- **Text Box**: Shows text data, such as status updates or alerts.

Once you select a widget type, for example a gauge widget, you'll be prompted to link it to a feed you created earlier and customize the gauge settings like widget title, minimum and maximum values to represent the data range, low and high threshold values, decimal places and more. Once done, click "**Create Block**" button to add widget.

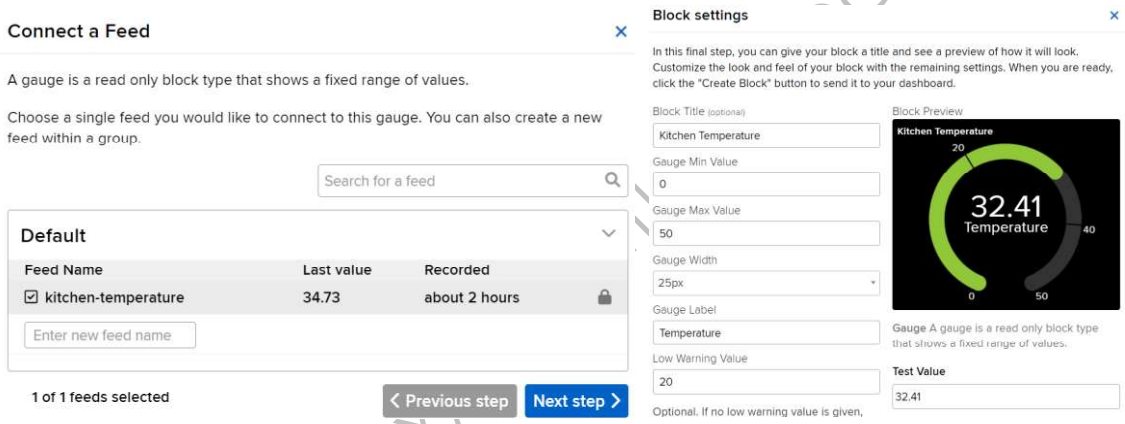



**Figure 25** *Customizing gauge widget and linking it to a Feed.*

Repeat this process to add additional widgets as needed. After adding widgets, you can customize their layout and appearance within the dashboard by clicking the "**Edit Layout**" button in the Dashboard Settings menu. You may drag widgets to reposition them around the dashboard for an organized layout. You can also resize widgets by clicking and dragging the bottom-right corner to adjust their dimensions. Below is the arranged layout of the dashboard's gauge and line chart widgets.

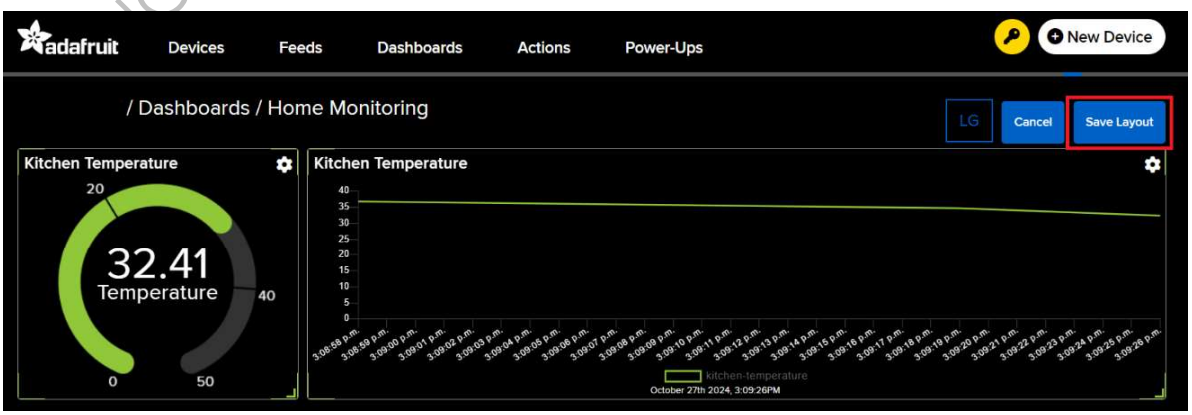



**Figure 26** *Adafruit IO dashboard layout customization and settings.*

After adding and configuring all desired widgets, click the "**Save Layout**" button in the dashboard's top right corner to exit edit mode. You can interact with the widgets you added, view real-time data, and control your devices as needed. The next section will connect your IoT device using the Adafruit MQTT API to communicate with Adafruit IO using the MQTT protocol.

## 9.5 MQTT Connection to IoT Platform Using ESP32

The Adafruit IO MQTT application programming interface (API) allows you to connect your IoT devices to the Adafruit IO platform for data publishing and subscribing. IoT devices can send data to feeds and receive updates using this API, ensuring only authorized devices can communicate with your Adafruit IO account. On the Adafruit IO main page, the API key can be found on the key symbol in the top-right corner, along with the "**+ New Device**" button.


**Figure 27** *Adafruit IO API authentication key.*

A window will appear, and your API key consisting of the username and active key will be displayed. Keep your API key secure and only use it in trusted applications or devices. Do not share it publicly, as anyone with access to this key can interact with your Adafruit IO resources. If you believe your active key has been compromised, you can regenerate it by clicking the "**REGENERATE KEY**" button. This will invalidate the old key, requiring you to update your device code with the new key.

