

UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

LAUREA TRIENNALE - INFORMATICA

Sviluppo di un'architettura in cloud per la virtualizzazione di sensori e modelli di crescita in agricoltura

Autore:

Gianni DONADON

Matricola:

VR437200

Relatore:

Davide QUAGLIA

Co-Relatore:

Elia BRENTAROLLI

Anno accademico 2022-2023

Indice

Introduzione	4
1 Il protocollo MQTT	5
1.1 Definizione e caratteristiche principali	5
1.2 L'architettura publisher/subscriber in MQTT	6
2 Broker MQTT Mosquitto	7
2.1 Implementazione nel progetto	7
2.2 Installazione	7
2.2.1 Comandi per l'installazione	7
2.2.2 Come controllare l'esecuzione del servizio	7
2.2.3 Problemi nell'installazione	8
2.3 Configurazione	8
2.3.1 Configurazione iniziale	9
2.3.2 Configurazione parametri	9
2.3.3 Metodi di autenticazione	10
2.3.4 Certificati SSL/TLS	11
2.4 Disinstallazione	13
2.4.1 Eliminazione della configurazione	13
2.4.2 Rimozione del broker	14
3 Client MQTT	15
3.1 Implementazione nel progetto	15
3.2 Installazione	15
3.2.1 Python	15
3.2.2 L'ambiente virtuale	15
3.2.3 I requisiti	16
3.2.4 La libreria paho-mqtt	17
3.3 Le componenti del client	17
3.3.1 Gestione della connessione	17
3.3.2 Invio di messaggi	20
3.3.3 Ricezione di messaggi	20
3.3.4 Autenticazione	20
3.3.5 Connessione con TLS	21
3.4 Implementazione con altri script	21
3.5 Esecuzione automatica	22
3.5.1 Installazione	22

3.5.2	Configurazione di un processo	22
3.5.3	Comandi della modalità interattiva	24
4	Flask web server	26
4.1	Implementazione nel progetto	26
4.2	Installazione	26
4.3	Strutturazione del web server	26
4.3.1	La base del flusso di esecuzione	27
4.3.2	Blueprint	31
4.3.3	Template	35
4.3.4	I moduli di supporto	38
4.4	Funzionalità del progetto	38
4.4.1	Accesso e autenticazione	39
4.4.2	Interfaccia principale	40
4.4.3	Gestione account	40
4.4.4	Gestione dei file	41
4.4.5	Creazione dei modelli	41
4.4.6	Chi siamo	43
4.5	Deployment	43
4.5.1	Gunicorn	44
4.5.2	NGINX	45
4.5.3	Avvio automatico di Gunicorn	46
5	Home Assistant	47
5.1	Implementazione nel progetto	47
5.2	Servizi di auto-discovery	47
5.3	Integrazione con MQTT	49
5.4	Visualizzazione dei dati	50
	Conclusioni	52
	Ringraziamenti	53
A	Configurazioni broker MQTT	54
A.1	/mosquitto.conf	54
A.2	/conf.d/localhost-TLS.config	54
A.3	Script rinnovo certificati	55
B	Skeleton client MQTT	57
C	Configurazioni Home Assistant	62
C.1	Definizioni di sensori MQTT	62
C.2	Definizioni di cards	63

Elenco delle figure

1.1	Architettura Pub/Sub	5
4.1	Pagina di Login	39
4.2	Pagina di Registrazione	39
4.3	Pagina di Dashboard	40
4.4	Pagina di gestione account	40
4.5	Pagina di gestione file	41
4.6	Pagina del calcolo del modello di regressione	42
4.7	Pagina di chi siamo	43
5.1	Plancia RL	51

Introduzione

Lo scopo di questo documento è fornire una guida per creare un'architettura di sensori e modelli di crescita basata su MQTT. Il capitolo [1](#) spiega brevemente il protocollo MQTT e l'architettura publisher/subscriber. Il capitolo [2](#) è dedicato all'installazione e alla configurazione del Broker MQTT Mosquitto. Nel capitolo [3](#) viene spiegato come usare una libreria per implementare i Client MQTT e viene indicato come eseguirli automaticamente. Nel capitolo [4](#) si affronta la struttura e il funzionamento del progetto Flask fino al deployment. Infine nel capitolo [5](#) si mostra come configurare Home Assistant per la visualizzazione dei dati. Alla fine del documento sono presenti varie appendici che riportano codici e script rilevanti.

Capitolo 1

Il protocollo MQTT

1.1 Definizione e caratteristiche principali

Message Queuing Telemetry Transport, abbreviato con la sigla MQTT, è un protocollo di messaggistica usato su architetture con pattern publisher/subscriber. Si tratta di un protocollo open-source che funziona sullo stack TCP/IP, ma può essere implementato anche su altri stack come quello Bluetooth. È stato pensato per le comunicazioni in cui è necessario usare le risorse e la banda disponibili in modo efficiente e con un basso impatto energetico. Infatti questo protocollo è molto popolare in ambito IoT, Internet of Things, per la sua implementazione leggera e viene impiegato in vari casi come:

- Raccolta dati da sensori e pubblicazione in un server.
- Pubblicazione di dati mission-critical direttamente da un sensore al dispositivo dell'utente.
- Configurazione remota di dispositivi IoT.
- Invio di configurazione o aggiornamenti a tutti i dispositivi connessi.

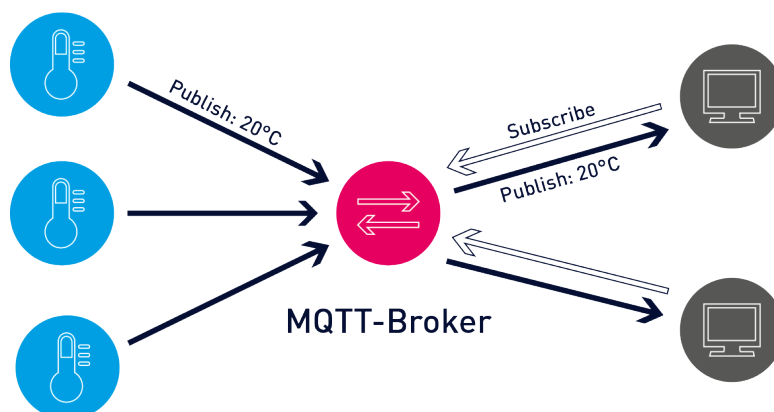


Figura 1.1: Architettura Pub/Sub

1.2 L'architettura publisher/subscriber in MQTT

Il modello architetturale publisher/subscriber si basa su tre componenti:

- **Publisher**
Invia messaggi al broker.
- **Subscriber**
Richiede al broker di ricevere dei messaggi.
- **Broker**
Trasferisce i messaggi dai publisher ai subscriber.

I messaggi all'interno dell'architettura possono essere pubblicati in vari formati e possono essere filtrati in due modi: in base al topic ("canale") oppure in base al contenuto. Il filtro con topic prevede che i messaggi vengano categorizzati secondo dei canali. Il publisher trasmette su un canale e il subscriber si deve iscrivere a quel canale per ricevere solo quei messaggi. Invece nel filtro con contenuto i messaggi non sono categorizzati. I publisher inviano i messaggi e i subscriber definiscono un criterio con cui ricevere i messaggi.

L'architettura permette il disaccoppiamento tra publisher e subscriber: un publisher non conosce né gli altri publisher né i subscriber, e lo stesso vale per i subscriber. Inoltre permette la comunicazione asincrona: il publisher non attende una risposta dopo l'invio.

Nel caso specifico del protocollo MQTT, i messaggi sono organizzati secondo il filtro con topic e vengono pubblicati in uno dei seguenti formati: JSON, XML, dati binari o testo. I componenti assumono le seguenti denominazioni:

- **Client publisher**
Pubblica sul broker messaggi su uno specifico topic. Questa operazione viene fatta da dispositivi come sensori.
- **Client subscriber**
Invia al broker la richiesta di iscrizione a uno o più topic. Questa operazione viene fatta da dispositivi che devono raccogliere o visualizzare i dati.
- **Broker MQTT**
È il sistema backend, chiamato impropriamente server, che coordina i diversi messaggi che arrivano dai client. Si può comparare a un ufficio postale che riceve i messaggi dai client "publisher" e li smista secondo un topic specifico. Successivamente li deve distribuire ai client "subscriber" iscritti a quel topic.

Un client in base alle funzioni che deve svolgere può essere publisher o subscriber oppure entrambi.

Capitolo 2

Broker MQTT Mosquitto

2.1 Implementazione nel progetto

Alla base del progetto c'è il broker che permette la comunicazione tra le varie componenti. Si è scelto di usare come broker Eclipse Mosquitto perché è open source e implementa il protocollo MQTT fino alla versione 5.0. Di seguito viene spiegato come installare Mosquitto da terminale e come configurarlo per l'accesso locale e da remoto. La guida è stata pensata per sistemi Linux.

2.2 Installazione

2.2.1 Comandi per l'installazione

L'installazione si può eseguire in più modi, come spiegato nel sito ufficiale [1]:

- **Metodo 1** - tramite la repository ufficiale (consigliato).
In questo modo viene installata l'ultima versione disponibile delle release.

```
1 sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
2 sudo apt-get update
3 sudo apt-get install mosquitto
```

- **Metodo 2** - tramite la repository di linux.
In questo modo viene installata l'ultima versione disponibile nel package manager.

```
1 sudo apt-get install mosquitto
```

2.2.2 Come controllare l'esecuzione del servizio

Il broker MQTT viene eseguito all'avvio della macchina tramite un servizio che viene aggiunto nell'installazione. Il servizio si trova in:

```
/etc/systemd/system/mosquitto.service
```


È possibile controllare il broker con i consueti comandi per la gestione di un servizio elencati di seguito. È necessario usare sudo, quindi si devono avere i diritti di root.

- lo stato del servizio:

Solo questa opzione ha la possibilità di essere eseguita senza il comando sudo.

```
1 sudo service mosquitto status
```

- fermare il servizio:

```
1 sudo service mosquitto stop
```

- avviare il servizio:

```
1 sudo service mosquitto start
```

- riavviare il servizio:

```
1 sudo service mosquitto restart
```

Per vedere il log dettagliato di Mosquitto, si deve guardare nel file:

```
/var/log/mosquitto/mosquitto.log
```

2.2.3 Problemi nell'installazione

Si potrebbero presentare principalmente 2 tipi di problemi:

- Mancata creazione della configurazione.
Vedere nella sezione [2.3.2](#) in cui si possono trovare esempi di configurazione.
- Nel caso di una precedente installazione:
 - Si potrebbe avere un errore di database.
In questo caso si deve cancellare il database che si trova in:
var/lib/mosquitto/mosquitto.db
Alla successiva esecuzione di Mosquitto questo database viene ricreato.
 - Si potrebbero avere file di configurazione sbagliati o corrotti.
In questo caso si consiglia di creare un backup completo della cartella **/etc/mosquitto**. Eliminare tutti i file nella sotto cartella conf.d, rieseguire Mosquitto e vedere se funziona. Se il problema persiste eliminare anche il file con estensione .conf nella cartella mosquitto.

2.3 Configurazione

Viene riportata la struttura della cartella contenente le configurazioni.

```
/ (/etc/mosquitto/)
├─ mosquitto.conf (configurazione di default, da non modificare)
├─ conf.d (cartella per la configurazione .conf)
├─ certs (cartella per i certificati del server)
└─ ca_certificates (cartella per i certificati della C.A.)
```

2.3.1 Configurazione iniziale

La configurazione del broker al momento dell'installazione è data dal file `mosquitto.conf`.

Se la cartella `conf.d` non contiene un file `.conf` viene caricata una configurazione di default propria del broker. È impostata come segue:

```
1 listener 1883 #abilita il broker in ascolto alla porta 1883
2 allow_anonymous true #abilita l'accesso al broker a chiunque
```

Va sottolineato che con questa configurazione l'accesso al broker è abilitato a chiunque e non ci sono restrizioni, perciò deve essere usato solo come ambiente di sviluppo.

2.3.2 Configurazione parametri

Può capitare che l'installazione non crei la struttura dei file e delle cartelle. Si deve procedere a creare nella cartella `/etc/mosquitto` almeno il file `mosquitto.conf` (si veda [A.1](#)) e la cartella `conf.d`.

Alcuni esempi di configurazione sono presenti in:

`/usr/share/doc/mosquitto/examples`

Per configurare il server in base alle proprie necessità, si deve creare un file con estensione `.conf` all'interno della cartella `conf.d`. Questa sarà la configurazione che verrà caricata quando si eseguirà il riavvio di Mosquitto.

Di seguito vengono riportati alcuni parametri rilevanti che sono stati usati. Si può fare riferimento alla documentazione ufficiale per la lista completa [\[2\]](#).

- **per_listener_settings [true|false] :**
Abilita la gestione di autenticazione in modo differente per ogni listener. In questo modo si può avere, ad esempio, localhost senza password, mentre per l'accesso esterno serve l'autenticazione. Di default è False.
- **listener (port-number) [ip address/host name/unix socket path] :**
Dice a Mosquitto su quale porta mettersi in ascolto per le connessioni in entrata. Si deve specificare il numero della porta (le porte consigliate sono 1883 senza SSL e 8883 con SSL). Inoltre si deve associare il listener a un indirizzo ip o ad un host o ad un socket. Si possono definire più listener, uno per ogni porta, ed è possibile avere impostazioni diverse per ogni listener. Una volta definito un listener, le successive configurazioni sono associate a quel listener.
- **allow_anonymous [true|false] :**
Determina se i client senza username sono abilitati alla connessione. Se è false si devono aggiungere configurazioni per il sistema di autenticazione. Viene associato ad un listener. Di default è False.
- **password_file (file-path) :**
Imposta il percorso del password file. Il file contiene la coppia utente:password per ogni riga, con la password salvata come hash. È un primo metodo di autenticazione e si associa ad un listener. Per ulteriori dettagli si veda [2.3.3](#).

- **certfile (file-path) :**

Imposta il percorso del certificato PEM del server. Viene usato insieme al parametro `keyfile` per abilitare la crittografia su TLS.

Nell'appendice [A.2](#) viene riportata la configurazione applicata al broker Mosquitto. Sono presenti altri parametri che vengono spiegati successivamente.

2.3.3 Metodi di autenticazione

Come descritto nella guida ufficiale [\[3\]](#), esistono 3 tipi di autenticazione:

- `unauthorized/anonymous access`: nessun controllo.
- `password files`: gestione basilare.
- `authentication plugins`: controllo più avanzato.

Inoltre si possono avere differenti metodi di autenticazione per ogni listener, configurando più parametri listener, [2.3.2](#).

Nel progetto è stata usata l'autenticazione tramite *password files* dato l'esiguo numero di account richiesti.

Password file

Questo è un metodo semplice per gestire gli utenti in un singolo file, tuttavia non è scalabile perché non permette la gestione puntuale di ogni client. Inoltre richiede l'esecuzione di comandi dal terminale, quindi è applicabile per casi non complessi come questo progetto.

Questo metodo richiede le seguenti caratteristiche:

- l'utente deve avere un nome univoco.
- la password deve essere immessa da terminale o in un file in chiaro. Una volta inserita viene salvato il suo hash.
- nel file di configurazione si devono applicare queste modifiche:
 - **allow_anonymous false**
Per disabilitare l'accesso anonimo e usare solo l'autenticazione utente e password.
 - **password_file /etc/mosquitto/nomefile**
Per dire al broker il percorso del file contenente gli utenti e le password validi per quel listener.

Per la creazione del file e degli utenti si possono eseguire i seguenti comandi:

- **mosquitto_passwd -c nomefile nomeutente**
Permette di creare nel percorso corrente un file con nome *nomefile* e un utente con nome *nomeutente*. Viene richiesto l'inserimento della relativa password. Questo metodo può essere usato per aggiungere un utente senza sovrascrivere il file se esiste già.

