

RELAZIONE DI PROGETTO DI “APPLICAZIONI E
SERVIZI WEB”

Chirp Air Station

Antonio Rotundo – 0001006574 {antonio.rotundo2@studio.unibo.it}

11/09/2023

Indice

Indice	2
1 Introduzione	3
2 Requisiti	4
3 Design.....	5
4 Tecnologie	9
5 Codice.....	12
5.1 Server HTTP Gateway	12
5.2 MQTT Service	13
5.3 Web Service	14
5.3.1 Gestione letture real-time	14
5.3.2 Gestione routes e Express+Socket.IO	14
5.3.3 Gestione di autorizzazione	15
6 Test.....	18
7 Deployment.....	22
8 Conclusioni	23
Bibliografia	24

1 Introduzione

Il progetto si pone l'obiettivo di realizzare un servizio di stazioni meteorologiche per la rilevazione della temperatura, pressione, umidità e indice di qualità dell'aria; connesse allo stesso ambiente urbano o in città diverse.

Si compone dell'uso del micro-controllore ESP32, dotato di modulo di comunicazione LoRa e Wi-Fi, assieme al sensore BME680 della Bosh SensorTec [1], i quali invieranno i dati di rilevazione ai Gateway, tramite connessione a radiofrequenza LoRa. La scelta di questo protocollo si basa sulle potenzialità di ampio raggio di copertura, della potenziale scalabilità e dei consumi energetici ridotti.

Il Gateway, invece, riceverà le informazioni da parte di tutti i sensori all'interno del proprio raggio di copertura LoRa e invierà, attraverso interfaccia di rete Wi-Fi, quindi attraverso rete Internet, i relativi dati pre-elaborati in formato consono ad un broker MQTT. In questo caso, come broker MQTT, è stato scelto un servizio ad accesso pubblico (HiveMQ) [2], per dimostrare le potenzialità del sistema a prescindere dalla posizione del broker.

Le informazioni pubblicate dai Gateway, verranno pubblicate su un determinato topic, definito esclusivamente per il progetto e verranno, successivamente, catturate da un apposito servizio, denominato MQTT Reader, il quale avrà il compito di sottoscriversi allo stesso topic, in modo tale da ricevere le informazioni provenienti dalla rete sensoristica e conservarle all'interno di un database.

Il servizio MQTT Reader sfrutterà quindi un Database Service, il quale, basandosi sul servizio database MongoDB [3], anch'esso opportunamente containerizzato, darà a disposizione una serie di API per le normali operazioni sul database (ricerca, inserimento, aggregazione, etc).

Infine, viene servito un Web Service che disporrà un client realizzato mediante framework React.js [4], tramite il quale sarà possibile accedere al servizio mediante opportune credenziali. Una volta autenticatosi, il client potrà usufruire di opportune API servite dallo stesso Web Service che gli permetterà di visionare le informazioni inerenti all'utente e alla propria sensoristica e di poter visionare, in tempo reale, le letture da parte di un sensore specifico.

2 Requisiti

L'obiettivo del progetto è quello di offrire un servizio di monitoraggio ambientale, ricavando dati come la temperatura, umidità, pressione atmosferica e qualità dell'aria, in maniera localizzata, che sia scalabile, anche in ambienti metropolitani.

Come requisito, tutti i dispositivi sensoristici devono poter comunicare attraverso un protocollo a risparmio energetico, in modo tale da poter garantire una maggior autonomia. I Gateway, oltre a poter comunicare tramite lo stesso protocollo dei sensori, dovranno essere muniti di connettività tramite un protocollo IP (come ad esempio Wi-Fi, Ethernet, etc), tramite il quale verranno inviati, attraverso Internet, i dati provenienti dai sensori. Tale architettura dovrà essere scalabile orizzontalmente, in modo tale che la stessa rete possa essere ampliata da chiunque. La sensoristica a bordo dei dispositivi sensoristici dovrà permettere il rilevamento di temperatura, umidità, pressione atmosferica e la resistenza dell'aria. Queste informazioni dovranno essere utilizzate per stimare la qualità dell'aria.

I dati raccolti da tutti i dispositivi Gateway, appartenenti allo stesso progetto, dovranno essere inviati attraverso Internet, mediante un protocollo applicativo che permetta lo scambio di informazioni, indipendentemente dalla ubicazione del servizio principale. Il protocollo applicativo dovrà inoltre consentire la ricezione e gestione di tali dati in tempo reale e dovranno essere strutturate mediante un protocollo inerente all'applicativo stesso. Tale protocollo dovrà intendersi scalabile e accessibile a tutti i dispositivi Gateway e permettere una comunicazione bilaterale. È necessario che tale protocollo sia compatibile, indipendentemente dalle reti IP ove sono collocati i dispositivi Gateway e il servizio applicativo (come ad esempio reti sotto NAT o Firewall).

Tutte le informazioni ricevute mediante il protocollo applicativo scelto, dovranno essere conservate all'interno di una struttura dati o database consono. Il database, scelto opportunamente, dovrà consentire la memorizzazione di dati non omogenei, permettendo l'ampliamento delle funzionalità del servizio. Inoltre, il database è da intendersi scalabile orizzontalmente, in modo tale da poter sostenere un numero di sensori e di utenti che possa crescere esponenzialmente. Il servizio che avrà il compito di memorizzare tali informazioni, considerando il tempo di arrivo e possibili duplicati degli stessi dati.

Infine, è richiesto un servizio che metta a disposizione una serie di interfacce applicative ed interfacce utente, tramite le quali dovrà essere possibile visionare i dati raccolti e interagire con lo stesso sistema. Tali operazioni dovranno essere opportunamente protette mediante un sistema di autenticazione utente e di permessi relativi ai ruoli utente. Tra i ruoli principali dovranno essere identificati almeno i ruoli di utente normale e di amministratore di sistema. L'utente normale avrà accesso ai dati relativi ai propri sensori e al proprio account. L'amministratore di sistema, oltre a poter aver accesso ai propri dati e ai propri sensori, dovrà aver accesso a tutte le informazioni di tutti gli utenti e di tutti i sensori registrati presso la stessa piattaforma. Tale piattaforma dovrà inoltre permettere la visione dello storico dei dati ricevuti, la visione dei dati in ricezione in tempo reale e di visualizzare, tramite grafici, l'andamento di tali informazioni. La piattaforma dovrà essere fruibile attraverso qualsiasi sistema multimediale (computer, tablet, smartphone, etc) moderno.

3 Design

In funzione dei requisiti richiesti per lo sviluppo del progetto, come prima fase si sono definite delle aree operazionali del progetto, ove in ognuna delle quali si andranno a definire i servizi, i dispositivi e i protocolli che faranno parte della propria area. Tra le aree definite nell'architettura di progetto abbiamo l'Edge Field, l'IP Local Field, l'Internet Field ed il Service Field. Questa suddivisione è da considerarsi strettamente figurativa in modo tale da comprendere le varie aree.

Di seguito è descritta l'architettura progettuale considerando le aree sopra definite e le componenti di ogni area.

