POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Politecnico
di Torino

Tesi di Laurea Magistrale

Dashboard per il monitoraggio continuo della qualità ambientale

Relatore **Candidato**

Prof. Antonio SERVETTI **Luca ERRANI**

Ottobre 2022

Sommario

La rapida espansione del mercato dei microcontrollori avvenuta negli ultimi anni ha permesso la diffusione massiva di dispositivi efficienti, discretamente potenti e a basso prezzo; la capacità di calcolo e la possibilità di essere connessi a internet tramite reti senza fili, ha reso questi componenti estremamente utili per la realizzazione di sistemi di sensori distribuiti in grado di raccogliere dati e comunicarli a server di aggregazione online.

L'interesse delle aziende verso questo settore, ha permesso la rapida crescita di un mercato di sensori sempre più specifici e sofisticati, in grado di raccogliere, per quello che riguarda l'analisi della qualità ambientale, informazioni come la quantità di CO2 nell'aria, il livello di pressione sonora, il particolato presente in una stanza e vari altri fattori.

Nel corso di questa tesi verrà analizzato un caso d'uso delle tecnologie appena descritte, utilizzate per l'implementazione di una dashboard per il monitoraggio e l'analisi dei dati raccolti da vari sensori in ambienti indoor come uffici e open spaces. Verranno quindi esposti gli obiettivi della tesi, lo stato dell'arte delle tecnologie utilizzate e l'uso che ne è stato fatto per realizzare gli scopi prefissati.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Il progetto a cui il lavoro di questa tesi ha partecipato riguarda la realizzazione di una piattaforma per la raccolta, l'aggregazione e la visualizzazione dei dati sulla qualità ambientale nei luoghi di lavoro o di studio. Data l'eterogeneità dei componenti necessari al funzionamento della piattaforma, il lavoro è stato suddiviso su diversi dipartimenti del Politecnico di Torino in modo da consentire uno sviluppo rapido e parallelo su più fronti contemporaneamente.

Il lavoro è stato organizzato in modo da consentire al dipartimento di Ingegneria Informatica embedded di realizzare un prototipo di dispositivo multi-sensore in grado di ricavare dati vari ed eterogenei dall'ambiente circostante e comunicarli alla piattaforma di accentramento. Un secondo gruppo di lavoro si è concentrato sulla definizione delle metriche, delle informazioni e della frequenza con cui i dati debbano essere raccolti per poter ottenere una fotografia efficace e sempre aggiornata dell'ambiente in cui i vari multi-sensori vengono dislocati.

I dati raccolti riguardano 4 principali aree di interesse:

- **Temperatura**: i dati raccolti su temperatura attuale e umidità relativa (Relative Humidity, RH) concorrono al calcolo dell'indice di comfort per quello che riguarda l'area della temperatura

- **Illuminamento**: l'indice di comfort che è influenzato dalla qualità e quantità di illuminazione in una stanza

- **Suono**: l'analisi del livello di pressione sonora (SPL, Sound Pressure Level) permette di ricavare il relativo indice di comfort

- **Qualità dell'aria**: influenzata da vari fattori come la quantità di anidride carbonica (CO2), monossido di carbonio (CO), particolato nell'aria (PM2.5, PM10), formaldeide (CH2O) e materiali presenti nell'aria (VOC, Volatile Organic Compounds)

I fattori sopraelencati, e i relativi indici di comfort, permettono di estrarre un indice unico, aggregando le informazioni raccolte, denominato IEQ (Indoor Environmental Quality) che indica in valore percentuale la qualità effettiva dell'ambiente chiuso in cui si trova il multisensore.

Oltre alle misurazioni in tempo reale, la piattaforma deve consentire la visualizzazione dello storico dei dati raccolti su archi temporali predefiniti e aggregati in modo che i dati così ottenuti arricchiscano l'informazione mostrata all'utente finale. In particolare è richiesto che l'utente sia in grado di visualizzare lo storico delle varie metriche raccolte mediate a gruppi di 3 ore, 12 ore, 24 ore e una settimana, per monitorare l'andamento dei vari indicatori nel corso del tempo.

Gli ostacoli principali a cui porre attenzione per la realizzazione del progetto riguardano la scalabilità e la manutenibilità delle tecnologie adottate: vista l'eterogeneità di gruppi di lavoro e la durata del progetto, è fondamentale che gli approcci e le tecnologie utilizzate siano quanto più possibile standard e conosciute in modo che il lavoro possa essere proseguito da persone diverse. Inoltre, vista la quantità di dati raccolti dai sensori e il numero stesso dei sensori in campo, è fondamentale dimensionare correttamente la piattaforma al fine di garantire il funzionamento sotto stress e la potenziale scalabilità ed estensione per supportare più alti numeri di sensori e di metriche collezionate da questi.

Nel prossimo capitolo viene esposto nel dettaglio il progetto per quello che riguarda l'architettura della piattaforma e i vincoli da rispettare per la realizzazione; nei capitoli successivi vengono analizzate le tecnologie più diffuse per l'implementazione di casi d'uso analoghi e il loro utilizzo in questo particolare contesto. Infine, vengono esposte quelle che rappresentano le potenziali estensioni del progetto e gli sviluppi futuri per i quali il lavoro di questa tesi ha posto le basi.

Capitolo 2

Obiettivi della tesi

Nel seguente capitolo vengono analizzati gli obiettivi della tesi, distinguendo le varie componenti da progettare e realizzare in funzione del loro apporto alla piattaforma finale.

2.1 Progettazione della piattaforma

L'obiettivo principale della tesi, analizzato ad alto livello, è quello di creare una piattaforma in grado di gestire un insieme di sensori (o più precisamente multi-sensori, vista la loro capacità di raccogliere più metriche contemporaneamente) in grado di comunicare con un sistema centralizzato di raccolta e aggregazione delle informazioni, così che queste possano essere fruite tramite una dashboard online dall'utente finale (uno schema riassuntivo dell'architettura ad alto livello è rappresentata in figura 2.1).

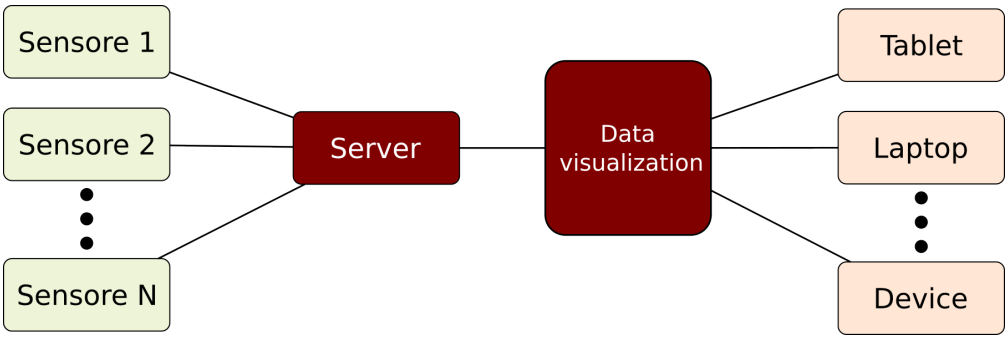

Figura 2.1: Architettura ad alto livello del sistema in analisi

Il progetto è realizzato in collaborazione con diversi dipartimenti del Politecnico di Torino, in particolare il DENERG e il , con i quali è stata progettata una prima versione della piattaforma di raccolta dei dati che ha posto i seguenti come principali obiettivi del lavoro di questa tesi:

1. Gestione della raccolta dati

   I principali fattori che concorrono alla progettazione di una piattaforma correttamente dimensionata per la raccolta dei dati dai sensori sono la quantità delle informazioni raccolte e la frequenza con cui queste vengono campionate.

   Ideare un sistema scalabile, in grado di tollerare carichi costanti di dati in entrata senza che questi impattino sul suo funzionamento, è l'obiettivo più importante da raggiungere affinché la piattaforma funzioni correttamente

2. Gestione della visualizzazione dei dati

   Una volta avvenuta la raccolta dei dati, è necessario processarli per estrarne metriche che portino valore al visualizzatore; gli algoritmi di aggregazione dei dati in informazioni complete, le modalità di scrittura di tali informazioni su disco o su basi dati opportune e l'aspetto grafico con cui queste vengono presentate, sono punti importanti affinché la piattaforma funzioni correttamente e venga utilizzata dall'utente finale che deve trarne beneficio.

3. Integrazione con il lavoro di altri colleghi

   Vista la partecipazione al progetto di diversi dipartimenti e colleghi, un altro punto fondamentale per la corretta riuscita del progetto di tesi è quello del coordinamento e della integrazione del lavoro con quello degli altri partecipanti al progetto. In particolare, il lavoro di visualizzazione dei dati proposto in questa tesi è stato integrato con quello di un'altra tesi mirata alla creazione di una interfaccia web per permettere agli utenti di rispondere a domande soggettive sulla propria percezione dell'ambiente circostante.

Capitolo 3

Stato dell'arte

In questo capitolo viene esposto lo stato dell'arte delle tecnologie correntemente utilizzate per la realizzazione di piattaforme analoghe per il monitoraggio di dati raccolti tramite dispositivi IoT. Verranno quindi illustrate le tecnologie adottate per la realizzazione di questa tesi e le motivazioni che hanno spinto alla selezione di tali tecnologie, con relativi vantaggi e svantaggi.

3.1 Comunicazione tra sensori e backend

Come analizzato nel capitolo precedente (2), la gestione della corretta scalabilità di un sistema di monitoraggio in ambito IoT risulta essere cruciale, vista la quantità e la costanza con cui le informazioni vengono trasmesse alla piattaforma di backend.

Nella realizzazione della comunicazione tra sensori e backend, il primo punto focale è proprio la gestione delle connessioni tra i sensori e il server. I principali approcci possibili per l'implementazione di questa comunicazione sono due:

- Molteplici connessioni client-server

  Il metodo più intuitivo per la gestione delle connessioni tra sensori e backend è quello di fare si che i sensori comunichino direttamente al server, tramite una connessione TCP dedicata, i dati raccolti e il server riceva i dati da tutti i sensori interessati nel sistema.

  Questo approccio, per quanto semplice, nasconde alcune problematiche che possono minare la scalabilità e l'efficienza della piattaforma. In particolare, la problematica principale risiede nella necessità di sviluppare

una piattaforma di backend che deve operare su più livelli contemporaneamente dovendo gestire la connessione con N sensori, l'accumulo dei dati e la loro aggregazione e le eventuali politiche di ri-connessione in caso di guasti alla rete. L'affidabilità del backend inoltre risulta cruciale poiché in caso di fallimenti a livello software o hardware non è concesso perdere i dati che sarebbero arrivati nel frattempo.

Un ultimo punto ma non meno importante a cui prestare attenzione è il numero di connessioni in gioco al momento del funzionamento del sistema, nel caso in cui più di un server debba ricevere i dati raccolti dai sensori. Se ci fossero 2 server a raccogliere e aggregare i dati, anche per scopi diversi, sarebbero necessarie 2 connessioni per ogni sensore, una verso un server e una verso l'altro. Se ci fossero N sensori che raccolgono dati, da trasmettere a M server, le connessioni necessarie (e quindi da proteggere, monitorare e ristabilire in caso di guasti) sarebbero $N \times M$ come graficamente rappresentato in figura 3.1

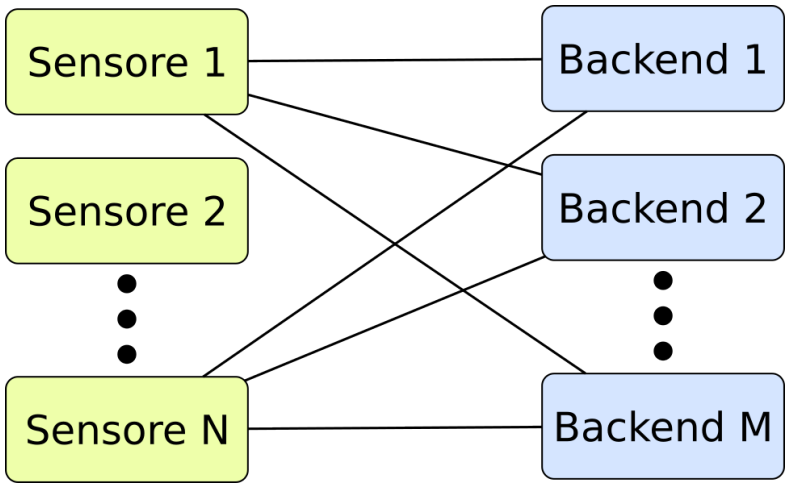

Figura 3.1: Schema per la comunicazione molti-a-molti tra sensori e server

- Meccanismi di publish-subscribe

  Una modalità più avanzata per la gestione dei flussi di dati tra sensori e backend è quella di adottare un meccanismo di publish-subscribe, implementato in un protocollo esistente, che consenta di risolvere i problemi sopracitati e semplificare l'architettura e la progettazione del sistema.