- **mosquitto_passwd -b nomefile nomeutente password**
Permette l'aggiunta di un utente e della relativa password al file indicato.
- **mosquitto_passwd -D nomefile nomeutente**
Permette l'eliminazione di un utente e della relativa password dal file indicato.
- **mosquitto_passwd -U nomefile**
Dato un file in qualsiasi formato contenete utente e password in chiaro, uno per ogni riga, esegue l'hash delle password. È un metodo più veloce per aggiungere utenti.

Terminate le modifiche si deve ricaricare la configurazione del broker. Per ricaricare la configurazione senza riavviare il servizio, si devono avere i diritti root:

```
1 sudo pkill -HUP -x mosquitto
```

Per l'elenco completo dei comandi si guardi la documentazione di Mosquitto [4]. La guida che spiega come implementare la connessione al broker da parte di un client con autenticazione è riportata nella sitografia finale [5].

2.3.4 Certificati SSL/TLS

Il broker Mosquitto supporta l'autenticazione e la connessione criptata tramite SSL. È possibile generare dei certificati self signed, come spiegato nella documentazione [6]. Tuttavia si possono incontrare programmi o script che non gradiscono questo tipo di certificati, quindi si deve optare per dei certificati di una vera CA, Certificate Authority.

Si è optato per la configurazione dei certificati tramite la CA LetsEncrypt. Di seguito vengono spiegati i vari passaggi di configurazione di *certbot* per LetsEncrypt, per poi soffermarsi sul funzionamento con il broker e i relativi problemi.

Vengono riportate man mano varie fonti, dato che non è stato possibile trovare un'unica documentazione per risolvere i problemi [7] [8] [9].

Certbot

Certbot è un tool che permette di gestire i certificati, le richieste e i rinnovi. Per la creazione del certificato si deve avere l'accesso root alla macchina. Inoltre, è necessario avere l'accesso alle impostazioni DNS del dominio associato alla macchina, oppure, in alternativa, un web server sulla macchina in cui si installa Mosquitto.

Si consiglia di seguire la documentazione per la corretta configurazione ad ogni passaggio [7].

La procedura prevede:

1. Installazione Certbot

Si esegue il comando:

```
1 sudo apt-get -y install certbot
```

2. Creazione di un certificato tramite DNS

Il certificato che si crea è di tipo wildcard, quindi se il dominio è mydomain.it

è valido anche per i sottodomini, così da poterlo usare per altri servizi. Il dominio DNS deve essere già associato alla macchina. Si esegue il comando:

```
1 certbot certonly --manual --preferred-challenge dns -d *.  
   mydomain.it
```

Si seguono le istruzioni a schermo, che prevedono l'aggiunta di un record DNS al dominio per validarlo. Al termine si ha generato dei certificati validi.

3. Rinnovo dei certificati

Il rinnovo si può eseguire:

- **Manualmente**

Si esegue il comando:

```
1 sudo certbot renew
```

- **Automaticamente**

Si deve modificare il crontab per eseguire il comando di auto rinnovo periodicamente [10]. Si esegue:

```
1 sudo crontab -e
```

Poi si aggiungono alla fine del file le seguenti righe:

```
1 # Auto-renew let's encrypt SSL certificates  
2 0 * * * * sudo certbot renew
```

4. Accesso ai certificati

Per accedere ai certificati si usano i symbolic link presenti nella cartella:

```
/etc/letsencrypt/live/mydomain.it/
```

Non si accede direttamente ai certificati perché questi hanno una scadenza, di solito di alcuni mesi. Questi link sono mantenuti da Certbot e puntano all'ultima versione dei certificati.

Funzionamento con Mosquitto

Mosquitto funziona sulla macchina con un utente che non ha i diritti root, quindi non ha accesso alle cartelle dei certificati. Tuttavia, seguendo le indicazioni fornite su Github dalla stessa Eclipse Mosquitto, si riesce a copiare i certificati permettendo l'accesso a Mosquitto [8]. La procedura prevede:

1. Copia automatica dei certificati

Lo script presente nell'appendice A.3 permette la copia e il cambio proprietario ad ogni rinnovo dei certificati. Lo script deve essere modificato nella riga 16, per specificare il dominio dei certificati interessati, e nella riga 18, per specificare la cartella di Mosquitto in cui si copieranno i certificati. Successivamente si copia lo script nella cartella (servono i diritti root):

```
/etc/letsencrypt/renewal-hooks/deploy/
```

Si rende eseguibile lo script:

```
1 sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/  
  mosquitto-copy.sh
```

Con questa procedura Mosquitto ha ad ogni rinnovo dei certificati l'ultima versione.

2. Copia manuale dei certificati

Probabilmente la prima volta si devono copiare manualmente i certificati. Si devono eseguire i seguenti comandi che fanno gli stessi passaggi dello script:

```
1 # copia dei file  
2 sudo cp /etc/letsencrypt/live/mydomain.it/fullchain.pem /  
  etc/mosquitto/certs/fullchain.pem  
3 sudo cp /etc/letsencrypt/live/mydomain.it/privkey.pem /  
  etc/mosquitto/certs/privkey.pem  
4 # imposta l'owner a Mosquitto  
5 sudo chown mosquitto: /etc/mosquitto/certs/fullchain.pem /  
  etc/mosquitto/certs/privkey.pem  
6 # imposta i diritti solo per Mosquitto  
7 sudo chmod 0600 /etc/mosquitto/certs/fullchain.pem /etc/  
  mosquitto/certs/privkey.pem  
8 # dice a Mosquitto di ricaricare certificati e  
  configurazione  
9 sudo pkill -HUP -x mosquitto
```

3. Modifica configurazione Mosquitto

Nella configurazione .conf di mosquitto si devono modificare 2 parametri impostando il path dei certificati:

```
1 certfile /etc/mosquitto/certs/fullchain.pem  
2 keyfile /etc/mosquitto/certs/privkey.pem
```

Terminate queste modifiche Mosquitto può ricevere connessioni con TLS sul listener configurato. I client che si connettono devono abilitare la connessione con TLS e opzionalmente la validazione del certificato.

2.4 Disinstallazione

2.4.1 Eliminazione della configurazione

Prima di procedere con la disinstallazione si consiglia di fare il backup delle configurazioni e dei certificati presenti nelle rispettive cartelle. Successivamente si consiglia di eliminare la cartella **/etc/mosquitto** per evitare futuri problemi. Nel caso in cui siano stati configurati listener con TLS, si deve eliminare lo script di copia automatica dei certificati.

2.4.2 Rimozione del broker

Per rimuovere la repository e l'installazione di Mosquitto si eseguono i seguenti comandi:

```
1 sudo apt-add-repository --remove ppa:mosquitto-dev/mosquitto -  
   ppa  
2 sudo apt-get update  
3 sudo apt-get remove mosquitto  
4 sudo apt-get autoremove
```

Capitolo 3

Client MQTT

3.1 Implementazione nel progetto

Ogni componente deve avere un client per comunicare tramite il broker. Nel progetto è stata scelta come client la libreria python "paho-mqtt", in quanto viene sviluppata dalla fondazione Eclipse, la stessa del broker Mosquitto. Inoltre si è deciso di sviluppare i client con il linguaggio di programmazione Python, perché è un linguaggio interpretato spesso usato per l'analisi di dati e permette di eseguire il debug in modo semplice. La seguente guida è stata pensata per sistemi Linux Ubuntu.

3.2 Installazione

3.2.1 Python

Per poter procedere è necessario installare Python nel proprio sistema operativo. Si consiglia di installare l'ultima versione di Python 3 supportata dal proprio sistema (si veda il sito ufficiale [\[11\]](#)). L'esecuzione di Python avviene dal terminale dei comandi o tramite script. Si noti che, in base a come sono impostate le variabili di ambiente, python può essere eseguito con il nome "python" o "python3". In base al sistema operativo in uso è possibile che Python sia già installato. Per verificare la versione installata si procede con il seguente comando:

```
1 python3 -V
2 # oppure
3 # python -V
```

Se non risulta installato, nei sistemi Ubuntu si procede all'installazione con:

```
1 sudo apt install python3-pip
```

Successivamente si consiglia di chiudere e riaprire il terminale per usare le variabili di ambiente aggiornate.

3.2.2 L'ambiente virtuale

L'uso di un ambiente virtuale (virtual environment) è un aspetto spesso sottovalutato quando si programma in Python. Un ambiente virtuale è isolato dall'installazione

base di Python e ha una cartella propria in cui si trovano le dipendenze e le configurazioni per l'esecuzione del proprio codice. In questo modo si possono avere diverse versioni di una libreria su diversi ambienti che sfruttano la stessa installazione Python. Molto spesso la funzionalità dell'ambiente virtuale viene installata insieme a Python, dato che dalla versione 3.3 di Python è stata introdotta nel modulo "venv" una versione basilare della libreria "virtualenv". Per sfruttare a pieno le funzionalità della virtualizzazione si può installare la libreria "virtualenv" [12], ma per questo progetto è più che sufficiente quella integrata in Python. Per verificare se la libreria è installata si procede con:

```
1 python3 -m venv
```

Se la libreria è già installata viene notificato un errore di mancanza di argomenti; in caso contrario si deve installare con:

```
1 sudo apt install python3-venv
```

Una volta terminata l'installazione si può procedere alla creazione di un ambiente, posizionandosi nella cartella in cui lo si vuole generare ed eseguendo:

```
1 python3 -m venv venv
```

L'ultimo parametro è quello che specifica il path in cui verrà creato l'ambiente. In questo caso si crea nella cartella corrente un ambiente chiamato "venv". Una volta creato, si deve dire a Python di cambiare l'ambiente che sta usando. Si procede con l'attivazione:

```
1 source venv/bin/activate
```

Questo comando richiama uno script all'interno dell'ambiente e attiva l'ambiente solo nel terminale corrente. Da questo punto in poi tutto quello che si esegue modifica solo questo ambiente virtuale. Si può procedere con l'installazione delle varie librerie e pacchetti.

Per disattivare un ambiente e ritornare all'ambiente base di Python si esegue:

```
1 deactivate
```

Per eliminare un ambiente è sufficiente eliminare la cartella dell'ambiente virtuale.

3.2.3 I requisiti

Spesso nei progetti Python si ha a disposizione un file denominato "requirements" (requisiti), in cui è presente la lista delle librerie e la versione usata dal progetto. Per l'installazione dei requisiti si esegue:

```
1 python3 -m pip install -r requirements.txt
```

In questo modo nell'ambiente virtuale sono installati tutti i componenti che il progetto richiedeva quando è stato sviluppato. Da questo punto in poi si è pronti a eseguire il progetto.

Per salvare la lista delle dipendenze del corrente ambiente si usa il comando:

```
1 python3 -m pip freeze > requirements.txt
```

3.2.4 La libreria paho-mqtt

Si procede a installare la libreria come descritto nella documentazione ufficiale [13], nell'ambiente in cui si vuole eseguire il codice, con il seguente comando:

```
1 python3 -m pip install paho-mqtt
```

Viene installata la libreria nell'ultima versione compatibile con le varie dipendenze. Ora è possibile creare dei client per interagire con il broker. Per rimuovere la libreria è sufficiente eseguire la disinstallazione:

```
1 python3 -m pip uninstall paho-mqtt
```

3.3 Le componenti del client

La libreria è formata principalmente da 3 moduli: client, publish e subscribe. Il modulo "client" è quello completo che contiene tutto, dalla connessione alla pubblicazione e l'iscrizione. I moduli "publish" e "subscribe" invece sono una versione semplificata della pubblicazione e dell'iscrizione, quindi non ci sono tutte le funzionalità per gestire la connessione.

Per il progetto è stato usato il modulo "client" per la creazione degli skeleton dei client così da non avere limitazioni per futuri sviluppi. Ci si è basati sulla documentazione [14], nello specifico quella per il client in cui sono descritte le varie funzionalità [15]. Sulla repository GitHub della libreria vengono forniti anche esempi di script già preimpostati [16], tuttavia si è preferito creare da zero gli script per soddisfare le necessità. Inoltre è stata molto utile la seguente guida che spiega in dettaglio passo a passo le fasi per l'invio e la ricezione da un topic [17].

3.3.1 Gestione della connessione

Per iniziare si deve creare un file con estensione .py (file python). Poi si deve importare il modulo "client" della libreria con la seguente riga:

```
1 import paho.mqtt.client as mqtt
```

Alla libreria viene associato il nome mqtt per facilità.

Istanza

Si procede con la creazione dell'istanza del client per la connessione, che viene associata a una variabile:

```
1 istanza_client = mqtt.Client(client_id=None, clean_session=True)
```

Alla funzione sono stati passati 2 parametri che sono necessari. Il parametro client_id è posto a None, dato che non ci interessa avere un id univoco per questa istanza, quindi la libreria alla connessione ne genera uno casuale. Inoltre è necessario anche il parametro clean_session posto a True per dire al broker che al termine della connessione deve rimuovere le informazioni collegate all'id client generato casualmente.

È stata applicata questa impostazione perché viene usato il protocollo MQTT versione 3, che si può specificare con il parametro `protocol=mqtt.MQTTv311`; mentre se si usa la versione 5 il parametro `clean_session` non è accettato e bisogna usarne un altro quando si avvia la connessione (vedere la documentazione per i dettagli). Ci sono altri parametri opzionali tra cui:

- `userdata`, per passare informazioni alle callbacks dell'istanza
- `transport`, per indicare il meccanismo di trasporto (di default è TCP)

Callbacks

Successivamente si possono impostare le callbacks sull'istanza, cioè le funzioni che vengono eseguite in base a un evento. Queste permettono di chiamare una funzione chiamabile e passare le informazioni necessarie per gestire quell'evento. Ne esistono numerose, ma quelle rilevanti sono le seguenti:

```
1 istanza_client.on_connect = funzione_alla_connessione
2 istanza_client.on_subscribe = funzione_alla_iscrizione
3 istanza_client.on_message = funzione_alla_ricezione
4 istanza_client.on_publish = funzione_alla_pubblicazione
5 istanza_client.on_log = funzione_per_i_log
```

In questo modo ad ogni evento è stata associata la rispettiva funzione chiamabile. A queste funzioni possono essere passati dei parametri (si consiglia di consultare la documentazione per la lista completa dato che cambiano in base alla versione del protocollo MQTT). Ogni funzione deve essere definita precedentemente con i parametri che si vogliono usare all'interno. Di seguito viene mostrato come si possono implementare le funzioni con il protocollo versione 3 senza spiegare i parametri.

Funzione alla connessione:

```
1 def funzione_alla_connessione(client, userdata, flags, rc):
2     print(client, userdata, flags, rc)
```

Permette di sapere se il broker ha risposto alla richiesta in modo positivo o se ci sono problemi.

Funzione all'iscrizione:

```
1 def funzione_alla_iscrizione(client, userdata, mid,
2     granted_qos):
3     print(client, userdata, mid, granted_qos)
```

Viene eseguita dopo che il broker ha risposto a una richiesta di iscrizione, quindi è possibile sapere se ci sono problemi.

Funzione alla ricezione:

```
1 def funzione_alla_ricezione(client, userdata, message):
2     print(client, userdata, message)
```

Con questa funzione all'arrivo del messaggio si possono eseguire delle elaborazioni e poi pubblicare un messaggio con i risultati.

Funzione alla pubblicazione:

```
1 def funzione_alla_connezione(client, userdata, mid):  
2     print(client, userdata, mid)
```

Viene eseguita dopo che viene trasmesso un messaggio al broker. In base al livello di QoS impostato nel messaggio, questa funzione può essere chiamata subito all'invio o dopo l'hand-shakes.

La funzione per i log:

```
1 def funzione_per_i_log(client, userdata, level, buf):  
2     print(client, userdata, level, buf)
```

Viene usata per il parametro buf che contiene informazioni utili in fase di debug e permette di vedere tutto quello che l'istanza sta facendo.