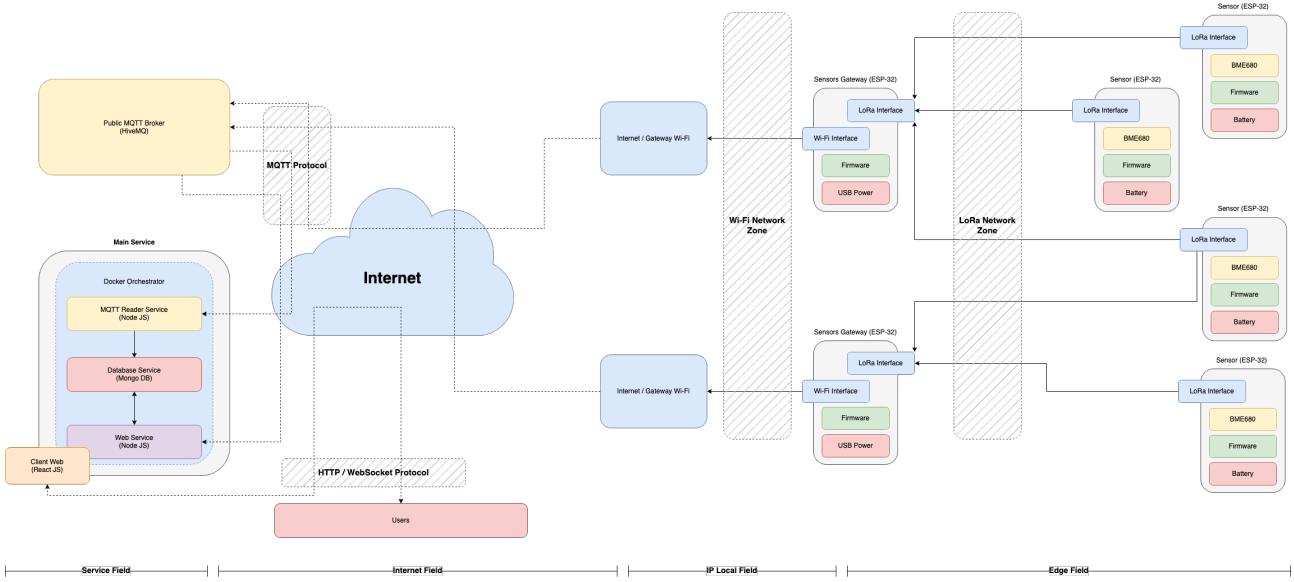


Figura 1 Architettura del progetto

In primo luogo è stata progettata l'area Edge Field. In quest'area vengono definiti i relativi componenti (device e Gateway) e il protocollo a basso consumo energetico opportunatamente scelto per il progetto. Considerando che il progetto debba essere scalabile orizzontalmente, che i dispositivi sensoristici debbano poter operare mediante alimentazione da batteria e che debba poter comunicare con un dispositivo Gateway a lunga distanza, è stato scelto come protocollo di comunicazione il protocollo LoRa. Il protocollo LoRa (Long Range) è un protocollo basato sulla modulazione di frequenza a spettro espanso, che si basa sul Chirp Spread Spectrum (CSS), sviluppato dalla Semtech [5]. Questo protocollo utilizza delle bande di radiofrequenza libere e consente trasmissione a lungo raggio (oltre i 10 kilometri nelle zone rurali e circa 5 kilometri nelle zone fortemente urbanizzate) a basso consumo energetico. Questo permetterà al sistema di poter scalare a prescindere dalla copertura che si vuole ottenere.

Il dispositivo, oltre a dover avere un modulo di comunicazione LoRa, dovrà essere dotato di un sensore di temperatura, umidità, pressione e della resistenza dell'aria. Come sensore è stato scelto il BME680 della Bosch Sensortec. Per limitare l'uso di periferiche superflue, è stato scelto di usare il protocollo I2C per interfacciare il sensore con il microcontrollore.

Come micro-controllore è stato scelto l'ESP32, un microcontrollore di nuova generazione sviluppato dalla Espressif Systems [6], che offre una serie di funzionalità avanzate, come l'elaborazione parallela, consentendo di gestire più processi contemporaneamente. Questa particolarità rende la gestione della connettività e della gestione dell'elaborazione dei dati provenienti dal sensore BME680 più performante e soprattutto più breve, limitando il tempo di attività del dispositivo stesso.

L'ESP32 inoltre dispone di una modalità deep-sleep che gli consente di andare in modalità risparmio energetico e di riattivarsi dopo un tempo configurabile (in caso di test ogni 5 secondi, in produzione ogni 60 minuti). In funzione di questo micro-controllore, è stata scelta una scheda di sviluppo composta dallo stesso microcontrollore, da un modulo LoRa e relativa antenna e da uno slot SD e un monitor LCD, questi due utilizzati esclusivamente durante lo sviluppo per debug.



Figura 2 - Foto dispositivo Gateway

Sia nell'Edge Field che nell'IP Local Field troviamo i dispositivi Gateway. Il dispositivo Gateway sarà dotato di due modalità operazionali. La prima modalità è la modalità configurazione, durante la quale l'utente avrà modo di configurare la connettività dello stesso Gateway, mediante un'interfaccia grafica web servita dallo stesso dispositivo.

CAS - Gateway

SSID	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="save"/>	

Figura 3 - Pannello Web Gateway (modalità configurazione)

Una volta configurato il dispositivo, entrerà in modalità operativa. Durante la prima fase della modalità operativa il dispositivo proverà a connettersi alla rete configurata e di connettersi ad Internet. Qualora dovesse esserci un errore di connessione, il dispositivo si riavvierà in modalità configurazione, altrimenti procederà con la modalità operativa. Nella seconda fase di modalità operativa, il Gateway avrà il compito di ricevere i messaggi dai dispositivi sensoristici tramite connettività LoRa, ristrutturare i dati in arrivo in un formato JSON e di inviare questi dati mediante protocollo di rete applicativo. È giusto precisare che la scelta della connettività Wi-Fi è dato dal fatto che il micro-controllore ESP32 ne sia già dotato, tuttavia è ovvio precisare pure che potrà essere utilizzata qualsiasi connettività IP che possa dare accesso alla rete Internet, come ad esempio Ethernet, Fibra Ottica, connessione Cellulare, etc.

Come protocollo applicativo per l'invio dei dati attraverso la connessione ad Internet è stato scelto il protocollo MQTT (Message Queuing Telemetry Transport) [7]. MQTT è un protocollo di messaggistica basato su uno standard ed è utilizzato dalla comunicazione tra macchine, in particolar modo nell'IoT, dove generalmente si hanno risorse limitate assieme alla larghezza di banda limitata. MQTT è un protocollo di tipo Publisher/Subscriber che permette di disaccoppiare il classico mittente del messaggio (editore) dal destinatario del messaggio (sottoscrittore). Inoltre, si ha un terzo componente denominato Broker, il quale ha il compito di filtrare tutti i messaggi in arrivo dai Publisher e di distribuirli correttamente a tutti i Subscriber. Come Broker MQTT è stato scelto un Broker pubblico e gratuito denominato HiveMQ, il quale può essere sostituito da un altro Broker pubblico o Broker privato. La scelta di MQTT permette inoltre ai Gateway e al servizio applicativo di poter comunicare a prescindere dallo stato e dall'architettura della propria rete.