The Adafruit IO API key is used as the password in your MQTT configuration, along with your Adafruit IO username as the MQTT username. To connect an IoT device to Adafruit IO via MQTT, configure the MQTT client with the following connection parameters [31]:

- **Host**: io.adafruit.com
- **Port**:
    - 1883 for unencrypted MQTT connections.
    - 8883 for SSL/TLS encrypted connections (recommended)
- **Username**: Your Adafruit IO username
- **Password**: Your Adafruit IO Active Key typically starts with the prefix "aio_"

Adafruit IO uses a specific topic format that defines where and how data is sent and received within your account. Adafruit IO organizes data within feeds, which act as a container of information. Each feed can hold multiple data points that can be published or subscribed to. The standard topic format for accessing feeds in Adafruit IO is as follows.

<username>/feeds/<feed name or feed key>

Above is the primary topic for publishing or receiving data values for a specific feed. If you use a memory-constrained client such as a microcontroller, a short topic format is provided to save memory.

<username>/f/<feed name or feed key>

For example, if your username is juan_dela_cruz and you have a kitchen-temperature feed, the following topics are valid for accessing the feed.

- juan_dela_cruz/feeds/kitchen-temperature
- juan_dela_cruz/f/kitchen-temperature

Now you know how MQTT topics are structured, next is how Adafruit IO structured the data for publishing and subscribing. The two primary structured data formats supported are JavaScript Object Notation (JSON) and Comma-Separated Values (CSV). To publish JSON data to Adafruit IO, format your data as key-value pairs that must include a value key, with optional keys for latitude (lat), longitude (lon), and elevation (ele). Here's an example of a valid JSON object:

```
{
  "value": 27.39,
  "lat": 14.8582,
  "lon": 120.8142,
  "ele": 9.4
}
```

The JSON data structure can be published using the following topics:

- <username>/feeds/<feed name or feed key>
- <username>/feeds/<feed name or feed key>/json
- <username>/f/<feed name or feed key>
- <username>/f/<feed name or feed key>/json

The use of /json endpoint specifies that you're publishing in JSON format. But both topics will process JSON data, and Adafruit IO will update multiple fields based on the JSON structure. CSV is a simpler format, suitable for handling lists of values in a plain-text format. The CSV format expects the following structure:

value,latitude,longitude,elevation

For instance, to publish the value 27.39 with latitude 14.8582, longitude 120.8142, and elevation 9.4, the following string would be sent to the feed as "27.39,14.8582,120.8142,9.4".

The topic for publishing CSV data uses /csv endpoint.

- <username>/feeds/<feed name or feed key>/csv
- <username>/f/<feed name or feed key>/csv

In this format, Adafruit IO interprets each value separated by commas and can store or process them accordingly. Each format can be used with specific feed endpoints to accommodate diverse IoT application needs. CSV is a simple list of data and lightweight data

transmission. JSON is best for structured data and multi-field payloads. Using JSON, you may send multiple sensor values in one payload.

To send real-time data such as temperature using an IoT device like the ESP32 on the Adafruit IO platform using MQTT, you'll need to configure the device. **Listing *1*** is the provided example code that will demonstrate how to configure the device, including connecting to a Wi-Fi network, establishing MQTT connections, publishing messages, and handling incoming messages [35].

**Listing 1** *ESP32 MQTT Client with Adafruit IO unencrypted connection*

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

These libraries handle Wi-Fi and MQTT connections. The PubSubClient.h library is a lightweight, open-source library designed to support MQTT protocol communication on Arduino-compatible devices, including the ESP32. You need to download the PubSubClient.h library first. Navigate the Arduino library manager from the left tab of IDE. In the search bar, type "**pubsubclient**." Once the search results appear, look for the **PubSubClient** library by **Nick O'Leary** as shown in **Figure 28**. Click the "**INSTALL**" button next to the library to begin the installation process.

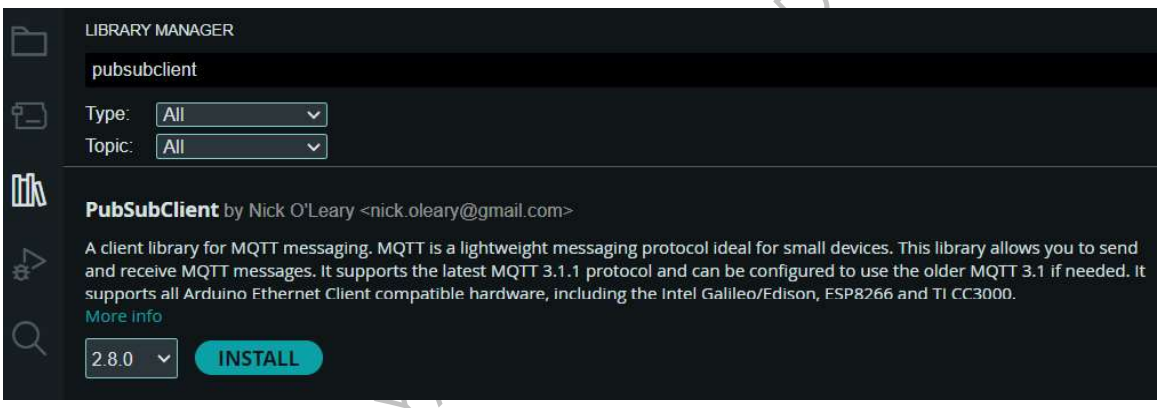



**Figure 28** *Installing PubSubClient MQTT Arduino library by Nick O'Leary.*

```
// Wi-Fi network credentials
const char* wifi_ssid = "REPLACE_WITH_WIFI_SSID";
const char* wifi_auth = "REPLACE_WITH_WIFI_PASSWORD";
```