Il protocollo scelto per questo progetto è MQTT che verrà approfondito nel paragrafo successivo (3.1.1), ma che in sintesi permette di disaccoppiare la comunicazione tra sensori e backend in due fasi separate, introducendo un elemento intermediario (definito broker MQTT) che mantiene e gestisce le connessioni con i sensori e con i server.

Nello scenario proposto in precedenza dove N sensori devono comunicare con M server, l'introduzione di un broker MQTT permette di ridurre il numero di connessioni a soltanto $N + M$ come rappresentato in figura 3.2.

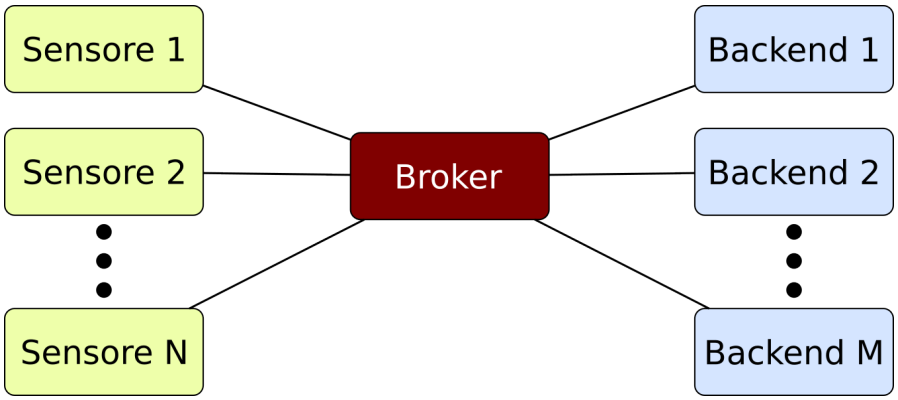

Figura 3.2: Schema per la comunicazione molti-a-molti tra sensori e server con broker MQTT

Adottando questa tecnologia, i sensori diventano *publisher* di messaggi che vengono inoltrati ai server che sono i *subscriber* a determinate categorie di messaggi, e questo scambio di messaggi viene gestito dal broker.

Un secondo grande vantaggio di questo approccio è sicuramente nella bidirezionalità della comunicazione. Con la stessa architettura è possibile rendere i sensori *subscriber* di determinati messaggi e i server *publisher*, in modo da permettere anche messaggi da backend a sensori (ad esempio per effettuare la configurazione da remoto di alcuni parametri).

3.1.1 MQTT - Message Queue Telemetry Transport

Il protocollo MQTT (il logo è in figura 3.3) è un protocollo di messaggistica di tipo publish-subscribe implementato a livello applicativo (sfruttando

Figura 3.3: Logo del protocollo MQTT

connessioni TCP). È stato inventato da Andy Stanford-Clark di IBM, e Arlen Nipper di Cirrus Link Solutions nel 1999 e il suo scopo principale è proprio quello di essere un protocollo semplice (da implementare e debuggare) e leggero in modo che possa essere adottato da qualsiasi dispositivo di qualsiasi fascia (anche low-power) senza che questo introduca overhead nel funzionamento del dispositivo stesso.

Il funzionamento di MQTT si basa principalmente sulla presenza di un broker MQTT (ne esistono diverse implementazioni anche open source) che gestisce le connessioni con i client aprendole tramite una sorta di meccanismo di handshake iniziale, per poi accettare dei messaggi di tipo subscribe a determinati *topic*. I topic in MQTT sono dei raggruppamenti logici di messaggi ai quali un client può iscriversi e ricevere tramite il broker tutti i messaggi che sono stati pubblicati da altri client su quei determinati topic. In questo modo è possibile utilizzare un solo broker per gestire diverse categorie di messaggi e fare si che ogni client che si iscrive a un topic riceva solo ed esclusivamente i messaggi ai quali è interessato. La figura 3.4 illustra un esempio di come un client possa pubblicare messaggi su due topic diversi e altrettanti client possano essere in ascolto ognuno su un topic differente e ricevere solo i messaggi opportuni.

3.2 Visualizzazione dei dati

La seconda fase del progetto di questa tesi è, come esposto nel capitolo 2, la gestione della visualizzazione dei dati raccolti.

Uno dei problemi principali che accomuna le due fasi del progetto, è proprio la mole di dati raccolti dai sensori e processati dai server. La scelta di una strategia e di un insieme di componenti adatte ad essere utilizzate

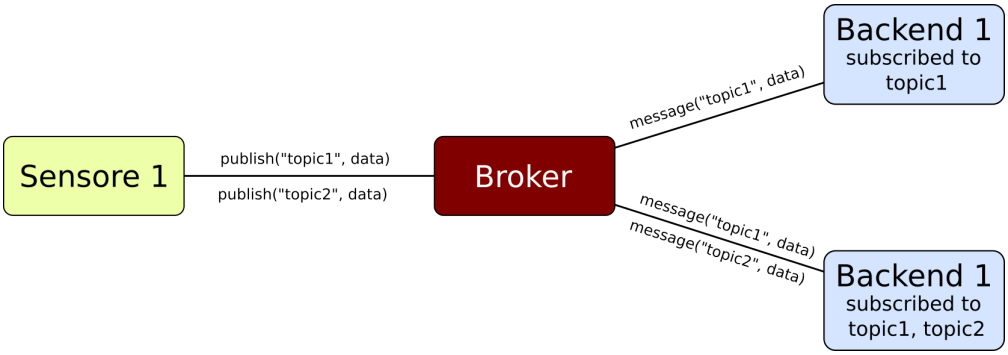

Figura 3.4: Esempio di funzionamento di MQTT con topic multipli

per un caso d'uso simile e in grado di essere manutenute nel tempo per possibili espansioni o correzioni, è di vitale importanza vista la complessità del progetto e la numerosità dei componenti in azione.

Per questi motivi, è possibile seguire due percorsi principali per la realizzazione e la gestione della visualizzazione dati in modo efficace:

- Implementazione di una dashboard ex-novo come webapp

  La prima proposta attuabile per l'implementazione di una interfaccia dashboard è quella di realizzare una web-application in grado di connettersi alle sorgenti dati tramite API e renderizzare i grafici tramite librerie JS esistenti.

  Questa soluzione presenta non poche insidie in quanto richiede di implementare anche un componente di backend apposito in grado di esporre le informazioni presenti nelle sorgenti dati (DBMS o affini) in un formato comprensibile e accessibile al frontend (ad esempio API Rest con messaggi JSON). Inoltre, per il mantenimento della piattaforma si rendono necessari sviluppatori competenti in grado di modificare il codice sorgente sia del backend che espone le API sia del frontend che manipola i dati per rappresentarli graficamente.

  Il vantaggio principale di questo approccio è sicuramente l'alto grado di libertà in termini di personalizzazione dei grafici, nel caso in cui sia necessario seguire una proposta grafica esistente da replicare.

- Uso di tecnologie esistenti per il monitoring dei dati

  L'adozione di strumenti esistenti e largamente diffusi per la creazione di grafici e piattaforme di monitoring è la soluzione più efficace per

questo caso d'uso. Il software più diffuso è Grafana, approfondito nel prossimo paragrafo (3.2.1), che consente di automatizzare il processo di ottenimento dei dati configurando le sorgenti da cui questi devono essere prelevati, semplificare la creazione di grafici tramite tool intuitivi e integrati nel software.

L'uso di strumenti conosciuti come Grafana, consente inoltre una maggiore manutenibilità del progetto, in quanto sono semplici da utilizzare per personale non tecnico e possono essere estesi e integrati da personale più tecnico grazie alla vasta documentazione che è possibile trovare online.

Una delle principali criticità nell'utilizzo di software plug-in, è la scarsa libertà in termini di personalizzazione. Alcuni strumenti permettono di modificare pochi aspetti grafici, rendendo l'integrazione del risultato su altre piattaforme più complesso.

Per riassumere i vantaggi e svantaggi delle due soluzioni, si riporta la tabella 3.1.

	WebApp ex-novo	**Soluzione esistente**
Personalizzazione	Alta, controllo totale sull'aspetto	Bassa, controllo limitato sull'aspetto
Gestione sorgenti dati	Manuale, necessaria creazione di API	Automatica, delegata al software che se ne occupa
Complessità di sviluppo	Alta, necessaria conoscenza di varie tecnologie	Bassa, soluzione plug-and-play
Testabilità	Da testare completamente	Già testata e utilizzata
Manutenibilità	Complessa, servono appositi sviluppatori	Semplice, tool grafici e soluzione standard largamente diffusa

Tabella 3.1: Pro e contro dello sviluppo della visualizzazione dati ex-novo o con soluzioni esistenti

3.2.1 Grafana

Figura 3.5: Logo di Grafana

Grafana è un software open source realizzato nel 2014 in Go e TypeScript che mira a realizzare una piattaforma unica per la connessione con sorgenti dati eterogenee e la visualizzazione dei dati raccolti tramite dashboard e pannelli. La semplice interfaccia, la quantità di sorgenti dati supportate

nativamente e la complessità dei grafici realizzabili, hanno reso Grafana il tool di riferimento per i processi di DevOps [1].

I punti di forza principali di questo tool sono i seguenti:

- Semplicità di setup e deployment

  Grafana viene rilasciato come software installabile manualmente, ma più semplicemente è possibile utilizzare le immagini Docker ufficiali per avviare un istanza base precaricata con le configurazioni di default. Questo aspetto, insieme alla possibilità di persistere i dati dei container Grafana tra un avvio e l'altro tramite l'uso di un semplice volume Docker (montato in `/var/lib/grafana`), e alla configurazione tramite un singolo file .ini da inserire nel container (sotto al percorso `/etc/grafana/grafana.ini`), rendono Grafana una soluzione facile da impostare all'avvio e da migrare da un sistema all'altro senza grandi complicazioni.

- Supporto nativo per diverse sorgenti dati

  È possibile collegare una istanza di Grafana a una o più sorgenti dati, dalle quali questo software può estrarre dati a intervalli regolari secondo le regole di chi crea nuovi grafici. L'insieme di piattaforme supportate da Grafana include i più diffusi DBMS relazionali (PostgreSQL, MySQL, Oracle Database), DB non relazionali (MongoDB, Elasticsearch) e altri servizi largamente utilizzati in sistemi di monitoring come Prometheus e Loki.

  La presenza nativa di questi connettori fa si che si riduca il numero di componenti esterni necessari al funzionamento del sistema.

- Aggiornamento automatico dei dati e dei grafici

  Una volta collegata la sorgente dati a grafana, è possibile realizzare dei grafici definendo l'intervallo di aggiornamento dei dati. Questi infatti verranno richiesti da Grafana alle sorgenti dati non appena un grafico lo richiede (quindi quando scade l'intervallo di tempo prefissato); una volta ottenuti nuovi dati, i grafici si aggiornano automaticamente.

- Gestione dell'autenticazione

  Grafana supporta nativamente svariate forme di autenticazione [2] che possono essere integrate per permettere l'accesso solo a determinati utenti e solo a determinate dashboard. È sufficiente specificare nel file

.ini di configurazione le opzioni iniziali per ogni sistema di autenticazione (documentate sul sito di Grafana) per permettere che Grafana gestisca internamente questo aspetto e non richieda ulteriori sforzi all'utente finale.

- Possibilità di embedding dei grafici

  I grafici di Grafana possono essere visualizzati direttamente sulla macchina in cui Grafana è messo in esecuzione, ma è inoltre possibile effettuare l'embed dei grafici in altre pagine, caricando il singolo grafico all'interno di un tag HTML `<iframe>` che consente di effettuare esattamente questo tipo di operazione.

