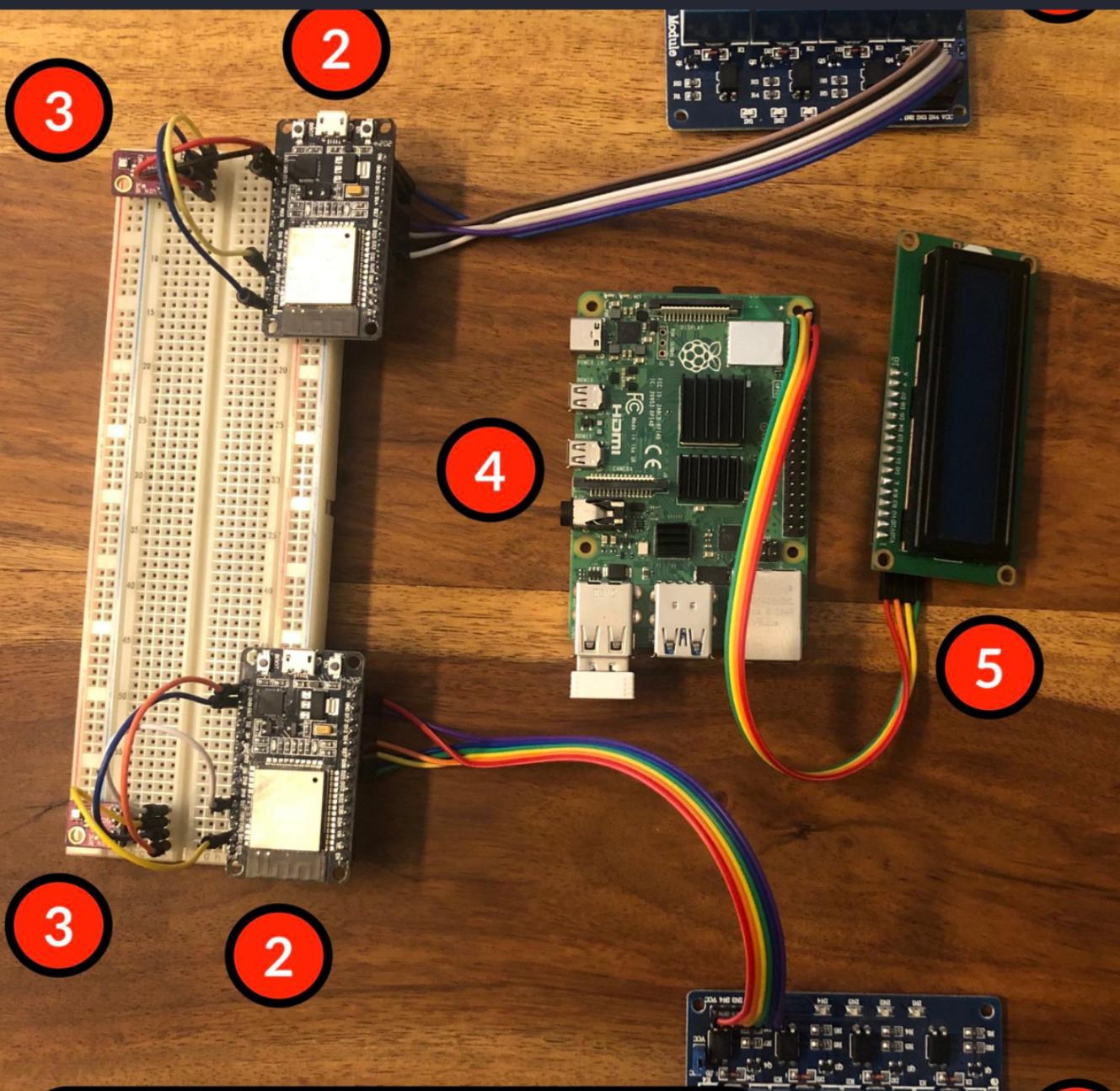


Un trittico vincente

ESP32, RASPBERRY PI E EMQ X EDGE



1. Modulo da quattro relè
2. ESP32 Dev Kit
3. GY-BME 280
4. Raspberry Pi 4 Model B+ 8GByte
5. Display LCD 16x2

ANTONIO MUSARRA

Table of Contents

Informazioni su questo eBook	1
1. Introduzione	2
2. Descrizione della soluzione IoT	3
3. Overview sull'architettura della soluzione IoT	6
3.1. I componenti hardware	6
3.2. I componenti software (principali)	8
3.3. Schema dell'architettura	9
3.4. Schema elettrico della soluzione IoT	10
3.5. Implementazione del prototipo	11
4. Installazione di EMQ X Edge su Raspberry Pi	14
4.1. Configurazione utenze e Access Control List (ACL)	16
5. Configurazione dei topic, QoS e struttura dei payload	20
5.1. Topic esp32/command	20
5.2. Topic esp32/telemetry_data	21
5.3. Topic esp32/releay_{id}_status	22
5.4. Qualche nota sul QoS	22
5.5. Struttura dei payload	24
6. Come programmare l'ESP32	27
7. Scrittura del software per l'ESP32	29
7.1. Compilazione del progetto	42
7.2. Upload del software sul device ESP32	46
8. Primo test d'integrazione	49
9. Scrittura del software per il display LCD	51
10. Implementazione della Dashboard	55
11. Conclusioni	70
12. Risorse	71

Informazioni su questo eBook

Antonio Musarra, Un trittico vincente: ESP32, Raspberry Pi e EMQ X Edge

Copertina e impaginazione di Antonio Musarra

Edizione digitale Luglio 2024 (v1.0.0)

Serie: Elettronica&Informatica (#elettronica-informatica)

Promosso da: Antonio Musarra's Blog (<https://www.dontesta.it>)

Profilo LinkedIn <https://www.linkedin.com/in/amusarra/>

Il progetto di esempio realizzato per quest'opera è disponibile sul repository GitHub all'indirizzo <https://github.com/amusarra/esp32-mqtt-publish-subscribe>

Quest'opera è la versione eBook dell'articolo pubblicato sul blog di Antonio Musarra all'indirizzo <https://www.dontesta.it/en/2021/04/13/esp32-raspberry-pi-emqx-edge-mqtt-publish-subscribe/> nel mese di Aprile 2021.

Quest'opera è stata realizzata usando l'approccio doc-as-code e il testo è stato scritto in formato AsciiDoc.

Nel caso di errori, segnalazioni o suggerimenti, si prega di aprire una issue o discussione sul repository GitHub.

Progetto Smart Card Contactless Raspberry Pi

1. Apertura issue <https://github.com/amusarra/esp32-mqtt-publish-subscribe/issues>
2. Apertura discussione <https://github.com/amusarra/esp32-mqtt-publish-subscribe/discussions>

Nell'ambito del social coding e del contributo alla comunità, è possibile contribuire al progetto con una pull request per migliorare questa guida e il progetto di esempio.

Note sul Copyright

Tutti i diritti d'autore e connessi sulla presente opera appartengono all'autore Antonio Musarra. Per volontà dell'autore quest'opera è rilasciata nei termini della licenza Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International il cui testo integrale è disponibile alla pagina web <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.it>

Tutti i marchi riportati appartengono ai legittimi proprietari; marchi di terzi, nomi di prodotti, nomi commerciali, nomi corporativi e società citati possono essere marchi di proprietà dei rispettivi titolari o marchi registrati d'altre società e sono stati utilizzati a puro scopo esplicativo e a beneficio del possessore, senza alcun fine di violazione dei diritti di copyright vigenti.

1. Introduzione

Così come per realizzare un buon piatto abbiamo bisogno dei giusti ingredienti, anche per costruire un buon progetto hardware e software, è importante combinare in modo adeguato i vari elementi.

Con l'avvento dell'**Internet OfThings** (o **IoT**), il numero di dispositivi e delle apparecchiature connesse alla rete sta aumentando in maniera vertiginosa, basti pensare ai sensori per le automobili, elettrodomestici, telecamere o anche quelli utilizzati per il rilevamento dei parametri vitali.

Ormai l'acronimo IoT e il suo significato sta entrando sempre di più nell'uso comune e con questo articolo propongo lo sviluppo base di un progetto completo dal punto di vista hardware e software che implementa una delle più classiche necessità del mondo IoT, ovvero, acquisire dati da sensori distribuiti in campo, elaborare questi dati localmente e poi eventualmente distribuire quest'ultimi nel cloud.

Per lo sviluppo del progetto è necessario toccare un numero considerevole di argomenti e cercherò di trattarli tutti con il giusto livello di profondità. Darò per scontato che abbiate la conoscenza di alcuni di essi, in caso contrario l'articolo assumerebbe delle dimensioni considerevoli. Mi impegnerò a lasciare tutti i riferimenti utili per ogni vostro approfondimento.

A questo punto direi d'iniziare, mettervi comodi e munitevi della giusta concentrazione perché la lettura di questo articolo sarà abbastanza impegnativa ma spero interessante.

2. Descrizione della soluzione IoT

Lo schema generale di un sistema IoT prevede gli elementi brevemente descritti e indicati ad seguire.

- **Producer:** normalmente rappresentati da sensori o device. Questi devono essere connessi al resto del sistema utilizzando protocolli standard come HTTP, **CoAP** (Constrained Application Protocol), **MQTT** (Message Queuing Telemetry Transport) e **AMQP** (Advanced Message Queuing Protocol).
- **Gateway:**
 - deve acquisire i dati che provengono dai sensori o dai device; deve quindi "**parlare**" la stessa lingua di quest'ultimi;
 - deve essere capace di acquisire un grande volume di dati in tempo reale;
 - deve filtrare i dati "**cattivi**", per esempio provenienti da sensori rotti;
 - deve se richiesto arricchire i dati (data enriching) e trasformarli;
 - deve controllare l'identità dei dati per accertarsi che effettivamente siano di sua competenza;
 - deve essere capace di riconoscere dati duplicati.
- **Processing:**
 - è il cuore di una soluzione IoT;
 - i dati provenienti dai producer attraverso il loro gateway devono essere elaborati e in molti casi in tempo reale;
 - il software di elaborazione deve fornire un set di API per eseguire pulizia, aggregazione, combinazione, analisi in tempo reale, etc.
- **Storage** (ulteriore elaborazione):
 - i dati elaborati vengono archiviati per la presentazione (su dashboard) o ulteriori elaborazioni come analisi batch da parte di analista di dati, apprendimento automatico, indicizzazione, etc.;
 - i requisiti chiave sono l'archiviazione di grandi volumi di dati e la fornitura di un throughput elevato. In generale, i dati vengono archiviati in grandi database distribuiti e utilizzati da strumenti di elaborazione batch.

In questo eBook vi mostrerò una soluzione molto ridimensionata (rispetto a quanto sopra descritto) ma che comunque toccherà le basi di un sistema IoT. Per restare in un ambito più familiare, prendiamo in considerazione il classico scenario della stazione meteorologica.

In campo abbiamo disseminato alcuni sensori che acquisiscono dati ambientali come temperatura, umidità e pressione atmosferica. Questi sensori sono connessi a un **microcontrollore** (o **MCU** che sta per Micro Controller Unit), responsabile della lettura dei valori ambientali e del successivo invio di questi al **Gateway**. All'MCU sono inoltre collegati degli attuatori che possono essere azionati per esempio nel momento in cui viene superata la soglia impostata su uno o più dei parametri ambientali.

Il diagramma a seguire mostra il flusso delle informazioni che parte dal sensore fino ad arrivare alla dashboard, dove questi dati saranno visualizzati. **Environmental Sensors** e MCU, insieme costituiscono il device, ovvero, il **Producer**.

Com'è possibile vedere dal diagramma, il Gateway è in grado d'inviare un comando verso il device (step 5) con lo scopo di scatenare un'azione sull'attuatore connesso all'MCU. **Questo flusso d'informazioni come fa ad arrivare fino alla Dashboard?**

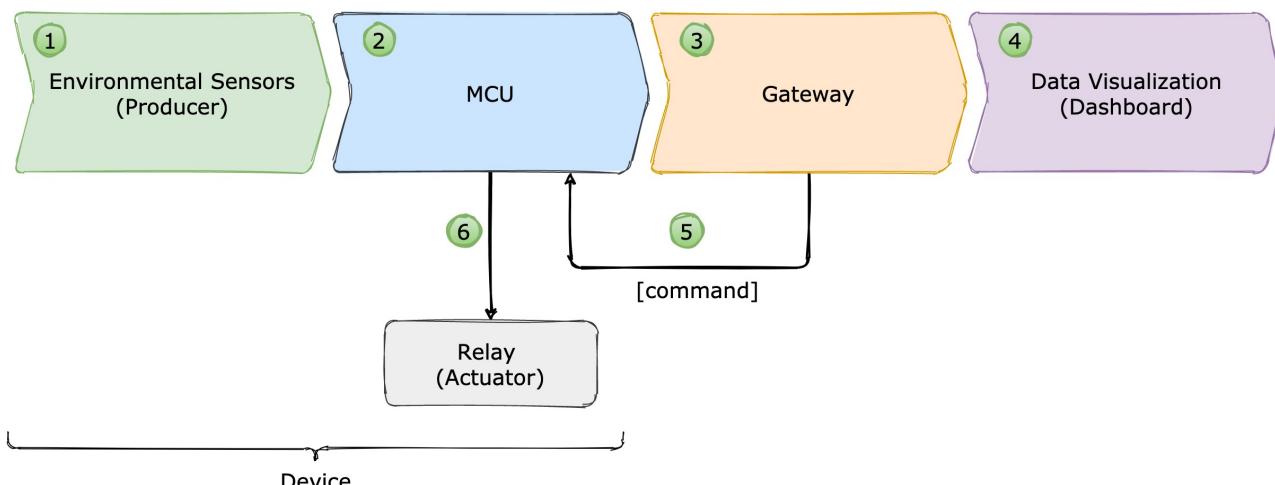


Figura 1 - Flusso dei dati della soluzione IoT proposta

In un sistema IoT, Producer e Gateway devono parlare la stessa lingua e nel caso della nostra soluzione la lingua adottata è il protocollo **MQTT**. Grazie a questo protocollo che i vari componenti sono in grado di colloquiare tra loro, Dashboard compresa. L'adozione di questo protocollo implica che la soluzione adotti quello che viene chiamato generalmente **MQTT Message Broker**. Questo è un componente software centrale che all'interno di un sistema IoT risiede generalmente all'interno del Gateway.

Il Message Broker e nel caso specifico l'MQTT Broker, utilizza lo stile architettonico chiamato generalmente **Publish/Subscribe**. Cerchiamo di comprendere le basi di questo modello. Nel pattern di Publish/Subscribe, un client che pubblica un messaggio viene disaccoppiato dall'altro client o client che ricevono il messaggio. I client non sanno dell'esistenza degli altri client. Un client può pubblicare messaggi di un tipo specifico e solo i client interessati a tipi specifici di messaggi riceveranno i messaggi pubblicati.

Tutti i client stabiliscono una connessione con il broker. Il client che invia un messaggio tramite il broker è noto come **publisher**. Il broker filtra i messaggi in arrivo e li distribuisce ai client interessati al tipo di messaggi ricevuti. I client che si registrano al broker come interessati a tipi specifici di messaggi sono noti come **subscriber**. Pertanto, sia i publisher sia i subscriber stabiliscono una connessione con il broker. Il **topic** è un canale logico denominato ed è indicato anche come canale o soggetto. Il broker invierà ai publisher solo i messaggi pubblicati sui topic a cui sono iscritti.

È facile capire come funzionano le cose con un semplice diagramma, e quello a seguire mostra due device, una dashboard e un mobile phone. In questo caso e nel contesto del pattern Publish/Subscribe, questi componenti assumono il ruolo sia di publisher sia di subscriber e il broker è il dispatcher dei messaggi.

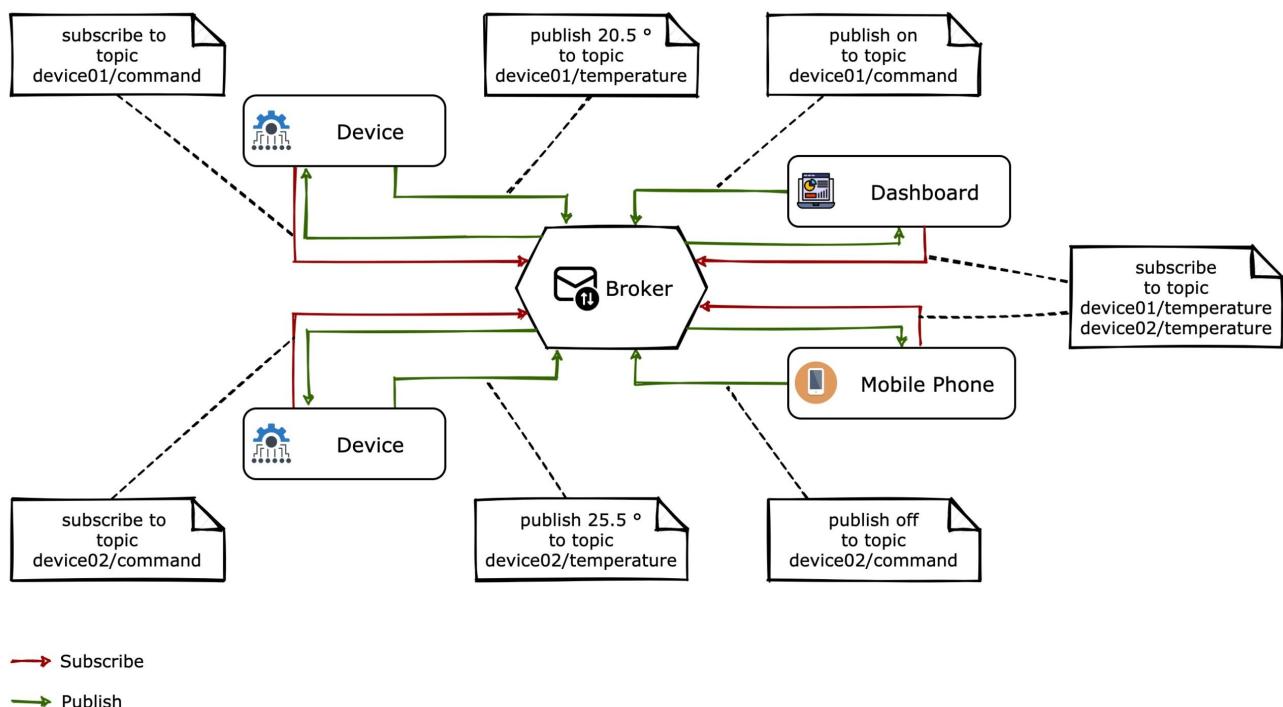


Figura 2 - Diagramma che mostra lo stile architetturale Publish/Subscribe

La Dashboard e il Mobile Phone indicano al broker che vogliono iscriversi a tutti i messaggi che appartengono ai **topic** device01/temperature e device02/temperature. Questo significa che riceveranno tutti i messaggi pubblicati dai device (device01 e device02) sui rispettivi topic. La Dashboard e il Mobile Phone pubblicano dei messaggi di "**comando**" sui due topic device01/command e device02/command, a questi due topic sono sottoscritti i rispettivi device che riceveranno quindi i comandi per attivare o disattivare per esempio dei relè o attuatori.

L'[animazione](#) (pubblicata sul mio blog) vi aiuterà ancora di più a comprendere il flusso dei messaggi in questo contesto.

Fino a questo momento abbiamo visto in superficie i componenti base di un sistema IoT. Dal successivo capitolo inizieremo a scendere più nello specifico della soluzione che vogliamo realizzare nel corso di questo articolo. Quello che andremo a realizzare sarà ovviamente in forma di prototipo e offrirà sicuramente degli ottimi spunti per la realizzazione di sistemi più complessi.

3. Overview sull'architettura della soluzione IoT

L'architettura di questa soluzione include nel suo insieme componenti hardware e software.
Quali sono i componenti che costituiscono questa soluzione IoT?

- **Componenti Hardware**
 - Sensore dei dati ambientali: GY-BME280
 - MCU: Espressif ESP32
 - Attuator: Modulo Relay (x4)
 - Gateway: Raspberry Pi 4
- **Componenti Software(principali)**
 - MQTT Broker: EMQ X Edge
 - Data Visualization: Node-RED
 - MCU Firmware

3.1. I componenti hardware

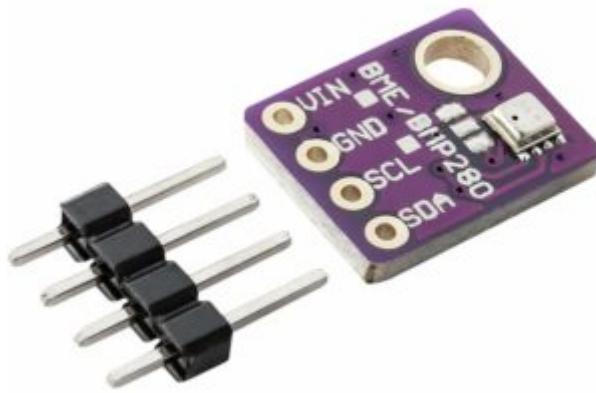


Figura 4 - Sensore parametri atmosferici GY-BME280

Il sensore **GY-BME280** consente di misurare la pressione atmosferica, oltre che la temperatura dell'aria e altre condizioni atmosferiche basilari per la creazione di una stazione meteo DYI. Il modulo può essere controllato tramite il bus **I2C** o **SPI**. Lo specifico modello utilizzato per questa soluzione è quello che dispone solo del bus I2C. Per tutte le informazioni di dettaglio di questo sensore fare riferimento alla [scheda tecnica](#).



Figura 5 - Kit di sviluppo ESP32

L'ESP32 è l'MCU prodotta da [Espressif System](#), un sistema dual-core CPU Harvard Architecture Xtensa LX6. La scelta di questa MCU è stata dettata per l'assoluta flessibilità e semplicità di programmazione e in particolare perché integra il modulo WiFi, quest'ultimo necessario per connettere l'MCU alla rete e di conseguenza poter comunicare con il Message Broker. Visto che quello che andremo a realizzare è un prototipo, utilizzeremo una scheda di sviluppo basata sul chip [ESP-WROOM-32](#).

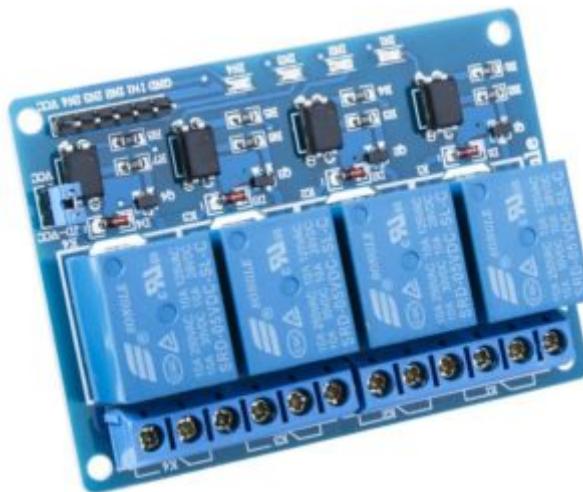


Figura 6 - Modulo da quattro Relè su GPIO

Il modulo composto da quattro relè è connesso all'MCU fruttando le porte **GPIO** (General Purpose Input/Output). Questo modulo è progettato opportunamente per essere collegato direttamente alle porte GPIO dell'MCU e funziona con una tensione di alimentazione pari a 5 volt. Sul portale di Elegoo sono riportate tutte le specifiche tecniche del modulo [ELEGOO Relay Module With Optocoupler](#).

Per il Gateway di questa soluzione ho deciso di adottare la single-board computer (**SBC**) [Raspberry Pi 4](#) e in particolare il modello con 8 GByte di memoria RAM.

