

Applicazione di integrazioni IoT ad un sistema di monitoraggio della qualità dell'aria

Andrea Cantarutti (141808)
Lorenzo Bellina (142544)

20 ottobre 2021

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 4 |
| 2 | Il sensore VINDRIKTNING di Ikea | 5 |
| 3 | Obiettivi e Architettura del sistema | 5 |
| 3.1 | Caratterizzazione dei requisiti | 5 |
| 3.2 | Definizione dei principali servizi | 5 |
| 4 | Modifiche e personalizzazioni apportate a VINDRIKTNING | 6 |
| 4.1 | Obiettivi | 6 |
| 4.2 | Personalizzazione dell'hardware | 6 |
| 4.3 | Costo di realizzazione | 8 |
| 4.4 | Implementazione di un firmware ad-hoc | 8 |
| 4.4.1 | Decodifica del payload | 8 |
| 4.4.2 | Salvataggio e recupero dei parametri di configurazione | 9 |
| 4.4.3 | Aggiornamento del firmware da remoto | 9 |
| 4.4.4 | Configurazione di parametri personalizzati | 10 |
| 4.4.5 | Comunicazione con il Broker MQTT | 11 |
| 5 | Definizione del Broker MQTT | 12 |
| 5.1 | Containerizzazione di Eclipse Mosquitto | 12 |
| 6 | Ricezione dei dati | 14 |
| 6.1 | Engine | 14 |
| 6.2 | Containerizzazione | 14 |
| 6.3 | Comunicazione con VINDRIKTNING | 15 |
| 6.4 | Trasmissione dei dati ricevuti | 15 |
| 7 | Definizione di un database per lo storage dei dati | 16 |
| 7.1 | Scelta del DBMS | 16 |
| 7.2 | Containerizzazione | 16 |
| 8 | AirPI | 17 |
| 8.1 | Autenticazione | 17 |
| 8.2 | Ricezione e serializzazione dei dati | 17 |
| 8.3 | Notifica degli utenti | 17 |
| 8.3.1 | Bot Telegram | 17 |
| 8.3.2 | Implementazione | 17 |
| 8.3.3 | Comandi implementati | 18 |
| 8.3.4 | Invio di notifiche | 18 |
| 8.3.5 | Esempio di conversazione | 18 |
| 8.4 | Gestione delle utenze | 19 |
| 8.4.1 | Integrazione di un database relazionale | 19 |
| 8.4.2 | Gestione degli utenti Telegram | 20 |
| 8.4.3 | Gestione degli utenti di Monitoring Tool | 20 |
| 8.4.4 | Struttura della directory | 20 |
| 8.5 | Monitoring Tool | 21 |
| 8.5.1 | Barra di Navigazione | 21 |
| 8.5.2 | Homepage | 21 |
| 8.5.3 | Gestione degli utenti Telegram | 22 |
| 8.5.4 | Gestione degli utenti di Monitoring Tool | 22 |
| 8.6 | Installazione di un server WSGI | 23 |
| 8.7 | Variabili d'ambiente utilizzate | 23 |
| 8.8 | Containerizzazione del servizio | 24 |
| 9 | Deployment | 26 |

| | | |
|-----|---|----|
| 9.1 | Soluzione adottata per il dispiegamento dei servizi | 26 |
| 9.2 | Aspetti di docker-compose coinvolti | 26 |
| 9.3 | Deployment su Raspberry PI | 28 |
| 9.4 | Rappresentazione grafica dell'architettura finale | 29 |

1 Introduzione

Il seguente elaborato espone lo sviluppo di un'**integrazione IoT** atta a conferire ad un preesistente strumento per la rilevazione della qualità dell'aria caratteristiche “smart”, prevalentemente rivolte allo storage dei dati raccolti, al loro monitoraggio e all'interazione con l'utilizzatore. Il progetto si basa su quanto è stato osservato dallo sviluppatore Sören Beye (@Hypfer) che, a seguito di un'attività di *reverse engineering*, ha descritto un procedimento per l'installazione di un modulo ESP8266 all'interno del rilevatore originale, permettendo la raccolta e il processing dei dati rilevati senza alterare le funzionalità di base del sistema.

2 Il sensore VINDRIKTNING di Ikea

Lo strumento di rilevazione di qualità dell'aria utilizzato è un articolo commercializzato da Ikea sotto il nome di **VINDRIKTNING**. Quest'ultimo è in grado di rilevare la quantità di polveri sottili presenti in ambienti chiusi o all'aperto (se contenuti) in base alla classificazione PM 2.5. Sulla base di specifici threshold riportati nel manuale d'uso del sensore, il valore assoluto rilevato permette, rispettivamente, l'accensione di:

- Una luce a led di colore verde atta ad indicare un buon livello di qualità dell'aria
- Una luce a led di colore giallo atta ad indicare un degradamento della qualità dell'aria
- Una luce a led di colore rosso atta ad indicare un pessimo livello di qualità dell'aria.



Il sensore è costituito da un parallelepipedo in **ABS** di dimensioni pari a 6x6x9 cm. Al suo interno contiene:

- Un sensore **Cubic PM1006**
- Un microcontrollore che si occupa dell'accensione dei led sulla base dei dati rilevati dal sensore
- Una ventola azionata al fine di favorire il ricircolo dell'aria in prossimità del sensore

VINDRIKTNING non dispone, tuttavia, di ulteriori funzionalità, assestandosi in una fascia di prezzo inferiore a 10€ e venendo spesso proposto a supporto del purificatore d'aria FÖRNUFTIG.

3 Obiettivi e Architettura del sistema

3.1 Caratterizzazione dei requisiti

In seguito all'acquisto di due unità VINDRIKTNING e ad un breve periodo di utilizzo, sulla base delle necessità individuate sono stati descritti i seguenti requisiti:

- Possibilità di osservare e analizzare l'andamento della qualità dell'aria in un determinato lasso di tempo
- Possibilità di aggregare i dati provenienti da più sensori collocati in diverse stanze e/o diverse abitazioni
- Facoltà di interrogare i sensori da remoto, ricevendo notifiche nel caso del superamento dei threshold specificati.

3.2 Definizione dei principali servizi

Al fine di poter attuare gli obiettivi preposti, sono stati delineati i principali servizi necessari all'implementazione di un sistema di supporto ad un **flusso di dati** generato da **più sensori** connessi allo stesso dispositivo. Si rende, di conseguenza, necessario lo sviluppo di:

- Un **firmware personalizzato** in grado di permettere ai sensori di comunicare via rete le rilevazioni effettuate
- Un servizio per la **ricezione centralizzata dei dati**
- Un **database** adibito allo storage e all'interrogazione dei dati raccolti
- Un applicativo rivolto alla **fruizione** dei dati raccolti e alla **configurazione** del sistema
- Un servizio per la **notifica di uno o più utenti** in caso di cambiamenti notevoli.

Al fine di favorire in partenza lo sviluppo **indipendente** di ognuno dei servizi sopracitati, è stata adottata una strategia implementativa basata sull'organizzazione e la coordinazione di molteplici container. Tramite quest'ultimi, infatti, le risorse possono essere isolate e i processi avviati e gestiti separatamente. Personalizzazioni e modifiche possono, inoltre, essere apportate senza compromettere il funzionamento complessivo del sistema.

Si descrivono, di seguito, le tecnologie adottate e le implementazioni svolte al fine di attuare l'architettura descritta.

