POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Architettura di rete per il deploy di un cluster di applicazioni web

Relatore

Prof. Antonio SERVETTI

Candidato

Vittorio ARPINO

Ottobre 2023

Indice

Elenco delle Figure

Elenco delle Tabelle

Introduzione

Il progetto di Tesi riguarda la realizzazione di un processo di autenticazione che sia il più indipendente possibile rispetto alla piattaforma Web che ne vuole fare uso. In particolare, il suo sviluppo viene posto all'interno della realizzazione di una applicazione web per la visualizzazione e la raccolta dei dati sulla qualità ambientale mediante l'utilizzo di sensori. La finalità del progetto è di andare ad inserire il processo di autenticazione su una piattaforma web, in precedenza, già sviluppata nel modo più semplice possibile. L'utente dovrà essere in grado di visualizzare dei grafici messi a disposizione da una piattaforma esterna di raccolta solo se è autenticato ed è autorizzato a visualizzarli. Per quanto possano essere simili i termini di autenticazione e autorizzazione, bisogna precisare che sono due cose ben diverse. Infatti, in ambito di sicurezza dei sistemi informativi, l'autenticazione rappresenta il processo attraverso il quale l'utente è in grado di identificarsi rispetto terzi; mentre per autorizzazione si intende il processo mediante il quale viene effettuata la verifica sulla possibilità da parte dell'utente di accedere la risorsa. L'autenticazione, da un nostro punto di vista, viene gestita completamente da un provider di autenticazione come Google, Twitter, Facebook ecc. ma nel nostro caso faremo riferimento a Keycloak. L'autorizzazione, invece, viene gestita direttamente dalla piattaforma di raccolta dati che definisce una serie di ruoli, Infatti, in base al ruolo associato, si ha la capacità o meno di accedere ad una risorsa. Mentre l'autorizzazione può essere semplice da gestire, non è così per l'autenticazione dato che esistono innumerevoli processi che danno più o meno le stesse garanzie. Un altro punto focale su cui si è concentrati, è il processo di registrazione, in particolare si vuole che l'utente sia in grado di registrarsi autonomamente. Tale processo sarà supportato dallo stesso provider che fornisce l'autenticazione realizzando ulteriori requisiti come: l'invio della e-mail di verifica oppure la validazione del dominio dell'e-mail stessa.
Da un punto di vista architetturale, lo sviluppo del processo di autenticazione viene considerato come un micro-servizio con un proprio database degli utenti e delle API pubbliche che nascondono la logica di business. La motivazione che ci porta verso questa soluzione architetturale, nonostante l'introduzione di maggiore complessità nello sviluppo della piattaforma, è sicuramente la possibilità di identificare un servizio come un entità indipendente rispetto ad altri diversamente da uno sviluppo monolitico dell'applicazione. Infatti, gli ostacoli principali a cui bisogna prestare attenzione durante lo sviluppo è l'enorme eterogeneità di lavoro dietro alle applicazioni web rendendo difficile sia l'integrazione di nuove "features" che rimanere al passo con le nuove tecnologie imposte dal mercato. Per tale motivi, il processo di autenticazione verrà considerato come una singola entità con un proprio database degli utenti e con una propria interfaccia di API pubblica e il suo sviluppo si baserà su tecnologie ampiamente utilizzate ai fini della manutenibilità e della scalabilità.

Nel prossimo capitolo, descriverò in dettaglio il progetto, concentrandomi sull'architettura della piattaforma e sui requisiti che devono essere soddisfatti durante la sua implementazione. Esplorerò le diverse componenti del sistema e le interazioni tra di esse, evidenziando le scelte progettuali che hanno guidato il suo sviluppo. Successivamente, esaminerò le tecnologie più utilizzate per implementare casi d'uso simili, con particolare attenzione all'applicazione di tali tecnologie nello specifico contesto del progetto in questione. Analizzerò i punti di forza e di debolezza di ciascuna tecnologia, nonché considerazioni pratiche per il loro uso efficace nell'implementazione del progetto. Infine, esporrò le possibili estensioni del progetto, delineando le opportunità per arricchire ed espandere le funzionalità della piattaforma. Discuterò i potenziali sviluppi futuri per i quali il lavoro di questa tesi ha posto le basi, evidenziando le potenziali sfide e gli obiettivi da raggiungere nell'evoluzione del sistema. Nel complesso, questa tesi offrirà un'ampia panoramica della piattaforma proposta, presentandone sia l'aspetto tecnico e architettonico, sia le sue prospettive di sviluppo futuro. L'obiettivo è quello di fornire una visione completa e dettagliata del progetto, aiutando i lettori a comprenderne appieno la rilevanza, le sfide affrontate e le opportunità che offre per migliorare e ottimizzare l'area di interesse.

Obiettivi della tesi

Nel seguente capitolo, verranno esaminati gli scopi fondamentali della tesi, suddividendo le diverse parti da sviluppare e implementare in base al loro contributo alla creazione della piattaforma finale.

Progettazione complessiva del sistema

Lo sviluppo della Tesi ha come obiettivo principale la realizzazione di un processo di autenticazione e autorizzazione che sia agnostico rispetto all'Applicazione Web già sviluppata. Il contesto di sviluppo è legato alla presenza di diverse sottoreti, tutte con gli stessi bisogni computazionali ma con utenti diversi a cui sono assegnati ruoli diversi che nell'ambito del sistema di raccolta dati siano gli stessi. L'obiettivo è dare la possibilità a terzi di replicare nel modo in maniera semplice la sottorete che verrà personalizzata in base alle proprie esigenze. Un'idea del design ad alto livello in cui porre il processo di autenticazione è il seguente:

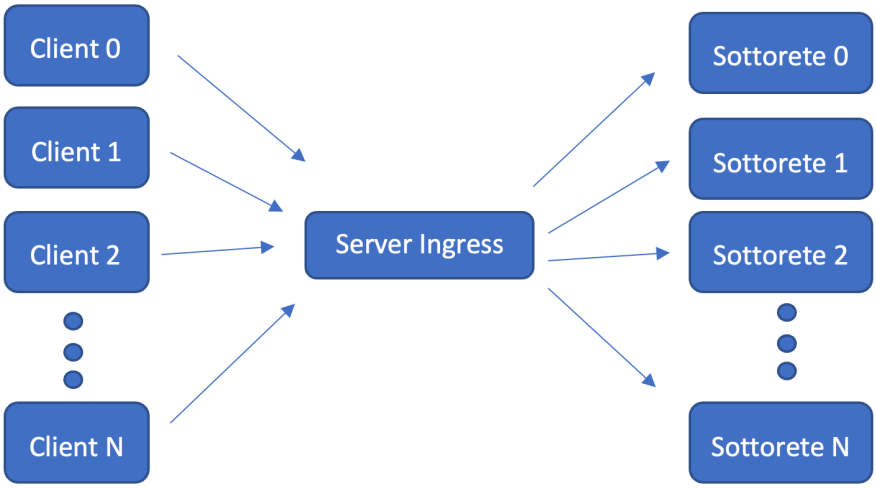

Figura 1. Design di alto livello dell'architettura complessiva

Un generico utente N può accedere all'applicazione Web mediante un certo URL (immaginiamoci che la comunicazione avvenga mediante HTTPS) e può visualizzare i grafici solo se autenticato. Il Server Ingress rappresenta il punto di contatto con il quale l'utente potrà contattare una delle sottoreti e, quando vorrà accedere alla dashboard del sistema di raccolta oppure all'applicazione Web per visualizzare un grafico, dovrà effettuare il processo di autenticazione. Internamente, la sottorete N sarà composta da diverse entità che comunicano tra di loro al fine di fornire realizzare la serie di obiettivi che sono stati predisposti.