3.2. I componenti software (principali)

Il componente software cardine di questa soluzione è senza ombra di dubbio il Message Broker. Dopo varie ricerche, test e confronti, per quest'architettura ho scelto di utilizzare [EMQ X Edge](#) come implementazione dell'MQTT Broker. Questo componente software è quello che risiede all'interno del Gateway.

EMQ X Edge è un MQTT Broker leggero, open source e concesso in licenza con Apache versione 2.0. Implementa le specifiche dei protocolli MQTT V3.1/V3.1.1 e V5.0 e supporta MQTT, TCP, WebSocket e diversi protocolli di settore come [ModBus](#). Può essere eseguito su diversi tipi di dispositivi edge con risorse limitate, come Raspberry Pi, gateway e server industriali. Può essere utilizzato come ponte tra i terminali locali e il broker remoto. Per eventuali approfondimenti fare riferimento alla documentazione di [EMQ X Edge](#) e [EMQ X Broker](#).

Sul progetto GitHub di [MQTT.org](#) e in particolare all'interno della Wiki c'è la [lista completa dei Broker](#). Sul documento [Comparison of MQTT implementations](#) sono ampiamente descritte le caratteristiche dei più diffusi software che implementano il protocollo MQTT.

Due documenti che invito a leggere che potrebbero aiutarvi nella scelta del Message Broker più indicato per i vostri progetti sono: [A Comparison of MQTT Brokers for Distributed IoT Edge Computing](#) (una pubblicazione Springer) e [Edge Based MQTT Broker Architecture for Geographical IoT Applications](#) (una pubblicazione di IEEE).

Per visualizzare i dati ambientali raccolti dai sensori, ho deciso d'utilizzare **Node-RED**, uno strumento di facile utilizzo e perfetto per questo genere di applicazioni.

Alla lista non mancano certamente i "pezzi" software che dovremo sviluppare e in particolare il software che andrà poi "flashato" sull'MCU che in questo caso è l'ESP32. Questo software è sostanzialmente responsabile della lettura dei dati ambientali e successiva trasmissione di questi al Gateway tramite il protocollo MQTT.

3.3. Schema dell'architettura

Una volta visto quali sono i componenti hardware e software che andranno a costituire la nostra soluzione, non rimane altro che metterli insieme. La figura a seguire mostra lo schema dell'architettura. All'interno dello schema è possibile notare ogni componente hardware e software che abbiamo descritto nei precedenti paragrafi. In questa architettura ho inserito volutamente due device che ho anche battezzato con un nome, ipotizzando che questi siano in zone diverse del campo. Certamente potremmo inserire altri device dello stesso tipo in questa architettura.

Il dialetto comune utilizzato per le comunicazioni tra Gateway, Device e Dashboard è l'MQTT. L'ESP32 comunica con il sensore ambientale tramite il bus I2C, mentre per perseguire azioni verso il modulo dei relè utilizza quattro canali GPIO.

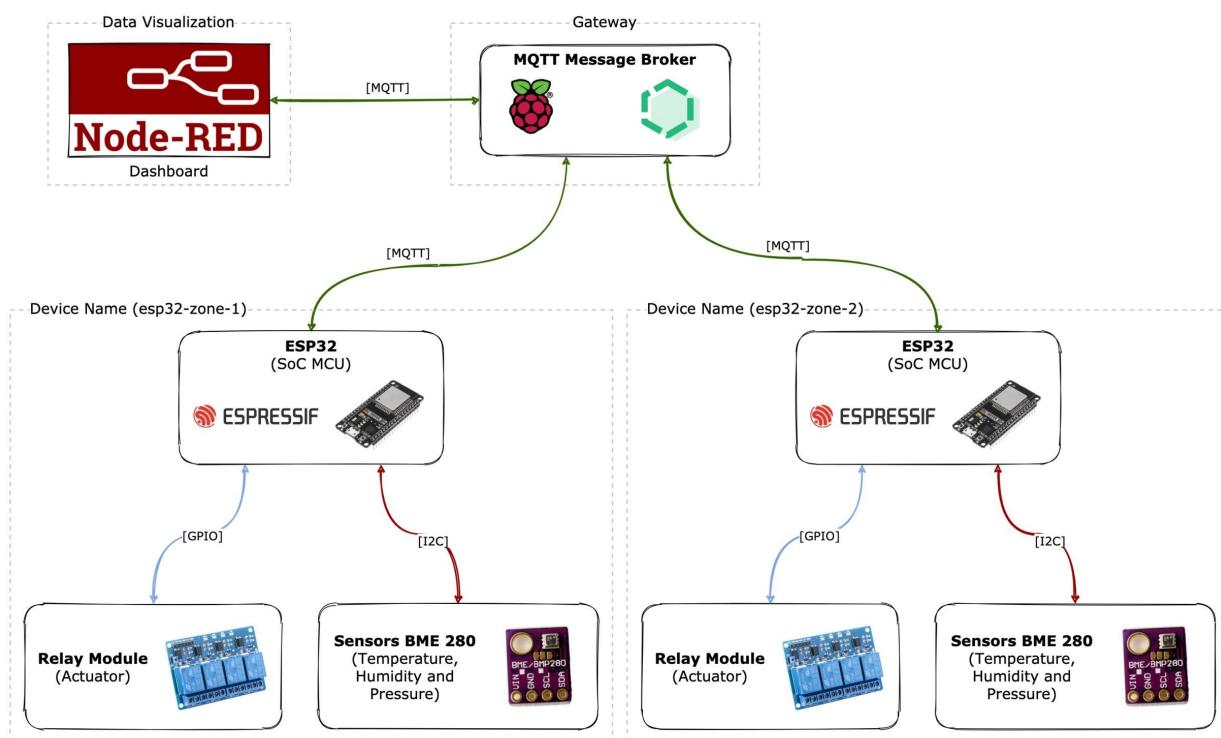


Figura 7 - Architettura della soluzione IoT con evidenza dei componenti hardware e software

3.4. Schema elettrico della soluzione IoT

Per implementare questa soluzione dal punto di vista hardware, dobbiamo sapere come realizzare i collegamenti tra i pezzi di "ferro" affinché il tutto possa funzionare nel modo corretto. Lo schema elettrico riporta i collegamenti che riguardano il device, quindi il sensore BME 280 e il modulo relè alla scheda di sviluppo [ESP32 DOIT DEV KIT](#) (da 30 pin). Lo schema riporta inoltre il collegamento del Gateway con un display [LCD 16x2](#), inserito con lo scopo di mostrare i dati ambientali e lo stato dei relè, informazioni la cui sorgente è rappresentata da ogni device.

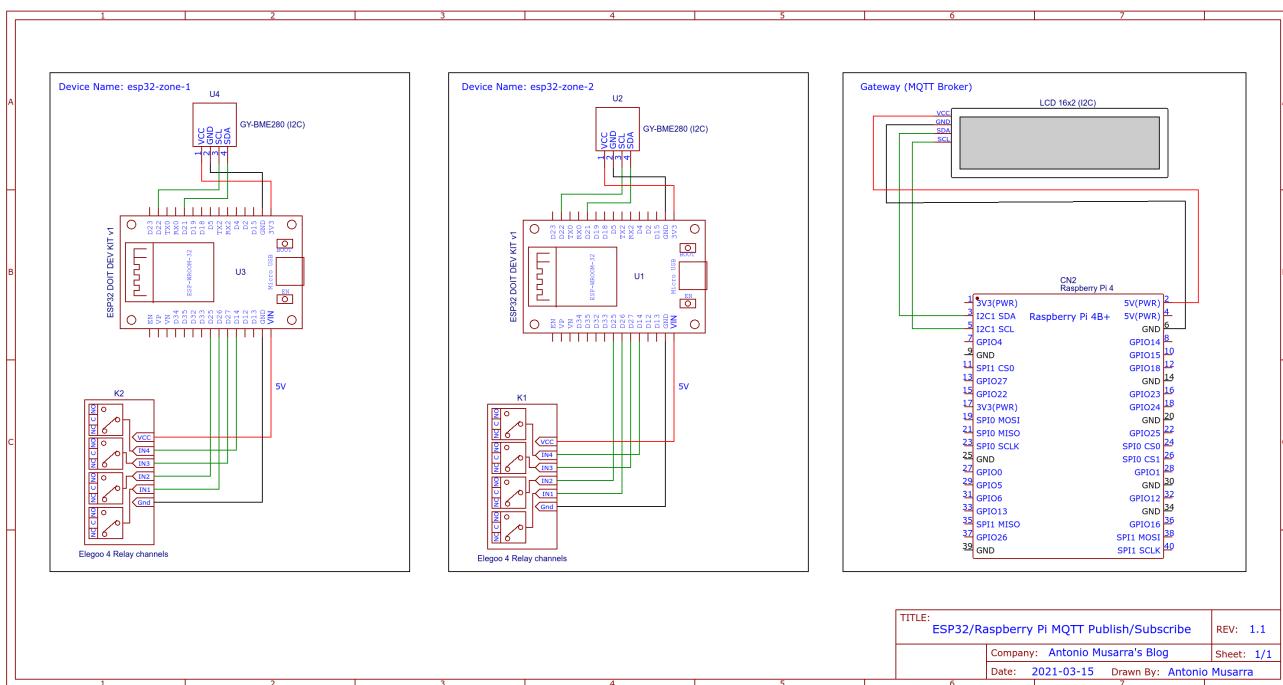


Figura 8 - Schema elettrico dei componenti hardware che costituiscono la soluzione IoT

La scheda di sviluppo dell'ESP32 è dotata di una porta Micro-USB per l'alimentazione del modulo stesso e l'upload del firmware. Questa scheda di sviluppo supporta l'upload automatico, non sarà quindi necessario cambiare manualmente il modo di upload e di esecuzione nel momento in cui sia necessario caricare il nuovo software.

Quello che vedete a seguire è lo schema di dettaglio del [PINOUT](#) della scheda di sviluppo utilizzata per questo progetto. Questo schema è stato realizzato da [Renzo Mischianti](#) e pubblicato sul suo [blog](#) e credo che sia uno dei migliori che possiamo trovare in giro per la rete.

ESP32 DEV KIT V1 / PINOUT

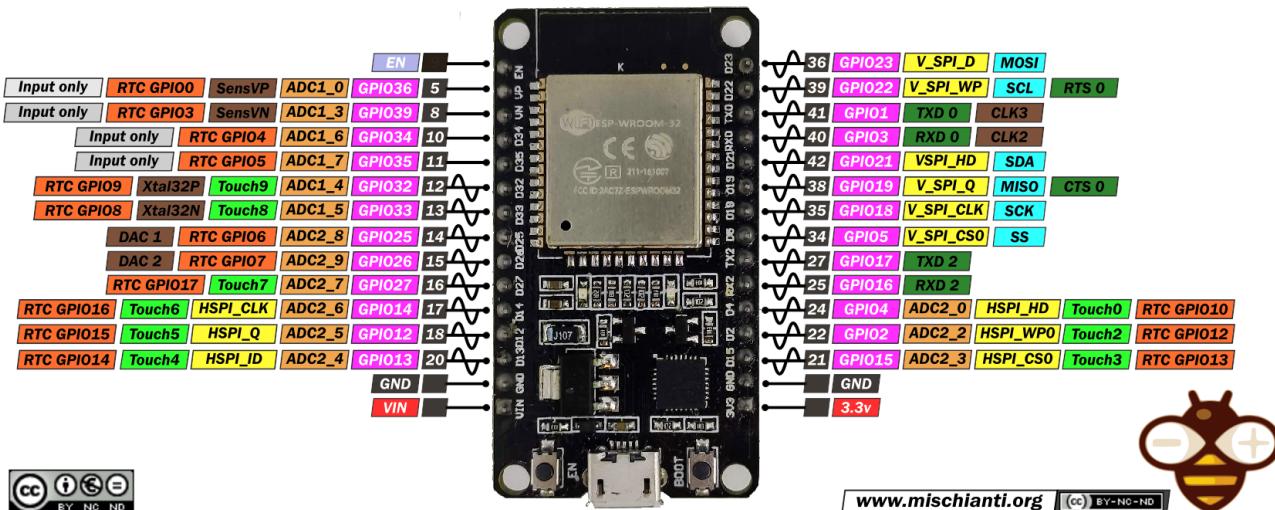


Figura 9 - PINOUT del modulo di sviluppo ESP32 Dev Kit V1 30 pin

3.5. Implementazione del prototipo

Adesso che abbiamo chiaro lo schema elettrico della nostra soluzione IoT, non resta altro che realizzare il prototipo. Tutto ciò che serve è indicato dalla tabella a seguire. Il prezzo mostrato è quello al momento della scrittura di questo articolo, potrebbe quindi subire delle leggere variazioni nel corso del tempo.

Tabella 1 - Elenco dei componenti hardware per la realizzazione del prototipo della soluzione IoT

Descrizione	Prezzo (unitario)	Store
Breadboard (830 contatti) (non necessaria)	€4,99	Amazon
Jumper Wire Cable Cavo F2F	€ 4,79	Amazon
Jumper Wire Cable Cavo M2M	€ 5,29	Amazon
Elegoo 4 Channel DC 5V Modulo Relay con Accoppiatore Ottico (x2)	€ 6,85	Amazon
GY-BME280 Sensore Pressione Barometrica (x2)	€ 9,49	Amazon
Scheda di Sviluppo ESP-WROOM-32 (2 Pack)	€15,99	Amazon
Raspberry Pi 4 Model B 8GB RAM	€ 83,90	Melopero
Display LCD 16x2	€ 12,99	Amazon
Totale	€ 160,63	

Una volta ottenuti tutti i pezzi indicati nel precedente elenco, è possibile procedere con l'assemblaggio dei vari componenti facendo riferimento allo schema elettrico mostrato in precedenza. Avrete bisogno anche di un saldatore stilo, questo perché il sensore BME 280 è fornito di pettine ma non è saldato sul PCB, operazione che dovremo fare da noi. Così facendo dovreste ottenere il risultato mostrato dalla figura a seguire.

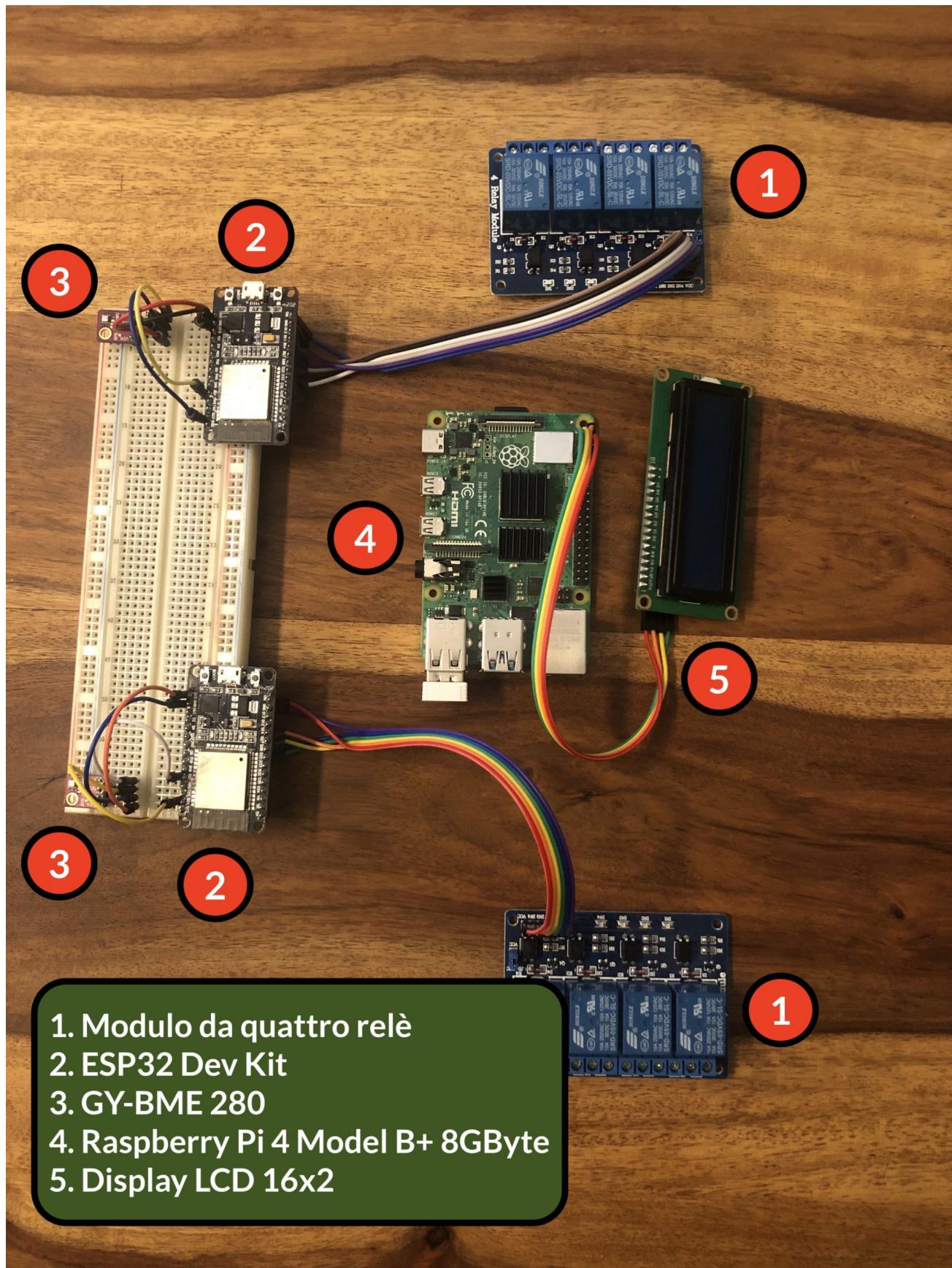


Figura 10 - Assemblaggio dei componenti hardware della soluzione IoT secondo lo schema elettrico specificato in Figura 8

L'implementazione del prototipo è così ultimata e con i "pezzi di ferro" abbiamo terminato. Possiamo quindi mettere al momento il prototipo da parte per continuare adesso in poi con la sezione software, iniziando dall'installazione dell'MQTT Broker sul Gateway (ovvero Raspberry Pi).

4. Installazione di EMQ X Edge su Raspberry Pi

Esistono vari modi per [installare EMQ X Edge su Raspberry Pi](#). Visto che il Raspberry Pi in mio possesso è già predisposto con [Ubuntu 20.04 LTS](#) e [Docker](#), la strada naturale è l'installazione via Docker, sfruttando l'ultima versione stabile dell'immagine di EMQ X Edge che in questo momento è la [4.2.9](#).

Per [installare Docker Engine sul vostro Raspberry Pi](#) vi rimando alla documentazione ufficiale disponibile sul sito di Docker. La versione di Docker installata sul mio Raspberry Pi 4 è la 20.10.15.

A seguire è mostrato il comando Docker per avviare il container di EMQ X Edge partendo dall'immagine [emqx/emqx-edge:4.2.9](#). Il comando successivo mostra lo stato di EMQ X Edge utilizzando il tool di

gestione [eqmx_ctl](#) e il relativo output, che in questo caso informa circa l'attuale status e la versione. Nell'avvio del container è stato specificato tramite l'environment `EMQX_ALLOW_ANONYMOUS` impostata a false, di non consentire l'accesso a utenti anonimi, questo per ovvi motivi di sicurezza.

Console 1 - Installazione EMQ X Edge

```
# Create a new Docker Network Bridge
docker network create edge-iot

# Run EMQ X Edge as Docker Image
docker run -d --network edge-iot --name mqtt-broker -e EMQX_ALLOW_ANONYMOUS=false -p 8083:8083 -p 18083:18083 -p 1883:1883 emqx/emqx-edge:4.2.9

# Get Status of the EMQ X Edge
docker exec -it mqtt-broker /bin/bash ./bin/emqx_ctl status

# Output of the EMQ X Edge
Node 'cee88f3106d3@172.17.0.2' is started
emqx 4.2.9 is running
```

A seguire è mostrato lo screencast di [How to install EMQ X Edge on Raspberry Pi 4 from Docker Image](#) in modo che possiate vedere esattamente l'esecuzione degli step indicati in precedenza.

```

amusarra@amusarra-pi-4b:~$ neofetch
amusarra@amusarra-pi-4b
-----
OS: Ubuntu 20.04.2 LTS aarch64
Host: Raspberry Pi 4 Model B Rev 1.4
Kernel: 5.4.0-1032-rasp
Uptime: 6 hours, 35 mins
Packages: 1781 (dpkg), 4 (snap)
Shell: bash 5.0.17
Terminal: /dev/pts/0
CPU: BCM2835 (4) @ 1.500GHz
Memory: 322MiB / 7811MiB
[Color Bar]

amusarra@amusarra-pi-4b:~$ docker run -d --network edge-iot --name emqx-broker -e EMQ_X_MQTT_FOLLOW_ANONYMOUS=false -p 8083:8083 -p 18083:18083 -p 1883:1883 emqx/emqx-edge:4.2.9
Unable to find image 'emqx/emqx-edge:4.2.9' locally
4.2.9: Pulling from emqx/emqx-edge
47185b9379cb: Pull complete
8e9701b49cfa: Pull complete
ea736e07312a: Pull complete
474c0a360cf3: Pull complete
ldic53032fa7: Pull complete
e3c6de3252c4: Pull complete
712a42bdcc4d: Extracting [=====] 13.76MB/22.03MB
[Progress Bar]

```

Screencast 1 - How to install EMQ X Edge on Raspberry Pi 4 from Docker Image

Una volta che il container di EMQ X Edge è up, possiamo raggiungere la Dashboard di amministrazione attraverso il nostro browser puntando all'indirizzo [http://{\\$IP_GATEWAY_RPI}:18083](http://{$IP_GATEWAY_RPI}:18083). La username e password di accesso di default sono: admin/public (consigliato il cambio password). La figura a seguire mostra la home della dashboard di EMQ X Edge.

Figura 11 - Dashboard di EMQ X Edge su Raspberry Pi 4

4.1. Configurazione utenze e Access Control List (ACL)

Una volta che l'istanza di EMQ X Edge è up, dobbiamo procedere con la configurazione delle utenze di accesso e delle **ACL**, quest'ultime definiscono le azioni che le utenze posso eseguire sui topic. Le azioni sui topic sono: pubblicazione e sottoscrizione. Per la configurazione delle utenze e ACL, è necessario abilitare il modulo [Mnesia](#) utilizzando il [tool di amministrazione emqx_ctl](#).

Per questa soluzione sono previste le utenze e ACL descritte a seguire e raggruppate nella tabella successiva.

- L'utente **esp32-device** sarà l'utenza utilizzata dai dispositivi ESP32 (vedi schema elettrico di Figura 8) che dovrà essere capace di pubblicare i dati ambientali sul topic **esp32/telemetry_data** e pubblicare lo stato degli attuatori (i relè) sui topic **esp32/releay_{relay_id}_status**. L'utenza dovrà essere inoltre capace di sottoscriversi al topic **esp32/command**.
- L'utente **gw-rpi4-device** sarà l'utenza utilizzata dal software residente e in esecuzione sul Raspberry Pi 4. Questo software sarà in particolare responsabile di visualizzare i dati ambientali sul display LCD collegato al Raspberry Pi e inviare i comandi ai device. L'utenza dovrà quindi essere capace di sottoscriversi ai topic **esp32/telemetry_data** e **esp32/releay_{relay_id}_status** e pubblicare sul topic **esp32/command**.
- L'utente **mqtt-dashboard** sarà l'utenza utilizzata dal software che implementa la dashboard di visualizzazione dei dati ambientali e per inviare i comandi ai device. L'utenza dovrà quindi essere capace di sottoscriversi ai topic **esp32/telemetry_data** e **esp32/releay_{relay_id}_status** e pubblicare sul topic **esp32/command**.

Il capitolo successivo scenderà nel dettaglio riguardo i topic di cui al momento sono stati indicati solo i nomi.