You need to define settings to connect an ESP32 device to a Wi-Fi network where wifi_ssid and wifi_auth are variables that store the Wi-Fi network name (SSID) and password. The following definitions are used to configure the MQTT connection on Adafruit IO, where data from ESP32 will be published and can be accessed or subscribed to on a dashboard.

```
// Mosquito MQTT connection settings
#define mqtt_name "AdafruitIO"
#define mqtt_host "io.adafruit.com"
#define mqtt_port 8883
#define mqtt_username "aio_username"
#define mqtt_password "aio_key"
#define mqtt_topic " aio_username/f/kitchen-temperature"
#define mqtt_QoS 0
#define mqtt_retain true
#define mqtt_timeout 10000
```

The mqtt_username and mqtt_password are authentication credentials for connecting to the Adafruit IO MQTT broker. Replace "**aio_username**" with your Adafruit IO username, and

"**aio_key**" with your unique Adafruit IO active key, which can be found in your Adafruit IO API key. Here, "**aio_username/f/kitchen-temperature**" specifies a feed for kitchen temperature data under the same username aio_username should be replaced with your Adafruit IO username.

Adafruit IO supports only two QoS levels when publishing data.

- **0 (At Most Once)**: Messages are delivered once, but delivery isn't guaranteed.
- **1 (At Least Once)**: Messages are guaranteed to be delivered at least once, but duplicates may occur.

Here, 0 is chosen for simplicity and efficiency, as it is the most lightweight. When mqtt_retain is set to true, the broker retains the last published message for a topic, allowing new subscribers to receive the last known value immediately upon subscription. This is useful for IoT applications where subscribers need the most recent data right away. The mqtt_timeout defines the maximum time (in milliseconds) to wait for an unsuccessful MQTT connection before retrying. Here, 10000 milliseconds (10 seconds) are set, which can help manage connectivity issues without overwhelming the broker with frequent connection attempts.

```
// Create WiFi and MQTT client objects
WiFiClient wifi_client;
PubSubClient mqtt_client(wifi_client);
```

The following lines create instances of the WiFiClient and PubSubClient classes. The first line creates an instance of the WiFiClient class, called wifi_client. It provides the ESP32 with the ability to establish TCP/IP connections over Wi-Fi, allowing it to communicate with external servers, such as an MQTT broker. The next line creates an instance of the PubSubClient class named mqtt_client and associates it with the wifi_client instance. It provides all the necessary functions to publish and subscribe to MQTT topics and manages the connection to the MQTT broker over the existing Wi-Fi network.

```
// Timer for tracking the last connection attempt
unsigned long int mqtt_last_connection_time_ms = 0;
```

The mqtt_last_connection_time_ms is a variable that stores the time in milliseconds since the last MQTT connection attempt. This timer is used to manage reconnection attempts to the MQTT broker, helping to prevent continuous reconnection attempts and manage resources efficiently.

```
// MQTT callback function that executes upon receiving a subscribed message
void mqtt_client_cb(char* topic, byte* payload, unsigned int length) {
  Serial.print("[MQTT] Subscribe message: [ topic: ");
  Serial.print(topic);
  Serial.print(", payload: ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.print(", size: ");
  Serial.print(length);
  Serial.println(" ]");
}
```

This function serves as a callback for handling messages received on topics that the ESP32 is subscribed to on the MQTT broker. The purpose of this function is to print the received message details to the serial monitor. This is useful for debugging and verifying that the ESP32 is correctly receiving messages from the MQTT broker.

```
// Function to print the connection status to MQTT broker
void mqtt_connection_status() {
  switch (mqtt_client.state()) {
    case -4:
      // the server didn't respond within the keepalive time
      Serial.println("[MQTT] MQTT_CONNECTION_TIMEOUT");
      break;
    case -3:
      // the network connection was broken
      Serial.println("[MQTT] MQTT_CONNECTION_LOST");
      break;
    case -2:
      // the network connection failed
      Serial.println("[MQTT] MQTT_CONNECT_FAILED");
      break;
    case -1:
      // the client is disconnected cleanly
      Serial.println("[MQTT] MQTT_DISCONNECTED");
      break;
    case 0:
      // the client is connected
      Serial.println("[MQTT] MQTT_CONNECTED");
      break;
    case 1:
      // the server doesn't support the requested version of MQTT
      Serial.println("[MQTT] MQTT_CONNECT_BAD_PROTOCOL");
      break;
    case 2:
      // the server rejected the client identifier
      Serial.println("[MQTT] MQTT_CONNECT_BAD_CLIENT_ID");
      break;
    case 3:
      // the server was unable to accept the connection
      Serial.println("[MQTT] MQTT_CONNECT_UNAVAILABLE");
      break;
    case 4:
      // the username/password were rejected
      Serial.println("MQTT_CONNECT_BAD_CREDENTIALS");
      break;
    case 5:
      // the client was not authorized to connect
      Serial.println("[MQTT] MQTT_CONNECT_UNAUTHORIZED");
      break;
  }
}
```

