POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

PROMET&O: Piattaforma web in ambiente IoT per il monitoraggio del comfort ambientale misurato e percepito

Relatore
Prof. Antonio Servetti

Candidata
Sara Bellatorre

Anno Accademico 2023-2024

Ringraziamenti

A mia mamma, che ha sempre visto in me la luce anche quando io riuscivo a vedere solo buio. Che ha visto una farfalla anche quando ero solo un piccolo bruco. Grazie per spingermi ogni giorno verso la versione migliore di me stessa, quella che non so neanche io di avere.

A mio papà, che non ha mai dubitato delle mie capacità. Grazie per aver gioito con me e più di me dei miei successi, anche i più piccoli, e di avermi dato coraggio e sicurezza davanti ai fallimenti, non tentennando mai. Sei sempre riuscito a farmi sentire la persona giusta, nel posto giusto, nel momento giusto.

A Flavia, la mia roccia eterna, la mia compagna per la vita, il regalo più grande e più bello che i miei genitori potessero farmi. Grazie per riuscire sempre a tirarmi su, anche nei momenti di sconforto più totale, d'altronde assieme a te *la vita no, non sembra mai dura.*

A Bernard che ha reso questi ultimi sei anni i più belli della mia vita. Molto probabilmente senza di te io, questa tesi e la pergamena ci saremmo incontrati molto più in là. Grazie per essermi stato affianco sia nel buono che nel cattivo tempo, per aver creduto in me sotto ogni punto di vista e soprattutto grazie per essere in grado di interpretare ogni bit del mio cervello e del mio cuore.

A Matteo, compagno di migliaia di progetti, uomo dei fattoidi ma, prima di tutto, amico fidato. Grazie per essere una spalla su cui so di poter sempre contare e per aver condiviso con me sia LADISPErazione che gli scleri scolastici - e non - di questa magistrale.

A Luca, a cui voglio un bene proporzionale alla frequenza di prese in giro. Chi l'avrebbe mai detto che da quelle del primo anno sarebbe nato un legame così bello? A proposito, colgo l'occasione per chiederti: ma l'hai mai visto Star Wars?

Sommario

L'Indoor Environmental Quality (IEQ) svolge un ruolo fondamentale nella promozione della salute e del benessere degli occupanti degli edifici. PROMET&O si presenta come una soluzione progettata per monitorare dettagliatamente l'IEQ, sia da un punto di vista oggettivo che soggettivo.

La piattaforma sfrutta un insieme di sensori IoT per raccogliere i dati oggettivi e correlarli a quelli soggettivi ottenuti tramite questionari compilati dagli utenti. Questo approccio consente di ottenere una panoramica completa e approfondita della percezione dell'ambiente e di confrontarla con una misurazione oggettiva dello stato dello stesso.

Inizialmente implementata come soluzione serverless su AWS, PROMET&O è stata migrata in un ambiente locale. Questa nuova configurazione contiene la versione aggiornata dell'applicazione, basata su un backend Express e un frontend React, interamente dockerizzata e integrata in una rete di container. L'utilizzo di questa rete garantisce l'indipendenza da servizi esterni attraverso un API gateway che consente di gestire autonomamente aspetti critici come l'autenticazione e l'autorizzazione degli utenti tramite, rispettivamente, un authorization server e un identity provider.

All'interno di questa infrastruttura, la piattaforma di analytics Grafana riveste un ruolo centrale nel recupero e nella visualizzazione, sia real-time che su intervalli di tempo, dei dati provenienti dai sensori.

PROMET&O rappresenta una soluzione versatile e facilmente replicabile per il monitoraggio del comfort ambientale indoor. La sua flessibilità e modularità la rendono ideale per un'ampia gamma di contesti, da grandi aziende a piccoli uffici.

Indice

Introduzione e obiettivi

PROMET&O è una piattaforma web che mira al monitoraggio e alla raccolta dati dell'Indoor Environmental Quality (IEQ). L'analisi dell'IEQ avviene tramite due principali processi: il collezionamento dei dati oggettivi tramite multisensori e l'analisi della percezione ambientale soggettiva mediante sondaggi posti agli utenti. Tramite la dashboard è possibile visualizzare la correlazione tra i due tipi di dati raccolti e ottenere un quadro completo delle condizioni ambientali al fine di migliorarle.

Inizialmente PROMET&O era composta da un'applicazione web React e da un backend di tipo serverless basato sui servizi offerti da Amazon Web Services (AWS). Se da un lato ciò era vantaggioso per la completezza e varietà dei servizi, dall'altro diminuiva il controllo diretto da parte dell'Istituto sul sistema a fronte di una spesa maggiore.

Si è deciso, quindi, di trasformare PROMET&O in una piattaforma web locale, ospitata dai server del Politecnico di Torino. L'applicazione web di partenza è ora inserita in un contesto di microservizi implementati tramite una rete di container Docker. Questo garantisce non solo il pieno controllo della piattaforma, ma anche una maggiore modularità della stessa.

Nel corso di questa tesi si andranno ad analizzare i vari passaggi che sono stati necessari per raggiungere gli obiettivi preposti, ovvero la migrazione dai servizi sopracitati e l'integrazione dell'applicazione web all'interno di una solida rete di container.

In seguito, verranno analizzate alcune nuove funzionalità dell'applicazione web come il supporto a questionari multipli, la visualizzazione dei dati di diversi sensori nella dashboard e il dump delle risposte ai questionari in un file Excel.

Capitolo 1

Architettura precedente

1.1 Descrizione architettura

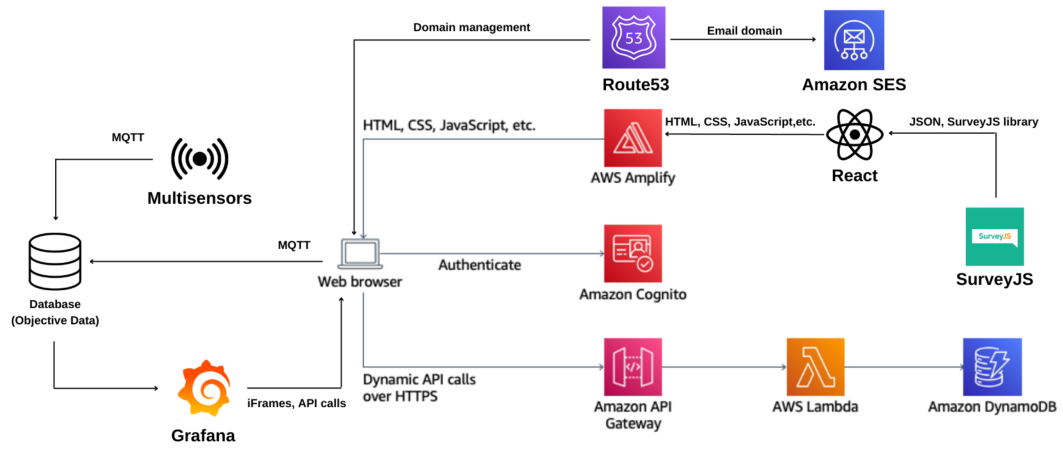

Figura 1.1. Architettura precedente

Inizialmente l'architettura della piattaforma PROMET&O sfruttava diversi servizi AWS.

Amazon API Gateway fungeva da porta d'ingresso, ricevendo le richieste HTTP degli utenti che venivano poi inoltrate a funzioni serverless gestite da AWS Lambda.

I dati venivano archiviati e recuperati tramite Amazon DynamoDB, un database NoSQL altamente scalabile.

Per semplificare lo sviluppo, è stato utilizzato Amplify CLI, strumento che consente di creare facilmente le API e le funzioni Lambda necessarie per interagire con il database, permettendo di eseguire operazioni CRUD (Create, Read, Update, Delete) sui dati.

La sicurezza era garantita da Amazon Cognito, che gestiva le sessioni utente e l'autenticazione.

Nel corso di questa tesi, sarà esaminata la trasformazione radicale della piattaforma, passando da un'architettura serverless a una basata su microservizi implementati localmente tramite una rete di container Docker.

La migrazione della piattaforma da un ambiente serverless a uno locale ha portato numerosi vantaggi. Prima di tutto, ha eliminato le dipendenze da servizi esterni, consentendo ai futuri sviluppatori studenti di utilizzare le tecnologie apprese durante i loro studi anziché dover affrontare nuove tecniche da zero.

Un ulteriore vantaggio è che tutto il processo è gestito internamente dal Politecnico di Torino. In precedenza, la complessa autorizzazione AWS richiedeva tempo e risorse aggiuntive, mentre ora, con l'ambiente locale, il controllo completo risiede all'interno dell'istituto. Questo non solo semplifica la gestione, ma riduce anche la dipendenza da fornitori esterni, garantendo un maggiore controllo e sicurezza sui dati e sulle risorse dell'applicazione.

Inoltre, con la separazione dei due domini, l'applicazione React non si occupa più della fase di autenticazione, che è gestita da un microservizio dedicato, liberando il frontend da questo carico e semplificando il suo sviluppo e manutenzione.

Infine, l'abbandono dell'ecosistema AWS ha comportato significativi risparmi, poiché l'intera infrastruttura può ora essere ospitata sui server del Politecnico di Torino, eliminando la necessità di sottoscrivere abbonamenti a servizi di terze parti.

1.2 Migrazione da AWS

Il primo passo della migrazione da AWS ad un contesto locale è stata la rimozione del servizio Amazon Amplify. Quest'ultimo viene utilizzato come un'interfaccia per effettuare richieste al backend: nello specifico si occupa di risolvere i domini utilizzando quelli dei server di Amazon. Per ottenere ciò, si è effettuato manualmente ciò che eseguiva Amazon Amplify, traducendo l'URL utilizzato dalle API in modo tale da rendere l'applicazione indipendente.

Una volta effettuata la rimozione di Amplify, è stato sviluppato un backend Express che supportasse tutte le API localmente. Nel processo di creazione del backend, al fine di migliorare la leggibilità e la modularità del codice, è stato creato un nuovo file, `API.js`, contenente tutte le API, così da raggrupparle e facilitarne la consultazione e l'eventuale modifica. Durante lo sviluppo del server, si è preferito un approccio standard con il frontend in esecuzione sulla porta 3000 e il backend sulla porta 5000.

Le prime API ad essere convertite sono state quelle necessarie per la gestione e il recupero delle risposte alle due tipologie di questionari: le domande di carattere personale (dati anagrafici, occupazione...) e il questionario soggettivo (per la valutazione del benessere ambientale). Le rimanenti API, riguardanti principalmente l'autorizzazione e l'autenticazione degli utenti, sono state gestite da uno dei microservizi di cui si tratterà successivamente. (3 - *AuthServer*)

Il database, inizialmente non relazionale di tipo key-value (DynamoDB), è stato ora realizzato tramite MySQL. Si è optato per questo approccio in quanto la semplicità dei dati non richiedeva una struttura complessa come quella di un database non relazionale.

Una volta effettuati i test sul buon funzionamento della migrazione, l'intero client (frontend React e backend Express) è stato messo in esecuzione sulla porta 5000 per vincoli imposti dalla rete di container.

A questo punto è stato possibile svincolare il codice di autenticazione e autorizzazione dal client, rendendo entrambe le parti più leggere e facili da mantenere.

Nella seguente tabella sono riassunte le trasformazioni dei diversi servizi presenti nella piattaforma.

Precedente	**Attuale**
Amazon API Gateway	API Gateway Nginx
Amazon Cognito	OAuth Server e Keycloak
Amazon DynamoDB	MySQL
Amazon SES	Keycloak

Tabella 1.1. Migrazione dei servizi

Capitolo 2

Descrizione della piattaforma PROMET&O

2.1 Ingress + nsXX

PROMET&O è stata progettata con un'architettura che pone al centro un server di ingresso Nginx, che funge da punto di accesso principale per la rete. Questo server svolge un ruolo fondamentale nella gestione del traffico in arrivo dai client e instradare le richieste ai server web appropriati.

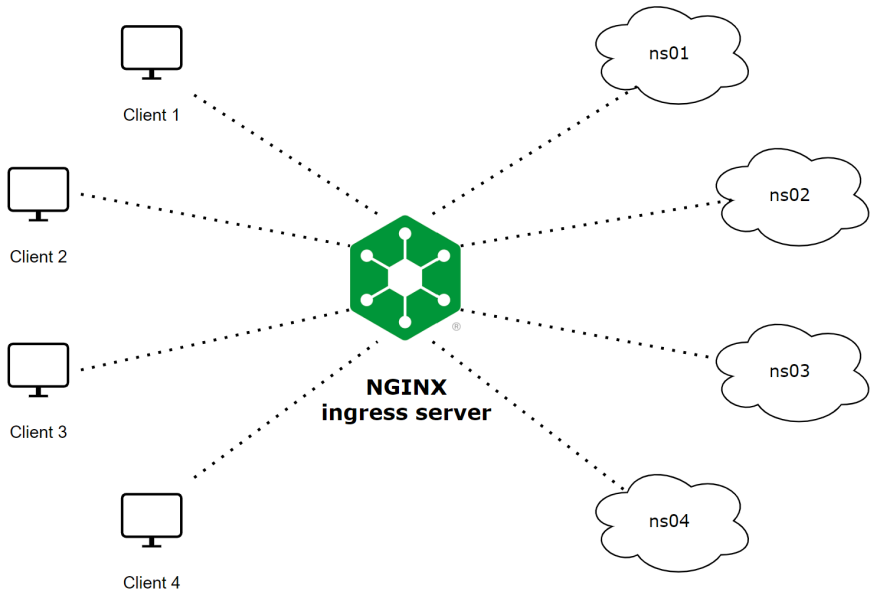

Figura 2.1. Sistema complessivo

Ogni sottorete presenta un livello di separazione del sistema di autenticazione e autorizzazione dalla singola applicazione web. Ciascuna di queste sottoreti può ospitare qualsiasi tipo di applicazione web, l'unico requisito è che sia dockerizzata e attiva sulla porta 5000. Nel nostro contesto, ogni `nsXX` presenta una replica dell'applicazione web base, PROMET&O. Questo porta a vantaggi significativi, come per esempio la separazione dei domini utente (assegnare ruoli diversi allo stesso utente) o avere delle versioni che contengono delle funzionalità diverse in base al contesto d'utilizzo.

