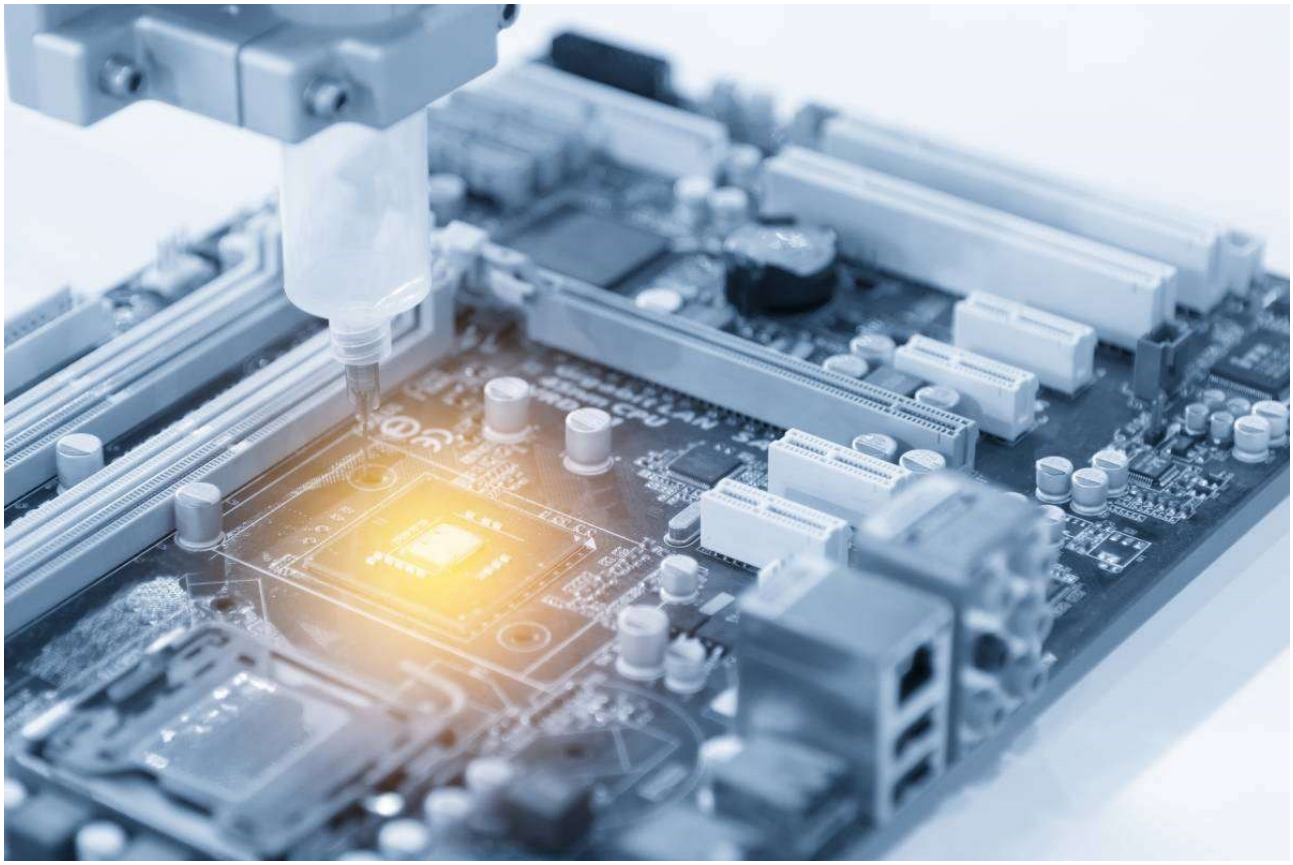
1

PROGETTO SISTEMI OPERATIVI DEDICATI



GIADA GATTI

1108648

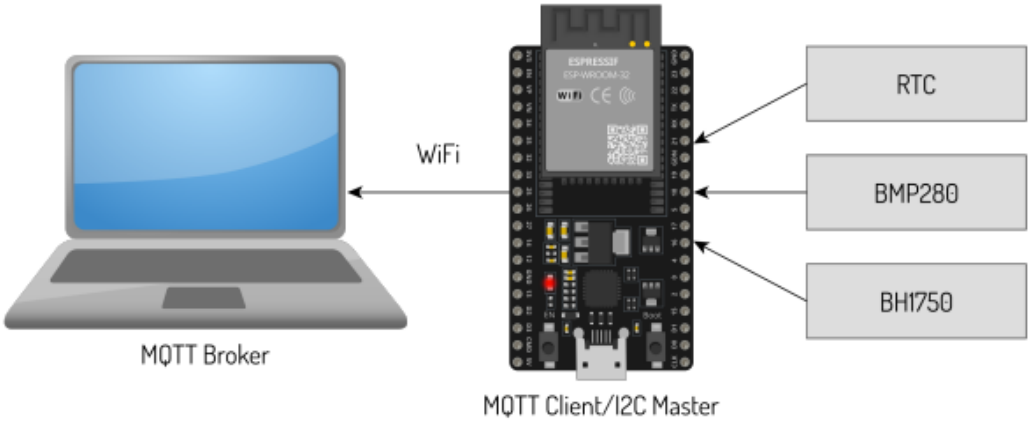
LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Link al repository GitHub

Sommario

INTRODUZIONE

Il progetto mira a sfruttare le potenzialità della scheda microcontroller **ESP32**, integrando l'acquisizione dei dati da sensori, come **BMP280** e **BH1750**, la gestione multithreading attraverso *FreeRTOS* e la comunicazione tramite il protocollo *MQTT*.
Il risultato finale è un sistema IoT che consente la visualizzazione dei dati attraverso un'interfaccia Web, fornendo una solida base per applicazioni piu' complesse.