This function checks the current connection status of the MQTT client and prints a relevant message to the serial monitor. It uses a switch statement to interpret the return value of the mqtt_client.state() function, which provides status codes indicating various connection states or errors. Each case corresponds to a specific status code returned by mqtt_client.state(). This is helpful for debugging connectivity issues, as each status message provides specific feedback on why a connection attempt failed or whether the client is successfully connected.

```
void setup() {
  // Initialize serial monitor with baud rate 115200
  Serial.begin(115200);

  // Set ESP32 WiFi to station mode
```

```
  WiFi.mode(WIFI_STA);

  // Initiate the Wi-Fi connection
  WiFi.begin(wifi_ssid, wifi_auth);
  Serial.println("[WIFI] Connecting to WiFi Network...");

  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(100);
  }

  Serial.println("");
  Serial.print("[WIFI] Connected to the Wi-Fi network with local IP: ");
  Serial.println(WiFi.localIP());

  // Configure MQTT server and callback function
  mqtt_client.setServer(mqtt_host, mqtt_port);
  mqtt_client.setCallback(mqtt_client_cb);
```

The setup routine function initializes the ESP32, sets up a Wi-Fi connection, and configures the MQTT client. After initializing the Serial communication for debugging and the device is successfully connected to the Wi-Fi network, it needs to set the MQTT broker's host or IP address and port number, enabling the ESP32 to know where to send and receive MQTT messages. Then specifies mqtt_client_cb as the callback function for handling incoming MQTT messages, which will automatically be triggered whenever a message is received on any subscribed topic.

```
  // Check MQTT broker connection status
  while (!mqtt_client.connected()) {
    // Connect to MQTT broker
    Serial.print("[MQTT] Connecting to MQTT broker...");
    if (mqtt_client.connect(mqtt_name, mqtt_username, mqtt_password)) {
      Serial.println("connection succeeded.");
    } else {
      Serial.println("connection failed!");
    }
    // Print MQTT connection status
    mqtt_connection_status();
  }
  // Subscribe to the topic only once
  mqtt_client.subscribe(mqtt_topic, mqtt_QoS);
}
```

This is the last part of the setup routine function that completes the initial connection to the MQTT broker and sets up a subscription to a specific MQTT topic. It continually attempts to connect to the MQTT broker until a successful connection is established. After each attempt at each connection, the function mqtt_connection_status() is called to print the detailed status, helping debug issues if the connection fails. Then subscribe to an MQTT topic using mqtt_client.subscribe() function to start listening for any messages published on mqtt_topic.

```
void loop() {
  // Check MQTT connection status
  if (!mqtt_client.connected()) {
    if (millis() - mqtt_last_connection_time_ms > mqtt_timeout) {
      mqtt_last_connection_time_ms = millis();
      // Print MQTT connection status
      mqtt_connection_status();
      // Attempt to reconnect
      Serial.print("[MQTT] Reconnecting to MQTT broker...");
```

```
      if (mqtt_client.connect(mqtt_name, mqtt_username, mqtt_password)) {
        Serial.println("connection succeeded.");
        // Reset timer after successful reconnection
        mqtt_last_connection_time_ms = 0;
        // Re-subscribe to a topic upon reconnection
        mqtt_client.subscribe(mqtt_topic, mqtt_QoS);
      } else {
        Serial.println("connection failed!");
      }
      // Print MQTT connection status
      mqtt_connection_status();
    }
```

In the loop routine function, the first half of the code monitors the connection status with the MQTT broker and attempts to reconnect if the connection is lost. If the device is disconnected from the broker for more than 10 seconds, it calls a function that prints the current MQTT client state and tries to reconnect to the MQTT broker. Once the connection succeeds, we need to re-subscribe to a topic that ensures the client will receive messages on the desired topic after reconnecting. This is necessary because subscriptions are typically lost when the connection drops.

```
  } else {
    // Publish temperature data every 10 seconds
    static unsigned long int timer = millis();
    if (millis() - timer >= 10000) {
      timer = millis();

      // Simulate random temperature readings ranges from 20°C - 40°C
      float temperature = random(2000, 4000) / 100.00;

      // Buffer the temperature data as message in JSON format
      const int payload_size = 50;
      char mqtt_payload[payload_size];
      snprintf(mqtt_payload, payload_size, "{\"value\": %.2f}", temperature);

      // Publish message to topic
      mqtt_client.publish(mqtt_topic, mqtt_payload, mqtt_retain);

      // Print the message / payload
      Serial.printf("[MQTT] Publish message: %s \n", mqtt_payload);
    }
    // Handle MQTT client tasks like receiving messages
    mqtt_client.loop();
  }
}
```