Come detto in precedenza, esistono molti tipi di callbacks in modo tale da gestire minuziosamente errori o eventi.

Connessione

Una volta definite tutte le callbacks necessarie si può procedere con l'avvio della connessione. Si esegue la funzione connect sull'istanza.

```
1 istanza_client.connect(host=localhost, port=1883, keepalive  
    =60)
```

La funzione richiede come parametro obbligatorio l'host, cioè l'indirizzo ip o il dominio a cui deve collegarsi. Se non si specifica altro la connessione viene aperta con la porta 1883, che è quella di default. Gli ulteriori parametri con il protocollo versione 3 sono port, per indicare una porta differente, e keepalive, per indicare l'intervallo massimo di secondi in assenza di comunicazioni. Il risultato della connessione viene gestito dalla callback on_connect vista precedentemente.

Mantenimento della connessione

Dopo che la connessione è stata avviata con successo è necessario dire all'istanza di rimanere attiva in un loop, altrimenti l'esecuzione del codice prosegue e quindi termina la connessione. Ci sono vari modi per mantenere la connessione, ma quello che viene usato frequentemente è il seguente:

```
1 istanza_client.loop_forever()
```

Questa funzione permette di mandare l'esecuzione in un loop e può essere utile per attendere l'arrivo dei messaggi. Ha come parametri opzionali timeout, i secondi di attesa per il traffico della rete prima di terminare, e retry_first_connection, per ritentare la connessione in caso di disconnessione. Nelle callback ci può essere anche la funzione di disconnessione e se questa viene chiamata il loop termina.

Disconnessione

Quando si vuole terminare la connessione si esegue la funzione disconnect.

```
1 istanza_client.disconnect()
```

Ci si è quindi scollegati dal broker e per sapere il risultato si usa la callback `on_disconnect`.

3.3.2 Invio di messaggi

L'invio di un messaggio si può avere solo dopo che la connessione è stata aperta. Si possono eseguire l'invio e la disconnessione separatamente oppure avviare l'istanza in un loop e in una callback eseguire l'invio. Si possono avere diversi invii su topic diversi con la stessa istanza client, tranne nei casi di particolari politiche di accesso del broker. La funzione di invio è la seguente:

```
1 istanza_client.publish(topic=nome, payload=messaggio, retain=True)
```

I parametri `topic` e `payload` sono necessari per l'invio. `Topic` indica su quale canale deve essere inviato il messaggio. `Payload` è il messaggio stesso, se è vuoto o `None` viene inviato un messaggio con lunghezza 0. I parametri opzionali per il protocollo versione 3 sono: `retain`, per dire al broker di mantenere quel messaggio su quel topic fino ad uno nuovo, e `qos`, per indicare il livello da usare per quality of service.

Il risultato dell'invio si ha nella callback `on_publish`.

3.3.3 Ricezione di messaggi

La ricezione di un messaggio si può avere solo dopo che la connessione è stata aperta. Si imposta il topic su cui si riceve, si avvia l'istanza in un loop e si riceve il messaggio nella callback `on_message`. Ci si può iscrivere su topic diversi con la stessa istanza client, tranne nei casi di particolari politiche di accesso del broker. La funzione di ricezione è la seguente:

```
1 istanza_client.subscribe(topic=nome)
```

Ha come parametro obbligatorio il nome del topic da cui si vogliono ricevere i messaggi. La funzione accetta il parametro `topic` in varie forme: stringa, tupla o lista di tuple. Nel progetto è stata usata la forma stringa, più che sufficiente in base alle necessità. La forma tupla integra dentro il parametro `qos`, mentre una lista di tuple serve principalmente per iscriversi con un solo comando a più topic. Come parametro opzionale c'è `qos`, che specifica il livello di qualità del servizio. Se non inizializzato viene posto a 0.

3.3.4 Autenticazione

Prima di avviare la connessione si può impostare l'autenticazione. In questo modo si fornisce al broker la username e la password. Il broker a cui ci si connette deve avere il protocollo MQTT versione 3.1 o superiore. L'impostazione si ha con:

```
1 istanza_client.username_pw_set(username, password)
```

Ha come parametri 2 stringhe:

- **username**: se è presente abilita l'autenticazione. Il valore dell'username non è relazionato con il `client_id` per la connessione.

- **password:** può essere assente anche se l'username è presente. La stringa viene codificata in UTF-8.

3.3.5 Connessione con TLS

Prima di avviare la connessione si può abilitare il supporto SSL/TLS, che permette di configurare la crittazione della comunicazione con il broker. Si deve aver configurato sul broker il certificato sulla porta su cui ci si conatterà. Per impostare la crittazione si usa la funzione:

```
1 istanza_client.tls_set(ca_certs=path_to_cert)
```

Si deve specificare il parametro `ca_certs` indicando il path al certificato della Certificate Authority. Se il file è nella stessa cartella dello script basta specificare il nome del file con l'estensione. In questo modo, quando si avvierà la connessione, il client verificherà il certificato del broker, allo stesso modo in cui il browser web verifica l'autenticità di un sito web. Altri parametri opzionali molto utili sono:

- **cert_reqs:** permette di specificare i requisiti che il client impone al broker. Se non specificato, il valore di default è `ssl.CERT_REQUIRED`, che impone al broker di fornire un certificato. Nel progetto non è stato specificato, dato che serviva la funzionalità di default.
- **tls_version:** indica la versione del protocollo SSL/TLS da usare. Se non specificato, il valore di default è 2, che corrisponde al TLS versione 1.2. Nel progetto è stata esplicitata la versione anche se è quella di default.

Infine ci si deve ricordare che la porta di connessione con TLS del broker di default è la numero 8883, ma potrebbe essere configurata diversamente.

3.4 Implementazione con altri script

Completata la creazione dello skeleton del client si può procedere con l'integrazione con altri script. Nell'appendice [B](#) viene fornito lo skeleton con il client configurato per iscriversi e pubblicare. Nel progetto gli script avevano le seguenti necessità:

- **Pubblicazione di dati a intervalli**

Lo script che veniva fornito leggeva un file in formato CSV, faceva il parsing di ogni riga e popolava un array con i messaggi da inviare in formato JSON. Poi lo script doveva entrare in un loop infinito in cui mandava al broker un messaggio alla volta fino alla fine, per poi ricominciare. I messaggi dovevano essere inviati a intervalli regolari, quindi si metteva in pausa l'esecuzione per poi inviare il messaggio successivo. Questo è il punto in cui è stato inserito il client che pubblica verso il broker. Viene chiamato passandogli il topic e il messaggio da inviare, quindi esegue la connessione, invia e si disconnette. Terminato il client, si ritorna allo script iniziale e si attende l'intervallo di tempo.

- **Ricezione di dati e pubblicazione dell'elaborazione**

In questo caso lo script fornito veniva chiamato con un JSON in input, eseguiva l'elaborazione e terminava con un risultato in JSON. Si è usato lo skeleton che iscrive e pubblica per ottenere e inviare i dati JSON. Alla base c'è il client che, una volta configurato per iscriversi a un topic, rimane in attesa per un messaggio. All'arrivo del messaggio in formato JSON si esegue la relativa callback, in cui il client chiama la funzione dello script da integrare e passa il messaggio. Terminata l'elaborazione si ritorna alla callback con il messaggio JSON da inviare e il client esegue una publish su un topic. A questo punto termina la callback e il client rimane in attesa.

3.5 Esecuzione automatica

Finora per eseguire gli script creati si usava il terminale, quindi si lanciavano manualmente. Tuttavia se questi script devono essere sempre funzionanti è consigliabile farli eseguire in modo automatico. Per risolvere questo problema nel progetto si è deciso di usare Supervisor, un programma per il monitoraggio e la gestione di processi in sistemi Linux. Si è preferito usare Supervisor al posto di Systemd in quanto è più semplice e intuitivo aggiungere, modificare e gestire un processo. Ci si è basati sulla guida [18] e sulla documentazione ufficiale [19].

3.5.1 Installazione

Supervisor può essere installato con più metodi: tramite Pip di Python o direttamente da Python oppure con il gestore di pacchetti, disponibile solo in alcune distribuzioni Linux. Nel nostro caso il sistema Ubuntu ha la possibilità di installazione tramite pacchetto, ma si devono avere i diritti root. Si procede con il comando:

```
1 sudo apt install supervisor
```

Viene installato il programma e viene messo nella lista di programmi eseguiti automaticamente all'avvio del sistema operativo. In questo modo è sempre avviato e pronto a gestire la lista dei processi assegnati.

3.5.2 Configurazione di un processo

Per aggiungere un processo alla lista di Supervisor, come prima cosa si deve creare un file con alcuni parametri. Si crea un file con il nome del processo e con estensione .conf. Un esempio di configurazione è la seguente:

```
1 [program:example]
2 directory=/home/utente/cartella
3 command=python3
4 user=utente
5 autostart=true
6 autorestart=true
7 stopasgroup=true
8 killasgroup=true
```

```
9 stderr_logfile=/home/utente/cartella/stderr.log
10 stdout_logfile=/home/utente/cartella/stdout.log
```

Il file di configurazione può essere diviso in più sezioni, che permettono di impostare finemente l'esecuzione (si veda la documentazione [20]). In questo esempio c'è solo una sezione, "program", che consente di impostare il programma da associare. Alcuni parametri riportati sono:

- **[program:example]**

Indica la sezione program ed è seguito dal nome del processo. La sezione può eseguire un solo processo o più processi in base a come viene configurata. Il nome indicato sarà quello visualizzato da Supervisor ed è necessario. Per la lista completa dei parametri si veda la documentazione [21].

- **directory=**

Imposta il path della cartella in cui verrà eseguito il comando, ovvero la cartella di lavoro. Di solito viene indicato il path completo, cioè assoluto, fino alla cartella. Non ha nessun valore di default, quindi se non impostato viene eseguito nella cartella di Supervisor.

- **command=**

Indica il comando che deve essere eseguito. Il programma da eseguire può essere specificato con il path assoluto o relativo. Di solito si ha dopo una sequenza di argomenti che vengono passati al programma. Il comando che si scrive è identico a quello che si scriverebbe da linea di comando per eseguire il programma. Non ha nessun valore di default ed è l'unico parametro che deve essere impostato obbligatoriamente.

- **user=**

Specifica con quale utente deve essere eseguito questo comando. Questo è possibile solo se Supervisor viene eseguito come root, come nel nostro caso. Non ha nessun valore di default, quindi se non impostato viene eseguito come root.

- **autostart=**

È un booleano che permette di impostare l'avvio automatico del programma dopo che Supervisor è stato avviato. Il valore di default è True.

- **autorestart=**

Serve per impostare il tipo di riavvio del processo se quest'ultimo termina con un errore. Può essere settato nei seguenti modi: false, per non riavviare; unexpected, se termina con un codice non definito; true, per riavviarlo a qualsiasi terminazione. Il valore di default è unexpected.

- **stopasgroup=**

È un booleano che permette di impostare l'invio del segnale stop a tutto il gruppo di processi. Se viene posto a True implica che il parametro killasgroup sia posto a True. Il valore di default è False.

- **killasgroup=**

È un booleano che permette di impostare l'invio del segnale stop a tutto il gruppo di processi e ai figli. Viene usato per terminare anche i processi figli del processo che si sta eseguendo. È obbligatoriamente posto a True se lo è anche stopasgroup. Il valore di default è False.

- **stderr_logfile=**

Imposta il path del file in cui scrivere lo standard error. Può essere impostato con: un path, per indicare un file specifico; AUTO, per far sì che Supervisor crei automaticamente un file; NONE, per non creare nessun log. Il valore di default è AUTO.

- **stdout_logfile=**

Imposta il path del file in cui scrivere lo standard output. Può essere impostato con: un path, per indicare un file specifico; AUTO, per far sì che Supervisor crei automaticamente un file; NONE, per non creare nessun log. Il valore di default è AUTO.

Una volta terminata l'impostazione, è necessario spostare la configurazione nella cartella delle configurazioni di Supervisor. Il path delle configurazioni è il seguente:

```
/etc/supervisor/conf.d/
```

Una volta posizionato qui il file si può procedere ad aggiungerlo alla lista di Supervisor.

3.5.3 Comandi della modalità interattiva

Per comandare Supervisor ci sono 2 modi: da linea di comando o tramite la modalità interattiva. In questo progetto è stata usata sempre la modalità interattiva, dato che l'esecuzione dei comandi è più intuitiva, mentre nella linea di comando è più facile commettere errori vista l'assenza dell'auto completamento. Per entrare nella modalità interattiva si esegue:

```
1 sudo supervisor
```

All'avvio viene mostrato lo stato di tutti i processi che gestisce Supervisor e a inizio riga si avrà "supervisor>". Da questo momento in poi si mandano solo comandi per Supervisor. Di seguito vengono spiegati i comandi usati durante il progetto (per la lista completa vedere la documentazione [22]).

Rilettura configurazioni

Il primo comando da eseguire quando sono state aggiunte delle configurazioni è il seguente:

```
1 reread
```

In questo modo vengono lette le configurazioni e si può verificare se sono corrette. Questo comando esegue solo una lettura senza aggiungere le configurazioni alla lista di Supervisor.

Aggiornamento configurazioni

Per aggiungere una configurazione alla lista che gestisce Supervisor si esegue:

```
1 update
```

Le configurazioni vengono rilette e verificate, poi viene aggiornata la lista delle configurazioni. Successivamente vengono avviati i processi aggiunti e riavviati i processi modificati.

Stato

Per vedere lo stato di tutti i processi che gestisce supervisor:

```
1 status
```

Mostra per ogni riga un processo e le relative informazioni.

Avvio

Per avviare un processo specifico si esegue il comando seguito dal nome:

```
1 start nome
```

Per avviare tutti i processi della lista:

```
1 start all
```

Fermare

Per fermare un processo specifico si esegue il comando seguito dal nome:

```
1 stop nome
```

Per fermare tutti i processi della lista:

```
1 start all
```

Uscita

Per uscire dalla modalità interattiva di Supervisor si lancia:

```
1 quit
```

Si ritorna alla linea di comando normale.

Questi comandi possono essere eseguiti anche al di fuori della modalità interattiva. Prima di ogni comando basta aggiungere "sudo supervisor".

Al termine di queste operazioni gli script sono configurati per l'esecuzione automatica e l'implementazione dei client con gli script è completata.

Capitolo 4

Flask web server

4.1 Implementazione nel progetto

Flask è un framework web scritto in Python per la creazione di siti e interfacce web ed è una applicazione WSGI (Web Server Gateway Interface). Viene considerato un micro-framework perché ha un nucleo semplice e minimale, ma è estendibile con librerie di terze parti. In questo modo si possono aggiungere le funzionalità comuni presenti in altri framework più complessi. La sua modularità fa sì che possa essere impiegato sia da principianti che da professionisti. In questo progetto Flask è stato usato per l'esecuzione di script python da browser per facilitare la generazione dei modelli utili per la virtualizzazione di sensori e modelli di crescita. Di seguito viene spiegato come funziona il framework e come è stata strutturata l'applicazione web.

4.2 Installazione

L'installazione può essere eseguita su qualsiasi sistema operativo in cui può essere installato Python. Nella guida ufficiale [23] viene specificato che la versione 3 di Flask è compatibile con Python a partire dalla versione 3.8, tuttavia è possibile usare senza problemi la versione precedente per Python meno recenti. Si seguono gli stessi passaggi della guida, quindi come prima cosa si eseguono gli stessi passaggi visti in 3.2.1 per installare Python. Poi si procede alla creazione di un ambiente virtuale, visto in precedenza 3.2.2. Infine si può installare Flask con il seguente comando, a meno che non sia presente già nei requisiti 3.2.3:

```
1 python3 -m pip install flask
```

Da questo punto in poi si è pronti per creare il web server.