2.2 Descrizione di nsXX

Nella prossima sezione si andrà ad analizzare la rete di container ospitata all'interno di un generico `nsXX`. All'interno di ciascuna sottorete sono presenti sei container, ognuno dei quali svolge un ruolo specifico nell'ecosistema

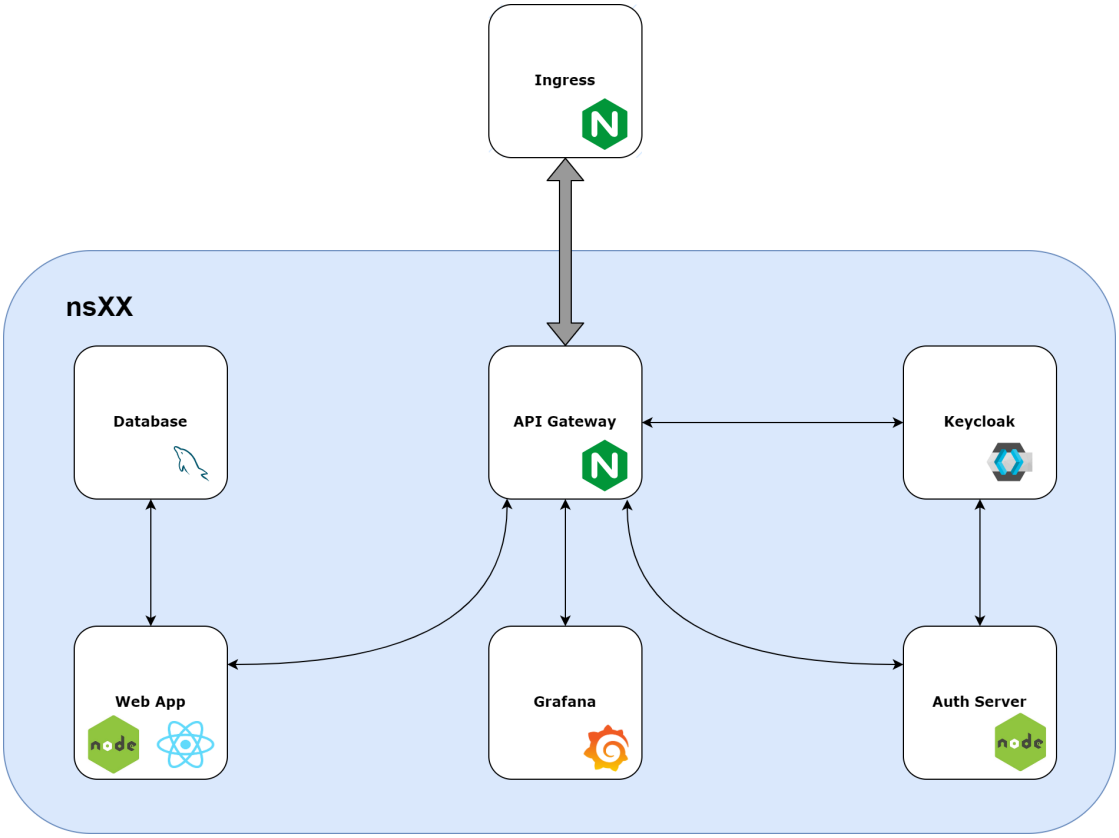

Figura 2.2. Container in nsXX

1. **Web App** - costituisce il punto di accesso primario all'applicazione web, fornendo anche eventualmente il servizio backend associato
2. **Keycloak** - identity provider
3. **API Gateway** - reverse proxy che gestisce il flusso del traffico sia tra i vari container interni che verso l'esterno
4. **AuthServer** - OAuth server
5. **Grafana** - piattaforma di analisi e monitoraggio che fornisce strumenti avanzati per visualizzare e interpretare i dati provenienti dai sensori
6. **Database** - garante della persistenza dei dati dell'applicazione web

Quando un utente generico effettua una richiesta a uno dei domini delle sottoreti, il flusso di comunicazione segue una sequenza ben definita. Innanzitutto, la richiesta passa attraverso l'ingress server, che funge da punto di ingresso primario per il traffico di rete. Successivamente, l'ingress server instrada la richiesta al sottodominio specificato. All'interno di quest'ultimo, la richiesta attraversa inizialmente il reverse proxy interno, che si occupa di inoltrarla al container specificato.

In questo modo, partendo dal presupposto che si considera affidabile la comunicazione tra container appartenenti alla stessa sottorete, si evitano pesanti e complessi algoritmi di sicurezza. L'unica comunicazione su cui è necessario garantire la riservatezza e l'integrità, utilizzando un canale SSL, è quella tra ingress server e utente.

Nei prossimi capitoli, sarà dedicata un'attenzione particolare all'analisi dettagliata di ciascun container all'interno della rete, con un focus più approfondito sul container *Web App*. Quest'ultimo sarà trattato in modo più dettagliato poiché rappresenta il fulcro delle attività svolte.

Capitolo 3

AuthServer

Il container *AuthServer*, basato sull'immagine Node.js, assume la responsabilità del processo di autenticazione degli utenti. Questo interagisce direttamente con il container *Keycloak*, utilizzando le API esposte per facilitare l'autenticazione. Nel quadro della rete di container precedentemente delineata, il container *AuthServer* può essere considerato come un server OAuth: quando un utente cerca di accedere a una risorsa protetta, l'applicazione lo reindirizza all'OAuth server per l'autenticazione, dove l'utente fornisce le proprie credenziali.

Il *AuthServer* offre la possibilità di autenticare gli utenti anche senza richiedere un username e una password, ma tramite un token di accesso. Questa soluzione è stata adottata presso il Politecnico di Torino con lo scopo di agevolare l'autenticazione di numerosi studenti che desideravano compilare un questionario senza dover gestire la registrazione individuale di ognuno sulla piattaforma. Inoltre, attraverso lo stesso meccanismo di token, è stato possibile anche garantire l'accesso a risorse più riservate.

Nelle sezioni successive, esamineremo più approfonditamente il ruolo del container *AuthServer* all'interno della sottorete.

3.1 Processo di autenticazione

Il server è stato sviluppato utilizzando la libreria OpenID di PassportJS, che mette a disposizione una serie di API per facilitare la comunicazione diretta con l'Identity Provider, in questo caso Keycloak.

Dopo aver istanziato con successo una nuova strategia di autenticazione, con particolare riferimento alla modalità "oidc" (OpenID Connect), si può sfruttare appieno le funzionalità di serializzazione e deserializzazione degli utenti all'interno della sessione.

Nello specifico, si definiscono diverse chiamate API per operazioni fondamentali come il login, il logout, il controllo sull'autenticazione utente e l'autenticazione tramite chiave privata. Queste chiamate costituiscono l'interfaccia attraverso cui il server interagisce con l'IdP per autenticare gli utenti e gestire le sessioni.

È da notare che tutte queste chiamate sono configurate anche all'interno del file di configurazione del reverse proxy Nginx. Questo approccio permette di dirigere in modo

appropriato le richieste in arrivo verso i rispettivi container all'interno dell'architettura del sistema.

Quando un utente desidera accedere alla piattaforma mediante le proprie credenziali, viene inviata una richiesta alla route `/auth` gestita dal reverse proxy Nginx. Questo, agendo come intermediario, inoltra la richiesta all'AuthServer che, a sua volta, reindirizza automaticamente l'utente alla pagina di login fornita da Keycloak. Una volta completata la procedura di autenticazione con successo, vengono impostati i cookie contenenti i dati relativi all'utente autenticato. Queste informazioni sono quindi condivise con l'applicazione web, permettendo all'utente di accedere e utilizzare le funzionalità del sistema.

Alla fine della sessione, l'utente può effettuare il logout tramite la route `/logout`. Questa operazione, come di consueto, viene gestita attraverso il reverse proxy. In caso di successo, vengono cancellati tutti i cookie associati all'utente, garantendo così la corretta terminazione della sessione e la sicurezza delle informazioni personali.

3.2 Token di accesso

Come precedentemente delineato nell'introduzione, il server ha la capacità di autenticare gli utenti che posseggono un particolare token, ovvero una key. Tale key consiste in due elementi distinti, ossia la chiave stessa e il suo corrispettivo hash, i quali sono separati da un punto. L'hash della chiave viene generato mediante l'utilizzo di un segreto che è posseduto esclusivamente dal server.

Al momento in cui un utente intende accedere all'applicazione autenticandosi tramite la suddetta chiave, sarà tenuto a sfruttare un link appositamente strutturato, il quale segue il formato seguente:

`https://<dominio>/auth/key?key=<key>.<hashedKey>&path=<redirectURL>`

- `/auth/key` rappresenta l'endpoint specifico del server dedicato all'autenticazione mediante la chiave
- `key=<key>.<hashedKey>` contiene sia la chiave di accesso che la sua versione hashata, presentate in coppia
- `path=<redirectURL>` indica il percorso verso cui l'utente sarà automaticamente reindirizzato una volta completata con successo la procedura di autenticazione. Questo parametro risulta particolarmente utile quando si desidera instradare l'utente verso una pagina specifica, eventualmente con dei parametri di query impostati. Ad esempio, nel contesto del client PROMET&O, è possibile includere il parametro `boardID` nell'URL di reindirizzamento

Al momento della conclusione dell'autenticazione, saranno immagazzinati all'interno dei cookie una serie di dati fondamentali per il corretto funzionamento dell'applicazione web, i quali risultano necessari per determinare con precisione il profilo dell'utente.

In questo contesto di autenticazione, verranno conservati i seguenti valori all'interno dei cookie:

- `User`: `Anonymous`
- `Role`: `Student` | `Private`
- `Key`: `<key>.<hashedKey>`

L'accesso tramite token è limitato a due ruoli distinti: *Student* e *Private*, come specificato. (vd. 4.3 - *Gestione ruoli*)

Nel server, sono conservati due segreti, ciascuno destinato all'Hash-based Message Authentication Code (HMAC) di una chiave generata casualmente composta da 9 caratteri. Uno dei due segreti è associato al ruolo *Student*, mentre l'altro al ruolo *Private*.

La responsabilità di generare le chiavi e distribuirle agli utenti ricade sull'amministratore dell'applicazione, il quale deve prestare particolare attenzione alle persone con cui condivide la chiave di tipo *Private*, poiché a questo ruolo possono essere associati dei privilegi più elevati rispetto al ruolo *Student*.

Il file per la generazione delle chiavi è denominato `keyGen.msj` e accetta due parametri distinti per creare le chiavi associate ai ruoli *Student* e *Private*. Questi parametri sono specificati utilizzando i caratteri `-s` o `-p`.

Quindi, l'utilizzo del comando per generare una chiave per il ruolo *Student* sarebbe simile a quanto segue

```
node ./keyGen [-s|-p]
```

Durante il processo di autenticazione, il server effettuerà una verifica per determinare con quale dei due segreti è stato generato l'HMAC relativo alla chiave fornita. Questo controllo viene eseguito all'interno del middleware `ensurePermissions` della chiamata `/auth/key`.

```
const ensurePermissions = async (req, res, next) => {
    // Pass the key throguh an header
    try {
        const passedKey = req.header("x-key");

        let [key, hashKey] = passedKey.split(".");
        let isValid = compareKeys(hashKey, key);
        console.log("isValid", isValid);

        if (isValid) {
          req.role = isValid;
          console.log("isValid", isValid);
          next();
        }
        else return res.status(401).send({ message: "Invalid key", code: 401 });
    } catch {
        return res.status(401).send({ message: "Invalid key", code: 401 });
    }
};
```

In caso di corrispondenza tra il segreto e l'HMAC, il processo di autenticazione verrà completato con successo e saranno impostati i cookie per l'utente autenticato. Al contrario, se non viene individuata alcuna corrispondenza, il processo di autenticazione fallirà e l'utente riceverà un messaggio di errore del tipo *Unauthorized*.

Capitolo 4

Web App

All'interno della rete di container, la Web App riveste il ruolo principale, ovvero contiene l'interfaccia utente di PROMET&O. Tramite quest'ultima, gli utenti hanno la possibilità sia di compilare i questionari che di consultare e confrontare i dati rilevati dai sensori in base a diversi intervalli temporali. La Web App è composta da frontend e backend, sviluppati utilizzando rispettivamente i framework di Javascript React ed Express. Sono state scelte queste tecnologie sia per la loro semplicità sia per vincoli imposti dalla continuità del progetto sviluppato in precedenza. In particolare, la nuova versione della Web App sfrutta un server locale in sostituzione alla precedente soluzione tramite AWS (vd. 1 - *Architettura precedente*).

Nelle successive sezioni verranno analizzati dettagliatamente tutti i componenti della Web App e le interazioni tra di loro.

4.1 Frontend

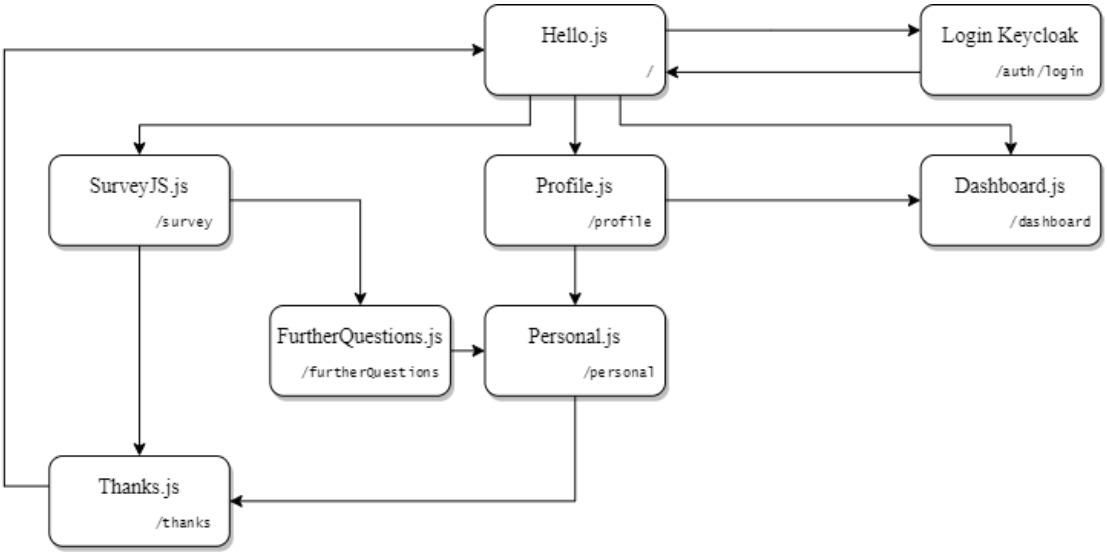

Figura 4.1. Struttura navigazione

Nell'illustrazione soprastante, è visibile il diagramma che delinea la navigazione tra le diverse pagine dell'applicazione PROMET&O.

L'intera logica di routing è gestita dal componente `Routes` definito all'interno del file `App.jsx`. Ogni route è accessibile in base ai privilegi conferiti all'utente in fase di autenticazione. I diversi componenti custom che si occupano del controllo dei permessi saranno successivamente descritti in modo dettagliato in 4.3 - *Gestione Ruoli*.

Le seguenti sezioni del documento si concentreranno sull'analisi dettagliata di ciascuna delle pagine presenti all'interno della figura 4.1.

4.1.1 Hello

`Hello.js` rappresenta la homepage dell'applicazione. Questa presenterà dei componenti diversi al suo interno, a seconda del ruolo dell'utente che vi farà accesso (vd. 4.3 - *Gestione Ruoli*).