This final part of the loop routine function publishes simulated temperature data to the MQTT broker and ensures that the MQTT client processes incoming messages. Using the timer variable allows it to keep track of the elapsed time since the last publish, it checks if 10 seconds have passed. Then it generates a random temperature data range from 20.00°C to 40.00°C that simulates real-world temperature readings. Then the temperature data passes through the JSON-formatted string payload and sends it to the specified MQTT topic. It is important to note that the Adafruit IO MQTT broker imposes a limited rate to prevent excessive load on the service. The current rate limit is at most 30 requests per minute for free accounts. Thus, the publishing rate should be at least every 2 seconds. If you perform too many publishing actions in a short period, then some of the requests will be rejected and an error message will be published.

The last line calls the mqtt_client.loop() function that processes any incoming messages and maintains the connection with the broker. It must be called regularly to ensure that the MQTT client remains responsive and can handle incoming subscriptions, including executing the mqtt_client_cb() function when messages are received.

If you want to use an MQTT encrypted connection through TLS/SSL, you need to make a few adjustments as demonstrated in **Listing 2**.

**Listing 2** *ESP32 MQTT Client with Adafruit IO encrypted connection*

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

You need to include WiFiClientSecure.h library for a secure Wi-Fi client, then update the MQTT port from 1883 to 8883 to specify the secure port.

```
#define mqtt_port 8883
...
WiFiClientSecure wifi_client;
```

Then change the client type instance instead of WiFiClient to WiFiClientSecure which enables SSL/TLS. In the setup routine function, after the Wi-Fi connection add a function wifi_client.setInsecure() to bypass SSL certificate verification when establishing a secure connection for development or testing purposes.

```
void setup() {
 ...
  // Allows for 'insecure' connections. It turns off CA certificate checking
  wifi_client.setInsecure();
```

For best practice, you should either set the valid SSL certificate or use the correct SSL fingerprint to ensure that your IoT device is communicating with the intended server and not a malicious MQTT broker. Moreover, it can protect you from man-in-the-middle attacks (MITM), where a hacker could intercept the data being sent between your ESP32 and MQTT broker because the connection is not secured against a fake server [36].

## 9.6 Mini-Project 8: Scalable Environmental Monitoring and Control System Using MQTT and Adafruit IO

### 9.6.1 Introduction

To complete the theoretical and practical insights you have gained from this chapter, this project will empower you to apply your learned lessons in practice, bridging the basic principles of MQTT with the real-world design of IoT systems. You will implement a scalable environmental monitoring and control system using an ESP32 microcontroller as a sensor node, interfaced with a variety of sensors. The node establishes a secure MQTT connection to the Adafruit IO cloud platform, publishes sensor data in JSON format, and subscribes to topics for remote actions, such as turning the LED on and off or the relay on and off, when the user switches the widget on the dashboard. In addition, you configure the Adafruit IO interactive dashboard, which includes visualization widgets such as meters, line charts, and switch buttons to allow you to monitor and control your data in real time.

By addressing potential complexities, integrating notifications you need to configure Adafruit IO's feed action mechanisms, which defines conditional alerts, such as e-mail or the dashboard widget notification, which is activated when sensor data reaches critical thresholds. Additionally, incorporating a status widget on the dashboard elevates system

availability, requiring you to use the MQTT's Last Will and Testament feature to publish device connectivity status if online, or offline to a dedicated feed, which you can then visualize using the Adafruit IO status widget.

Upon completing this project, you will be able to:

- Implement bidirectional MQTT communication by programming the ESP32 to publish sensor data in structured formats such as JSON to designated feeds on Adafruit IO at predefined intervals.
- Subscribe to command topics for remote actuation, such as toggling relays based on button toggle widget status, or feed notifications and reactive action based on data threshold conditions.
- Design and deploy an interactive IoT dashboard on Adafruit IO, incorporating visualization widgets, and button toggle widgets to control the actuators, as well as status widget for device connectivity, and actuator status.
- Incorporate Adafruit IO advanced features such as notifications and feed actions by configuring triggers to send alerts via email when sensor readings exceed specified thresholds or automate responses through subscribed topics.
- Incorporate device status management using MQTT's Last Will and Testament feature to publish connectivity status to a dedicated feed, visualized via status widgets or text widget on the dashboard.

## 9.6.2 Components Needed

Hardware Components:

1. LILYGO TTGO LoRa32 OLED ESP32 Development Board
2. Digital Temp & Humidity Sensor (AM2302/DHT22)
3. Capacitive Soil Moisture Sensor
4. Light Intensity Sensor (Photoresistor (LDR) GL5528)
5. OLED Display (SSD1306 128x64 pixels)
6. LED Indicator
7. Jumper Wires
8. Power Source (Battery or USB power)

Software Tools:

1. Arduino IDE
2. Mosquitto MQTT Broker
3. MQTTX MQTT Desktop Client
4. Adafruit IO

## 9.6.3 System Architecture

In this mini project you will design and implement an MQTT-based IoT architecture using a lightweight publish-subscribe model to allow efficient and scalable communication between a sensor node based on ESP32 microcontroller and the Adafruit IO cloud platform as shown in **Figure 29**. The architecture consists of three core components: a sensor node, an MQTT broker, and an IoT cloud dashboard. The ESP32, which is interfaced with various sensors, is a data acquisition unit that collects environmental data and publishes it in JSON format to specific topics on the Adafruit IO MQTT server over a secure Wi-Fi connection using credentials such as username and authentication keys. At the same time, ESP32 subscribes to command topics, which allow remote control of devices such as LEDs to represent actuators that control environmental parameters, such as the ventilation system, when the thresholds are exceeded, demonstrating the bidirectional communication efficiency of MQTT over HTTP.