4.3 Strutturazione del web server

Per proseguire è necessario avere delle conoscenze minime di HTML, CSS e JavaScript, che sono alla base di ogni sito web. La spiegazione è strutturata basandosi sul flusso di esecuzione del codice e sul risultato finale del progetto. Comunque all'inizio verrà spiegata la struttura base di un'applicazione.

4.3.1 La base del flusso di esecuzione

Per poter capire perché i file nel progetto hanno una certa organizzazione, si parte creando un'applicazione semplice. Innanzitutto ci si deve posizionare nella cartella padre che contiene l'ambiente virtuale creato. Poi si deve creare la seguente struttura di file e cartelle:

```
/      (progetto)
├── venv      (cartella che contiene l'ambiente virtuale creato)
├── run_flask.py      (script per avviare Flask)
├── flaskr      (cartella contenente il codice per Flask)
│   ├── __init__.py      (script di inizializzazione di Flask)
│   ├── conf.py      (file per la configurazione di Flask)
│   ├── routes.py      (file per le rotte di Flask)
│   ├── model.py      (file per i modelli di dati di Flask)
│   ├── templates      (cartella per i template HTML di Flask)
│   │   ├── index.html      (template HTML principale)
│   ├── static      (cartella per file CSS e Javascript di Flask)
│   │   ├── index.css      (file CSS per il template index.html)
│   │   └── index.js      (file Javascript per il template index.html)
```

Il file da cui tutto parte è `flask_run.py`, che una volta chiamato provvede ad avviare l'applicazione. Di seguito il codice:

```
1 from flaskr import app, db
2
3 if __name__ == '__main__':
4     with app.app_context():
5         db.create_all()
6     app.run(host='0.0.0.0', debug=True)
7 else:
8     with app.app_context():
9         db.create_all()
```

La prima riga importa due componenti dal file `__init__.py`: `app`, che è l'applicazione in sé, e `db`, che è il database.

Poi è presente un semplice costrutto condizionale `if else` che permette di avviare il web server in modalità sviluppo o produzione. Quando il file viene chiamato tramite Python, viene eseguito l'`if` che avvia il server WSGI di sviluppo interno alla libreria Flask. Il comando è il seguente:

```
1 python3 flask_run.py
```

Se il file viene eseguito da un'altra applicazione come Gunicorn, il server WSGI usato nel progetto [4.5](#), l'`else` viene eseguito avviando il framework in modalità produzione. La differenza tra sviluppo e produzione è rilevante: nella modalità sviluppo si hanno tutti gli strumenti per il debug, mentre in produzione il codice è ottimizzato per fornire solo pagine web. In entrambi i casi si prende l'istanza dell'applicazione e la si avvia nel suo contesto, poi si crea il database se non è già esistente. Quello che cambia è che nell'`if` l'applicazione deve essere eseguita con `"app.run"`, mentre nell'`else` ci pensa il server WSGI.

La base della inizializzazione avviene in `__init__.py`, di seguito riportato.

```
1 from flask import Flask
2 from flaskr.config import Config
3 from flask_sqlalchemy import SQLAlchemy
4 from flask_login import LoginManager
5
6 app = Flask(__name__, static_folder='static', template_folder
7             ='templates')
8 app.config.from_object(Config)
9
10 db = SQLAlchemy(app)
11
12 login_manager = LoginManager(app)
13 login_manager.login_view = 'access_bp.login'
14 login_manager.login_message_category = 'warning'
15
16 from flaskr import routes
```

Le prime righe importano dei moduli. Quella più rilevante è la prima, in cui viene importata la libreria Flask, seguita dalla classe Config del file config.py, in cui sono presenti alcune nostre configurazioni. Successivamente si importano altre 2 librerie: flask_sqlalchemy e flask_login. Queste ultime sono da installare nell'ambiente perché la prima serve per la gestione del database e la seconda per le utenze. In questo flusso non verranno usate, ma solo configurate.

Le due righe successive inizializzano l'istanza dell'applicazione. La prima crea l'istanza chiamando la libreria Flask e passando come parametri il nome dell'istanza, il nome della cartella che contiene i file statici e il nome della cartella che contiene i template. Il primo parametro è obbligatorio perché specifica il nome che di solito viene impostato a `__name__` come da documentazione. Gli ultimi parametri non sono obbligatori, infatti sono stati esplicitati solo i valori di default. Poi si passa alla configurazione Config. La riga successiva istanzia il database collegato all'applicazione. Le tre righe successive istanziano e impostano il gestore delle utenze. L'ultima riga è quella che importa le rotte presenti nel modulo "routes".

La configurazione presente in config.py è:

```
1 class Config:
2     SECRET_KEY = 'c3635ab8314r6d199d623a135x3te340 '
3     SQLALCHEMY_DATABASE_URI = 'sqlite:///site.db '
```

Sono state create due costanti. La prima è una chiave segreta che serve anche per l'autenticità dei cookie della sessione. Questa chiave non deve essere rivelata in pubblico, ma tenuta segreta nel server. La seconda è l'URL per la connessione al database. In questo caso si usa il database sqlite messo a disposizione dalla libreria e non uno esterno. Il database viene creato in automatico all'interno della cartella flaskr.

Nel file `models.py` viene definito come sono strutturati l'utente e il database e come deve essere gestita la login. Di seguito il file:

```
1 from flaskr import db, login_manager
2 from flask_login import UserMixin
3
4 @login_manager.user_loader
5 def load_user(user_id):
6     return User.query.get(int(user_id))
7
8 class User(db.Model, UserMixin):
9     id: int = db.Column(db.Integer, primary_key = True)
10    email: str = db.Column(db.String(20), nullable=False,
11                           unique=True)
12    password: str = db.Column(db.String(80), nullable=False)
13
14    def __repr__(self):
15        return f"User('{self.email}')
```

La prima riga importa dal file `__init__.py` le istanze `db` e `login_manager`. La seconda importa dalla libreria `flask_login` una classe che implementa dei metodi di default necessari per gestire la login. Successivamente viene definita la funzione che ottiene dal database un utente specifico. Questa funzione ha come decoratore `@login_manager.user_loader` e viene usata per associare alla callback `user_loader` la funzione. La query che viene eseguita si basa sulla chiave primaria di quella tabella. Infine viene definita la classe `Utente` che prende i valori dal database e a cui vengono associati i metodi di default per la login. All'interno della classe vengono definite le proprietà delle colonne del database: il tipo di dato, la chiave primaria, se è accettato il valore nullo e se il valore deve essere univoco. Viene definita una funzione che ritorna il valore che rappresenta l'utente.

La gestione delle rotte si trova in `routes.py` ed è la seguente:

```
1 from flaskr import app
2 from flask import redirect, render_template
3
4 @app.route('/')
5 def index():
6     return redirect('/main')
7
8 @app.route('/main', methods=['GET', 'POST'])
9 def login():
10    return render_template('index.html')
```

Qui vengono definite le rotte che servono per associare un URL ad una funzione che ritorna una pagina HTML. La prima riga importa l'istanza dell'applicazione da `__init__.py`. La seconda riga importa dalla libreria `Flask` 2 funzioni spiegate successivamente. Poi si ha la prima rotta, la funzione `index`. Questa ha come decoratore `@app.route('/')` che associa all'URL `/` questa funzione da eseguire. Quindi, se si digita l'URL `localhost:5000` o `localhost:5000/` sul browser, il server risponde con

questa funzione. La funzione ritorna una redirect, cioè una funzione che rimanda ad un altro URL, in questo caso `'/main'`. La seconda rotta è appunto `'/main'`. Al decoratore della funzione `login` sono stati specificati solo i metodi HTTP che accetta a scopo esemplificativo. La funzione ritorna una renderizzazione del template chiamato `'index.html'`. Quello che fa è andare nella cartella `template`, prendere il file corrispondente, elaborarlo se necessario e ritornarlo alla richiesta HTTP.

Il codice `index.html` è il seguente e deve essere elaborato da Flask prima di essere mandato al browser.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1.0" />
8     <title>Models Calculator</title>
9     <link
10      rel="stylesheet"
11      type="text/css"
12      href="{{ url_for('static', filename='index.css') }}"
13    />
14     <script
15      type="text/javascript"
16      src="{{ url_for('static', filename='index.js') }}"
17    ></script>
18   </head>
19   <body>
20     <div class="main">
21       <h2>Main page Works</h2>
22       <button onclick="onChange()">Show / hide some text</
23         button>
24       <div id="text-hidden" hidden>You can hide me</div>
25     </div>
26   </body>
27 </html>
```

Nell'head dell'HTML è presente un riferimento a un foglio di stile e a uno script. Entrambi hanno un URL molto particolare che viene interpretato da Flask. La funzione `url_for`, contenuta all'interno di due parentesi graffe, dice a Flask di creare l'URL per quella risorsa statica, presente nella cartella `static`, con il path indicato dopo. Una volta che Flask ha creato gli URL, questi vengono messi al posto di quanto contenuto nelle due parentesi graffe. Si ha quindi l'HTML elaborato e pronto per essere mandato al browser. Questo procedimento viene eseguito ogni volta che viene chiamata la rotta che renderizza questo HTML. I file CSS e Javascript non vengono riportati in quanto non sono rilevanti per comprendere il flusso.

Se si procede con l'esecuzione dell'applicazione con il [comando visto prima](#), il server

di sviluppo viene eseguito ed è raggiungibile all'URL localhost:5000 che ritorna la pagina index.html. Questo è il flusso che avviene dall'avvio dell'applicazione fino alla richiesta di una rotta.

4.3.2 Blueprint

Nell'applicazione del progetto è stato usato il concetto di Blueprint che permette di organizzare in gruppi pagine HTML e il relativo codice. In questo modo si ha il vantaggio di distinguere logicamente il codice in base alla sua funzione nell'applicazione, rendendo più facile la manutenzione e l'aggiunta di funzionalità. Qui viene spiegato l'uso ai fini del progetto (si consiglia di leggere la documentazione per tutti i dettagli [24]). Di seguito viene riportata la struttura della cartella del progetto Flask.

```
flaskr/
├── routes          (cartella contenente tutte le rotte)
│   ├── access      (cartella per le rotte di accesso)
│   │   ├── __init__.py
│   │   ├── forms.py
│   │   ├── routes.py      (blueprint delle rotte di accesso)
│   │   ├── templates      (cartella dei template per l'accesso)
│   │   │   ├── mainAccess.html
│   │   │   └── pages
│   │   │       ├── login.html
│   │   │       ├── register.html
│   │   │       └── recover.html
│   ├── api          (cartella per le rotte di API)
│   │   ├── __init__.py
│   │   ├── routes.py      (blueprint delle rotte di API)
│   │   └── templates
│   ├── interface    (cartella per le rotte dell'interfaccia)
│   │   ├── __init__.py
│   │   ├── forms.py
│   │   ├── routes.py      (blueprint delle rotte dell'interfaccia)
│   │   ├── templates      (cartella dei template per l'interfaccia)
│   │   │   ├── mainInterface.html
│   │   │   └── pages
│   │   │       ├── about.html
│   │   │       ├── account.html
│   │   │       ├── dashboard.html
│   │   │       ├── file_manager.html
│   │   │       └── regressionCalculator.html
│   └── root_errors   (cartella per le rotte di errore)
│       ├── __init__.py
│       ├── routes.py      (blueprint delle rotte di errore)
│       ├── templates      (cartella dei template per gli errori)
│       │   ├── mainErrors.html
│       │   └── pages
```


- 404.html
 - general.html
- static (cartella di default dei file statici)
 - A-Logo_Univr_Dip_Informatica_2016-02.png
 - favicon.ico
 - access (cartella dei file statici usati nell'accesso)
 - mainAccess.css
 - api (cartella dei file statici usati nell'API)
 - bootstrap (cartella della libreria Bootstrap)
 - bootstrap-4.3.1.bundle.min.js
 - bootstrap-4.3.1.bundle.min.js.map
 - bootstrap-4.3.1.min.css
 - bootstrap-4.3.1.min.css.map
 - bootstrap-4.3.1.min.js
 - bootstrap-4.3.1.min.js.map
 - jquery-3.3.1.slim.min.js
 - popper-1.16.1.min.js
 - popper-1.16.1.min.js.map
 - interface (cartella dei file statici usati nell'interfaccia)
 - css
 - about.css
 - account.css
 - dashboard.css
 - fileManager.css
 - mainInterface.css
 - regressionCalculator.css
 - image
 - Volantino-CREA.jpg
 - Volantino-data-science.jpg
 - Volantino-generale.jpg
 - javascript
 - account.js
 - fileManager.js
 - mainInterface.js
 - regressionCalculator.js
 - root_errors
- templates (cartella di default dei template)
- utility (cartella del codice di supporto all'applicazione)
 - file_system.py
 - run_scripts.py
 - smtp_client.py
 - smtp_server.py
- __init__.py
- config.py
- models.py
- site.db
- views.py (file in cui vengono importati i blueprint)

L'organizzazione dei file è molto differente da quella del flusso iniziale. Si può notare che non esiste più il file `routes.py` dentro alla cartella `flaskr`, ma al suo posto si ha il file `views.py`. La cartella `templates` è stata svuotata, mentre la cartella `statics` ha acquisito nuovi file. La cosa più rilevante è che adesso esistono più file `routes.py` nelle sottocartelle della cartella `routes` e sono accompagnati da una cartella `templates`. Con il concetto di Blueprint si ha avuto la possibilità di separare le rotte e il relativo codice in base al loro ruolo. Ad esempio nell'applicazione ci sono rotte che vengono usate solo per accedere, quindi sono state raggruppate sotto la cartella `access`. Lo stesso vale anche per le chiamate API che riceve il server web, la gestione degli errori e tutta l'interfaccia che si ha dopo l'accesso. Quindi si hanno 4 Blueprint nella cartella `routes`. Un altro vantaggio è che ogni Blueprint permette di definire una propria cartella `templates` e `statics` oltre a quelle di default. Il codice HTML è stato suddiviso nello stesso modo, mentre la suddivisione dei file statici si è preferito effettuarla nella cartella `statics` di default. Non sono state create le rispettive cartelle `statics` perché Flask, nel momento in cui crea l'URL per il file statico contenuto nel Blueprint, rivela tutta l'organizzazione del codice. Infatti l'URL per il file `mainAccess.css` nel Blueprint sarebbe stato

```
localhost:5000/routes/access/statics/mainAccess.css
```

mentre, dato che i file statici sono rimasti nella cartella `static` di default, l'URL è

```
localhost:5000/statics/access/mainAccess.css
```

Facendo così si espone una sola cartella per le richieste di file statici e si separa fisicamente il codice eseguito da Python alla richiesta di un URL, la cartella `routes`, da quello che è una risorsa, la cartella `statics`.

Si procede spiegando come cambia il flusso con questo nuovo concetto. Si parte dal file `views.py` che sostituisce `routes.py` del flusso iniziale.