La progettazione ad alto livello di ogni sottorete può essere quindi riassunta nel seguente grafico:

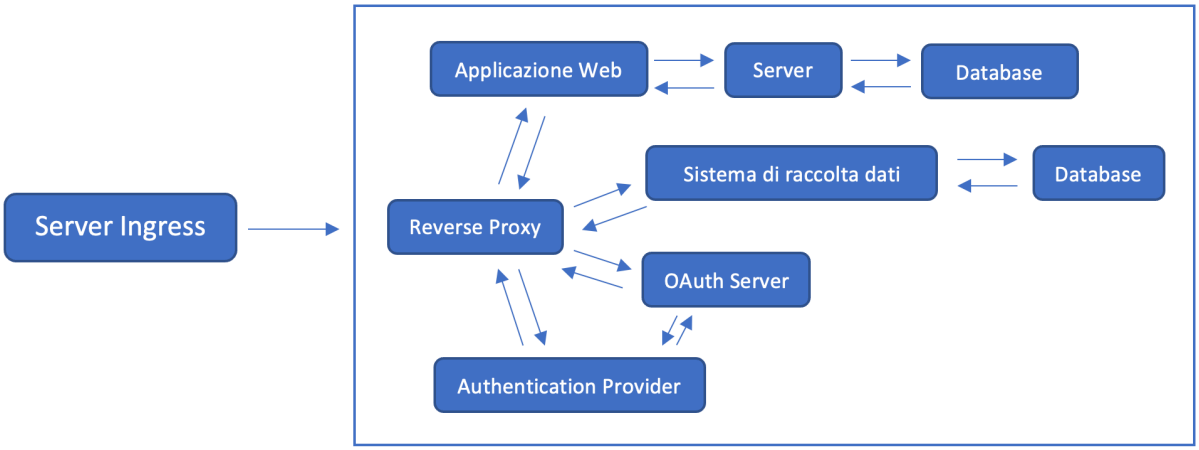

Figura 2. Architettura della sottorete

Come possiamo vedere dal grafico, l'utente accede a un determinato sottodominio mediante il Server Ingress e il reverse proxy interno fa visualizzare l'Applicazione Web per un determinato dominio sul "path" principale. La comunicazione tra tutti i componenti avviene con l'uso del protocollo HTTP e il Server Ingress funge da terminatore SSL. Ciò significa che la comunicazione Utente - Server Ingress avviene mediante HTTPS e la comunicazione tra le entità interne alle sottoreti è mediante HTTP. La motivazione dietro a questa soluzione è legata alla necessità di nascondere e mantenere tutte le informazioni di stato, mediante canale SSL, tutta la comunicazione Utente - Server Ingress (tra l'altro vi è anche la necessità di verifica dei certificati da parte del Server in modo da essere sicuro di star dialogando con il "vero" server e non con un "falso"). Internamente alla sottorete non vi è la necessità di "fidarsi" di un componente dato che si parte dal presupposto che all'interno del dominio ogni componente è fidata e, ai fini dell'ottimizzazione, è inutile mantenere lo stato di diversi canali di comunicazione oppure presentare il certificato per essere sicuri per fidarsi. Notiamo, però, che la comunicazione tra Sistema di raccolta dati - Database non ha bisogno di passare per il reverse proxy perché non interessa alle altre componenti ed inoltre ci immaginiamo che il numero di sensori a cui bisogna fare riferimento sia molto elevato per cui non ci possiamo affidare alla memoria locale del sistema ma a un Database Management System (DBMS) esterno che raccoglie i dati provenienti da sensori diversi e li rende disponibili al Sistema di Raccolta per il rendering grafico. Ulteriormente, l'Applicazione Web ha un suo Database e un suo Server perché, oltre a garantire la visualizzazione grafica dei dati raccolti, avrà anche ulteriori scopi come la compilazione di questionari sulle temperature ambientali. Ovviamente, anche in

questo caso, le altre componenti sono all'oscuro di questa comunicazione dato che i dati e le API non interessano direttamente le altre componenti della sottorete.
Partendo da questo contesto ci siamo soffermati su diversi obiettivi nell'ambito di sviluppo del progetto di Tesi:

1. Progettazione del sistema di autenticazione e autorizzazione
   Il processo di autenticazione/autorizzazione può essere visto come l'obiettivo principale da raggiungere. L'idea è quella di fornire un processo che sia il più possibile semplice e facile da integrare su Applicazioni Web esistenti garantendo anche la possibilità di visualizzare dashboard messe a disposizione sia dall'Identity Provider, per la gestione degli utenti e personalizzazioni del processo di autenticazione; che dal Sistema di Raccolta, per la visualizzazione dei grafici.

2. Fornire un processo di registrazione
   Il processo di registrazione è stato un obiettivo secondario rispetto al processo di autenticazione e che si basa sulla gestione della verifica della e-mail inserita dall'utente, dall'assegnazione del ruolo base e dalla validazione della e-mail che ci immaginiamo far parte dello stesso dominio a cui l'utente vuole accedere. Ulteriormente, ai fini dello sviluppo si è cercato di implementare un processo di whitelisting delle e-mail per la registrazione.

3. Gestire la visualizzazione dei grafici
   L'obiettivo è, basandosi sul risultato del processo di autenticazione, garantire la visualizzazione dei dati all'interno dell'applicazione Web. Inoltre, in questo caso, l'idea è anche di fornire all'Applicazione Web la possibilità di effettuare query personalizzabili che permettano, per esempio, di visualizzare solo i dati ottenuti da un sensore, scartando gli altri.

4. Integrazione della Applicazione Web
   Vista l'enorme eterogeneità delle tecnologie usate per lo sviluppo di applicazioni Web, si è cercato di rendere il lavoro il più indipendente possibile. In particolare, tale lavoro di Tesi nasce proprio basandosi sulla presenza di un'Applicazione Web già esistente ed è stato poi integrato con il lavoro di Tesi svolto dagli altri partecipanti al progetto. L'idea è stata quindi quella di integrare il lavoro svolto sull'autenticazione su un'Applicazione Web sviluppata da terzi.

Stato dell’arte

Nel capitolo successivo, esamineremo lo stato attuale delle tecnologie sviluppate per semplificare il processo di autenticazione all'interno delle applicazioni web. Saranno approfondite le tecnologie impiegate nel corso di questa tesi e verranno analizzate le motivazioni che hanno spinto verso l'adozione di esse, mettendone in luce i vantaggi e gli svantaggi correlati.

Autenticazione e Autorizzazione

La gestione del processo di autenticazione all’interno del nostro sistema, come analizzato nel capitolo precedente, è cruciale. Il suo sviluppo deve essere il più possibile indipendente dall’applicazione web e semplice da usare. Il processo di autenticazione/autorizzazione deve coincidere, inoltre, con la gestione effettuata dagli utenti dalla dashboard di visualizzazione dei grafici. Prima di analizzare i vari sistemi già messi a disposizione, cerchiamo di comprendere cosa intendiamo per autenticazione e autorizzazione:

- Nell’ambito dei sistemi informativi, l’autenticazione è il processo attraverso il quale un sistema informatico, un computer, un software o un utente verifica la corretta, o almeno presunta, identità di un altro computer, software o utente che vuole comunicare attraverso una connessione, autorizzandolo ad usufruire dei relativi servizi associati (LINFO, 2006). Possiamo quindi dire che corrisponde a un sistema che verifica se l’individuo è chi effettivamente dice di essere.
- Autorizzazione significa specificare i privilegi di accesso alle risorse legate alla sicurezza delle informazioni, alla sicurezza informatica e in particolare al controllo degli accessi. In maniera più formale, autorizzare significa conferire ad un utente il diritto ad accedere a risorse specifiche del sistema, sulla base della sua identità.