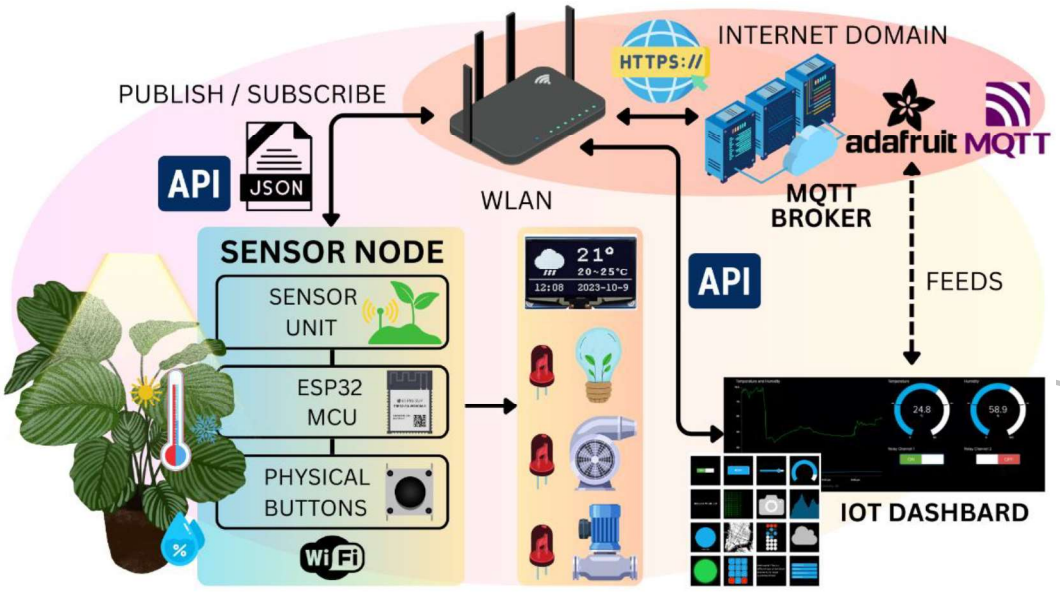



**Figure 29** *IoT System architecture for scalable environmental monitoring and control using MQTT and Adafruit IO IoT platform.*

MQTT broker is a cloud service of Adafruit IO, acting as a central hub, managing the routing, storage and quality of service level of the service to ensure reliable data delivery in the face of network instability. You configure the broker to support remote access and implement the Last Will and Testament feature of MQTT to publish the status of device connection, which is one of the important features when deploying IoT devices on the field. The Adafruit IO platform provides an interactive dashboard in which you can integrate visualization widgets such as meters, line graphs, switches and status indicators to monitor the real-time data from sensors and the status of the device, as well as trigger alerts based on predefined threshold conditions, such as if the actual temperature exceeds 30°C. This architecture not only improves your understanding of MQTT API integration and low bandwidth optimization but also challenges you to deal with complexity such as secure payload processing and automatic feed processing and prepares you for advanced IoT applications from environmental monitoring into smart homes, and industrial automation.

## 9.6.4 Project Plan

You must complete the following tasks to meet the project objectives.

- Use your Adafruit IO account and obtain your key from the dashboard settings to authenticate MQTT connections.
- Configure the ESP32 to connect to your Wi-Fi network by updating the Arduino sketch with your network's SSID and password. In the sketch, include the PubSubClient library.
- Set the MQTT server to io.adafruit.com with port 8883, using your Adafruit IO username and authentication key.
- Create feeds on Adafruit IO for sensor data, actuator control flag status, and device status with appropriate data format.
- Program the ESP32 to publish sensor data in JSON format to the respective feeds every time interval, and subscribe to the control feed without a delay, to toggle the actuator status based on received commands.
- Implement MQTT's Last Will and Testament feature in the sketch to publish an "offline" message to the status feed if the ESP32 disconnects unexpectedly. When back to online, publish a message "online" to the status feed. Test the functionality using a local Mosquitto broker first, then transition to Adafruit IO.