4 Modifiche e personalizzazioni apportate a VINDRIKTNING

4.1 Obiettivi

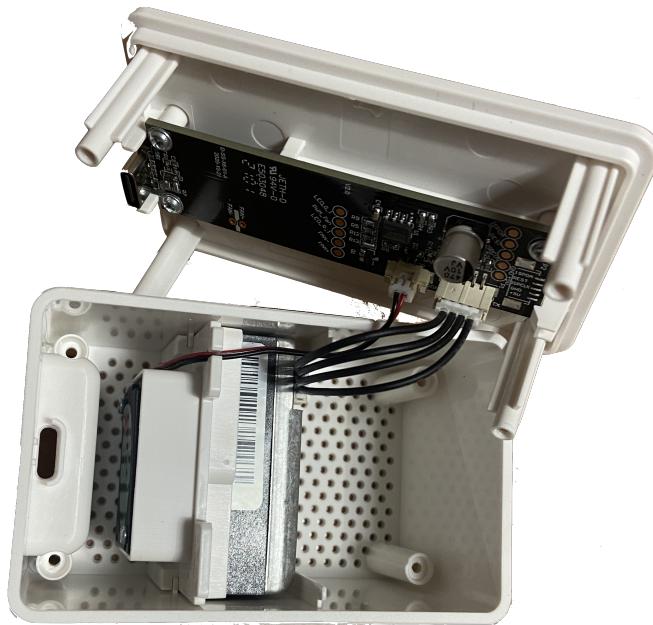
Al fine di permettere a VINDRKTNING una regolare comunicazione via rete della qualità dell'aria rilevata, è stata adottata una strategia basata su **protocollo MQTT**. In questo modo, ogni sensore connesso alla rete comunica, in qualità di **publisher**, aggiornamenti costanti ad un **broker** appositamente predisposto. Tale soluzione permette, inoltre, l'introduzione di nuovi sensori senza richiedere specifiche configurazioni e/o il riavvio del sistema.

4.2 Personalizzazione dell'hardware

Come precedentemente specificato, le modifiche apportate all'hardware del sensore si basano sull'attività di reverse engineering svolta dallo sviluppatore Sören Beye (@Hypfer) e accuratamente documentata su GitHub.

Una volta aperto il contenitore di VINDRIKTNING svitando le quattro viti che lo mantengono chiuso, è immediatamente visibile un microcontrollore adibito all'accensione dei led, al quale sono connessi, per mezzo di appositi connettori:

- Il sensore **Cubic PM1006** (la cui rilevazione viene letta tramite protocollo seriale)
- La sottostante ventola per il ricircolo dell'aria.



Si osserva, inoltre, come la breakout board del microcontrollore presenti svariati pin inutilizzati, fra cui:

- +5V e GND (passthru per l'alimentazione ricevuta tramite cavo USB)
- ISPDA e ISPCLK (che forniscono una connessione ai pin SCL e SDA per la comunicazione tramite protocollo I2C)
- REST (che fornisce un test point per il pin seriale RX)
- LED_G_1 e LED_R_1 (che forniscono un punto d'accesso per la comunicazione con led appositi)
- PWM_Fan, FAN- e FAN+ (che permettono l'alimentazione e il controllo della velocità della ventola tramite segnali PWM)



Risulta, di conseguenza, possibile la connessione di un'unità esterna che, una volta saldata ai pin di alimentazione (+5V e GND) e al test point seriale REST, permette l'acquisizione del valore rilevato dal sensore **PM1006** e letto dal microcontrollore originale. In particolare, è stata selezionata un'unità **ESP8266** (nello specifico, un clone del **D1 Mini by Wemos** prodotto da AZDelivery), che in dimensioni estremamente ridotte fornisce:

- La possibilità di essere alimentata a 5V grazie al regolatore di tensione built-in (e quindi di ricevere la corrente di alimentazione direttamente dalla porta USB di VINDRIKTNING)
- La connettività via Wi-Fi in modalità Access Point e Station Mode, entrambe necessarie allo sviluppo previsto
- La possibilità di essere programmato tramite il framework **Arduino**, con il conseguente accesso alla moltitudine di librerie disponibili in rete

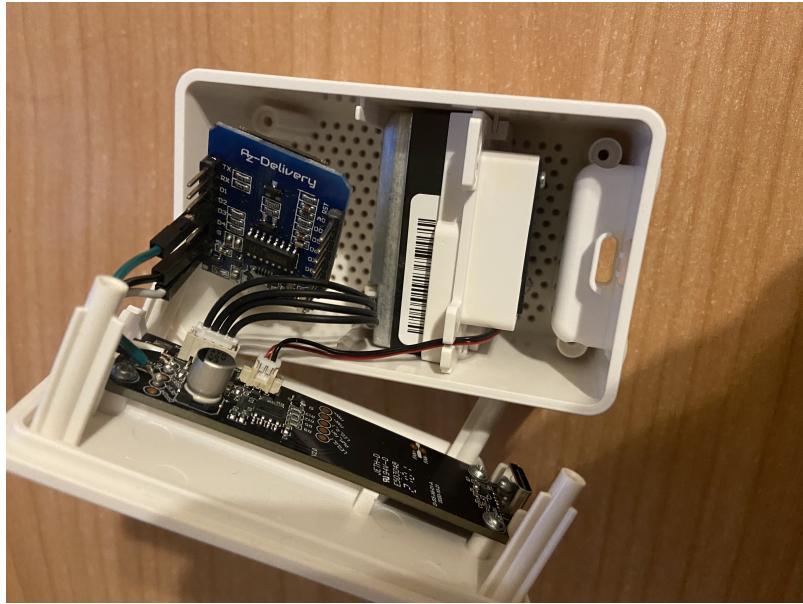
Cavi dupont appositamente modificati sono stati saldati ai punti di accesso citati in precedenza, ottenendo il seguente risultato.



Infine, i seguenti pin del modulo D1 mini sono stati impiegati per effettuare la connessione:

| D1 Mini | Punto di Accesso |
|---------|------------------|
| +5V | +5V |
| GND | GND |
| D2 | REST |

VINDRIKTNING permette, infine, l'alloggiamento del modulo al suo interno, grazie all'ampio spazio disponibile.



4.3 Costo di realizzazione

Le modifiche apportate hanno coinvolto l'utilizzo dei seguenti materiali:

- Unità VINDRIKTNING originale
- Cavi dupont
- Cacciavite di tipo PH0 per l'apertura del vano posteriore
- Strumentazione per la saldatura
- Modulo ESP8266 (D1 Mini by Wemos)

Al netto dell'attrezzatura già in possesso, il costo necessario all'apporto delle modifiche sopracitate viene di seguito descritto:

| Componente | Costo Individuale | Quantità acquistate |
|--------------|-------------------|---------------------|
| VINDRIKTNING | 9,95 € | 2 |
| D1 Mini | 4,00 € | 2 |
| Cavi Dupont | 3,00 € | 1 |

Il costo coinvolto nella personalizzazione di una singola unità VINDRIKTNING corrisponde, quindi, alla cifra di **16,95 €**. La personalizzazione di due unità richiede, invece, una spesa complessiva pari a **30,90 €**.