SPECIFICHE DEL SISTEMA

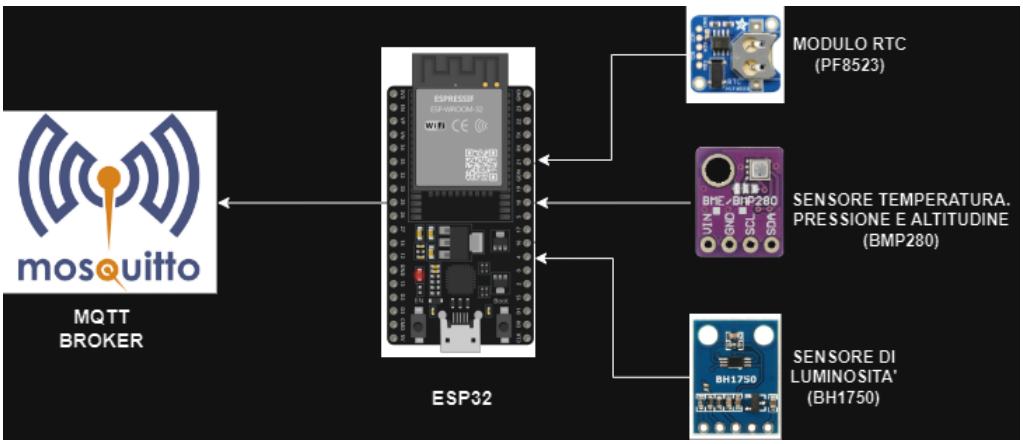
L'architettura del sistema è stata progettata per garantire flessibilità e scalabilità.
La **ESP32** funge da nodo centrale, orchestrando la lettura dei dati dai sensori, la gestione del tempo tramite un modulo **RTC** e la comunicazione *MQTT*.
Un broker *MQTT*, implementato su una macchina virtuale Linux, svolge il ruolo di intermediario nella distribuzione dei dati.
L'interfaccia utente è fornita attraverso un server Web che permette la visualizzazione immediata dei dati acquisiti.