- You will design an interactive dashboard on Adafruit IO to visualize and control the IoT system. Access the Adafruit IO dashboard interface and create a new dashboard, adding widgets such as a gauge for temperature, a line chart for humidity trends, a toggle button for actuator control, and an indicator widget for device status.
- Link each widget to the corresponding feed source, layout the dashboard using the widgets and configure widget settings such as value ranges for optimal visualization. Test the dashboard by publishing sample data from the ESP32 and verifying real-time updates on the widgets. Adjust widget properties to enhance readability, such as proper labels, thresholds, and color-coded alerts on the gauge.
- Configure feed notifications and reactive actions on Adafruit IO. Set up a trigger in the Adafruit IO dashboard to send an email notification when a sensor data feed exceeds critical threshold, specifying the condition and recipient details. For automation, create a feed action that publishes a command MQTT message to the actuators feed when the sensor data threshold is met, enabling the ESP32 to activate the actuator for a simulated environmental condition.
- Ensure the subscription callback function parses incoming commands correctly and implements error handling to manage invalid payloads or network delays. Use QoS level 1 to ensure at-least-once message delivery for critical actions.
- Test the notifications and actions by simulating sensor critical readings, verifying that alerts are received and the reactive actions toggles command flag status as expected.

## 9.6.5 Guidelines and Tips

- You can subscribe to the project's feeds using the MQTTX Desktop Client and publish test messages to verify that ESP32 is receiving and processing commands, such as switching the status of an actuator, or Adafruit IO dashboard displaying data properly.
- You should implement secure MQTT communication. To communicate with Adafruit IO MQTT broker securely, make sure that SSL and TLS encryption is enabled by using port 8883 instead of 1883, and modify the sketch to include the WiFiclientSecure library to enable SSL and TLS for PubSubClient.
- Add a reconnection function that attempts to reconnect to the Wi-Fi network and MQTT broker if the connection is lost.
- When configuring the Adafruit IO notifications and feed actions, you should perform thorough testing to ensure that they are reliable under different conditions. Create multiple trigger scenarios, such as temperature above 30 degrees Celsius or humidity below 40 percent and verify that the notifications are delivered quickly by e-mail or automatic Adafruit IO actions.
- You should organize your Adafruit IO feeds using a hierarchical structure of topics. Create feeds for different sensor data, actuator command flags, and device status with clear labels or descriptive names.
- Group related feeds under a common category, such as “environmental sensor data” or “actuator command flag” in the Adafruit IO dashboard for better organization, to reduce the number of requests needed by your device, and to enable single MQTT topic publishing/subscribing to multiple feeds. You can then add the feeds to the group by navigating to the feed page, choosing “Create New Feed”, or by moving the existing feeds to the group.
- Adafruit IO can integrate with third party services such as IFTTT or Discord using Webhooks. It has two types of Webhook functionality to connect to external platforms. Feed webhooks receive data from other services and automatically publish it to the Adafruit IO feed, and reactive webhooks that, if a certain condition is met, will publish the Adafruit IO message to other services.

## References


[1] "MQTT - The Standard for IoT Messaging," MQTT Org. Accessed: Oct. 08, 2024. [Online]. Available: https://mqtt.org/

[2] "MQTT - IBM Documentation," IBM Documentation. Accessed: Oct. 08, 2024. [Online]. Available: https://www.ibm.com/docs/en/integration-bus/10.0?topic=applications-mqtt

[3] M. Yuan, "Getting to know MQTT," IBM Developer. Accessed: Oct. 08, 2024. [Online]. Available: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/

[4] T. Prokosch, "MQTT Protocol Explained: The Complete Guide," Cedalo. Accessed: Aug. 30, 2025. [Online]. Available: https://cedalo.com/blog/complete-mqtt-protocol-guide/

[5] I. Craggs, "MQTT Vs. HTTP for IoT," HiveMQ. Accessed: Oct. 09, 2024. [Online]. Available: https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/

[6] The ThingsBoard Authors, "MQTT protocol | MQTT Broker," ThingsBoard. Accessed: Aug. 30, 2025. [Online]. Available: https://thingsboard.io/docs/mqtt-broker/user-guide/mqtt-protocol/

[7] G. Bello, "What Is HTTP?," Postman Blog. Accessed: Aug. 30, 2025. [Online]. Available: https://blog.postman.com/what-is-http/

[8] Z. Yung, "MQTT Publish/Subscribe with Mosquitto Pub/Sub Examples," Cedalo. Accessed: Aug. 30, 2025. [Online]. Available: https://cedalo.com/blog/mqtt-subscribe-publish-mosquitto-pub-sub-example/

[9] "Connection management in HTTP/1.x," Mozilla. Accessed: Aug. 30, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Connection_management_in_HTTP_1.x

[10] IBM Documentation, "Message Persistence in MQTT Clients," IBM Corporation. Accessed: Aug. 30, 2025. [Online]. Available: https://www.ibm.com/docs/en/ibm-mq/9.3.x?topic=concepts-message-persistence-in-mqtt-clients

[11] The ThingsBoard Authors, "Keep Alive | MQTT Broker," ThingsBoard. Accessed: Aug. 30, 2025. [Online]. Available: https://thingsboard.io/docs/mqtt-broker/user-guide/keep-alive/

[12] The ThingsBoard Authors, "Retained Messages | MQTT Broker," ThingsBoard. Accessed: Aug. 30, 2025. [Online]. Available: https://thingsboard.io/docs/mqtt-broker/user-guide/retained-messages/