4.4 Implementazione di un firmware ad-hoc

A seguito dell'installazione del modulo D1 Mini è stato sviluppato un firmware parzialmente ispirato a quello proposto da Sören Beye, con l'obiettivo di fornire le seguenti funzionalità:

- Lettura e decodifica del payload inviato dal sensore sulla porta seriale
- Semplice procedura per la configurazione e la connessione di VINDRIKTNING alla rete Wi-Fi
- Persistenza dei parametri di configurazione anche in caso di riavvio del dispositivo
- Possibilità di eseguire aggiornamenti del firmware da remoto, senza richiedere la riapertura del contenitore
- Invio regolare dei dati ad un broker MQTT appositamente configurato

4.4.1 Decodifica del payload

Al fine di memorizzare efficacemente ogni singola rilevazione, viene mantenuta la struttura dati di seguito descritta.

```

struct particleSensorState_t {
    uint16_t avgPM25 = 0;
    uint16_t measurements[5] = {0, 0, 0, 0, 0};
    uint8_t measurementIdx = 0;
    boolean valid = false;
    uint8_t status = 0;
};

}

```

Quest'ultima incapsula un insieme di cinque misurazioni (svolte consecutivamente per aumentare la precisione della rilevazione), la loro media e ulteriori flag che indicano la classe di qualità rilevata dal sensore e la validità della misurazione. Il sensore viene regolarmente interrogato dal microcontrollore originale, inviando in risposta un payload di 20 byte. Di questi:

- I primi tre sono costanti (in caso di payload valido) e costituiscono l'**header**
- I byte 5 e 6 codificano il valore di qualità dell'aria rilevato

Il namespace **SerialCom** contenuto all'interno dell'header file **Utils.h** fornisce le funzionalità per:

- Leggere i dati dalla porta seriale (configurata tramite SoftwareSerial sul pin 2D del modulo ESP8266) all'interno di un buffer
- Effettuare cinque letture consecutive del valore di qualità dell'aria, calcolandone la media
- Individuare la classe di qualità alla quale appartiene il valore medio rilevato
- Verificare la validità dell'header (i cui tre byte devono corrispondere a 0x16 0x11 0x0B)
- Verificare la validità del checksum (la somma dei venti byte deve essere pari a 0)

In particolare, l'ottenimento del numero intero (codificato da due byte) relativo alla misurazione di PM2.5 da parte del sensore viene permesso dalla seguente operazione bitwise, che applica un padding destro di 8 bit al primo byte e, successivamente, effettua un OR tra il primo e secondo byte. Il risultato, una volta codificato come dato di tipo **uint16_t**, viene salvato nell'apposita struttura dati **struct particleSensorState_t**.

```
const uint16_t pm25 = (serialRxBuf[5] << 8 | serialRxBuf[6]);
```

4.4.2 Salvataggio e recupero dei parametri di configurazione

Al fine di rendere disponibile il salvataggio dei parametri di configurazione personalizzabili e il loro successivo recupero in seguito ad un eventuale riavvio del microcontrollore, all'interno dell'headerfile **Utils.h** il namespace **Config** definisce due funzioni che permettono, rispettivamente, di serializzare i parametri di configurazione in un file JSON all'interno della memoria flash del dispositivo e di leggere i parametri a partire da un eventuale file già presente in memoria.

Ciò è reso possibile dalla libreria **LittleFS**, che permette l'indicizzazione di un filesystem all'interno della memoria flash del dispositivo (la quale risulta avere una capienza pari a 4MB) tramite tecniche di wear levelling dinamico che ne limitano l'usura.

4.4.3 Aggiornamento del firmware da remoto

L'aggiornamento via rete del firmware è reso possibile dalla libreria **ArduinoOTA** (On The Air), che permette di istruire il microcontrollore alla ricezione di nuovi binari precompilati tramite una socket TCP appositamente aperta.

Successivamente, è stato possibile inoltrare aggiornamenti al microcontrollore tramite il seguente comando disponibile nel framework offerto da **PlatformIO**:

```
pio run --target upload --upload-port [VINDRIKTNING-IP]
```

La risposta ottenuta dipende dalla configurazione di ArduinoOTA all'interno della funzione di setup. L'aggiornamento di VINDRIKTNING presenta il seguente output:

```

CURRENT: upload_protocol = esptool
Uploading .pio/build/d1_mini_lite/firmware.bin
00:15:42 [DEBUG]: Options: {'esp_ip': '192.168.1.10', 'host_ip': '0.0.0.0', 'esp_port': 8266, 'host_port': 19571, 'auth': 'ikea'}
00:15:42 [INFO]: Starting on 0.0.0.0:19571
00:15:42 [INFO]: Upload size: 393296
00:15:42 [INFO]: Sending invitation to: 192.168.1.10
Authenticating...OK
00:15:42 [INFO]: Waiting for device...

Uploading: [                                         ]  0%
Uploading: [                                         ]  0%
Uploading: [                                         ]  0%
Uploading: [=                                       ]  1%
Uploading: [=                                       ]  1%
Uploading: [=                                       ]  1%
Uploading: [=                                       ]  2%
Uploading: [==                                      ]  2%
Uploading: [==                                      ]  2%

```

Al fine di prevenire upload accidentali, è stata definita una password che viene richiesta per completare la procedura di aggiornamento.

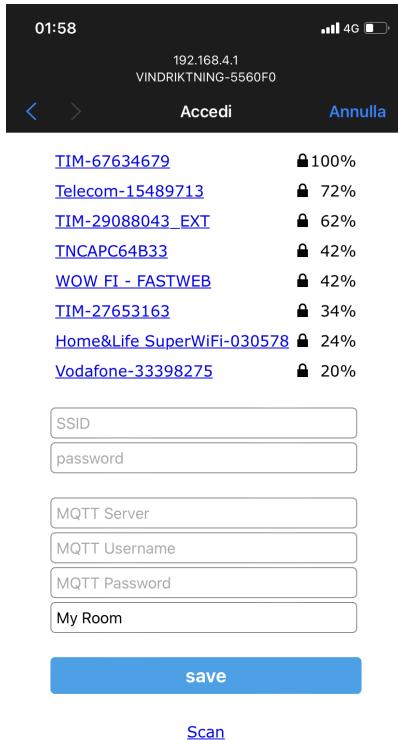
4.4.4 Configurazione di parametri personalizzati

Per permettere all'utilizzatore il collegamento alla propria rete e la configurazione di parametri per la comunicazione con il Broker MQTT, è stata impiegata la libreria **WiFiManager**. Quest'ultima permette, sulla base della presenza o assenza di una rete WiFi alla quale il microcontrollore è in grado di connettersi, la conversione automatica del modulo ESP8266 da modalità **SoftAccessPoint** a **Station** e viceversa.

In modalità SoftAccessPoint, è possibile connettersi direttamente al microcontrollore, che espone una pagina web la quale permette all'utente di effettuare lo scan delle reti disponibili, la connessione ad una rete specifica e la configurazione dei parametri personalizzabili. Nel caso di VINDRIKTNING, l'utente ha la facoltà di specificare:

- L'indirizzo IP del Broker MQTT
- La porta del Broker MQTT
- Il nome utente e la password per inviare messaggi al Broker
- Un nome da assegnare allo specifico sensore (ad esempio, *Cucina*)

La schermata di configurazione visualizzato da uno smartphone è la seguente:



Nel caso in cui la connessione alla rete WiFi selezionata vada a buon fine, il modulo passa automaticamente a modalità Station ed entra a far parte degli host connessi alla rete specificata. I dati ottenuti da WiFiManager vengono, infine, serializzati in memoria flash tramite le funzionalità precedentemente descritte. In questo modo, non si rende necessario ripetere la procedura a fronte di un semplice riavvio del sistema.

4.4.5 Comunicazione con il Broker MQTT

La gestione della connessione e dell'invio di messaggi al Broker MQTT è stata, invece, affidata alla libreria **PubSubClient**. Ad intervalli specificati dalla macro **MQTT_PUBLISH_INTERVAL_MS**, un'eventuale rilevazione valida viene inviata al Broker sull'apposito topic di aggiornamento dello stato di qualità dell'aria. Nel caso di assenza di connettività, invece, il sistema tenta regolarmente una riconnessione finché questa non risulta avvenuta.