COMPONENTI PRINCIPALI

I componenti principali di questo sistema sono i seguenti:

- **ESP32**: Questa scheda di sviluppo è il cuore del sistema, gestendo la connessione Wi-Fi, le operazioni multithreading tramite FreeRTOS e la comunicazione MQTT.
- **Sensori(BMP280 e BH1750)**: Essi forniscono dati dettagliati sulla temperatura, pressione e luminosità ambientale.
- **Modulo RTC(PCF8523)**: fornisce un timestamp per ogni misura acquisita.
- **FreeRTOS**: Sistema operativo in tempo reale che consente l'esecuzione di concorrente di task indipendenti.

- **Mosquitto(Broker MQTT)**: Broker che facilita la comunicazione asincrona tra la ESP32 e altri dispositivi.
- **Browser Web**: Interfaccia intuitiva per l'utente, permettendo la visualizzazione dei dati in tempo reale.

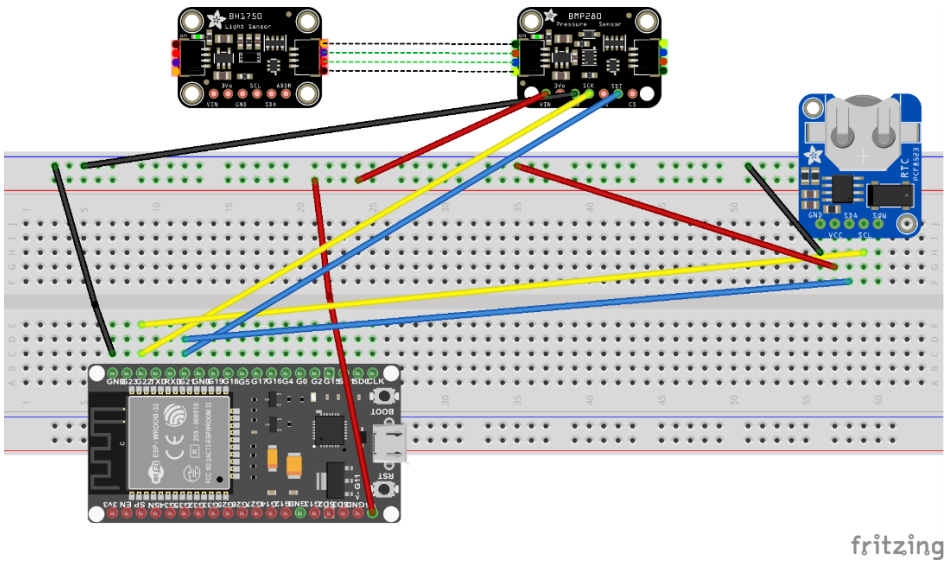


IMPLEMENTAZIONE DEL CODICE

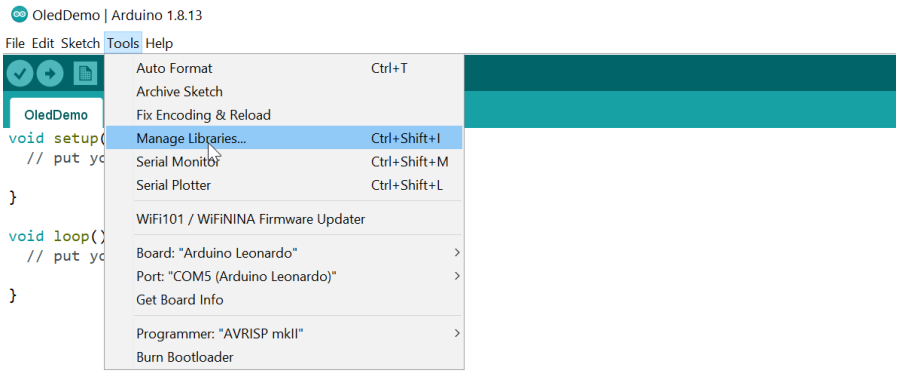
Qui si andranno a descrivere le varie fasi e soluzioni adottate per la realizzazione del sistema.
Il codice per la **ESP32** è suddiviso in task distinti per leggere i dati dai sensori , gestire la comunicazione *MQTT* e ottenere il timestamp dal modulo **RTC**.
La connessione Wi-Fi è configurata per consentire la comunicazione con il broker *MQTT*.