Considereremo i quattro tipi di accesso possibili:

1. **Accesso base**: la schermata sarà visibile quando si farà accesso all'applicazione senza alcun tipo di autenticazione, accendendo direttamente a `https://<dominio>/`

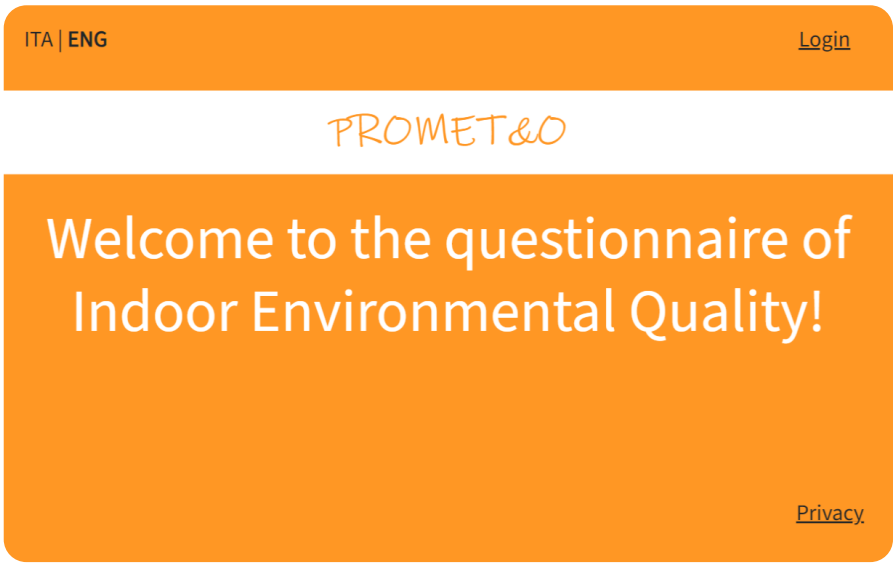

Figura 4.2. Homepage base

2. **Accesso con token *Studente***: utilizzando il token *Studente*, agli utenti viene concessa l'opportunità di accedere al questionario dell'applicazione. L'autenticazione dell'utente avviene tramite l'AuthServer (vd. 3 - *AuthServer*). Se l'autenticazione avrà esito positivo, il ruolo dell'utente, ossia `Student`, verrà memorizzato nel cookie `Role` (vd. 4.3 - *Gestione Ruoli*).

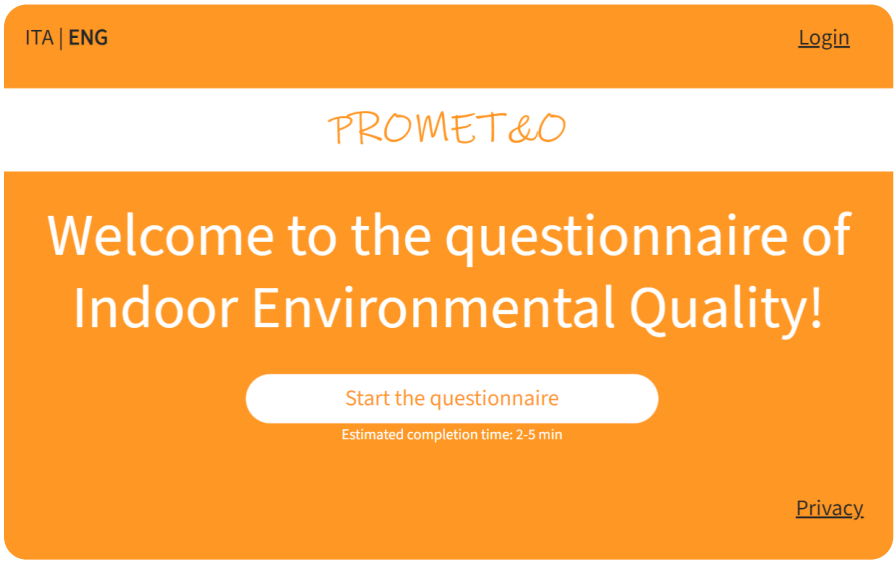

Figura 4.3. Homepage con token "studente"

3. **Accesso con token *Private***: per gli utenti che utilizzano il token *Private*, l'accesso non è limitato solo al questionario, ma comprende anche la dashboard dell'applicazione. Come per il token *Studente*, l'autenticazione dell'utente avviene attraverso il reverse proxy (vd. 3 - *AuthServer*) ma in questo caso al `Role` dei cookie verrà assegnato il valore `Private` (vd. 4.3 - *Gestione Ruoli*).

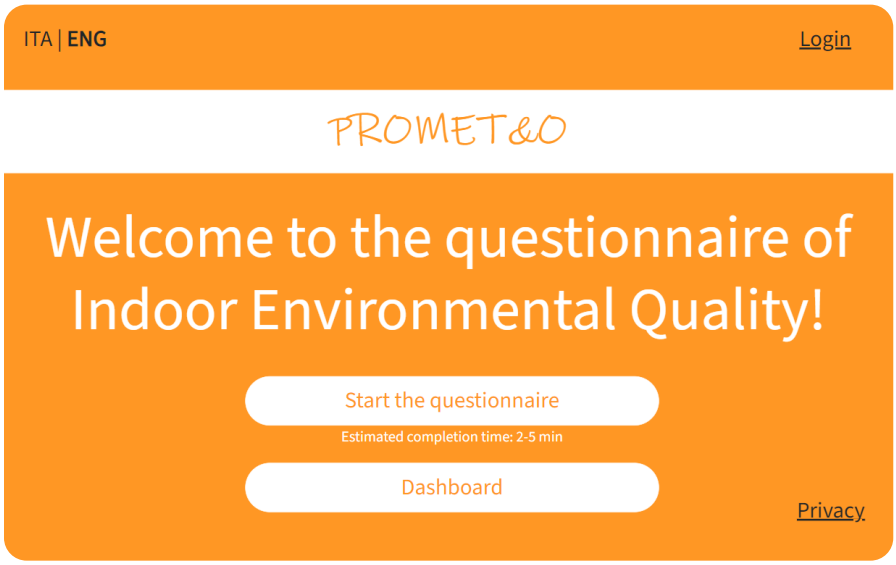

Figura 4.4. Homepage con token "privato"

4. **Accesso con username e password**: l'ultima schermata proposta è invece quella visualizzata dopo che l'utente ha effettuato il login con username e password con successo. In questo caso sarà possibile accedere all'intera applicazione, in particolare al profilo dell'utente, dove sono visualizzati tutti i questionari effettuati, pagina visualizzabile solo con questo tipo di autenticazione. Qui il ruolo memorizzato nei cookies potrà essere `Admin`, `Editor` o `Viewer`, a seconda di ciò che è stato settato per l'utente che sta effettuando l'accesso nel realm di Keycloak. I ruoli menzionati sono inoltre utilizzati da Grafana per gestire il livello di autorizzazione degli utenti all'interno della piattaforma.

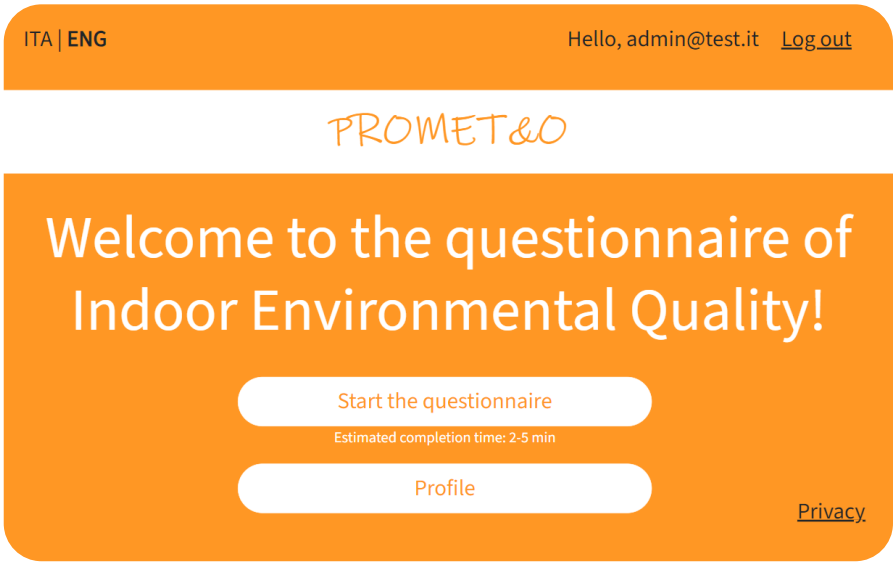

Figura 4.5. Homepage utente loggato

4.1.2 SurveyJS

L'intera sezione relativa al questionario per la raccolta di dati soggettivi è attualmente implementata nel file `SurveyJS.jsx` e gestita tramite SurveyJS, una libreria JavaScript open source specializzata nella creazione di questionari. Per la creazione e modifica dei questionari attualmente presenti all'interno dell'applicazione è stato utilizzato il tool online che, oltre ad offrire un'interfaccia grafica semplice ed intuitiva, fornisce la possibilità di importare ed esportare il file JSON contenente i dati e la logica del questionario.

surveyID e boardID

A differenza della versione precedente dell'applicazione, in cui era gestito un unico questionario, si è lavorato per sviluppare un supporto che consentisse la gestione di diversi tipi di questionari, ognuno associato a un boardID specifico.

La mappatura tra surveyID e boardID (vd. 4.1.8 - *boardID*) avviene mediante il file `boardIDtoSurvey.json` memorizzato nella cartella `assets/`. Questo contiene un array di oggetti, ciascuno di questi rappresentante un sensore.

Un esempio di oggetto:

```
{
  "boardID": 1001,
  "serialNumber": "17120003-00-19/001-000201",
  "sensorNumber": 1,
  "classroomName": "Aula 1P",
  "surveyID": 1
}
```

L'informazione effettivamente utilizzata all'interno dell'applicazione è limitata all'associazione tra surveyID e boardID. Tuttavia, per eventuali sviluppi futuri, sono state integrate anche ulteriori informazioni come il numero seriale, il numero del sensore e il nome dell'aula associata al sensore. In `SurveyJS.jsx`, è implementato l'utilizzo del file `boardIDtoSurvey.json` attraverso la seguente dichiarazione di importazione: `import bToS from "../assets/boardIDtoSurvey.json"`. Questa viene quindi utilizzata per reperire il surveyID:

```
const surveyID = bToS.find(item =>
    item.boardID == boardID)?.surveyID || null;
```

Import dinamico

Il fulcro della gestione di un multi-questionario risiede nell'importazione dinamica dei modelli e dei loro relativi CSS. Questo comportamento all'interno dell'applicazione viene coordinato dalla seguente useEffect, triggerata al caricamento della pagina:

```
useEffect(() => {
    const importOfFiles = async () => {
      if (loading && boardID && surveyID) {
        try {
          import(`../assets/surveys/survey${surveyID}.json`)
            .then((json) => {
              import(`../assets/surveys/survey${surveyID}.scss`)
                .then((css) => {
                  let survey = new Model(json);
                  if (props.ita) survey.locale = "it";
                  survey.css = css;
                  setSurvey(survey);
                  setLoading(false);
                })
                .catch(() => {
                  console.log("CSS not found for boardID: " +
                      boardID + " surveyID: " + surveyID + ".");
                });
            })
            .catch(() => {
              console.log("JSON not found for boardID: " +
```

```
                boardID + " surveyID: " + surveyID + ".");
          });
      } catch {
        setSurvey(false);
        setLoading(false);
      }
    }
  };
  importOfFiles();
}, [loading]);
```

Dopo aver controllato le variabili essenziali, si procede con l'import del file JSON e CSS corrispondenti, entrambi localizzati nella cartella `assets/surveys/`.

Dopo che l'import dei due file è avvenuto con successo, è possibile associarli al Model (struttura dati di `survey-react-ui` necessaria per la creazione del questionario). Questo oggetto, dopo aver associato correttamente il JSON, il CSS e la lingua del questionario, verrà poi salvato all'interno dello stato `survey`.

La prima domanda del questionario verrà quindi mostrata all'utente quando `loading` sarà settato a `false` e `survey` conterrà dei dati.

Il tag contenente il questionario è il seguente

```
<Survey
    id={"surveyModel" + surveyID}
    model={survey}
    onComplete={sendDataToServer}
/>
```

Per agevolare la modularizzazione dell'applicazione e garantire un'identificazione univoca per ciascun questionario, è stato associato a ognuno di essi un ID univoco. Tale ID è utilizzato come elemento identificativo dello stile all'interno dei fogli CSS, prevenendo così possibili conflitti tra questionari differenti.

Stile e SCSS

Nel perseguire l'obiettivo di modularizzare gli stili tra i vari questionari, è stato preferito l'utilizzo di SCSS rispetto a CSS. Ciò consente di sfruttare le regole dei selettori nidificati, semplificando la struttura e la comprensione del codice, soprattutto quando si definiscono stili specifici per gli elementi figli identificati dall'ID del questionario.

La maggior parte delle regole di stile nei file SCSS rappresenta un override delle classi fornite da SurveyJS, applicato a tutte le domande del questionario X considerato, indipendentemente. Tuttavia, è anche possibile personalizzare in modo diverso la stessa classe in base alla domanda presentata. Per ottenere questo comportamento, è necessario identificare la domanda all'interno del foglio di stile utilizzando l'attributo `[data-name="Qxx"]`.

Ad esempio, nel questionario numero 1, nel foglio di stile `survey1.scss` sono presenti quattro override distinti per i tag `h5` figli di elementi con un attributo specifico `data-name`. La classe `.question-base` è stata definita come uno stile di base che può essere esteso in ciascuno di questi override. In questo caso, l'obiettivo è associare ad ogni

domanda uno sfondo con un colore diverso. La palette dei colori è definita all'interno di un ulteriore file SCSS denominato `color-theme.scss`, che viene importato all'inizio di `survey1.scss`.

```
@import "color-theme";

#surveyModel1 {
    .question-base {
        border-radius: 50px !important;
        padding: 20px;
        text-align: center;
        font-size: 250%;
    }

    [data-name="Q03"],
    [data-name="Q04"] {
        h5 {
          @extend .question-base;
          background-color: $thermal-color !important;
          border-bottom: 1px solid var(--thermal-color) !important;
        }
    }
    ...
```

Il nome della domanda è definito all'interno del file JSON che verrà passato al Model di SurveyJS durante la sua creazione. È bene ricordarsi di prestare attenzione anche alle regole di stile qualora si decidesse di modificare il nome di una domanda.

Submit dei dati

Quando l'utente terminerà il proprio questionario, verrà chiamata la funzione definita nella `onComplete` del tag `Survey`, ovvero la `sendDataToServer`, la quale preparerà le strutture dati che verranno inviate al server dell'applicazione in un secondo momento.

```
function sendDataToServer(sur) {
    let data = sur.data;

    sur.getAllQuestions()
        .map(q => q.jsonObj.name)
        .filter( q => q.match(/[Q]\d/g)).map(q => {
            if (!data.hasOwnProperty(q))
            data[q] = "";
        })

    comfort.evaluateComfort(sur.data).then((res) => {
      [...]
    });
  }
```

La struttura dati principale è rappresentata dall'oggetto `data`, che successivamente verrà inviato al server dell'applicazione. Inizialmente, questo oggetto viene popolato con un'altra struttura dati generata da SurveyJS. Quest'ultima contiene l'associazione tra il

nome identificativo della domanda e il valore della risposta fornita dall'utente (sia esso un intero, una stringa o un vettore di oggetti).