Come possiamo ben comprendere, l’autenticazione è qualcosa di completamente diverso rispetto l’autorizzazione: con il primo controlliamo solo l’identità dell’entità ma non consideriamo i suoi diritti da accesso; mentre per il secondo, come abbiamo sancito più volte, ci affidiamo al sistema di visualizzazione dei grafici per cui non ci poniamo il problema di capire come implementarla ma di come dare visibilità al sistema di visualizzazione grafici di reperire le informazioni mediante le quali gestire un controllo di accesso. Per quanto riguarda l’autenticazione, il primo problema da risolvere è cercare di comprendere che tipologia di autenticazione vogliamo dato che esistono innumerevoli sistemi di cui ne analizziamo tre tipologie.

Autenticazione tramite password

L'autenticazione basata su password è quella più comunemente usata nel caso di autenticazione di persone, dato che basta solo ricordare una "parola d'ordine" che verrà usata per la protezione dei sistemi informatici. Ovviamente, la scelta della password deve essere fatta nel modo più attento possibile dato che la sua perdita potrebbe comportare furti di dati sensibili. Ci devono essere, però, una serie di precauzioni che devono essere prese nella scelta della password come:

- La scelta di una password non banale e non riconducibile a una semplice parola (1234, abcd ecc.) oppure a un'informazione personale (anno di nascita, nome, cognome ecc.)
- Non riutilizzare la stessa password per più siti diversi perché la perdita potrebbe mettere a rischio molti dati
- Non avere password troppo corte poiché più è lunga una password e maggiore è il lavoro da affrontare per l'attaccante per riuscire ad indovinare la password

Una volta scelta la password basandosi su queste "Best Practice", l'utente sarà in grado di usarla per l'accesso al sistema. Sebbene, nella maggior parte dei casi, il sistema basato su password ha dei problemi intrinsechi legati alla capacità dell'utente stesso di memorizzare password complesse e lunghe, è il sistema più usato ed è quello a cui noi faremo riferimento. Allo scopo di garantire la memorizzazione di password complesse, sono stati studiati i Password Wallet ovvero dei portafogli digitali protetti in cui salvare le password per siti diversi.

Autenticazione tramite token

Un'altra metodologia di autenticazione è basata sull'ausilio dei token di sicurezza rilasciati da dispositivi fisici e utilizzati come password. I sistemi di questo tipo nascono per un motivo molto semplice: la ridondanza della password su siti diversi. Allo scopo, esistono diverse metodologie come: One Time Password (OTP), ovvero ogni volta che devo autenticarmi viene rilasciata una password; Time-based OTP, ovvero genero delle password a scadenza temporale; Out of Band OTP, ovvero uso username e password per autenticarmi ma uso un dispositivo esterno registrato, come il cellulare, per confermare l'identità. Sebbene la soluzione dei token sia interessante, da un punto di vista del non riutilizzo della password, è molto costosa dato che l'hardware per generare token è costoso e bisogna considerare che la maggior parte di questi sistemi richiede una sincronizzazione client/server (per esempio, con il Time Based OTP devo accertarmi che il token sia ancora valido per il server).

Autenticazione biometrica

Un'ultima metodologia di autenticazione è basata sull'uso delle informazioni biometriche dell'utente. Esempi di autenticazione si basano sull'uso dell'impronta digitale, degli occhi, del viso ecc. Tali sistemi, per quanto possano essere sofisticati, hanno delle problematiche legate all'affidabilità perché un utente potrebbe avere problemi fisici per cui si considera il False Reject Ratio, cioè il numero di persone che non autentico perché non riconosco; e il False Acceptance Ratio, cioè il numero di persone che autentico ma che non hanno requisiti; come parametri per la valutazione delle performance. Inoltre, un ulteriore limite di tali sistemi è legato al furto delle informazioni biometriche che potrebbe rendere impossibile autenticarsi successivamente.

Scelta del sistema di autenticazione

Sulla base delle considerazioni effettuate, quello che noi vogliamo è un sistema di autenticazione che sia il più semplice possibile per cui ha senso pensare a un sistema di autenticazione basato sull'uso della password. L'introduzione di sistemi biometrici oppure dell'uso di dispositivi per il rilascio di un token non fa altro che rendere il processo di autenticazione, per il nostro caso, molto più dispendioso e pesante da un punto di vista computazionale. Nella maggior parte dei casi, infatti, nelle applicazioni web che fanno riferimento a dati non troppo sensibili si preferisce usare la password perché è la soluzione meno complessa e costosa.

Sistema Stateless e Statefull

Una volta individuata la tipologia di autenticazione bisogna soffermarsi su soluzioni di tipo Statefull o Stateless. Nel primo caso, una volta che l'utente è autenticato viene mantenuta una sessione tra Client e Server per cui qualsiasi richiesta al Server è autenticata mentre, nel secondo caso, l'utente si autentica e viene rilasciata un'informazione aggiuntiva che dovrà essere inserita nelle richieste successive al fine di autenticarle. L'uso di una soluzione rispetto a un'altra deve essere ben motivata dato che risulta cruciale ai fini della progettazione del sistema stesso per cui andiamo ad analizzare vantaggi e svantaggi. Da un punto di vista delle informazioni che possono essere rubate, in una soluzione Statefull è quasi impossibile rubare dati di sessione dato che abbiamo solo un identificativo associato all'utente; in una soluzione Stateless, invece, posso rubare tutte le informazioni che voglio dato che sono contenute in oggetti aggiuntivi alla richiesta (a meno che tali informazioni non siano cifrate). Da un punto di vista di consumo delle risorse, una soluzione Statefull va a reperire tutte le informazioni accedendo allo storage del browser causando il consumo

addizionale di risorse; dall'altro canto soluzioni Stateless contengono tutte le informazioni in particolari oggetti. Da un punto di vista dell'implementazione e della scalabilità, le soluzioni Statefull non sono le migliori perché se andiamo ad usare un database per salvare le sessioni è ovvio che bisogna avere anche un database in cui rendere persistenti le sessioni e, durante l'aggiunta di nuove sessioni, bisogna considerare la scalabilità per la loro archiviazione. Su queste problematiche, la soluzione Stateless offre maggiori vantaggi perché per la sua implementazione mantengo tutto all'interno di oggetti e l'aggiunta di nuove sessioni non richiede dell'Effort aggiuntivo (Open Identity Platform). Riportiamo, infine, nella seguente tabella una versione riassuntiva dei vantaggi e svantaggi delle due soluzioni.

	Statefull	**Stateless**
Rubare informazioni di sessione	Non possibile, solo un identificativo.	Possibile, contiene le informazioni
Consumo di risorse	Maggiore, richiede accesso all'archivio.	Minore, nessun accesso all'archivio.
Implementazione	Complessa, richiede persistenza DB.	Semplice, dati nel token.
Scalabilità	Complicata, richiede scale all'archivio.	Semplice, non richiede cambiamenti.
Sicurezza	Più sicuro, solo sistema autenticazione.	Meno sicuro, compromissione del token.
Dimensione token	Piccolo, solo identificativo.	Può crescere, dati influenzano dimensione.
Limitazione accesso	Possibile, parti accedono ai dati necessari.	Tutte le parti accedono a tutto.
Revoca sessione	Possibile.	Impossibile, token ha data di scadenza.
Modifica dati sessione	Possibile, modifiche nell'archivio.	Impossibile, token non modificabile.
Implementazione SSO	Facilitata, integrazione tramite gateway.	Complicata, richiede cambiamenti.