1.1 Acquisizione dati dai sensori

Come prima cosa si vanno a collegare i sensori all'**ESP32** tramite una breadboard.



Dopo avere fatto questi collegamenti, attraverso l'ide di sviluppo *Arduinolde*, si vanno ad installare ed importare sul codice le varie librerie che ci serviranno poi per poter leggere i dati dagli stessi.
Per poter installare queste librerie, basterà aprire *Arduinolde* → *Tools* → **Manage** *Libraries* ed installare la libreria corretta, come mostrato in figura.



Le librerie utilizzate nel nostro caso sono:

- ***Wire***.h: permette una corretta connessione con i sensori e i moduli RTC.
- ***Adafruit_Sensor***.h: permette un corretto funzionamento dei sensori dell'azienda Adafruit.
- ***Adafruit_BMP280.h***: libreria per il sensore BMP280 dell'azienda Adafruit.
- ***RTCLib***.h: libreria per il modulo RTC dell'azienda Adafruit.
- ***WiFi***.h: permette di connettersi ad una rete WiFi.
- ***PubSubClient***.h: libreria per utilizzare il protocollo MQTT.
- ***BH1750***.h: libreria per il sensore BH1750 dell'azienda Adafruit.

Dopodichè, si va creare un task, gestito con *FreeRTOS*, che ha il compito di leggere i dati dai vari dispositivi (temperatura, pressione, luminosità e timestamp) e memorizzarli su delle variabili globali. Tali variabili saranno poi utilizzate in un differente task per l'invio delle misure tramite protocollo *MQTT*.

```
void readSensorData(void *pvParameters){
  TickType_t lastWakeTime = xTaskGetTickCount();
  while(1){

    DateTime pcTime = DateTime(F(__DATE__), F(__TIME__));
    rtc.adjust(pcTime);

    float currentLuminosity = lightMeter.readLightLevel();
    float currentTemperature = bmp.readTemperature();
    float currentPressure = bmp.readPressure() / 100.0F;


    DateTime now = rtc.now();
    String currentTimestamp = String(now.year()) + "-" +
                              String(now.month()) + "-" +
                              String(now.day()) + "" +
                              String(now.hour()) + ":" +
                              String(now.minute()) + ":" +
                              String(now.second());
//richiesta di accesso al semaforo
xSemaphoreTake(dataMutex, portMAX_DELAY);

    luminosity = currentLuminosity;
    temperature = currentTemperature;
    pressure = currentPressure;
    timestamp = currentTimestamp;

//rilascia il semaforo acquisito precedentemente
xSemaphoreGive(dataMutex);

Serial.println("Sensor data: ");
Serial.print("Luminosity: ");
Serial.println(luminosity);
Serial.print("Temperature: ");
Serial.println(temperature);
Serial.print("Pressure: ");
Serial.println(pressure);
Serial.print("Timestamp: ");
Serial.println(timestamp);

vTaskDelayUntil(&lastWakeTime, pdMS_TO_TICKS(5000));
}
}
```

Inoltre, per quanto riguarda la sincronizzazione del modulo RTC si dovrà installare un software di sincronizzazione del tempo, NTP tramite riga di comando, per configurare il PC come sorgente di tempo :

```
sudo apt-get install ntp
```

Successivamente, da codice, si ottengono la data e l'ora attuali dal PC

```
    DateTime pcTime = DateTime(F(__DATE__), F(__TIME__));
    rtc.adjust(pcTime);
```