In questo modo, l'oggetto `data` contiene inizialmente solo le domande a cui l'utente ha effettivamente risposto. Tuttavia, per ottenere un elenco completo di tutte le domande presenti nel questionario, comprese quelle che potrebbero non essere mai state visualizzate all'utente, si fa uso dell'API di SurveyJS, chiamando la funzione `getAllQuestions()`. Questa API restituisce l'elenco completo delle domande, da cui vengono estratti i nomi delle domande (formato `Qxx`), se uno di questi non è presente all'interno di `data`, quest'ultimo viene aggiunto all'oggetto associandogli una stringa vuota. Questo passaggio è di fondamentale importanza per il successivo processo di esportazione dei dati, come sarà approfondito nei capitoli successivi.

A questo punto vengono calcolate le percentuali soggettive di comfort, basate sulle risposte fornite dall'utente. La funzione di calcolo di comfort si trova all'interno del file `evaluateComfort.jsx` e supporta anch'essa la possibilità di accettare diversi tipi di questionari.

```
async function evaluateComfort(answers) {
    let result = {/* inizializzazione */};

    const apiPromises = [];

    ["Temp", "Light", "Air", "Sound", "IEQ"].forEach((measure) => {
        const promise = API.getDashboardValues(...);
        apiPromises.push(promise);
    });

    await Promise.allSettled(apiPromises).then((apiResults) => {
        /* controllo valori oggettivi ritornati */
        const surveyID = bToS.find((item) =>
            item.boardID ==
                localStorage.getItem("boardID"))?.surveyID || null;

        /* controllo surveyID */
        /* assegnazione valori oggettivi a result */
        // [...]

        switch(surveyID) {
          case 1: result = survey1(answers, result); break;
          // case 2: result = survey2(answers, result); break;
          default : result = survey1(answers, result)
        }
    });

    return result;
}
```

L'oggetto `result` conterrà sia dati oggettivi che soggettivi. I dati oggettivi saranno aggiunti all'oggetto dopo il completamento di tutte le chiamate API, mentre i dati soggettivi saranno calcolati successivamente e inseriti nei campi dedicati dell'oggetto.

Per distinguere i due tipi di campi in modo chiaro, sono stati aggiunti i suffissi `_obj` e `_sbj` alle chiavi dell'oggetto.

Il supporto a più questionari è gestito attraverso uno `switch... case` che, in base al valore di `surveyID`, dirige l'oggetto `answers`, verso la funzione appropriata associata al questionario. Questa funzione contiene la logica necessaria per calcolare le metriche di comfort soggettive specifiche per il questionario in esame.

Dopo aver calcolato il valore finale di `result`, è possibile concludere l'analisi della funzione `sendDataToServer` precedentemente presa in considerazione.

```
comfort.evaluateComfort(sur.data).then((res) => {
    data.mqttMessage = res;
    data.resultID = uuid();
    data.boardID = boardID;
    data.userID = props.anon ?? localStorage.getItem("email");
    props.setAnswers(data);
    localStorage.setItem("completedSurvey",
    comfort.sentenceFun(res, props.ita));
    navigate(props.anon ? "/furtherQuestions" : "/thanks");
});
```

L'oggetto restituito dalla funzione `evaluateComfort` costituirà il contenuto del messaggio MQTT (vd. 4.2 - *Backend*) che il server dell'applicazione si occuperà di pubblicare nel topic associato al corretto `boardID`.

Successivamente, all'oggetto `data` vengono associati un identificativo `uuid` (campo `resultID` dell'oggetto `result`), il `boardID` e l'ID dell'utente (un ID univoco nel caso in cui l'utente sia anonimo, oppure l'email se l'utente è autenticato).

Il valore memorizzato all'interno del localStorage, con chiave `completedSurvey`, rappresenta il risultato della funzione `sentenceFun`. Questa funzione ha il compito di trasformare i valori di comfort risultanti dal questionario appena compilato in una frase, che verrà mostrata all'utente nella pagina `thanks`. Inoltre, la presenza di questa chiave nel localStorage determina la visualizzazione stessa delle pagine `thanks` e `furtherQuestions`. Queste pagine saranno accessibili dall'utente solo se quest'ultimo ha effettivamente completato un questionario, altrimenti, verrà reindirizzato alla homepage.

Creazione di un nuovo questionario

Per aggiungere un nuovo questionario all'interno dell'applicazione, sono necessari alcuni passi per coordinare tutti i componenti coinvolti nella sua gestione. Nel caso in cui si desideri creare una versione modificata del primo questionario, si possono seguire questi passaggi:

1. **Creazione dei file**

   Inizialmente, è necessario creare una copia dei file `survey1.json` e `survey1.scss` all'interno della directory `assets/surveys/`. È importante mantenere il formato del nome del file come "survey" seguito da un numero e l'estensione (.json o .scss), a causa della struttura dell'importazione dinamica, come precedentemente illustrato

in questo capitolo. Ad esempio, si possono chiamare i nuovi file `survey2.json` e `survey2.scss`.

2. **Associazione al boardID**

   Per scopi dimostrativi, si può creare un sensore demo con `boardID = 1` e associare ad esso il nuovo tipo di questionario. Queste informazioni vanno aggiunte all'interno del file `boardIDToSurvey.json`:

```
{
  "boardID": 1,
  "serialNumber": "",
  "sensorNumber": 0,
  "classroomName": "",
  "surveyID": 2
},
```

   Dopo aver eseguito i necessari rebuild dell'applicazione, sarà possibile accedere a `https://<dominio>/survey?boardID=1`, dove verrà visualizzato il modello JSON di survey2. Per garantire che anche il CSS funzioni correttamente, è necessario modificare l'id `#surveyModel1` nel file `survey2.scss` in `#surveyModel2`, o meglio, utilizzare lo stesso numero presente nel nome del file.

3. **Modifica del modello JSON**

   Per ottenere un feedback immediato delle modifiche apportate al modello JSON, è possibile copiare il file all'interno di un nuovo survey su https://surveyjs.io nella sezione "JSON Editor". Dopo l'importazione, tutte le modifiche effettuate tramite qualsiasi sezione del sito ("Designer", "Logic", etc.) saranno automaticamente riflesse nell'editor. Al termine, sarà necessario sostituire il contenuto di `survey2.json` con il contenuto presente nel sito.

4. **Modifica dello stile**

   Qualsiasi override alle classi di SurveyJS o qualsiasi modifica dello stile, deve essere incluso all'interno dell'id padre `#surveyModel2`. Questo approccio è essenziale per evitare conflitti tra stili di diversi questionari, garantendo una corretta modulazione e separazione degli stili all'interno dell'applicazione.

5. **Calcolo percentuali di comfort**

   L'ultimo passo cruciale per integrare correttamente un nuovo questionario all'interno dell'applicazione consiste nell'associare al questionario una nuova funzione per il calcolo degli indici di comfort soggettivi. Attualmente, l'applicazione include la funzione `survey1` nel file `evaluateComfort.jsx`. Se le modifiche apportate a `survey2` non hanno influenzato i nomi delle domande o la logica di base, è possibile riutilizzare la stessa funzione. In caso contrario, sarà necessario modificarla adeguatamente, aggiornando i nomi delle domande o apportando le modifiche necessarie alla logica.

L'associazione tra il nuovo questionario e la funzione avviene inserendo un nuovo `case` nel costrutto `switch` presente nella funzione `evaluateComfort` all'interno di `evaluateComfort.jsx`:

```
switch(surveyID) {
  case 1: result = survey1(answers, result); break;
  //ipotizzando di aver creato una nuova funzione survey2
  case 2: result = survey2(answers, result); break;
  default : result = survey1(answers, result)
}
```

4.1.3 Profile

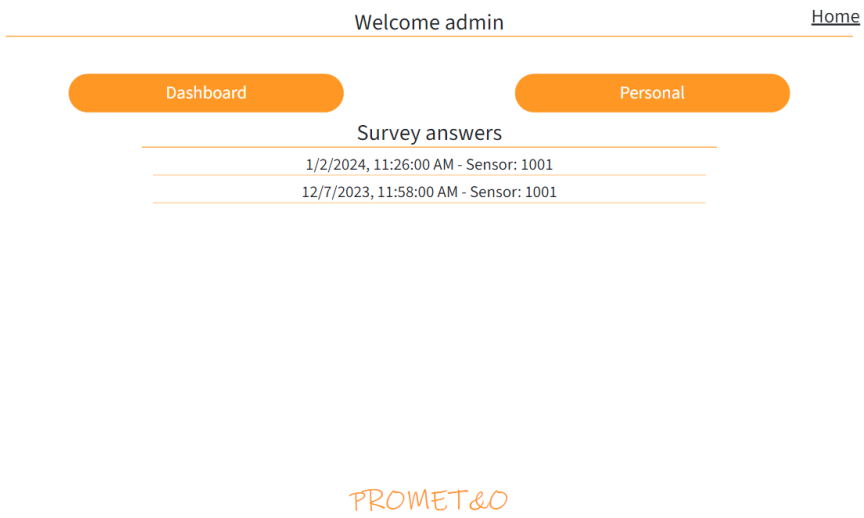

Figura 4.6. Pagina profilo

La sezione del profilo utente è dedicata esclusivamente agli utenti che hanno completato con successo il processo di registrazione e hanno effettuato l'accesso. Questa sezione fornisce un dettagliato elenco dei questionari svolti dall'utente a partire dalla creazione del profilo. Ogni questionario sottomesso è visualizzato insieme alla data di completamento e al relativo boardID associato. La pagina del profilo non solo offre un resoconto delle attività dell'utente, ma funge anche da punto di accesso per la Dashboard e per il questionario dedicato alle domande personali. Un'analisi approfondita di questi aspetti sarà condotta nelle sezioni successive.

4.1.4 Further Questions

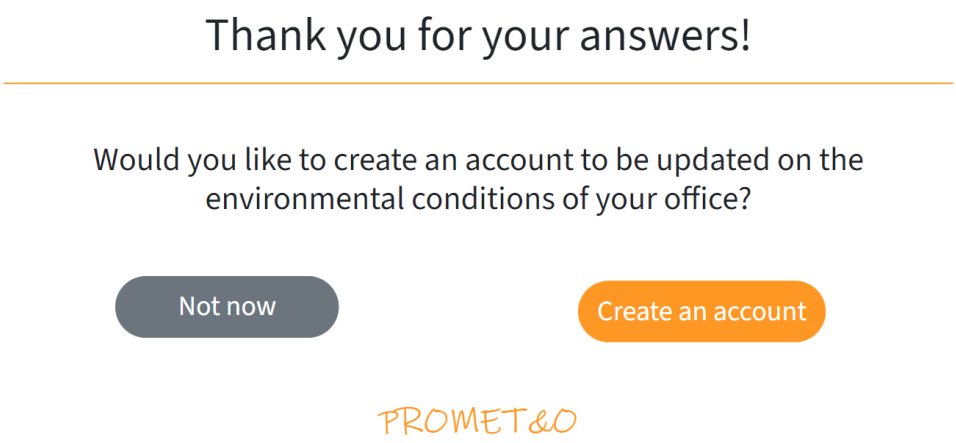

Figura 4.7. Further Questions

La schermata illustrata sarà visualizzata esclusivamente dagli utenti che rivestono un ruolo di tipo ”Studente” o ”Privato” al termine del questionario dedicato alla raccolta di dati soggettivi. Viene quindi concessa la facoltà di creare un nuovo account oppure di eludere questa fase mediante cliccando il pulsante ”Non ora”. In quest'ultimo scenario, l'utente sarà indirizzato alla pagina contenente le domande personali (vd. 4.1.5 - *Personal*).

4.1.5 Personal

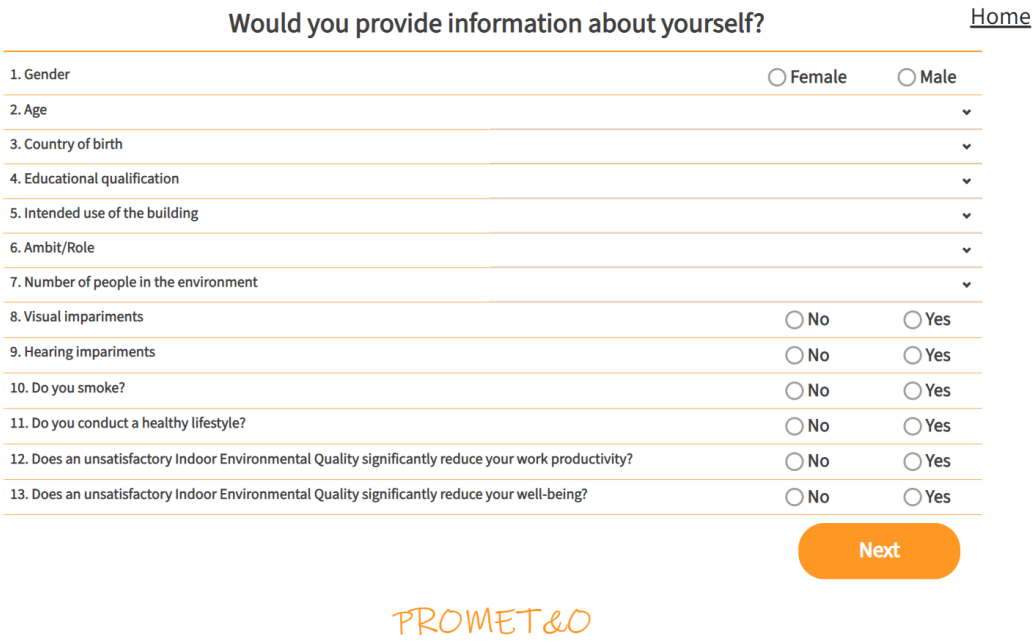

Figura 4.8. Domande personali

La route `/personal` è accessibile agli utenti autenticati mediante token o tramite username e password. Gli utenti autenticati tramite token potranno accedere alla route solo dopo aver completato il questionario, utilizzando il pulsante "Not now" nella pagina `/furtherQuestions`. Al contrario, gli utenti autenticati tramite username e password avranno la possibilità di modificare le proprie domande personali ogni volta che lo ritengano necessario. Per farlo, potranno accedere alla sezione utilizzando il pulsante "Personal" nella pagina `/profile`.

La pagina "Personal" contiene un insieme di domande che riguardano l'utente e l'ambiente circostante in quel momento.

Il questionario personale è gestito in maniera analoga al questionario per la raccolta dei dati soggettivi. Nei paragrafi successivi analizzeremo analogie e differenze.

Import dei file

A differenza del questionario per la raccolta dei dati soggettivi, non si ha la necessità di supportare questionari multipli, di conseguenza gli import dei file JSON e SCSS sono statici e non dinamici.

```
import * as surveyJSON from "../assets/surveys/personal.json";
import * as surveyCSS from "../assets/surveys/personal.scss";

[...]
function Personal(props) {
    const survey = new Model(surveyJSON);
    if (props.ita) survey.locale = "it";
    survey.css = surveyCSS;
    return (
    <Container fluid>
      <Survey
        id="personal"
        model={survey}
        onComplete={sendDataToServer}
       />
    </Container>
    );
}
```

Stile e SCSS

Il meccanismo utilizzato per modificare lo stile del questionario è lo stesso già precedentemente illustrato nella sezione *SurveyJS - Stile e SCSS* (4.1.2). In questo contesto l'id padre è `#personal`.