Tabella 2 - Definizione delle utenze e ACL sui topic necessari per l'implementazione della soluzione IoT

Utente	Topic	ACL
esp32-device	esp32/telemetry_data	pub=true, sub=false
	esp32/relay_00_status	pub=true, sub=false
	esp32/relay_01_status	pub=true, sub=false
	esp32/relay_02_status	pub=true, sub=false
	esp32/relay_03_status	pub=true, sub=false
	esp32/command	pub=false, sub=true
gw-rpi4-device	esp32/telemetry_data	pub=false, sub=true
	esp32/relay_00_status	pub=false, sub=true
	esp32/relay_01_status	pub=false, sub=true
	esp32/relay_02_status	pub=false, sub=true
	esp32/relay_03_status	pub=false, sub=true
	esp32/command	pub=true, sub=false
mqtt-dashboard	esp32/telemetry_data	pub=false, sub=true
	esp32/command	pub=true, sub=false
	esp32/relay_00_status	pub=false, sub=true
	esp32/relay_01_status	pub=false, sub=true
	esp32/relay_02_status	pub=false, sub=true
	esp32/relay_03_status	pub=false, sub=true

I comandi a seguire devono essere lanciati direttamente dalla console del nostro Gateway o Raspberry Pi e sono necessari affinché siano aggiunti i tre utenti e le ACL descritti e indicati nella tabella precedente. Per aggiungere utenti e ACL utilizzeremo le API REST fornite da EMQ X Edge e richiamate utilizzando il tool **curl**. Sul repository GitHub Gist [EMQ X Edge 4.2.9 - Add User and ACL \(emqx_auth_mnesia module\)](#) sono disponibili i documenti JSON per l'inserimento degli utenti e ACL.

È possibile ottenere lo stesso risultato utilizzando la CLI di EMQ X Edge che al momento però presenta qualche problema con la definizione delle ACL, ecco il motivo per cui sono stato costretto a "regredire" all'uso delle API REST.

Console 2 - Configurazione EMQ X Edge: creazione utenti e ACL

```
# Enable the Mnesia Module
docker exec -it mqtt-broker /bin/bash ./bin/emqx_ctl plugins load emqx_auth_mnesia

# Add users
git clone https://gist.github.com/fa3294ff35c8794d2eb8a2b5792901df.git emqx-user-acl
curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-
acl/add_users.json http://localhost:18083/api/v4/mqtt_user

# Add ACL for User/Topic
curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-
acl/acl_rules_user_esp32_device.json http://localhost:18083/api/v4/mqtt_acl

curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-
acl/acl_rules_user_gw_rpi4_device.json http://localhost:18083/api/v4/mqtt_acl

curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-
acl/acl_rules_user_mqtt_dashboard_device.json http://localhost:18083/api/v4/mqtt_acl

# Restart EMQ X Edge container
docker restart mqtt-broker

# Check User ACL
docker exec -it mqtt-broker /bin/bash ./bin/emqx_ctl mqtt-acl show esp32-device
docker exec -it mqtt-broker /bin/bash ./bin/emqx_ctl mqtt-acl show gw-rpi4-device
docker exec -it mqtt-broker /bin/bash ./bin/emqx_ctl mqtt-acl show mqtt-dashboard
```

A seguire è mostrato lo screencast di [How to add users and ACLs on EMQ X Edge using the Mnesia module](#) in modo che possiate vedere esattamente l'esecuzione degli step indicati in precedenza.

```

amusarra@amusarra-pi-4b:~$ # Enable the Mnesia Module
amusarra@amusarra-pi-4b:~$ docker exec -it emqxre /bin/emqx_ctl plugins load emqx_auth_mnesia
Plugin emqx_auth_mnesia loaded successfully.
amusarra@amusarra-pi-4b:~$ # Add User
amusarra@amusarra-pi-4b:~$ git clone https://gist.github.com/fa3294ff35c8794d2eb8a2b5792901df.git emqx-user-acl
amusarra@amusarra-pi-4b:~$ git clone https://gist.github.com/fa3294ff35c8794d2eb8a2b5792901df.git emqx-user-acl
Cloning into 'emqx-user-acl'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 856 bytes | 171.00 KiB/s, done.
amusarra@amusarra-pi-4b:~$ ls -l emqx-user-acl/
total 16
-rw-rw-r-- 1 amusarra amusarra 1367 Apr  5 00:16 acl_rules_user_esp32_device.json
-rw-rw-r-- 1 amusarra amusarra 1391 Apr  5 00:16 acl_rules_user_gw_rpi4_device.json
-rw-rw-r-- 1 amusarra amusarra 1391 Apr  5 00:16 acl_rules_user_mqtt_dashboard_device.json
-rw-rw-r-- 1 amusarra amusarra  295 Apr  5 00:16 add_users.json
amusarra@amusarra-pi-4b:~$ curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-acl/add_users.json http://192.168.1.83:18083/api/v4/mqtt_user
{"data":{"mqttdashboard":"ok","gw-rpi4-device":"ok","esp32-device":"ok"}, "code":0} amusarra@amusarra-pi-4b:~$ 
amusarra@amusarra-pi-4b:~$ # Add ACL for User/Topic
amusarra@amusarra-pi-4b:~$ curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-acl/acl_rules_user_esp32_device.json http://192.168.1.83:18083/api/v4/mqtt_acl
{"data":{"esp32-device":"ok"}, "code":0} amusarra@amusarra-pi-4b:~$ 
amusarra@amusarra-pi-4b:~$ curl -u admin:public -H "Content-Type: application/json" -d @emqx-user-acl/acl_rules_user_gw_rpi4_device.json http://192.168.1.83:18083/api/v4/mqtt_acl
{"data":{"gw-rpi4-device":"ok"}, "code":0} amusarra@amusarra-pi-4b:~$ 
amusarra@amusarra-pi-4b:~$ 

```

Screencast 2 - How to add users and ACLs on EMQ X Edge using the Mnesia module

Molto bene. Il Message Broker MQTT è stato configurato per i nostri scopi ed è quindi pronto all'uso. Nel prossimo capitolo andremo a dettagliare il ruolo di ogni topic, il significato di **Quality of Service** (o QoS) nel contesto MQTT e per finire vedremo la struttura del payload, quest'ultimo è l'effettivo contenuto del messaggio che viene scambiato tra i vari attori.

5. Configurazione dei topic, QoS e struttura dei payload

Affinché la nostra soluzione funzioni, quindi, che i vari componenti che fanno parte dell'architettura colloquino tra loro, è necessario configurare gli opportuni topic sul Message Broker. **Quali sono i topic di cui abbiamo bisogno?**

Per questa soluzione esistono tre tipi di topic per tre diversi scopi:

1. il topic su cui ogni device pubblicherà i dati ambientali provenienti dai sensori BME280. Il nome di questo topic sarà **esp32/telemetry_data**;
2. i topic su cui ogni device pubblicherà lo stato di ogni relè. In questo caso avremo quattro diversi topic dove pubblicare lo stato di ogni relè
 - a. esp32/relay_00_status
 - b. esp32/relay_01_status
 - c. esp32/relay_02_status
 - d. esp32/relay_03_status
3. il topic destinato a recepire i comandi necessari per l'attivazione o disattivazione dei relè connessi ai device o per ricevere lo stato degli stessi. Il nome di questo topic sarà **esp32/command**.

5.1. Topic esp32/command

Topic per la ricezione dei comandi che dovranno essere eseguiti sui device. Questi comandi consentono di attivare o disattivare i relè connessi al device o di richiedere lo stato dei relè.

Il comando è costituito da una stringa di testo con un formato prestabilito all'interno della quale sono contenute le informazioni circa: nome del device, entità, identificativo dell'entità, comando.

1. I publisher di questo topic sono: dashboard, mobile phone o altro dispositivo adibito al comando.
2. I subscriber di questo topic sono: i device (esp32-zone-1 e esp32-zone-2).
3. Il QoS di questo topic è 1.

5.2. Topic esp32/telemetry_data

Topic su cui ogni device pubblicherà i dati ambientali provenienti dai sensori BME280. Ogni messaggio pubblicato conterrà le seguenti informazioni:

1. **clientId**: Identificativo univoco di ogni device connesso alla rete e di conseguenza al Message Broker;
2. **deviceName**: nome logico assegnato al dispositivo (esempio: esp32-zone-1);
3. **time**: timestamp della lettura dei valori ambientali;
4. **temperature**, **humidity**, **pressure**: valori delle misure dei rispettivi parametri ambientali;
5. **interval**: intervallo delle letture in ms (millisecondi);
6. **counter**: numero della lettura;
7. **relaysStatus**: array contenente le informazioni circa lo stato dei relè collegati al device (ESP32).

Il formato del payload del messaggio è JSON.

1. I publisher di questo topic sono: i device (esp32-zone-1 e esp32-zone-2).
2. I subscriber di questo topic sono: dashboard, mobile phone o altro dispositivo (o generico client).
3. Il QoS di questo topic è 0.

5.3. Topic esp32/releay_\${id}_status

Topic su cui ogni device pubblicherà lo stato di ogni relè connesso. Ogni cambiamento di stato, dovuto per esempio all'invio di un comando, comporterà la pubblicazione su questo topic dell'avvenuto cambio di stato.

Ogni messaggio pubblicato conterrà le seguenti informazioni:

1. **clientId**: Identificativo univoco di ogni device connesso alla rete e di conseguenza al Message Broker;
2. **deviceName**: nome logico assegnato al dispositivo (esempio: esp32-zone-1);
3. **time**: timestamp della lettura dei valori ambientali;
4. **relayId**: identificativo del relè. I valori ammessi sono nel range 0-3, questo perchè i relè sono quattro per ogni device;
5. **status**: valore intero che indica lo stato del relè. 0 se non attivo 1 se attivo.

Il formato del payload del messaggio è JSON.

1. I publisher di questo topic sono: i device (esp32-zone-1 e esp32-zone-2).
2. I subscriber di questo topic sono: dashboard, mobile phone o altro dispositivo (o generico client).
3. Il QoS di questo topic è 1.

5.4. Qualche nota sul QoS

Ad ogni topic è stato assegnato il **Quality of Service** (o QoS). **Cosa indica la Quality of Service (o la qualità del servizio)?**

Il livello di QoS è un accordo esplicito tra il mittente di un messaggio e il destinatario di un messaggio che definisce la garanzia di consegna per un messaggio specifico. Ci sono 3 livelli di QoS in MQTT:

1. *At most once* (o Al massimo una volta) = 0
2. *At least once* (o Almeno una volta) = 1
3. *Exactly once* (o Esattamente una volta) = 2.

Quando parliamo di QoS in MQTT, occorre considerare i due lati della consegna del messaggio:

1. Consegna dei messaggi dal client di pubblicazione (publisher) al broker.
2. Consegna del messaggio dal broker al client di sottoscrizione (subscriber).

Esamineremo separatamente i due lati della consegna del messaggio perché ci sono sottili differenze tra i due. Il client che pubblica il messaggio sul broker definisce il livello QoS del messaggio quando invia il messaggio al broker. Il broker trasmette questo messaggio ai client di sottoscrizione utilizzando il livello QoS definito da ciascun client di sottoscrizione durante il processo di sottoscrizione. **Se il client di sottoscrizione definisce un QoS inferiore rispetto al client di pubblicazione, il broker trasmette il messaggio con la qualità del servizio inferiore.**

QoS è una caratteristica chiave del protocollo MQTT. QoS offre al client la possibilità di scegliere un livello di servizio che corrisponda alla sua affidabilità di rete e alla logica dell'applicazione. Poiché MQTT gestisce la ritrasmissione dei messaggi e garantisce la consegna (anche quando il trasporto sottostante non è affidabile), QoS semplifica notevolmente la comunicazione nelle reti inaffidabili.

Per i messaggi del topic esp32/telemetry_data possiamo benissimo decidere di adottare un QoS 0 (livello minimo). Questo livello di servizio garantisce una consegna con il massimo impegno. Non c'è garanzia di consegna. Il destinatario non conferma la ricezione del messaggio e il messaggio non viene archiviato e ritrasmesso dal mittente. Il livello QoS 0 è spesso chiamato "**fire and forget o spara e dimentica**" e fornisce la stessa garanzia del protocollo TCP sottostante.

La scelta di questo QoS è dettata dal fatto che la perdita di qualche messaggio non è critica per il tipo di applicazione, considerando anche che la pubblicazione dei dati avviene a intervalli regolari e frequenti, inoltre non abbiamo la necessità di accodare i messaggi per i client disconnessi.

Per i messaggi del topic esp32/command e esp32/releay_{id}_status dovremmo scegliere il QoS 1. Questo livello di servizio garantisce che un messaggio venga recapitato almeno una volta al destinatario. Il mittente memorizza il messaggio fino a quando non riceve un pacchetto PUBACK dal destinatario che conferma la ricezione del messaggio. È possibile che un messaggio venga inviato o consegnato più volte.

La scelta di questo QoS è dettata dall'importanza di ricevere (almeno una volta) sia i messaggi che contengono i comandi per i nostri attuatori sia i messaggi che contengono lo stato dei nostri attuatori (i relè). L'eventualità di messaggi duplicati non rappresenta un problema per la nostra applicazione.

Il livello QoS 2 non è utilizzato dalla nostra soluzione perché non è necessario un livello di servizio così elevato come quello offerto dal livello 2. Adottare questo livello di servizio quando è fondamentale per la tua applicazione ricevere tutti i messaggi esattamente una volta. Questo è spesso il caso se una consegna duplicata può danneggiare gli utenti dell'applicazione o i client in sottoscrizione. Bisogna quindi essere consapevoli del sovraccarico e che l'interazione QoS 2 richiede più tempo per essere completata.

5.5. Struttura dei payload

A seguire sono mostrate le tre strutture dei payload per i messaggi dei topic descritti nella precedente tabella.

Tabella 4 - Struttura dei messaggi MQTT utilizzati dalla soluzione IoT

Topic	Tipo formato	Formato/Esempio
esp32/command	Stringa formattata	<p>Formato:</p> <p><code> \${device-name}:\${entity};\${relay-id};\{off on status}</code></p> <p>Esempi:</p> <ol style="list-style-type: none"> 1. Comando per attivare il relè 1 del device chiamato esp32-zone-1: <code>esp32-zone-1:relay;1;off</code> 2. Comando per disattivare il relè 0 del device chiamato esp32-zone-2: <code>esp32-zone-2:relay;0;on</code> 3. Comando per richiedere lo stato del relè 1 del device chiamato esp32-zone-1: <code>esp32-zone-1:relay;1;status</code>
esp32/telemetry_data	JSON	Vedi Payload 1
esp32/releay_{id}_status	JSON	Vedi Payload 2

Payload 1 - Esempio payload per il topic esp32/telemetry_data

```
{
  "clientId": "esp32-client-11fa",
  "deviceName": "esp32-zone-1",
  "time": 1618223579,
  "temperature": 19.49,
  "humidity": 39.67969,
  "pressure": 93904,
  "interval": 5000,
  "counter": 1,
  "relaysStatus": [
    0,
    0,
    0,
    0
  ]
}
```

Payload 2 - Esempio payload per il topic esp32/telemetry_data

```
{
  "clientId": "esp32-client-2c2a",
  "deviceName": "esp32-zone-1",
  "time": 1618225166,
  "relayId": 3,
  "status": 0
}
```

Le immagini a seguire mostrano alcuni dei messaggi MQTT gestiti dalla nostra soluzione. Dai messaggi MQTT, oltre a vedere il messaggio (codificato), è possibile notare le varie caratteristiche, quali per esempio il QoS, il retain, il topic, etc. Per il topic esp32/command è possibile notare il QoS pari a 1. Le informazioni sono state estratte catturando il traffico di rete direttamente dal Gateway (il Raspberry Pi) tramite il tool **tshark**.

```
"mqtt": {
  "mqtt.hdrFlags": "0x00000030",
  "mqtt.hdrFlags_tree": {
    "mqtt.msgtype": "3",
    "mqtt.dupflag": "0",
    "mqtt.qos": "0",
    "mqtt.retain": "0"
  },
  "mqtt.len": "212",
  "mqtt.topic_len": "20",
  "mqtt.topic": "esp32/telemetry_data",
  "mqtt.msg": "7b:22:63:6c:69:65:6e:74:49:64:22:3a:22:65:73:70:33:32:2d:63:6c:69:65:6e:74:2d:65:37:30:22:2c:22:64:65:76:69:63:65:4e:61:6d:65:22:3a:22:65:73:70:33:2d:7a:6f:6e:65:2d:32:22:2c:22:74:69:6d:65:22:3a:31:36:31:38:32:32:38:39:32:31:2c:22:74:65:6d:70:65:72:61:74:75:72:65:22:3a:31:39:2e:37:37:2c:22:68:75:6d:69:64:69:74:79:22:3a:33:33:2e:39:37:38:35:32:2c:22:70:72:65:73:73:75:72:65:22:3a:39:33:38:35:36:2c:22:69:6e:74:65:72:76:61:6c:22:3a:35:30:30:30:2c:22:63:6f:75:6e:74:65:72:22:3a:37:37:35:20:22:72:65:6c:61:79:73:53:74:61:74:75:73:22:3a:5b:30:2c:30:2c:31:2c:30:5d:7d"
},
}
^C10 packets captured
}
amusarra@amusarra-pi-4b:~/mqtt-topic-subscribe$ sudo tshark -i wlan0 -f "src 192.168.1.125 || 192.168.1.126" -Y 'mqtt.topic == \"esp32/telemetry_data\"' -T json
```

Figura 12 - Cattura dei pacchetti MQTT pubblicati sul topic esp32/telemetry_data dai device ESP32

```

},
"mqtt": {
  "mqtt.hdrFlags": "0x00000032",
  "mqtt.hdrFlags_tree": {
    "mqtt.msgtype": "3",
    "mqtt.dupFlag": "0",
    "mqtt.qos": "1",
    "mqtt.retain": "0"
  },
  "mqtt.len": "41",
  "mqtt.topic_len": "13",
  "mqtt.topic": "esp32/command",
  "mqttmsgid": "34",
  "mqtt.msg": "65:73:70:33:32:2d:7a:6e:65:2d:31:3a:72:65:6c:61:79:3b:30:3b:6f:66:66"
},
}
^C8 packets captured
]
amusarra@amusarra-pi-4b:~/mqtt-topic-subscribe$ sudo tshark -i wlan0 -f "dst 192.168.1.125 || 192.168.1.126" -Y 'mqtt.topic == "esp32/command"' -T json

```

Figura 13 - Cattura dei pacchetti MQTT pubblicati sul topic esp32/command dalla Dashboard Node-RED

Per catturare il traffico MQTT che riguarda i messaggi pubblicati sul topic esp32/telemetry_data dai due dispositivi, è sufficiente utilizzare il comando mostrato a seguire.

Console 3 - Catturare il traffico MQTT del topic esp32/telemetry_data

```
# Cattura del traffico tramite tshark verso il topic esp32/telemetry_data
sudo tshark -i wlan0 -f "src 192.168.1.125 || 192.168.1.126" -Y 'mqtt.topic == "esp32/telemetry_data"' -T json
```

Per catturare il traffico MQTT che riguarda i messaggi pubblicati sul topic esp32/command dalla dashboard, è sufficiente utilizzare il comando:

Console 4 - Catturare il traffico MQTT del topic esp32/command

```
# Cattura del traffico tramite tshark verso il topic esp32/command
sudo tshark -i wlan0 -f "dst 192.168.1.125 || 192.168.1.126" -Y 'mqtt.topic == "esp32/command"' -T json
```

Ovviamente gli indirizzi IP possono variare, di conseguenza i comandi mostrati in precedenza devono essere revisionati.

Dopo aver chiarito il ruolo dei topic e la struttura dei messaggi per questa soluzione, possiamo mettere le mani in pasta al codice iniziando dal software che bisogna realizzare per l'ESP32.

6. Come programmare l'ESP32

ESP-IDF è il framework di sviluppo IoT ufficiale di Espressif per le serie di SoC ESP32 ed ESP32-S. Fornisce un SDK autosufficiente per qualsiasi sviluppo di applicazioni generiche su queste piattaforme, utilizzando linguaggi di programmazione come C e C++. ESP-IDF attualmente alimenta milioni di dispositivi sul campo e consente di costruire una varietà di prodotti connessi in rete, che vanno da semplici lampadine e giocattoli a grandi elettrodomestici e dispositivi industriali.

Per questa soluzione ho scelto però di adottare il framework di **Arduino** basato su **Wiring**, per tre semplici motivi: semplice da usare, ampia documentazione e ampiamente diffuso e conosciuto.

Utilizzeremo quindi il classico ambiente di programmazione Arduino IDE? No, propongo qualcosa di diverso.

Sono sempre stato dell'idea che dal momento in cui si crea un progetto software, per quanto possibile, è meglio non legare questo ad uno specifico **IDE**(Integrated Development Environment). Mi piace il fatto che i progetti siano indipendenti dall'IDE, devo essere nelle condizioni di poter eseguire la build del progetto da linea di comando: **git clone, build and install**.

Il qualcosa di diverso è **PlatformIO**. È un framework cross-platform, **cross-architecture, multiple-framework**: uno strumento per ingegneri di sistemi embedded e per sviluppatori software che scrivono applicazioni per prodotti embedded.

Senza entrare troppo in profondità nei dettagli di implementazione di PlatformIO, il ciclo di lavoro del progetto sviluppato utilizzando PlatformIO è il seguente:

1. Si scelgono le schede di sviluppo interessate specificandole all'interno del file platformio.ini (file di configurazione del progetto).
2. Sulla base di questo elenco di schede, PlatformIO scarica le toolchain richieste e le installa automaticamente.
3. Si inizia lo sviluppo del codice e PlatformIO si assicura che sia compilato, preparato e caricato su tutte le schede di interesse.

Per la scrittura del codice è possibile utilizzare il vostro editor o IDE preferito, in alternativa si può utilizzare **PlatformIO per VSCode**.

L'unico requisito è l'installazione di **PlatformIO Core (CLI)** sulla propria macchina di sviluppo. PlatformIO Core è scritto in **Python** e funziona quindi su Windows, macOS, Linux, FreeBSD e ARM-based credit-card sized computer (**Raspberry Pi, BeagleBone, CubieBoard, Samsung ARTIK**, etc.).

La versione minima di PlatformIO Core da installare è la versione 5.x. Se avete già installata una versione di PlatformIO, è possibile eseguire l'upgrade attraverso il comando [pio upgrade](#).

Adesso che abbiamo scelto PlatformIO come strumento di supporto allo sviluppo del software, possiamo passare ad analizzare il codice sviluppato.

7. Scrrittura del software per l'ESP32

Le responsabilità maggiori per il software da realizzare per l'ESP32 sono: la lettura dei dati ambientali e la successiva pubblicazione sul topic esp32/telemetry_data attraverso il Message Broker, oltre a ricevere i comandi dal topic esp32/command e la successiva esecuzione degli stessi. Le macro responsabilità sono indicate a seguire.

1. Setup I2C per il sensore GY-BME280
2. Setup GPIO per i relè
3. Setup della connessione alla rete WiFi
4. Setup della connessione al Message Broker MQTT
5. Lettura dei dati ambientali dal sensore GY-BME280
6. Preparazione del messaggio con i dati ambientali in formato JSON
7. Preparazione del messaggio con i dati dello stato dei relè in formato JSON
8. Pubblicazione dei dati ambientali via MQTT
9. Pubblicazione dello stato dei relè via MQTT
10. Lettura ed esecuzione dei comandi che arrivano sul topic esp32/command

Per comodità l'intero progetto è disponibile sul repository GitHub [ESP32 MQTT Publish & Subscribe](#).

La tabella a seguire mostra le librerie che sono state usate per la realizzazione del software. Queste librerie sono specificate all'interno del file [platformio.ini](#), e queste saranno scaricate da PlatformIO qualora non già disponibili sul proprio ambiente di sviluppo.