[13] KaaIoT, "MQTT vs HTTP for IoT: Detailed Protocol Comparison," KaaIoT. Accessed: Oct. 09, 2024. [Online]. Available: https://www.iotforall.com/mqtt-vs-http-for-iot-detailed-protocol-comparison

[14] HiveMQ Team, "TLS/SSL - MQTT Security Fundamentals," HiveMQ. Accessed: Aug. 30, 2025. [Online]. Available: https://www.hivemq.com/blog/mqtt-security-fundamentals-tls-ssl/

[15] MQTTX Team, "Get Started - MQTTX Documentation," EMQ Technologies Inc. Accessed: Oct. 19, 2024. [Online]. Available: https://mqttx.app/docs/get-started

[16] O. Rusnak, "MQTT Connection Beginner's Guide — How it Works," Cedalo. Accessed: Oct. 10, 2024. [Online]. Available: https://cedalo.com/blog/mqtt-connection-beginners-guide/

[17] L. Dallinger, "HTTP vs MQTT: Choose the Best One for an IoT Project," Cedalo. Accessed: Oct. 10, 2024. [Online]. Available: https://cedalo.com/blog/http-vs-mqtt-for-iot/

[18] S. Orlivskyi, "Mosquitto MQTT Broker on Windows Installation - Ultimate Guide," Cedalo. Accessed: Aug. 30, 2025. [Online]. Available: https://cedalo.com/blog/how-to-install-mosquitto-mqtt-broker-on-windows/

[19] A. Singh, "Raspberry Pi 4 Pinout, Specifications and Applications," Hacktronic. Accessed: Aug. 30, 2025. [Online]. Available: https://www.hackatronic.com/raspberry-pi-4-specifications-pin-diagram-and-description/

[20] "Getting started - Raspberry Pi Documentation," Raspberry Pi. Accessed: Aug. 30, 2025. [Online]. Available: https://www.raspberrypi.com/documentation/computers/getting-started.html

[21] M. Trovò, "How to Set Up and Run the Mosquitto MQTT Broker on Raspberry Pi," Cedalo. Accessed: Aug. 30, 2025. [Online]. Available: https://cedalo.com/blog/mqtt-broker-raspberry-pi-installation-guide/

[22] R. Light, "mosquitto_sub man page | Eclipse Mosquitto," Eclipse Foundation. Accessed: Oct. 19, 2024. [Online]. Available: https://mosquitto.org/man/mosquitto_sub-1.html

[23] R. Light, "mosquitto_pub man page | Eclipse Mosquitto," Eclipse Foundation. Accessed: Oct. 19, 2024. [Online]. Available: https://mosquitto.org/man/mosquitto_pub-1.html

[24] R. Light, "Mosquitto man page | Eclipse Mosquitto," Eclipse Foundation. Accessed: Oct. 19, 2024. [Online]. Available: https://mosquitto.org/man/mosquitto-8.html#

[25] MQTTX Team, "Installation - MQTTX Documentation," EMQ Technologies Inc. Accessed: Aug. 30, 2025. [Online]. Available: https://mqttx.app/docs/downloading-and-installation

[26] MQTTX Team, "Advanced - MQTTX Documentation," EMQ Technologies Inc. Accessed: Aug. 30, 2025. [Online]. Available: https://mqttx.app/docs/advanced

[27] MQTTX Team, "Get Started - MQTTX Documentation," EMQ Technologies Inc. Accessed: Aug. 30, 2025. [Online]. Available: https://mqttx.app/docs/get-started

[28] "MQTTX: Your All-in-one MQTT Client Toolbox," EMQ Technologies Inc. Accessed: Oct. 19, 2024. [Online]. Available: https://mqttx.app/

[29] B. Rubell, "Overview | Welcome to Adafruit IO | Adafruit Learning System," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://learn.adafruit.com/welcome-to-adafruit-io

[30] J. Cooper, "Getting Started | Adafruit IO | Adafruit Learning System," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://learn.adafruit.com/adafruit-io/getting-started

[31] J. Cooper, "MQTT API | Adafruit IO | Adafruit Learning System," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://learn.adafruit.com/adafruit-io/mqtt-api

[32] "Welcome to Adafruit IO," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://io.adafruit.com/

[33] T. Treece and B. Rubell, "Overview | Adafruit IO Basics: Dashboards | Adafruit Learning System," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://learn.adafruit.com/adafruit-io-basics-dashboards/overview

[34] T. Treece, "Overview | Adafruit IO Basics: Feeds | Adafruit Learning System," Adafruit IO. Accessed: Aug. 30, 2025. [Online]. Available: https://learn.adafruit.com/adafruit-io-basics-feeds?view=all

[35] D. Tao, "MQTT on ESP32: A Beginner's Guide | EMQ," EMQ Technologies Inc. Accessed: Oct. 25, 2024. [Online]. Available: https://www.emqx.com/en/blog/esp32-connects-to-the-free-public-mqtt-broker

[36] H. Fereidouni, O. Fadeitcheva, and | Mehdi Zalai, "IoT and Man-in-the-Middle Attacks," *Security and Privacy*, vol. 8, no. 2, p. e70016, Mar. 2025, doi: 10.1002/SPY2.70016.