Recupero ultime risposte

Se l'utente ha effettuato l'autenticazione in precedenza utilizzando le proprie credenziali, verrà effettuata una chiamata API al fine di recuperare un'eventuale ultima sequenza di risposte fornite alle domande personali. Queste risposte, se presenti, saranno quindi integrate nel questionario come domande già affrontate.

```
useEffect(() => {
    const fetchData = async () => {
      try {
        if (!props.anon) {
          const resp = await API.getOldValues(user);
          const oldValues = JSON.parse(resp.answers);
          if (oldValues && Object.keys(oldValues).length > 0) {
            Object.entries(oldValues)
              .filter(([key]) =>
                !["personalID", "user", "timestamp"].includes(key))
              .forEach(([key, value]) => survey.setValue(key, value));
          }
        }
      } catch (err) { console.error(err);}
    };
    fetchData();
}, []);
```

Submit dei dati

Il processo di preparazione dei dati da inviare al server è del tutto analogo a quello già affrontato nella sezione *SurveyJS - Submit dei dati* (4.1.2). Le differenze sostanziali risiedono nel formato del nome delle domande (qui definite come "pxx" anziché "Qxx") e, ovviamente, nella mancanza del calcolo del comfort. Per quest'ultimo motivo, la `completedSurvey` nel localStorage verrà inizializzata a semplicemente a `true` e non con il risultato della `sentenceFun`.

4.1.6 Thanks

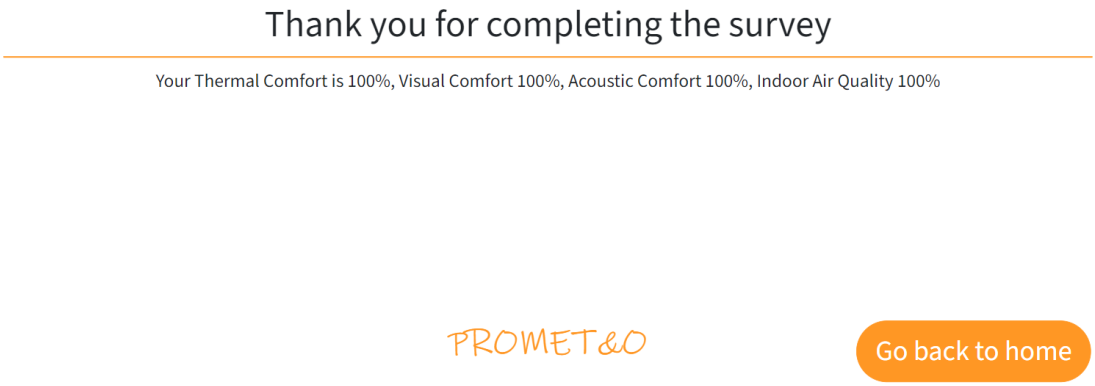

Figura 4.9. Thanks

La route `/thanks` viene visualizzata da qualsiasi tipo di utente autenticato al termine del questionario per la raccolta dei dati soggettivi. Qui vengono mostrati gli indici di comfort basati sulle risposte fornite nel questionario appena compilato. Similmente, al termine del questionario personale, gli utenti autenticati tramite username e password visualizzeranno la medesima schermata, sebbene senza la visualizzazione degli indici di comfort.

4.1.7 Dashboard

La dashboard rappresenta il fulcro della visualizzazione di dati oggettivi. Qui è possibile esaminare gli indicatori di comfort insieme alle loro misurazioni corrispondenti, monitorare l'evoluzione nel tempo attraverso grafici e confrontare le diverse metriche.

Sono presenti due visualizzazioni chiave: la prima, *insight*, fornisce un quadro dettagliato dei valori espressi in forma numerica, consentendo agli utenti di ottenere una visione immediata delle condizioni ambientali. La seconda visualizzazione, *timeline*, presenta invece il grafico che traccia l'andamento temporale della misura presa in considerazione. É possibile passare da una visualizzazione all'altra grazie al pulsante in basso a sinistra "Show/Hide the graph".

Visualizzazione *insight*

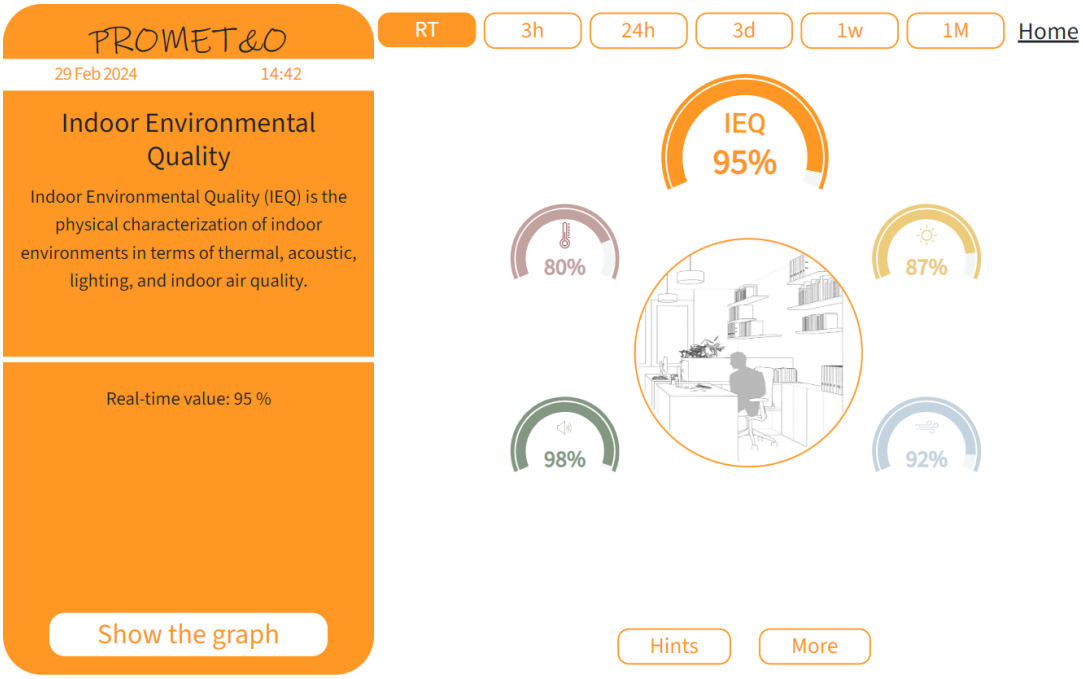

Figura 4.10. Dashboard - *insight*

Nella visualizzazione *insight* sono collocati cinque gauge, disposti in senso orario dall'alto, che delineano i parametri di comfort ambientale, comfort visivo, qualità dell'aria, comfort acustico e comfort termico. Per ottenere dettagli specifici su ciascun aspetto del comfort, è sufficiente cliccare sul rispettivo indicatore, aprendo così una serie di cerchi rappresentanti le misurazioni associate.

Sulla colonna di sinistra è possibile accedere a ulteriori dettagli sulla natura e l'andamento di ogni misurazione considerata. Per finestre temporali superiori a tre ore, è disponibile la visualizzazione di parametri statistici come la media, la deviazione standard, il decimo e novantesimo percentile, nonché il valore massimo e minimo.

La visualizzazione dei dati nella dashboard può essere personalizzata in base a diverse finestre temporali, quali tempo reale, ultime 3 ore, ultime 24 ore, ultima settimana e ultimo mese.

I pulsanti "Hints" e "More" situati nella parte inferiore della pagina consentono all'utente di esplorare in modo più approfondito la natura delle misurazioni presenti nella dashboard. Forniscono suggerimenti su come migliorare gli indici di comfort e informazioni dettagliate sulla misurazione specifica considerata.

Visualizzazione *timeline*

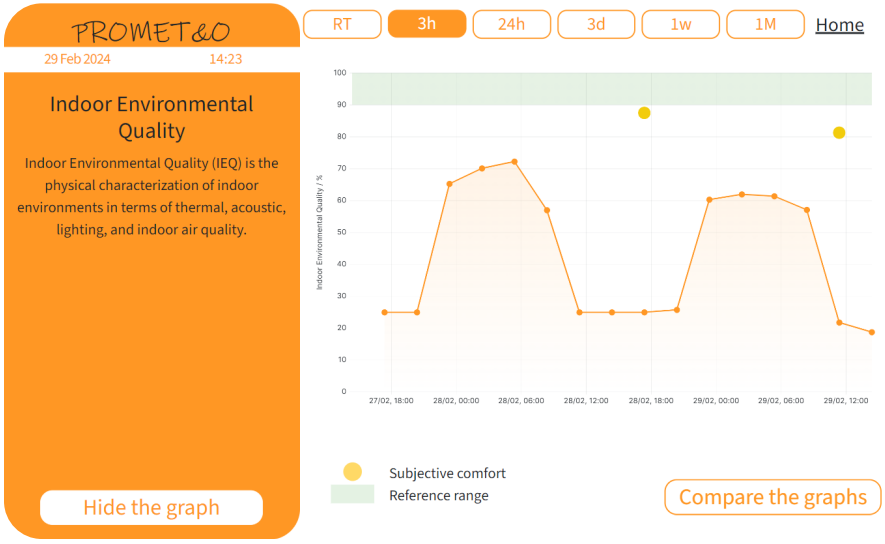

Figura 4.11. Dashboard - *timeline*

Nella schermata della *timeline*, l'utente può esaminare l'andamento di una specifica misura all'interno di un intervallo temporale (RT, 3h, 12h, 1w, 1M).

Figura 4.12. Paragone grafici

Il pulsante nell'angolo inferiore destro consente di confrontare fino a quattro grafici rappresentanti misure diverse (selezionabili dalla colonna arancione a sinistra) o finestre temporali differenti (corrispondenti a quelle presenti nella visualizzazione *insight*).

Overview dei componenti

Nelle sezioni successive analizzeremo i componenti fondamentali che sorreggono la dashboard e il loro ruolo all'interno della stessa.

measures.json

Per garantire un disaccoppiamento tra le unità di misura adottate dall'applicazione e quelle impiegate dalla API, è stato introdotto un file JSON denominato `measures.json` localizzato in `/client/src/assets`. Questo file contiene un vettore di oggetti rappresentanti le unità di misura utilizzate dall'applicazione. Utilizzando questo approccio è possibile associare ad ogni misura ulteriori dettagli. Si prenda a titolo esemplificativo l'umidità relativa:

```
"RH": {
    "labelText": {
      "it": "Umidità relativa",
      "en": "Relative Humidity"
    },
    "labelSymbol": "RH",
    "grafana":{
      "unitName": "humidity",
      "panelId": 3
    },
```

```
    "unit": "%",
    "referenceValue": "25-60%",
    "law": "EN 16798-1",
    "more": {
      "en": "[...]",
      "it": "[...]"
    },
    "hints": {
      "en": "[...]",
      "it": "[...]"
    },
    "description": {
      "en": "[...]",
      "it": "[...]"
    }
  }
```

Ogni elemento presente all'interno dell'oggetto viene utilizzato in una parte diversa della Dashboard:

- `labelText` e `labelSymbol` contengono il nome esteso e contratto di come quella misura viene denominata all'interno dell'applicazione
- `grafana` invece contiene i dati da utilizzare per richiamare l'API corretta, ovvero come la misura viene denominata all'interno del suo dominio e a quale pannello è associata
- `unit` rappresenta l'unità di misura
- `referenceValue` indica quali siano i parametri di riferimento, in modo tale da fornire all'utente la possibilità di analizzare da sé i dati presentati nella Dashboard
- `more` e `hints` sono stringhe che verranno presentate nei modal triggerati dai pulsanti "Hints" e "More" nella pagina principale della Dashboard
- `description` contiene una breve definizione della misura presa in considerazione, questa verrà mostrata nella colonna arancione di sinistra nella pagina principale della Dashboard

Compliances.jsx

Il componente principale per il recupero dei valori degli indici di comfort e delle loro misurazioni specifiche è il file `Compliances.jsx`. Questo componente gioca un ruolo fondamentale nell'ottenere tali dati utilizzando le API fornite da Grafana. Quest'ultimo, infatti, funge da intermediario tra l'applicazione Promet&o e la sorgente dati. In questa dinamica, Grafana agisce come un proxy, facilitando la comunicazione e garantendo un'efficace integrazione tra la sorgente dei dati oggettivi e la loro rappresentazione nella dashboard.

I valori all'interno della dashboard verranno aggiornati automaticamente tramite la funzione `fetchAndSetData` ogni volta che si verifica una delle seguenti condizioni: se l'utente resta sulla pagina per oltre cinque minuti o se cambia la finestra temporale.

```
// data refresh every five minutes
useEffect(() => {
const APIcall = setInterval(() => { fetchAndSetData();}, 300000);
    return () => clearInterval(APIcall);
}, []);

// data refresh everytime user changes timewindow
useEffect(() => {
    fetchAndSetData();
}, [props.timeWindow]);
```

La funzione `fetchAndSetData` si occupa di chiamare l'API `getDashboardValues`, dedicata al recupero dei dati per ciascuna misura definita nel file `measures.json`. I valori recuperati vengono visualizzati nella pagina solo dopo che tutte le richieste sono state completate. Questo comportamento è reso possibile grazie all'utilizzo di `Promise.allSettled`. Le richieste non riuscite vengono quindi filtrate, mentre per quelle eseguite con successo, i risultati vengono salvati nello stato `RTValues` come un insieme di coppie chiave-valore. In questo insieme, la chiave rappresenta il nome della misura, mentre il valore corrisponde al dato corrispondente alla finestra temporale specificata.

La struttura dell'URL utilizzato per ottenere i dati è la seguente:

`https://<dominio>/chart/api/datasources/proxy/uid/<dashboardID>/api/board/<boardID>/unit/<query>?ts=<timeSpan>&avg=<avg>`

- `dashboardID`: identificativo della dashboard di Grafana (vd. 5.1 - *Datasource*)
- `boardID`: parametro memorizzato nel localStorage che identifica il sensore (vd. 4.1.8 - *boardID*)
- `query`: unità di misura rappresentata nel grafico, corrispondente al campo `grafana.unitName` nel file `measures.json`
- `timeSpan`: variabile rappresentante l'aggregazione
- `avg`: finestra temporale

Gauges

I gauge costituiscono gli elementi fondamentali nella visualizzazione dell'insight, fungendo da strumenti intuitivi per comunicare agli utenti gli indici di comfort. La disposizione di tutti i gauge e dei rispettivi indicatori è definita nel file `components/Dashboard/DashGauges.jsx`. Questo, non solo determina la struttura della pagina nella visualizzazione dell'insight, ma delinea anche la logica per la visualizzazione selettiva dei cerchi. In particolare, il cerchio associato a una specifica misura

sarà visibile solo se si clicca sul corrispondente gauge o sull'IEQ. Cliccando su qualsiasi altro gauge, i cerchi relativi al comfort selezionato precedentemente verranno nascosti.