```
1 from flaskr import app
2
3 from flaskr.routes.root_errors.routes import root_errors_bp
4 from flaskr.routes.access.routes import access_bp
5 from flaskr.routes.interface.routes import interface_bp
6 from flaskr.routes.api.routes import api_bp
7 from flaskr.utility.file_system import FileSystem
8
9 file_system = FileSystem()
10
11 app.register_blueprint(root_errors_bp)
12
13 app.register_blueprint(access_bp)
14
15 app.register_blueprint(interface_bp)
16
17 app.register_blueprint(api_bp, url_prefix='/api')
```

Questo file viene importato da `__init__.py` cambiando la riga 15 e mettendo views al posto di routes. Il codice importa l'istanza dell'applicazione, poi i 4 Blueprint e infine il modulo che gestisce i file che verranno creati quando si userà l'interfaccia. Si crea l'istanza che gestisce i file. Poi si hanno 4 righe in cui si registrano nell'istanza dell'applicazione i Blueprint. Ogni Blueprint ha un nome che lo identifica e, nella fase della registrazione, le rotte che contiene vengono associate all'applicazione in modo che venga a conoscenza della loro esistenza. Nell'ultima registrazione si specifica il parametro `url_prefix` che permette di anteporre un segmento di URL alle rotte appartenenti a quel Blueprint.

Per comprendere come deve essere definito un blueprint si prende come esempio l'implementazione con le rotte di accesso. Per gli altri blueprint del progetto l'implementazione è molto simile, ma ovviamente cambiano gli URL e le funzioni associate. Il file `routes.py` delle rotte di accesso è il seguente:

```
1 from flask import Blueprint
2 from flask import render_template, flash, redirect, url_for,
   request
3 from flask_login import login_user, current_user, logout_user
4
5 from flaskr.models import User
6 from flaskr.routes.access.forms import LoginForm,
   RegistrationForm, RecoverPasswordForm
7 from flaskr import db, bcrypt
8
9 from flaskr.utility.file_system import FileSystem
10 from flaskr.utility.smtp_client import email_Sender
11
12 access_bp = Blueprint(
13     'access_bp',
14     __name__,
15     template_folder='templates',
16     static_folder='static'
17 )
18
19 file_system = FileSystem()
20
21 @access_bp.route('/logout')
22 def logout():
23     logout_user()
24     return redirect(url_for('access_bp.login'))
25
26 @access_bp.route('/login', methods=['GET', 'POST'])
27 def login():
28     if current_user.is_authenticated:
29         return redirect(url_for('interface_bp.dashboard'))
30     ....
31     ....
32     return render_template('pages/login.html', title='Login',
```

```

        form=form)
33
34 @access_bp.route('/register', methods=['GET', 'POST'])
35 def register():
36     ....
37     ....
38     return render_template('pages/register.html', title='
        Registrazione', form=form)
39
40 @access_bp.route('/recover_password', methods=['GET', 'POST'])
41 def recover_password():
42     ....
43     ....
44     return render_template('pages/recover.html', title='
        Recupero', form=form)

```

Il contenuto delle funzioni è stato omissso per motivi di spazio. L'import rilevante è il primo, Blueprint, che è quello che permette di eseguire questa separazione di codice. Successivamente ci sono i vari import per la renderizzazione, gestione dell'utente, i form, l'istanza del database, il modulo per la gestione dei file e l'invio di email. Si ha poi la definizione della variabile **access_bp** come istanza Blueprint. Vengono definiti il nome del Blueprint, che corrisponde al nome della variabile, la cartella dei template e la cartella dei file statici. Queste due cartelle si trovano in due path diversi: la cartella dei templates si riferisce a quella presente nello stesso path del file routes.py, mentre l'altra si riferisce a quella di default presente all'interno della cartella flaskr. Si ha questa differenza di riferimento perché la prima serve a flask per sapere dove sono i template del Blueprint, mentre la seconda serve per creare l'URL delle risorse del Blueprint. La cartella dei file statici ha lo stesso nome di quella di default perché l'URL che viene generato si basa su questo valore e sul prefisso associato al Blueprint, come quello presente alla riga 17 del [file views.py](#). L'URL generato coincide con la cartella di default dei file statici dato che il prefisso è stato aggiunto solo alle API. Nel file la variabile access_bp viene trattata come se fosse la @app del file routes.py del [flusso iniziale](#). Infatti i decorator che associano l'URL a una funzione hanno le stesse funzionalità ed è questo il vantaggio di usare i Blueprint separando il codice. Tuttavia questa separazione non impedisce il passaggio tra Blueprint diversi. Nel codice riportato sono state lasciate alcune righe per evidenziare questa interoperabilità. Nella riga 24 il reindirizzamento permette di chiamare un metodo all'interno del Blueprint stesso. Mentre nella riga 29 il reindirizzamento è a un metodo appartenente ad un altro Blueprint. Questo tipo di riferimento è possibile anche all'interno dei template HTML.

4.3.3 Template

Finora si è fatto riferimento al codice HTML sempre con la parola template, perché non è la stessa cosa della pagina finale che arriva al browser. Come visto anche nel flusso iniziale, il codice HTML deve essere elaborato prima da Flask per poi essere inviato come risposta. Per spiegare questo concetto e come è stato sfruttato vengono riportate alcune parti di template (riferirsi alla documentazione ufficiale

per i dettagli [25]). Si prende in considerazione la creazione della pagina dashboard. La sua creazione richiede il file mainInterface.html e il file dashboard.html presenti nella cartella templates del Blueprint dell'interfaccia. Viene riportata una parte di mainInterface.html:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-
      scale=1.0" />
7     <link rel="stylesheet" type="text/css"
8       href="{ { url_for('static', filename='bootstrap/bootstrap
      -4.3.1.min.css') } }"/>
9     <link rel="stylesheet" type="text/css"
10      href="{ { url_for('interface_bp.static', filename='
      interface/css/mainInterface.css') } }"/>
11     {% block css %}
12     {% endblock css %}
13     <script type="text/javascript"
14      src="{ { url_for('interface_bp.static', filename='interface/
      javascript/mainInterface.js') } }"
15     ></script>
16     <script type="text/javascript"
17      src="{ { url_for('static', filename='bootstrap/popper
      -1.16.1.min.js') } }"
18     ></script>
19     {% block scripts %}
20     {% endblock scripts %}
21     <title>Models Calculator</title>
22   </head>
23   <body>
24     <nav class="navbar navbar-expand-sm navbar-light main-
      navbar">
25       ...
26       ...
27     </nav>
28     {% block content %}
29     {% endblock %}
30   </body>
31 </html>
```

Viene riportata una parte di dashboard.html:

```
1 {% extends "mainInterface.html" %}
2
3 {% block css %}
4 <link
5 rel="stylesheet"
6 type="text/css"
```

```

7 href="{{ url_for('interface_bp.static', filename='interface/
  css/dashboard.css') }}"
8 />
9 {% endblock css %}
10
11 {% block scripts %}
12 {% endblock scripts %}
13
14 {% block content %}
15
16 <div>
17     ...
18     ...
19 </div>
20
21 {% endblock content %}

```

Si può subito notare che `dashboard.html` non contiene tutti i tag per la visualizzazione corretta di codice HTML. Inoltre in entrambi i file sono presenti i delimitatori speciali `{{ }}` e `{% %}`. Il primo delimitatore serve per contenere delle espressioni che verranno interpretate da Flask e verranno sostituite con il risultato. Come anticipato alla [fine del flusso iniziale](#), quando Flask trova la funzione `url_for()` all'interno di questi delimitatori sa che deve creare l'URL per quella risorsa. Con quella funzione è possibile generare un URL per qualsiasi risorsa statica che appartenga a un Blueprint o meno, come si vede nelle righe 8, 10, 12, 17 di `mainInterface.html` e nella riga 7 di `dashboard.html`. Inoltre questi delimitatori servono per passare variabili al template quando questo viene renderizzato. Ad esempio nell'ultima riga di ogni funzione delle [rotte di accesso](#) vengono passate altre variabili oltre al template. Infatti se si va nel file `login.html` si vede che al suo interno la variabile `form` viene usata molto spesso e serve per dire a Flask come deve strutturare il form di login. Il secondo tipo di delimitatori serve per indicare la dichiarazione di un flusso di controllo che deve eseguire Flask. Nei due file riportati sopra, questi delimitatori servono principalmente per comporre la pagina che verrà mandata al browser. Il template `dashboard.html` alla prima riga comunica a Flask che estende un altro template `mainInterface.html`. Successivamente vengono definiti dei blocchi di codice che devono essere inseriti nel template che estende e vengono identificati dal nome che segue `block` e `endblock`. Quindi per creare la pagina finale si prende `mainInterface.html` e, nei punti in cui è presente la stessa dichiarazione di un blocco, viene aggiunto il codice proveniente da `dashboard.html`. Un altro uso è quello con i costrutti `if` e `for`, che permette di mostrare codice in base a certe condizioni o valori che vengono passati come variabili. In questo modo si possono generare pagine differenti avendo un unico template. Nel progetto questo tipo di uso è frequente nei template in cui ci sono i form, come in `login.html`, `register.html`, `recover.html` e `regressionCalculator.html`. Quindi questi due costrutti evitano di dover scrivere pagine HTML statiche per ogni rotta e permettono a Flask di generare pagine HTML dinamicamente a partire da dei template.

4.3.4 I moduli di supporto

Nella cartella utility all'interno di flaskr [4.3.2](#) sono presenti i moduli di supporto all'applicazione. Di seguito vengono descritti brevemente.

Gestione file

La gestione dei file caricati dall'interfaccia e creati dagli script viene fatta dal modulo `file_system.py`. Questo modulo viene sempre caricato all'avvio dell'applicazione. Alla sua istanziazione controlla che nella cartella padre di flaskr sia presente la cartella `dir_of_models`, altrimenti la crea. Questa cartella contiene tutti i dati caricati e i risultati delle elaborazioni. All'interno è presente una cartella per ogni utente registrato e viene usato il numero di iscrizione come nome della cartella. Alla registrazione viene creata la cartella utente e al suo interno vengono create le cartelle `.uploads` e `.tmp_models_dir`. La cartella `.uploads` è quella in cui vengono messi i file caricati dall'interfaccia web. Invece la cartella `.tmp_models_dir` serve agli script come punto in cui salvare i risultati delle computazioni. Quando l'utente esegue la creazione dei modelli, il gestore file esegue varie azioni in background. Per prima cosa mette i file caricati all'interno della cartella `.uploads`. Una volta che lo script di creazione è terminato, crea la cartella che viene richiesta nel form, se non è già presente. Poi sposta i risultati dello script dalla cartella `.tmp_models_dir` in quella appena creata. Se invece l'utente è nell'interfaccia di gestione dei file questo modulo risponde alle richieste Javascript con un JSON contenente la struttura della cartella utente. Tutte le azioni richieste da questa interfaccia, come scaricare, rinominare ed eliminare, vengono gestite da questo modulo.

Esecuzione di script

Il modulo `run_scripts.py` permette di eseguire gli script richiesti dalla applicazione. Contiene due classi: `GenerateRegressionModel`, per la creazione dei modelli, e `DeleteFileTimer`, per la cancellazione degli zip. Entrambe le classi provvedono a impostare il comando Python con cui poi verrà eseguito il processo.

Invio email

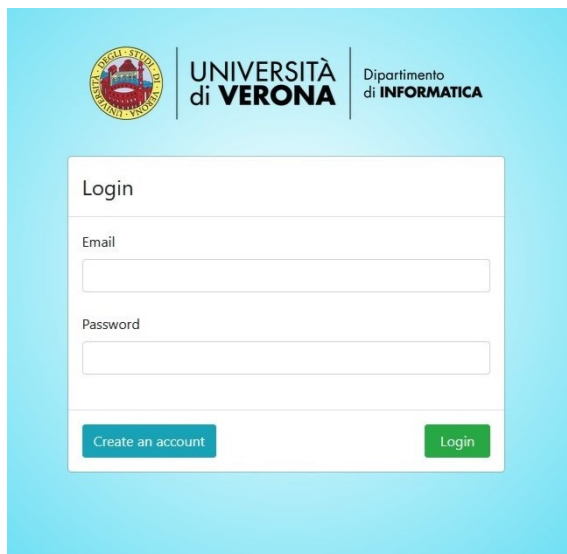
Il modulo `smtp_client.py` è stato predisposto per la funzionalità di invio email per il recupero password. La classe prevede la creazione di un messaggio, l'aggiunta di un allegato al messaggio e l'invio.

4.4 Funzionalità del progetto

Di seguito viene mostrata e spiegata brevemente ogni schermata che viene visualizzata nel progetto. Per l'interfaccia grafica è stata sfruttata la libreria Bootstrap 4.

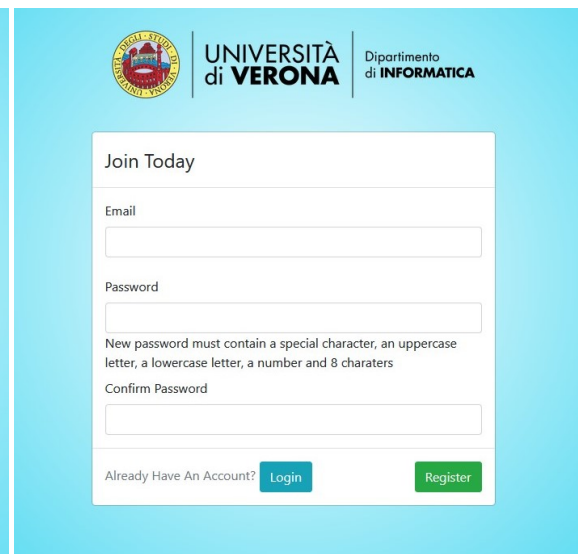
4.4.1 Accesso e autenticazione

Per accedere si ha un form per inserire email e password. Se l'inserimento dei dati non è corretto viene mostrato un errore. Nella stessa pagina si ha anche un pulsante che permette di andare nella pagina di creazione dell'account. Per creare l'account si devono inserire un'email e una password. È stato messo l'obbligo di ripetere l'inserimento della password per verificare che l'utente non abbia fatto errori di battitura. Quando si avvia la creazione dell'account, se l'email è stata già usata o la password non soddisfa i criteri richiesti, viene mostrato un errore, mentre, se i dati passano il controllo, viene mostrata la pagina di login con il messaggio di avvenuta creazione. Nella pagina di login è stato predisposto il pulsante di recupero password accanto al pulsante per creare l'utenza. Questa funzionalità e la relativa pagina non vengono mostrate in quanto mancano il server SMTP per l'invio dell'email e la logica per la generazione di una password random.



The screenshot shows the login interface. At the top, there is a header with the University of Verona logo and the text "UNIVERSITÀ di VERONA" and "Dipartimento di INFORMATICA". Below the header is a white box titled "Login". Inside this box, there are two input fields: "Email" and "Password". At the bottom of the box, there are two buttons: "Create an account" (blue) and "Login" (green).

Figura 4.1: Pagina di Login



The screenshot shows the registration interface. At the top, there is a header with the University of Verona logo and the text "UNIVERSITÀ di VERONA" and "Dipartimento di INFORMATICA". Below the header is a white box titled "Join Today". Inside this box, there are three input fields: "Email", "Password", and "Confirm Password". Below the "Password" field, there is a note: "New password must contain a special character, an uppercase letter, a lowercase letter, a number and 8 charaters". At the bottom of the box, there are two buttons: "Login" (blue) and "Register" (green). Above the "Login" button, there is a link: "Already Have An Account?".

Figura 4.2: Pagina di Registrazione

4.4.2 Interfaccia principale

La pagina mostra le opzioni disponibili nel web server. In alto è presente la barra per eseguire il logout. All'interno di ogni funzionalità viene mostrata nella barra la possibilità di ritornare nella dashboard.

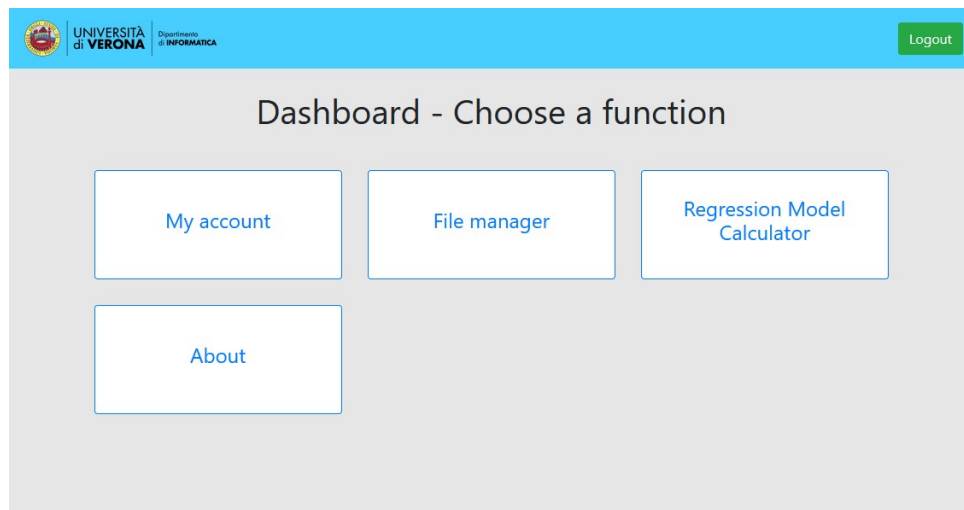


Figura 4.3: Pagina di Dashboard

4.4.3 Gestione account

La pagina permette di cambiare email e password dell'utente connesso. È possibile modificare solo l'email o la password o entrambi. Viene mostrata l'email corrente, ma non la password per ragioni di sicurezza. Quando si inseriscono dei valori si ha la possibilità di premere il tasto per aggiornare i dati. Se viene superato il controllo del form, viene aggiornato il database e poi viene mostrato il messaggio di avvenuta modifica.