Tabella 5 - Lista delle librerie utilizzate per la realizzazione del software da installare sui device (ESP32)

Nome Libreria	Versione	Descrizione	Sito
Adafruit Sensor Unified	1.1.4	Il driver Adafruit Unified Sensor cerca di armonizzare i dati del sensore in un unico "tipo", più leggero, e astrae anche i dettagli sul sensore.	https://learn.adafruit.com/using-the-adafruit-unified-sensor-driver/introduction
Adafruit BME280	2.1.2	Libreria per il sensore BME280	https://github.com/adafruit/Adafruit_BME280_Library
ArduinoJson	6.17.3	Libreria per la manipolazione dei documenti JSON.	https://arduinojson.org/
ArduinoLog	1.0.3	Libreria per il logging	https://www.arduino.cc/reference/en/libraries/arduinolog/
ESP32Ping	1.7	Libreria per il Ping di macchine su rete TCP/IP	https://github.com/marian-craciunescu/ESP32Ping
PubSubClient	2.8	Libreria che fornisce un client per la pubblicazione/sottoscrizione di messaggi con un server che supporta MQTT.	https://github.com/knolleary/pubsubclient
NTPClient	3.1.10	Libreria che fornisce un semplice client per un server NTP.	https://github.com/arduino-libraries/NTPClient

Le funzioni fondamentali del programma [esp32_mqtt_publish_subscribe.cpp](#) sono:

1. **void setup_wifi()**: la responsabilità di questa funzione è quella di instaurare la connessione alla rete WiFi. I parametri di connessione (SSID e username) possono essere impostati durante la fase di build; successivamente vedremo come. La rete WiFi a cui dobbiamo connettere i device deve garantire la connettività verso il Message Broker;
2. **void reconnect()**: la responsabilità di questa funzione è quella di instaurare la connessione al Message Broker e gestire eventuali ri-connessioni, per esempio nei casi in cui cada la connessione alla rete WiFi. Subito dopo la connessione al Message Broker è eseguita la sottoscrizione al topic esp32/command con QoS pari a uno e l'inizializzazione dello stato dei relè a off;
3. **void update_relay_status(int relayId, const int status)**: la responsabilità di questa funzione è quella di aggiornare lo stato dei relè pubblicando i messaggi sui relativi topic esp32/releay_{id}_status;
4. **int * get_relays_status()**: la responsabilità di questa funzione è quella di restituire un array di quattro elementi che contenente lo stato dei relè i cui valori possono essere: 0 per off e 1 per on;
5. **void callback(char *topic, byte *message, unsigned int length)**: questa è la funzione di callback richiamata ogni qualvolta viene ricevuto un messaggio sul topic per cui si ha una sottoscrizione attiva e in questo caso il topic per cui c'è una sottoscrizione attiva è esp32/command. La responsabilità di questa funzione è quella di acquisire i messaggi che arrivano sul topic esp32/command, fare il parsing del comando ricevuto ed eseguire l'azione corrispondente che può essere: attivare/disattivare il relè o pubblicare lo stato del relè.

Adottando il framework Arduino, portiamo dietro anche le funzioni obbligatorie [setup\(\)](#) e [loop\(\)](#). La funzione [setup\(\)](#) è eseguita una sola volta, allo star-up, e il body di questa funzione prevede:

1. inizializzazione dell'interfaccia di [comunicazione seriale](#);
2. inizializzazione del framework di logging;
3. inizializzazione e check del canale I2C per il sensore BME280;
4. inizializzazione della variabile *clientId* utilizzata per identificare il device come client nella connessione MQTT verso il Message Broker;
5. inizializzazione della connessione alla rete WiFi;
6. inizializzazione dei [pin mode](#) per i relè (o attuatori);
7. inizializzazione dello stato dei relè (via [digitalWrite\(\)](#)) portandoli tutti e quattro allo stato disattivato;
8. inizializzazione dell'NTP Client (con timezone UTC).

La funzione `loop()` è invece eseguita in "continuazione" e il body di questa funzione prevede:

1. l'aggiornamento del time client via NTP;
2. il controllo della connessione al Message Broker ed eventuale ri-connesione;
3. chiamata del metodo `loop()` sul client MQTT per consentire l'elaborazione dei messaggi in arrivo e mantenere la connessione al server;
4. preparazione del messaggio in formato JSON che contiene i dati ambientali provenienti dal sensore BME280;
5. pubblicazione dei dati ambientali sul topic `esp32/telemetry_data`;
6. invio sulla seriale del messaggio JSON pubblicato (per scopi di debug)

Il codice riportato è commentato nella parti salienti e non credo sia necessario commentare qui ogni linea di codice. A seguire è mostrato il codice sorgente di ogni funzione fondamentale descritta in precedenza.

Source Code 1 - Setup the WiFi Connection

```
/**  
 * Setup the WiFi Connection  
 */  
void setup_wifi()  
{  
    // We start by connecting to a WiFi network  
    Log.notice(F("Connecting to WiFi network: %s (password: %s)" CR),  
              ssid, password);  
  
    WiFi.begin(ssid, password);  
    WiFi.config(INADDR_NONE, INADDR_NONE, INADDR_NONE, INADDR_NONE);  
    WiFi.setHostname(clientId.c_str());  
  
    while (WiFi.status() != WL_CONNECTED)  
    {  
        delay(500);  
        Serial.print(".");  
    }  
  
    Serial.println("");  
  
    Serial.println("WiFi connected :-)");  
    Serial.print("IP Address: ");  
    Serial.print(WiFi.localIP());  
    Serial.println("");  
    Serial.print("Mac Address: ");  
    Serial.print(WiFi.macAddress());  
    Serial.println("");  
    Serial.print("Hostname: ");  
    Serial.print(WiFi.getHostname());  
    Serial.println("");  
    Serial.print("Gateway: ");  
    Serial.print(WiFi.gatewayIP());  
    Serial.println("");  
  
    bool success = Ping.ping(mqtt_server, 3);  
  
    if (!success)  
    {  
        Log.error(F("Ping failed to %s" CR), mqtt_server);  
        return;  
    }  
  
    Log.notice(F("Ping OK to %s" CR), mqtt_server);  
}
```

Source Code 2 - Reconnect to MQTT Broker

```
/**  
 * Reconnect to MQTT Broker  
 */  
void reconnect()  
{  
    // Loop until we're reconnected  
    while (!client.connected())  
    {  
        Log.notice(F("Attempting MQTT connection to %s" CR), mqtt_server);  
  
        // Attempt to connect  
        if (client.connect(clientId.c_str(), mqtt_username, mqtt_password))  
        {  
            Log.notice(F("Connected as clientId %s :-)" CR), clientId.c_str());  
  
            // Subscribe  
            client.subscribe(topic_command, 1);  
            Log.notice(F("Subscribe to the topic command %s " CR), topic_command);  
  
            // Init Status topic for Relay  
            update_relay_status(Relay_00, relay_status_off);  
            update_relay_status(Relay_01, relay_status_off);  
            update_relay_status(Relay_02, relay_status_off);  
            update_relay_status(Relay_03, relay_status_off);  
        }  
        else  
        {  
            Log.error(F("{failed, rc=%d try again in 5 seconds}" CR), client.state());  
            // Wait 5 seconds before retrying  
            delay(5000);  
        }  
    }  
}
```

Source Code 3 - Update Relay status on the topic

```
/**  
 * Update Relay status on the topic  
 *  
 * relayId: Identifier of the relay  
 * status: Status of the relay. (0 or 1)  
 */  
void update_relay_status(int relayId, const int status)  
{  
    // Allocate the JSON document  
    // Inside the brackets, 200 is the RAM allocated to this document.  
    // Don't forget to change this value to match your requirement.  
    // Use arduinojson.org/v6/assistant to compute the capacity.  
    StaticJsonDocument<200> relayStatus;  
  
    relayStatus["clientId"] = clientId;  
    relayStatus["deviceName"] = device_name;  
    relayStatus["time"] = timeClient.getEpochTime();  
    relayStatus["relayId"] = relayId;  
    relayStatus["status"] = status;  
  
    char relayStatusAsJson[200];  
    serializeJson(relayStatus, relayStatusAsJson);  
  
    switch (relayId)  
    {  
        case Relay_00:  
            client.publish(topic_relay_00_status, relayStatusAsJson);  
            break;  
        case Relay_01:  
            client.publish(topic_relay_01_status, relayStatusAsJson);  
            break;  
        case Relay_02:  
            client.publish(topic_relay_02_status, relayStatusAsJson);  
            break;  
        case Relay_03:  
            client.publish(topic_relay_03_status, relayStatusAsJson);  
            break;  
    }  
}
```

Source Code 4 - Return the relays status

```
/**  
 * Return the relays status  
 */  
int * get_relays_status()  
{  
    static int relaysStatus[4];  
  
    digitalWrite(Relay_00_Pin) == LOW ? relaysStatus[0] = HIGH : relaysStatus[0] = LOW;  
    digitalWrite(Relay_01_Pin) == LOW ? relaysStatus[1] = HIGH : relaysStatus[1] = LOW;  
    digitalWrite(Relay_02_Pin) == LOW ? relaysStatus[2] = HIGH : relaysStatus[2] = LOW;  
    digitalWrite(Relay_03_Pin) == LOW ? relaysStatus[3] = HIGH : relaysStatus[3] = LOW;  
  
    return relaysStatus;  
}
```

Source Code 5 - MQTT Callback

```
/**
 * MQTT Callback
 *
 * If a message is received on the topic esp32/command (es. Relay off or on).
 * Format: {$device-name}:{relay;relayId;command}
 * Es:
 *   esp32-zone-1:relay;3;off (switch off relay 3 of the specified device)
 *   esp32-zone-1:relay;2;on (switch on relay 2 of the specified device)
 *   esp32-zone-1:relay;3;status (get status of the relay 3 of the specified device)
 */
void callback(char *topic, byte *message, unsigned int length)
{
    String messageTemp;

    for (int i = 0; i < length; i++)
    {
        messageTemp += (char)message[i];
    }

    Log.notice(F("Message arrived on topic: %s" CR), topic);
    Log.notice(F("Message Content: %s" CR), messageTemp.c_str());

    if ((String)topic == (String)topic_command)
    {

        /**
         * Parsing of the received command string. This piece of code
         * could be written using regular expressions.
         * ([a-zA-Z,0-9,-]{3,12}):(\w+);([0-3]);(off|on|status) this could be the
         * regular expression which should be sufficient to satisfy the given format.
         */
        int indexOfDeviceSeparator = messageTemp.indexOf(":");

        String deviceName = messageTemp.substring(0, indexOfDeviceSeparator);
        String statement = messageTemp.substring(indexOfDeviceSeparator + 1,
                                                messageTemp.length());

        int indexOfStatementSeparator = statement.indexOf(";");
        int relayId = statement.substring(indexOfStatementSeparator + 1,
                                         indexOfStatementSeparator + 2)
                      .toInt();
        String command = statement.substring(statement.lastIndex0f(";") + 1,
                                              statement.length());

        if (!deviceName.isEmpty() && !statement.isEmpty() && !command.isEmpty() &&
            (String)device_name == deviceName)
        {
    }
}
```

```

Log.notice(F("Try to execute this statement (command %s): %s for relay %d on
the device name: %s" CR),
           command.c_str(), statement.c_str(), relayId, deviceName.c_str());

/***
 * The following code block is responsible for executing the instructions
 * received from the command topic. This block of code is purely
 * educational and can be optimizing to avoid redundant code.
 */
switch (relayId)
{
case Relay_00:
    if (command == RELAY_COMMAND_ON)
    {
        digitalWrite(Relay_00_Pin, LOW);
        update_relay_status(Relay_00, relay_status_on);

        Log.notice(F("Switch On relay 0" CR));
    }
    else if (command == RELAY_COMMAND_OFF)
    {
        digitalWrite(Relay_00_Pin, HIGH);
        update_relay_status(Relay_00, relay_status_off);

        Log.notice(F("Switch Off relay 0" CR));
    }
    else if (command == RELAY_COMMAND_STATUS)
    {
        digitalWrite(Relay_00_Pin) == LOW ? update_relay_status(Relay_00,
relay_status_on) : update_relay_status(Relay_00, relay_status_off);
    }
    break;
case Relay_01:
    if (command == RELAY_COMMAND_ON)
    {
        digitalWrite(Relay_01_Pin, LOW);
        update_relay_status(Relay_01, relay_status_on);

        Log.notice(F("Switch On relay 1" CR));
    }
    else if (command == RELAY_COMMAND_OFF)
    {
        digitalWrite(Relay_01_Pin, HIGH);
        update_relay_status(Relay_01, relay_status_off);

        Log.notice(F("Switch Off relay 1" CR));
    }
    else if (command == RELAY_COMMAND_STATUS)
    {

```

```

    digitalRead(Relay_01_Pin) == LOW ? update_relay_status(Relay_01,
relay_status_on) : update_relay_status(Relay_01, relay_status_off);
}
break;
case Relay_02:
if (command == RELAY_COMMAND_ON)
{
    digitalWrite(Relay_02_Pin, LOW);
    update_relay_status(Relay_02, relay_status_on);

    Log.notice(F("Switch On relay 2" CR));
}
else if (command == RELAY_COMMAND_OFF)
{
    digitalWrite(Relay_02_Pin, HIGH);
    update_relay_status(Relay_02, relay_status_off);

    Log.notice(F("Switch Off relay 2" CR));
}
else if (command == RELAY_COMMAND_STATUS)
{
    digitalRead(Relay_02_Pin) == LOW ? update_relay_status(Relay_02,
relay_status_on) : update_relay_status(Relay_02, relay_status_off);
}
break;
case Relay_03:
if (command == RELAY_COMMAND_ON)
{
    digitalWrite(Relay_03_Pin, LOW);
    update_relay_status(Relay_03, relay_status_on);

    Log.notice(F("Switch On relay 3" CR));
}
else if (command == RELAY_COMMAND_OFF)
{
    digitalWrite(Relay_03_Pin, HIGH);
    update_relay_status(Relay_03, relay_status_off);

    Log.notice(F("Switch Off relay 3" CR));
}
else if (command == RELAY_COMMAND_STATUS)
{
    digitalRead(Relay_03_Pin) == LOW ? update_relay_status(Relay_03,
relay_status_on) : update_relay_status(Relay_03, relay_status_off);
}
break;
default:
Log.warning(F("No relayId recognized" CR));
break;
}

```

```

    }
}
}
}
```

A seguire è mostrato il codice sorgente delle funzioni setup() e loop() descritte in precedenza.

Source Code 6 - Setup e Loop

```

/**
 * Setup lifecycle
 */
void setup()
{
    Serial.begin(115200);

    // Initialize with log level and log output.
    Log.begin(LOG_LEVEL_VERBOSE, &Serial);

    // Log ESP Chip information
    Log.notice(F("ESP32 Chip model %s Rev %d" CR), ESP.getChipModel(),
               ESP.getChipRevision());
    Log.notice(F("This chip has %d cores" CR), ESP.getChipCores());

    // Start I2C communication
    if (!bme.begin(0x76))
    {
        Log.notice("Could not find a BME280 sensor, check wiring!");
        while (1)
            ;
    }

    clientId += String(random(0xffff), HEX);

    // Connect to WiFi
    setup_wifi();

    // Setup PIN Mode for Relay
    pinMode(Relay_00_Pin, OUTPUT);
    pinMode(Relay_01_Pin, OUTPUT);
    pinMode(Relay_02_Pin, OUTPUT);
    pinMode(Relay_03_Pin, OUTPUT);

    // Init Relay
    digitalWrite(Relay_00_Pin, HIGH);
    digitalWrite(Relay_01_Pin, HIGH);
    digitalWrite(Relay_02_Pin, HIGH);
    digitalWrite(Relay_03_Pin, HIGH);
```

```

// Init NTP
timeClient.begin();
timeClient.setTimeOffset(0);
}

/**
 * Loop lifecycle
 */
void loop()
{
    while (!timeClient.update())
    {
        timeClient.forceUpdate();
    }

    long now = millis();

    if (!client.connected())
    {
        reconnect();
    }

    client.loop();

    if (now - lastMessage > interval)
    {
        lastMessage = now;

        // Allocate the JSON document
        // Inside the brackets, 200 is the RAM allocated to this document.
        // Don't forget to change this value to match your requirement.
        // Use arduinojson.org/v6/assistant to compute the capacity.
        StaticJsonDocument<256> telemetry;

        /**
         * Reading humidity, temperature and pressure
         * Temperature is always a floating point, in Centigrade. Pressure is a
         * 32 bit integer with the pressure in Pascals. You may need to convert
         * to a different value to match it with your weather report. Humidity is
         * in % Relative Humidity
         */
        temperature = bme.readTemperature();
        humidity = bme.readHumidity();
        pressure = bme.readPressure();

        telemetry["clientId"] = clientId.c_str();
        telemetry["deviceName"] = device_name;
        telemetry["time"] = timeClient.getEpochTime();
        telemetry["temperature"] = temperature;
    }
}

```

```

telemetry["humidity"] = humidity;
telemetry["pressure"] = pressure;
telemetry["interval"] = interval;
telemetry["counter"] = ++counter;

JSONArray relaysStatusJSONArray = telemetry.createNestedArray("relaysStatus");

int * relaysStatus = get_relays_status();

for (int i = 0; i <= 3; i++)
{
    relaysStatusJSONArray.add(relaysStatus[i]);
}

char telemetryAsJson[256];
serializeJson(telemetry, telemetryAsJson);

client.publish(topic_telemetry_data, telemetryAsJson);

serializeJsonPretty(telemetry, Serial);
Serial.println();
}
}

```

Ci siamo! Il prossimo step da affrontare è la fase di build e upload del software sui device ESP32.

7.1. Compilazione del progetto

La compilazione del progetto è abbastanza semplice e proprio come piace a me: **git clone e build**. Ricordo che prima di proseguire oltre, il requisito richiesto è l'installazione di PlatformIO Core sulla vostra macchina di sviluppo.

Fino a questo momento non abbiamo fatto nessun riferimento a come configurare i parametri di accesso alla rete WiFi e al Message Broker. **Cosa bisogna fare per configurare i parametri di accesso alla rete WiFi e il Message Broker?**

In questi casi solitamente preferisco percorrere la strada dei **build flags**. Questi flag influenzano i processi del pre-processore, compilazione, assemblaggio e collegamento per il codice C e C++. È possibile utilizzare tutti i flag del compilatore e del linker. In questo caso utilizzeremo il build flag **-D name=definition** che influenza sulla variabile di build **CPPDEFINES**. I contenuti della definizione (definition) vengono tokenizzati ed elaborati come se fossero apparsi durante la terza fase di traduzione in una direttiva `#define`. Per maggiori informazioni invito al leggere **C preprocessor**.

All'inizio del codice sorgente [esp32_mqtt_publish_subscribe.cpp](#) ci sono le varie sezione **ifdef...endif** per la lettura di tutti i build flag. A seguire sono mostrati i build flag definiti sul file [platformio.ini](#) del progetto. Tra i build flag, oltre a quelli che riguardano i parametri di accesso

alla rete WiFi e al Message Broker, abbiamo anche un parametro di build che imposta il nome del device. Da notare che i valori di questi parametri provengono dalle relative variabili di ambiente.

Configurazione 1 - Configurazione di PlatformIO

```
[env:esp32dev]
platform = espressif32 ①
board = esp32dev ②
framework = arduino ③
build_flags = ④
-DWIFI_SSID=${sysenv.WIFI_SSID}
-DWIFI_PASSWORD=${sysenv.WIFI_PASSWORD}
-DMQTT_USERNAME=${sysenv.MQTT_USERNAME}
-DMQTT_PASSWORD=${sysenv.MQTT_PASSWORD}
-DMQTT_SERVER=${sysenv.MQTT_SERVER}
-DMQTT_PORT=${sysenv.MQTT_PORT}
-DDEVICE_NAME=${sysenv.DEVICE_NAME}
```

① Specifica la piattaforma di destinazione

② Specifica il tipo di board

③ Specifica il framework

④ Specifica quali flag usare per la build del progetto

Prima di proseguire con il processo di compilazione, occorre quindi impostare le variabili di ambiente sopra indicate. Dato che abbiamo due device su cui installare il software, facciamo attenzione a modificare la variabile di ambiente **DEVICE_NAME**. Questa variabile di ambiente può assumere i seguenti valori: esp32-zone-1 e esp32-zone-2. L'informazione sul device name è riportata sul messaggio JSON che contiene i dati ambientali.

A seguire sono indicati i comandi necessari per portare a buon fine la compilazione del progetto e generare così l'artefatto (**firmware.bin**) che andrà successivamente installato sul device ESP32.

Console 3 - Procedura per la compilazione del progetto e generazione firmware

```

# Export Environment Variables
export WIFI_SSID="your-wifi-ssid"
export WIFI_PASSWORD="your-wifi-access-password"

# Username e password defined and created
# in the previous step (EMQ X Edge Install)
export MQTT_USERNAME="esp32-device"
export MQTT_PASSWORD="esp32-device"
export MQTT_SERVER="your-ip-address-of-mqtt-broker"
export MQTT_PORT="1883"

# Export Device Name
export DEVICE_NAME="esp32-zone-1"

# Check version of the PlatformIO Core
pio --version

# Update installed platforms, packages and libraries
pio update

# Upgrade PlatformIO to the latest version
pio upgrade

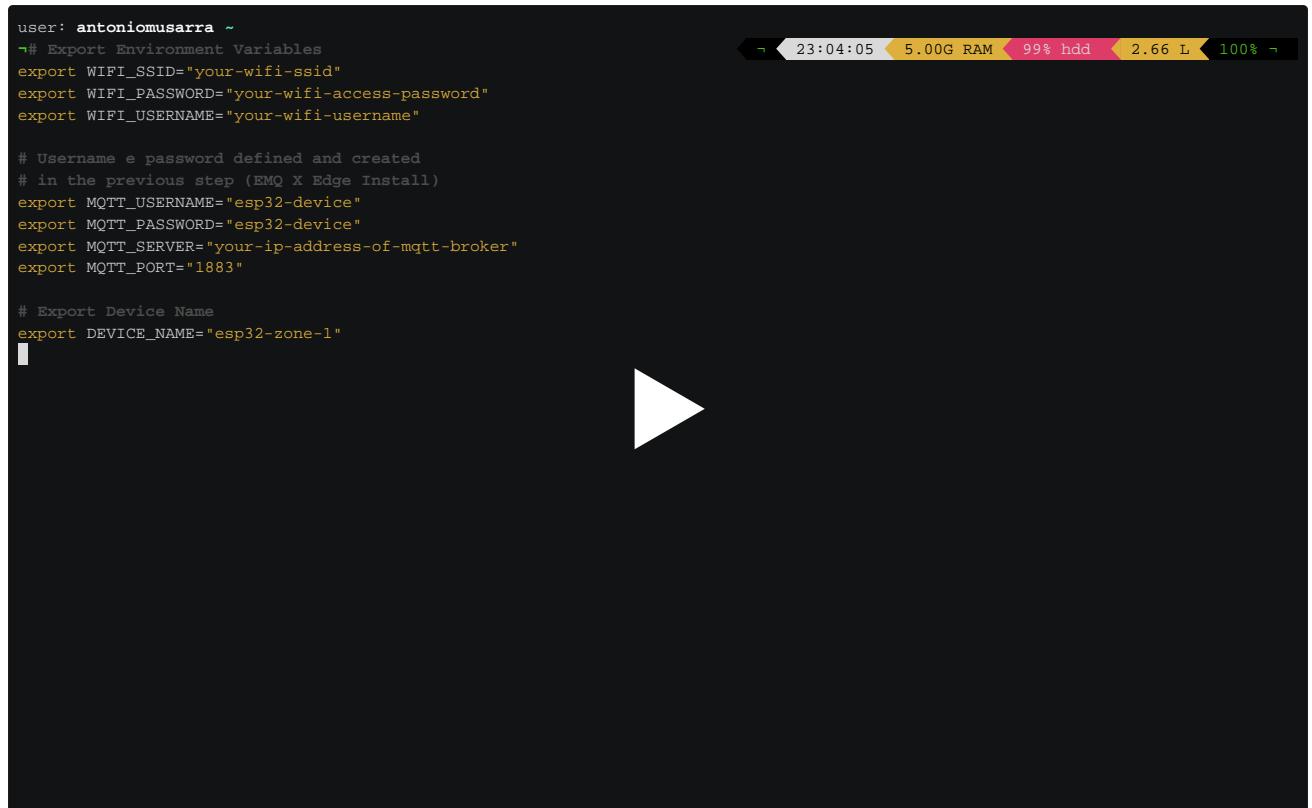
# Clone Project from GitHub esp32-mqtt-publish-subscribe
git clone https://github.com/amusarra/esp32-mqtt-publish-subscribe.git

# Build Project
cd esp32-mqtt-publish-subscribe
pio run --environment esp32dev

```

Durante il processo di build, PlatformIO scarica anche le librerie da cui il progetto dipende e che sono indicate nella sezione **lib_deps** del file platformio.ini (vedi anche Tabella 5).