Due ulteriori messaggi vengono, infine, comunicati al broker al momento della connessione:

- Dichiarazione di **connessione** da parte di VINDRIKTNING al Broker
- Dichiarazione di **testamento** da parte di VINDRIKTNING al Broker

Quest'ultima permette la definizione di uno specifico messaggio con flag di ritenzione attiva che viene inviato dal broker a tutti i **subscriber** in caso di una brusca ed imprevista disconnessione da parte del sensore.

Al fine di fornire ad ogni sensore un identificatore univoco per agevolarne la comunicazione con il broker, viene descritto un apposito **sensorID** costituito dalla stringa **VINDRIKTNING-[chip-id]**, dove **chip-id** rappresenta l'**UUID** del modulo ESP8266. Di conseguenza, nonostante il nome a livello utente sia quello dichiarato nell'apposito campo "Name", un ulteriore identificativo viene reso disponibile al fine di poter individuare rapidamente il sensore nell'insieme di quelli connessi alla rete.

Sulla base dell'identificatore univoco di ogni sensore, i topic coinvolti risultano, quindi, i seguenti:

1. **airquality/[sensorID]/online** (Connessione del sensore identificato da **[sensorID]**)
2. **airquality/[sensorID]/offline** (Disconnessione del sensore identificato da **[sensorID]**)
3. **airquality/[sensorID]/status** (Nuova rilevazione dal sensore identificato da **[sensorID]**)

Il codice sorgente del firmware implementato risulta accessibile all'interno del progetto PlatformIO contenuto all'interno della directory **firmware**.

5 Definizione del Broker MQTT

Al fine di poter comunicare i dati sulla rete, VINDRIKTNING necessita di un indirizzo valido che identifichi un **Broker MQTT** adibito alla ricezione all'eventuale ritenzione dei messaggi. A tal fine, è stata scelta l'adozione del software open source **Eclipse Mosquitto**.

5.1 Containerizzazione di Eclipse Mosquitto

Sulla base delle decisioni architettonali riportate in precedenza, è stato scelto di eseguire Eclipse Mosquitto all'interno di un container **Docker**, sfruttando l'immagine ufficiale.

```
FROM eclipse-mosquitto

ADD ./config/mosquitto.conf /mosquitto/config
ADD ./broker-entrypoint.sh /

ENTRYPOINT ["sh", "./broker-entrypoint.sh"]

CMD ["/usr/sbin/mosquitto", "-c", "/mosquitto/config/mosquitto.conf"]
```

A partire dall'immagine disponibile dal repository di *Docker Hub*, sono stati aggiunti:

- Il file **mosquitto.conf**, che letto all'avvio del servizio permette:
 - L'abilitazione della persistenza dei dati all'interno di un'apposita directory interna al container
 - La definizione di una destinazione per i file di log scritti dal Broker
 - La specifica della porta utilizzata e di un modello di autenticazione

```
persistence true
persistence_location /mosquitto/data

user mosquitto

log_dest file /mosquitto/log/mosquitto.log
log_dest stdout

listener 1883
allow_anonymous true
password_file passwordfile
```

- Lo script `broker-entrypoint.sh` (eseguito come entrypoint all'avvio), la cui esecuzione comporta:
 - L'attribuzione dei corretti permessi alle cartelle relative alla configurazione di Mosquitto
 - La verifica della presenza di un nome utente e una password all'avvio del container
 - La definizione delle credenziali tramite l'utility `mosquitto_passwd`

```

set -e

# Fix write permissions for mosquitto directories
chown --no-dereference --recursive mosquitto /mosquitto/log
chown --no-dereference --recursive mosquitto /mosquitto/data

mkdir -p /var/run/mosquitto \
&& chown --no-dereference --recursive mosquitto /var/run/mosquitto

if ( [ -z "${MOSQUITTO_USERNAME}" ] || [ -z "${MOSQUITTO_PASSWORD}" ] ); then
  echo "MOSQUITTO_USERNAME or MOSQUITTO_PASSWORD not defined"
  exit 1
fi

# create mosquitto passwordfile
touch passwordfile
mosquitto_passwd -b passwordfile $MOSQUITTO_USERNAME $MOSQUITTO_PASSWORD

exec "$@"

```

Una volta eseguito, il servizio risulterà accessibile alla porta 1883 del container.

6 Ricezione dei dati

6.1 Engine

La ricezione dei dati trasmessi dalle unità VINDRIKTNING è stata affidata ad un servizio denominato **Engine**. Quest'ultimo, una volta collegatosi al Broker MQTT in qualità di **subscriber** ed iscrittosi ai topic di interesse, interagisce con i dati ottenuti dai sensori connessi svolgendo le seguenti attività:

- Trasmissione dei dati al servizio centralizzato di storage
- Esecuzione di opportune routine per la notifica del superamento di determinati threshold

6.2 Containerizzazione

Il servizio è stato implementato in **Python** sfruttando le librerie **Requests** e **Paho MQTT Python Client** ed è stato, successivamente, containerizzato a partire dall'immagine **Alpine** tramite il Dockerfile di seguito riportato. Quest'ultimo prevede:

- L'installazione delle librerie necessarie
- La creazione di una cartella che andrà a contenere il file di log prodotto dal programma nel corso della sua esecuzione
- L'aggiunta del programma `engine.py` e dello script `entrypoint.sh` per l'inizializzazione del container

```
FROM alpine

RUN apk add --update --no-cache python3
RUN python3 -m ensurepip
RUN pip3 install --no-cache --upgrade pip setuptools
RUN pip3 install --no-cache --upgrade paho-mqtt
RUN pip3 install --no-cache --upgrade requests
RUN mkdir /log

ADD ./engine.py /
ADD ./entrypoint.sh /

ENTRYPOINT ["sh", "/entrypoint.sh"]

CMD ["python3", "/engine.py"]
```

Lo script `entrypoint.sh` esegue le seguenti istruzioni al fine di mantenere disponibile il logfile **antecedente** all'ultimo riavvio del sistema nel caso in cui questo sia disponibile grazie all'impiego di un volume persistente. Questo permette all'utilizzatore di individuare le motivazioni che hanno provocato la più recente interruzione del programma.

```
set -e

mkdir -p /log
touch /log/logfile.log
cp /log/logfile.log /log/report.log
cat /dev/null > /log/logfile.log
exec "$@"
```

Il codice sorgente dell'applicativo è consultabile al path `engine/engine.py` della repository.

6.3 Comunicazione con VINDRIKTNING

La ricezione dei dati trasmessi dalle unità VINDRIKTNING al Broker MQTT sui topic d'interesse richiede l'autenticazione di Engine. A tal fine, il container prevede la definizione delle variabili d'ambiente `MOSQUITTO_USERNAME` e `MOSQUITTO_PASSWORD` in riferimento, rispettivamente, al nome utente e alla password previsti da Eclipse Mosquitto a seguito della configurazione sopradescritta.