In figura 3.6 è rappresentata una dashboard di esempio realizzata con Grafana, che mostra grafici ottenuti da sorgenti dati differenti e rappresentati in forma grafica eterogenea.

Figura 3.6: Esempio di dashboard realizzata con Grafana

Capitolo 4

Realizzazione degli obiettivi

Nel seguente capitolo vengono analizzate le modalità, le tecnologie e gli approcci adottati per il raggiungimento degli obiettivi della tesi, con particolare attenzione alle problematiche incontrate nel corso dell'implementazione di tale soluzione e ai modi in cui questi problemi sono stati risolti.

4.1 Architettura realizzata

Sulla base di quanto posto come obiettivo ed analizzato nel capitolo 2, la piattaforma finale è stata progettata in linea con quanto rappresentato in figura 4.1.

Dove in particolare, gli agenti in azione sono:

- Sensori

  Sviluppati dal team del dipartimento di Ingegneria Informatica Embedded, in grado di raccogliere dati su diversi indicatori come temperatura, umidità, livello di pressione sonora e qualità dell'aria. Questi sensori comunicano con la piattaforma di backend tramite il protocollo MQTT.

- Backend

  La componente di backend del sistema si occupa di effettuare la subscribe al broker MQTT che riceve i dati dai sensori, per poi storicizzarli su storage locale. Questo componente è inoltre lo stesso che effettua le prime aggregazioni sui dati e salva anche queste aggregazioni su disco.

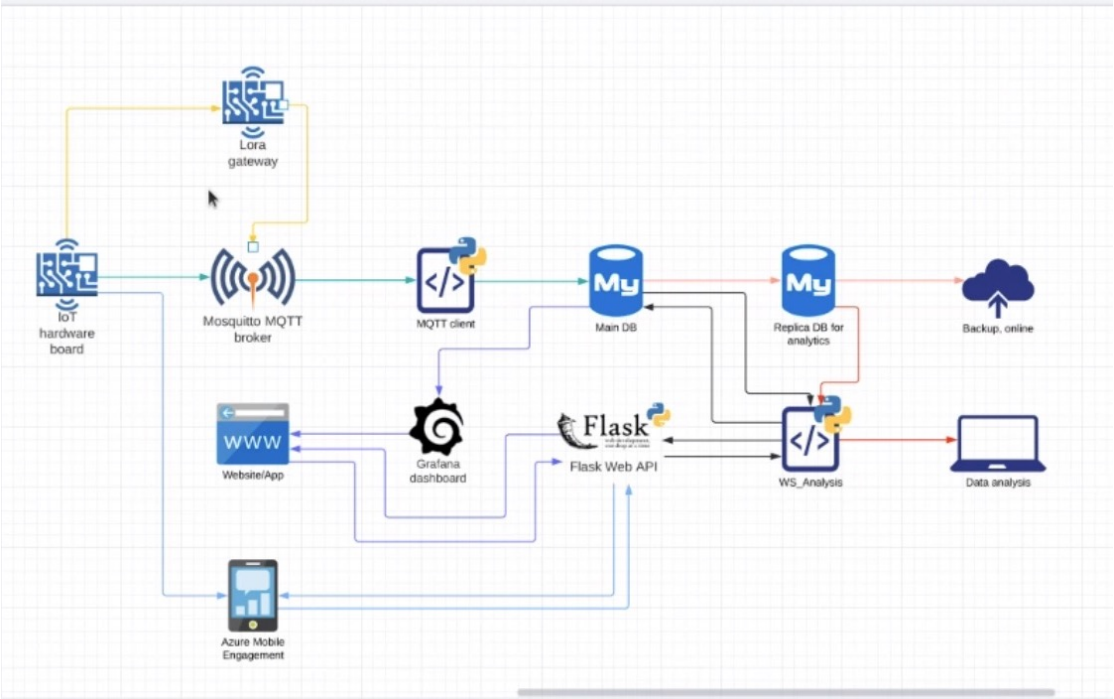

Figura 4.1: Architettura completa della piattaforma

Nella rappresentazione grafica in figura 4.1 è rappresentato da un insieme di elementi, ovvero il broker MQTT (Mosquitto nel caso di questo progetto), il client MQTT che effettua la subscribe ai topic dei sensori e il componente denominato Main DB, ovvero l'istanza MySQL per storicizzare i dati ed persisterli su disco.

- Dashboard

  Composta dal'unione di Grafana e della webapp che serve a interagire con l'utente e a mostrare i grafici finali.

Per motivi di ottimizzazione, si è deciso di replicare il DB principale in una seconda istanza, in modo da permettere al primo di ricevere i dati dai sensori e al secondo di restituire i dati storicizzati per le operazioni di aggregazione e calcolo di metriche più complesse. Questa decisione deriva dal fatto che la frequenza con cui i dati dei sensori arrivano al database e, di conseguenza, la frequenza con cui le aggregazioni e i calcoli devono essere effettuati, potrebbero interferire con le prestazioni del DB principale, rallentando l'intero sistema e possibilmente rendendolo meno affidabile.

4.2 Piattaforma per la gestione dei sensori

La prima fase del progetto di tesi è stata quella di progettare e realizzare una piattaforma per la gestione dei sensori, all'interno della quale tutti i sensori utilizzati potessero essere registrati, le loro informazioni modificate e le unità di misura e le configurazioni salvate su DB.

Nei prossimi paragrafi viene descritto come questa piattaforma è stata implementata, quali tecnologie sono state adoperate e quali criticità sono state rilevate.

4.2.1 Backend per la gestione dei sensori in Flask

La componente di backend della piattaforma per la gestione dei sensori è stata realizzata in Python grazie alla libreria Flask, che permette di realizzare webserver RESTful efficienti e che non appesantiscono le risorse del sistema.

La struttura delle tabelle con le quali il database è stato configurato è rappresentato graficamente tramite schema ER in figura 4.2

Come evidente dallo schema ER, le entità principali sono quelle rappresentate nella tabella `BOARD_TABLE`; questa tabella infatti contiene il registro di tutte le schede disponibili e registrate nel sistema, alle quali possono essere collegati N sensori fisici, ognuno collegato a sua volta con M sensori logici. Queste relazioni sono evidenziate dal collegamento molti-a-molti tra le tabelle `BOARD_TABLE` e `PHYSICAL_SENSOR_TABLE` unite dalla tabella `BOARD_SENSOR_CONNECTION_TABLE`, necessaria perché in istanti diversi di tempo è possibile collegare lo stesso sensore fisico a diverse schede, e la connessione tra `PHYSICAL_SENSOR_TABLE` e `LOGICAL_SENSOR_TABLE` per la quale il vincolo precedente non è contemplato. La differenziazione tra sensori fisici e logici è dovuta al fatto che:

- Sensori fisici: rappresentano il dispositivo fisico acquistabile che si collega alla scheda e viene gestito da questa. Ogni sensore fisico può essere dotato di più sensori logici, ad esempio un sensore fisico per il particolato presente nell'aria, può avere internamente diversi sensori logici che misurano la quantità di PM10, PM2.5, CO2 e CO separatamente.
- Sensori logici: rappresentano una entità (anche logica) che misura una singola metrica, come ad esempio la CO2 nell'aria o il PM10.

Per i motivi sopra illustrati la tabella `LOGICAL_SENSOR_TABLE` è connessa ad altre due tabelle chiamate `UNIT_OF_MEASURE_TABLE` e `MEASURE_TABLE`

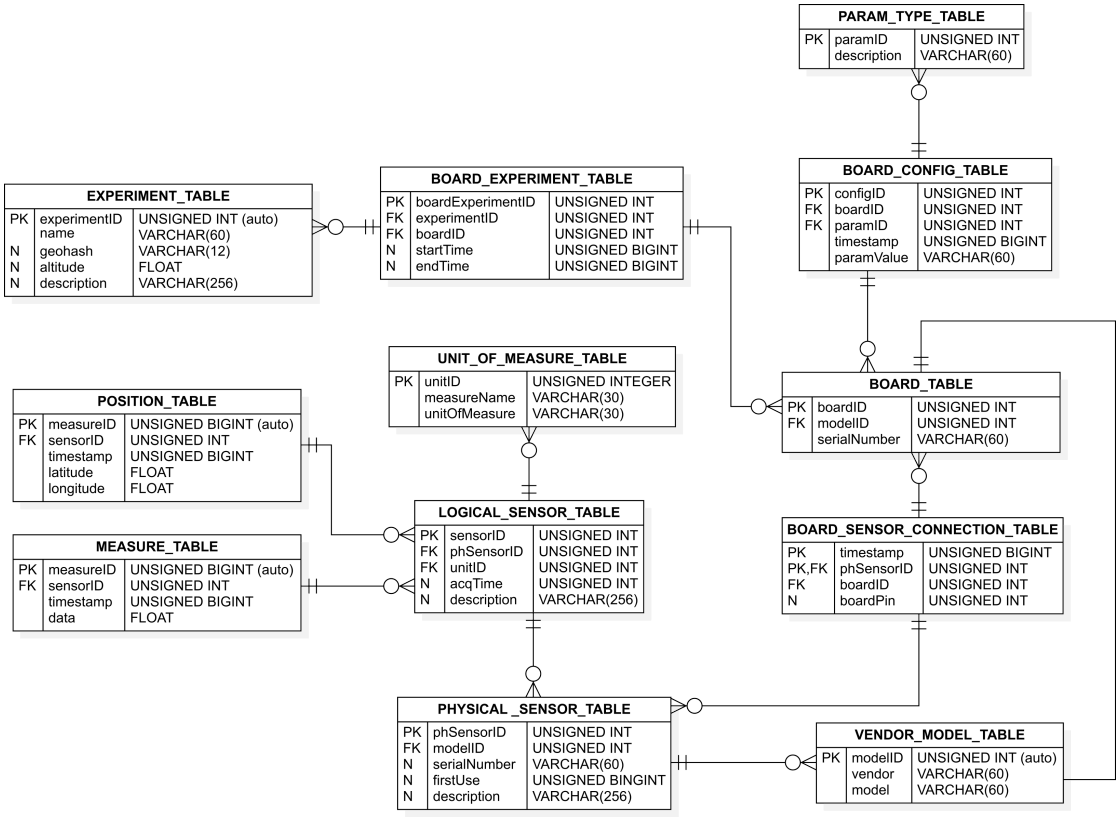

Figura 4.2: Schema ER delle tabelle realizzate per il backend in Flask

che permettono di risalire, dato un sensore logico, al tipo di misura che questo effettua (PM10, PM2.5, CO2, etc.) e allo storico effettivo delle misure effettuate da quel determinato sensore.

Per motivi di praticità, è stata aggiunta alla piattaforma la possibilità di registrare e storicizzare altre informazioni sulle schede, come ad esempio i produttori (nella tabella `VENDOR_MODEL_TABLE`) e gli esperimenti in cui una determinata scheda è stata utilizzata (nella tabella `EXPERIMENT_TABLE` collegata con una relazione molti-a-molti alla `BOARD_TABLE`).

Per quello che riguarda l'implementazione del webserver, la scelta di realizzarlo in Python utilizzando il framework Flask è stata dettata dalla necessità di utilizzare lo stesso linguaggio con cui una versione precedente di una piattaforma analoga era stata sviluppata; in questo modo sarebbe stato possibile anche per altri sviluppatori in futuro riuscire a manutenere il codice dell'intera piattaforma con un solo linguaggio.