A seguire è mostrato lo screencast di [Compiling the ESP32 MQTT Publish/Subscribe project with PlatformIO](#) in modo che possiate vedere esattamente l'esecuzione degli step indicati in precedenza.



```

user: antoniomusarra ~
# Export Environment Variables
export WIFI_SSID="your-wifi-ssid"
export WIFI_PASSWORD="your-wifi-access-password"
export WIFI_USERNAME="your-wifi-username"

# Username e password defined and created
# in the previous step (EMQ X Edge Install)
export MQTT_USERNAME="esp32-device"
export MQTT_PASSWORD="esp32-device"
export MQTT_SERVER="your-ip-address-of-mqtt-broker"
export MQTT_PORT="1883"

# Export Device Name
export DEVICE_NAME="esp32-zone-1"

```

Screencast 3 - Compiling the ESP32 MQTT Publish/Subscribe project with PlatformIO

Nel caso in cui installiate o abbiate già installato PlatformIO per VSCode, le stesse operazioni di compilazione del progetto posso essere portate a termine anche dall'IDE, così come mostrato dalla figura a seguire.

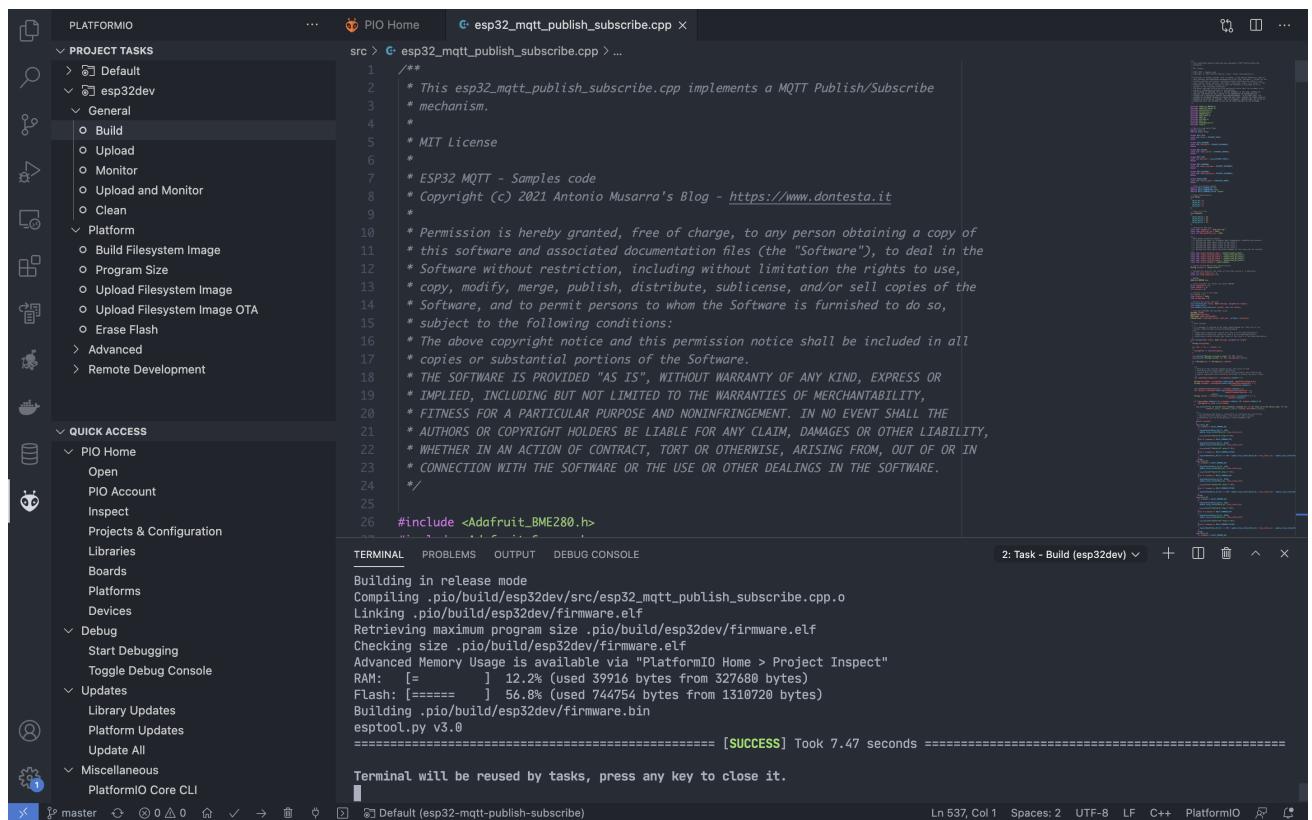


Figura 14 - PlatformIO IDE per VSCode

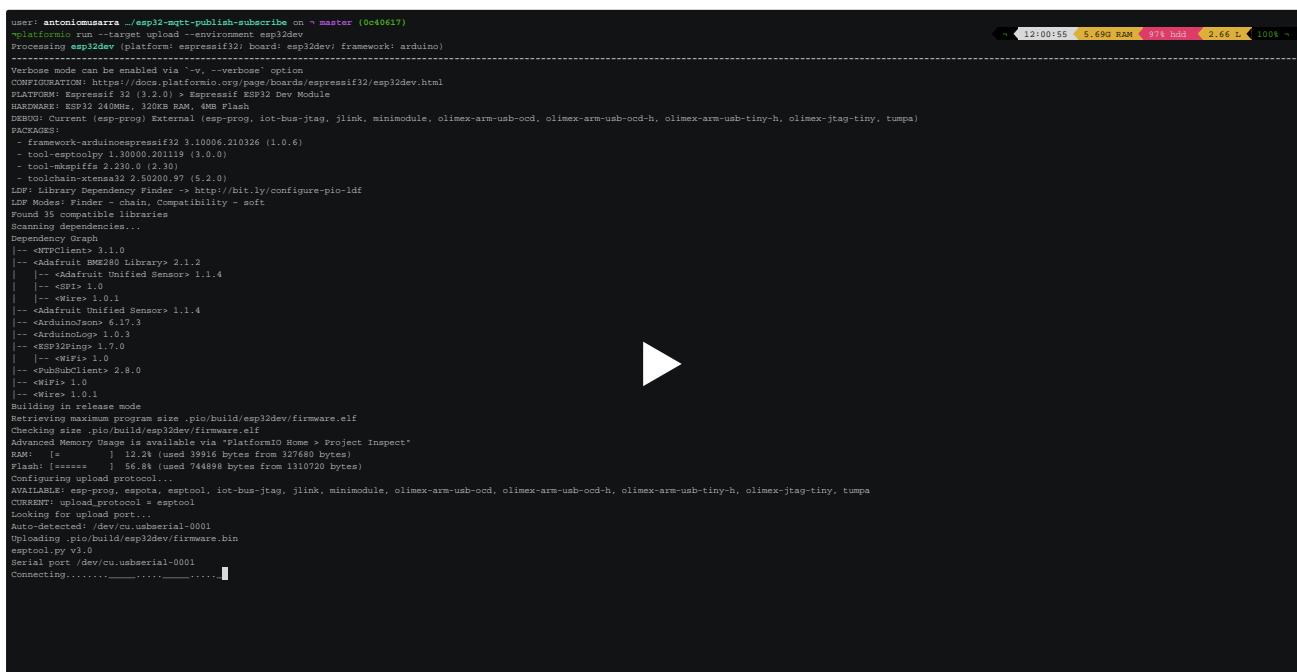
Una volta accertato che la compilazione del progetto sia andata a buona fine, possiamo passare alla fase di upload del software sui due device ESP32.

7.2. Upload del software sul device ESP32

Siamo arrivati allo step finale per quel che riguarda i device ESP32, ovvero, l'upload di quello che può essere anche definito il firmware dei nostri device. L'operazione di upload è sempre demandata a PlatformIO utilizzando il semplice comando: `pio run --target upload --environment esp32dev`. La stessa operazione è possibile eseguirla anche attraverso l'IDE.

In realtà è anche possibile saltare il processo di compilazione del progetto, il processo di upload, in caso fosse necessario, penserà ad avviare la compilazione del codice per proseguire poi con l'upload sull'ESP32.

A seguire è mostrato lo screencast di [Upload software to ESP32 Device via PlatformIO](#) in modo che possiate vedere esattamente l'esecuzione della fase di upload del firmware .pio/build/esp32dev/firmware.bin sul device ESP32.



```

user: antoniuomusarr ~ esp32-mqtt-publish-subscribe on ~ master (0c40617)
$platformio run --target upload --environment esp32dev
Processing esp32dev (platform: espressif32) board: esp32devi framework: arduino
Verbosity mode can be enabled via "--verbose" option
CONFIGURATION: https://docs.platformio.org/page/boards/espressif32/esp32dev.html
PLATFORM: Espressif 32 (3.2.0) > Espressif ESP32 Dev Module
HARDWARE: ESP32 240MHz, 320KB RAM, 4MB Flash
DEBUG: Current (esp-prog, iot-bus-jtag, jlink, minimodule, olimex-arm-usb-ocd, olimex-arm-usb-ocd-h, olimex-arm-usb-tiny-h, olimex-jtag-tiny, tumpa)
PACKAGES:
- framework_arduinoespressif32 3.10006.210326 (1.0.6)
- tool-esp32ulp 1.20000.201119 (3.0.0)
- tool-esp32if 2.230.0 (2.30)
- toolchain-xtensa32 2.50200.97 (5.5.0)
LDF: Library Dependency Finder -> http://bit.ly/configure-pio-ldf
LDF Modes: Finder - chain, Compatibility - soft
Found 35 compatible libraries
Scanning dependencies...
Dependencies (35):
- <HTTPClient> 1.1.0
--- <Adafruit_BME280 Library> 2.1.2
| |- <Adafruit Unified Sensor> 1.1.4
| |- <SPI> 1.0
| | |- <Wire> 0.1
--- <Adafruit Unified Sensor> 1.1.4
--- <ArduinoJson> 6.17.3
--- <ArduinoLoops> 1.0.0
--- <ESP32Ping> 1.7.0
| |- <WiFi> 1.0
--- <PubSubClient> 2.8.0
| |- <WiFi> 1.0
--- <WiFi> 1.0.1
Building in release mode
Retrieving maximum program size .pio/build/esp32dev/firmware.elf
Checking size .pio/build/esp32dev/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM: [ ] 12.2k (used 39916 bytes from 327680 bytes)
Flash: [ ] 56.0k (used 74498 bytes from 1310720 bytes)
Configuring upload protocol...
AVAILABLE: esp-prog, espota, espTool, iot-bus-jtag, jlink, minimodule, olimex-arm-usb-ocd, olimex-arm-usb-ocd-h, olimex-arm-usb-tiny-h, olimex-jtag-tiny, tumpa
CURRENT: upload_protocol = espTool
Looking for upload port...
Auto-detected: /dev/cu.usbserial-0001
Uploading .pio/build/esp32dev/firmware.bin
esp32: v3.0
serial port: /dev/cu.usbserial-0001
Connecting.....

```

Screencast 4 - Upload software to ESP32 Device via PlatformIO

Sull'immagine a seguire è possibile notare lo stato dei led dopo l'upload del software sui device e in particolare del led di colore blue. Quando il led blue è acceso in modo fisso, vuol dire che:

1. la connessione alla rete WiFi è andata a buon fine;
2. il ping verso il Message Broker è andato a buon fine;
3. la connessione verso il Message Broker EMQ X Edge è andata a buone fine;
4. la sottoscrizione al topic esp32/command è andata a buon fine.

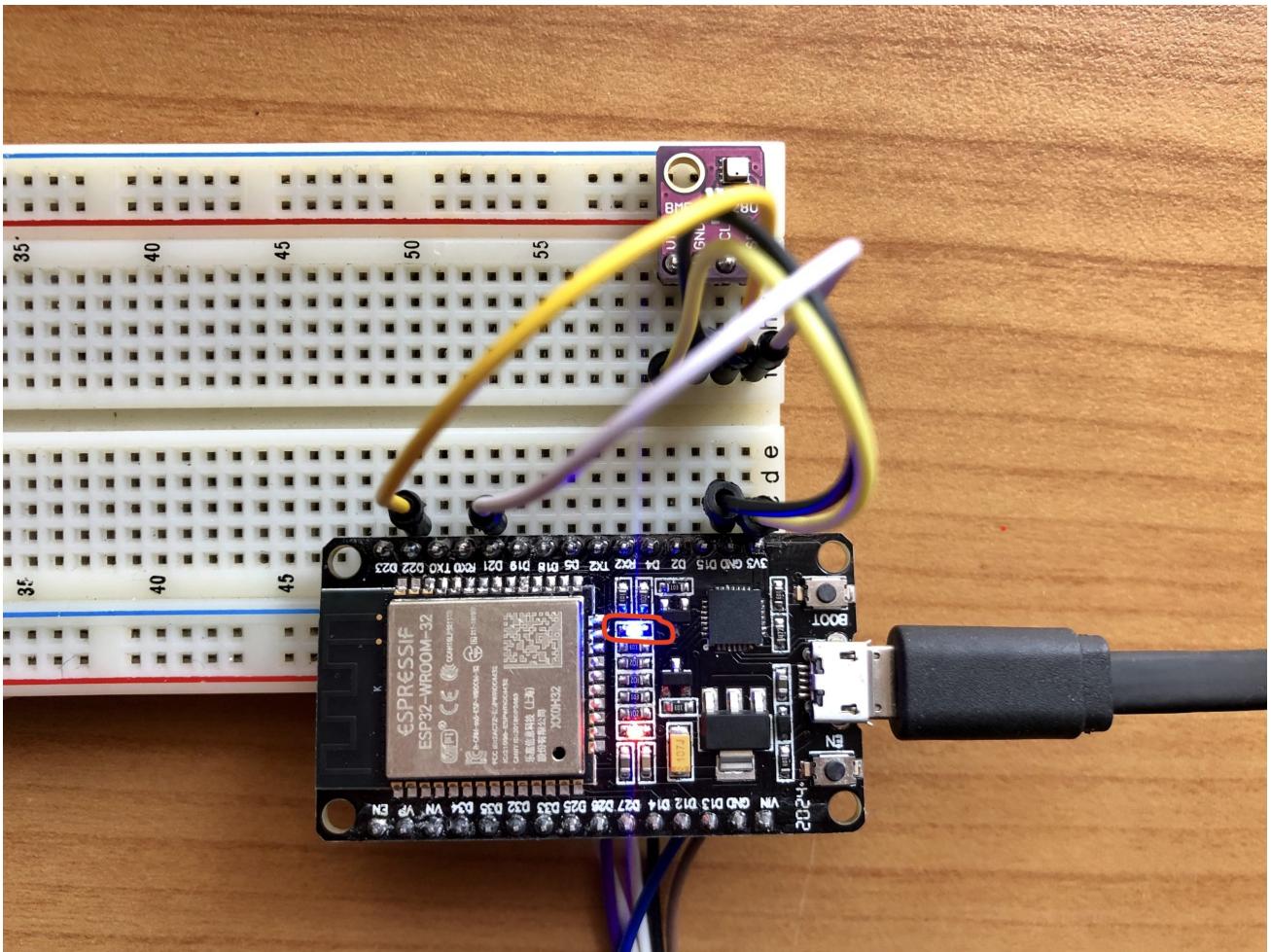


Figura 15 - Indicazione dell'operatività del device ESP32 tramite il led blue presente sulla scheda di sviluppo

In definitiva questo led ([attestato sul GPIO 2](#)) ci da indicazioni se il tutto sta andando per il verso giusto. In questo caso siamo fortunati, il led è acceso e siamo quindi sicuri che i dati ambientali vengono pubblicati sul topic esp32/telemetry_data e che il dispositivo è pronto a ricevere comandi sul topic esp32/command. Se volessimo controllare l'attività del device, potremmo connetterci al monitor seriale sfruttando sempre PlatformIO, utilizzando il comando: `pio device monitor --environment esp32dev`

L'immagine a seguire mostra l'output del comando indicato in precedenza. Com'è possibile vedere, otteniamo diverse informazioni in output. Informazioni sul modello del chip, informazioni sulla rete WiFi a cui è connesso il device, informazioni sul Message Broker a cui il device è connesso e per finire il JSON pubblicato sul topic esp32/telemetry_data.

```

> pio device monitor --environment esp32dev
--- Available filters and text transformations: colorize, debug, default, direct, esp32_exception_decoder, hexlify, log2file, nocontrol, printable, send_on_enter, time
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/cu.usbserial-0001 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---

load:0x3ff-00j
load:0x40078000,len:10124
load:0x40080400,len:5828
entry 0x400806a8
N: ESP32 Chip model ESP32-D0WDQ6 Rev 1
N: This chip has 2 cores
N: Connecting to WiFi network: TIM-22816093 (password: ██████████)
.....
WiFi connected :)
IP Address: 192.168.1.125
Mac Address: 4C:11:AE:8B:53:B0
Hostname: esp32-client-11fa
Gateway: 192.168.1.1
N: Ping OK to 192.168.1.124
N: Attempting MQTT connection to 192.168.1.124
N: Connected as clientId esp32-client-11fa :)
N: Subscribe to the topic command esp32/command
{
  "clientId": "esp32-client-11fa",
  "deviceName": "esp32-zone-1",
  "time": 1618223579,
  "temperature": 19.49,
  "humidity": 39.67969,
  "pressure": 93904,
  "interval": 5000,
  "counter": 1,
  "relaysStatus": [
    0,
    0,
    0,
    0
  ]
}
{
  "clientId": "esp32-client-11fa",
  "deviceName": "esp32-zone-1",
  "time": 1618223584,
  "temperature": 19.51,
  "humidity": 39.68066,
  "pressure": 93902,
  "interval": 5000,
  "counter": 2,
  "relaysStatus": [
    0,
    0,
    0,
    0
  ]
}

```

<https://www.dontesta.it> | [@anto](https://github.com/amusarra)



Antonio

The ideal solu

20
21
22
23
24
25
26
27
28

Figura 16 - Attach del monitor seriale su uno dei device ESP32 al fine di monitorare le attività in corso

Dobbiamo ripetere le stesse operazioni di compilazione e upload del software anche per il secondo device e una volta fatto possiamo definire conclusa questa fase e andare a realizzare il primo test d'integrazione.

8. Primo test d'integrazione

Siamo arrivati a un bel punto! Ci sono le condizioni necessarie per poter fare i primi test d'integrazione. **In cosa consistono questi test d'integrazione?**

In questo istante quello che abbiamo è il Message Broker installato sul Raspberry Pi e i due device ESP32 con il software appena "flashato". Prima di procedere con il test d'integrazione dobbiamo accendere il Raspberry Pi e avviare il Message Broker e successivamente alimentare i due device ESP32. Il test d'integrazione prevede la verifica dei seguenti punti:

1. Dalla dashboard di EMQ X Edge verificare che risultino connessi i due device ESP32 (dalla voce di menù Clients). I due device connessi dovrebbero essere identificati dal prefisso **esp32-client-** (così come scritto sul [codice sorgente](#));
2. Connettersi ai Serial Monitor (via PlatformIO) dei due device ESP32 e accettare che entrambi i dispositivi siano connessi alla rete WiFi e che stiano pubblicando i dati ambientali a intervalli regolari di cinque secondi;
3. Dalla dashboard di EMQ X Edge, attraverso il tool Websocket eseguire la sottoscrizione al topic `esp32/telemetry_data` utilizzando l'account `gw-rpi4-device` e verificare la ricezione dei messaggi contenenti i dati ambientali che provengono dai due dispositivi;
4. Dalla dashboard di EMQ X Edge, attraverso il tool Websocket eseguire la pubblicazione dei comandi di attivazione/disattivazione relè e di status dei relè sul topic `esp32/command` (per entrambi i device), utilizzando sempre l'account `gw-rpi4-device`. Tramite il Serial Monitor verificare la corretta ricezione dei messaggi su entrambi i device e l'effettiva esecuzione dei comandi.

Le figure successive mostrano i task di verifica descritti nel precedente elenco.