Come possiamo vedere dal codice, si è scelto di utilizzare un semaforo per garantire la corretta gestione delle risorse condivise tra i task in un ambiente concorrente. In particolare, essi implementano la mutua esclusione, garantendo che solo un task alla volta abbia accesso a determinate risorse o sezioni critiche del codice.
Nel nostro caso, viene utilizzato per proteggere l'accesso e la manipolazione delle variabili globali (*luminosity, temperature, pressure, timestamp).*

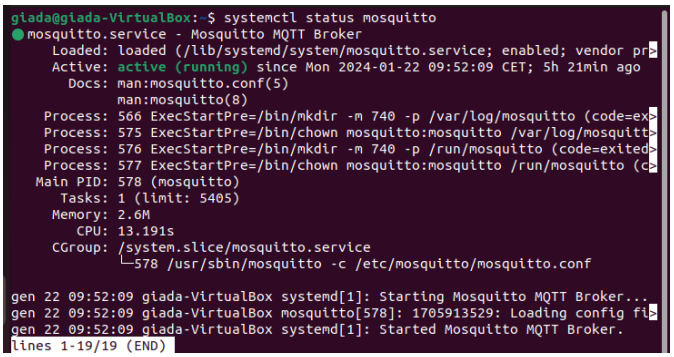
1.2 Comunicazione MQTT

Per la gestione della comunicazione MQTT sarà necessario installare e configurare un broker MQTT.

Nel progetto è stato scelto di installare, all'interno di una macchina virtuale con OS Ubuntu 22.04, il software *Mosquitto*. Per raggiungere suddetto obiettivo da terminale, sarà necessario eseguire la seguente riga di comando:

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

Per verificare che l'installazione è stata completata correttamente, si può, sempre da riga di comando, tramite terminale, digitare il comando: *systemctl status mosquitto* e assicurarsi che il servizio sia attivo, come in figura.



Nel momento in cui, si dovrà avviare la connessione *MQTT* basterà far partire *Mosquitto* tramite il comando:

```
sudo systemctl start mosquitto
```

Mentre, per abilitare il servizio in modo che si avvii all'avvio del sistema, basterà eseguire il seguente comando:

```
sudo systemctl enable mosquitto
```

Per quanto concerne la parte sull'ESP32, la comunicazione dei dati mediante protocollo MQTT è stata affidata ad un task specifico.

```
76  void sendMQTTData(void *pvParameters){
77    TickType_t lastWakeTime = xTaskGetTickCount();
78    while(1){
79      if(!client.connected()){
80        if(client.connect("ESP32Client")){
81        Serial.println("Connesso al broker MQTT!");}
82      else{
83        Serial.println("Connessione al broker MQTT fallita");
84    vTaskDelayUntil(&lastWakeTime, pdMS_TO_TICKS(5000));
85    continue;
86      }
87    }
88
89    xSemaphoreTake(dataMutex, portMAX_DELAY);
90
91    client.publish("luminosity", String(luminosity).c_str());
92    client.publish("temperature", String(temperature).c_str());
93    client.publish("pressure", String(pressure).c_str());
94    client.publish("timestamp", timestamp.c_str());
95
96    xSemaphoreGive(dataMutex);
97
98    Serial.println("MQTT Data Sent");
99
100   vTaskDelayUntil(&lastWakeTime, pdMS_TO_TICKS(1000));
101   }
102 }
```

Nella prima parte del task andiamo a controllare che la connessione con il broker sia attiva. In caso il controllo dia esito negativo, si tenta una riconnessione. Se questa non va a buon fine, si attenderanno 5 secondi e si proverà nuovamente la riconnessione.

```
76  void sendMQTTData(void *pvParameters){
77    TickType_t lastWakeTime = xTaskGetTickCount();
78    while(1){
79      if(!client.connected()){
80        if(client.connect("ESP32Client")){
81        Serial.println("Connesso al broker MQTT!");}
82      else{
83        Serial.println("Connessione al broker MQTT fallita");
84    vTaskDelayUntil(&lastWakeTime, pdMS_TO_TICKS(5000));
85    continue;
86      }
87    }
```

Se la connessione con il broker è presente, si procede con l'invio dei dati al broker MQTT mediante la funzione *client.publish()*.