Il principale vantaggio nell'adozione di un framework come Flask per

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello world"

if __name__ == '__main__':
    app.run()
```

Figura 4.3: Esempio di definizione di metodo HTTP GET in Flask

la realizzazione di API Restful è, oltre all'automatismo con il quale Flask associa un metodo Python a un gestore per una chiamata HTTP raffigurato in figura 4.3, la possibilità di estendere questo modulo con un componente ORM (Object Relational Mapper), che consente di astrarre la logica del backend dal sistema di storage e dal tipo di DBMS, permettendo di lavorare con queste informazioni a più alto livello in termini di oggetti e collezioni invece che di righe e tabelle.

Per questo motivo è stato immediato l'utilizzo di un DBMS embedded semplificato per lo sviluppo in locale come SQLite, per poi migrare lo stesso backend a un ambiente di produzione con un DBMS più completo come MySQL.

La libreria ORM utilizzata per questo progetto si chiama SQLAlchemy e consente, come tutti gli altri maggiori progetti ORM, di mappare automaticamente le classi Python a tabelle e colonne in SQL e gestire in autonomia lo stato degli oggetti ottenuti dal database tracciando le modifiche che avvengono su questi e riapplicandole alla riga corrispondente del database.

4.2.2 Frontend per la gestione dei sensori in React

Per permettere all'utente di interagire con le API del backend, è stato sviluppato un componente di frontend grafico in grado di essere servito tramite browser come web application.

Anche in questo caso, la decisione per quello che riguarda il framework di riferimento è stata dettata dalla necessità di rendere il software il più possibile manutenibile nel tempo da persone diverse, per questo motivo è stata scelta una libreria matura, largamente utilizzata e diffusa come React.

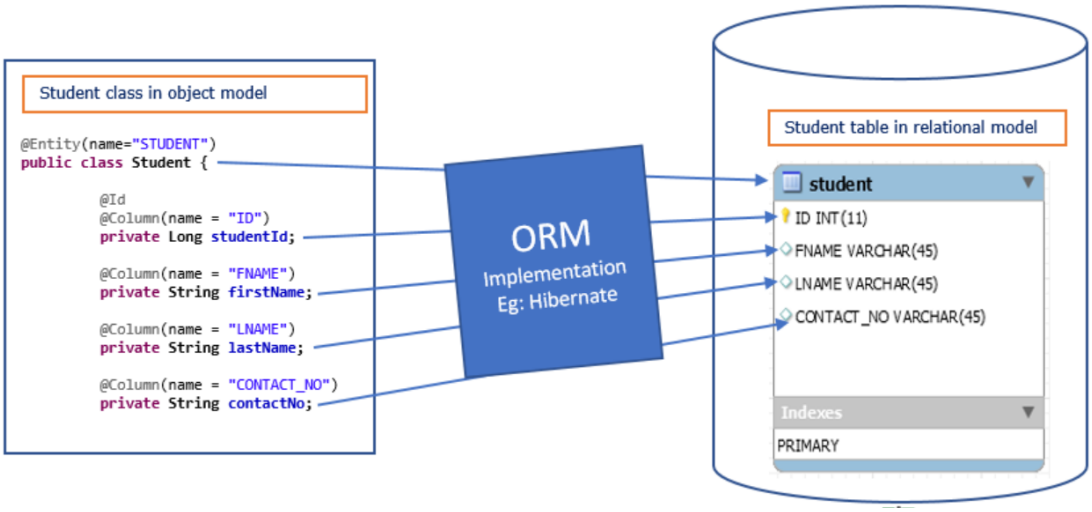

Figura 4.4: Funzionamento ad alto livello di un framework ORM

La struttura dell'applicazione è semplice poiché il suo ruolo è semplicemente quello di intermediario tra l'utente e il database. Rappresentare le informazioni salvate sul database e rendere accessibile tutte le operazioni possibili tramite bottoni e controlli grafici fa si che lo sviluppo del frontend segua pedissequamente quello del backend per quello che riguarda le entità in gioco, le loro relazioni e le operazioni che è possibile eseguire su di essi.

In figura 4.5 è possibile visualizzare la prima schermata mostrata all'utente al momento dell'avvio della applicazione.

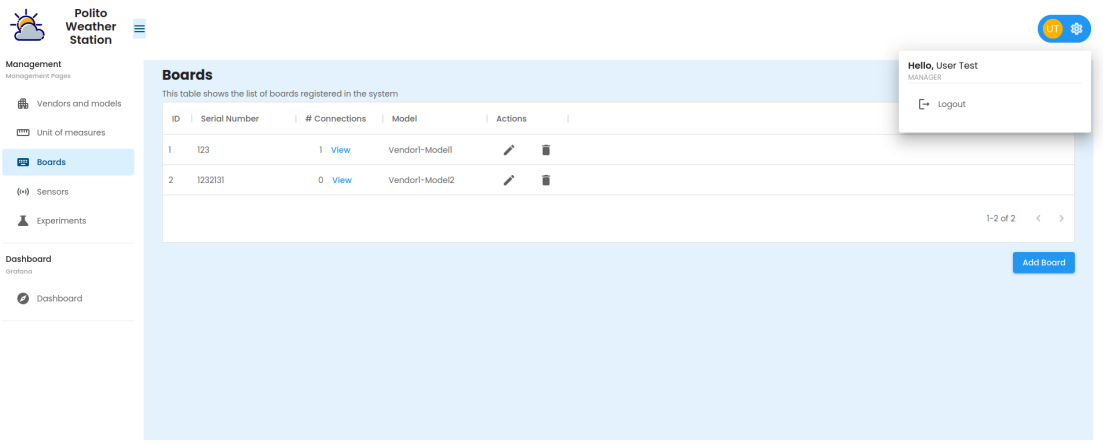

Figura 4.5: Schermata di avvio dell'applicazione per la gestione dei sensori

Tramite i pannelli visibili sul lato sinistro è possibile accedere alle informazioni presenti nelle corrispettive tabelle e aggiungere, modificare, eliminare tali elementi mappando tutte le interazioni con l'applicazione in richieste HTTP RESTful inviate al backend Flask.

4.2.3 Deployment della piattaforma per la gestione dei sensori

Per consentire un deployment leggero e replicabile di questa piattaforma, è stato deciso di creare un insieme di servizi Docker, accomunati in un `docker-compose` riportato in figura 4.6

```
version: "3.3"
services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: "frontend"
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: "backend"
    volumes:
      - "${PWD}/db:/app/db"
  proxy:
    image: nginx
    container_name: "proxy"
    volumes:
      - "${PWD}/nginx.conf:/etc/nginx/nginx.conf"
    ports:
      - "8080:80"
```

Figura 4.6: File docker-compose per il deployment della piattaforma di gestione dei sensori

In particolare, i servizi definiti sono 3:

1. `frontend`: viene compilato con un `Dockerfile` contenuto nella cartella `frontend` che parte da una immagine base di `node` che permetta di compilare il frontend React e trasformarlo in semplici file HTML, JS e CSS, per poi copiare questi file in un container `nginx` che li serve come file statici sulla porta 80.

2. `backend`: similmente al frontend viene compilato con un `Dockerfile` nella cartella di `backend` che parte da una immagine base di `python` e, senza alcuna compilazione, serve le richieste HTTP sulla porta di

default per il webserver utilizzato (che è Gunicorn, e la porta di default è la 5000)

3. `proxy`: l'unico container effettivamente esposto sulla porta 8080 dell'host è un semplice container `nginx` che riceve le richieste HTTP sulla propria porta 80 e le redirige agli altri due container sulla base dell'url: le richieste che iniziano per `/api` vengono redirette al container di backend, tutte le altre a quello di frontend.

   Nel caso in cui questo docker-compose venisse utilizzato in un ambiente di produzione, il container `proxy` è quello che si occupa della TLS termination, ovvero la procedura di cifratura delle comunicazioni per implementare il protocollo HTTPS.

4.3 Dashboard con Grafana

La seconda fase del progetto ha visto come punto fondamentale la realizzazione della dashboard vera e propria, tramite l'uso del software open source Grafana.

Per la realizzazione di questo componente si è deciso di procedere per gradi, cercando prima di tutto di avviare Grafana e comprendere i suoi meccanismi di persistenza e deployment, per poi configurarlo impostando l'autenticazione tramite JWT e infine modificare le impostazioni in modo che l'istanza di Grafana venisse messa in esecuzione su un sub-path di un dominio (e.g., non direttamente su `www.sito-grafana.it` ma su `www.sito-grafana.it/chart`).

Una volta completato questi step, sarebbe stato necessario implementare il meccanismo di embedding di Grafana su una web app secondaria sviluppata dai colleghi del progetto.

4.3.1 Setup iniziale

Essendo largamente utilizzato in ambito DevOps, Grafana viene distribuito principalmente sotto forma di immagine Docker, configurabile tramite variabili d'ambiente e file .ini. In particolare, per un caso d'uso base, è sufficiente lanciare Grafana con un docker-compose simile a quello rappresentato in figura 4.7.

Una volta dato il comando di `docker-compose up` il container chiamato `grafana` verrà creato e verrà montato all'interno di esso la cartella `grafana`.

```
version: "3"
services:
  grafana:
    image: grafana/grafana
    container_name: grafana
    volumes:
      - "${PWD}/grafana:/var/lib/grafana"
    ports:
      - 3000:3000
```

Figura 4.7: File docker-compose per il deployment di Grafana in versione base

Questa operazione di volume mount è fondamentale per fare si che le impostazioni, le dashboard e i grafici creati sul container lanciato, possano persistere in una cartella dell'host tra un avvio e l'altro della stessa immagine (questi dati verrebbero altrimenti persi).

Per accedere a questa istanza di Grafana, è sufficiente aprire un browser e navigare al sito `http://localhost:3000` sul quale è stata esposta la porta 3000 interna del container di Grafana e l'applicazione viene servita.

Per configurare le impostazioni di base di Grafana, è sufficiente creare un nuovo file allo stello livello del `docker-compose.yml` chiamato `grafana.ini` e montarlo all'interno del container aggiungendo la riga corretta ai volumi montati. Il risultato è riportato in figura 4.8

La struttura e il contenuto del file .ini sono ampiamente documentati sulla pagina ufficiale del progetto di Grafana [3] e in figura 4.9 è riportato un esempio di contenuto del file `grafana.ini` che permette anche agli utenti non autenticati di visualizzare l'applicazione creata.

Nei paragrafi successivi vengono analizzate altre opzioni che è possibile passare all'istanza di Grafana per configurarlo come necessario.

4.3.2 Autenticazione tramite JWT

Secondo la documentazione ufficiale Grafana supporta i JWT come sistema di autenticazione [2]. Per poter attivare l'autenticazione tramite JSON Web

```
version: "3"
services:
  grafana:
    image: grafana/grafana
    container_name: grafana
    volumes:
      - "${PWD}/grafana:/var/lib/grafana"
      - "${PWD}/grafana.ini:/etc/grafana/grafana.ini"
    ports:
      - 3000:3000
```

Figura 4.8: File docker-compose per il deployment di Grafana con file di configurazione esterno

```
[auth.anonymous]
enabled = true
org_name = Main Org.
org_role = Viewer
```

Figura 4.9: Contenuto del file grafana.ini per l'accesso ad utenti non autenticati

Token in Grafana è necessario passare i parametri mostrati in figura 4.10

Il file .ini nella sezione `[auth.jwt]` descrive come i token vadano validati e in quale header HTTP questi debbano essere cercati. Nell'esempio riportato in figura 4.10 nel dettaglio vengono descritti:

- `enabled = true`: attiva l'autenticazione tramite JWT (disabilitata di default)
- `header_name = X-Auth-Token`: nome dell'header HTTP in cui Grafana deve cercare il JWT. Per consentire a Grafana l'interoperabilità con diversi sistemi di autenticazione, la configurazione permette di specificare questo header in modo da delegare all'applicazione di frontend o ad un reverse proxy dedicato di riempire questo campo con il JWT corretto