6.4 Trasmissione dei dati ricevuti

Ad ogni nuovo messaggio ricevuto, Engine comunica la nuova osservazione o l'eventuale aggiornamento di stato di un sensore ad un servizio denominato **AirPI** (descritto nel dettaglio nel corso dei capitoli successivi). La completa comunicazione fra Engine ed AirPI richiede la presenza di:

- Un endpoint per la comunicazione di **aggiornamenti di stato** di un'unità VINDRIKTNING
- Un endpoint per la comunicazione di **nuove rilevazioni** da parte di un sensore
- Un endpoint per l'innesco di un **evento di notifica** all'atto del superamento di specifici threshold

Questi ultimi vengono rispettivamente codificati, all'interno del programma, dalle seguenti variabili d'ambiente:

- `SENSOR_STATUS_ENDPOINT`
- `DATA_ENDPOINT`
- `NOTIFICATION_ENDPOINT`

La comunicazione prevede, infine, un'autenticazione di Engine presso AirAPI basata su Json Web Tokens. A tal fine, l'applicativo necessita della presenza di:

- Un endpoint per l'**autenticazione** del servizio
- Un **identificativo** e un **secret** al fine di poter richiedere un token di autenticazione

Tali parametri vengono codificati dalle seguenti variabili d'ambiente:

- `AUTH_ENDPOINT`
- `AUTH_APPNAME`
- `AUTH_APPPASS`

7 Definizione di un database per lo storage dei dati

7.1 Scelta del DBMS

Al fine di permettere il salvataggio e la successiva fruizione dei dati, è stata impiegata una base di dati organizzata come servizio indipendente, anch'esso containerizzato. In particolare, è stata scelta l'adozione del DBMS **InfluxDB** in risposta alle seguenti necessità:

- Organizzazione dei dati orientata ai **timestamp**
- Definizione di una **retention policy** al fine di permettere l'eliminazione dei dati al di fuori del periodo di interesse

7.2 Containerizzazione

La containerizzazione del servizio ha richiesto la sola definizione del seguente Dockerfile che, nello specifico, prevede l'inclusione di un opportuno script di inizializzazione all'interno dell'immagine **InfluxDB** originale al fine di inizializzare il database utilizzato e la retention policy.

```
FROM influxdb:1.8
ADD ./createdb.iql /docker-entrypoint-initdb.d/
```

In particolare, lo script `createdb.iql` (il cui contenuto è di seguito riportato) prevede la definizione di:

- Un database denominato `airquality` con retention-policy pari a 7 giorni
- Un utente con permessi di scrittura e lettura sulla base di dati
- Un utente con permessi di sola lettura sulla base di dati

```
CREATE DATABASE airquality WITH DURATION 7d
CREATE USER api WITH PASSWORD 'apisecret'
CREATE USER reader WITH PASSWORD 'read'
GRANT READ ON airquality TO api
GRANT READ ON airquality TO reader
GRANT WRITE ON airquality TO api
```

Una volta avviato, il servizio risulterà disponibile all'uso ed accessibile tramite la porta **8086** del container.

8 AirPI

Al fine di fornire uno strumento per la **ricezione centralizzata dei dati** e l'invio di **notifiche in tempo reale** all'utilizzatore, è stato implementato il servizio **AirPI**. Quest'ultimo è costituito da un'API di tipo REST (implementata tramite il microframework Flask), che può essere amministrata e configurata da uno o più utenti tramite un apposito applicativo web-based denominato **VINDRKTNING Station - Monitoring Tool**.

8.1 Autenticazione

Al fine di permettere la ricezione di dati e aggiornamenti da parte dei soli servizi Engine autorizzati e al fine di garantire l'amministrazione del sistema ai soli utenti autorizzati, AirPI presenta un sistema di **autenticazione** basato su **JSON Web Tokens**, implementato tramite il middleware **flask_jwt_extended**.

Eventuali richieste ed aggiornamenti provenienti da servizi Engine richiedono, pertanto, la presenza di un apposito **token** all'interno dell'header della richiesta. Quest'ultimo presenta una durata limitata e può essere richiesto, in cambio di apposite credenziali, interrogando l'endpoint di autenticazione **/api/auth**. Allo stesso modo, l'accesso agli endpoint utilizzati dall'applicativo Monitoring Tool richiede un token, ottenuto in seguito ad avvenuto log-in per mezzo di username e password.

8.2 Ricezione e serializzazione dei dati

La ricezione di aggiornamenti relativi ai sensori e alla qualità rilevata avviene per mezzo di richieste HTTP di tipo **POST** tramite gli endpoint:

- **/api/airquality** (predisposto alla ricezione di una nuova rilevazione)
- **/api/status** (predisposto alla registrazione di una variazione di stato da parte di uno specifico sensore)

In seguito ad una verifica della loro autenticità, i dati ricevuti vengono scritti all'interno dell'apposito database tramite apposite query di inserimento. La connessione e la successiva comunicazione con il DBMS dipendono dalla libreria **InfluxDBClient**, la quale permette l'istanziazione di una connessione con il database e il successivo inserimento dati tramite il metodo **write_points()**.

8.3 Notifica degli utenti

8.3.1 Bot Telegram

Al fine di poter notificare gli utenti in tempo reale nel caso di variazioni notevoli della qualità dell'aria misurata dai sensori, è stato definito un **Bot Telegram** in grado di:

- Comunicare variazioni di qualità tramite messaggi inviati agli utenti iscritti
- Permettere agli utenti la verifica dello stato del sistema tramite semplici comandi

8.3.2 Implementazione

Considerato il limitato insieme di funzionalità necessarie, l'implementazione del bot non si appoggia a librerie esterne ed è consultabile al percorso file **airpi/app/bot.py**. In particolare, la ricezione di nuovi messaggi avviene tramite una costante attività di **long polling** all'endpoint **getUpdates** dell'API di Telegram, mentre l'invio di eventuali risposte e notifiche avviene tramite apposite richieste all'endpoint **sendMessage**.

Il bot può essere istanziato come di seguito, utilizzando come **token** quello fornito da **BotFather** in seguito alla procedura di creazione del bot:

```
from bot import Bot

b = Bot('token-provided-by-telegram')
```

Successivamente, è possibile definire apposite funzioni di callback da eseguire in seguito alla ricezione di specifici messaggi. In particolare, la specifica avviene secondo lo schema seguente.

```

def callback_routine(chat_id, username, params):
    # pass

b.on('/command', callback_routine)

```

L'invio di messaggi è, infine, reso disponibile tramite il metodo `push_notification`, che prevede la presenza di due parametri in riferimento al contenuto del messaggio e al gruppo di uno o più utenti a cui recapitarlo (identificati dal rispettivo chat id).

```
b.push_notification('message', [chat_id1, ..., chat_idN])
```

8.3.3 Comandi implementati

Sulla base delle funzionalità necessarie, nel caso di AirPI sono stati implementati i seguenti comandi:

- `/status` (permette all'utente di ottenere informazioni sullo stato dei sensori noti)
- `/info [sensorName]` (permette di ottenere l'ultima rilevazione effettuata da sensori avente nome corrispondente)
- `/bind` (permette agli utenti abilitati di attivare la ricezione di notifiche)
- `/start` (permette l'invio di un messaggio all'inizio della conversazione)