Client ID	Username	IP Address	Keepalive(s)	Expiry Interval(s)	Subscriptions Count	Connect Status	Created At	Operation
esp32-client-e770	esp32-device	192.168.1.126.64022	15	0	1	CONNECTED	2021-04-12 10:57:28	Kick Out
esp32-client-2c2a	esp32-device	192.168.1.125.63126	15	0	1	CONNECTED	2021-04-12 10:59:26	Kick Out

Figura 16 - Client connessi al Message Broker

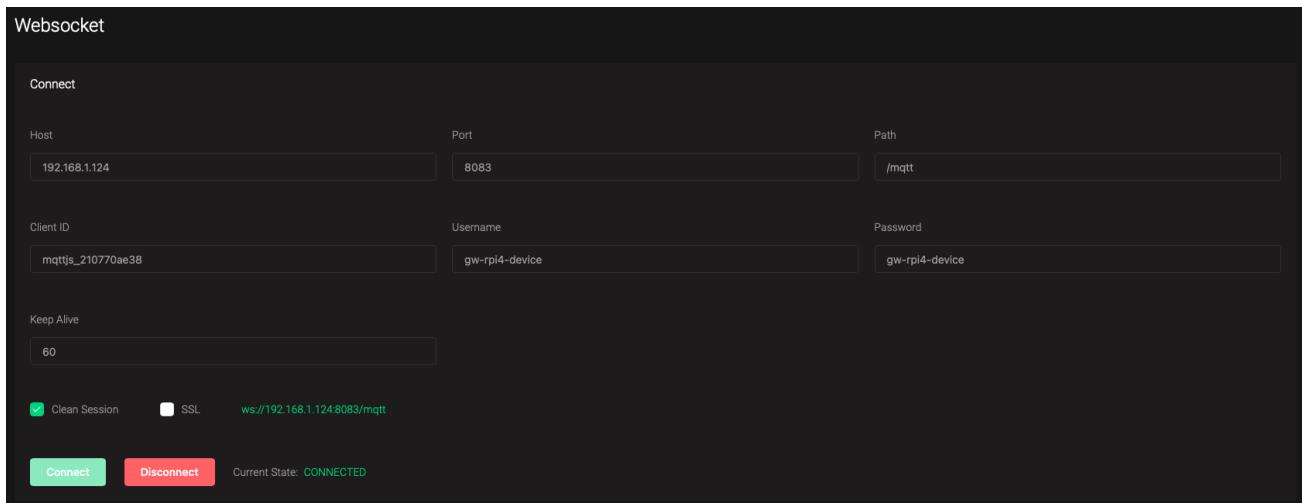


Figura 17 - Test di connessione al Message Broker tramite la porta Websocket specificando l'utente gw-rpi4-device

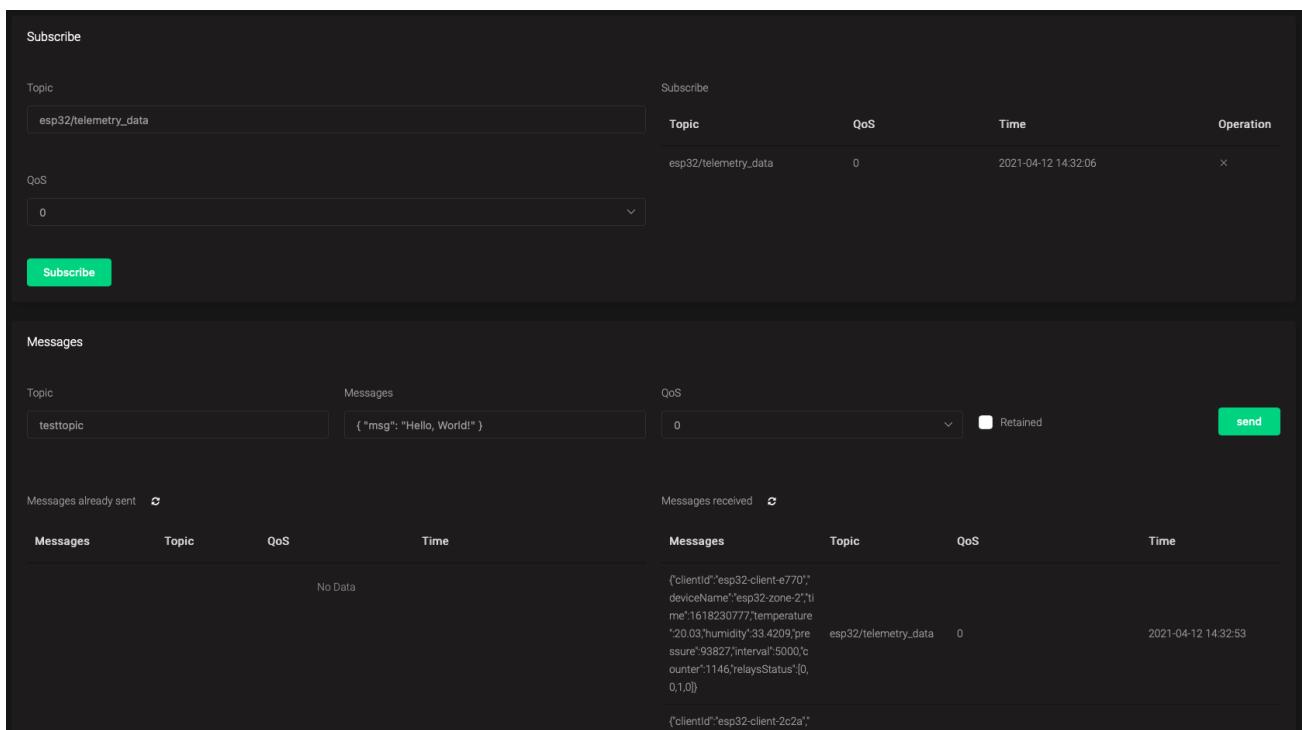


Figura 18 - Test di sottoscrizione al topic esp32/telemetry_data utilizzando l'utente gw-rpi4-device

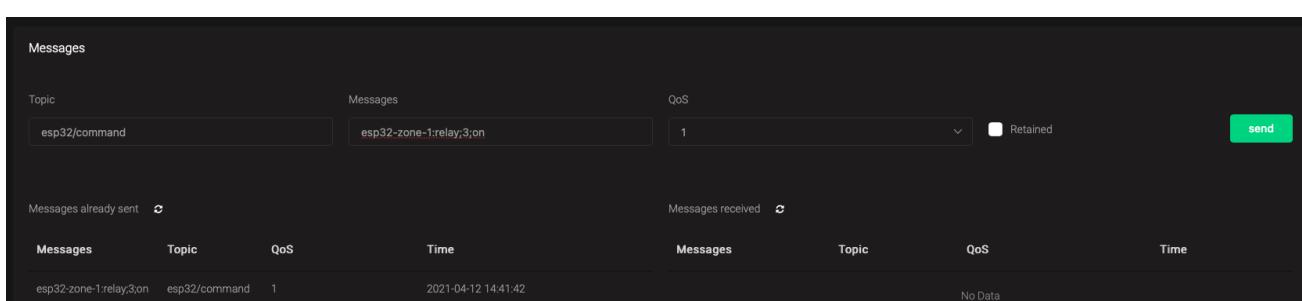


Figura 19 - Test di pubblicazione sul topic esp32/command con l'utente gw-rpi4-device

Una volta verificato che tutti i punti siano soddisfatti, possiamo ritenere concluso il test d'integrazione ed essere felici per il risultato raggiunto.

9. Scrrittura del software per il display LCD

Non è ancora finita! **Ricordate il display LCD che abbiamo connesso al Raspberry Pi?** Utilizzeremo il display LCD per visualizzare i dati ambientali pubblicati dai device ESP32 e per visualizzare lo stato dei relè nel caso in cui sia intervenuto un cambio di stato.

Il software che andremo a scrivere (utilizzando il linguaggio Python) sarà quindi responsabile di:

1. effettuare la connessione al Message Broker utilizzando l'utenza gw-rpi4-device;
2. effettuare la sottoscrizione al topic esp32/telemetry_data;
3. effettuare la sottoscrizione ai topic esp32/releay_{id}_status;
4. leggere i dati ambientali ricevuti dal Message Broker e visualizzare le informazioni di temperatura, umidità e pressione per ogni device ESP32 sul display LCD;
5. attivare/disattivare i relè sulla base di un threshold impostato sulla temperatura inviando il comando adeguato sul topic esp32/command.

Il software in questione sarà eseguito sul Raspberry Pi (quindi il Gateway), è pertanto necessario che sia installato almeno Python 3.8 e la libreria [paho](#), quest'ultima può essere installata tramite il comando: `pip install paho-mqtt`.

Se ricordate bene, dal diagramma di Figura 2, la responsabilità di inviare messaggi di comando verso il topic esp32/command è della dashboard ma ho voluto mostrare come ottenere lo stesso risultato anche a questo livello.

Lo script [mqtt_topic_subscribe_display_lcd.py](#) che implementa le cinque funzionalità sopra descritte non presenta particolari complessità, per questo motivo vi invito a visionare il codice direttamente sul repository GitHub. Parecchio del codice Python deriva da questi due miei articoli: [Un primo maggio 2020 a base di Raspberry Pi, Bot Telegram, Display LCD e Relè](#) e [Raspberry PI Sense HAT: Come pubblicare i dati su CloudAMQP via MQTT](#).

Se adesso volessimo visionare i dati ambientali pubblicati dai due device e lo stato dei relè direttamente sul display LCD collegato al Raspberry Pi, non dovremmo fare altro che eseguire il clone del repository <https://github.com/amusarra/rpi-mqtt-topic-subscribe.git> sul Gateway e lanciare successivamente il comando (dall'interno della directory del progetto):

Console 5 - Lettura dei dati ambientali

```
$ ./mqtt_topic_subscribe_display_lcd.py -s localhost -p 1883 -u gw-rpi4-device -P gw-rpi4-device`.
```

Il comando prende in input il nome o indirizzo ip del Message Broker, la porta TCP/IP e username/password di accesso. All'avvio dello script, in console vedremo diverse informazioni utili a capire ciò che sta accadendo.

A seguire è mostrato lo screencast di [How to view environmental data via MQTT on LCD connected to the Raspberry Pi](#) in modo che possiate vedere esattamente l'esecuzione degli step indicati in precedenza.

```
amusarra@amusarra-pi-4b:~$ cd rpi-mqtt-topic-subscribe/
amusarra@amusarra-pi-4b:~/rpi-mqtt-topic-subscribe$ ls -lrt
total 32
-rwxrwxr-x 1 amusarra amusarra 6413 Apr 12 16:39 mqtt_topic_subscribe_display_lcd.py
-rw-rw-r-- 1 amusarra amusarra 44 Apr 12 16:39 README.markdown
-rw-rw-r-- 1 amusarra amusarra 2463 Apr 12 16:39 PCF8574.py
-rw-rw-r-- 1 amusarra amusarra 1142 Apr 12 16:39 LICENSE.markdown
-rw-rw-r-- 1 amusarra amusarra 296 Apr 12 16:39 CHANGELOG.markdown
-rw-rw-r-- 1 amusarra amusarra 7511 Apr 12 16:39 Adafruit_LCD1602.py
amusarra@amusarra-pi-4b:~/rpi-mqtt-topic-subscribe$ ./mqtt_topic_subscribe_display_lcd.py -s localhost -p 1883 -u gw-rpi4-device -P gw-rpi4-device
2021-04-12 16:39:33,135 :: INFO :: <module> :: 219 :: Raspberry Pi MQTT Subscriber is starting...
2021-04-12 16:39:33,317 :: INFO :: on_log :: 88 :: Sending CONNECT (ul, pl, wr0, wq0, wf0, c1, k60) client_id=b'amusarra-pi-4b-5758eb18-7571-4b3e-9dal-16569323b5be'
2021-04-12 16:39:33,320 :: INFO :: on_log :: 88 :: Received CONNACK (0, 0)
2021-04-12 16:39:33,321 :: INFO :: on_connect :: 93 :: Connected to MQTT broker (RC: 0)
2021-04-12 16:39:33,321 :: INFO :: on_log :: 88 :: Sending SUBSCRIBE (d0, m1) [(b'esp32/telemetry_data', 0)]
2021-04-12 16:39:33,322 :: INFO :: on_log :: 88 :: Sending SUBSCRIBE (d0, m2) [(b'esp32/relay_00_status', 1)]
2021-04-12 16:39:33,322 :: INFO :: on_log :: 88 :: Sending SUBSCRIBE (d0, m3) [(b'esp32/relay_01_status', 1)]
2021-04-12 16:39:33,322 :: INFO :: on_log :: 88 :: Sending SUBSCRIBE (d0, m4) [(b'esp32/relay_02_status', 1)]
2021-04-12 16:39:33,323 :: INFO :: on_log :: 88 :: Sending SUBSCRIBE (d0, m5) [(b'esp32/relay_03_status', 1)]
2021-04-12 16:39:33,325 :: INFO :: on_log :: 88 :: Received SUBACK
2021-04-12 16:39:33,326 :: INFO :: on_subscribe :: 84 :: Data subscribe (Mid: 1)
2021-04-12 16:39:33,326 :: INFO :: on_log :: 88 :: Received SUBACK
2021-04-12 16:39:33,326 :: INFO :: on_subscribe :: 84 :: Data subscribe (Mid: 2)
2021-04-12 16:39:33,327 :: INFO :: on_log :: 88 :: Received SUBACK
2021-04-12 16:39:33,327 :: INFO :: on_subscribe :: 84 :: Data subscribe (Mid: 3)
2021-04-12 16:39:33,327 :: INFO :: on_log :: 88 :: Received SUBACK
2021-04-12 16:39:33,328 :: INFO :: on_subscribe :: 84 :: Data subscribe (Mid: 4)
2021-04-12 16:39:33,328 :: INFO :: on_log :: 88 :: Received SUBACK
2021-04-12 16:39:33,328 :: INFO :: on_subscribe :: 84 :: Data subscribe (Mid: 5)
2021-04-12 16:39:33,753 :: INFO :: on_log :: 88 :: Received PUBLISH (d0, q0, r0, m0), 'esp32/telemetry_data', ... (190 bytes)
2021-04-12 16:39:33,753 :: INFO :: on_message :: 120 :: message received ["clientId": "esp32-client-2c2a", "deviceName": "esp32-zone-1", "time": 1618245573, "temperature": 20.2, "humidity": 39.62988, "pressure": 93699, "interval": 5000, "counter": 4077, "relaysStatus": [0, 0, 1, 0]}
2021-04-12 16:39:33,754 :: INFO :: on_message :: 121 :: message topic: esp32/telemetry_data
2021-04-12 16:39:33,754 :: INFO :: on_message :: 122 :: message qos=0
2021-04-12 16:39:33,755 :: INFO :: on_message :: 123 :: message retain flag=0
2021-04-12 16:39:35,916 :: INFO :: on_log :: 88 :: Received PUBLISH (d0, q0, r0, m0), 'esp32/telemetry_data', ... (191 bytes)
2021-04-12 16:39:35,916 :: INFO :: on_message :: 120 :: message received ["clientId": "esp32-client-e770", "deviceName": "esp32-zone-2", "time": 1618245574, "temperature": 20.12, "humidity": 40.39063, "pressure": 93731, "interval": 5000, "counter": 4101, "relaysStatus": [0, 0, 1, 1]}
2021-04-12 16:39:35,917 :: INFO :: on_message :: 121 :: message topic: esp32/telemetry_data
2021-04-12 16:39:35,917 :: INFO :: on_message :: 122 :: message qos=0
2021-04-12 16:39:35,917 :: INFO :: on_message :: 123 :: message retain flag=0
```

Screencast 5 - How to view environmental data via MQTT on LCD connected to the Raspberry Pi

Le due immagini a seguire mostrano le diverse informazioni che sono visualizzate sul display LCD. Le informazioni fanno riferimento in particolare ai dati ambientali di temperatura e umidità e allo stato dei relè. Sul display LCD è riportata inoltre l'informazione di quale dispositivo sia la sorgente dati.

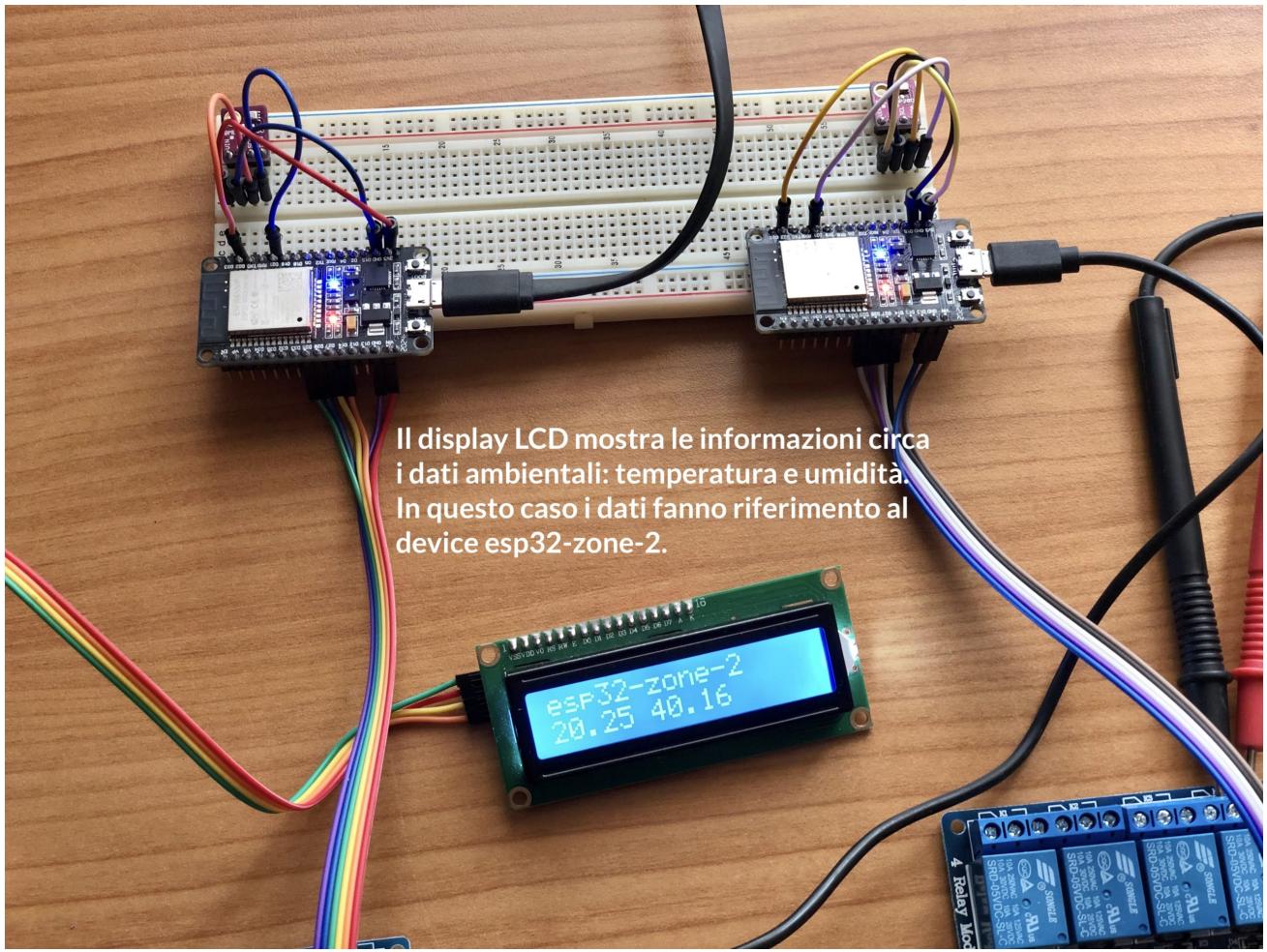


Figura 20 - Display LCD che mostra i dati ambientali di temperatura e umidità per ogni device

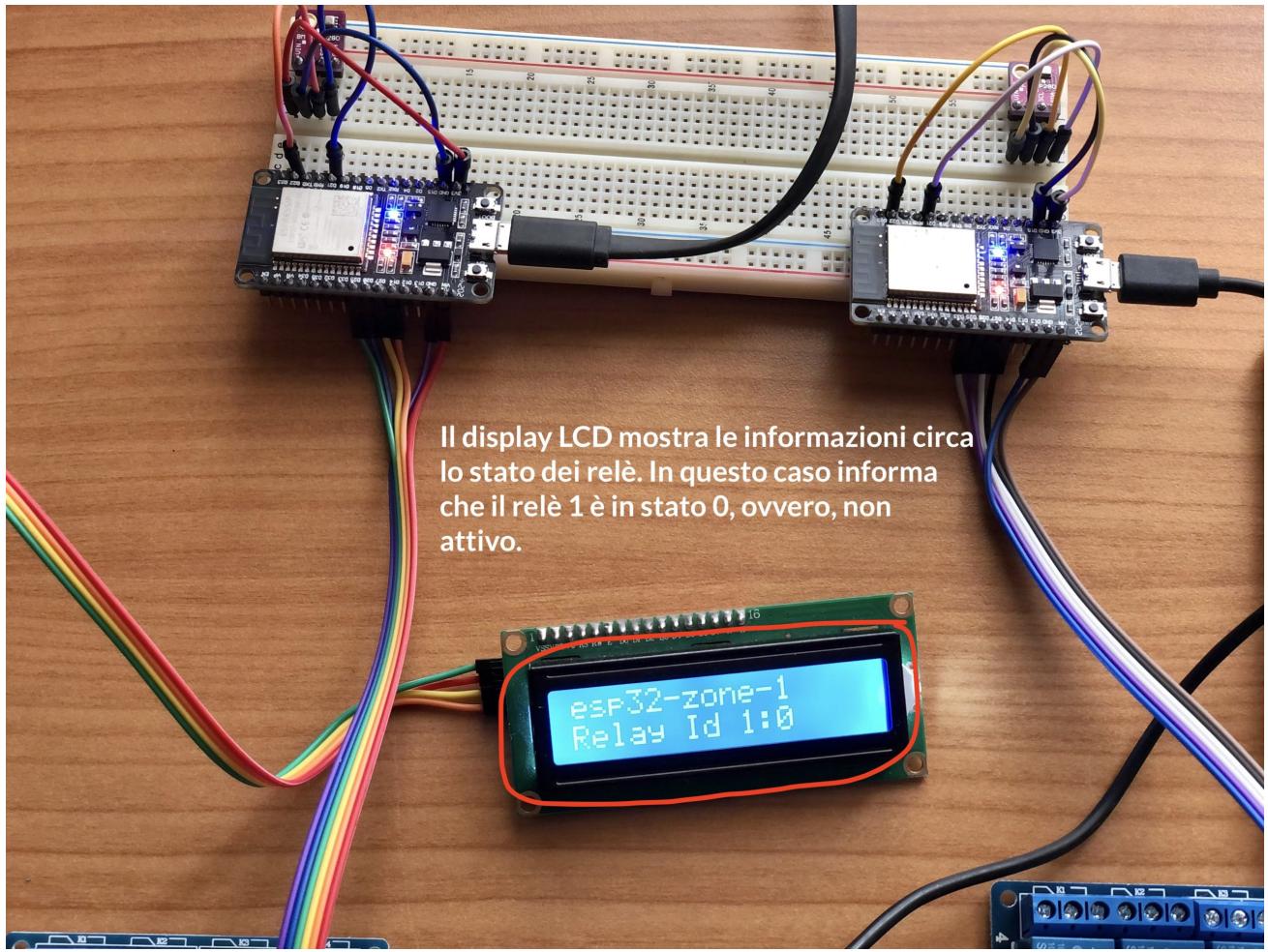


Figura 21 - Display LCD che mostra lo stato dei relè per ogni device

Step by step ci stiamo avvicinando alla metà. Il prossimo passo riguarda l'implementazione della dashboard.

10. Implementazione della Dashboard

Per completare la soluzione IoT presentata all'inizio, dobbiamo in qualche modo realizzare la dashboard la cui responsabilità è la visualizzazione dei dati ambientali e l'attivazione/disattivazione dei relè sia in modalità manuale sia in modalità automatica, ovvero, sulla base di valori di threshold (o di soglia) per temperatura, umidità e pressione.

Lo strumento che ho deciso di adottare per assolvere questo compito è [Node-RED](#).

Node-RED è uno tra i più noti tool di **flow-based programming** (o FBP) per l'Internet of Things. Questo strumento consente di dare a tutti, anche a chi non è esperto di programmazione, la possibilità di collegare tra loro diversi dispositivi (con eventuali relativi sensori ed attuatori), oltre a API e servizi online per poter realizzare sistemi altamente integrati e complessi in modo del tutto semplice e intuitivo. È molto utile anche in fase di prototipazione di un progetto per individuare da subito funzionalità e potenzialità.

Quelli indicati a seguire sono i tre punti che ho preso in considerazione per adottare questo strumento.

- Poiché può essere eseguito su dispositivi edge (preinstallati su versioni specifiche del sistema operativo Raspberry Pi), è ideale per la gestione dei dati a livello di dispositivo.
- Poiché può essere eseguito in ambiente cloud (fornito come servizio predefinito in IBM Cloud), è facile collegarlo al middleware di archiviazione e analisi.
- Supporta i protocolli MQTT e HTTP, quindi è molto facile scambiare dati tra il dispositivo e il server di elaborazione, che può risiedere anche in ambiente cloud.

Il primo step da fare è l'installazione di Node-RED sul nostro Gateway, ovvero, sul Raspberry Pi. Ormai Docker è il nostro amico, procederemo quindi con [l'installazione di Node-RED via Docker](#). A seguire sono indicati i comandi necessari per installare l'ultima versione di Node-RED (in questo momento la 1.3.1).