```
[auth.jwt]
enabled = true
header_name = X-Auth-Token
jwk_set_file = /etc/grafana/jwks.json
username_claim = sub
email_claim = sub
auto_sign_up = true
```

Figura 4.10: Contenuto del file grafana.ini per l'accesso tramite JWT

in modo che Grafana, decodificando la richiesta, sappia dove cercare questo token di autenticazione

- `jwk_set_file = /etc/grafana/jwks.json`: il file JWKS (JSON Web Key Set) contenente un elenco di chiavi pubbliche nel formato JWK (JSON Web Key), questo aspetto viene approfondito nel paragrafo sucessivo (4.3.3).

- `username_claim = sub`: il campo del JWT che Grafana userà come username. Per ovviare al mismatch tra il sistema degli utenti interno di Grafana e quello potenzialmente esterno gestito tramite JWT, Grafana consente di specificare il campo del JWT che corrisponde all'identificativo universale dell'utente, che deve essere unico.

  È possibile specificare qualsiasi campo del JWT ma, secondo lo standard [4], il campo adatto a contenere l'informazione che identifica l'utente è il campo `sub`.

- `email_claim = sub`: un ulteriore campo del JWT attraverso il quale Grafana identificherà l'utente

- `auto_sign_up = true`: imposta Grafana in modo che un utente con JWT autorizzato venga automaticamente serializzato nel DB degli utenti di Grafana anche se non precedentemente esistente. Questa opzione risulta utile nel caso in cui i processi di autenticazione come login, registrazione e generazione dei JWT avviene tramite servizi terzi e non internamente a Grafana; in questo modo Grafana deve solo verificare che il JWT sia valido.

La versione correntemente utilizzata di Grafana (versione 9.0.1) supporta l'autenticazione tramite JWT generati esclusivamente da coppie di chiavi pubbliche e private, rendendo impossibile quindi l'inserimento di un segreto condiviso usato per firmare e allo stesso modo validare i token. Nel paragrafo suCcessivo viene analizzato il funzionamento della generazione di questi token tramite l'uso di chiavi pubbliche e private

4.3.3 Generazione dei JWKS

Gli algoritmi di cifratura asimmetrica più popolari si basano sull'uso di due chiavi, una pubblica e una privata, complementari per la generazione e la validazione delle informazioni processate. Lo standard JWK [5] descrive un nuovo formato con il quale chiavi pubbliche e private possono essere rappresentate sotto forma di oggetto JSON e raggruppate in set definiti JWKS (JSON Web Key Set).

Secondo lo standard JWK ogni chiave è provvista di un *kid* (Key ID) con il quale la chiave viene identificata; questo kid viene codificato all'interno dello stesso JWT in modo che chi ha il compito di validarlo può estrarne l'identificativo, verificare che sia stato firmato con la chiave corrispondente e permettere o meno l'accesso. Il meccanismo basato su JWKS e chiavi con un identificativo, permette la rotazione periodica delle chiavi, secondo la quale i token possono essere firmati con chiavi che cambiano periodicamente e i dispositivi che devono validare tali token possono riconoscere la sorgente della firma tramite identificativo.

Per generare un JWKS, che per l'appunto non è altro che una collezione di JWK, è sufficiente utilizzare il modulo `jwcrypto` in Python e generare la coppia di chiavi pubbliche e private nel formato richiesto dallo standard (un esempio di codice per la generazione delle chiavi è riportato in figura 4.11).

Lo script in figura 4.11 genera due file con estensione `.json` con il formato descritto in figura 4.12, uno contenente un JWKS con una chiave pubblica e l'altro un JWKS con una chiave privata:

Grazie a questi file è possibile quindi generare dei JWT codificando al loro interno delle informazioni sull'utilizzatore (grazie alla chiave privata) e validare il JWT generato verificando che sia stato creato dalla chiave privata giusta (grazie alla chiave pubblica). Come si evince dallo script, l'identificativo della chiave per questo esempio è `this-is-a-test-kid`.

Grazie allo script in figura 4.13 è possibile leggere la chiave privata creata in precedenza e generare un JWT corrispondente.

```python
from os import O_CREAT, O_RDWR
from jwcrypto import jwk
import json

key = jwk.JWK.generate(kty='RSA', size=2048, alg='RSA-256', use='sig', kid='this-is-a-test-kid')

print("PUBLIC KEY:")
print(key.export_public())
print("PRIVATE KEY:")
print(key.export_private())

public_keys = []
public_keys.append(key.export_public(as_dict=True))

payload = {'keys': public_keys}

with open("../jwks-pub.json", "w") as f:
    f.write(json.dumps(payload))

private_keys = []
private_keys.append(key.export_private(as_dict=True))

payload = {'keys': private_keys}

with open("../jwks-priv.json", "w") as f:
    f.write(json.dumps(payload))
```

Figura 4.11: Script Python per la generazione di chiavi in formato JWKS

```
{
    "keys": [
        // JWK object 1,
        // JWK object 2,
        // ...
    ]
}
```

Figura 4.12: Esempio di struttura di un file JSON contenente un JWKS

Il token generato può essere decodificato tramite un qualsiasi decoder Base64 per verificare che le informazioni contenute al suo interno siano corrette, e può essere anche validato con la chiave pubblica corrispondente.

4.3.4 Reverse proxy per la modifica delle richieste

L'inserimento di un reverse proxy tra il cliente e Grafana è necessario poiché, non avendo il controllo diretto sulla applicazione di Grafana (che è una webapp React compilata e servita sulla porta 3000), sarebbe stato impossibile fare si che il frontend inviasse in ogni richiesta l'header contenente il JWT corretto per permettere a Grafana di autenticare l'utente; in uno

```python
import os
from jwcrypto import jwk, jwt

with open("../jwks-priv.json") as f:
    keyset = jwk.JWKSet.from_json("".join(f.readlines()))

key = keyset.get_key("this-is-a-test-kid")


jwt_header = {
    'alg':'RS256',
    'kid':'this-is-a-test-kid',
}

jwt_claims = {
    'sub':'Test User',
    'roles': ['Editor'],
    'org': 'Test Organization'
}

jwt_token = jwt.JWT(
        header = jwt_header,
        claims = jwt_claims,
    )

jwt_token.make_signed_token(key)
print(jwt_token.serialize())
```

Figura 4.13: Script Python per la generazione di un JWT con chiavi in formato JWKS

scenario normale, si potrebbe avere il controllo sulla prima richiesta effettuata a Grafana, ma tutte le sucessive vengono eseguite in automatico dalla webapp di Grafana stesso, e queste richieste non includerebbero l'header di autenticazione poiché nessuno le ha configurate per farlo.

Una soluzione proposta per questa tesi, è quella di inserire nella prima richiesta il token di autenticazione non come header HTTP ma come cookie, in modo che sia il browser stesso a corredare tutte le richieste verso lo stesso dominio con lo stesso cookie, ed in seguito estrarre tale informazione dalla richiesta e posizionarla nell'header corretto grazie a un reverse proxy HTTP.

Per fare in modo che il JWT generato al paragrafo 4.3.3 sia inoltrato correttamente a Grafana, è possibile sfruttare la capacità dei proxy più diffusi di manipolare le richieste in ingresso prima dell'inoltro al vero destinatario.

In particolare, avendo scelto Nginx come reverse proxy di riferimento per

questa tesi, è stato possibile manipolare le richieste tramite la configurazione mostrata in figura 4.14.

```
server {
  listen 80;

  # Proxy Grafana Live WebSocket connections.
  location /api/live/ {
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
    proxy_set_header Host $http_host;

    # Pick the JWT from cookie and place
    # it into the correct header
    proxy_set_header X-Auth-Token $cookie_JWT_COOKIE;
    proxy_pass http://grafana:3000;
  }

  location / {
    proxy_pass http://grafana:3000/;

    # Pick the JWT from cookie and place
    # it into the correct header
    proxy_set_header X-Auth-Token $cookie_JWT_COOKIE;
  }
}
```

Figura 4.14: Estratto di configurazione Nginx per la modifica delle richieste HTTP per Grafana

Come si nota dalle direttive `proxy_set_header`, il proxy Nginx è impostato per aggiungere a tutte le richieste (websocket e HTTP) l'header `X-Auth-Token` (così chiamato anche in figura 4.10) e valorizzarlo con il contenuto del cookie chiamato `JWT_COOKIE`. In questo modo, tutte le richieste che passano dal browser al proxy verranno modificate per estrarre il cookie e posizionarlo nell'header in cui Grafana si aspetta di leggere il JWT.

Per testare questa configurazione, è sufficiente impostare il cookie `JWT_COOKIE`

nella prima richiesta fatta a Grafana dal browser (tramite i developer tools) e, siccome che questo cookie verrà inviato per ogni richiesta sucessiva, Nginx si occuperà di riscrivere la richiesta inserendo il JWT nell'header così che Grafana possa trovarlo e autorizzare l'utente all'accesso.

Per completare l'esempio, viene riportato in figura 4.15 il file docker compose completo per le operazioni descritte in questo paragrafo.

```
version: "3"
services:
  proxy:
    image: nginx
    volumes:
      - "${PWD}/nginx.conf:/etc/nginx/nginx.conf"
    ports:
      - 8080:80
  grafana:
    image: grafana/grafana
    container_name: grafana
    volumes:
      - "${PWD}/grafana-volume:/var/lib/grafana"
      - "${PWD}/grafana.ini:/etc/grafana/grafana.ini"
      - "${PWD}/jwks-pub.json:/etc/grafana/jwks.json"
```

Figura 4.15: Esempio di configurazione docker compose per il deploy del setup Nginx + Grafana

Come evidente dalla figura 4.15, ora è necessario montare all'interno del container Grafana anche il file .json contenente la chiave pubblica con cui i token verranno validati, così che questa venga letta come riportato nel file .ini di configurazione 4.10.

Al fine di migliorare l'esperienza utente, è possibile inoltre rimuovere da Grafana la possibilità di effettuare sign in e sign out, poiché queste operazioni verranno effettuate da servizi esterni e chi arriverà sulla pagina di Grafana sarà già autenticato e non potrà de-autenticarsi da Grafana. Le impostazioni che consentono questa configurazione sono riportate in figura 4.16.

```
[auth]
disable_login_form = true
disable_signout_menu = true
```

Figura 4.16: Esempio di configurazione Grafana per disabilitare il form di login e logout

4.3.5 Embedding dei grafici

Una volta configurato Grafana per gestire l'autenticazione tramite JWT ed essere protetto da un reverse proxy come illustrato nei paragrafi precedenti, è possibile procedere con l'impostazione del software per consentire l'embed dei singoli grafici in altre pagine tramite gli `<iframe>` HTML.

Per impostazioni di sicurezza di default e per evitare attacchi di tipo clickjacking [6], Grafana non consente l'uso di `<iframe>` per effettuare l'embed dei grafici in altre pagine. Questa opzione può essere invece consentita tramite l'uso della direttiva nel file ini di configurazione (4.17).

```
[security]
allow_embedding = true
```

Figura 4.17: Configurazione Grafana per permettere l'embedding

Alcuni browser tuttavia non consentono la procedura di embedding da qualsiasi sito a qualsiasi sito per motivi di sicurezza. Per questo motivo è possibile specificare tramite degli header HTTP, l'elenco dei siti sui quali è consentito l'embedding di Grafana.

Per compiere questa operazione è possibile sfruttare nuovamente il reverse proxy Nginx introdotto al paragrafo 4.3.4 per arricchire la risposta HTTP con nuove informazioni. In particolare l'opzione che consente ai browser di verificare se il sito che sta effettuando l'embedding è autorizzato, è contenuta nell'header Content-Security-Policy (CSP) [7].

Modificando il file di configurazione di Nginx aggiungendo l'opzione rappresentata in figura 4.18 è possibile specificare il dominio entro il quale il browser è autorizzato a renderizzare l'iframe.

In qualsiasi caso venga visualizzato il grafico di Grafana, tutte le richieste (anche quelle per i grafici embedded) passano sempre per il reverse proxy

```
location / {
    proxy_pass http://grafana:3000/;