Tabella 1. Vantaggi e Svantaggi delle soluzioni Stateless e Statefull

Sulla base di ciò abbiamo optato per una soluzione di tipo Stateless a favore della semplicità di implementazione e della scalabilità della soluzione nonostante la soluzione Statefull possa garantire più sicurezza.

JSON Web Token

Tra le soluzioni Stateless messe a disposizione nel tempo, la più utilizzata è basata sull'uso dei JSON Web Token (JWT). Precedentemente, abbiamo fatto riferimento all'uso di "oggetti" aggiuntivi per indicare un informazione da aggiungere alle richieste che altro non è che un token inserito all'interno tramite un Header della richiesta. All'interno del token vengono inserite tutte le informazioni necessarie al server per verificare la sua identità. In genere, i JWT vengono firmati usando un segreto (con un algoritmo tipo HMAC) oppure utilizzando una coppia chiave pubblica/privata con l'algoritmo RSA. Sebbene i token possano essere cifrati, per cui le informazioni non sono direttamente visibili, concentriamoci sui token firmati. Grazie alla firma, possiamo verificare l'integrità di quello che viene scritto mentre, nel caso della cifratura, stiamo nascondendo le informazioni (nel nostro caso, introdurremmo solo maggiore computazione dato che le informazioni che vogliamo rendere disponibili non sono così sensibili come ruolo, nome, cognome ecc.). Ci sono diversi scenari in cui ha senso usare i token JWT:

- Autorizzazione: rappresenta lo scenario più comune in cui usare il JWT. Una volta che l'utente effettua il login, ogni richiesta conterrà il JWT permettendo all'utente di accedere solo ai path, servizi e risorse a cui gli è permesso. Inoltre, in questo caso, viene fatto un uso molto estensivo in sistemi di tipo Single Sign-On (SSO), ovvero sistemi in cui l'autenticazione viene rediretta a un singolo provider che provvede a fornire un token che sarà utilizzato su più siti diversi in modo da permettere all'utente di autenticarsi una volta e poi rimanere autenticato per siti diversi. La motivazione è proprio legata al piccolo overhead introdotto e al fatto che il token è condivisibile su più domini.

- Scambio di informazioni: i JWT si presentano come un efficace mezzo per trasmettere informazioni tra diverse entità con sicurezza. Il loro utilizzo permette di garantire l'autenticità dei mittenti mediante la firma, che può avvenire attraverso chiavi pubbliche e private assicurando che l'identità sia autentica. Oltre a ciò, la firma si basa sull'intestazione e il payload, consentendo la verifica dell'integrità dei contenuti.

La struttura di un JWT consiste di tre parti che vengono separate mediante un punto (.), che sono:

- Header, in cui tipicamente viene inserito il tipo di token (nel nostro caso JWT) e l'algoritmo di firma ;
- Payload, in cui vengono dichiarati i claim che rappresentano le dichiarazioni fatte su un entità. In questo caso, esistono diversi tipi di claim che noi suddividiamo in:
    - Registrati, che sono quelli che in genere vengono previsti nel token JWT o che sono generalmente raccomandati (ne riportiamo i più importanti nella tabella alla fine del paragrafo);
    - Pubblici, che vengono definiti a piacimento da coloro che utilizzano i JWT. Tuttavia, al fine di evitare collisioni, potrebbero essere definiti nel registro di IANA destinato ai JWT o comunque definiti in un URI che contiene uno spazio dei nomi resistente alle collisioni;
    - Privati, che rappresentano dei valori custom definiti tra le parti e che vengono personalizzate al fine di condividere informazioni;
- Signature, in cui viene calcolato un valore basato sull'Header, sul payload e un segreto usando l'algoritmo specificato nell'Header. Il vantaggio della firma è che posso verificare sia l'integrità del messaggio e, nel caso di firma con chiave privata, posso verificare anche l'identità di chi ha rilasciato il token.

Un esempio di token JWT è il seguente:

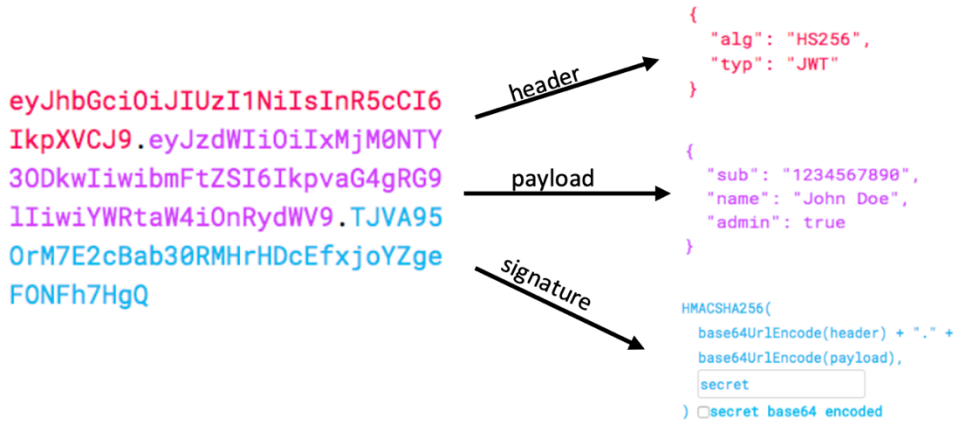

Figura 3. Esempio di Token JWT

Come possiamo vedere, la parte di Header indica come tipo di token "JWT" e utilizza un algoritmo HS256 per la firma. Nel payload sono contenute alcune intestazioni standard (sub) e altre come name e admin. La firma è stata valutata tramite

l'applicazione dell'algoritmo HMACSHA256, come indicato nell'Header, basandosi su una codifica a base64 dell'insieme: Header, Payload e un segreto. L'uso del JWT, come detto più volte, è basato sull'inserimento all'interno di un Header nella richiesta inviata al Resource Server che ci garantirà accesso alla risorsa. In particolare, al momento, supponiamo di conoscere l'Authorization Server che ci rilascerà il token dopo il login effettuato e che, allo stesso tempo, utilizzeremo per verificare il token. Il processo di autenticazione mediante JWT funziona nel seguente modo:

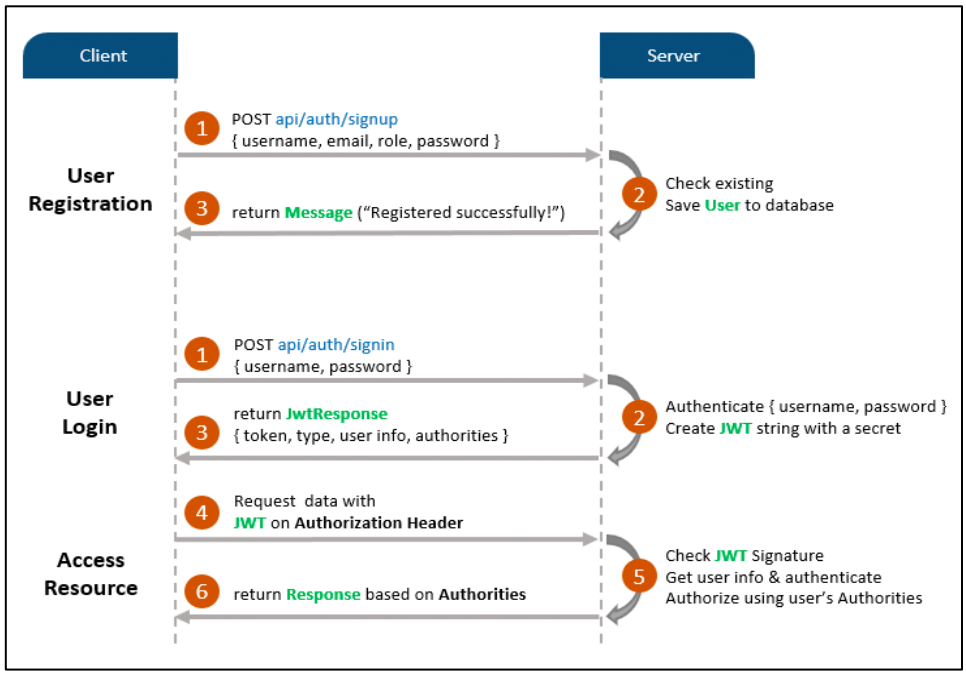

Figura 4. Uso del token JWT

Immaginandoci di aver provveduto già alla registrazione dell'utente, quando viene fatto il login viene rilasciato un token. In qualsiasi richiesta inviata al server, deve essere inserito un Header "Authorization" affidandomi a un Server che verifica se il token sia valido o meno (in genere, chi verifica se il token sia valido o meno è lo stesso server che lo rilascia). Per cui quando invio una richiesta al server che contiene la risorsa a cui sono interessato, questo si occupa di contattare l'Authorization server che si occuperà di fornire una risposta sull'autenticità del token.

Fino ad ora, abbiamo dato per scontato che l'uso del JWT sia migliore rispetto ad altre tecniche di tipo Stateless senza darne una vera e propria motivazione. Per questo motivo, cerchiamo di analizzare i vantaggi dell'uso del JWT confrontandolo con un token di tipo Security Assertion Markup Language (SAML). I token di tipo SAML utilizzano la codifica XML che è più pesante da leggere rispetto alla codifica JSON utilizzata dai JWT. Dal punto di vista della sicurezza, entrambi possono essere firmati utilizzando una coppia chiave pubblica/privata ma da un punto di vista del processo di

firma è possibile introdurre in maniera più semplice falle di sicurezza rispetto alla semplicità della firma del JWT. Inoltre, la maggior parte dei linguaggi di programmazione hanno a disposizione dei parser JSON cosa diversa per i documenti XML che è difficile mappare su degli oggetti. Per questo motivo è molto più semplice lavorare con i JWT. Per quanto riguarda l'utilizzo, il JWT viene utilizzato su grande scala, evidenziandone la facilità della loro elaborazione lato client su più piattaforme, in particolare sui dispositivi mobili. (Auth0)

Sub	Identifica il soggetto del JWT
Iss	Definisce chi rilascia il JWT
Aud	Elenca a chi è destinato il JWT
Exp	Contiene la scadenza del token
Nbf	Definisce il tempo prima del quale l'accettazione del token non è valida
Iad	Definisce il tempo in cui è stato rilasciato il token

Tabella 2. Claim standard di un JWT

O-Auth Server

Fino ad ora, abbiamo analizzato il funzionamento dei processi di autenticazione e cercato di comprendere quale tipologia di autenticazione potrebbe rispondere alle nostre esigenze, individuando come soluzione un sistema basato su password di tipo Stateless in cui si utilizzano dei JWT. Per quanto possa essere semplice ed efficace l'uso dei token per un'applicazione Web, non ci garantisce l'indipendenza e l'astrazione di cui abbiamo bisogno ed è per questo motivo che vogliamo racchiudere tutta la logica legata all'autenticazione a un authentication server a cui noi faremo le richieste senza passare direttamente per l'Identity Provider, che è colui che si occupa direttamente di provvedere al rilascio dei token JWT che poi vengono ottenuti grazie un endpoint di login esposto dall'authentication server. Per comprendere meglio ciò di cui stiamo parlando, consideriamo la seguente figura:

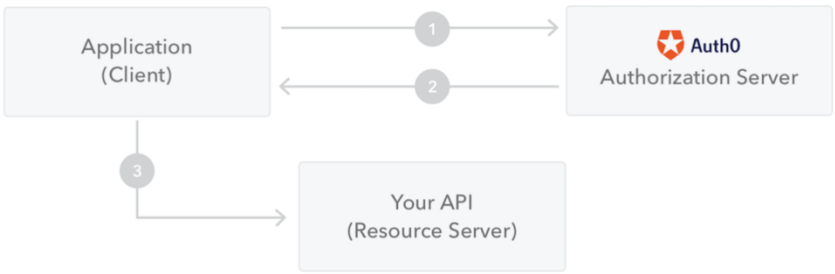

Figura 5. Ruolo dell'Authorization Server

Immaginiamoci che l'entità "Application" sia l'applicazione che abbiamo sviluppato per visualizzare i grafici previa autenticazione. Per fare il login, si rivolge a un authentication server che fornisce un token JWT, successivamente mediante le informazioni ottenute viene fatta una richiesta verso il Resource server (immaginiamoci il Resource Server come la nostra piattaforma di visualizzazione grafici) che controllerà, mediante il ruolo definito all'interno del JWT, se abbiamo i permessi per visualizzare. Qualsiasi richiesta che arriva al Resource server ovviamente verrà controllata per cui chiameremo delle API pubbliche dell'Authorization Server per validare il JWT. In questa figura, non abbiamo contezza della struttura completa della nostra architettura ma ci da un'idea del ruolo svolto Authentication Server (nella figura non è presente l'entità dell'Identity Provider e per il momento supponiamo che l'Authorization Server dia direttamente i token JWT). Questa idea si basa sul protocollo Open Authorization (O-Auth) che poggia le basi sulla delega di accesso, garantendo agli utenti di concedere ai fornitori di servizi (es. siti web oppure applicazioni) l'accesso alle proprie informazioni senza fornire loro la password (ENTRUST). Il protocollo può riassumersi in cinque passi:

1. Richiesta di accesso
2. Richiesta di scambio di token/segreto
3. Autenticazione
4. Accedere allo scambio di token
5. Accesso garantito

Nel tempo, ci sono state diverse implementazioni di O-Auth Server a cui ci si è ispirati per l'implementazione di un proprio server di autorizzazione. In particolare, cercheremo di descrivere una soluzione esistente come Oauth2-proxy e una soluzione ex-novo.

OAuth2-Proxy vs OAuth ex-novo server

Figura 6. Logo OAuth2 Proxy

OAuth2-proxy è uno strumento progettato per semplificare il processo di autenticazione e autorizzazione all'interno delle applicazioni web attraverso

l'implementazione del protocollo OAuth. Questo protocollo è ampiamente utilizzato per consentire l'accesso sicuro alle risorse dell'applicazione da parte degli utenti, senza dover condividere le loro credenziali direttamente con l'applicazione stessa. All'accesso a un'applicazione web protetta da OAuth2-proxy, esso agisce come intermediario tra l'utente e l'applicazione. Il processo di autenticazione avviene attraverso i seguenti passaggi:

1. Reindirizzamento: Quando un utente tenta di accedere all'applicazione, OAuth2-proxy lo reindirizza a un fornitore di identità esterno (Identity Provider), come Google o GitHub.
2. Autenticazione: L'utente viene quindi autenticato dall'Identity Provider tramite il suo account esistente presso il provider.
3. Generazione del Token: Una volta autenticato con successo, il provider di identità genera un token di accesso che rappresenta l'autorizzazione dell'utente all'applicazione.
4. Verifica del Token: Il token di accesso viene inviato a OAuth2-proxy, che lo verifica con il provider di identità per garantire la sua validità.
5. Autorizzazione: Se il token è valido, OAuth2-proxy concede l'accesso all'utente all'interno dell'applicazione web.