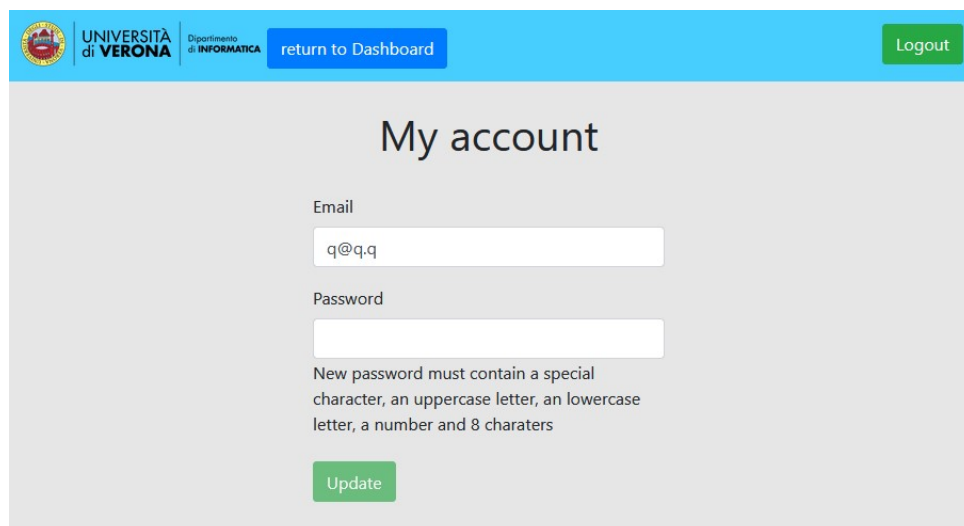


Figura 4.4: Pagina di gestione account

4.4.4 Gestione dei file

La pagina permette la gestione delle cartelle e dei file generati con la creazione dei modelli. L'utilizzo di Javascript ha permesso di implementare la visualizzazione modificando il DOM (Document Object Model), il quale rappresenta la pagina HTML visualizzata. Javascript esegue delle chiamate API al server e mostra la lista delle cartelle e dei file richiesti. Sono state implementate inoltre varie funzionalità, come la possibilità di scaricare un file singolo o l'intera cartella, di rinominare e di eliminare file e cartelle. Nel download è possibile scegliere se scaricare un singolo file oppure una cartella in formato zip. Le funzioni rinomina e cancella mostrano entrambe una finestra per confermare l'azione che si sta eseguendo.

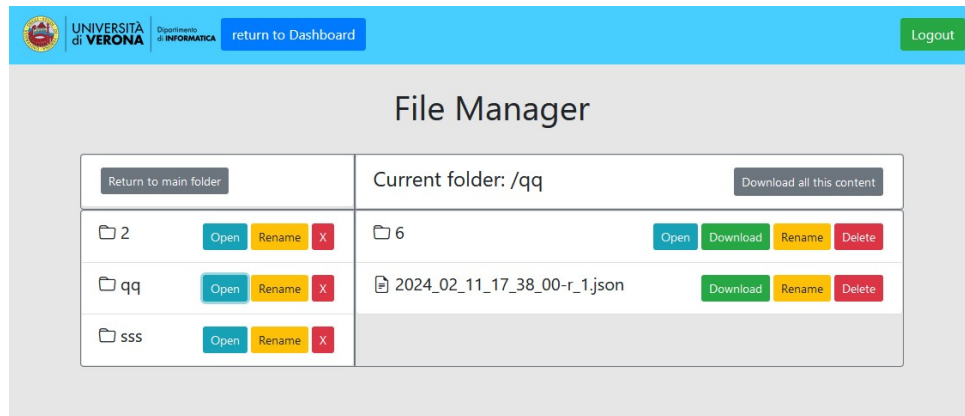


Figura 4.5: Pagina di gestione file

4.4.5 Creazione dei modelli

La pagina permette la creazione dei modelli di regressione lineare. La pagina è stata strutturata in due colonne. In quella di sinistra è presente il form per l'inserimento dei dati e dei file, mentre quella a destra mostra le fasi di elaborazione. Il form è strutturato nel seguente modo:

- **Model Output Directory:** permette di creare o scegliere una cartella in cui il server salverà i file dopo l'esecuzione dello script. La cartella è accessibile tramite la pagina di gestione file.
- **Model Result Name:** permette di specificare quale sarà il nome dei file ottenuti dallo script. È un valore opzionale perché in automatico vengono messi sempre come prefisso la data e l'ora corrente.
- **Model Input:** permette di caricare un file in formato CSV contenente i dati storici relativi al sensore.
- **Model expected output:** permette di caricare uno o più file in formato CSV che rappresentano la variabile di risposta da prevedere.
- **Windows:** è un valore espresso come numero intero non negativo, che stabilirà quanti istanti temporali precedenti verranno considerati durante il processo di creazione del modello.

- **Test:** permette di scegliere se eseguire il test del modello. In caso affermativo, i dati verranno divisi in addestramento (80%) e test (20%) per la verifica della precisione.

Una volta terminata la compilazione del form, se i campi necessari sono stati completati viene abilitato il pulsante per inviare i dati. Se premuto, i dati vengono inviati e viene verificato se soddisfano i requisiti. Se l'esito è positivo, nella parte destra viene mostrata la fase 1, che dà esito positivo con OK oppure segna ERROR con indicati i campi errati. Dopo l'esito positivo viene avviata la fase 2 in cui il codice Javascript chiama una API del server per avviare lo script che esegue la creazione dei modelli. L'utente vedrà un timer e una barra di caricamento nell'attesa. Al termine dello script il timer si ferma e viene mostrata la fase 3 in cui sono presenti i risultati dell'elaborazione. Viene anche mostrato un pulsante per andare al gestore dei file.

L'esecuzione dello script può richiedere tempo: per risolvere il problema, l'avvio del calcolo è stato fatto tramite Javascript in modo che la richiesta non vada in timeout, ma aspetti il termine della computazione.

The screenshot shows the 'Regression Model Calculator' web application. The interface is divided into two main sections: 'Calculation settings' on the left and 'Phases' on the right.

Calculation settings:

- Model Output Directory:** A text input field containing 'qq' and a 'Select' button.
- Model Result Name:** A text input field containing 'a'. Below it, a note states: 'Before the name there is always current year_month_day_hour_min_sec'.
- Supported input date:** A list of supported date formats: '- 18/12/2000 10:15:00', '- 2000-12-20 10:15:00', and '- 2000_12_20_10_1'.
- Model input:** A 'Choose a input file' button and a 'Browse' button.
- Model expected outputs:** A 'Choose multiple expected output' button and a 'Browse' button.
- Window:** A text input field with the placeholder 'Enter a integer value from 0' and a dropdown arrow.
- Test:** A checkbox labeled 'Test:'.
- Run Script:** A blue button at the bottom of the settings section.

Phases:

- Phase 1 - Checking settings.** Status: OK.
- Phase 2 - Start processing.** Status: This may take some minutes. Processing time: 00h 00m 24s.
- Phase 3 - Result.**
 - Missing values in: [3m, 4m, 5m, 6m, 7m]
 - AdjR2 del modello: 0.9893
 - BMSEadj del modello: 0.0652
 - RRSE del modello: 0.1032

At the bottom of the Phases section, a note states: 'All result files start with "2024_03_02_17_46_50-a" and are saved in folder "qq"'. Below this note is a button labeled 'Go to file manager to download or delete'.

Figura 4.6: Pagina del calcolo del modello di regressione

4.4.6 Chi siamo

In questa pagina sono presenti l'elenco di chi ha collaborato al progetto e le immagini dei volantini presenti alla fiera.

UNIVERSITÀ di VERONA Dipartimento di INFORMATICA return to Dashboard Logout

About

This website is part of the project "Fiera dell'agricoltura".
It was made in collaboration with:

Catello Pane
Claudio Tomazzoli
Davide Quaglia
Elia Brentarolli
Gianni Donadon
Mariano Ceccato
Matteo Fiorini
Paolo Guarino
Sara Migliorini
Tiziano Villa

Digital Twin / Gemello Digitale
Ci dedichiamo all'AGRICOLTURA con la stessa PRECISIONE con cui abbiamo fatto INTERNET, veicoli INTELLIGENTI e sistemi di AUTOMAZIONE INDUSTRIALE.
Digital Twin / Gemello Digitale
« Questo intelligente partner che c'è nel tuo software di gestione del tuo campo (agricoltura) »
« Il tuo partner digitale è diventato smart! »
« Ha la capacità di supportare le decisioni, utilizza gli algoritmi più recenti di deep learning e di machine learning »
« Questo ti dà la possibilità di prendere decisioni più intelligenti e accurate »
« La tua fiera di produzione è fatta la tecnologia blockchain per garantire la tracciabilità dei prodotti »
« Questi dati possono essere il tuo argomento di negoziazione »
Contatto: davide.quaglia@univr.it - 3403345704

SUPPORTO ALLE DECISIONI NELLA GESTIONE DELLA MALATTIA IN CAMPO BASATA SUL GEMELLO DIGITALE
Elia Brentarolli, Matteo Fiorini, Catello Pane, Davide Quaglia
Dipartimento di Informatica, Università di Verona
CREA - Centro di ricerca Orticoltura e Florovivaismo, via Cavallotti 55, Piacenza (PR) 43044
Schema del sistema a supporto delle decisioni
Dato su prodotto, qualità, integrità, che si merita con la presenza di macchine intelligenti di campo (robot) sulle foglie
Contatto: elia.brentarolli@univr.it, moio: 3403345704

Ci occupiamo di ANALISI DI DATI con componente SPAZIO TEMPORALI
come le misurazioni rilevate in vari punti di una serra, al fine di creare MODELLI PREDITTIVI efficienti ed efficaci
Pol (t,c), (t,c), (t,c)
C
« stai sfruttando al meglio le opportunità offerte dall'agricoltura 4.0? »
« come stai raccogliendo i dati sul campo? »
« come stai sfruttando i dati raccolti? »
« quanti sensori di servizio per sviluppare un modello predittivo utile? »
« stai usando il tuo robot agricolo per raccogliere dati? »
Contatto: sara.migliorini@univr.it - 045 8007900

Figura 4.7: Pagina di chi siamo

4.5 Deployment

Finora il progetto è sempre stato avviato in modalità sviluppo (development). Tuttavia il server WSGI interno a Flask non è pensato per la messa in esercizio del web server quindi si deve usare un altro programma. Il server WSGI di Flask è sia un server WSGI che un server HTTP, quindi si devono trovare due server corrispondenti per il deployment. Nella documentazione ufficiale vengono elencati alcuni server che si possono usare [26]. Nel progetto è stato usato Gunicorn come server WSGI perché funziona interamente su Python ed è semplice da configurare. Il server WSGI (Web Server Gateway Interface) è un protocollo di trasmissione che stabilisce e descrive comunicazioni ed interazioni tra server ed applicazioni web scritte nel linguaggio Python. È quindi l'interfaccia standard del web service per la programmazione in Python. Questo protocollo specifica come i server devono farsi carico delle richieste provenienti dai browser/client e come devono inoltrare le informazioni richieste alle

relative applicazioni, oltre a come utilizzare le informazioni di cui si sono fatti carico e come rispondere. Perciò Gunicorn è il server che si occuperà di eseguire l'applicazione Flask e di convertire le richieste HTTP per l'ambiente WSGI e le risposte WSGI in risposte HTTP. Di questo server si spiegheranno l'installazione e la configurazione. Manca però il server HTTP che gestisce le richieste e le risposte HTTP. Nel nostro caso è stato usato NGINX dato che era già presente nella macchina, quindi si spiegherà la configurazione necessaria. Infine si deve impostare Gunicorn per l'avvio automatico tramite Supervisor.

4.5.1 Gunicorn

Di seguito vengono spiegati i comandi usati nel progetto (per i dettagli vedere la documentazione ufficiale [27]). Nello stesso ambiente virtuale in cui si è installato Flask si può installare anche Gunicorn. Quindi una volta avviato l'ambiente si può procedere con il seguente comando:

```
1 pip install gunicorn
```

Per eseguire il progetto Flask in modalità produzione si deve eseguire il file `run_flask.py` tramite Gunicorn. Il file `run_flask.py` è quello commentato nella spiegazione del flusso base 4.3.1 ed è rimasto invariato nelle modifiche del progetto. Il comando per eseguire tramite Gunicorn è il seguente:

```
1 gunicorn -w 3 run_flask:app --timeout 0
```

I parametri che vengono passati a Gunicorn sono tre. Il parametro `"-w 3"` indica quanti processi devono essere avviati per servire il web server. In questo caso sono indicati 3 processi controllati da Gunicorn che gestiranno le richieste, così da poter rispondere a più richieste in contemporanea. Il parametro `"run_flask:app"` indica lo script che deve essere eseguito e il nome della variabile con cui deve essere eseguito Flask. Infine il parametro `"--timeout 0"` indica che i processi di Gunicorn non devono essere interrotti se l'esecuzione richiede troppo tempo. L'impostazione a 0, cioè tempo infinito, è stata necessaria dato che l'esecuzione dello script di calcolo della regressione può richiedere svariati minuti e il timeout di default è troppo breve. Per non passare tutti i parametri è possibile creare una configurazione per Gunicorn. Nel progetto è stata messa nel file `gunicorn-config.py` ed è la seguente:

```
1 bind = '127.0.0.1:8001'
2 workers = 3
3 timeout = 0
```

Oltre ai due parametri precedenti è stato aggiunto quello di `bind`. Questo parametro è servito per indicare l'host e su quale porta Gunicorn deve essere in ascolto per le richieste. Nel progetto si è dovuto specificare che l'host di ascolto è solo locale e che la porta da usare è la numero 8000, perché la porta di default 8080 era occupata da un altro servizio.

Per eseguire Gunicorn con il file di configurazione, il comando è:

```
1 gunicorn -c ./deploy/gunicorn-config.py run_flask:app
```

Il server WSGI è stato quindi avviato correttamente. Per terminarlo basta premere CTRL + C.