Infine nel Service Field, individuiamo tutte le componenti a micro-servizio del progetto. Tra i micro-servizi definiti in progettazione vi abbiamo il micro-servizio MQTT Reader. Questo micro-servizio ha il compito di sottoscriversi allo stesso topic MQTT del progetto, sul quale i Gateway pubblicheranno i dati sensoristici in arrivo dai dispositivi sensoristici, in modo tale da poter evitare possibili duplicati dello stesso messaggio e di conservarlo adeguatamente all'interno del servizio database.

Un secondo servizio definito durante la progettazione è il Web Service. Questo servizio ha il compito di fornire dovute interfacce applicative (API) tramite il protocollo HTTP e interfacce utente con le quali sarà possibile poter interagire con i dati conservati all'interno del servizio database, ricevuti dalla rete sensoristica. Queste interazioni sono opportunamente protette e gestite da un sistema di autenticazione interno al progetto e che può essere esteso mediante altri sistemi di autenticazione. Questo servizio, inoltre, ha l'onere di permettere all'utente di visualizzare i dati provenienti da uno specifico sensore in tempo reale, permettendo agli utenti di avere un'interazione maggiore con i propri sensori. Il servizio è progettato per essere scalabile orizzontalmente, e quindi permette la duplicazione del servizio applicando un bilanciatore di carico, sia verticalmente, ovvero potendo ampliare le funzionalità dello stesso servizio. Infine, il servizio offre un accesso multi-platform agli utenti con il quale potranno autenticarsi ed interagire con le funzionalità senza dover implementare alcuna interfaccia.

Un terzo ed ultimo servizio definito durante la progettazione è il Database Service. Questo servizio ha il compito di gestire una struttura dati database, possibilmente non relazionale. La scelta di questa tipologia di database è data dal fatto che in questo modo sarà possibile poter memorizzare dati facenti parte della stessa categoria/collezione senza dover essere per forza omogenei, in questo modo più sensori potranno inviare ulteriori dati rispetto ai sensori normali (come l'aggiunta di nuovi sensori o parametri). Inoltre, questa tipologia di database è facilmente scalabile orizzontalmente.

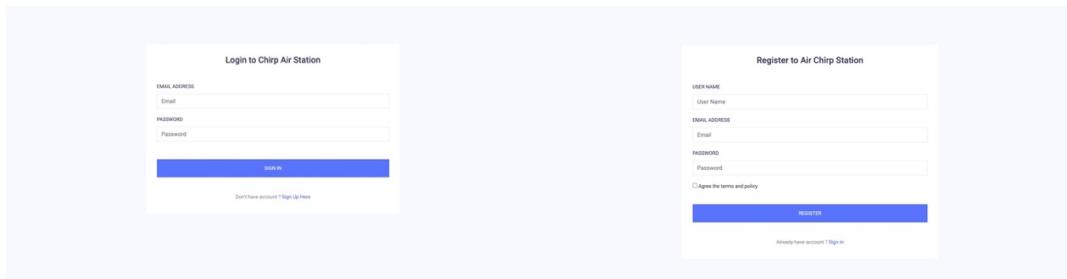


Figura 4 - Schermata login e registrazione

L'interfaccia Web è stata progettata per poter sfruttare un sistema a layout a prescindere dalla view alla quale si vuole accedere. Tra i layout abbiamo quello pubblico, ovvero una versione semplificata dell'interfaccia grafica inerente a qualsiasi visitatore della piattaforma e quello autenticato, ovvero una versione dello stesso layout compreso però di menù di navigazione e barra utente, tipicamente posizionata in maniera fissa sul lato superiore dell'interfaccia.

Tra le view pubbliche vi troviamo la view di login e quella di registrazione utente, la prima delle quali utilizzata come view di root della piattaforma. È importante sottolineare che è possibile effettuare la navigazione tra queste due view attraverso un link specifico all'interno di ogni form. Come è possibile notare, inoltre, il form di login permetterà l'autenticazione dell'utente mediante l'uso di e-mail e password e sarà sfruttato lo stesso indirizzo e-mail per identificare univocamente l'utente all'interno del database. Nel caso in cui un utente dovesse sbagliare ad effettuare l'operazione di login, la view notificherà mediante messaggio di errore l'evento appena avvenuto, considerando molteplici casi, come ad esempio password errata o account non esistente. Tramite la view registrazione è possibile effettuare la creazione di un nuovo account, richiedendo all'utente alcune delle sue informazioni oltre che alla sua e-mail e alla sua password. In questa fase il client effettuerà i dovuti controlli dei dati inseriti dall'utente, controlli che verranno effettuati anche da parte del server, per questioni di

sicurezza. Anche qui, in caso di errori dovuti ai controlli o alla creazione dell'account, l'utente verrà notificato di tali eventi mediante messaggi di errore opportunamente gestiti. Dopo la creazione dell'account o dopo aver effettuato l'autenticazione mediante login, l'utente verrà autenticato dal server, avviandone la relativa sessione, spostando automaticamente lo stesso utente alla prima view del layout autenticato.

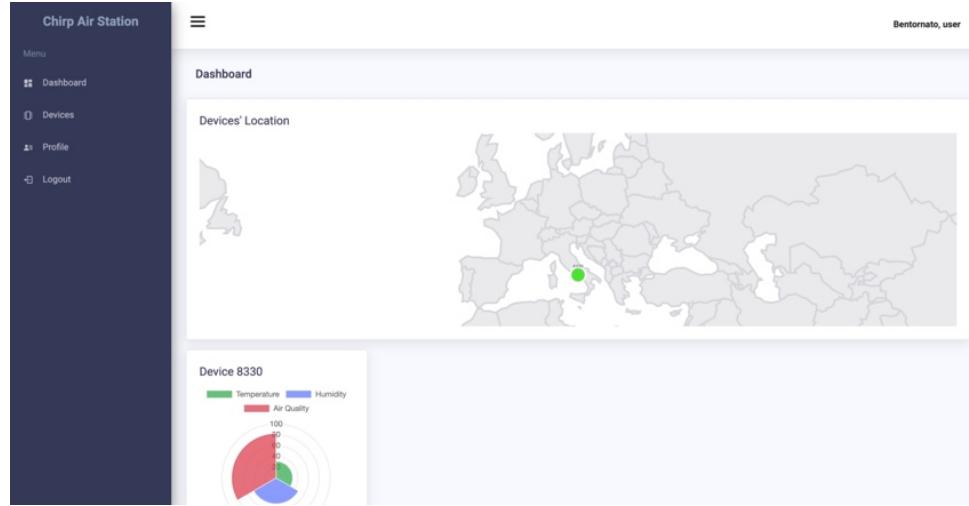


Figura 5 - Dashboard utente

4 Tecnologie

Per lo sviluppo del Web Service è stato impiegato NodeJS [8], un interprete/framework Javascript che permette di sfruttare il suddetto linguaggio a livello back-end. A differenza degli interpreti Javascript utilizzati nei moderni browser Web, NodeJS ha a disposizione tutto il set di chiamate di sistema che permettono di poter interagire completamente con il sistema ospitante. Si basa sull'utilizzo di modulo Javascript, ovvero librerie esterne o create dallo stesso sviluppatore e ne permettono l'esportazione ed impostazione di funzionalità aggiuntive. Questi moduli e librerie possono essere installate e gestite mediante il Node Package Manager (npm) [9] che assieme al package.json permette la gestione del progetto e delle sue dipendenze. Questa meccanica permette un più semplice deploy dello stesso progetto e installazione delle sue dipendenze. NodeJS si basa sulla gestione ad eventi, in particolar modo sfruttando la classe EventEmitter, la quale rappresenta la classe principale sulla quale tutti i moduli con comportamenti asincroni si basano. Infine, NodeJS permette di gestire eventi asincroni in un contesto mono-thread, il quale facilita per sua natura la scalabilità orizzontale.