Come si può notare, l'invio è stato inserito all'interno di una sezione critica. Questa scelta è dettata dalla necessità di proteggere le variabili globali da un'eventuale scrittura parallela, che renderebbe i dati non consistenti, dovuta al task di gestione dei sensori

```
89    xSemaphoreTake(dataMutex, portMAX_DELAY);
90
91    client.publish("luminosity", String(luminosity).c_str());
92    client.publish("temperature", String(temperature).c_str());
93    client.publish("pressure", String(pressure).c_str());
94    client.publish("timestamp", timestamp.c_str());
95
96    xSemaphoreGive(dataMutex);
97
98    Serial.println("MQTT Data Sent");
99
100   vTaskDelayUntil(&lastWakeTime, pdMS_TO_TICKS(1000));
101   }
102 }
```

Questo task gestisce, quindi, in modo efficiente l'invio dei dati *MQTT*, gestendo la connessione, la pubblicazione dei dati e garantendo la protezione dell'accesso concorrente alle variabili globali.

1.3 Funzione Setup

```
104  void setup() {
105
106  Serial.begin(115200);
107  Wire.begin();
108
109   if(!bmp.begin()){
110     Serial.println("Errore inizializzazione BMP280");
111   while(1);
112   }
113
114   if(!lightMeter.begin()){
115     Serial.println("Errore inizializzazione BH1750");
116   while(1);
117   }
118
119   if(!rtc.begin()){
120     Serial.println("Impossibile trovare il modulo RTC");
121   while(1);
122   }
123
124 //crea un semaforo di mutua esclusione
125 dataMutex = xSemaphoreCreateMutex();
126
127   //Connessione alla WiFi
128 WiFi.begin(ssid, password );
129   while(WiFi.status() != WL_CONNECTED) {
130     delay(250);
131     Serial.println("Connessione WiFi in corso...");
132   }
133   Serial.println("WiFi Connesso");
134   Serial.println(WiFi.localIP());
```

```
136   //Configurazione del client MQTT
137   client.setServer(mqtt_server, mqtt_port);
138
139   xTaskCreatePinnedToCore(
140     readSensorData,
141     "TaskSensorData",
142     10000,
143     NULL,
144     1,
145     &TaskSensorData,
146     0); //core 0
147
148
149
150   xTaskCreatePinnedToCore(
151     sendMQTTData,
152     "TaskMQTT",
153     10000,
154     NULL,
155     1,
156     &TaskMQTT,
157     1); }
```

In questa funzione viene inizializzata la comunicazione seriale e la libreria *Wire* per la comunicazione **I2C**, che è utilizzata per comunicare con sensori e altri dispositivi.
Si va poi a verificare se i sensori sono inizializzati correttamente. In caso l'inizializzazione di uno di questi non vada a buon fine, il codice si bloccherà all'interno di un ciclo while infinito.

```
104  void setup() {
105
106  Serial.begin(115200);
107  Wire.begin();
108
109   if(!bmp.begin()){
110     Serial.println("Errore inizializzazione BMP280");
111   while(1);
112   }
113
114   if(!lightMeter.begin()){
115     Serial.println("Errore inizializzazione BH1750");
116   while(1);
117   }
118
119   if(!rtc.begin()){
120     Serial.println("Impossibile trovare il modulo RTC");
121   while(1);
122   }
```

Per quanto riguarda la connessione del dispositivo alla rete WiFi, si vanno a specificare le credenziali e si attende la connessione.

```
127    //Connessione alla WiFi
128  WiFi.begin(ssid, password );
129    while(WiFi.status() != WL_CONNECTED) {
130      delay(250);
131      Serial.println("Connessione WiFi in corso...");
132    }
133    Serial.println("WiFi Connesso");
134    Serial.println(WiFi.localIP());
```