Nella versione iniziale dell'applicazione, l'utilizzo di iframe era la prassi per i gauge. Tuttavia, nell'attuale versione, si è preferito adottare una libreria esterna[1], trasferendo a quest'ultima il valore recuperato dall'API, come descritto nella sezione precedente. Questo cambio di approccio è stato motivato dalla necessità di accelerare la visualizzazione dei dati, poiché il recupero e la visualizzazione tramite una libreria esterna risultano più efficienti rispetto al caricamento degli iframe di Grafana.

La scelta è stata indirizzata verso la libreria `react-gauge-component`, ritenuta la più simile alla versione precedente dell'applicazione. Tuttavia, poiché le opzioni di personalizzazione fornite dalla libreria non soddisfacevano appieno le esigenze per ottenere il risultato desiderato, si è optato per la duplicazione del componente specifico all'interno della cartella `lib/GaugeComponent`. Successivamente, il componente è stato importato nel file `Gauge.jsx`. Questa procedura è stata necessaria a causa delle limitazioni riscontrate nella modifica dello sfondo predefinito del gauge, che presentava un grigio troppo intenso per l'estetica complessiva della dashboard. Per apportare tali modifiche, è stato necessario intervenire nel file `arc.ts` nella cartella `lib/GaugeComponent/hooks`

```
//This is the grey arc that will be displayed when the gauge is not full
let secondSubArc = {
    value: 1 - currentPercentage,
    color: "#F4F5F5"
}
```

Grafici

All'interno dell'applicazione, tutti i grafici incorporano pannelli provenienti da Grafana attraverso l'uso di iframe. Come evidenziato nella sezione relativa al file measures.json, ogni misura in questo file è associata a un panelID nel formato JSON. Questo panelID corrisponde all'identificativo del pannello nella specifica istanza di Grafana. Una dettagliata analisi della struttura del file utilizzato per il provisioning della dashboard di Grafana sarà presentata nel capitolo dedicato a Grafana (vd. 5.1 - *Datasource*).

L'URL utilizzato per l'embed segue la seguente struttura:

```
https://<dominio>/chart/d-solo/<dashboardID>/<dashboardName>?orgId=1
    &var-boardId=<boardID>&var-unitName=<unitName>&var-avg=<avg>
    &var-ts=<ts>&panelId=<panelID>
```

- `dashboardID`: identificativo della dashboard di Grafana (vd. 5.1 - *Datasource*)
- `dashboardName`: nome della dashboard nel contesto di Grafana
- `boardID`: parametro memorizzato nel localStorage che identifica il sensore (vd. 4.1.8 - *boardID*)
- `unitName`: unità di misura rappresentata nel grafico, corrispondente al campo `grafana.unitName` nel file `measures.json`

- `avg`: finestra temporale
- `ts`: variabile rappresentante l'aggregazione
- `panelID`: definito sotto il campo `grafana.panelID` del file `measures.json`

La funzione responsabile della composizione di questo URL è situata in `components/Dashboard/GraphFrames.jsx` e si chiama `calcUrl`. Accetta come parametri il nome della misura e la finestra temporale (`avg`). La mappatura tra `avg` e `ts` è definita nello stesso file, nell'oggetto `avgToTimeSpan`, che è strutturato come un dizionario. Ciascuna chiave in questo dizionario è associata a un valore rappresentante la finestra temporale e contiene un oggetto con le seguenti chiavi:

- `ts`: il timestamp in formato stringa
- `ms`: il timestamp in formato millisecondi
- `offset`: parametro utilizzato per calcolare l'ascissa massima del grafico

I parametri `avgToTimeSpan[...].ms` e `avgToTimeSpan[...].offset` rivestono un ruolo cruciale durante la modifica della finestra temporale. Grazie all'implementazione di una versione personalizzata di Grafana, sviluppata da un altro tesista magistrale di Ingegneria Informatica, è stato possibile ottimizzare l'interazione con l'iframe in modo significativo. In particolare, questa personalizzazione ha eliminato la necessità di ricaricare completamente l'iframe ad ogni modifica della finestra temporale.

Il fulcro di questa migliorata gestione si trova all'interno della seguente `useEffect`

```
useEffect(() => {
    const sendMessageToGrafana = () => {
      let iframe = document.getElementById('dashGraph');
      if (iframe != null) {
        const currentTime = Date.now();
        const from = currentTime - avgToTimeSpan[props.time_span].ms;
        const to = currentTime + avgToTimeSpan[props.time_span].offset * 60000;
        const message = {
          variables: [
            { key: "avg",
              value: props.time_span == "RT" ?
                props.time_span.toLowerCase() : props.time_span },
            { key: "ts",
              value: avgToTimeSpan[props.time_span].ts }
          ],
          timeRange: { from: from, to: to }
        }
        iframe.contentWindow.postMessage(message, localUrl);
      }
    }
if (localUrl != "") sendMessageToGrafana();
}, [props.time_span]);
```

Questa viene attivata solo al primo caricamento dell'iframe, quando l'URL dell'embed è ancora nullo. In seguito, le successive modifiche della finestra temporale vengono propagate in modo efficiente a Grafana attraverso l'utilizzo di `contentWindow.postMessage`.

4.1.8 boardID

All'interno dell'applicazione, uno dei parametri fondamentali è il `boardID`, che funge da identificatore del sensore. Questo parametro è utilizzato per recuperare i dati relativi al sensore specifico nella dashboard o per sottoporre il questionario relativo.

La lista dei `boardID` accettati all'interno dell'applicazione è memorizzata nel JSON `/assets/boardIDToSurvey.js`. Il file viene utilizzato principalmente come mappatura tra il `boardID` e il `surveyID`, quindi come associazione tra il numero del sensore e il modello del questionario da utilizzare (vd. 4.1.2 *SurveyJS - surveyID e boardID*). Ogni oggetto presente nel file, però, fornisce ulteriori informazioni per ogni `boardID` presente.

Un esempio:

```
{
  "boardID": 1001,
  "serialNumber": "17120003-00-19/001-000201",
  "sensorNumber": 1,
  "classroomName": "Aula 1P",
  "surveyID": 1
}
```

Il `boardID` può essere definito in due modi:

1. **Query parameter**

   Se il parametro `boardID` è incluso nell'URL, il suo valore viene memorizzato nel localStorage (es. `https://<dominio>?boardID=<boardID>`)

2. **API**

   Se il parametro `boardID` non è presente nell'URL e l'utente è autenticato tramite username e password, viene eseguita una chiamata API per recuperare il `boardID` associato all'ultima compilazione del questionario. Una volta ricevuto il valore `boardID` dalla risposta API, questo viene salvato nel localStorage.

In entrambi i casi, il valore di `boardID` impostato tramite il query parameter ha la priorità rispetto a quello impostato tramite l'API.

Tuttavia, se il `boardID` non è presente nel file `/assets/boardIDToSurvey.js`, l'applicazione reindirizzerà l'utente a una pagina di errore specificata nel file `BoardIDNotFound.jsx`.

4.2 Backend

Il backend dell'applicazione è implementato utilizzando Express e insieme all'applicazione React sono entrambi contenuti e eseguiti all'interno dello stesso container Docker `Web App`. Tanto l'applicazione React quanto il server Express operano sulla stessa porta (5000). Ciò consente di avviare entrambi contemporaneamente con lo script `yarn start` che sottende i comandi `yarn run build && (cd server && yarn start)`.

Il backend è stato suddiviso in tre layer per la gestione delle richieste:

- `index`: rappresenta il punto di ingresso principale del server Express. Contiene la definizione degli endpoint, i middleware, e la gestione generale del flusso dell'applicazione.

- `services`: contengono i file che gestiscono la logica di business dell'applicazione

- `dao`: contengono i file che si occupano della comunicazione diretta con il database

Come appena citato, l'`index` (situato in `client/server/index.js`) è il componente che si occupa della gestione degli endpoint. Nel caso in cui si trovi a gestire una richiesta che non corrisponde ad alcun endpoint, viene attivata una gestione specifica:

```
app.get("/*", (req, res) => {
  res.sendFile(path.join(__dirname, "..", "dist", "index.html"));
});
```

In questo caso, viene inoltrato al client il file `index.html` che si trova nella cartella in cui viene compilata staticamente l'applicazione React, ovvero `client/dist/`. Questo è il meccanismo che sottende la possibilità di visualizzare le diverse route dell'applicazione React in modo parallelo alla gestione degli endpoint del server.

Le chiamate API all'interno dell'applicazione React possono essere distinte in due categorie principali. La prima categoria riguarda le chiamate dedicate alla gestione dell'autenticazione dell'utente, mentre la seconda categoria si occupa della manipolazione e del recupero dei dati dell'applicazione. Tutte le chiamate appartenenti a quest'ultima categoria seguono una struttura del tipo `/api/prv/*`. Prima di procedere con il matching completo di una richiesta in questa seconda categoria, la richiesta stessa è sottoposta a un controllo preliminare:

```
app.all("/api/prv/*", function (req, res, next) {
  try {
    console.log(
      "Auth:",  getAuthentication(req.cookies),
      "Role:",  getRole(req.cookies),
      "User:",  getUser(req.cookies),
      "Email:", getEmail(req.cookies)
    );
    next();
  } catch {
    res.status(401).json("Unauthorized");
  }
});
```

Tutte le informazioni necessarie per questa verifica sono estratte dai cookie impostati dal server di autenticazione. I cookie costituiscono una fonte affidabile e sicura, poiché l'utente non può manipolarli, essendo costantemente sovrascritti dall'AuthServer. Se l'utente non è autenticato, l'accesso a campi specifici scatenerà un errore, portando a un contesto in cui l'utente risulta non autenticato e riceve una risposta `401 Unauthorized`.

Nella sezione successiva si analizzeranno nello specifico le chiamate API disponibili.

Endpoints

- **Submit di un nuovo questionario**

```
POST /api/prv/survey
```

Body della richiesta

- `answers` (object): Oggetto restituito da SurveyJS contenente le risposte del questionario
- `userID` (string): Identificativo dell'utente che ha compilato il sondaggio (ID random per utenti anonimi, email per utenti autenticati)
- `boardID` (string): Identificativo della scheda associata al sondaggio
- `mqttMessage` (string): Messaggio MQTT associato al completamento del sondaggio (approfondimento a fine sezione)
- `key` (string): Chiave di autorizzazione, utilizzata per identificare campagne dei questionari

Esempio richiesta

```
{
    "resultID": "3e453c8e-980f-4aaa-9d0b-6f3d2d86876c",
    "timestamp": "2024-02-01T14:27",
    "boardID": "1001",
    "userID": "username@domain.it",
    "answers": {
        "Q01": "1",
        "Q02.5": [
            "INDOOR AIR QUALITY"
        ],
        "Q02": "",
        "key": null
    }
}
```

Risposte

- `201 Created`: La richiesta è stata elaborata con successo
- `401 Unauthorized`: Richiede autenticazione
- `500 Internal Server Error`: Errore interno del server

- **Recupero questionari di un utente**

```
GET /api/prv/survey/user
```

Esempio risposta


```
[
    {
        "resultID": "3e453c8e-980f-4aaa-9d0b-6f3d2d86876c",
        "timestamp": "2024-02-01T14:27:00.000Z",
        "boardID": 1001,
        "userID": "username@domain.it",
        "answers": "
            {\"Q01\":\"1\",
            \"Q02.5\":[\"INDOOR QUALITY\"],\"Q02\":\"\",\"key\":null}"
    }
]
```


Risposte

- `200 OK`: La richiesta è stata elaborata con successo
- `401 Unauthorized`: Richiede autenticazione
- `500 Internal Server Error`: Errore interno del server

- **Recupero boardID associato all'ultimo questionario compilato**

```
GET /api/prv/survey/boardID/latest
```

Esempio risposta


```
{"boardID":1001,"maxTimestamp":"2024-02-01T14:27:00.000Z"}
```


`maxTimestamp` rappresenta il timestamp dell'ultimo questionario compilato

Risposte

- `200 OK`: La richiesta è stata elaborata con successo
- `401 Unauthorized`: Richiede autenticazione
- `500 Internal Server Error`: Errore interno del server

- **Data dump in un file Excel**

  I dettagli implementativi verranno approfonditi nella sezione 4.4

  ```
  GET /api/prv/survey/all
  ```

  API accessibile solo ad utenti con ruolo `Editor` o `Admin`

  Parametri query (opzionali)

  - `boardID`: Identificativo del boardID. Se non specificato vengono restituiti tutti i boardID, suddivisi in fogli del file Excel
  - `from`: Data di inizio dell'intervallo, se non specificata è utilizzata la data corrente meno un mese
  - `to`: Data di fine dell'intervallo, se non specificata è utilizzata la data corrente

  Risposte

  - Download del file Excel
  - `401 Unauthorized`: Richiede autenticazione
  - `500 Internal Server Error`: Errore interno del server

- **Submit risposte alle domande personali**

  ```
  POST /api/prv/personal
  ```

  Body della richiesta

  - `answers` (object): Oggetto restituito da SurveyJS contenente le risposte alle domande personali
  - `userID` (string): Identificativo dell'utente che ha risposto alle domande personali (ID random per utenti anonimi, email per utenti autenticati)

  Esempio richiesta

  ```
  {
      "answers": {"p01":"Male","p02":null,"p03":null,"p04":null},
      "userID": "username@domain.it"
  }
  ```

  Risposte

  - `201 Created`: La richiesta è stata elaborata con successo
  - `401 Unauthorized`: Richiede autenticazione
  - `500 Internal Server Error`: Errore interno del server

- **Recupero informazioni personali**

```
GET /api/prv/personal
```

Esempio risposta

```
{
    "userID": "username@domain.it",
    "timestamp": "2024-02-05T10:18:49.000Z",
    "answers": {"p01":null,"p02":null,"p03":null}
}
```

Risposte

  - `200 OK`: La richiesta è stata elaborata con successo
  - `401 Unauthorized`: Richiede autenticazione
  - `500 Internal Server Error`: Errore interno del server

- **Recupero dati utente**

```
GET /userinfo
```

Esposizione dell'endpoint del server di autenticazione (vd. 3 - *AuthServer*)

- **Logout**

```
GET /api/prv/personal
```

Esposizione dell'endpoint del server di autenticazione (vd. 3 - *AuthServer*)