    # Allow embedding only into specific webpages
    # (change mydomain.it with the correct domain)
    add_header Content-Security-Policy "frame-ancestors mydomain.it";
}
```

Figura 4.18: Esempio di configurazione Nginx per effettuare l'embedding dei grafici

Nginx e vengono inoltrate a Grafana. Per questo motivo è necessario che per richiedere tali grafici si disponga del cookie `JWT_COOKIE` esposto ai paragrafi precedenti per avere accesso.

Nei prossimi paragrafi, viene illustrato come l'uso di cookie per verificare l'autenticazione degli utenti anche per grafici embedded risulti problematico in alcuni casi, e come per questa tesi il problema sia stato risolto.

4.3.6 Cross-site cookie

Per la realizzazione della piattaforma finale, è stato inizialmente deciso di suddividere in sottodomini le due componenti principali:

- `chart.domain.it` come sotto-dominio per il deployment di Grafana e del suo reverse proxy
- `survey.domain.it` come sotto-dominio per il deployment della piattaforma di questionari e dashboard in cui effettuare l'embedding dei grafici provenienti da `chart.domain.it`

Seguendo l'architettura proposta nei paragrafi precedenti, è necessaria particolare attenzione alla gestione dei cookie, poiché per motivi di privacy, i browser più diffusi tendono a rigettare o bloccare i cookie di questo genere (chiamati cookie di terze parti) [8], poiché spesso utilizzati per scopi di marketing e per tracciare il comportamento degli utenti tra un sito e l'altro. In determinati browser, per il funzionamento di questa piattaforma, è necessario disabilitare l'opzione relativa nelle impostazioni del browser stesso.

Per fare si che l'utente possa essere autenticato sul sito `survey.domain.it` e di conseguenza visualizzare i grafici trasmessi da `chart.domain.it` senza effettuare operazioni particolari, è necessario che il primo dominio imposti il JWT nei cookie correttamente e con i seguenti parametri affinchè i browser non lo rigettino:

- `Name`: il nome del cookie, è importante che venga impostato a `JWT_COOKIE` per permettere al reverse proxy Nginx di trovarlo.

- `Value`: stringa del JWT appena creato, generato con la chiave privata generata in precedenza e la corrispondente chiave pubblica trasmessa a Grafana

- `Secure=true`: da usare su connessioni HTTPS, altrimenti il cookie viene ignorato

- `SameSite=None`: per richiedere ai browser di inviare questo cookie da qualsiasi altro sito che referenzia o incapsula il sito principale. Ad esempio, se un grafico di Grafana venisse incapsulato su un sito il cui dominio non ha niente a che fare con quello di Grafana, l'opzione `SameSite=None` comunica al browser di inviare comunque il cookie per tutte le richieste effettuate dentro all'iframe.

- `Domain=domain.it`: l'opzione forse più importante, segnala al browser che questo cookie non è da intendersi come cookie del dominio `survey.domain.it` ma del dominio `domain.it`, in questo modo il browser riterrà valido questo cookie e lo invierà insieme a tutte tutte le richieste dirette a `domain.it` e, soprattutto, a tutti i suoi sottodomini (comprendendo quindi anche `chart.domain.it`)

Per alcuni browser più vecchi, o che non rispettano l'attuale RFC sulla gestione dei meccanismi HTTP [9], è possibile che sia necessario specificare il campo `Domain` con un carattere punto (".") iniziale, per forzare il browser a inoltrare tale cookie anche per i sotto-domini, che verrebbero altrimenti esclusi.

4.3.7 Problematiche dell'infrastruttura cross-dominio

La soluzione seguita fino a questo punto, nonostante funzioni correttamente, presenta alcune criticità che è necessario risolvere per un corretto funzionamento della piattaforma e una migliore manutenibilità nel tempo del sistema. In particolare, i problemi più evidenti sono:

- Costo di gestione dei diversi domini

  Per quanto economica e semplice da impostare, la manutenzione di domini multipli non è adatta per l'utente inesperto, poiché operazioni come il rinnovo dei certificati HTTPS, il rinnovo dell'acquisto del dominio e la gestione dei record DNS che dai sotto-domini puntano ai servizi veri e propri, non sono operazioni elementari.

- Gestione dei cookie cross-dominio

  Come analizzato nei paragrafi precedenti, un cookie generato per `survey.domain.it` deve essere disponibile anche per qualsiasi altro sotto-dominio di `domain.it`. Vista la rapidità con cui gli standard di privacy e sicurezza cambiano per stare al passo con il sempre crescente numero di attacchi che sfruttano queste funzionalità, i maggiori browser potrebbero impedire di default l'utilizzo di cookie cross-dominio e restringere l'accesso ai cookie in generale, rendendo quanto fatto vano e bloccando completamente la piattaforma.

- Embedding tramite `iframe`

  Nonostante gli iframe vengano effettivamente caricati e renderizzati, l'applicazione HTML e JS che li mostra non ha accesso al DOM della finestra interna all'iframe. Questa funzionalità è bloccata da tutti i browser nel momento in cui l'origine della applicazione e quella della finestra nell'iframe è diversa (vale anche per i sotto-domini). La policy che descrive questa restrizione è chiamata Same-origin policy [10].

  Questo blocco impedisce quindi l'interazione tra l'applicazione esterna e quella interna, nonostante entrambe siano sotto lo stesso dominio ma in sotto-domini differenti, e di conseguenza potenzialmente appartenenti alla stessa azienda o progetto.

I limiti appena esposti hanno spinto a modificare l'approccio seguito fino ad ora, facendo si che invece di avere sotto-domini multipli, la piattaforma definisce un unico dominio e, tramite la differenziazione degli URL, permette

di smistare il traffico da un'applicazione all'altra. Nel prossimo paragrafo viene affrontato questo approccio e viene descritto come ciò riesca a risolvere i principali problemi sollevati fino ad ora.

4.3.8 Gestione di applicazioni multiple tramite URL

La modifica principale apportata al sistema è il modo in cui le richieste vengono smistate tra server in cui è effettuato il deployment di Grafana e il server su cui è rilasciata la web application per il questionario e la dashboard tramite embedding. In particolare sono state modificati gli URL:

- `chart.domain.it` diventa `domain.it/chart`
- `survey.domain.it` diventa `domain.it/survey`

La modifica in questi termini rende necessaria l'introduzione di un terzo elemento che funge da reverse proxy, in grado di separare le richieste per un servizio o per l'altro in funzione della route richiesta dal client. Per questo compito è stato selezionato il servizio Route 53 di AWS che, oltre ad essere un servizio DNS, permette anche di gestire il traffico sulla base degli URL.

Tramite questo sistema è possibile risolvere i problemi elencati al paragrafo precedente nel seguente modo:

- Costo di gestione dei diversi domini

  Non è più necessario gestire diversi domini, l'unico dominio è quello della piattaforma e il redirect per le due route non ha bisogno di manutenzioni una volta impostato

- Gestione dei cookie cross-dominio

  Non sono più utilizzati cookie cross-dominio, siccome che sia la piattaforma `survey` che la piattaforma `chart` sono sullo stesso identico dominio `domain.it`. I browser vedono i cookie generati su questo dominio come semplici cookies interni utilizzati dalla stragrande maggioranza di applicazioni sul web.

- Embedding tramite `iframe`

  Utilizzando un unico dominio l'origine delle pagine è esattamente lo stesso (identificato dalla tripla protocollo + dominio + porta), in questo modo viene rispettata la Same-origin policy [10] e il browser permette l'accesso al DOM interno degli iframe anche all'applicazione esterna.

L'unico problema principale di questa soluzione risiede nel fatto che Grafana, di default, serve le sue pagine React e risponde alle chiamate API come se fosse in ascolto sull'URL root (`domain.it/`) mentre invece tutte le richieste devono essere traslate e rese relative a `domain.it/chart`. Questa operazione rende necessaria la modifica sia dei file di configurazione di Grafana (che deve essere a conoscenza del sub-path sul quale viene visualizzato) sia del file di configurazione di Nginx (che deve dirigere il traffico correttamente in base ai nuovi path).

In particolare, per quello che concerne Grafana, il file di configurazione grafana.ini va modificato come rappresentato in figura 4.19.

```
[server]
root_url = %(protocol)s://%(domain)s:%(http_port)s/chart/
```

Figura 4.19: Esempio di configurazione Grafana per l'hosting in una sotto-route

Come possibile vedere in figura 4.19 è necessario specificare come parametro di avvio di Grafana il percorso sotto al quale questo servizio viene mostrato. In particolare, onde evitare di codificare nel file .ini il dominio sul quale Grafana viene messo in esecuzione, è possibile usare la *variable expansion* di Grafana per sostituire i campi salienti con i dati relativi, trasformando quindi `protocol` nel protocollo di origine della richiesta (HTTP, HTTPS, WS, WSS), `domain` con il dominio da cui proviene la richiesta (sempre `domain.it`) e `http_port` con la porta sulla quale viene effettuata la richiesta.

A chiudere, è visibile il sub-path `/chart/` che indica il termine di configurazione di cui si è parlato a inizio paragrafo, e che permette a Route53 di contattare grafana correttamente.

Per quello che invece riguarda Nginx, la soluzione è semplice e riportata in figura 4.20.

Come visibile, sono stati modificati il blocco `location` per le web socket ed è stata aggiunta una direttiva di `rewrite` per permettere a Nginx di riscrivere le richieste provenienti dal client e inoltrarle al servizio di Grafana formattate correttamente.

```
server {
    listen 80;

    # Proxy Grafana Live WebSocket connections.
    location /chart/api/live/ws {
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection $connection_upgrade;
      proxy_set_header Host $http_host;

      # Pick the JWT from cookie and place it into the correct header
      proxy_set_header X-Auth-Token $cookie_JWT_COOKIE;
      proxy_pass http://grafana:3000/;
    }

    location / {
      rewrite  ^/chart/(.*)  /$1 break;
      proxy_pass http://grafana:3000/;

      # Pick the JWT from cookie and place it into the correct header
      proxy_set_header X-Auth-Token $cookie_JWT_COOKIE;
    }
  }
```

Figura 4.20: Esempio di configurazione Nginx per l'hosting in una sotto-route

4.3.9 Deployment e integrazione con la piattaforma di questionari

Le ultime fasi della realizzazione della dashboard con Grafana hanno riguardato principalmente la messa in esecuzione del sistema, integrando effettivamente il servizio di Grafana con quello in esecuzione su `domain.it/survey` permettendo a quest ultimo di effettuare l'embedding dei grafici di Grafana e disporli come richiesto dalla proposta grafica della piattaforma.

In figura 4.21, viene mostrato il risultato della dashboard contenente l'embedding di tutti i grafici creati con Grafana e inseriti come blocchi

`<iframe>` nella webapp principale.

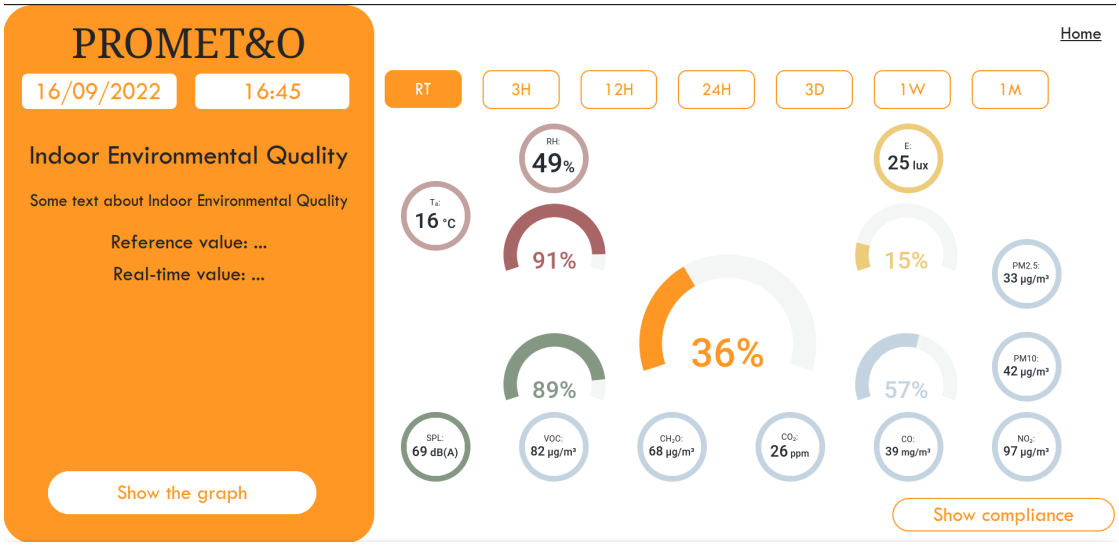

Figura 4.21: Visualizzazione finale della dashboard completa

La piattaforma risulta essere completa e interattiva; effettuare l'embedding di un grafico Grafana in un secondo sito web permette l'auto-aggiornamento dei grafici a intervalli regolari (tale intervallo è controllabile tramite il *query parameter* chiamato *refresh*) e lascia la possibilità all'utente di interagire con il grafico zoomando e accedendo col cursore all'area dei grafici di tipo time-series per leggere il valore istantaneo in un determinato punto.

4.3.10 Simulazione di sorgenti dati realistiche

Al fine di testare correttamente lo sviluppo della piattaforma nel contesto reale in cui questa verrà rilasciata, si è deciso di prepararae una proof of concept che consentisse di mostrare le funzionalità di Grafana per quello che concerne la connessione alle sorgenti dati. Per realizzare questa simulazione sono stati aggiunti due componenti al sistema, che vengono tradotti in due container aggiuntivi ai servizi gestiti dal docker compose di base, che sono rispettivamente:

- MySQL: uno dei più diffusi e conosciuti DBMS relazionali, completamente open source e anche per questo motivo ricco di connettori e librerie di

supporto per l'interfacciamento di altri programmi. Il container, per essere configurato, richiede alcune opzioni che vengono passate come variabili d'ambiente, come ad esempio `- MYSQL_ROOT_PASSWORD=<password>` che consente di specificare all'avvio la password con cui l'utente `root` può autenticarsi e `MYSQL_DATABASE=<database_name>` ovvero il nome del database con cui il DBMS verrà inizalizzato.

- FakeDataGenerator: un componente sviluppato appositamente per questa tesi e scritto in Python, con il semplice scopo di creare le tabelle iniziali nel database e, periodicamente, inserire all'interno di queste dati randomici, simulando quindi l'inserimento di tali informazioni da parte dei sensori. Il codice sorgente di questo componente è mostrato in figura 4.22.

Come visibile in figura 4.22, anche questo componente è in grado di essere configurato tramite variabili d'ambiente, attraverso le quali è possibile specificare l'indirizzo dell'host che contiene l'istanza MySQL (tramite la variabile `GNR_MYSQL_HOST`) e altri parametri di accesso come il DB a cui connettersi, l'utente con cui autenticarsi e la password corrispondente (rispettivamente le variabili `GNR_MYSQL_DB`, `GNR_MYSQL_USER` e `GNR_MYSQL_PASSWORD`). È possibile configurare ulteriormente il componente specificando con `GNR_SECONDS_INTERVAL` l'intervallo in secondi tra un inserimento e l'altro di dati casuali nel DB, mentre tramite la variabile `GNR_VERBOSE` un valore booleano che indica se il componente debba o meno stampare più log del necessario per scopi di debug e di test.

Vista la natura stessa del DBMS scelto, è necessario che le informazioni che questo salva sul proprio disco vengano persistite anche sulla macchina host nel contesto container in cui questo componente viene messo in esecuzione; per questo motivo è possibile specificare tramite il volume-mapping visibile in figura 4.23 quale cartella dell'host mappare a quale cartella del container, che per MySQL è di default la cartella `/var/lib/mysql`. In questo modo i dati inseriti e le tabelle create verranno mantenute tra un avvio e l'altro del container. Il componente di Fake Data Generator non ha bisogno di una configurazione analoga.

Per connettere Grafana a questa sorgente dati, è sufficiente aggiungere una nuova data source a Grafana tramite il menù apposito e configurarla con le informazioni con cui il DB MySQL è stato creato. In particolare è necessario specificare le informazioni di host, porta, nome utente con cui effettuare l'accesso e password. Una volta registrata la data source in Grafana, questa