4.5.2 NGINX

Nella macchina su cui si doveva fare l'installazione era già presente NGINX, quindi si può fare riferimento alla documentazione ufficiale per l'installazione e i dettagli [28]. Il file di configurazione per il nostro server web è stato creato nel file `nginx-web-server-flask` (si noti che non ha un'estensione). La configurazione è la seguente:

```
1 server {
2     listen 8000 ssl;
3     server_name mydomain.it;
4
5     client_max_body_size 100M;
6
7     ssl_certificate /etc/letsencrypt/live/mydomain.it/fullchain
8         .pem;
9     ssl_certificate_key /etc/letsencrypt/live/mydomain.it/
10        privkey.pem;
11
12    error_page 497 301 =307 https://$server_name:
13        $server_port$request_uri;
14
15    location /static {
16        alias /path_to_folder/flaskr/static;
17    }
18
19    location / {
20        proxy_pass http://localhost:8001;
21        include /etc/nginx/proxy_params;
22        proxy_redirect off;
23    }
24 }
```

Nella prima riga viene specificata la porta in ascolto, la 8000, e che le richieste devono essere gestite con la sicurezza SSL/TLS. Poi si indica qual è il nome del dominio del server. Si specifica la dimensione massima del body contenuto nelle richieste, che viene impostata a 100 MB dato che è più che sufficiente per le nostre esigenze. Poi vengono indicati i due path necessari per l'SSL. A differenza di Mosquitto qui viene usato il path di riferimento fornito da Certbot, perché NGINX viene eseguito come utente root quindi non ha problemi ad accedere ai file. La riga successiva serve per fare il reindirizzamento delle richieste dal protocollo HTTP a HTTPS. In questo modo si usa sempre una connessione sicura quando si fanno le richieste. Poi si specificano due locazioni. La prima serve per fornire i file statici dell'applicazione Flask, che vengono richiesti sempre con la prima parte dell'URL formata da `/static`. Deve essere specificato il path assoluto della cartella dei file statici. È anche per questo motivo che sono stati mantenuti tutti i file statici all'interno della cartella `static` di default, rendendo accessibile alle richieste dirette solo quella cartella. La seconda locazione serve invece per girare la richiesta HTTP al server WSGI Gunicorn che eseguirà l'applicazione. La richiesta viene girata tramite il proxy al processo Gunicorn che viene eseguito in locale alla porta 8001.

Una volta creata, la configurazione deve essere spostata nel path delle configurazioni di NGINX, che è il seguente (si deve essere root per eseguire questa azione):

```
/etc/nginx/sites-enabled/
```

Dopo la copia della configurazione si deve riavviare NGINX con il seguente comando:

```
1 sudo systemctl restart nginx
```

A questo punto l'applicazione Flask è raggiungibile al dominio indicato alla porta 8000.

4.5.3 Avvio automatico di Gunicorn

Come ultimo step si deve impostare l'avvio automatico per Gunicorn, mentre per NGINX non serve fare nulla perché viene gestito come servizio con systemctl. Per configurare l'avvio automatico di Gunicorn viene sempre usato Supervisor, visto precedentemente [3.5](#). Viene riportato il file di configurazione:

```
1 [program:web-server-flask]
2   directory=/home/utente/web-server-flask
3   command=/home/utente/web-server-flask/venv/bin/gunicorn -c ./
   deploy/gunicorn-config.py run_flask:app
4   user=utente
5   autostart=true
6   autorestart=true
7   stopasgroup=true
8   killasgroup=true
9   stderr_logfile=/home/utente/web-server-flask/log/gunicorn/
   flaskblog.err.log
10  stdout_logfile=/home/utente/web-server-flask/log/gunicorn/
   flaskblog.out.log
```

La configurazione segue la spiegazione fornita precedentemente [3.5.2](#). La cosa più rilevante è il comando che viene passato. La prima parte indica l'intero path dell'ambiente virtuale in cui è installato Gunicorn. Si è dovuto specificare il suo path all'interno dell'ambiente virtuale perché Supervisor non avvia prima l'ambiente virtuale.

Poi si procede con lo spostamento della configurazione [nella cartella delle configurazioni di Supervisor](#). Infine si eseguono il comando di rilettura configurazione [3.5.3](#) e aggiornamento configurazione [3.5.3](#) per aggiungere Gunicorn come processo automatico.

Si è quindi terminato tutto il necessario per il deployment. Quando la macchina viene riavviata, l'interfaccia web ritorna disponibile.

Capitolo 5

Home Assistant

5.1 Implementazione nel progetto

Home Assistant è un software open-source e gratuito per la domotica, progettato per essere una piattaforma di integrazione indipendente dall'ecosistema dell'Internet of Things (IoT) e un sistema di controllo centrale per i dispositivi smart, con particolare attenzione al controllo locale e alla privacy. Permette di raccogliere i dati provenienti da diversi sensori e di mostrarli in un'interfaccia grafica. Nel progetto questo software era già installato in una macchina, per i dettagli sull'installazione si rimanda alla documentazione ufficiale [29]. Si doveva solo procedere con la configurazione per ricevere i dati da vari topic del Broker MQTT e mostrare i valori in un'interfaccia apposita. Tuttavia prima viene spiegata una modifica che si è dovuto fare per risolvere un problema con i servizi di auto-discovery del software.

5.2 Servizi di auto-discovery

Home Assistant fornisce vari servizi tra cui quelli di auto-discovery. Questi permettono di scoprire nella rete LAN a cui si è connessi le possibili sorgenti dati da integrare nel sistema. Tuttavia dato che Home Assistant era installato all'interno della rete universitaria, questi servizi creavano moltissime richieste a causa dell'elevato numero di dispositivi presenti. È quindi stato necessario modificare la configurazione per disabilitare l'auto-discovery. La configurazione di Home Assistant si trova all'interno del file `configuration.yaml`. Questo file in base al tipo di installazione eseguita può essere modificato da interfaccia grafica o da riga di comando. Nel nostro caso l'installazione è stata fatta tramite container Docker, quindi il file di configurazione era modificabile da riga di comando e si trovava nella seguente cartella:

```
/var/docker
```

Nel file è presente di default la seguente configurazione:

```
default_config:
```

Con questa configurazione si abilitano una serie di componenti che di solito sono utili per semplificare l'uso del software, tra cui quelli di auto-discovery. Per risolvere il problema è stata commentata quella configurazione e sono state aggiunte tutte

le configurazioni a cui corrispondeva disabilitando quelle di auto-discovery. In base alla versione di Home Assistant, la configurazione può corrispondere a un numero maggiore o minore di servizi. In questa installazione era presente la versione 2023.3.1 e quel parametro corrispondeva ai seguenti servizi:

```
1 #automation: !include automations.yaml ALREADY PRESENT
2 backup:
3 bluetooth:
4 #cloud:
5 config:
6 counter:
7 #dhcp:
8 energy:
9 #frontend: ALREADY PRESENT
10 hardware:
11 history:
12 homeassistant_alerts:
13 image:
14 input_boolean:
15 input_button:
16 input_datetime:
17 input_number:
18 input_select:
19 input_text:
20 logbook:
21 map:
22 media_source:
23 mobile_app:
24 my:
25 #network:
26 person:
27 schedule:
28 #scene: !include scenes.yaml ALREADY PRESENT
29 #script: !include scripts.yaml ALREADY PRESENT
30 #ssdp:
31 #stream:
32 sun:
33 system_health:
34 tag:
35 timer:
36 usb:
37 webhook:
38 #zeroconf:
39 zone:
```

I parametri che hanno un cancelletto davanti sono disattivati. I parametri seguiti da ALREADY PRESENT sono stati attivati in modo diverso da un'altra parte nel file della configurazione. I parametri importanti da disattivare erano cloud, dhcp, network, ssdp, stream e zeroconf. In questo modo sono stati disabilitato completamente i servizi di auto-discovery e sono state ridotte al minimo le richieste

che Home Assistant eseguiva nella rete. Si rimanda alla documentazione per sapere il significato di ogni parametro [30]. Nella community di Home Assistant ci sono state altre persone che hanno dovuto affrontare lo stesso problema, riporto alcuni casi che mi sono stati utili [31] [32].

5.3 Integrazione con MQTT

Per far ricevere a Home Assistant i messaggi del broker MQTT si devono fare due passaggi. Il primo è configurare il client per connettersi al broker e il secondo è definire un sensore per ogni valore presente nel messaggio in arrivo dal Broker. Per configurare il client si deve eseguire la procedura da interfaccia web, come spiegato nella documentazione [33]. Quindi per inserire la configurazione si deve andare in "Impostazioni", poi "dispositivi e servizi", premere in basso a destra il pulsante + e selezionare dalla lista "MQTT". A questo punto si apre una finestra che deve essere popolata con la configurazione per la connessione come se fosse un normale client MQTT. La configurazione usata nel progetto è la seguente:

```
broker: mydomain.it
port: 8883
username: username
password: password
client ID: (vuoto)
time: (60 il valore di default)
Usa un certificato client: False
Convalida del certificato: automatico
Ignora la convalida del certificato del broker: True
Protocollo MQTT: 3.1.1
Trasporto MQTT: TCP
```

Poi si preme su avanti e si segue le richieste fino a completare la procedura. Si è quindi configurato il client che riceve i messaggi. Home Assistant supporta la configurazione di un broker alla volta. Nel secondo passaggio si deve modificare il file `configuration.yaml` per aggiungere i vari sensori. Nel file si deve aggiungere ad esempio il seguente codice:

```
1 mqtt:
2   sensor:
3     - name: Serra reale Temperatura Aria
4       state_topic: sensori
5       unique_id: sensor.reali.serraAirTemperature
6       unit_of_measurement: "°C"
7       value_template: >
8         {% if value_json.Temp__C | is_number %}
9           {{ value_json.Temp__C }}
10        {% else %}
11          {{ sensor.reali.serraAirTemperature }}
12        {% endif %}
```

La prima riga indica che il sensore che si sta aggiungendo ha come sorgente il broker MQTT. Poi nella seconda lista si specifica di che tipo è il sensore, in questo caso

è indicato un semplice sensore. Poi si definisce il sensore iniziando la dichiarazione con "- ". Viene indicato il nome visualizzato nell'interfaccia grafica, il topic da cui prendere il messaggio, un id unico per identificare il sensore, l'unità di misura e infine come il valore deve essere visualizzato. Nel campo valore è possibile definire la logica per eseguire controlli, modifiche o altro per ottenere il valore finale. Per aggiungere altri sensori, basta inserire altre definizioni prestando attenzione all'indentazione e ricordandosi di mettere "- " prima della definizione. Nell'appendice vengono riportati altri esempi di definizioni di sensori [C.1](#).

5.4 Visualizzazione dei dati

Per configurare come vengono visualizzati i dati, si deve agire da interfaccia grafica. Come prima cosa si deve creare una nuova plancia (in inglese dashboard). Si deve andare in "Impostazioni", poi "Plance" e in basso a sinistra premere su +. Si apre una finestra in cui si deve mettere il nome della plancia, un'icona, l'URL e altre impostazioni. Una volta configurato, si preme su crea e nella barra a sinistra si aggiungerà la nuova plancia. Successivamente si può iniziare a configurare la visualizzazione, si consiglia di guardare la documentazione per tutte le possibilità di personalizzazione [\[34\]](#). Si va nella plancia appena creata, in alto a destra si apre il menù e si seleziona "modifica plancia". Poi si clicca di nuovo sul menù in alto a destra e si seleziona "Editor di configurazione testuale". Viene visualizzato un file simile a quello di configurazione. Di seguito viene riportata una parte della configurazione di una plancia:

```
1 title: Oidio
2 background: center / cover no-repeat fixed url('/local/
  background_verde_sfumature.jpg')
3 views:
4   - theme: Backend-selected
5     title: Normale
6     icon: ''
7     path: normale
8     badges: []
9     cards:
10      - type: vertical-stack
11        cards:
12          - type: entities
13            title: Riepilogo Sensori
14            entities:
15              - entity: sensor.serra_reale_temperatura_aria
16                name: Temperatura Aria
17      - type: vertical-stack
18        cards:
19          - type: picture-elements
20            title: Stime Rischio Oidio
21            image: http://mydomain.it:8123/local/Mappa_new.png
22            elements:
23              - type: state-badge
```

```

24         entity: sensor.serra_stima_oidio_rischio_1
25         style:
26             top: 28%
27             left: 37%
28             scale: 90%
29             color: rgba(0,0,0,0)

```

Nella configurazione alla prima riga c'è il nome della plancia. La seconda riga definisce il background che deve avere la plancia. L'immagine è salvata in una cartella accessibile da Home Assistant che è

```
/var/docker/www
```

Poi viene definita una scheda della plancia con il tema, il titolo, nessuna icona e il path dell'URL. Poi si definiscono le cards, che sono i gruppi di elementi che devono essere mostrati. Il primo gruppo viene definito come un insieme di card verticali. Al suo interno c'è una card che contiene entità e ha nome "Riepilogo sensori". Le entità in questo caso sono solo una: un sensore identificato da un nome univoco e il nome visualizzato. Anche il secondo gruppo è un insieme di card verticali. Contiene una card di tipo immagine-elementi che permette di visualizzare un'immagine con sopra i valori dei sensori. Viene impostato il titolo, l'immagine in background e la lista di elementi. L'immagine è sempre contenuta nella cartella del background. Gli elementi in questo caso sono solo uno, che è di tipo badge. Viene specificato il nome univoco del sensore da cui prendere i dati e vengono definiti degli stili del badge. Nell'appendice vengono riportate le definizioni usate [C.2](#). Una volta terminata l'impostazione delle varie card e delle relative sorgenti, si preme "salva" in alto a destra e poi su "fatto". In questo modo si è ottenuta la visualizzazione di due sensori con sorgente MQTT modificando la plancia.

Di seguito viene riportata l'immagine di uno dei risultati che è stato raggiunto modificando opportunamente questo file.



Figura 5.1: Plancia RL

Conclusioni

Lo svolgimento del progetto mi ha permesso di applicare varie conoscenze apprese durante gli anni universitari. Da un lato ho potuto sperimentare la programmazione di complessi script Python e dall'altro ho potuto agire come sistemista per far collaborare tutti i vari componenti.

Alla base del progetto c'è il broker MQTT che fa da ponte per i dati tra le varie parti. Tutto parte dall'emulazione di sensori reali che viene fatta da alcuni client MQTT che pubblicano dati. Tramite l'interfaccia web vengono generati i modelli di elaborazione dei dati dei sensori. Questi modelli vengono usati da altri client MQTT per generare nuovi dati a partire dai sensori emulati. Infine con Home Assistant è possibile vedere in un'interfaccia grafica tutti i dati che vengono pubblicati sul broker.

È possibile scaricare il codice sviluppato dalla repository GitHub al [seguente link](#).

Ringraziamenti

Giunto alla conclusione di questa tesi voglio ringraziare una persona molto speciale, la dottoressa Cristina Calonego, l'amore della mia vita. Lei mi ha supportato in tutte le varie fasi del mio percorso universitario motivandomi a studiare e a portare a termine il percorso. Inoltre mi ha aiutato a revisionare questa tesi per renderla comprensibile a tutti e per controllare che non ci fossero porcherie grammaticali. Ringrazio la mia famiglia che mi ha supportato economicamente e moralmente in questi anni nonostante tutte le mie insicurezze. Infine voglio ringraziare il Prof. Davide Quaglia e il Dott. Elia Brentarolli per avermi dato la possibilità di partecipare a questo progetto che è stato presentato alla Fieragricola 2024 di Verona.

Appendice A

Configurazioni broker MQTT

A.1 /mosquitto.conf

Questa configurazione è quella che mosquitto dovrebbe creare al momento dell'installazione. Se non è presente potrebbe non funzionare.

Il seguente file deve essere inserito in **/etc/mosquitto**.

Listing A.1: mosquitto.conf

```
1 # Place your local configuration in /etc/mosquitto/conf.d/
2 #
3 # A full description of the configuration file is at
4 # /usr/share/doc/mosquitto/examples/mosquitto.conf.example
5
6 pid_file /var/run/mosquitto.pid
7
8 persistence true
9 persistence_location /var/lib/mosquitto/
10
11 log_dest file /var/log/mosquitto/mosquitto.log
12
13 include_dir /etc/mosquitto/conf.d
```

A.2 /conf.d/localhost-TLS.config

Questa configurazione è quella creata ad hoc. Ha 2 listener separati, uno per localhost con accesso libero, uno per tutti con accesso tramite credenziali e connessione tramite SSL/TLS.

Il seguente file deve essere inserito in **/etc/mosquitto/conf.d**.