OAuth2-proxy è spesso adottato quando si desidera fornire un accesso sicuro a un'applicazione web utilizzando l'autenticazione OAuth. È particolarmente utile quando si vuole integrare l'autenticazione attraverso servizi di terze parti come Google, GitHub o altri fornitori di identità. Inoltre, rappresenta una soluzione pratica per semplificare l'autenticazione e l'autorizzazione all'interno delle applicazioni web attraverso l'uso del protocollo OAuth. Con il suo approccio di reverse proxy, fornisce una struttura solida e sicura per gestire l'accesso degli utenti, liberando gli sviluppatori dall'onere di scrivere codice complesso per l'autenticazione.
Cerchiamo ora di mettere a confronto tale soluzione con una soluzione ex-novo, analizzandone vantaggi e svantaggi al suo utilizzo:

- Personalizzazione
  È ovvio che da un punto di vista della personalizzazione la soluzione ex-novo è nettamente in vantaggio dato che consente di progettare l'autenticazione in base alle specifiche esigenze dell'applicazione, consentendo quindi un alto grado di personalizzazione. Dall'altro canto, Oauth2-proxy limita la personalizzazione dato che si tratta di una soluzione già esistente e potrebbero trovarsi delle difficoltà di adattamento all'applicazione web.

- Sicurezza
  Le soluzioni già presenti sul web sono soluzioni testate per cui generalmente si tratta di soluzioni che garantiscono un certo grado di sicurezza, tra queste troviamo Oauth2-proxy. Invece, soluzioni ex-novo potrebbe far emergere vulnerabilità in ambito di sicurezza che devono essere gestite in maniera adeguata, a meno che non si usino librerie esterne per la progettazione del server.

- Implementazione
  Una soluzione già presente sul web come Oauth2-proxy è sicuramente migliore da un punto di vista dell'implementazione dato che riduce di molto il tempo di sviluppo. Una soluzione ex-novo ha delle difficoltà legate alla curva di apprendimento richiesta sia per quanto riguarda i concetti di OAuth e sia per quanto riguarda le librerie (esempio PassportJS).

In sintesi, la scelta tra l'implementazione ex-novo e l'utilizzo di OAuth2-proxy dipende dalle esigenze dell'applicazione, dalle risorse disponibili e dal livello di controllo e personalizzazione desiderato. Mentre una soluzione ex-novo offre un controllo completo e personalizzazione, richiede più tempo e competenze, mentre OAuth2-proxy offre una rapida implementazione e sicurezza testata, ma con alcune limitazioni di personalizzazione.

Keycloak

L'analisi effettuata fino ad ora ha messo in evidenzia un problema ovvero la necessità di definire la responsabilità sul rilascio del JWT. Per questo motivo occorre individuare un Identity Provider (IdP) che si occupi di rilasciare/validare dei token JWT mediante i quali definire gli utenti e la scelta è ricaduta su Keycloak. Questo altro non è che un IdP in formato open source, il cui codice è accessibile dall'esterno e la motivazione della scelta è proprio legata alla possibilità di non affidarsi a IdP esterni ma a dei provider che possano essere gestiti internamente al dominio senza affidarsi esternamente come succede nel caso di Google, Facebook ecc. Sotto questo punto di vista, abbiamo delle garanzie relative alla gestione dei dati e la possibilità di personalizzare nel miglior modo possibile il processo di autenticazione. In particolare, Keycloak introduce diversi vantaggi come:

- Single Sign-On: la possibilità da parte degli utenti di autenticarsi mediante Keycloak al posto di affidarsi alle singole applicazioni, ciò significa che non devo gestire direttamente i moduli di accesso, l'autenticazione degli utenti e la loro archiviazione nelle applicazioni (es. salvare gli utenti mediante un database). La

cosa interessante è che, una volta autenticato, l'utente potrebbe evitare il login mediante su un ulteriore applicazione web. Inoltre, il principio vale anche per il logout per cui un utente può effettuarlo una sola volta per essere disconnessi da tutte le applicazioni che usano Keycloak.

- Identity Brokering e Social Login

  Possiamo configurare il provider di autenticazione mediante l'admin console definita all'interno di Keycloak. Potremmo decidere di affidarci a un provider esterno come Google, GitHub, Facebook ecc. oppure basare l'autenticazione sui protocolli di OAuth (nel nostro caso faremo riferimento all'implementazione mediante OpenID Connect).

- Account Management Console
  Keycloak offre una console di amministrazione che ci dà la possibilità di gestire in maniera centrale tutte le funzionalità ad essa associate come: la possibilità di definire criteri di organizzazione granulari, gestire gli utenti, configurare la federazione degli utenti ecc.

- Standard Protocols
  Keycloak è basato sull'uso e l'implementazione di protocolli standard come OAuth 2.0, SAML oppure OpenID Connect.

- Authorization Services
  Se abbiamo necessità di andare oltre l'autorizzazione tramite ruoli, è possibile definire diversi livelli di autorizzazione molto più granulari. Questo ci da possibilità di gestire le autorizzazioni per tutti i nostri servizi dalla console di amministrazione, definendo le politiche di cui noi abbiamo bisogno.

Ciò che noi faremo sarà quindi affidarci a Keycloak per il processo di autenticazione/autorizzazione (per quanto riguarda il rilascio e la validazione dei JWT). Per capire meglio ciò che sta succedendo, allarghiamo lo schema aggiungendo il nostro IdP:

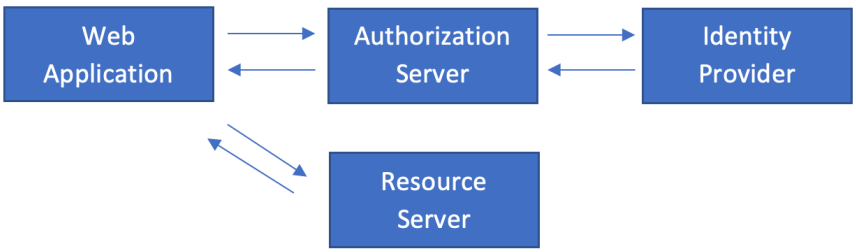

Figura 7. Schema di rilascio JWT con IdP

Per cui il JWT non viene rilasciato dall'Authorization server ma da Keycloak e il ruolo del server è quello di schermare l'IdP fornendo una serie di API pubbliche all'applicazione web per effettuare il login e validare il JWT.
La particolarità di Keycloak è che, offrendo una console di amministrazione, è possibile anche fare configurazioni particolari come, per esempio, legate ai requisiti del processo di registrazione per cui validazione dell'e-mail di dominio e invio dell'e-mail di verifica. Ciò andrà a nostro vantaggio perché ci faciliterà di molto lo sviluppo e l'implementazione del processo di registrazione. Un ulteriore vantaggio è anche la personalizzazione delle schermate di login e di registrazione nonché la possibilità di introdurre delle estensioni java, per esempio, per la validazione delle e-mail.

Visualizzazione dei grafici

Fino ad ora, l'analisi effettuata è stata principalmente legata al processo di autenticazione ma un altro problema da porci è come visualizzare i grafici all'interno della nostra applicazione web. Esistono innumerevoli sistemi di raccolta dati che sono stati sviluppati nel tempo e che danno la possibilità di effettuare un controllo sull'accesso alle risorse e, tra questi, il più famoso è sicuramente Grafana.