```
host = getenv("GNR_MYSQL_HOST")
user = getenv("GNR_MYSQL_USER")
password = getenv("GNR_MYSQL_PASSWORD")
db = getenv("GNR_MYSQL_DB")
interval = getenv("GNR_SECONDS_INTERVAL")
verbose = int(getenv("GNR_VERBOSE"))

# Database connection and table creation omitted

add_measure_query = """INSERT INTO GENERATED_DATA(
    autoincrement, timestamp, measure)
    VALUES(DEFAULT, %s, %s);"""

while True:
    measure = random.randrange(old_value-1, old_value+1)
    timestamp = int(time.time())
    if verbose:
        print(f"{add_measure_query} = {timestamp} {measure}")
    cursor.execute(add_measure_query, (timestamp, measure))
    dataBase.commit()
    time.sleep(float(interval))
```

Figura 4.22: Codice sorgente del componente che scrive su DB dati randomizzati

è resa disponibile a qualsiasi pannello voglia utilizzarla, permettendo tramite una GUI di configurare i grafici o di specificare direttamente in SQL la query che si vuole mostrare su un determinato grafico.

4.4 Ottimizzazioni della piattaforma

Nonostante la completezza in termini di aspetto e funzionamento della pagina principale della dashboard, questa risulta migliorabile sotto diversi aspetti. In particolare, ogni elemento che rappresenta un dato (i grafici di tipo *gauge* e i valori testuali all'interno dei cerchi colorati in figura 4.21) è un elemento HTML `<iframe>` che incapsula un grafico Grafana. Questo significa che la

```
mysql:
  image: mysql
  container_name: mysql
  environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_DATABASE=main_database
  volumes:
    - "${PWD}/mysql:/var/lib/mysql"
fake-data-generator:
  build: fake-data-generator
  container_name: fake-data-generator
  environment:
    GNR_MYSQL_HOST: "mysql"
    GNR_MYSQL_USER: "root"
    GNR_MYSQL_PASSWORD: "password"
    GNR_MYSQL_DB: "main_database"
    GNR_SECONDS_INTERVAL: "6"
    GNR_VERBOSE: "1"
  depends_on:
    - mysql
```

Figura 4.23: Servizi di MySQL e Fake Data Generator inseriti nel docker-compose.yml di progetto

webapp esterna, per caricare tutti i grafici, deve caricare la stessa applicazione Grafana che permette di renderizzare il grafico per un totale di 16 volte (5 grafici di tipo gauge e 11 di tipo testuale), rallentando notevolmente il primo avvio della pagina. Nonostante tutti gli `iframe` provengano dallo stesso sito ognuno risulta essere una applicazione indipendente, che porta con sé tutti i file statici di dipendenze che ogni applicazione sviluppata in React utilizza.

Nei paragrafi successivi, vengono analizzate alcune delle possibili strategie di ottimizzazione applicabili a questo contesto al fine di alleggerire la pagina all'avvio.

4.4.1 Riorganizzazione del contenuto informativo

Il primo tentativo proposto di snellimento della pagina web è meno tecnico, ma permette di ottenere prestazioni migliori per il caricamento iniziale. Riorganizzare, insieme al team grafico che partecipa al progetto, il contenuto informativo della pagina in modo che questa all'avvio richieda a Grafana meno informazioni possibili e mostri le altre solo quando necessario. Nella pratica, la proposta è quella di nascondere tutti i grafici di tipo testuale dalla pagina lasciando solo i gauge, per mostrarli invece quando l'utente seleziona il gauge della macroarea di interesse corrispondente. In figura 4.24 viene mostrata la pagina al primo avvio (sopra) e dopo il click dell'utente sul gauge in alto a sinistra (corrispondente alle misurazioni della temperatura).

Figura 4.24: Proposta di snellimento della pagina principale della piattaforma

Come evidente dalla figura 4.24, questa proposta riduce il numero di grafici da visualizzare da 16 a 5 al primo avvio, per poi caricare solo i sotto-grafici necessari quando richiesto dall'utente. Nei casi migliori, ovvero quelli in cui l'utente seleziona i gauge in basso a sinistra e in alto a destra (dell'intensità sonora e dell'illuminazione), il grafico aggiuntivo da caricare è soltanto uno

mentre nel caso della qualità dell'aria, sarebbero comunque necessari altri 7 grafici per completare la visualizzazione di questo indicatore.

Questo tipo di modifica, ha permesso di ridurre il tempo di caricamento totale della pagina iniziale da circa 12 secondi a 5 secondi su browser Firefox installato su una macchina dotata di un Intel Core i5 di settima generazione e connessione a 1Gbps verso internet. Il caricamento degli ulteriori grafici nel caso peggiore, ovvero quello della qualità dell'aria in basso a destra, richiede altri 5 secondi.

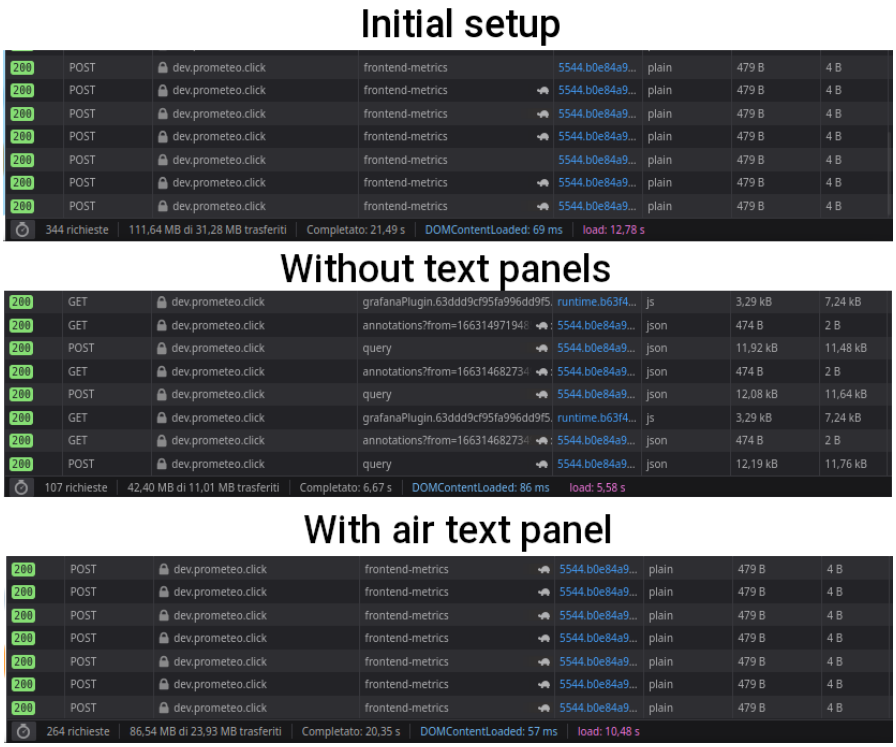

Figura 4.25: Variazione prestazioni della dashboard in funzione del numero di pannelli mostrati (a) stato iniziale (b) solo 5 pannelli (c) 5 pannelli + 7 pannelli testuali

In figura 4.25 è visibile come il numero di richieste effettuate dal browser al primo avvio, disattivando la cache del browser, vari in funzione del numero di pannelli mostrati. In particolare, nella figura in alto è visibile lo stato iniziale (344 richieste totali per 12,78 secondi di caricamento completo), le prestazioni dopo la rimozione dei pannelli testuali intorno ai gauge (107 richieste per 5,58 secondi di caricamento) e l'overhead di caricamento dei pannelli testuali

nel caso peggiore (quello della qualità dell'aria con 7 pannelli, 264 richieste per 10,48 secondi di caricamento).

4.4.2 Politiche di caching nel reverse proxy Nginx

Vista la necessità di inserire Grafana dietro ad un reverse proxy Nginx, come presentato negli scorsi paragrafi, è possibile sfruttare questa scelta progettuale per migliorare i tempi di caricamento della pagina al primo avvio. In particolare, è possibile configurare Nginx in modo che mantenga in una cache i file statici dell'applicazione React di Grafana e risponda al browser del cliente con il file salvato nella cache o con l'opportuno status code HTTP quando interpellato.

In particolare, è possibile specificare a Nginx alcuni header aggiuntivi da inoltrare insieme alle risposte inviate al browser del cliente in modo che questo non effettui nuovamente la richiesta per lo stesso identico file. Nella pratica, in figura 4.26 è stato specificato al reverse proxy di inserire negli header di risposta per i file statici (ovvero i file con estensione .js, .html e .css che essendo file di prodotto di Grafana non cambiano nel tempo) l'header `Cache-Control` che comunica al browser che la risorsa restituita non cambierà mai e che non è quindi necessario ri-scaricare tali file per altre istanze di Grafana.

Come illustrato in figura 4.26, è possibile specificare l'header `Cache-Control` in risposta ai files statici, per permettere al browser di mantenere in cache tali files ed evitare di ri-scaricarli per ogni grafico inserito negli `<iframe>`.