MQTT

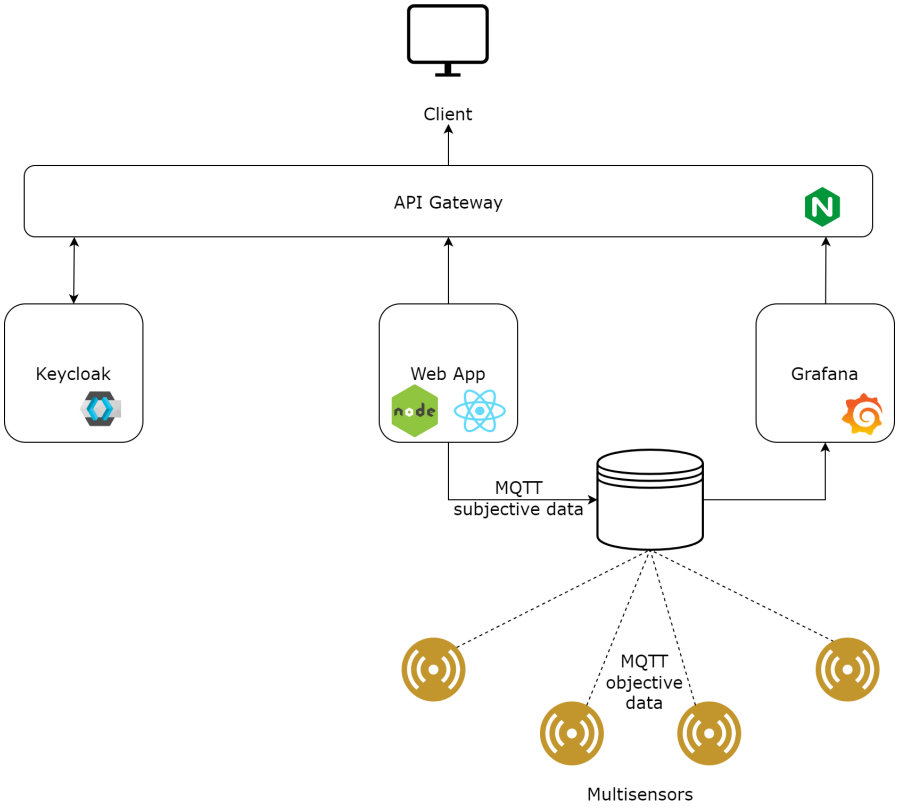

Figura 4.13. Diagramma dati MQTT

Il protocollo MQTT, utilizzato come intermediario nella comunicazione, facilita il trasferimento dei dati soggettivi raccolti al termine del questionario al relativo database esterno contenente le informazioni oggettive. Questa operazione è gestita tramite il dao, all'interno della funzione `newSurvey`, la quale si occupa di pubblicare i dati una volta inseriti nel database locale.

```
let client = mqtt.connect("wss://test.mosquitto.org:8081");

const newSurvey =
    async (resultID, timestamp, boardID, userID, answers, mqttMessage) => {
        //...
        let QoS = { qos: 1 };
        let topic = "prometeo/" + boardID + "/questionnaire";
        console.log(mqttMessage)
        client.publish(topic, JSON.stringify(mqttMessage), QoS);
}
```

L'oggetto da trasmettere, denominato `mqttMessage` viene generato dalla web app mediante l'esecuzione della funzione `evaluateComfort` e poi passato come parametro alla chiamata API `POST /api/prv/survey`. Questo messaggio MQTT contiene sia i dati soggettivi ottenuti dalla compilazione del questionario (indicati con `_sbj`), sia i dati oggettivi calcolati al momento della compilazione (indicati con `_obj`).

Un esempio del messaggio MQTT è il seguente:

```
{
  thermal_comfort_sbj: null,
  visual_comfort_sbj: 100,
  acoustic_comfort_sbj: null,
  air_quality_sbj: null,
  env_quality_sbj: 100,
  thermal_comfort_obj: null,
  visual_comfort_obj: null,
  air_quality_obj: 100,
  acoustic_comfort_obj: null,
  env_quality_obj: 51.62085555555554,
  timestamp: '2024-02-06T15:36:18.639'
}
```

La trasmissione dei dati tramite MQTT riveste un'importanza significativa per la loro successiva visualizzazione nei grafici della dashboard, dove saranno rappresentati mediante cerchi gialli.

4.3 Gestione ruoli

L'applicazione gestisce una varietà di ruoli utente, inizialmente differenziati in base al metodo di autenticazione utilizzato per accedere all'applicazione e successivamente in base al ruolo assegnato all'interno del realm di Keycloak.

- **Utente autenticato**: accesso tramite username e password. Associato alla `LoggedRoute`
  - Admin: amministratore dell'applicazione
  - Editor: utente dell'applicazione con privilegi, come il dump dei dati nel file excel (vd. 4.4 - *Data dump ed Excel*) o accesso alle route di Keycloak e Grafana
  - Viewer: utente base
- **Utente anonimo**: autenticazione tramite il query parameter `key` (vd. 3 - *Auth-Server*). All'utente verrà associato un `uuid` utilizzato come codice identificativo (al pari della email per gli utenti autenticati).

  Può essere di due tipi
  - Student: associato alla `AnonymousRoute`
  - Private: associato alla `ProtectedRoute`

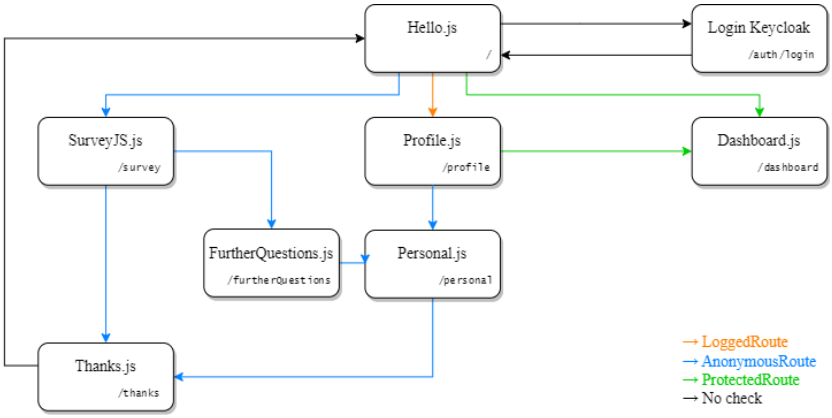

Figura 4.14. Autorizzazione delle route

La distinzione dei ruoli è utilizzata per concedere l'accesso alle varie route in base al ruolo specifico assegnato agli utenti.

	`LoggedRoute`	`ProtectedRoute`	`AnonymousRoute`
Utente autenticato	Si	Si	Si
Utente anonimo - `Private`	No	Si	Si
Utente anonimo - `Student`	No	No	Si

Tabella 4.1. Route e ruoli

Se un utente tenta di accedere a una route non autorizzata, l'applicazione lo reindirizzerà automaticamente alla pagina di login.

I componenti `LoggedRoute` e `AnonymousRoute` sono definiti in due file distinti aventi lo stesso nome e memorizzati sotto la cartella `components/`. Vengono quindi utilizzati nel file `App.jsx` come wrapper delle route.

In seguito, un estratto esemplificativo:

```
<Route exact path="/profile" element={<LoggedRoute /* props */ />}>
    <Route path="/profile" element={<Profile /* props */ />}/>
</Route>

<Route exact path="/dashboard" element={<ProtectedRoute /* props */ />}>
    <Route path="/dashboard" element={<Dashboard /* props */ />}/>
</Route>

<Route exact path="/personal" element={<AnonymousRoute /* props */ />}>
    <Route path="/personal" element={<Personal /* props */ />}/>
</Route>
```

4.4 Data dump ed Excel

Nel contesto amministrativo dell'applicazione, è emersa la necessità di ottenere un report completo dei questionari completati, accompagnato dalle informazioni personali degli utenti che li hanno compilati. Per soddisfare tale esigenza, è stata sviluppata e implementata un'API dedicata allo scaricamento di tutte le informazioni rilevanti in un unico file Excel.

È già stato menzionato l'endpoint corrispondente a questa API nella sezione riguardante il backend (vd. 4.2 - *Backend*), ovvero

```
POST /api/prv/survey/all
```

Tuttavia, questo endpoint è accessibile solamente agli utenti autenticati che possiedono i privilegi di `Editor` o `Admin`. Questa restrizione è fondamentale per proteggere e controllare l'accesso a informazioni sensibili.

Nel caso in cui la richiesta sia eseguita con successo, verrà avviato il download di un file Excel il cui nome sarà basato sul timestamp del momento in cui è stata effettuata la richiesta, seguendo il formato `yyyMMdd_HHmmSSSS`.

La creazione del file Excel è gestita dalla funzione `createWorkbook` presente in `client/server/services/surveyService.js`. Questa funzione sfrutta la libreria `SheetJS` [6], la quale offre varie funzionalità, tra cui la possibilità di creare un workbook da zero e gestire i worksheet e le intestazioni di ciascuno.

L'approccio adottato consiste nell'estrarre un set di tutti i `boardID` per cui si è stato sottomesso un questionario negli ultimi 30 giorni e creare un foglio di lavoro per ciascuno di essi. L'intestazione di ogni foglio di lavoro includerà campi fissi che contengono informazioni comuni a tutti i questionari, oltre a campi direttamente correlati alla struttura specifica del questionario associato al boardID.

I campi fissi sono:

- `boardID`: identificativo del sensore
- `timestamp`: istante di completamento del questionario
- `resultID`: identificativo univoco, utilizzato come chiave primaria del database
- `userID`: identificativo univoco dell'utente (email se utente autenticato, uuid creato dal client in caso di utente anonimo)
- `key`: valore presente solo in caso di utente anonimo, corrisponde alla chiave utilizzata per l'autenticazione

I campi variabili corrisponderanno all'elenco dei nomi delle domande presenti nel modello del questionario e ai nomi delle domande personali. Le prime avranno un formato del tipo `Qxx`, mentre le seconde `pxx`.

In questo passaggio si evidenzia l'importanza di includere, in fase di submit delle risposte al questionario e alle domande personali, anche i nomi delle domande non valorizzate. Infatti, il server non dispone delle informazioni sulla struttura specifica del questionario e può dedurla solo dall'unione di tutti i nomi delle domande a cui sono

state fornite risposte. Senza questa precauzione, se una domanda X non ricevesse mai una risposta nel tempo, non verrebbe mai inclusa nel file risultante. Tuttavia, includendo tutti i nomi delle domande, si garantisce che ogni boardID avrà un foglio di lavoro con una struttura standard e ripetibile.

Come precedentemente esaminato nella descrizione dell'API nella sezione dedicata al backend, è da sottolineare che tale API offre la flessibilità di modificare l'intervallo di tempo o di limitare il download dei dati solo a un boardID specifico tramite l'aggiunta di query parameters.

Capitolo 5

Grafana

Grafana [3] è una piattaforma interattiva open source per la visualizzazione dei dati sviluppata da Grafana Labs. Sebbene sia comunemente associata al monitoraggio dello stato delle applicazioni e dei servizi, nel contesto del nostro progetto è stata utilizzata come strumento per la visualizzazione dei dati provenienti dai sensori ambientali. In particolare, è stata sfruttata come gestore dei grafici e come proxy per l'acquisizione dei dati proveniente dal database esterno. Nelle prossime sezioni si analizzeranno nello specifico i file di provisioning utilizzati per inizializzare il container Docker dell'istanza di Grafana a cui farà accesso l'applicazione.

5.1 Datasource

Uno dei componenti più importanti da definire nei file di configurazione è la sorgente dati da cui Grafana trarrà le informazioni per generare i grafici e svolgere il suo ruolo di proxy. Un ostacolo affrontato è stato l'assegnazione di un ID variabile alla sorgente dati. Tale ID viene utilizzato come identificativo della sorgente da utilizzare nei pannelli della dashboard, ma non è modificabile tramite l'interfaccia web. Per questo motivo, nel file di configurazione, non era possibile associare ad ogni pannello l'identificativo della sorgente dati in maniera costante, siccome ad ogni reinizializzazione del container, la sorgente dati veniva creata con un identificativo diverso.

In seguito analizzeremo nello specifico i dettagli implementativi adottati per il raggiungimento di questo obiettivo. Per iniziare, è stata creata una directory chiamata `provisioning/`, contenente tutti i file essenziali per l'inizializzazione dei dati dell'istanza di Grafana. In questa sezione specifica, ci si concentrerà sul file `provisioning/datasources/datasources.yaml`, che contiene i parametri di connessione alla sorgente dati.

```
apiVersion: 1

datasources:
  - name: Thomas API 2
    type: marcusolsson-json-datasource
    url: http://web:5002
    editable: true
    basicAuth: true
    basicAuthUser: "user"
    isDefault: true
    uid: d1777f53-05fe-4a30-a97a-e15b40c4ff4e
    version: 9
    jsonData:
      database: Grafana
      maxIdleConnsAuto: true
```

In particolar modo è interessante evidenziare il tipo di sorgente e, come si è introdotto prima, l'`uid` costante. Per quanto riguarda il tipo di sorgente dati, è stato impiegato il plugin `marcusolsson-json-datasource` specializzato nella gestione di sorgenti di tipo JSON. Questo, viene installato durante la fase di inizializzazione dell'istanza di Grafana, come specificato nel `docker-compose.yml`, con la seguente direttiva:

```
GF_INSTALL_PLUGINS=marcusolsson-json-datasource
```

5.2 Descrizione pannelli

Ogni grafico a cui fa riferimento l'applicazione è un iframe contenente uno dei pannelli definito nella dashboard `prometeo` all'interno dell'istanza di Grafana. Ciascuno di questi fa riferimento ad una misura diversa.

Come si è analizzato nella sezione precedente, tutte le API attingono dalla sorgente dati definita tramite provisioning. I pannelli che trattano come unità di misura i vari tipi di comfort, effettueranno due tipi di chiamate: una per recuperare i dati oggettivi e una per i dati soggettivi. I primi verranno rappresentati come negli altri pannelli, ovvero con una linea spezzata, mentre i secondi con dei cerchi gialli.

Per creare un nuovo pannello o per modificare un pannello già esistente, è possibile sfruttare l'interfaccia web di Grafana ed accedere alla dashboard contenente i pannelli. Per ciascuno di questi si può modificare l'aspetto, il colore e le chiamate API da cui recupera i dati.

Ogni qual volta si effettua una modifica, è bene ricordarsi di esportare il JSON della dashboard e di aggiornare il file `provisioning/dashboards/dashboard.json` per evitare di perdere le modifiche nel caso in cui si dovesse ribuildare il container di Grafana da zero.

Capitolo 6

Database

Il container Docker del database fa uso dell'immagine MySQL e colleziona i dati dei questionari inviati da tutti gli utenti. La persistenza di tali dati è assicurata tramite l'utilizzo di un volume Docker, il quale è definito nel file `docker-compose.yml` come segue:

```
volumes:
    [...]
    - mysql_data:/var/lib/mysql
```

Questo volume, denominato `mysql_data`, è configurato per mappare la directory `/var/lib/mysql` all'interno del container MySQL. Grazie a questa configurazione, i dati all'interno del database persistono anche quando il contenitore Docker viene riavviato o ricreato, garantendo la conservazione e l'integrità delle informazioni immagazzinate nel sistema.

Durante il processo di creazione del container, viene eseguito uno script SQL che verifica l'esistenza del database e delle relative tabelle. Concretamente, tale codice viene eseguito solamente nel caso in cui il volume `mysql_data` non esista. Lo script in questione è chiamato `initQueries.sql` e nel `docker-compose.yml` file viene mappato all'interno della directory nel seguente modo:

```
volumes:
    - ./database/initQueries.sql:/docker-entrypoint-initdb.d/initQueries.sql
    [...]
```

Questa configurazione permette l'esecuzione automatica dell'inizializzazione all'avvio del container, garantendo che il database venga configurato correttamente.