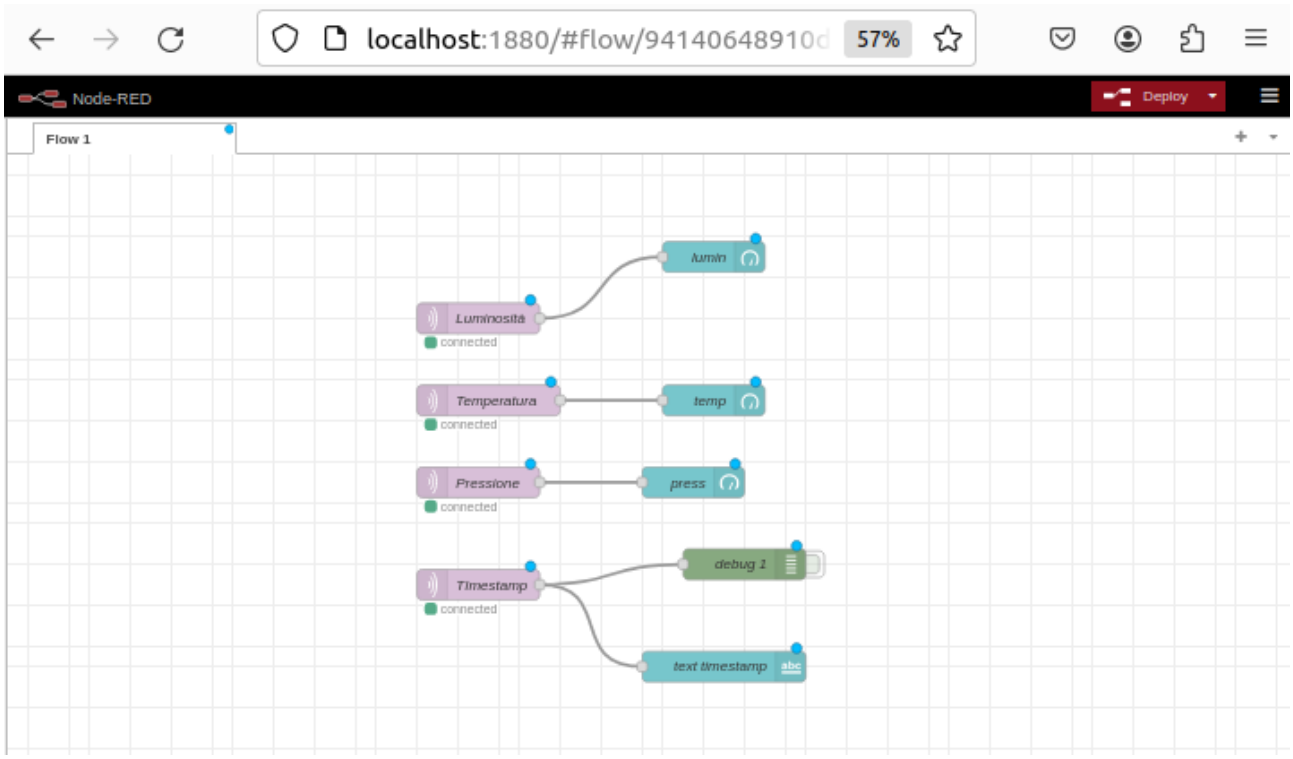
Questa funzione ha quindi il compito di inizializzare i dispositivi che saranno poi utilizzati dall'**ESP32**.