8.3.4 Invio di notifiche

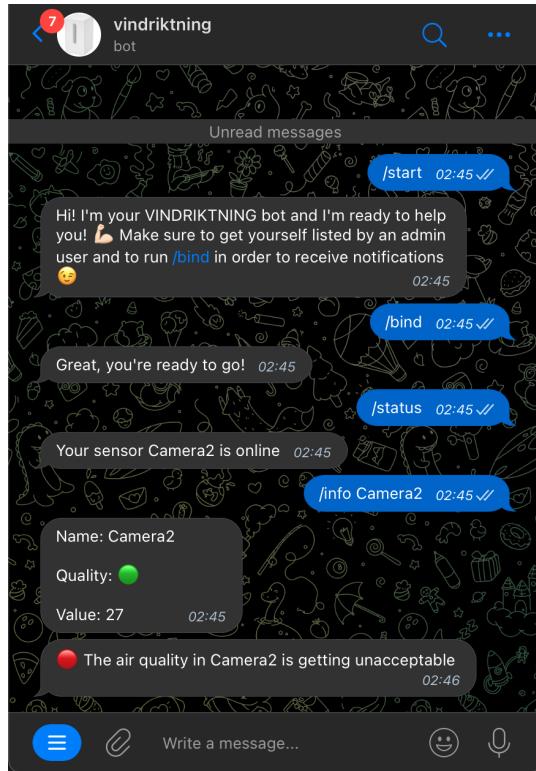
L'invio di specifici messaggi di notifica avviene in seguito ad apposite richieste HTTP di tipo POST inviate all'endpoint `/api/bot/notification` dal servizio Engine, specificando il messaggio da trasmettere a tutti gli utenti che hanno, in precedenza, eseguito il comando `/bind`. In particolare, come trattato in precedenza, l'implementazione di Engine prevede l'invio di uno dei seguenti messaggi ad ogni variazione di **classe di qualità** rilevata da un'unità VINDRKTNING:

| Messaggio | Classe di qualità |
|---|-------------------|
| The air quality in [sensorName] is getting good | 0 |
| The air quality in [sensorName] is getting unpleasant | 1 |
| The air quality in [sensorName] is getting unacceptable | 2 |

8.3.5 Esempio di conversazione

Di seguito viene riportato un esempio di conversazione, nel corso della quale:

1. Viene eseguito il comando `/start` (chiamato automaticamente all'inizio di una conversazione con un bot)
2. Viene eseguito il comando `/bind`
3. Viene richiesto lo stato dei sensori noti
4. Vengono richieste informazioni relative al sensore `Camera2`
5. Viene ricevuta una notifica relativa ad un peggioramento della qualità dell'aria



Si osserva, in particolare, come in risposta al comando `/info` l'utente riceva in risposta i parametri:

- **Sensor Name** (nome del sensore in questione)
- **Quality** (classe di qualità rilevata, rappresentata dal rispettivo colore attualmente attivo sul sensore)
- **Value** (valore assoluto relativo alla misurazione PM2.5 effettuata dal sensore)

Questi ultimi vengono ottenuti per mezzo della seguente query InfluxDB che, eseguita tramite InfluxDBClient, permette di ottenere la più recente rilevazione serializzata nel corso dell'ultimo minuto in relazione ad uno specifico sensore:

```
SELECT last("pm25"), "quality"
FROM "airquality"
WHERE time > now() - 1m AND
"name"=sensorName
```

Inoltre, si specifica come la trasmissione di notifiche coincida con il superamento di specifiche soglie (indicate nel manuale d'uso di VINDRIKTNING) per le quali consegue il cambiamento del colore identificativo della qualità rilevata.

Infine, l'invio di messaggi non riconosciuti e/o malformati, oppure l'invio di messaggi da parte di utenti non autorizzati, non solleva alcuna reazione da parte del bot.

8.4 Gestione delle utenze

8.4.1 Integrazione di un database relazionale

Al fine di memorizzare le **credenziali** relative agli utenti in grado di accedere all'applicativo web-based e i **nominativi** degli utenti telegram in grado di interagire con il bot telegram, è stato integrato l'utilizzo di un database relazionale all'interno dell'implementazione di AirPI.

In particolare, è stato impiegato un DBMS SQLite (il cui file di riferimento risulta disponibile al percorso file `/airpi/app/appdb.db`) interrogato per mezzo delle funzionalità fornite dalla libreria **SQLAlchemy** che, in qualità di Object Relational Mapper, permette l'interazione con un generico database relazionale sfruttando caratteristiche proprie del paradigma Object Oriented (facilitando, così, un'eventuale transizione ad un DBMS diverso).

8.4.2 Gestione degli utenti Telegram

La tabella `telegram_user` della base di dati permette la memorizzazione, tramite apposita richiesta da parte di un utente amministratore, di uno o più nomi utenti relativi a profili Telegram ai quali viene a tutti gli effetti concessa la facoltà di dialogare con il bot implementato. Tuttavia, come delinato in precedenza, il salvataggio nel database del **chat id** relativo ad ogni utente avviene solamente in seguito all'esecuzione del comando `/bind` da parte dell'utente stesso. Tale strategia fornisce una forma di sicurezza **bidirezionale**, in quanto garantisce:

- Che utenti non abilitati non possano comunicare con il bot
- Che un amministratore non possa sfruttare il bot per inviare messaggi indesiderati

Il modello SQLAlchemy definito per la tabella è il seguente:

```
class TelegramUser(db.Model):
    username = db.Column(db.String(50), primary_key=True)
    chat_id = db.Column(db.Integer, nullable=True)
```

8.4.3 Gestione degli utenti di Monitoring Tool

La tabella `user` della base di dati permette, invece, la memorizzazione degli utenti abilitati ad accedere al servizio Monitoring Tool. In particolare, per ogni utente vengono definiti:

- Un nome utente univoco
- Una password (della quale viene memorizzato un hash computato tramite l'utilizzo di SHA256+Salt)
- Una flag che definisce se l'utente è di tipo amministratore o meno.

In particolare, gli utenti amministratori si distinguono da quelli regolari in quanto:

- Presentano la possibilità di aggiungere e rimuovere utenti, o di modificarne le credenziali
- Presentano la possibilità di abilitare e disabilitare utenti al dialogo con il bot telegram

Il modello SQLAlchemy definito per la tabella è il seguente:

```
class User(db.Model):
    id = db.Column(db.Integer, db.Sequence('user_id_seq'), primary_key=True)
    name = db.Column(db.Text, unique=True)
    password = db.Column(db.Text)
    is_admin = db.Column(db.Boolean, default=False)
```

8.4.4 Struttura della directory

L'implementazione di AirPI è contenuta all'interno del percorso file `airpi/app` e prevede la presenza dei seguenti file:

- `app.py` (contenente l'implementazione di AirPI)
- `bot.py` (contenente l'implementazione del Bot Telegram)
- `first_user.py` (contenente uno script finalizzato all'inizializzazione del primo utente del database)
- `appdb.db` (contenente il database SQLite utilizzato per gestire le utenze)
- `static`
 - `static/css` (directory contenente i file CSS utilizzati da Monitoring Tool)
 - `static/img` (directory contenente le immagini utilizzate da Monitoring Tool)
 - `static/js` (directory contenente gli script e le librerie JavaScript utilizzate da Monitoring Tool)
- `templates` (directory contenente i template HTML descritti tramite Jinja2)

8.5 Monitoring Tool

Come riportato in precedenza, al fine di fornire uno strumento rivolto alla fruizione dei dati raccolti dai sensori e all'amministrazione delle utenze del sistema è stato definito un applicativo web-based denominato **Monitoring Tool**.

L'implementazione dell'app si appoggia alla libreria di templating HTML **Jinja2** integrata in Flask, che permette di inviare al richiedente pagine HTML appositamente formattate. Queste ultime sfruttano, a loro volta, richieste **fetch** (implementate lato client utilizzando il linguaggio JavaScript) per interrogare gli endpoint di AirPI rivolti all'ottenimento delle informazioni necessarie. Al fine di ottenere un layout responsive è stato utilizzato il framework **Bootstrap 4**, al quale sono state apportate minime integrazioni all'interno di un apposito file di stile.