Tra i moduli utilizzati nel progetto vi abbiamo ExpressJS, Express Session, Socket.IO e Mongoose. ExpressJS [10] è un modulo NodeJS che permette lo sviluppo di un server HTTP e che facilita la gestione delle route in funzione dei verbi HTTP utilizzati. Anch'esso si basa sulla classe EventEmitter e sul concetto di Middleware, ovvero la capacità di poter gestire eventuali richieste mediante una catena di controlli ed elaborazioni delle stesse richieste, semplificando la gestione e lo sviluppo di meccaniche di controlli e autorizzazioni. Inoltre, ExpressJS permette la definizione delle varie route anche su più Router, espandendo il concetto di route, potendo implementare ulteriori controlli su più sottogruppi di route.

Express-Session [11] è un modulo NodeJS, utilizzato assieme ad ExpressJS, per la gestione di sessioni utente e dei relativi cookie per il client. Permette di conservare le relative sessioni o in memoria o all'interno di altre strutture dati, come ad esempio database o cache, mediante l'ausilio di dovuti controller. Questo modulo permette di gestire le sessioni utente all'interno delle varie middleware di richiesta mediante l'oggetto che ne rappresenta la stessa richiesta.

Socket.IO [12] è un modulo NodeJS che permette di implementare, attraverso gli eventi, delle comunicazioni in real time. Per poter implementare questa comunicazione sfrutta due protocolli a livello applicativo, HTTP [13] e WebSocket [14]. La scelta di quale protocollo sfruttare dipende dalla compatibilità del client. Principalmente Socket.IO sfrutterà HTTP per simulare una connessione real time attraverso il polling, se il client dovesse essere compatibile al protocollo WebSocket, sfrutterà lo stesso HTTP per richiedere l'upgrade di protocollo. Questa meccanica è del tutto trasparente allo sviluppatore e viene gestita da una componente di Socket.IO denominato Engine.IO [15]. Il modulo permette lo scambio di messaggi tra client e server attraverso lo scatenare di eventi personalizzati. Socket.IO permette inoltre di gestire diversi namespace, stanze e di poter emettere eventi anche in maniera broadcast.

Mongoose [16] è un modulo NodeJS che permette di interfacciarsi con il database MongoDB mediante eventi asincroni. Il modulo permette di definire i modelli e le loro relative proprietà che andranno poi a essere utilizzato all'interno dell'applicazione. Infine, la stessa libreria permette, attraverso i modelli definiti, di effettuare le normali operazioni di ricerca, aggiornamento, creazione, eliminazione e aggregazione mediante le varie operazioni a catena native di MongoDB.

Il database MongoDB è un database orientato ai documenti e collezioni. Questo database permette, data la sua natura, di poter conservare informazioni non omogenee all'interno delle stesse collezioni, permettendo di avere un sistema dinamico al costo delle varie ricerche all'interno del database (operazione, il più delle volte, più costosa di una normale creazione o aggiornamento). Infatti, per poter effettuare delle operazioni di ricerca più complesse, dovranno essere effettuate delle

operazioni di aggregazione delle informazioni, sfruttando anche più operazioni a catena. MongoDB è un database che permette quindi di poter conservare informazioni diverse tra loro all'interno della stessa collezione e di poter essere distribuibile su più macchine server, rendendolo scalabile orizzontalmente.

Per poter deployare meglio i vari servizi, anche su macchine con sistemi diversi, è stata sfruttata la tecnologia Docker. Docker [17] è una tecnologia di containerizzazione dei servizi, nativa sui sistemi Unix. Questa permette la creazione di ambienti isolati (sandbox) all'interno dello stesso sistema. A differenza della normale virtualizzazione, ove vengono emulate anche le periferiche hardware, come ad esempio CPU e RAM, Docker sfrutta nativamente lo stesso hardware, passando direttamente dall'unico kernel attualmente presente nel sistema ospitante. Questo permette di non assegnare le risorse di sistema direttamente ad un singolo processo, ma di allocare le risorse richieste dal container Docker come se fosse un normale processo di sistema, garantendo delle performance "native" e un minor consumo di risorse. Docker permette di creare delle immagini con le quali potranno essere avviati anche più container. Queste immagini possono essere scaricate dal registro ufficiale di Docker, da un registro privato oppure create partendo da un file di specifiche (Dockerfile). I vari servizi, considerando anche le dipendenze degli uni verso gli altri, potranno essere avviati e gestiti mediante il tool Docker Compose [18] il quale, mediante la definizione di un file YAML nel quale verranno definiti i servizi e dipendenze, gestirà il building dell'immagine fino all'avvio di tutti i servizi.

MQTT (Message Queueing Telemetry Transport) [19] è un protocollo a livello applicativo basato sul pattern publisher/subscriber, uno dei maggiori protocolli compatibili IP utilizzato in ambito IoT. Questo protocollo permette di estrarre il concetto del client e del server, permettendo l'utilizzo di questo protocollo, indipendentemente dall'infrastruttura di rete dei vari nodi. Questo permette al protocollo di poter funzionare a dispetto delle varie impostazioni NAT della propria rete o Firewall. Per ottenere questo risultato è indispensabile la presenza di un Broker MQTT, il quale avrà il compito di gestire le connessioni e di filtrare i vari messaggi in arrivo da parte dei nodi. Le comunicazioni vengono definite attraverso i topic nei quali i vari nodi potranno pubblicare (inviare) nuovi messaggi oppure sottoscriversi (ricevere) i messaggi pubblicati. Il topic è rappresentato da una normale stringa ove però è possibile utilizzare determinati caratteri speciali per definire più sottolivelli dello stesso topic.

Le tecnologie sopracitate sono state impiegate esclusivamente lato back-end ed utilizzate per tutte le peculiarità citate per ogni tecnologia. Per quanto riguarda lo sviluppo front-end del progetto è stato utilizzato il framework React assieme ad altre librerie e framework, come ad esempio Axios, React Table, React Simple Maps, React ChartJS, Bootstrap e il sopracitato Socket.IO.

ReactJS è un framework per lo sviluppo front-end compatibile con Javascript e Typescript. A differenza dei vari competitor, React si concentra sulla progettazione e lo sviluppo di più micro-componenti. React ottimizza il processo di modifica e aggiornamento del DOM mediante il virtual DOM, ovvero una rappresentazione virtuale dello stesso DOM che permette di velocizzare le normali operazioni sullo stesso DOM al costo di un maggior utilizzo di memoria [20]. React permette di definire e di utilizzare librerie per integrare nuove funzionalità al framework.