INTERFACCIA WEB

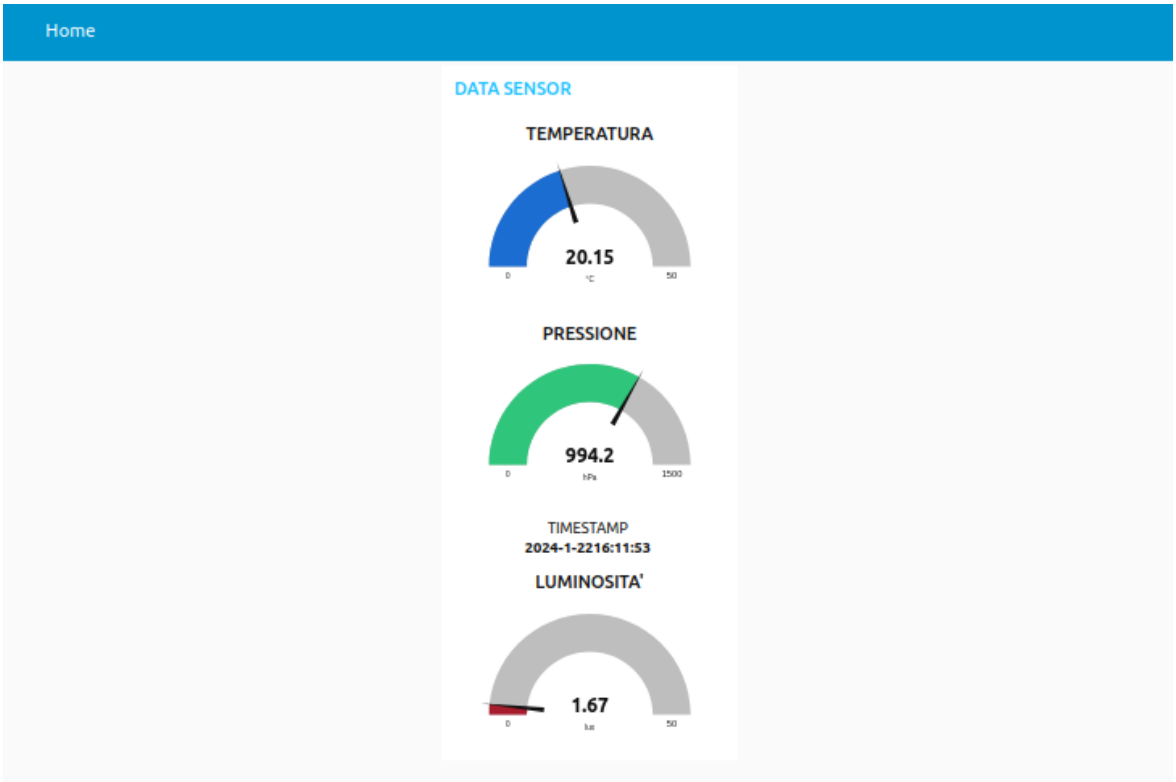
Questi dati, dopo essere stati inviati, tramite protocollo *MQTT*, al broker *MQTT*, saranno accessibili e visualizzabili attraverso un'interfaccia web creata grazie all'ausilio di ***Node-Red***, installato tramite terminale con i seguenti comandi:

```
npm install -g --unsafe-perm node-red
```

Per avviarlo manualmente, basterà scrivere, da terminale: *node-red*.
Sarà poi raggiungibile attraverso l'indirizzo http://localhost:1880/.
Attraverso la configurazione dei vari nodi(temperatura, pressione, luminosità e timestamp) con i relativi topic e indirizzamento, il risultato sarà un'interfaccia con i dati dei sensori.



Il risultato finale sarà del tipo:



CONCLUSIONI E PROBLEMI

Il progetto offre varie applicazioni nell'ambito dell'IoT, integrando diversi componenti in una sistema complesso.

La collaborazione tra i sensori, FreeRTOS e MQTT fornisce una base robusta per applicazioni future.

Per quanto riguarda le problematiche riscontrate inizialmente, sono state principalmente sulla connessione dei vari sensori con l'ESP32 perché, come prima idea si era pensato di fare i collegamenti solo tramite cavi jumper ma, andando avanti, diventava sempre piu impraticabile cosi da decidere di adottare la soluzione tramite una breadboard, che agevola sicuramente il lavoro.

Un altro problema riscontrato è stato quello riguardo l'installazione di Node-RED per creare l'interfaccia web finale, poiché, si doveva andare ad installare una specifica versione di Node-RED supportata.