Il file `initQueries.sql`:

```
CREATE DATABASE IF NOT EXISTS prometeo_db;

USE prometeo_db;

CREATE TABLE IF NOT EXISTS SURVEYS (
  resultID VARCHAR(255) NOT NULL PRIMARY KEY, --uuid
  timestamp TIMESTAMP NOT NULL,
  boardID INT NOT NULL,
  userID VARCHAR(255) NOT NULL, --uuid anonimo o email
  answers TEXT -- JSON contenente risposte
);

CREATE TABLE IF NOT EXISTS PERSONAL (
  userID VARCHAR(255) NOT NULL PRIMARY KEY, --uuid anonimo o email
  timestamp TIMESTAMP NOT NULL,
  answers TEXT -- JSON contenente risposte
);
```

Nella porzione di codice soprastante si possono anche osservare le strutture delle due tabelle del database.

Capitolo 7

Keycloak

Nell'ambito della gestione delle identità e degli accessi, si è scelto di utilizzare Keycloak[5], una soluzione open source che funge da robusta infrastruttura per l'autenticazione, l'autorizzazione e la gestione degli utenti. All'interno del nostro contesto, Keycloak è stato implementato all'interno di un container, interagendo con il server di autenticazione assicurando così una gestione sicura ed efficiente delle autorizzazioni degli utenti per le applicazioni e i servizi coinvolti.

La scelta di utilizzare un Identity Provider open source piuttosto che uno esterno fornisce la possibilità di personalizzare il processo di autenticazione, come per esempio tramite plugin.

Nello specifico, ne sono stati implementati due per migliorare la funzionalità e l'aspetto del sistema. L'import di questi plugin contenuti nella directory `plugins/` sotto forma di file JAR avviene nel docker-compose file, in modo tale da poterli utilizzare all'interno del container.

```
volumes:
    [...]
    - ./keycloak/plugins/.:/opt/keycloak/providers
```

1. **Keycloakify**[4] è stato aggiunto principalmente per motivi estetici: consente di personalizzare il tema delle schermate di autenticazione in modo da renderle coerenti con il resto del sistema.

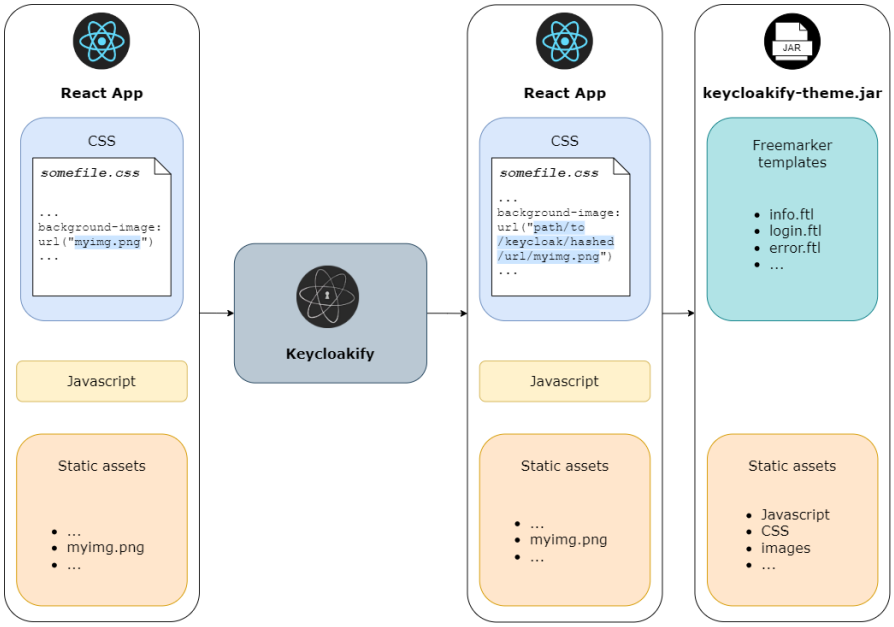

Figura 7.1. Creazione JAR di Keycloakify

Questo plugin è stato sviluppato internamente e consente di apportare modifiche estetiche tramite un file JAR contenente il nuovo tema da applicare. Tale file è stato generato all'interno del progetto e successivamente caricato nel container.

2. **Keycloak mail whitelisting**[2] introduce l'abilità di inserire un controllo sui domini mail e sulle mail specifiche accettate durante la fase di registrazione. Il progetto viene scaricato e, seguendo lo stesso processo di Keycloakify, viene creato un file JAR da inserire nella cartella locale `plugins/`. Da qui, è possibile modificare la whitelist da interfaccia web o tramite il file `realm.json` inserendo una lista di domini o di mail accettate separate dai caratteri `##`.

- Interfaccia web accedendo a `https://<dominio>/keycloak`

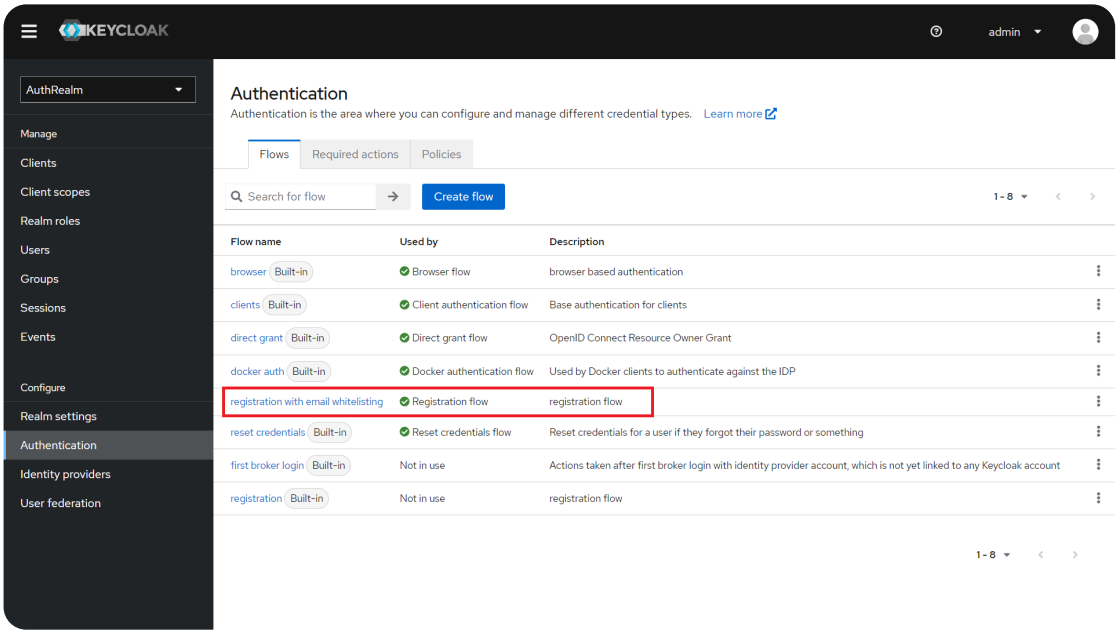

Figura 7.2. Accesso al plugin di whitelisting

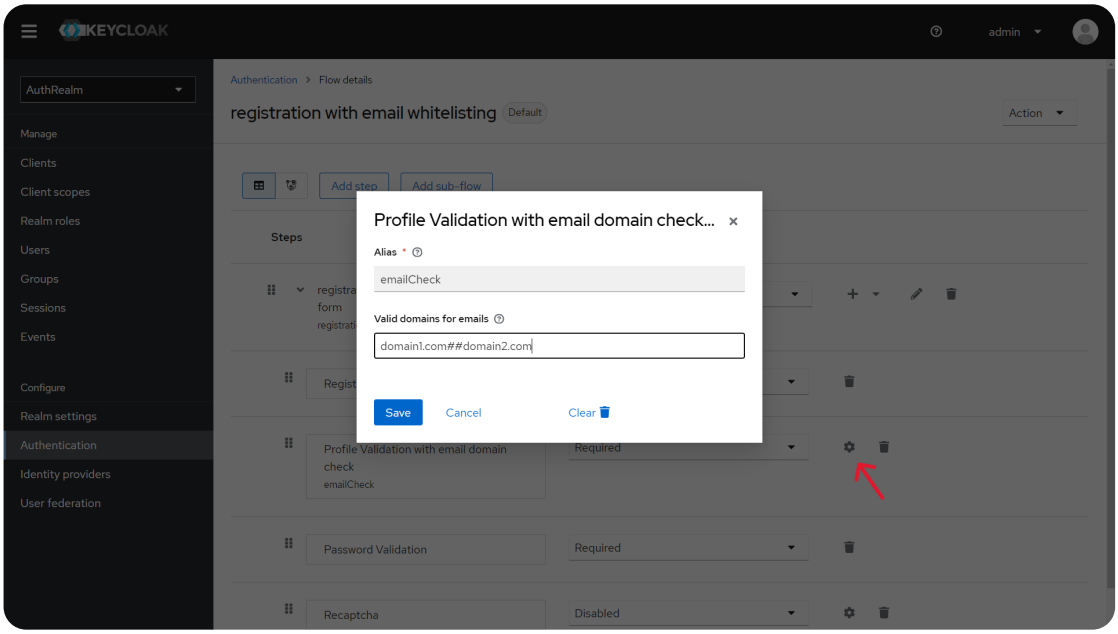

Figura 7.3. Modifica lista domini accettati

- File `realm.json`

```
{
    "id" : ...,
    "realm" : "MyRealm",
    ...
    "authenticatorConfig" : [
        ...
        {
            "id" : "53d764c6-aa81-4abd-b089-1f745d5eb01c",
            "alias" : "emailCheck",
            "config" : {
              "validEmails" :
                "email1@studenti.polito.it##email2@studenti.polito.it",
              "validDomains" :
                "domain1.com##domain2.com"
            }
        },
    ...
    ],
}
```

In questo esempio, `dominio1.com` e `dominio2.com` sono i domini accettati, mentre `mail1@dominio.com` e `mail2@dominio.com` sono email specifiche accettate.

Capitolo 8

Modularità

Come descritto nel capitolo 2 - *Descrizione della piattaforma PROMET&O*, ogni sottorete può essere facilmente replicabile all'interno del contesto della piattaforma PROMET&O in modo tale da ottenere diverse versioni dell'applicazione. In questo modo è possibile separare i vari contesti su domini distinti. Per esempio, uno specifico utente può avere livelli diversi di autorizzazione a seconda del dominio in cui è inserito. Allo stesso modo, è possibile personalizzare lo stile e i dati visualizzati nelle dashboard di ogni singola replica.

In questo capitolo si analizzeranno i diversi passaggi da seguire nel caso in cui si voglia clonare una delle sottoreti in questione.

Prima di tutto, è necessario copiare il contenuto di una delle sottoreti nsXX, assegnarle un nome diverso da quelli già presenti e apportare delle modifiche per ottenere una corretta configurazione.

La prima modifica da effettuare riguarda il file `.env` dove sono contenuti tutti i dati sensibili. In particolar modo bisogna cambiare il parametro `DOMAIN`, inserendo il nuovo nome del dominio della sottorete. Successivamente si deve sostituire il nome del dominio all'interno del file `realm.json` in quanto Keycloak non fornisce una parametrizzazione automatica.

Infine, bisogna apportare delle modifiche al container Ingress in quanto quest'ultimo si occupa dello smistamento delle richieste verso i vari sottodomini. Per fare ciò è necessario aggiornare nello specifico due file: `default.conf` e `docker-compose.yml`.

Nel primo si deve aggiungere il seguente codice, sostituendo opportunamente i dati contenuti tra le parentesi angolari:

```
server {
    listen 80;
    listen [::]:80;

    server_name <domainName>;

    resolver <privateIP>;

    location / { return 301 https://$host$request_uri; }

    access_log /dev/stdout;
    error_log  /dev/stderr;
}

server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;

    server_name <domainName>;

    ssl_certificate /etc/ssl/certs/<certificate>.pem;
    ssl_certificate_key /etc/ssl/certs/<domain_key>.pem;
    ssl_dhparam /etc/nginx/dhparam.pem;

    include /etc/nginx/includes/ssl-common.conf;

    location / {
        include /etc/nginx/includes/proxy.conf;
        proxy_pass http://<subnetName>-app-1;
    }

    access_log /dev/stdout;
    error_log /dev/stderr;
}
```

Nel secondo, invece, si deve aggiungere il nome della nuova sottorete come segue:

```
networks:
  [...]
  nsXX:
    name: nsXX_default
    external: true
```

L'obiettivo della modularizzazione del progetto è stato dunque quello di semplificare il più possibile la duplicazione di sottoreti e la realizzazione di un ambiente personalizzato. In questo modo, chiunque desideri utilizzare nel proprio contesto lavorativo la piattaforma PROMET&O avrà a disposizione un template a cui apporre modifiche minime in modo tale da renderlo su misura.

Conclusioni

In seguito a quanto spiegato, si può dire raggiunto l'obiettivo dell'indipendenza dai servizi di Amazon. Ciò è stato conseguito grazie all'implementazione di un backend locale tramite Express e la delega di alcune funzionalità ai microservizi appartenenti alla rete. In particolare è stato utilizzato Keycloak per il controllo dell'autorizzazione utente, l'OAuth server per l'autenticazione, Grafana per il recupero dei dati e un API Gateway Nginx per lo smistamento delle richieste verso i diversi container.

La migrazione della piattaforma in un contesto locale ha permesso di avere libera scelta sulle tecnologie da utilizzare. Si sono preferite, pertanto, le soluzioni studiate durante i corsi accademici. Questa scelta permetterà, in ottica di future implementazioni, il risparmio di tempo necessario ad apprendere il funzionamento di Amazon Web Services così da potersi concentrare immediatamente sull'implementazione di nuove funzionalità.

Nel processo della migrazione, si è posta particolare attenzione anche alla replicabilità e alla modularità della stessa. Ogni sottorete, infatti, può essere duplicata e adattata al contesto d'uso in modo tale da renderla praticamente su misura. Questo concetto può essere applicato sia a livello della rete dei container, basandosi sui servizi coinvolti, che a livello della singola Web App. Dal punto di vista dell'utente finale, sarà possibile sfruttare la capacità dell'applicazione di supportare diversi questionari e di riadattare la dashboard inserendo tutte le misure di interesse. Invece, dal punto di vista degli sviluppatori che si occuperanno del progetto in futuro, la riorganizzazione del codice permetterà una maggiore leggibilità e comprensione dello stesso, oltre che una maggiore facilità di implementazione.

Bibliografia

[1] Antonio Lago. *React gauge Component*. url: https://github.com/antoniolago/react-gauge-component.

[2] Cedric Couralet. *Keycloak - Email domain validation for registration*. url: https://github.com/micedre/keycloak-mail-whitelisting.

[3] Grafana Labs. *Grafana: The open-source analytics and monitoring solution for every database*. url: https://grafana.com/.

[4] Joseph Garrone. *Keycloak theming - Made easy. With React.* url: https://www.keycloakify.dev/.

[5] Keycloak. *Keycloak - Open Source Identity and Access Management*. url: https://www.keycloak.org/.

[6] SheetJS. *SheetJS Community Edition*. url: https://docs.sheetjs.com/. (accessed 28-November-2023).