Tra le librerie utilizzate vi è Axios [21], una libreria client che permette di gestire richieste HTTP ed eventuali errori in maniera semplificata, evitando l'uso di fetch. Un'altra libreria utilizzata per la rappresentazione dei dati inerenti al progetto è stata la React Table Component [22], la quale permette di creare in maniera asincrona una tabella dinamica sulla quale vi è possibile effettuare ricerche filtrate. Un'altra libreria per la rappresentazione di queste informazioni è la React Simple Maps [23], una libreria che permette di rappresentare una mappa vettoriale mediante dei file di mappa. Infine, sempre per una rappresentazione dei dati più dinamica, sono state utilizzate le librerie Socket.IO per la ricezione dei dati in tempo reale e React ChartJS [24] per la rappresentazione grafica di questi dati. Infine, per la progettazione e lo sviluppo dell'interfaccia grafica è stata utilizzata

la libreria Bootstrap [25], un framework CSS/JS che comprende una serie di componenti già definiti e pronti all'uso e facilmente personalizzabili attraverso CSS.

Infine, per quanto riguarda la progettazione e lo sviluppo dei dispositivi e del relativo firmware, è stato utilizzato il framework Arduino assieme all'estensione per Visual Studio Code PlatformIO [26].

5 Codice

5.1 Server HTTP Gateway

Per lo sviluppo del Gateway, in particolare per l'implementazione della modalità configurazione e di un pannello Web che permetta all'utente di poter configurare il proprio Gateway, è stata implementato un server HTTP asincrono.

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS,"/index.html","text/html",false);
});

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request){
    AsyncWebParameter* ssid_param = request->getParam(0);
    AsyncWebParameter* pass_param = request->getParam(1);
    if(ssid_param->isPost() && pass_param->isPost()){
        ssid = ssid_param->value();
        pass = pass_param->value();
        // salvataggio del contenuto all'interno di un file
        File file = SPIFFS.open("/config.txt", FILE_WRITE);
        // controllo se il file sono riuscito ad aprirlo in scrittura
        if(!file){
            request->send(200,"text/plain","Error on writing file the gateway will restart in 3 seconds");
            delay(3000);
            ESP.restart();
        }
        if(file.print(ssid + "##" + pass + '\n')){
            file.close();
            request->send(200,"text/plain", "Configuration saved, the gateway will restart in 3 seconds");
        } else{
            request->send(200,"text/plain", "Error on configuration saving, the gateway will restart in 3 seconds");
        }
        delay(3000);
        ESP.restart();
    }
});
server.serveStatic("/", SPIFFS, "/");
server.begin();
```

Come è possibile osservare dal codice firmware, il server asincrono è stato gestito nel seguente modo:

1. In primo luogo è stata gestita la possibile richiesta con metodo GET alla root dello stesso server, rispondendo al client con una pagina HTML contenente un form per l'invio delle impostazioni Wi-Fi
2. In secondo luogo è stata gestita la richiesta con metodo POST alla root dello stesso server, gestendo i dati in arrivo dal client, conservandoli all'interno di un file del filesystem del Gateway e rispondendo al client in funzione del risultato.

5.2 MQTT Service

Il servizio MQTT ha il compito di monitorare i dati in arrivo da tutti i Gateway del servizio, collegandosi allo stesso broker MQTT ed eseguendo la subscription sullo stesso topic.

```
mongoose.connect('mongodb://cas-db:27017/cas-db')

.then(() => {
  console.log('connection successful to DB');
  client.on('connect', function () {
    client.subscribe('cas/sensor', function (err) {
      if (!err) {
        console.log('successful subscription');
      } else {
        console.error(err);
      }
    })
  })

  client.on('message', function (topic, message) {
    // parse del messaggio ricevuto dal sensore
    const letturaRicevuta = JSON.parse(message.toString());
    // se l'oggetto 'listasensori' contiene il valore della chiave 'id' di 'letturaRicevuta'
    SensorModel.findOne({idSensor:letturaRicevuta.id}).then((obj)=>{
      if(!obj){
        console.log('sensor not found');
      } else {
        const objRead = new ReadModel();
        objRead.idSensor = obj._id;
        objRead.temperature = letturaRicevuta.temp;
        objRead.pressure = letturaRicevuta.press;
        objRead.humidity = letturaRicevuta.hum;
        objRead.gas = letturaRicevuta.gas;
        objRead.save();
      }
    }).catch((err)=>{
      console.log('sensor not found');
    })
  })
})

).catch(() => {
  console.log('connection failed to DB');
})
```

Il codice relativo a questo servizio esegue le seguenti operazioni:

1. Tramite la libreria Mongoose prova a collegarsi al servizio database MongoDB, se questo non dovesse riuscire il servizio notificherà l'errore.
2. Se la connessione al database dovesse riuscire, il servizio proverà a collegarsi al broker MQTT, eseguendo la subscription al topic dove i Gateway pubblicheranno le varie letture.

3. Infine, il servizio gestirà l'evento di ricezione messaggio da parte del broker, controllando che il pacchetto in arrivo sia in formato JSON del progetto e successivamente utilizzerà i modelli definiti per il database conservando la lettura ricevuta.

5.3 Web Service

5.3.1 Gestione letture real-time

Per la gestione delle letture in arrivo da parte dei Gateway di un particolare sensore, è stata sfruttata la meccanica delle room [27] di Socket.IO.

```
const clientMqtt = mqtt.connect('mqtt://broker.hivemq.com:1883');
// effettuo la sottoscrizione al topic, contenente le rilevazione in base all'id sensore
const mqttConnect = () => {
  clientMqtt.on('connect', function () {
    clientMqtt.subscribe('cas/sensor', function (err) {
      })
    })

  clientMqtt.on('message', function (topic, message) {
    // parse del messaggio ricevuto dal sensore
    const letturaRicevuta = JSON.parse(message.toString());
    // creo una stanza e indico i valori delle letture
    webSocketServer.to('device:' + letturaRicevuta.id).emit('new-read', {
      idsensor: letturaRicevuta.id,
      temperature: letturaRicevuta.temp,
      pressure: letturaRicevuta.press,
      humidity: letturaRicevuta.hum,
      gas: letturaRicevuta.gas
    })
  })
}
}
```

Il codice permette al Web Service di collegarsi allo stesso broker MQTT e di gestire gli stessi messaggi che gestisce l'MQTT Service. In questo caso però i dati non verranno salvati ma verranno inviati, emettendo un evento denominato “new-read” a tutti i client Socket.IO che avranno eseguito la join in una stanza definita come “device:ID_DEL_DISPOSITIVO”.

5.3.2 Gestione routes e Express+Socket.IO

In un contesto dove è necessario far coesistere un istanza Express Server e Socket.IO Server, è stato necessario sfruttare il modulo nativo di NodeJS HTTP, gestendo le varie meccaniche di Express e l'eventuale upgrade di protocollo WebSocket di Socket.IO.