```
location ~* ^/.*\.(css|js|png|gif)$ {
    add_header Cache-Control "public, max-age=604800, immutable";
}
```

Figura 4.26: Configurazione Nginx per aggiungere header di cache per file statici

In particolare, quello che viene specificato in figura 4.26, è di applicare un blocco di regole a tutte le richieste che corrispondono alla *regular expression* definita nel blocco `location`, dove si richiede qualsiasi richiesta termini con estensione .css, .js, .png e .gif; a queste richieste viene aggiunto l'header

`Cache-Control` che imposta i valori seguenti suggeriti dalla documentazione Mozilla [11]:

- `public`: indica che la risposta può essere salvata in una cache condivisa, permettendo anche a tale informazione di essere salvata in eventuali proxy intermedi tra il cliente e Grafana (oltre che sul browser stesso del cliente)
- `max-age=604800`: indica che la risposta rimane valida (*fresca* nel gergo tecnico) per 604800 secondi (una settimana). Questo consente alle cache (soprattutto del browser) di fornire questa risorsa per una settimana dopo il suo scaricamento
- `immutable`: indica che la risposta non verrà aggiornata nel suo periodo di validità. Di default, i browser che hanno una risorsa in cache mandano comunque una richiesta al server per sapere se la risorsa è stata modificata nel frattempo (tramite gli header condizionali HTTP) così che il server possa rispondere anche con 304 Not Modified [12] e lasciare che il browser usi la propria cache; con l'attributo `immutable` si suggerisce al browser che non è necessaria neanche questa richiesta, poiché è garantito che la risorsa non cambierà per il suo periodo di validità

Questa soluzione ha ridotto il numero di richieste effettuate dal browser al server da circa 107 (come visibile nella seconda immagine in figura 4.25) a poco meno di 30, riducendo il carico di lavoro sul browser e sul server. In figura 4.27 è possibile vedere lo screenshot degli strumenti di sviluppo di Mozilla Firefox che indica il tempo di caricamento sceso a (in media) 4,13 secondi.

Figura 4.27: Prestazioni della dashboard in seguito alla modifica delle politiche di caching di Nginx

In figura 4.27 si vedono ancora 100 richieste, ma questo numero include quelle che sono state ottenute dalla cache. Contandole dal formato tabellare le richieste effettivamente inviate al server sono 30.

Queste modifiche tuttavia hanno valore solo in un ambiente di deployment classico dove il browser, richiedendo le informazioni, contatta direttamente il server (il reverse proxy Nginx in questo caso). Nel caso di deployment di questa applicazione, questa situazione non è replicabile in quanto il sistema DNS Route 53 di AWS utilizzato per smistare il traffico (esposto al paragrafo 4.3.8) implementa dei proxy interni che gestiscono e detengono una cache locale delle informazioni scambiate. Questo impedisce l'impostazione sul reverse proxy di particolari attributi dell'header HTTP `Cache-Control` come `immutable` che viene sempre sovrascritto e rimosso. Di conseguenza, il browser sa che l'informazione che detiene in locale è ancora valida ma, per sicurezza, contatta in ogni caso il backend per averne una controprova.

4.4.3 Richiesta diretta delle informazioni dalle sorgenti dati

Un ulteriore proposta di semplificazione e ottimizzazione dell'interfaccia grafica al fine di ridurre i tempi di caricamento iniziali, è quella di bypassare la renderizzazione tramite `<iframe>` per i grafici di tipo testuale. Nonostante siano grafici semplici, anche questi richiedono al browser il caricamento di 11 `<iframe>` (16 totali meno i 5 gauge centrali) che potrebbero essere sostituiti con delle classiche caselle di testo HTML il cui contenuto viene modificato dall'applicazione React in seguito all'ottenimento della sola informazione tramite richiesta HTTP.

Invece che incapsulare il grafico Grafana, l'applicazione React può inviare direttamente una richiesta HTTP allo stesso indirizzo usato dal grafico per aggiornarsi, in modo da ottenere solo i dati e poterli rappresentare nella maniera più adatta. In particolare, prendendo l'esempio di un singolo grafico, è possibile ottenere i dati da visualizzare inviando una richiesta di tipo POST all'indirizzo `http://domain.it/chart/api/query`, formattando correttamente la richiesta e ottenere in risposta un oggetto JSON analogo a quello rappresentato in figura 4.28.

Per brevità, in figura 4.28 sono stati omessi alcuni campi in modo da concentrare il contenuto nei punti salienti. Ogni richiesta HTTP per dei dati, deve contenere l'elenco delle time-series da visualizzare e il range temporale di interesse, in risposta Grafana fornisce un oggetto JSON contenente il campo "results" dove ogni elemento è una delle time-series richieste. In figura è espanso a titolo di esempio l'ultimo campo, chiamato VOC, che contiene informazioni sullo schema dei dati (in questo caso essendo una serie

```
{
  "results": {
    "E": {...},
    "RH": {...},
    "SPL": {...},
    "Ta": {...},
    "VOC": {
      "frames": [
        {
          "schema": {
            "refId": "VOC",
            "fields": [
              { "name": "time" },
              { "name": "VOC-series" }
            ]
          },
          "data": {
            "values": [
              [ 1663568984000 ],
              [ 3.8509331579248864 ]
            ]
          }
        }
      ]
    }
  }
}
```


Figura 4.28: JSON di risposta alla chiamata HTTP per i dati di un grafico su Grafana

temporale vengono fornite due coordinate, il tempo e il valore in un istante) e i dati effettivi sotto `VOC.data.values`.

In questo modo, è possibile bypassare gli `<iframe>` e ottenere i dati direttamente da Grafana per poterli renderizzare in React. Questa ottimizzazione, ha quindi portato a 5 il numero totale di grafici Grafana embedded, riducendo

notevolmente il numero di richieste effettuate dal browser.

Capitolo 5

Conclusioni

Nel corso di questa tesi sono state studiate, approfondite e implementate diverse soluzioni che hanno contribuito al raggiungimento degli obiettivi prefissati e alla realizzazione di un prototipo di piattaforma per la visualizzazione dei dati ambientali.

Si è posto particolare attenzione agli ostacoli incontrati durante lo sviluppo e alle metodologie e soluzioni con le quali questi ostacoli sono stati superati, per giungere ad un risultato funzionale e soprattutto costruito su tecnologie standard e largamente utilizzate, in modo da consentire il prosieguo di questo lavoro partendo da basi comuni di conoscenze tecniche.

Per quello che riguarda in particolare il lavoro di questa tesi, la scelta di un software come Grafana ha consentito la prototipazione rapida di grafici e visualizzazioni dei dati e la sua modularità ha permesso di effettuare, con le dovute modifiche e configurazioni, l'embedding di tali grafici in altre pagine web. L'utilizzo di Grafana è risultato fondamentale per quello che riguarda l'implementazione di un layer intermedio tra i database che mantengono i dati salvati e la loro visualizzazione, esportando tali informazioni in due modi distinti:

- Tramite grafici nativi Grafana: che hanno permesso di prototipare i grafici principali della piattaforma raffiguranti gli indici di comfort e l'IEQ
- Tramite API REST: utile per l'accesso diretto ai dati per la loro rappresentazione in modalità esterna a Grafana, come nel caso dei grafici testuali che sono facilmente replicabili in React e appesantiscono l'interfaccia se ottenuti tramite l'embedding di Grafana

Il lavoro di questa tesi ha creato la base di partenza per la realizzazione della piattaforma completa; grazie alle tecnologie standard utilizzate questo lavoro può essere esteso e proseguito in modo da rifinire le funzionalità presenti ed aggiungerne eventualmente in futuro. In particolare, alcuni settori di questo progetto possono essere ulteriormente sviluppati, come ad esempio l'inserimento all'interno dei grafici dei dati soggettivi raccolti tramite questionari, da visualizzare in super-imposizione ai grafici rappresentanti i dati oggettivi e che consentirebbero di estrarre interessanti correlazioni tra quanto venga effettivamente misurato dai sensori e come invece l'ambiente viene soggettivamente percepito dai diversi utenti.

Un ulteriore sviluppo riguarda l'estensione dello stesso Grafana per aggiungere funzionalità non presenti nativamente ma potenzialmente utili a questo progetto; in particolare, la modifica di come i grafici a barre vengono renderizzati per poter supportare dati riguardanti la varianza dell'informazione mostrata e la visualizzazione di aree colorate sullo sfondo per rendere visibile al primo sguardo se i valori misurati rientrano in determinate soglie o meno (un esempio di tale grafico è riportato in figura 5.1).

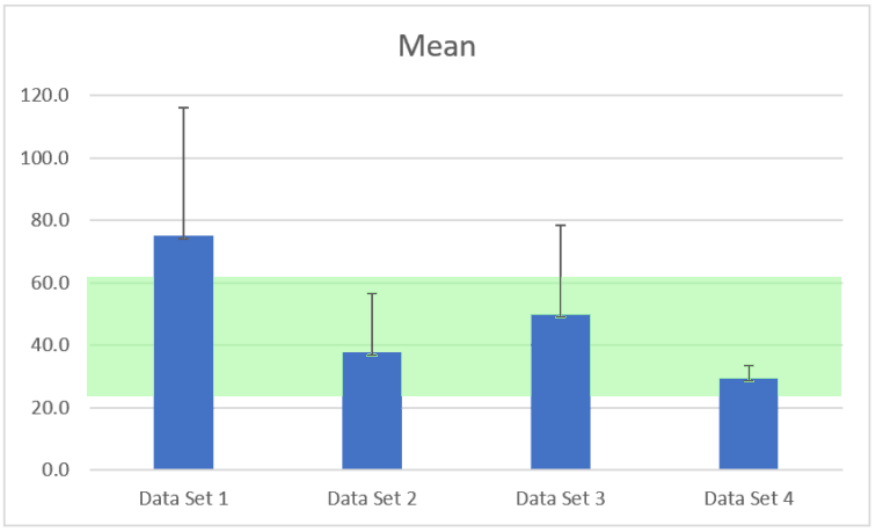

Figura 5.1: Grafico a barre con rappresentate deviazione standard e soglie sullo sfondo

Ulteriori sviluppi futuri riguardano l'ottimizzazione del motore di Grafana che permette di effettuare l'embedding dei grafici, eventualmente rimuovendo completamente tale funzionalità in favore di grafici realizzati su misura

tramite l'uso di librerie come Chart.js [13] o similari, utilizzando Grafana come tramite per l'interfacciamento con le differenti data sources e l'esposizione dei dati tramite API REST, che consentono appunto di svincolare lo sviluppo dell'interfaccia grafica da quello della connessione con i DB.

Bibliografia

[1] George Anadiotis. *DevOps and observability in the 2020s*. Gen. 2020. URL: https://www.zdnet.com/article/devops-and-observability-in-the-2020s/ (cit. a p. 12).

[2] Grafana Labs. *Supported authentication methods for Grafana*. URL: https://grafana.com/docs/grafana/latest/setup-grafana/configure-security/configure-authentication/ (cit. alle pp. 12, 23).

[3] Grafana Labs. *Grafana Documentation*. URL: https://grafana.com/docs/grafana/latest/ (cit. a p. 23).

[4] Internet Engineering Task Force (IETF). *JSON Web Token (JWT)*. URL: https://www.rfc-editor.org/rfc/rfc7519#section-4.1.2 (cit. a p. 25).

[5] Internet Engineering Task Force (IETF). *JSON Web Key (JWK)*. URL: https://www.rfc-editor.org/rfc/rfc7517 (cit. a p. 26).

[6] Gustav Rydstedt. *Clickjacking*. URL: https://owasp.org/www-community/attacks/Clickjacking (cit. a p. 31).

[7] Mozilla. *Content Security Policy (CSP)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP (cit. a p. 31).

[8] Mozilla. *Disable Dynamic State Partitioning*. URL: https://developer.mozilla.org/en-US/docs/Web/Privacy/State_Partitioning#disable_dynamic_state_partitioning (cit. a p. 32).

[9] Internet Engineering Task Force (IETF). *HTTP State Management Mechanism*. URL: https://www.rfc-editor.org/rfc/rfc6265#section-4.1.2.3 (cit. a p. 33).

[10] Mozilla. *Same-origin policy*. URL: `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy` (cit. alle pp. 34, 35).

[11] Mozilla. *Cache-Control*. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control` (cit. a p. 45).

[12] Mozilla. *HTTP response status codes*. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status` (cit. a p. 45).

[13] Chart.js community. *Chart.js*. URL: `https://www.chartjs.org/` (cit. a p. 51).