Console 6 - Setup Node-RED

```
# Create and run Node-RED Docker container
docker run -d --network edge-iot -p 1880:1880 --name node-red nodered/node-red:1.3.1

# Install Dashboard
docker exec -it node-red /bin/bash ①

# Inside the Node-RED container
npm install node-red-dashboard ②
exit

# On the Host
docker restart node-red
```

① Comando per ottenere la shell sul container di Node-RED

② Installazione della Dashboard NodeRED via `npm`

Come potete vedere dal primo comando Docker, ho inserito il container di Node-RED all'interno della stessa rete del Message Broker (EMQ X Edge). È possibile vedere l'effetto utilizzando il comando `docker network inspect edge-iot` che mostra in questo caso i due container che fanno parte della rete edge-iot e che sono: l'mqtt-broker e node-red. La figura a seguire mostra parte dell'output del comando.



```
"Containers": {
    "03611992f56745113b7d40cc964a0f64a859ed43294cdfdca8bfaa32ee34b74c": {
        "Name": "mqtt-broker",
        "EndpointID": "fae790691d4a934e015399035eddd1115065331be2b1edb5c52e51675070de24",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    },
    "103c8c5f1afdb20cf859c3a7594261d037f1a259905ecf80e9e21d5dec463e95": {
        "Name": "node-red",
        "EndpointID": "fe83b0b40f9673464ac37dbe7408a4709fc6784bc337fb3776358fe3043314b",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    }
},
"Options": {},
"Labels": {}
}
]
amusarra@amusarra-pi-4b:~$ docker network inspect edge-iot
```

Figura 22 - Inspect della rete Docker iot-edge creata per il Message Broker e Node-RED

A seguire è mostrato lo screencast di [Install Node-RED 1.3.1 via Docker on Raspberry Pi 4](#) in modo che possiate vedere esattamente l'esecuzione degli step indicati in precedenza.

```
> node-red-docker@1.3.1 start /usr/src/node-red
> node $NODE_OPTIONS node_modules/node-red/red.js $FLOWS "--userDir" "/data"

11 Apr 20:26:56 - [info]

Welcome to Node-RED
=====
11 Apr 20:26:56 - [info] Node-RED version: v1.3.1
11 Apr 20:26:56 - [info] Node.js version: v10.24.1
11 Apr 20:26:56 - [info] Linux 5.4.0-1032-raspi arm64 LE
11 Apr 20:26:57 - [info] Loading palette nodes
11 Apr 20:26:58 - [info] Settings file : /data/settings.json
11 Apr 20:26:58 - [info] Context store : 'default' [module=memory]
11 Apr 20:26:58 - [info] User directory : /data
11 Apr 20:26:58 - [warn] Projects disabled : editorTheme.projects.enabled=false
11 Apr 20:26:58 - [info] Flows file : /data/flows.json
11 Apr 20:26:58 - [warn]

-----
Your flow credentials file is encrypted using a system-generated key.

If the system-generated key is lost for any reason, your credentials
file will not be recoverable, you will have to delete it and re-enter
your credentials.

You should set your own key using the 'credentialSecret' option in
your settings file. Node-RED will then re-encrypt your credentials
file using your chosen key the next time you deploy a change.

-----
11 Apr 20:26:58 - [info] Server now running at http://127.0.0.1:1880/
11 Apr 20:26:58 - [info] Starting flows
11 Apr 20:26:58 - [info] Started flows
^C
amusarra@amusarra-pi-4b:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES
103c8c51afcd nodered/node-red:1.3.1 "npm --no-update-not..." 24 seconds ago Up 22 seconds (health: starting) 0.0.0.0:1880->1880/tcp
node-red
03611992f567 emqpx/emqpx-edge:4.2.9 "/usr/bin/docker-ent..." 19 minutes ago Up 17 minutes
4369-4370/tcp, 5369/tcp, 6369/tcp, 0.0.0.0:1883->1883/tcp
p, 8081/tcp, 8084/tcp, 8883/tcp, 0.0.0.0:18083->18083/tcp, 11883/tcp mqtt-broker
amusarra@amusarra-pi-4b:~# Install DashboardNode
```

Screencast 6 - Install Node-RED 1.3.1 via Docker on Raspberry Pi 4

Dopo l'avvio del container node-red, dovremmo poter raggiungere l'applicazione puntando il browser all'indirizzo [https://\\${IP_GATEWAY_RPI}:1880](https://${IP_GATEWAY_RPI}:1880) ottenendo una vista simile a quella mostrata dalla figura a seguire.

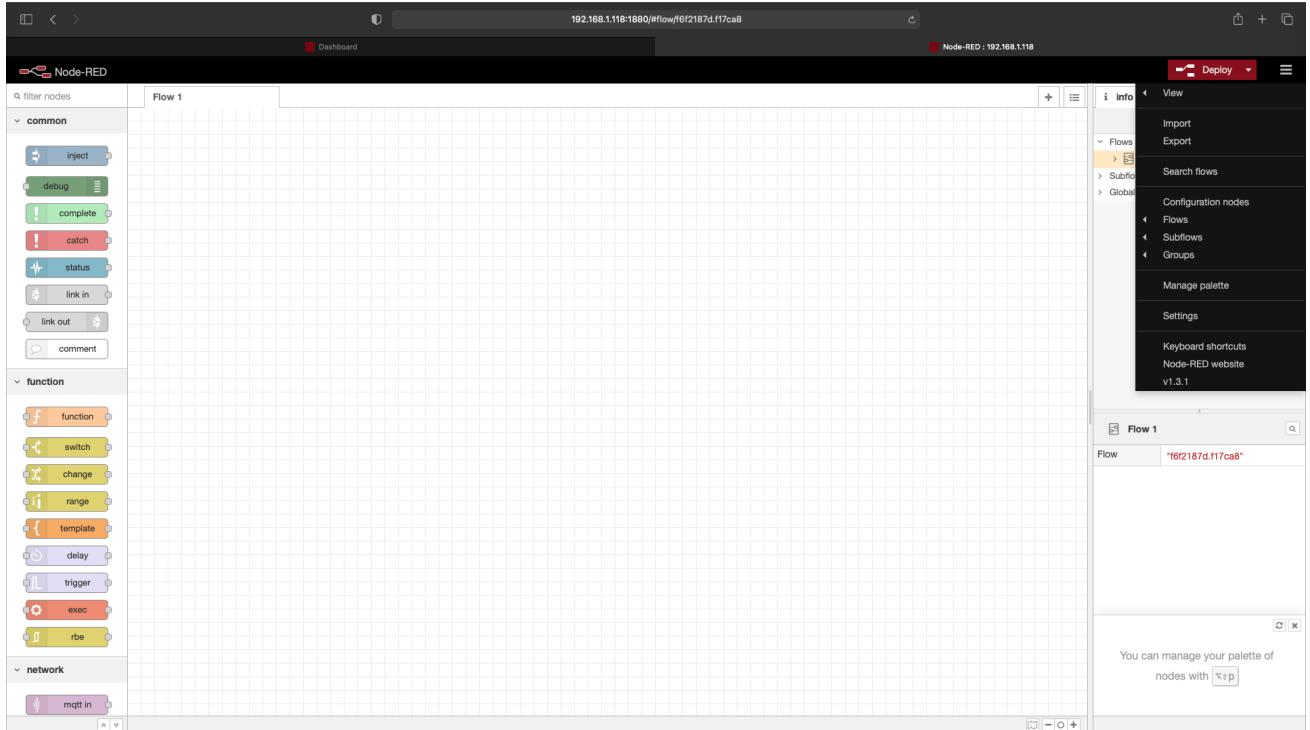


Figura 23 - Home Page di Node-RED subito dopo il primo avvio

A questo punto non resta che creare il flusso che ci consentirà di mostrare i dati ambientali di ogni device e di effettuare eventuali azioni sui relè. Non vi preoccupate, ho già bello e pronto il flusso [flow_esp32_mqtt_dashboard.json](#) da importare su Node-RED.

L'importazione del flusso è molto semplice, basta aprire il menù principale e poi cliccare sulla voce **Import**. Dalla maschera di dialogo abbiamo la possibilità di caricare il flusso tramite file o anche clipboard. Una volta scelto il metodo d'importazione e il flusso in formato JSON è visibile all'interno della text area, occorre cliccare sul pulsante **Import**. La figura a seguire mostra proprio l'importazione del nostro flusso.

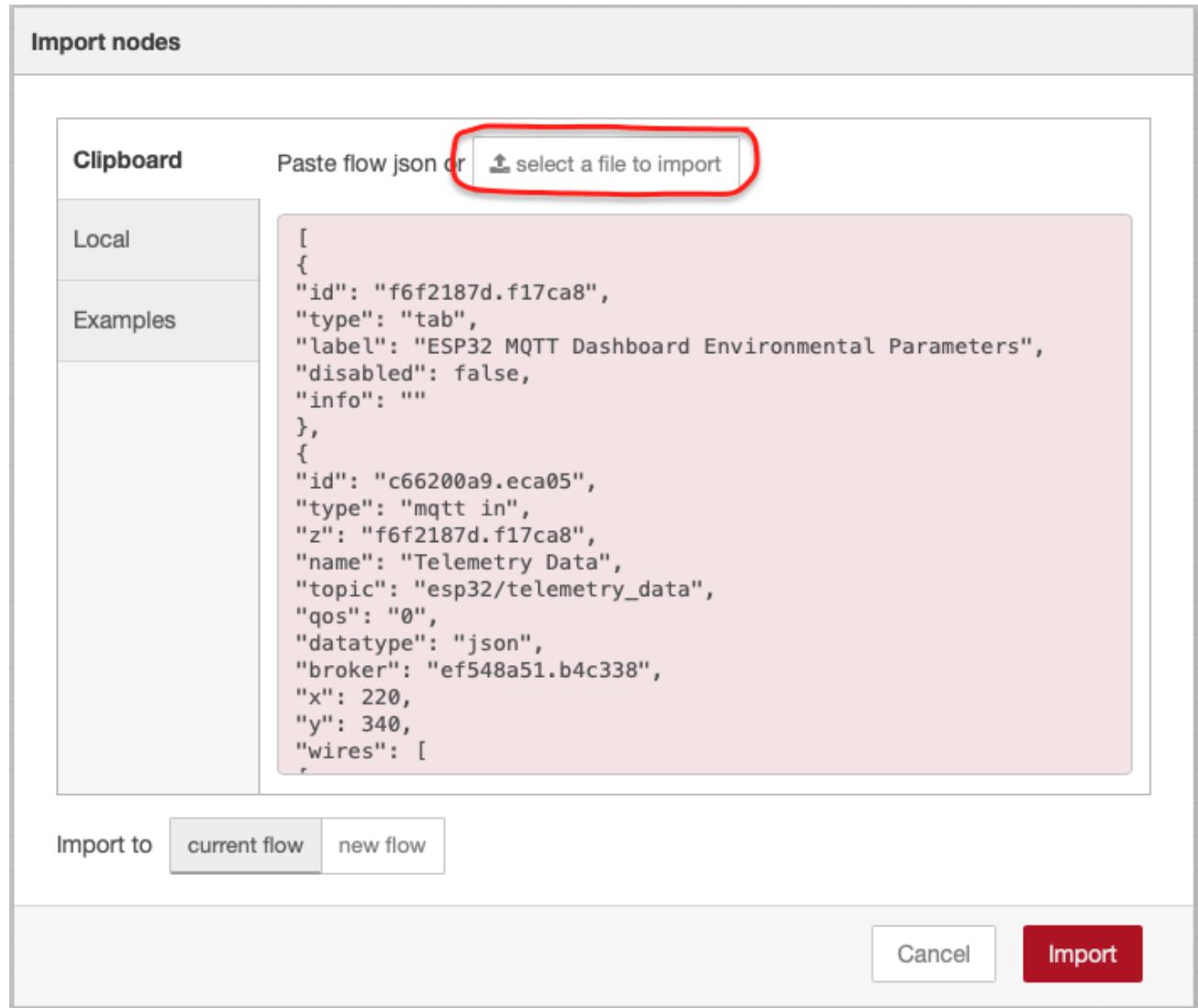


Figura 24 - Importazione flusso Node-RED della dashboard da file JSON o via clipboard

Dopo l'importazione dovreste vedere un nuovo sheet contenente il flusso Node-RED così come mostrato dalla figura a seguire. A prima vista il flusso potrebbe apparire complesso ma in realtà non lo è, anzi, credo che sia piuttosto chiaro da leggere anche per via dei nomi molto descrittivi assegnati a ogni nodo del flusso.

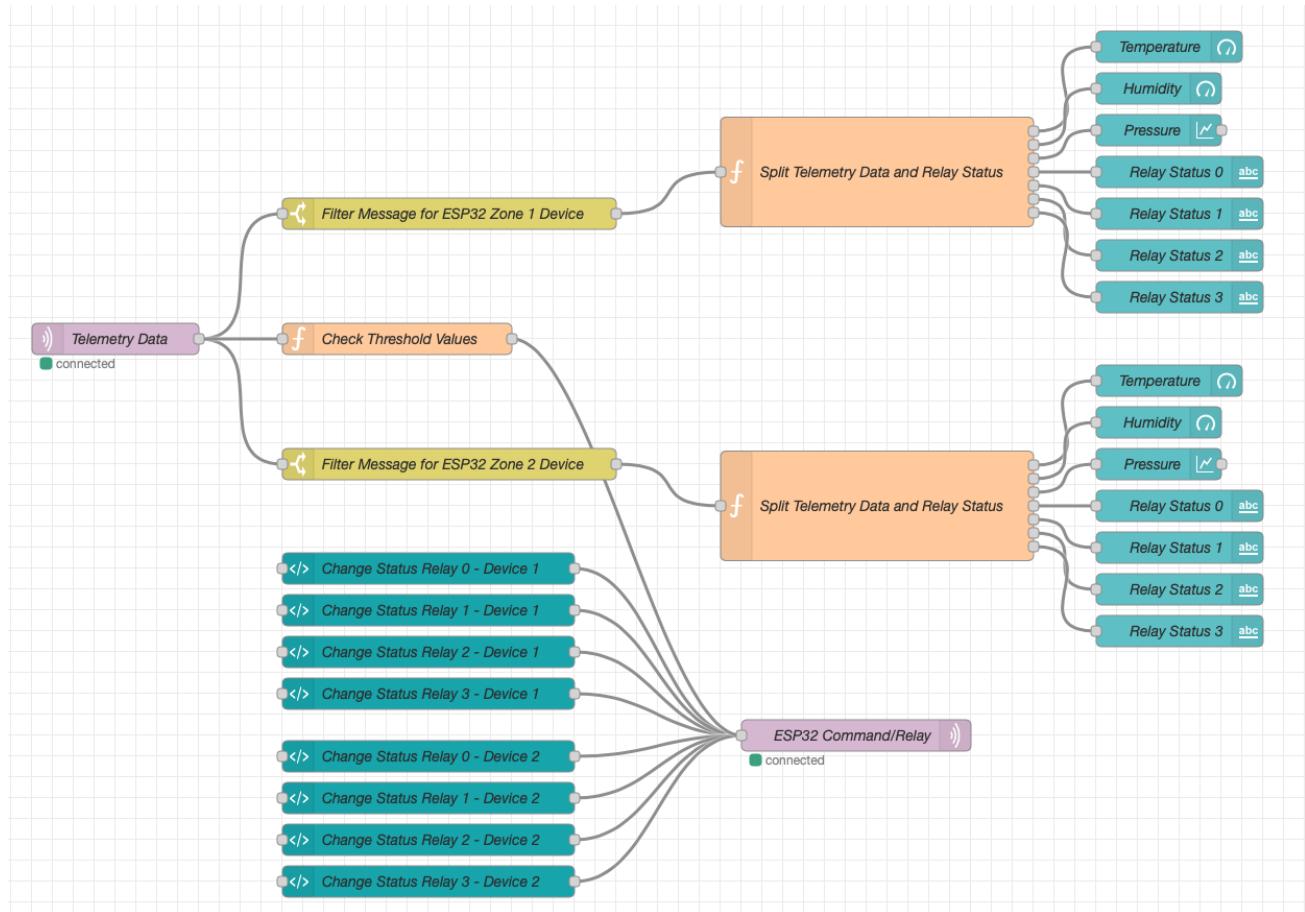


Figura 25 - Flusso Node-RED che implementa la dashboard MQTT della nostra soluzione IoT

Adesso vediamo il ruolo ricoperto da ogni singolo nodo del flusso mostrato nella figura precedente.

- **Telemetry Data:** è un nodo di tipo network specifico per un flusso MQTT in ingresso. In questo caso il nodo è stato configurato per essere sottoscritto al topic esp32/telemetry_data, riceverà quindi i dati ambientali provenienti dai device esp32-zone-1 e esp32-zone-2.
- **Filter Message for ESP32 Zone 1|2 Device:** sono due nodi di tipo **switch** il cui compito è quello di dividere il flusso dei messaggi che contengono i dati ambientali sulla base del device di origine. È possibile fare quest'operazione di filtro perchè la struttura del messaggio JSON dei dati ambientali, contiene l'attributo **deviceName** valorizzato con il nome logico del dispositivo di provenienza. In questo caso i dispositivi sono solo due: esp32-zone-1 e esp32-zone-2.
- **Split Telemetry Data and Relay Status:** sono due nodi di tipo **function** il cui compito è quello di applicare una funzione Javascript al messaggio d'ingresso per poi restituire ben 7 valori che rappresentano: temperatura, umidità, pressione, stato del relè 0, stato del relè 1, stato del relè 2 e stato del relè 3. È necessario estrapolare ogni singolo valore dal messaggio in formato JSON, perchè ognuno di essi dovrà essere rappresentato in formato grafico (gauge, line chart, text, etc.).
- **Check Threshold Values:** è un nodo di tipo **function** il cui compito è quello di verificare che i valori dei dati ambientali rispettino le soglie impostate e prendere le misure adeguate qualora non ci sia questo rispetto. L'azione consiste nell'invio di un messaggio di comando verso il topic esp32/command, questo comando scatenerà poi l'attivazione o la disattivazione del relè.
- **ESP32 Command/Relay:** è un nodo di tipo network specifico per un flusso MQTT in uscita. In questo caso il nodo è stato configurato per pubblicare messaggi sul topic esp32/command (con QoS pari a 1).
- **Dashboard Node:** in questa categoria rientrano tutti quei nodi che mostreranno le informazioni provenienti dal campo sulla dashboard in forma di grafico, testo o pulsanti di azione.

Le immagini a seguire mostrano le configurazioni dei principali nodi del flusso che abbiamo costruito. Una volta che il flusso è stato importato correttamente, possiamo procedere con l'operazione di deploy. Il deploy del flusso avviene cliccando sul pulsante deploy posizionato in alto a destra della GUI di Node-RED.

Edit mqtt in node > **Edit mqtt-broker node**

Delete
Cancel
Update

Properties

Name	EMQ X Edge		
Connection Security Messages			
Server	mqtt-broker	Port	1883
<input type="checkbox"/> Use TLS			
Protocol	MQTT V3.1.1		
Client ID	mqtt-dashboard		
Keep Alive	60		
Session	<input checked="" type="checkbox"/> Use clean session		

Figura 26 - Configurazione parametri di connessione al Message Broker EMQ X Edge

Possiamo specificare come nome del server MQTT direttamente il nome del container che abbiamo assegnato a EMQ X Edge in fase di creazione. Questo è possibile farlo in virtù del fatto che sia il Message Broker sia Node-RED sono entrambe parte della rete iot-edge.

Edit mqtt in node > **Edit mqtt-broker node**

Properties

Name: EMQ X Edge

Connection **Security** (selected) **Messages**

Username: mqtt-dashboard

Password:

Delete **Cancel** **Update**

Figura 27 - Configurazione username e password di accesso al Message Broker EMQ X Edge

Edit mqtt in node

Properties

Server: EMQ X Edge

Topic: esp32/telemetry_data

QoS: 0

Output: a parsed JSON object

Name: Telemetry Data

Delete **Cancel** **Done**

Figura 28 - Configurazione del nodo MQTT In configurato per la sottoscrizione al topic esp32/telemetry_data

Possiamo configurare l'output del nodo MQTT In per ottenere direttamente un oggetto JSON, sicuramente più agevole per lavorarci sui nodi successivi del flusso.

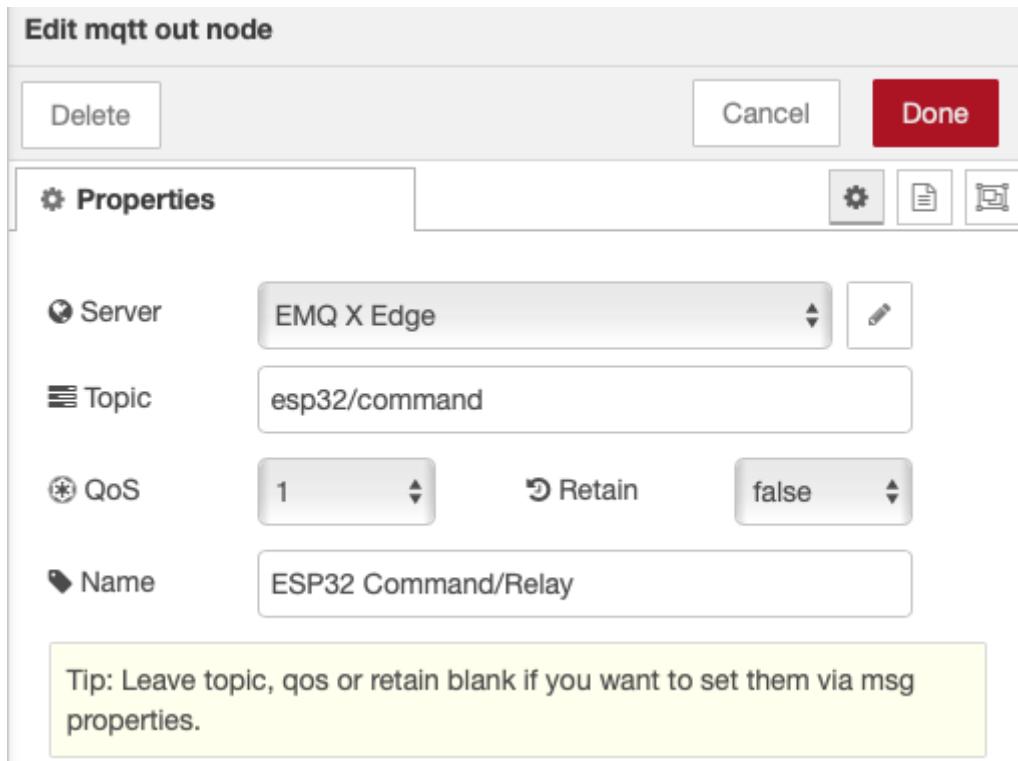


Figura 29 - Configurazione del nodo MQTT Out per la pubblicazione dei comandi sul topic esp32/command

Sul nodo MQTT Out è stato impostato il QoS a uno per il topic esp32/command senza però attivare il retain che per i nostri scopi non è necessario. La figura a seguire mostra la configurazione del nodo di tipo function Check Threshold Values la cui responsabilità è stata descritta in precedenza. I valori di threshold dei tre parametri ambientali possono essere cambiati sulla base delle proprie necessità come anche le azioni da perseguire. La funzione Javascript alla fine non fa altro che impostare il comando per attivare o disattivare il relè specificato.