Grafana

Grafana è un software open-source per la raccolta e visualizzazione grafica dei dati che, nel nostro caso, impieghiamo per il rendering di grafici basati sui dati raccolti da parte di sensori di temperatura. Tutto il backend dietro alla raccolta dati è già stato sviluppato in un precedente lavoro per cui questa Tesi si baserà sull'espandere le funzionalità dell'architettura già esistente. In particolare, ci sono diverse funzionalità che offre la piattaforma:

- La grande varietà di tipologie di grafici messe a disposizione: Grafana permette di visualizzare i dati in forma tabellare, sotto forma di diagrammi e così via;
- Dashboard interattive: gli utenti possono creare delle dashboard personalizzate con widget interattivi;
- Supporto per le query: Grafana consente di effettuare delle query sui dati raccolti che garantiscono la possibilità, per esempio, di visualizzare i dati di un sensore rispetto a un altro;
- Gestione degli utenti: la piattaforma garantisce la possibilità di gestire gli utenti e i ruoli ad essi associati, per cui anche l'accesso alle risorse mediante ruolo dell'utente;

- Connessione a diverse sorgenti dei dati: potrei pensare di voler connettere Grafana a diversi database usati per la raccolta dati;

Da un punto di vista di questa Tesi, è interessante il lavoro svolto per quanto riguarda le query e l'accesso alle risorse mediante ruolo. Immaginiamo di avere dati provenienti da diverse fonti: database, servizi cloud, applicazioni. Con Grafana, possiamo scrivere query personalizzate per estrarre esattamente le informazioni che ci interessano. Queste query ci permettono di filtrare, aggregare e analizzare dati complessi in modi significativi. Ad esempio, possiamo monitorare le prestazioni delle nostre applicazioni, analizzare il traffico del sito web o valutare il consumo di risorse del server. Tutto ciò è possibile grazie alle potenti capacità di query messe a disposizione, che ci consentono di creare visualizzazioni informative e precise. Inoltre, Grafana offre anche un sistema sofisticato di gestione degli accessi basato su ruoli. Immaginiamo di avere una squadra con diversi membri, ognuno con esigenze di visualizzazione dei dati diverse. Utilizzando i ruoli, possiamo assegnare privilegi specifici a ciascun utente o gruppo di utenti. Ad esempio, gli amministratori potrebbero avere accesso completo a tutte le risorse, mentre i membri del team di sviluppo potrebbero avere accesso solo alle visualizzazioni correlate al loro ambito. Questo livello di controllo garantisce che le informazioni vengano condivise solo con coloro che ne hanno bisogno, migliorando la sicurezza e la gestione delle risorse. Un'altra cosa interessante di Grafana, che lo rende famoso, è la possibilità di creare embedding di grafici presenti all'interno della dashboard incapsulandoli in oggetti chiamati iframe.

Realizzazione degli obiettivi

Nel seguente capitolo, viene analizzato l'insieme dei metodi e delle tecnologie utilizzate con lo scopo di raggiungere gli obiettivi predisposti dalla tesi andando ad analizzare le problematiche riscontrate durante l'implementazione della soluzione e a come sono state risolte, analizzandone le motivazioni.

Architettura complessiva

L'architettura che si è andata a sviluppare, allo scopo di realizzare gli obiettivi prefissati nel capitolo 2, nel suo complesso è la seguente:

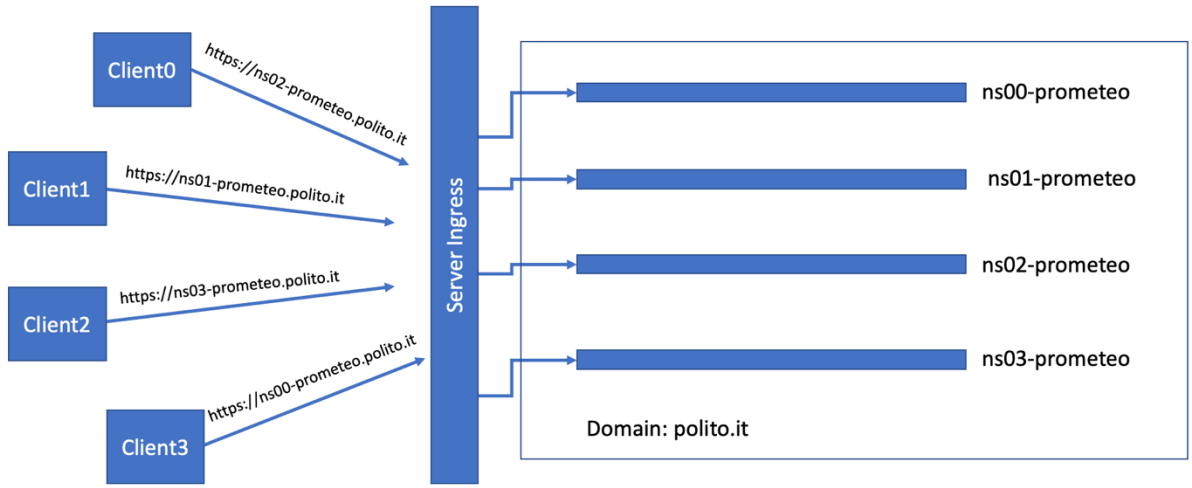

Figura 8. Design di alto livello dell'architettura complessiva

In particolare, da un punto di vista esterno si tratta di garantire a più sottoreti di essere raggiungibili mediante il Server Ingress, andando a corrispondere sottodomini dividendo il dominio base (polito.it). Qualsiasi utente esterno deve essere in grado di raggiungere i sottodomini mediante corrispettivo URL affiancato da HTTPS. La motivazione di un'architettura di questo tipo è legata prettamente alla necessità di aggiungere all'occorrenza nuove sottoreti in maniera semplice e flessibile che garantisca l'indipendenza di ognuna dalle altre in modo che la gestione degli utenti, così dei dati ad essi allegati, sia diversa. Una soluzione alternativa sarebbe quella di portare al di fuori della sottorete l'IdP (allo stesso livello del Server Ingress) e garantire

la sua raggiungibilità da tutte le sottoreti. Sebbene sia comunque una soluzione semplice, non risponde ai nostri requisiti di indipendenza dato che la richiesta è quella di garantire una gestione autonoma per ogni sottodominio degli utenti, andando così a diversificare i requisiti di registrazione (es. magari in un sotto dominio non ho la necessità di andare a verificare l'e-mail). Cerchiamo di comprendere ora come realizzare l'architettura della nostra sottorete partendo dagli obiettivi che ci siamo prefissati precedentemente. L'idea, come già ripetuto precedentemente, è quella di avere diverse entità che andranno a formare una sottorete ma non devono essere accessibili direttamente dall'esterno ma tramite path diversi sotto lo stesso dominio. Allo scopo è stato quindi introdotto un reverse proxy, implementato tramite NGINX, che si occuperà di redirigere non solo le richieste provenienti dagli utenti ma anche le richieste tra le entità. La differenza, in questo caso, è che le richieste degli utenti saranno fatte in HTTPS ma tra le entità sarà in HTTP. Come già detto precedentemente, l'obiettivo principale è di introdurre un sistema di autenticazione che si occupi in maniera completa di tutto quel riguarda i dati degli utenti, il rilascio/validazione dei token JWT, il login e la registrazione. A tale scopo, necessitiamo di un Identity Provider che si occupi di questo per cui utilizzeremo Keycloak. Il problema è che noi vogliamo una soluzione abbastanza sicura e flessibile per cui introduciamo ulteriormente un Authorization Server (Auth Server) scritto in NodeJS con cui la nostra applicazione comunicherà. Utilizzare un Auth Server come intermediario tra un client e un Identity Provider (IDP) è preferibile per diverse ragioni. Innanzitutto, protegge le credenziali sensibili evitando l'esposizione diretta nel client. Inoltre, permette una gestione centralizzata delle autorizzazioni, consentendo un controllo più dettagliato sull'accesso alle risorse rispetto all'IDP. L'Auth Server implementa protocolli di sicurezza come OAuth per garantire un'autenticazione sicura e un accesso autorizzato. Ciò riduce anche la complessità del client, eliminando la necessità di gestire logiche di autenticazione e autorizzazione complesse. Inoltre, l'Auth Server può supportare diverse fonti di identità, consentendo un'ampia flessibilità nell'integrazione con vari IDP. Centralizza anche logging e auditing, agevolando il monitoraggio delle attività. Infine, agisce come uno strato aggiuntivo di protezione in caso di vulnerabilità o problemi nell'IDP, contribuendo a mitigare i rischi di sicurezza complessivi. La soluzione è quella di introdurre un server NodeJS che fungerà da schermo all'IdP e che processerà tutte le richieste occupandosi direttamente di usufruire delle API dell'Identity Provider. La configurazione del sistema verrà quindi fatta per la maggior parte sul server NodeJS lasciandoci però alcune cose come la definizione dei ruoli su Keycloak.