Le funzionalità implementate, in particolare, prevedono:

- Una homepage che, a seguito di una corretta autenticazione da parte dell'utente, permette la visualizzazione di grafici riassuntivi in relazione all'andamento della qualità dell'aria nelle ultime 24 ore
- Una pagina per l'aggiunta e la rimozione di utenti Telegram in grado di dialogare con il bot (accessibile solo ad utenti amministratori)
- Una pagina per la creazione, modifica e rimozione di utenti (accessibile solo ad utenti amministratori)
- Una pagina per la modifica delle credenziali relative al proprio profilo
- Una pagina per permettere il log-in agli utenti autorizzati

8.5.1 Barra di Navigazione

L'accesso alle funzionalità previste da **Monitoring Tool** è permesso dall'apposita barra di navigazione, che presenta un insieme di opzioni diverse sulla base del livello di autenticazione dell'utente. In particolare,

- Nel caso di utenti non autenticati, non risulta presente alcuna voce
- Nel caso di utenti autenticati e non amministratori, risultano accessibili:
 - La pagina principale
 - La pagina di configurazione del profilo
 - L'opzione di Logout
- Nel caso di utenti autenticati e amministratori, risultano accessibili:
 - La pagina principale
 - La pagina di configurazione delle utenze Telegram
 - La pagina di configurazione delle utenze di Monitoring Tool
 - La pagina di configurazione del profilo
 - L'opzione di Logout

In particolare, ad ogni pagina corrisponde uno specifico endpoint di AirPI:

| Pagina | Endpoint |
|--------------------------------|-----------|
| Homepage | / |
| Login | /login |
| Configurazione utenti | /users |
| Configurazione utenti telegram | /telegram |
| Configurazione profilo | /me |
| Logout | /logout |

8.5.2 Homepage

La pagina principale (accessibile solamente a seguito di un'autenticazione da parte di utenti amministratori e non risulta incentrata sulla visualizzazione di due grafici (selezionabili per mezzo degli appositi selettori) riportanti:

- L'andamento della qualità dell'aria per ogni sensore noto nelle ultime 24 ore (nel caso del diagramma a linee)
- La qualità media rilevata da ogni sensore nel corso delle ultime 24 ore (nel caso del diagramma a barre)

L'implementazione dei grafici, in particolare, si appoggia alle librerie **chart.js**, **moment.js** e **chartjs-plugin-colorschemes.js**. Il loro popolamento si basa, invece, su apposite richieste HTTP di tipo GET effettuate agli endpoint **/api/data/line** e **/api/data/bar** definiti da AirPI e accessibili solo previa autenticazione. Questi ultimi prevedono:

- L'esecuzione di una query parametrizzata all'interno del database InfluxDB
- L'organizzazione della risposta in un array di oggetti JSON
- La trasmissione dei dati in risposta alla richiesta

La **query** impiegata per l'ottenimento dei dati necessari alla produzione del diagramma a linee è la seguente:

```
SELECT mean("pm25")
FROM "airquality"
WHERE time > now() - 24h
GROUP BY time(10m), "name"
fill(none)
```

In particolare, la restituzione del valore medio per ogni intervallo di dieci minuti prevista dalla query fornisce un livello di **aggregazione** dei dati serializzati all'interno del database al fine di ridurre la mole di informazioni trasmesse e, successivamente, utilizzate per la produzione e renderizzazione del grafico.

La query impiegata per l'ottenimento dei dati necessari alla produzione del diagramma a barre è, invece, la seguente:

```
SELECT mean("pm25")
FROM "airquality"
WHERE time > now() - 24h
GROUP BY "name"
```

8.5.3 Gestione degli utenti Telegram

La pagina relativa all'abilitazione degli utenti telegram prevede la renderizzazione di un'apposita tabella riportante gli **utenti attualmente abilitati** con annesso **chat id** (se disponibile). Inoltre, la rimozione di un utente è permessa a seguito di un click sull'apposito pulsante di eliminazione dell'utente, mentre l'aggiunta di un nuovo utente è permessa dal form modale accessibile clickando il pulsante riportante la scritta **Add a User**. Quest'ultimo propone un menu a comparsa che permette all'utente di digitare il nome utente da abilitare e inviare la richiesta.

Le operazioni di aggiunta di un nuovo utente, rimozione di un utente esistente e di ottenimento di tutti gli utenti attualmente abilitati sono permesse dall'endpoint `/api/telegram` di AirPI, interrogato rispettivamente da richieste HTTP di tipo POST, DELETE e GET.

In particolare, a seguito di ogni richiesta vengono apportate le dovute modifiche all'interno della tabella **TelegramUser** del database relazionale impiegato per l'organizzazione delle utenze.

8.5.4 Gestione degli utenti di Monitoring Tool

La pagina relativa all'inserimento, rimozione e modifica degli utenti di Monitoring Tool, similmente a quanto illustrato nel caso degli account Telegram, prevede il popolamento di una tabella riportante, per ogni utente, il relativo nome assieme ad un indicatore riferito all'eventuale possesso dei permessi di amministrazione. La rimozione di un utente è possibile tramite l'apposito pulsante di eliminazione, mentre la modifica delle informazioni è permessa da un form modale accessibile clickando sul pulsante di configurazione. La creazione di un nuovo utente è, infine, possibile tramite il form accessibile clickando sul pulsante riportante la scritta **Add New Profile**.

Le operazioni di visualizzazione, aggiunta, modifica e rimozione di un utente sono permesse dall'endpoint `/api/users` di AirPI, interrogato rispettivamente da richieste HTTP di tipo GET, POST, PUT e DELETE. In particolare, a seguito di ogni richiesta vengono apportate le dovute modifiche all'interno della tabella **User** del database relazionale impiegato per l'organizzazione delle utenze.

L'endpoint `/api/me` di AirPI permette, infine, ad utenti amministratori e non di apportare modifiche in relazione alle **sole** credenziali del proprio profilo, senza poter variare, però, il livello di permessi associato all'utente.

8.6 Installazione di un server WSGI

Nonostante Flask permetta l'esecuzione di un server di sviluppo tramite l'utilità `flask run`, è stato scelto di eseguire AirPI per mezzo di un server di produzione di tipo WSGI al fine di garantire maggiore efficienza, stabilità e sicurezza da parte dell'applicativo. È stata, quindi, impiegata la libreria **Waitress** che, una volta installata tramite `pip`, permette l'esecuzione di un'app Flask tramite il comando:

```
waitress-serve app:app
```

8.7 Variabili d'ambiente utilizzate

Al fine di una corretta esecuzione, l'applicativo prevede la presenza delle seguenti variabili ambientali:

| Variabile | Descrizione |
|-----------------------|---|
| AUTH_APPNAME | Nome utente del servizio di Engine a fini di autenticazione |
| AUTH_APPPASS | Secret trasmesso dal servizio di Engine a fini di autenticazione |
| TELEGRAM_BOT_TOKEN | Token utilizzato dalla classe <code>Bot</code> al fine di controllare il bot telegram creato |
| INFLUXDB_API_USER | Nome utente per connessione ad InfluxDB tramite <code>InfluxDBClient</code> |
| INFLUXDB_API_PASSWORD | Password per connessione ad InfluxDB tramite <code>InfluxDBClient</code> |
| AUTH_USERNAME | Nome utente del primo utente amministratore, creato automaticamente per Monitoring Tool |
| AUTH_USERPASS | Password del primo utente amministratore, creato appositamente per accedere a Monitoring Tool |