```
const sessionMiddleware = session({ secret: "cas-secret", saveUninitialized: true, resave: true, cookie: { maxAge: 60 * 60 * 24 * 1000 }, unset: "destroy" });
// mi collego al database mongodb, eseguo il middleware cors con express
mongoose.connect('mongodb://cas-db:27017/cas-db')
.then(() => {
```

```

app.use(cors({ credentials: true, origin: "http://localhost:3000", methods: ["POST", "GET", "PUT", "DELETE", "HEAD", "OPTIONS"] }));
app.use(sessionMiddleware);
webSocketServer.engine.use(sessionMiddleware);
app.use(express.static("./public"));
app.use(express.json());
app.use("/user", userRouter);
app.use("/sensor", sensorRouter);
app.use("/reads", readRouter);

mqttConnect();

webSocketServer.on('connection', (client) => {
  client.on('join', (idSensor) => {
    client.join('device:' + idSensor);
  });
});

// avvio il server node.js alla porta 8080
httpServer.listen(8080, () => {
  console.log("Server listen on port 8080");
});

).catch((err) => {
  console.log(err);
  process.exit();
});

```

Il codice principale del Web Service ha il seguente funzionamento:

1. Si definisce un middleware per la gestione della sessione utente che verrà utilizzata poco dopo
2. Si effettua la connessione al database MongoDB sfruttando la libreria Mongoose, gestendo anche un eventuale errore di connessione.
3. A connessione effettuata, si definisce una middleware globale per la gestione delle CORS [28], esclusivamente per i casi di sviluppo client di React in quanto, durante lo sviluppo, il client verrà servito da un server Web di React.
4. Si definiscono ulteriori due middleware globali, sia per Express che per Socket.IO, utilizzando la middleware di sessione creata prima
5. Si definiscono altre due middleware globali, una per servire la soluzione buildata del client React ed un'altra per gestire eventuali richieste HTTP aventi come body un messaggio JSON.
6. Definite le varie middleware globali, si definiscono le varie route sfruttando determinati router definiti per ogni categoria di richiesta e modello database.
7. Infine, si gestisce l'evento di richiesta di join in una stanza di Socket.IO da parte dei client.

5.3.3 Gestione di autorizzazione

Per la gestione delle autorizzazioni di eventuali richieste HTTP a determinate route, sono state definite ulteriori middleware da poter utilizzare nella definizione delle route all'interno dei vari router. Questa meccanica di Express permette di poter riutilizzare le stesse middleware in più route diverse, definendone anche la priorità di esecuzione in funzione dell'ordine in cui sono state inserite. Alcune di queste middleware sono le seguenti:

```

// verifica se l'utente non ha già fatto accesso, altrimenti procede
const allowNotAuthenticated = (req, resp, next) => {
  if (req.session.user) {
    resp.send({ msg: 'user already authenticated', error: false });
  } else {
    next();
  }
}

// verifica se l'utente ha già fatto accesso in tal caso procede, altrimenti restituisce un messaggio di errore
const allowLogged = (req, resp, next) => {
  if (req.session.user) {
    next();
  } else {
    resp.send({ msg: 'user not logged', error: true });
  }
}

// verifica se l'utente ha i privilegi di amministratore in tal caso procede, altrimenti restituisce un messaggio di errore
const allowAdmin = (req, resp, next) => {

  //console.log("Richiesta ADMIN ricevuta dall'utente", req.session.user);
  if (req.session.user.level == "admin") {
    next();
  } else {
    resp.send({ msg: 'permission denied', error: true });
  }
}

exports.allowNotAuthenticated = allowNotAuthenticated;
exports.allowLogged = allowLogged;
exports.allowAdmin = allowAdmin;

```

Queste middleware permettono di poter controllare se l'utente si sia autenticato o meno e soprattutto se l'utente sia un utente normale o un amministratore. Un esempio di applicazione di queste middleware si può vedere nella gestione delle API utente per il cambio di ruolo di un utente:

```

// api put per gestire il cambio ruolo per l'utente selezionato
router.put("/:idUser/role", allowLogged, allowAdmin, (req, resp) => {
  const { idUser } = req.params;
  const { newRole } = req.body;
  UserModel.findOne({ _id: idUser }).then((user) => {
    if (user) {
      user.level = newRole;
      user.save();
      resp.send({ msg: "role updated" , error: false });
    } else {
      resp.send({ msg: 'user not found', error: true });
    }
  }).catch((err) => {
    resp.send({ msg: err, error: true });
  });
})

```

Come è possibile vedere dal codice, le middleware utilizzate permettono di controllare se l'utente sia loggato e che abbia un livello di amministratore.

6 Test

In primo luogo, sono stati effettuati i primi test sui dispositivi hardware, in particolare sono state testate le seguenti caratteristiche: il raggio di comunicazione tra i due dispositivi sfruttando il protocollo LoRa, sia nel caso migliore (all'interno della stessa stanza), sia nel caso peggiore (un dispositivo indoor e il secondo dispositivo all'interno di una vettura in movimento, raggiungendo anche i 3km), la perdita di pacchetti e la banda a disposizione. Successivamente a questi test, è stato effettuato un primo test di usabilità per quanto riguarda l'installazione del Gateway. Per effettuare questo test è stato consegnato un Gateway mai acceso ad un utente, al quale gli è stato chiesto di configurarlo in modo tale che potesse collegarsi alla rete Wi-Fi domestica. Prima di effettuare il test l'utente è stato informato delle due modalità del Gateway e dei task che doveva eseguire per configurare correttamente il Gateway. Durante il primo test, l'utente ha compreso che il dispositivo Gateway, in modalità configurazione, facesse da Web server. Questo risultato è stato ottenuto in quanto il Gateway, alla connessione di un dispositivo come uno smartphone, presenta l'apertura automatica di una pagina web ove gli vengono richiesti i dati della propria connessione Wi-Fi. Durante lo stesso test l'utente ha però erroneamente inserito la password sbagliata, avviando il dispositivo in modalità operativa con una configurazione errata. Nonostante l'errore, l'utente è stato tempestivamente informato dell'errore e del riavvio in modalità configurazione mediante lo schermo LCD di cui è dotato il Gateway. Durante il secondo test, l'utente ha inserito correttamente i dati e il dispositivo si è configurato correttamente, informando l'utente che la configurazione fosse stata corretta e che il dispositivo fosse operativo, sempre mediante lo schermo LCD.

In secondo luogo, sono stati effettuati dei test tecnici riguardanti MQTT, monitorando mediante un client MQTT Web i messaggi che il Gateway dovesse pubblicare. Stabilito il corretto funzionamento di MQTT si è proceduto ai primi test sul salvataggio dei dati sul database. Durante questi test sono stati monitorati i salvataggi all'interno della collezione dati preposta e definita dai modelli di Mongoose, in modo tale che la formattazione dei dati fosse corretta e utile ai fini finali del progetto, come ad esempio il timestamp e altre informazioni.

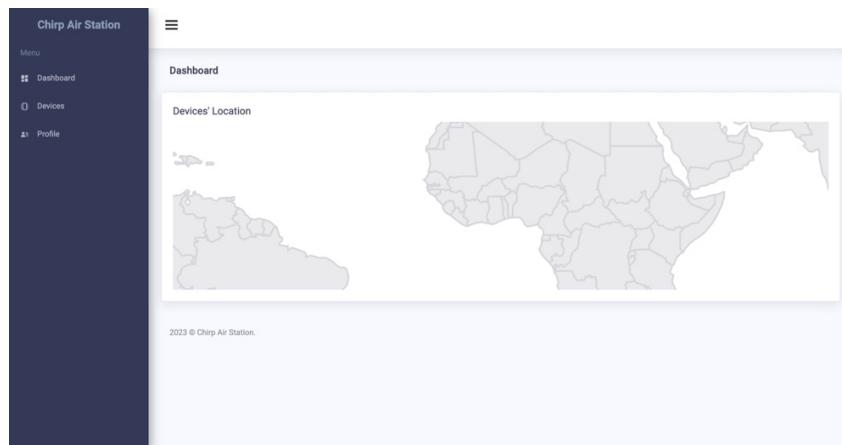
In una terza fase sono state testate le API HTTP, definite tramite Express, controllandone il reale funzionamento. Per fare ciò, abbiamo preso per noto che l'utente fosse autenticato, non implementando quindi nessun controllo di autenticazione o sessione. Per testare le API è stato utilizzato lo stesso Browser Web (per le chiamate GET) e un client apposito (Tester API, utilizzato per tutte le chiamate restanti). Convalidate le API, sono state testate le stesse implementando però i controlli di autenticazione e sessione, provando ad effettuare determinate richieste in maniera non autenticata e successivamente in maniera autenticata, sfruttando lo stesso client e i cookie generati dal server per utilizzare la sessione.

Infine, ultimato il client Web e avendo avviato tutti i servizi necessari al corretto funzionamento del progetto, sono stati effettuati una serie di test di usabilità dell'applicazione finale. Per fare ciò, allo stesso utente precedente, è stato chiesto di effettuare i seguenti task:

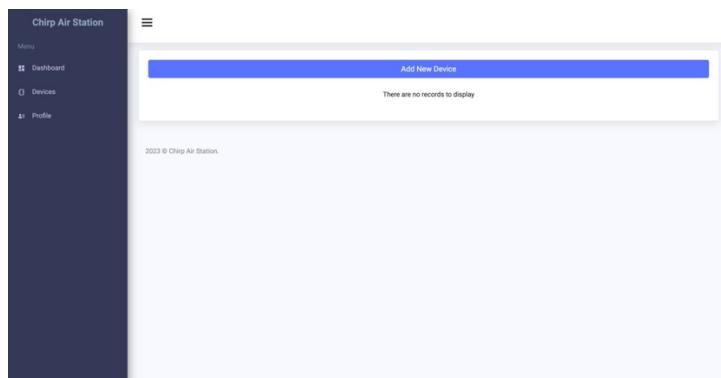
- Aprire, tramite qualsiasi strumento multimediale, la piattaforma Web
- Aperta l'applicazione Web, è stato chiesto all'utente, secondo lui, cosa richiedesse di fare la piattaforma
- Creare un nuovo account
- Eseguire il login (se necessario)
- Una volta effettuato il login, è stato chiesto all'utente, secondo lui, cosa permettesse di fare la piattaforma una volta che si è autenticato
- Registrare un nuovo dispositivo
- Monitorare in tempo reale i valori provenienti dal dispositivo
- Modificare la localizzazione del dispositivo

- Eliminare il dispositivo
- Effettuare il logout

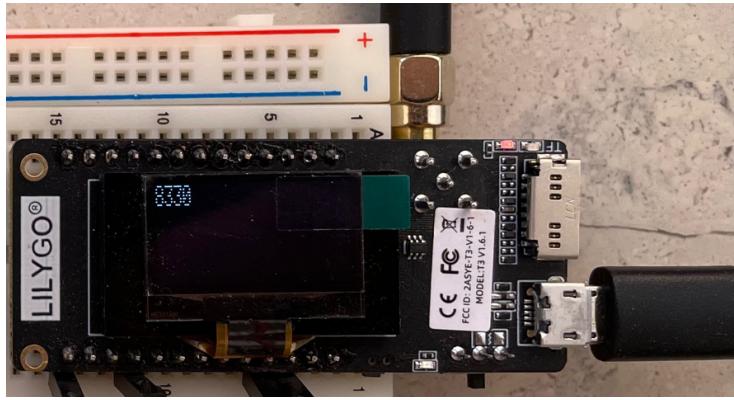
Una volta aperta l'applicazione, l'utente ha subito capito che la piattaforma richiedesse un login, portandolo quindi, in completa autonomia, a ricercare una funzionalità di registrazione la quale, è stata fin da subito trovata ed eseguita. Creato l'account, l'utente è stato informato della corretta creazione dell'account e riportato subito alla propria dashboard, facendogli comprendere che la piattaforma avesse eseguito il login automaticamente dopo la creazione.



Arrivato alla dashboard, l'utente ha subito notato il menù laterale, dal quale ha compreso subito le possibili sezioni della piattaforma, e una mappa centrale vuota. In merito a questa, l'utente si è dimostrato interessato al reale funzionamento della mappa, intuendo che nella quale andrà a visualizzare i propri dispositivi in funzione della loro posizione geografica.



Per quanto riguarda la registrazione del dispositivo, l'utente ha capito fin da subito di dover andare nella sezione denominata Devices, mostrandogli una tabella vuota che lo informava che non aveva alcun dispositivo registrato ed un pulsante visibile sopra la tabella che lo guidava nella registrazione di un nuovo dispositivo. Cliccando sul pulsante, la piattaforma ha chiesto all'utente l'ID del dispositivo che volesse registrare.



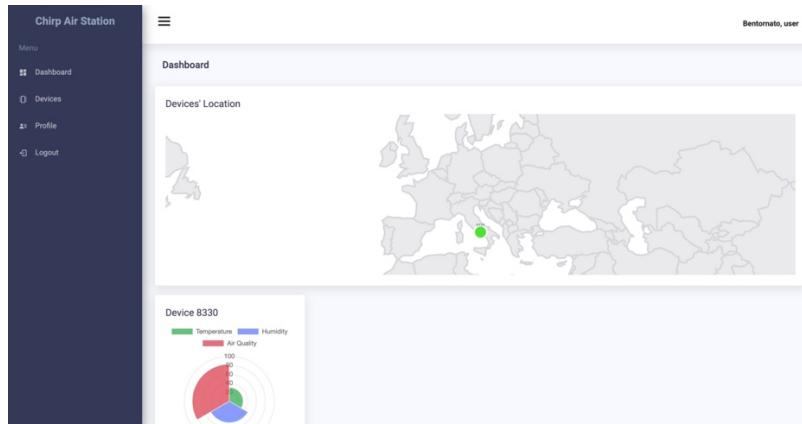
La piattaforma ha informato l'utente che l'ID del dispositivo è possibile ottenerlo all'accensione del dispositivo stesso, facendo ottenere l'informazione richiesta per la corretta registrazione del dispositivo.

idSensor	Latitude	Longitude	Actions
8330	41	14	Show Info Edit Delete

Una volta registrato il dispositivo, l'utente è stato informato della corretta registrazione facendo apparire il dispositivo nella stessa schermata, aggiungendo per ogni dispositivo una serie di pulsanti per l'interazione con esso.

Una volta che l'utente ha provato ad accedere alle informazioni del dispositivo, la piattaforma lo ha portato in una schermata dettagliata dello stesso dispositivo. Da questa schermata l'utente ha avuto modo di vedere fin da subito lo storico dei dati raccolti dal dispositivo e subito dopo, i dati in arrivo in tempo reale dallo stesso. Mediante la grafica minimale gli è stato possibile comprendere quali fossero gli ultimi dati provenienti dal dispositivo. Mediante invece il grafico a linee ha potuto monitorare il reale cambiamento dei dati. Accanto al grafico l'utente ha subito notato una mini mappa nella quale il dispositivo era localizzato, ma in una posizione errata. A questo punto l'utente, in

completa autonomia, ha provato a cambiare le impostazioni di localizzazione, settandone i giusti valori.



Prima di effettuare l'eliminazione, l'utente incuriosito è voluto tornare alla dashboard principale, trovando con stupore il proprio device appena registrato, con i dati ricevuti in tempo reale e con la possibilità di poter accedere direttamente al dispositivo mediante le schede al disotto della mappa. Una volta eliminato correttamente il dispositivo, l'utente ha proceduto per il logout.

Secondo l'esperienza utente, l'interazione con la piattaforma è stata sufficientemente semplice da utilizzare, senza troppi passaggi, riducendo i tempi delle varie operazioni dei task assegnati all'utente. Lo stesso test con gli stessi task è stato effettuato da un secondo utente, richiedendo però che venisse utilizzato il proprio smartphone. Il risultato di quest'ultimo test è stato analogo al primo, riportando un feedback positivo da ambo gli utenti.

7 Deployment

Per poter semplificare il deployment di tutto il servizio e quindi, di tutti i micro-servizi progettati necessari per il funzionamento, è stata utilizzata la tecnologia Docker. Per fare ciò sono stati definiti, per ogni servizio, un Dockerfile [29] in cui si è definita l'immagine di partenza, una serie di comandi per l'installazione di librerie e di dipendenze relative al servizio, alla creazione di un utente all'interno dell'immagine per evitare l'utenza root e una serie di comandi per l'avvio del servizio.

Una volta definiti i Dockerfile per ogni servizio sono state testate le varie immagini compilate mediante gli stessi Dockerfile, accertando così il corretto funzionamento di tutto il sistema.

Per poter semplificare ulteriormente la fase di deployment di tutti i servizi, una volta testati i vari Dockerfile, si è proceduto alla definizione di un file speciale che permetta il building delle varie immagini e l'avvio dei vari servizi, considerando anche le loro dipendenze e le loro configurazioni, mediante il tool Docker Compose.

Contenuto del file docker-compose.yaml