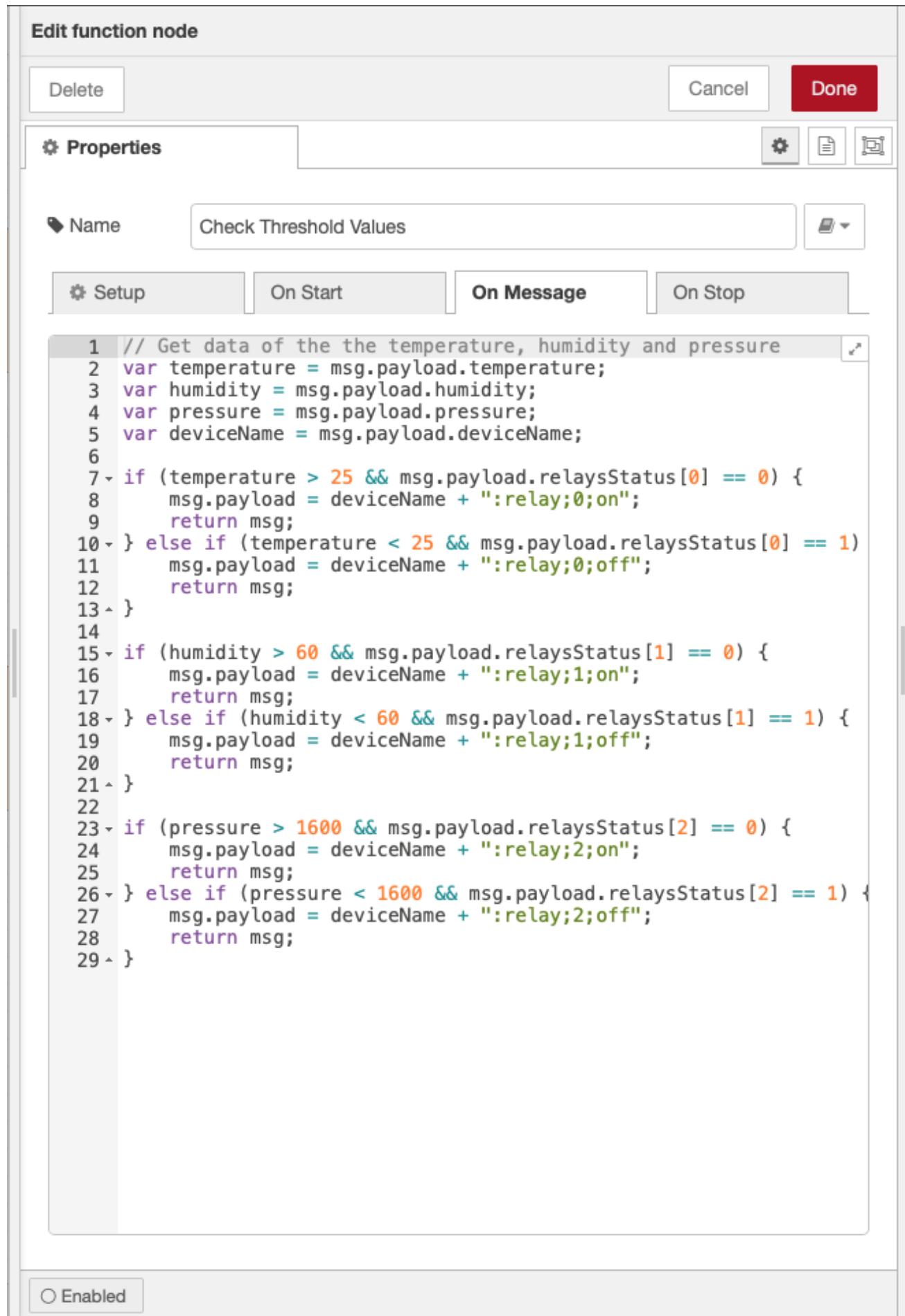


Figura 30 - Configurazione del nodo function per il check dei threshold

La figura a seguire mostra la configurazione del nodo di tipo function Split Telemetry Data and Relay Status la cui responsabilità è stata descritta in precedenza. La funzione Javascript che vedete, estraе i singoli valori dal messaggio JSON dei dati ambientali ricevuti in ingresso restituendo i valori distinti su output distinti.

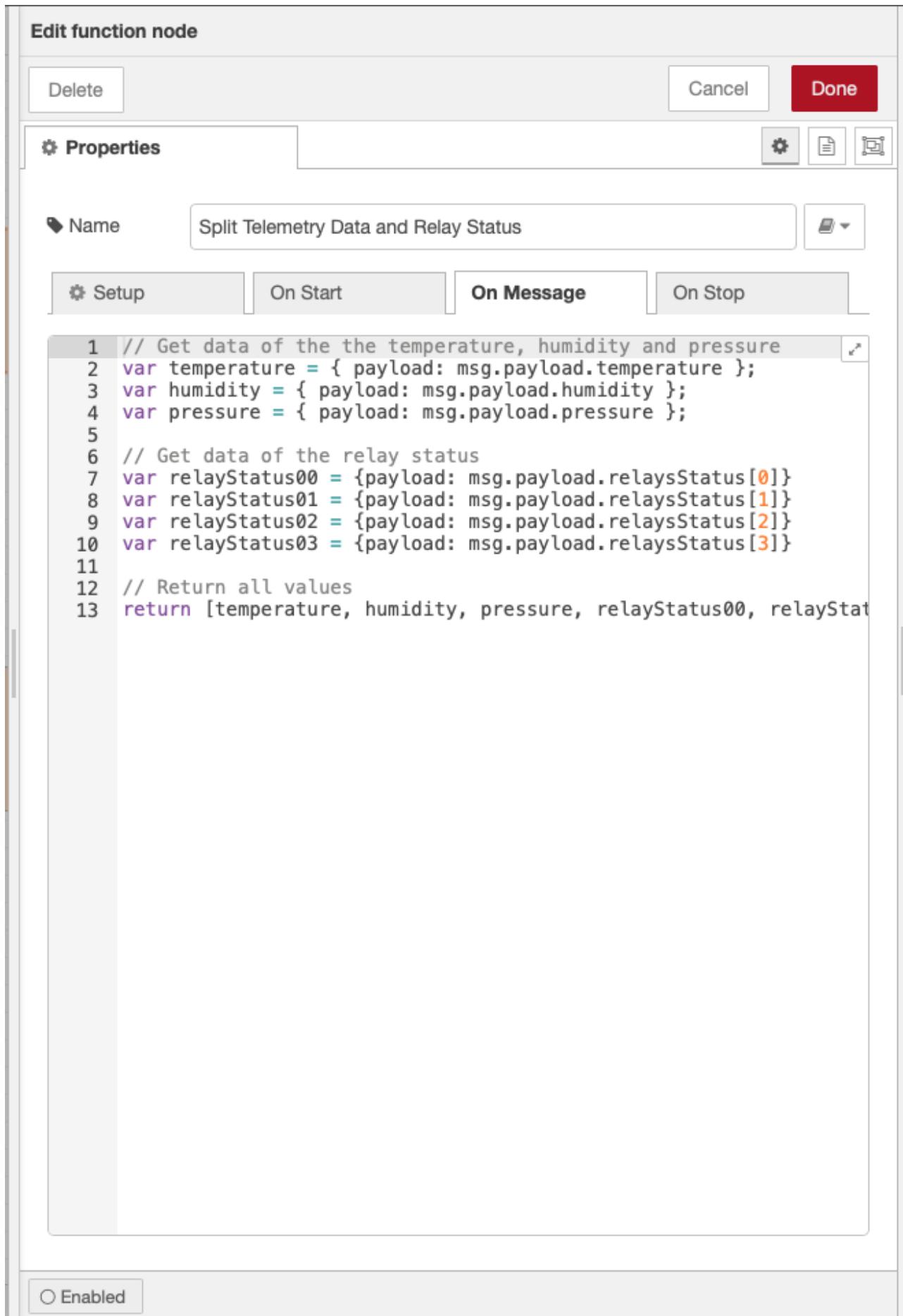
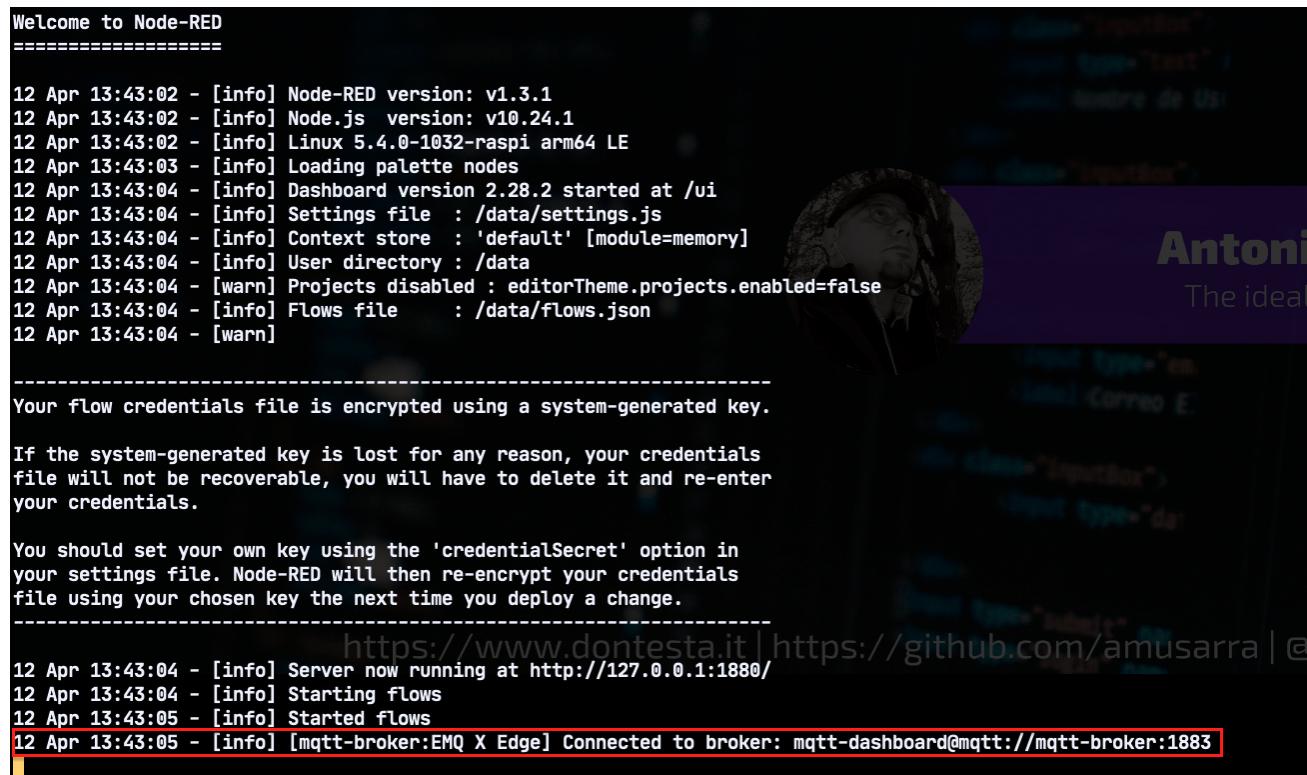


Figura 31 - Configurazione del nodo di tipo function che esegue lo split dei valori contenuti sul messaggio dei dati ambientali

Una volta che il deploy è stato eseguito senza errori, dovreste vedere sui nodi mqtt (in o out) del flusso, la dicitura connected, questo è un ottimo segno, vuol dire che la connessione verso il Message Broker è avvenuta con successo. La stessa informazione è comunque riportata sui log di Node-RED a cui è possibile accedere sempre attraverso Docker utilizzando il comando: `docker logs -f node-red`.



```
Welcome to Node-RED
=====
12 Apr 13:43:02 - [info] Node-RED version: v1.3.1
12 Apr 13:43:02 - [info] Node.js version: v10.24.1
12 Apr 13:43:02 - [info] Linux 5.4.0-1032-raspi arm64 LE
12 Apr 13:43:03 - [info] Loading palette nodes
12 Apr 13:43:04 - [info] Dashboard version 2.28.2 started at /ui
12 Apr 13:43:04 - [info] Settings file : /data/settings.js
12 Apr 13:43:04 - [info] Context store : 'default' [module=memory]
12 Apr 13:43:04 - [info] User directory : /data
12 Apr 13:43:04 - [warn] Projects disabled : editorTheme.projects.enabled=false
12 Apr 13:43:04 - [info] Flows file : /data/flows.json
12 Apr 13:43:04 - [warn]

-----
Your flow credentials file is encrypted using a system-generated key.

If the system-generated key is lost for any reason, your credentials
file will not be recoverable, you will have to delete it and re-enter
your credentials.

You should set your own key using the 'credentialSecret' option in
your settings file. Node-RED will then re-encrypt your credentials
file using your chosen key the next time you deploy a change.

https://www.dontesta.it | https://github.com/amusarra | @
12 Apr 13:43:04 - [info] Server now running at http://127.0.0.1:1880/
12 Apr 13:43:04 - [info] Starting flows
12 Apr 13:43:05 - [info] Started flows
12 Apr 13:43:05 - [info] [mqtt-broker:EMQ X Edge] Connected to broker: mqtt-dashboard@mqtt://mqtt-broker:1883
```

Figura 32 - Connessione dei nodi MQTT al Message Broker configurato per il flusso

Adesso ci vorrebbe proprio un rullo di tamburi! **Come accediamo alla Dashboard?** Basta puntare il proprio browser all'indirizzo `http://${IP_GATEWAY_RPI}:1880/ui` e dovreste vedere apparire la dashboard così come quella mostrata dalle figure a seguire.

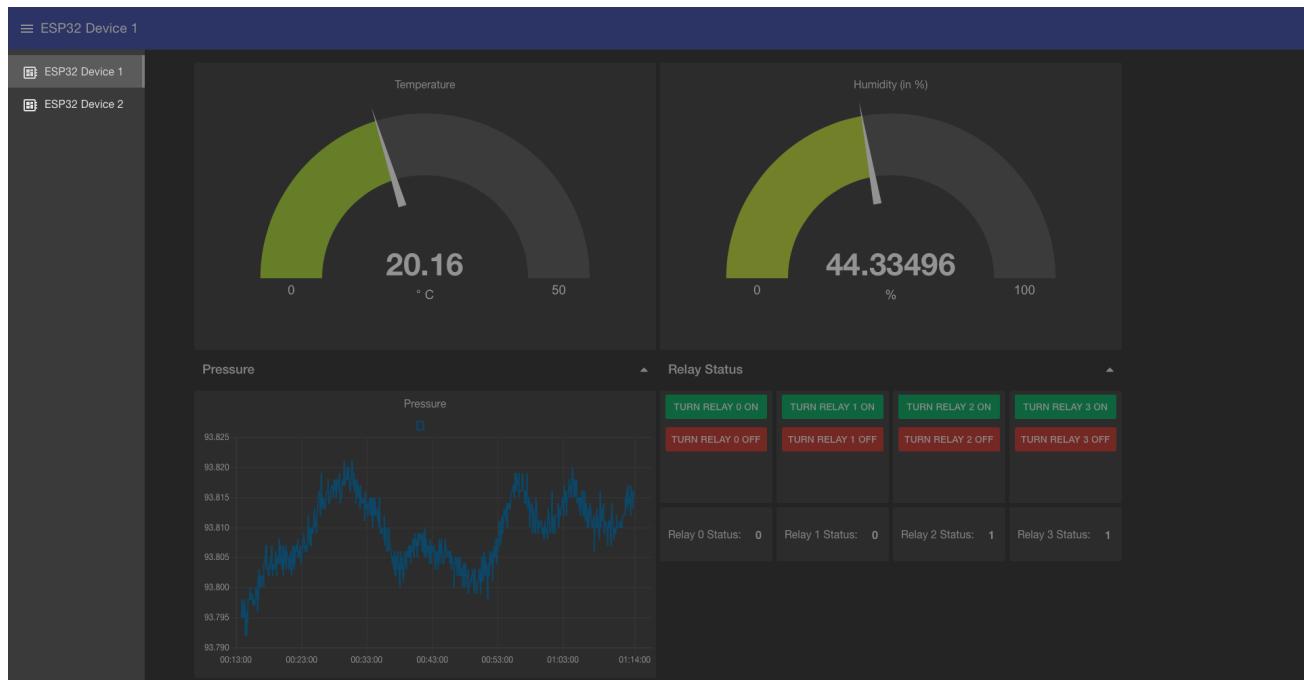


Figura 33 - MQTT Dashboard Node-RED con la possibilità di selezionare il device d'interesse

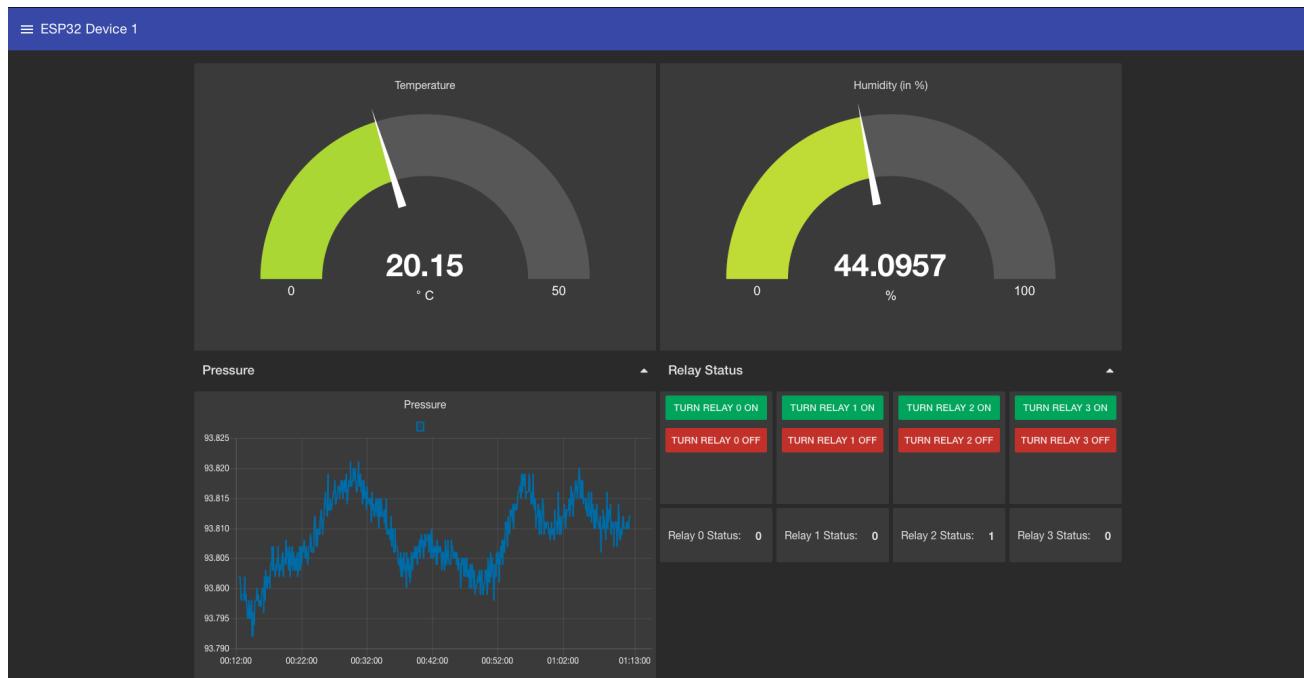


Figura 34 - MQTT Dashboard Node-RED per il ESP32 Device 1

La dashboard che abbiamo costruito, per ogni device (ESP32 Device 1 e ESP32 Device 2) mostra:

1. Un gauge che visualizza le temperatura in gradi centigradi (da 0 a 50);
2. Un gauge che visualizza la percentuale di umidità da (0 a 100);
3. Un grafico a linee che mostra l'andamento della pressione atmosferica nell'ultima ora;
4. La sezione Relay Status da cui è possibile vedere lo stato di ogni relè (se attivo o non attivo) e agire manualmente su di essi per attivare o disattivare.

Cosa ve ne pare di questa dashboard realizzata con Node-RED? Avendo a disposizione il codice del flusso Node-RED, potreste fare tutte le migliorie che ritenete utili per i vostri scopi.

11. Conclusioni

Ringrazio tutti voi per essere arrivati "incolumi" alla fine di questo lungo eBook sperando di non essere stato noioso ma di essere riuscito nell'intento di rendere interessanti gli argomenti trattati oltre che a spingere la vostra curiosità in avanti.

Avrei voluto approfondire altri argomenti come per esempio [EMQ X Kuiper](#), da introdurre in questa architettura per affiancarlo a EMQ X Edge con lo scopo di fare real-time streaming data processing, utile nel caso in cui dobbiamo eseguire delle azioni a fronte dei dati ambientali ricevuti. Il controllo dei valori di Threshold è una delle operazioni che andrebbe spostata per esempio su EMQ X Kuiper.

Avrei voluto ancora approfondire altri argomenti che riguardano l'EMQ X Edge, come per esempio l'uso del [Rule Engine](#).

Nell'attesa di pubblicare altri eBook o articoli di approfondimento, vi chiedo di scrivere le vostre impressioni, esperienze o altro di cui vi faccia piacere sia trattato, utilizzando i [commenti all'articolo](#) oppure condividendo questo eBook attraverso i vostri canali social.

12. Risorse

Come di consueto lascio una serie di risorse che ritengo utili ai fini dell'approfondimento degli argomenti trattati nel corso di questo articolo.

- Attuatori per maker - <https://amzn.to/3slHmE9>
- Raspberry Pi. La guida completa - <https://amzn.to/2RpYZWh>
- Electronics Projects With the ESP8266 and ESP32 - <https://amzn.to/3gixAQP>
- Docker: Sviluppare e rilasciare software tramite container - <https://amzn.to/3tiyO1W> di [Serena Sensini](#)
- Valutiamo se continuare a usare Docker o passare a Podman - <https://theredcode.it/devops/cos-e-podman/>
- Pillole di Docker - <https://www.youtube.com/watch?v=wAyUdtQF05w> di [Mauro Cicolella](#)
- Raspberry PI GPIO – Tutti i segreti del pinout - <https://www.moreware.org/wp/blog/2021/04/09/raspberry-pi-gpio-tutti-i-segreti-del-pinout/>
- Renzo Mischianti - A blog of digital electronics and programming - <https://www.mischianti.org/>
- Practical Node-RED Programming - <https://www.packtpub.com/product/practical-node-red-programming/9781800201590>

Un trittico vincente: ESP32, Raspberry Pi e EMQ X Edge

Scopri le potenzialità dell'Internet delle Cose (IoT) combinando l'ESP32 e il Raspberry Pi con EMQX Edge. Questo articolo guida passo dopo passo nella configurazione e nell'implementazione del protocollo MQTT (Message Queuing Telemetry Transport) per la comunicazione tra dispositivi.

Contenuti Principali:

- Introduzione all'IoT e MQTT:** Una panoramica su come MQTT facilita la comunicazione efficiente e in tempo reale tra dispositivi IoT.
- Configurazione di ESP32 e Raspberry Pi:** Istruzioni dettagliate per configurare questi dispositivi per funzionare con EMQX Edge, un potente broker MQTT.
- Implementazione del Publish/Subscribe:** Esempi pratici di come impostare e gestire la pubblicazione e la sottoscrizione di messaggi MQTT, consentendo una comunicazione bidirezionale tra ESP32 e Raspberry Pi.
- Applicazioni e Vantaggi:** Esplorazione delle applicazioni pratiche di questa configurazione nell'ambito dell'IoT, inclusi i vantaggi in termini di efficienza, scalabilità e facilità d'uso.

Perché Leggerlo:

Questo eBook è essenziale per chiunque voglia approfondire le proprie conoscenze sull'IoT e desideri implementare soluzioni MQTT efficienti utilizzando hardware popolare come ESP32 e Raspberry Pi.

Ho iniziato il mio viaggio nel mondo dell'informatica da un Olivetti M24 dotato di un processore Intel 8086 acquistato da mio padre esclusivamente per il suo lavoro. Non ho mai posseduto console di nessun genere (Commodore, Amiga, etc...) e inizialmente quell'enorme scatola mi terrorizzava, terrore durato poco; giorno dopo giorno prendevo rapidamente il controllo fino a quando....



Ho sempre creduto che la condivisione della conoscenza sia un ottimo mezzo per la crescita personale e questo è stato uno dei principali motivi che mi ha spinto a creare il mio blog personale www.dontesta.it.

Dicono di me che sono bravo nell'analizzare e risolvere rapidamente i problemi complessi. La mia attività odierna è quella di consulente in progetti enterprise che utilizzano tecnologie Web Oriented come J2EE, Web Services, ESB, TIBCO, Microservices, Cloud Native Application, Kubernetes.



www.linkedin.com/in/amusarra



github.com/amusarra