Listing A.2: mosquitto.conf

```
1 # abilita differenti configurazioni per listener
2 per_listener_settings true
3 #
4 # per accesso solo locale e libero
5 listener 1883 localhost
```

```

6 allow_anonymous true
7 #
8 # per accesso con credenziali e certificato SSL/TLS
9 listener 8883
10 allow_anonymous false
11 password_file /etc/mosquitto/passwordfile
12 certfile /etc/mosquitto/certs/fullchain.pem
13 keyfile /etc/mosquitto/certs/privkey.pem

```

A.3 Script rinnovo certificati

Questo script deve essere messo nella cartella `/etc/letsencrypt/renewal-hooks/deploy/` e reso eseguibile. Copia i certificati del dominio inserito nella cartella `/etc/mosquitto/certs/` in modo che Mosquitto possa accedervi. Quando copia, invia a Mosquitto un segnale per ricaricare i certificati.

Listing A.3: mosquitto-copy.sh

```

1 #!/bin/sh
2
3 # This is an example deploy renewal hook for certbot that
4 # copies newly updated
5 # certificates to the Mosquitto certificates directory and
6 # sets the ownership
7 # and permissions so only the mosquitto user can access them,
8 # then signals
9 # Mosquitto to reload certificates.
10
11 # RENEWED_DOMAINS will match the domains being renewed for
12 # that certificate, so
13 # may be just "example.com", or multiple domains "www.example.
14 # com example.com"
15 # depending on your certificate.
16
17 # Place this script in /etc/letsencrypt/renewal-hooks/deploy/
18 # and make it
19 # executable after editing it to your needs.
20
21 # Set which domain this script will be run for
22 MY_DOMAIN=example.com
23 # Set the directory that the certificates will be copied to.
24 CERTIFICATE_DIR=/etc/mosquitto/certs
25
26 for D in ${RENEWED_DOMAINS}; do
27     if [ "${D}" = "${MY_DOMAIN}" ]; then
28         # Copy new certificate to Mosquitto directory
29         cp ${RENEWED_LINEAGE}/fullchain.pem ${CERTIFICATE_DIR}/
30         server.pem

```



```
24     cp ${RENEWED_LINEAGE}/privkey.pem ${CERTIFICATE_DIR}/
      server.key
25
26     # Set ownership to Mosquitto
27     chown mosquitto: ${CERTIFICATE_DIR}/server.pem ${
      CERTIFICATE_DIR}/server.key
28
29     # Ensure permissions are restrictive
30     chmod 0600 ${CERTIFICATE_DIR}/server.pem ${
      CERTIFICATE_DIR}/server.key
31
32     # Tell Mosquitto to reload certificates and
      configuration
33     pkill -HUP -x mosquitto
34 fi
35 done
```

Appendice B

Skeleton client MQTT

config.py

File di configurazione per lo script principale.

```
1  """
2  Configurazione
3  """
4
5  # Config. per connessione
6  HOST = 'host'
7  USER = 'user'
8  PASSWORD = 'pass'
9  CERT_TLS = 'ISRG_Root_X1.pem'
10
11 # Nomi dei topic ricezione
12 TOPIC_RECEIVE_1 = "topic1"
13 TOPIC_RECEIVE_2 = "topic2"
14
15 # Nome topic invio
16 TOPIC_SEND_1 = "topic3"
17 TOPIC_SEND_2 = "topic4"
```

subscriber_e_publisher.py

File principale in cui ci si iscrive a dei topic. Alla ricezione di messaggi si elaborano e poi si pubblica il risultato.

```
1  import paho.mqtt.client as mqtt
2  import json
3
4  import config
5
6  #-----
7  # CLASSE PER I PARAMETRI
8  #
```

```

9 class configurazione_localhost:
10     def __init__(self, topic_ricezione_1, topic_ricezione_2,
11                   topic_invio_1, topic_invio_2):
12         self.dizionario = {"host": "localhost",
13                             "porta": 1883,
14                             "login": False,
15                             "username": "",
16                             "password": "",
17                             "topic-ricezione-1":
18                                 topic_ricezione_1,
19                             "topic-ricezione-2":
20                                 topic_ricezione_2,
21                             "topic-invio-1": topic_invio_1,
22                             "topic-invio-2": topic_invio_2,
23                             "cert_tls": ''}
24
25     def aggiorna_ad_esterna_login_tls(self, host, username,
26                                       password, nome_cert):
27         self.dizionario.update({"host": host})
28         self.dizionario.update({"porta": 8883})
29         self.dizionario.update({"login": True})
30         self.dizionario.update({"username": username})
31         self.dizionario.update({"password": password})
32         self.dizionario.update({"cert_tls": nome_cert})
33
34     def get_host(self):
35         return self.dizionario.get('host')
36
37     def get_porta(self):
38         return int(self.dizionario.get('porta'))
39
40     def get_login(self):
41         return bool(self.dizionario.get('login'))
42
43     def get_username(self):
44         return self.dizionario.get('username')
45
46     def get_password(self):
47         return self.dizionario.get('password')
48
49     def get_topic_ric_1(self):
50         return self.dizionario.get('topic-ricezione-1')
51
52     def get_topic_ric_2(self):
53         return self.dizionario.get('topic-ricezione-2')
54
55     def get_topic_inv_1(self):
56         return self.dizionario.get('topic-invio-1')
57
58     def get_topic_inv_2(self):

```

```

55         return self.dizionario.get('topic-invio-2')
56
57     def get_cert_tls(self):
58         return self.dizionario.get('cert_tls')
59
60 #-----
61 # FUNZIONE PER LA ELABORAZIONE ALLA RECEZIONE DI UN MESSAGGIO
62 # L'output deve essere un json/dizionario
63 #
64 def elaborazione(msg, topic, par):
65     global variabile_globale_1
66     global variabile_globale_2
67
68     # DEBUG - Stampa tutto il contenuto del messaggio
69     # print(msg)
70
71     # Converte il messaggio in un json (dizionario)
72     msg_json = json.loads(msg)
73
74
75     # Esegue il dump del json
76     nuovo_msg = json.dumps(msg_json)
77
78     # TOGLIERE IL COMMENTO PER DEBUG
79     # print(nuovo_msg)
80     return nuovo_msg
81
82 #-----
83 # CLIENT MQTT
84 # definizione di cosa deve fare alla connessione, iscrizione,
85 # ricezione, invio, log
86 #
87 def sub_pub_mqtt(par: configurazione_localhost):
88
89     def on_connect(client, userdata, flags, rc):
90         # TOGLIERE IL COMMENTO PER DEBUG
91         # print(client, userdata, flags, rc)
92         pass
93
94     def on_subscribe(client, userdata, mid, granted_qos):
95         # TOGLIERE IL COMMENTO PER DEBUG
96         # print("message topic=", mid)
97         # print("message qos=", granted_qos)
98         pass
99
100     def on_message(client, userdata, msg):
101         json_string = elaborazione(msg.payload.decode(), msg.
102                                     topic, par)
103
104         if json_string:

```

```

103         client_pub_sub.publish(
104             topic=par.get_topic_inv_1(),
105             payload=json_string,
106             retain=True
107         )
108
109     def on_log(client, userdata, level, buf):
110         print("log: ", buf)
111
112     # crea il client per la connessione
113     client_pub_sub = mqtt.Client(
114         client_id=None,
115         clean_session=True,
116         userdata=None,
117         protocol=mqtt.MQTTv311,
118         transport='tcp'
119     )
120
121     # callback
122     client_pub_sub.on_connect = on_connect # si puo'
123         commentare se non interessa
124     client_pub_sub.on_subscribe = on_subscribe # si puo'
125         commentare se non interessa
126     client_pub_sub.on_message = on_message
127     # -----
128     # PER DEBUG
129     # da commentare per non vedere i log
130     #client_pub_sub.on_log = on_log
131
132     # controlla se c'e' il login
133     if par.get_login():
134         client_pub_sub.username_pw_set(par.get_username(), par
135             .get_password())
136
137     # controlla la porta di invio e setta il certificato
138     if par.get_porta() != 1883:
139         client_pub_sub.tls_set(ca_certs=par.get_cert_tls(),
140             tls_version=2)
141
142     # esegue la connessione al broker
143     client_pub_sub.connect(
144         host=par.get_host(),
145         port=par.get_porta(),
146         keepalive=60
147     )
148
149     # Iscrizione ai topic di ricezione
150     client_pub_sub.subscribe(topic=par.get_topic_ric_1())
151     # CI SI PUO' ISCRIVERE A PIU' TOPIC
152     # client_pub_sub.subscribe(topic=par.get_topic_ric_2())

```

```

149
150     # attesa dei messaggi all'infinito, serve per eseguire i
      callback
151     client_pub_sub.loop_forever(timeout=1.0, max_packets=1,
      retry_first_connection=False)
152
153 #-----
154 # FUNZIONE MAIN
155 if __name__ == "__main__":
156
157     # definizione di variabili globali
158     global variabile_globale_1
159     global variabile_globale_2
160
161     variabile_globale_1 = False
162     variabile_globale_2 = False
163
164     # Connessione al broker e avvio del client mqtt
165     parametri = configurazione_localhost(config.
      TOPIC_RECEIVE_1, config.TOPIC_RECEIVE_2, config.
      TOPIC_SEND_1, config.TOPIC_SEND_2)
166     #
167     # togliere il commento per configurazione esterna alla
      macchina
168     #parametri.aggiorna_ad_esterna_login_tls(config.HOST,
      config.USER, config.PASSWORD, config.CERT_TLS)
169
170     sub_pub_mqtt(parametri)

```

Appendice C

Configurazioni Home Assistant

C.1 Definizioni di sensori MQTT

Qui sono riportate le definizioni di sensori MQTT usate nel file configuration.yaml.

Sensore con valore semplice

La configurazione di un sensore con valore numerico intero.

```
1 - name: Serra reale Temperatura Aria
2   state_topic: sensori
3   unique_id: sensor.reali.serraAirTemperature
4   unit_of_measurement: "°C"
5   value_template: >
6     {% if value_json.Temp__C | is_number %}
7       {{ value_json.Temp__C }}
8     {% else %}
9       {{ sensor.reali.serraAirTemperature }}
10    {% endif %}
```

Sensore con valore arrotondato

La configurazione di un sensore con valore numerico con la virgola che viene arrotondato alla seconda cifra decimale.

```
1 - name: Serra stima RL Temperatura 1
2   state_topic: sensori/modelliRL
3   unique_id: sensor.PredRL.serraTemperatura_1
4   unit_of_measurement: "°C"
5   value_template: >
6     {% if value_json.RL_1 | is_number %}
7       {{ value_json.RL_1 | float | round(2) }}
8     {% else %}
9       {{ sensor.PredRL.serraTemperatura_1 }}
10    {% endif %}
```

C.2 Definizioni di cards

Di seguito viene riportato il codice usato per la creazione delle card nell'interfaccia grafica.

Card per le entità

Permette di mostrare il nome e il valore del sensore.

```
1 - type: entities
2   title: Temperatura
3   entities:
4     - entity: sensor.serra_reale_temperatura_aria
5       name: Temperatura Aria
6     - entity: sensor.serra_reale_massima_temperatura_aria
7       name: Temperatura Aria massima
8     - entity: sensor.serra_reale_minima_temperatura_aria
9       name: Temperatura Aria minima
```

Card per immagini-elementi

Permette di mostrare un'immagine con sopra dei badge che contengono i valori dei sensori.

```
1 - type: picture-elements
2   title: Stime RL
3   image: http://mydomain.it:8123/local/Mappa_new.png
4   elements:
5     - type: state-badge
6       entity: sensor.serra_stima_rl_temperatura_1
7       style:
8         top: 28%
9         left: 37%
10        scale: 95%
11        color: rgba(0,0,0,0)
12    - type: state-badge
13      entity: sensor.serra_stima_rl_temperatura_3
14      style:
15        top: 50%
16        left: 37%
17        scale: 95%
18        color: rgba(0,0,0,0)
19    - type: state-badge
20      entity: sensor.serra_stima_rl_temperatura_5
21      style:
22        top: 74%
23        left: 37%
24        scale: 95%
25        color: rgba(0,0,0,0)
26    - type: state-badge
```



```

27     entity: sensor.serra_stima_rl_temperatura_7
28     style:
29         top: 95%
30         left: 37%
31         scale: 95%
32         color: rgba(0,0,0,0)
33 - type: state-badge
34     entity: sensor.serra_stima_rl_temperatura_2
35     style:
36         top: 28%
37         left: 62%
38         scale: 95%
39         color: rgba(0,0,0,0)
40 - type: state-badge
41     entity: sensor.serra_stima_rl_temperatura_4
42     style:
43         top: 50%
44         left: 62%
45         scale: 95%
46         color: rgba(0,0,0,0)
47 - type: state-badge
48     entity: sensor.serra_stima_rl_temperatura_6
49     style:
50         top: 74%
51         left: 62%
52         scale: 95%
53         color: rgba(0,0,0,0)
54 - type: state-badge
55     entity: sensor.serra_stima_rl_temperatura_8
56     style:
57         top: 95%
58         left: 62%
59         scale: 95%
60         color: rgba(0,0,0,0)

```

Card per iframe

Permette di mostrare una pagina web all'interno di un elemento. È stato usato per inserire i grafici di Grafana.

```

1 - type: iframe
2   url: >- url_to_web_page_to_embed
3   aspect_ratio: 40%

```

Bibliografia

- [1] <https://mosquitto.org/download/>.
- [2] <https://mosquitto.org/man/mosquitto-conf-5.html>.
- [3] <https://mosquitto.org/documentation/authentication-methods/>.
- [4] https://mosquitto.org/man/mosquitto_passwd-1.html.
- [5] <http://www.steves-internet-guide.com/mqtt-username-password-example/>.
- [6] <https://mosquitto.org/man/mosquitto-tls-7.html>.
- [7] <https://www.steves-internet-guide.com/using-lets-encrypt-certificate-mosquitto/>.
- [8] <https://github.com/eclipse/mosquitto/blob/master/README-letsencrypt.md>.
- [9] <https://mosquitto.org/man/mosquitto-tls-7.html>.
- [10] <https://tecadmin.net/auto-renew-lets-encrypt-certificates/>.
- [11] <https://www.python.org/downloads/>.
- [12] <https://virtualenv.pypa.io>.
- [13] <https://eclipse.dev/paho/index.php?page=clients/python/index.php>.
- [14] <https://eclipse.dev/paho/files/paho.mqtt.python/html/index.html>.
- [15] <https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html>.
- [16] <https://github.com/eclipse/paho.mqtt.python/tree/master/examples>.
- [17] <http://www.steves-internet-guide.com/into-mqtt-python-client/>.
- [18] <https://www.digitalocean.com/community/tutorials/how-to-install-and-manage-supervisor-on-ubuntu-and-debian-vps>.
- [19] <http://supervisord.org>.
- [20] <http://supervisord.org/configuration.html/>.

- [21] <http://supervisord.org/configuration.html#program-x-section-settings>.
- [22] <http://supervisord.org/running.html#supervisorctl-actions>.
- [23] <https://flask.palletsprojects.com/en/3.0.x/>.
- [24] <https://flask.palletsprojects.com/en/3.0.x/tutorial/views/>.
- [25] <https://flask.palletsprojects.com/en/3.0.x/tutorial/templates/>.
- [26] <https://flask.palletsprojects.com/en/3.0.x/deploying/>.
- [27] <https://docs.gunicorn.org/en/stable/index.html>.
- [28] <https://nginx.org/en/docs/>.
- [29] <https://www.home-assistant.io/installation/>.
- [30] https://www.home-assistant.io/integrations/default_config/#configuration.
- [31] <https://community.home-assistant.io/t/default-config-what-is-inside-as-of-2022-4/411015>.
- [32] <https://community.home-assistant.io/t/convert-away-from-default-config/204333/6>.
- [33] <https://www.home-assistant.io/integrations/mqtt/>.
- [34] <https://www.home-assistant.io/dashboards/>.