```
version: '3'
services:
  cas-mqtt-service:
    build:
      context: ./cas-mqtt-service
      dockerfile: Dockerfile
    image: cas-mqtt-service
    depends_on:
      - cas-database
  cas-web-service:
    build:
      context: ./cas-web-service
      dockerfile: Dockerfile
    image: cas-web-service
    depends_on:
      - cas-database
    ports:
      - 80:8080
  cas-database:
    container_name: cas-db
    image: mongo
    ports:
      - 27017:27017
```

Questo file permette a Docker Compose di definire i tre servizi del progetto. Tra i servizi troviamo il cas-database che sarà creato partendo dall'immagine ufficiale di MongoDB ed esponendo la porta 27017. Gli altri due servizi sono il cas-mqtt-service e il cas-web-service, quest'ultimo esponendo la porta interna 8080 sulla porta host 80. Per questi due servizi viene definita una dipendenza, ovvero il servizio cas-database. Questa dipendenza permette a Docker Compose di avviare questi due servizi solo se il servizio cas-database sia stato avviato correttamente. Infine, per questi due servizi viene definita anche come questi due debbano essere buildati e tramite quale Dockerfile, qualora le due immagini non dovessero essere presenti nella macchina host. In questo modo è possibile effettuare il deploy del servizio su qualsiasi sistema host o sistema cloud, potendo distribuire gli stessi servizi anche mediante tecnologie di orchestrazione come Kubernetes [30].

8 Conclusioni

Il progetto ottenuto rappresenta una piattaforma distribuita per l'analisi meteorologica e della qualità dell'aria localizzato in un contesto metropolitano. Il progetto si presenta espandibile a livello geografico mediante l'installazione di nodi sensoristici e Gateway in opportune locazioni della metropoli. Le informazioni provenienti dalla rete sensoristica vengono trasmessi mediante protocollo opportuno e adattabile a qualsiasi infrastruttura di rete che il servizio andrà ad utilizzare e conservate all'interno di opportuni database.

Oltre alla rete sensoristica espandibile, il progetto si compone anche di una serie di micro-servizi, distribuibili e deployabili attraverso tecnologia Docker. Tra i servizi, è stato progettato opportunamente un servizio che abbia il compito di ottenere le informazioni dalla rete sensoristica e di conservarle all'interno di un servizio database. Il servizio principale, il quale mette a disposizione una serie di API con le quali è possibile interagire con l'intero sistema, sfruttando anche una serie di meccaniche di autenticazione, è stato sviluppato in maniera tale e con le tecnologie necessarie affinché lo stesso servizio potesse essere scalabile, orizzontalmente e verticalmente, e distribuibile facilmente, anche su più host.

Infine, lo stesso servizio principale darà a disposizione un client Web, dotato di opportuna interfaccia grafica moderna, accessibile a qualsiasi utenza e compatibile con tutti i dispositivi multimediale, sviluppata mediante tecnologie e framework moderni, la quale dà a disposizione l'interazione completa con il sistema, compresa la possibilità di monitorare in tempo reale i propri dispositivi.

Il progetto così come sviluppato, permetterà in futuro di poter essere ampliato di nuove funzionalità, come ad esempio l'aggiunta di nuovi dati sensoristici (e quindi l'uso di ulteriori sensori) da parte della rete sensoristica, senza dover modificare ulteriormente il progetto. Inoltre, la struttura stessa del progetto ne permette la distribuzione su qualsiasi infrastruttura, che sia cloud pubblico o privato.

Il progetto è disponibile open-source al seguente indirizzo: <https://github.com/antoniorotundo2>

Bibliografia

1. <https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme680-ds001.pdf>
2. <https://www.hivemq.com/public-mqtt-broker/>
3. <https://www.mongodb.com/>
4. <https://react.dev/>
5. <https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan/>
6. https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf
7. <http://www.steves-internet-guide.com/mqtt-works/>
8. <https://nodejs.org/>
9. <https://www.npmjs.com/>
10. <https://expressjs.com/>
11. <https://www.npmjs.com/package/express-session>
12. <https://socket.io/>
13. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>
14. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
15. <https://socket.io/docs/v4/engine-io-protocol/>
16. <https://www.npmjs.com/package/mongoose>
17. <https://www.docker.com/>
18. <https://docs.docker.com/compose/>
19. <http://www.steves-internet-guide.com/mqtt-works/>
20. <https://code.pieces.app/blog/dom-virtual-dom-react>
21. <https://www.npmjs.com/package/axios>
22. <https://www.npmjs.com/package/react-data-table-component>
23. <https://www.npmjs.com/package/react-simple-maps>
24. <https://www.npmjs.com/package/react-chartjs-2>
25. <https://www.npmjs.com/package/react-bootstrap>
26. <https://platformio.org/>
27. <https://socket.io/docs/v4/rooms/>
28. <https://expressjs.com/en/resources/middleware/cors.html>
29. <https://docs.docker.com/engine/reference/builder/>
30. <https://kubernetes.io/>