8.8 Containerizzazione del servizio

Come per i servizi precedentemente descritti, anche per AirPI è stata adottata una strategia basata sull'utilizzo di Docker al fine di racchiudere il servizio all'interno di un apposito container. A tal fine, a partire dall'immagine ufficiale **Alpine** è stato prodotto il Dockerfile riportato di seguito.

```
FROM alpine

RUN apk add --update --no-cache build-base
RUN apk add --update --no-cache sqlite
RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python
RUN python3 -m ensurepip
RUN apk add py3-sqlalchemy
RUN pip3 install --no-cache --upgrade pip setuptools
RUN pip3 install --no-cache --upgrade flask
RUN pip3 install --no-cache --upgrade influxdb
RUN pip3 install --no-cache --upgrade flask_sqlalchemy
RUN pip3 install --no-cache --upgrade flask_jwt_extended
RUN pip3 install --no-cache --upgrade passlib
RUN pip3 install --no-cache --upgrade waitress
RUN pip3 install --no-cache --upgrade pyopenssl

RUN mkdir /app
RUN mkdir /log
RUN mkdir /certificates

WORKDIR /app

COPY ./app /app
COPY ./entrypoint.sh /entrypoint.sh

# RUN export FLASK_APP=app.py
# RUN export FLASK_ENV=development

ENTRYPOINT ["sh", "/entrypoint.sh"]

# CMD ["flask", "run", "--host=0.0.0.0", "--port=8080"]
CMD ["waitress-serve", "app:app"]
```

Quest'ultimo, in particolare, prevede:

- L'installazione del package **build-base**, necessario al building di alcune librerie installate nel corso dei passaggi successivi
- L'installazione di **Python**, **pip** e delle librerie necessarie alla corretta esecuzione dell'applicativo
- La creazione delle directory predisposte al contenimento del codice sorgente dell'applicativo e di eventuali file di log
- Il popolamento della cartella **/app** con i file presenti al percorso **airpi/app** della macchina host
- L'aggiunta dello script **/entrypoint.sh**, consultabile al path **airpi/entrypoint.sh**
- La definizione dello script di entrypoint all'interno del container
- La definizione del comando di esecuzione del server di produzione tramite **Waitress**

Le linee commentate permettono, se opportunamente sostituite al comando finale, l'esecuzione di AirPI per mezzo del server di sviluppo integrato in Flask. Il comando di entrypoint prevede, invece, l'esecuzione del codice contenuto

in `/app/first_user.py` prima dell'avvio dell'applicativo. In particolare, lo script Python (il cui codice è di seguito riportato) prevede, sulla base delle sopracitate variabili d'ambiente `AUTH_USERNAME` e `AUTH_USERPASS`, la configurazione del primo utente amministratore di AirPI.

```
import sqlite3
from passlib.hash import pbkdf2_sha256
import os
import sys

USERNAME = os.environ['AUTH_USERNAME']
PASSWORD = os.environ['AUTH_USERPASS']
DB = '/app/appdb.db'

conn = sqlite3.connect(DB)
c = conn.cursor()

try:
    c.execute(
        'INSERT INTO user(name, password, is_admin)
         VALUES (?, ?, 1)',
        (USERNAME, pbkdf2_sha256.hash(PASSWORD))
    )
    conn.commit()
except Exception as e:
    conn.rollback()
finally:
    conn.close()

sys.exit(0)
```

9 Deployment

9.1 Soluzione adottata per il dispiegamento dei servizi

Il dispiegamento dei servizi descritti è reso possibile dallo strumento **docker-compose**, che permette la definizione e organizzazione di applicativi multi-container tramite l'apposito file **docker-compose.yaml** consultabile alla root directory della repository.

In particolare, le due principali soluzioni di deployment risultano:

1. L'organizzazione di un singolo stack multi-container (ideale per un utilizzo all'interno di una rete domestica)
2. L'organizzazione di due stack multi-container, al fine di rafforzare l'indipendenza fra i servizi di MQTT Brokering ed Engine e quelli di Database ed AirPI (ideale al fine di poter dispiegare il database InfluxDB ed AirPI nel cloud)

9.2 Aspetti di docker-compose coinvolti

All'interno del file **docker-compose**, per ogni servizio vengono specificati:

- **Dockerfile** e **directory di riferimento** per il build della rispettiva immagine
- **Politiche di riavvio** dei container in seguito ad errori
- **Port mapping** (al fine di esporre le porte necessarie all'utilizzo)
- **Volume mapping** (al fine di mappare su volumi persistenti directory interne ai container)
- **Variabili d'ambiente** necessarie all'esecuzione
- **Nome** dei container
- **Dipendenze fra container** (al fine di permettere un corretto avvio e spegnimento del sistema, oltre che un comportamento consistente dello stack nel caso di errori)

Al fine di agevolare l'eventuale modifica delle variabili d'ambiente necessarie, quest'ultime sono state descritte all'interno di un file **.env** (consultabile all'interno della root directory della repository). Quest'ultimo viene, infatti, automaticamente letto dal file **docker-compose** durante la sua esecuzione.

La specifica delle variabili d'ambiente in **docker-compose.yaml** avviene, pertanto, mediante la sintassi

- **VARIABLE_NAME = \${VARIABLE_NAME_ENVFILE}**

Dove **VARIABLE_NAME** riferisce il nome assunto dalla variabile all'interno del container e **VARIABLE_NAME_ENVFILE** riferisce il nome della variabile della quale acquisire il valore specificato all'interno del file **.env**. Nello specifico, al fine di evitare ambiguità, è stato scelto di mantenere identico il nome di ogni variabile rispetto al corrispondente riferimento all'interno del file.

Il risultato finale viene riportato di seguito:

```
version: "3"
services:
  engine:
    build:
      context: ./engine
    container_name: engine
    environment:
      - MOSQUITTO_USERNAME=${MOSQUITTO_USERNAME}
      - MOSQUITTO_PASSWORD=${MOSQUITTO_PASSWORD}
      - SENSOR_STATUS_ENDPOINT=${SENSOR_STATUS_ENDPOINT}
      - DATA_ENDPOINT=${DATA_ENDPOINT}
      - NOTIFICATION_ENDPOINT=${NOTIFICATION_ENDPOINT}
      - AUTH_ENDPOINT=${AUTH_ENDPOINT}
      - AUTH_APPNAME=${AUTH_APPNAME}
      - AUTH_APPPASS=${AUTH_APPPASS}
    depends_on:
      - "broker"
```

```

    - "airpi"
  restart: always

airpi:
  build:
    context: ./airpi
  container_name: airpi
  environment:
    - INFLUXDB_API_USER=${INFLUXDB_API_USER}
    - INFLUXDB_API_PASSWORD=${INFLUXDB_API_PASSWORD}
    - TELEGRAM_BOT_TOKEN=${TELEGRAM_BOT_TOKEN}
    - AUTH_APPNAME=${AUTH_APPNAME}
    - AUTH_APPPASS=${AUTH_APPPASS}
    - AUTH_USERNAME=${AUTH_USERNAME}
    - AUTH_USERPASS=${AUTH_USERPASS}
  ports:
    - "8000:8080"
  depends_on:
    - "database"
  restart: always

broker:
  build:
    context: ./broker
  environment:
    - MOSQUITTO_USERNAME=${MOSQUITTO_USERNAME}
    - MOSQUITTO_PASSWORD=${MOSQUITTO_PASSWORD}
  container_name: broker
  ports:
    - "1883:1883"
  volumes:
    - ./broker/log:/mosquitto/log
  restart: always

database:
  build:
    context: ./influxdb
  container_name: database
  environment:
    - INFLUXDB_ADMIN_USER=${INFLUXDB_ADMIN_USER}
    - INFLUXDB_ADMIN_PASSWORD=${INFLUXDB_ADMIN_PASSWORD}
  volumes:
    - db:/var/lib/influxdb
  restart: always

volumes:
  grafana:
  db:

```

9.3 Deployment su Raspberry PI

testo

9.4 Rappresentazione grafica dell'architettura finale