Un'altra questione aperta è la raccolta dei dati e la loro visualizzazione, in particolare dobbiamo pensare a un sistema di raccolta e di rendering dei dati. La soluzione che proponiamo è di affidarci a Grafana che, come già analizzato precedentemente, introduce una serie di funzionalità come: il rendering dei dati in una grande varietà dei grafici, la possibilità di fare query personalizzate ecc. che fanno tutte al caso nostro. Per quanto possa essere semplice lavorare con i dati all'interno del database predisposto da tale piattaforma, nel nostro caso diventa complesso perché stiamo raccogliendo dati da sensori diversi per cui dobbiamo necessariamente introdurre una sorgente diversa che è un Database esterno. Lo sviluppo del backend di raccolta dati è stato proposto in un ulteriore lavoro di Tesi per cui noi simuleremo solamente la connessione con un'altra sorgente dati introducendo PostgreSQL.
Concludiamo l'analisi considerando la necessità di sviluppare un'Applicazione Web che consenta di visualizzare i grafici internamente previa autenticazione. In particolare, tale applicazione è stata sviluppata in un altro lavoro di Tesi ed è composta dalla tripletta: Applicazione ReactJS, Server NodeJS e MySQL. Inizialmente, tale sistema di visualizzazione consentiva solamente il render dei grafici in maniera anonima (senza autenticazione) e, successivamente, sono state messe a disposizione delle API esposte dal nostro OAuth Server che hanno garantito la possibilità di login, registrazione, autenticazione ecc. andando anche a introdurre la possibilità di compilare questionari o vedere il proprio profilo solo previa autenticazione.
In base a ciò, il design ad alto livello della singola rete è il seguente:

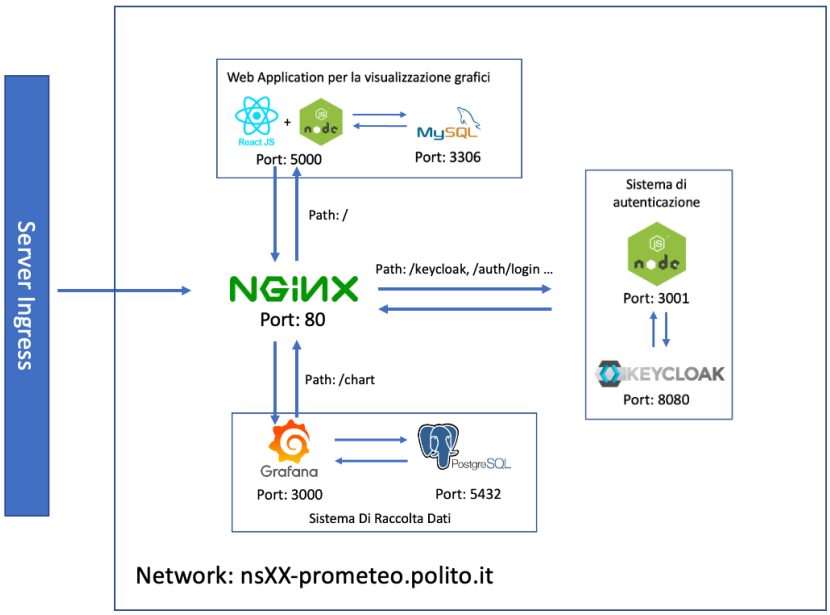

Figura 9. Design ad alto livello della sottorete

Come possiamo vedere dalla figura, ogni componente espone una porta che non è direttamente accessibile dall'esterno ma solo attraverso NGINX. In particolare, Keycloak, oltre ad esporre la serie di API di cui usufruirà il server NodeJS, mette a disposizione anche una dashboard attraverso la quale è possibile definire una serie di Realm, a cui il server farà riferimento, e una serie di configurazioni. Tra quelle più interessanti vi è sicuramente la gestione del processo di registrazione, in particolare a quale server mail fare riferimento per l'invio dell'e-mail di conferma e alla validazione del dominio dell'e-mail inserita. All'interno della figura, è possibile anche vedere l'iterazione tra NGINX e la dashboard sviluppata e, nella maggioranza dei casi, si tratta di una comunicazione verso il server NodeJS sviluppato con lo scopo di raggiungere la pagina di autenticazione.

Processo di Autenticazione

La prima fase del progetto è stata quella di sviluppare il processo di autenticazione mediante l'introduzione di un server e di un IdP interni al sotto dominio, in tal modo si garantisce indipendenza nella gestione degli utenti e nell'implementazione dei relativi requisiti. Nei paragrafi successivi, verrà descritto come si è deciso di implementare l'autenticazione e quali sono state le tecnologie che sono state utilizzate.

Backend del processo di autenticazione

Lo sviluppo del processo di autenticazione è stato effettuato mediante l'ausilio della libreria OpenID di PassportJS che permette di semplificare lo sviluppo di un applicazione web in JS/TS in ambito di autenticazione e autorizzazione mettendo a disposizione una serie di API che danno la possibilità di comunicare direttamente con l'IdP. L'obiettivo è implementare il seguente processo di autenticazione:

1. Il client richiede l'autenticazione, per esempio premendo un bottone sull'applicazione frontend, e viene rindirizzato verso la pagina di login di Keycloak in cui inserisce le sue credenziali;
2. Viene reindirizzato alla pagina precedente all'autenticazione

Questo funzionamento avviene ad alto livello ma cerchiamo di addentrarci più nel dettaglio di come è stato implementato e quali sono state le scelte tecnologiche adottate. Innanzitutto, diamo un occhiata al Docker Compose file:

```
keycloack:
    image: quay.io/keycloak/keycloak:latest
    environment:
      KEYCLOAK_ADMIN: ${KEYCLOAK_ADMIN}
      KEYCLOAK_ADMIN_PASSWORD: ${KEYCLOAK_ADMIN_PASSWORD}
      KC_FEATURES: declarative-user-profile
      KC_PROXY: edge
    volumes:
      - ./keycloak/realm/.:/opt/keycloak/data/import
      - ./keycloak/plugins/.:/opt/keycloak/providers
    command:
      - start-dev
      - --import-realm
      - --hostname-url=https://${DOMAIN}/
    restart: unless-stopped
    healthcheck:
      test: "exit 0"
server:
    image: server
    build:
      context: ./server
    environment:
      - keycloackClientId=${KEYCLOAK_CLIENT_ID}
      - keycloackSecretId=${KEYCLOAK_SECRET_ID}
      - adminUser=${KEYCLOAK_ADMIN_USER}
      - adminPassword=${KEYCLOAK_ADMIN_USER_PASSWORD}
      - domain=${DOMAIN}
      - realmName=${KEYCLOAK_REALM}
      - redirectPath=/auth/login/callback
      - logoutPath=/
    volumes:
      - "./server:/api/"
      - "/api/node_modules"
    depends_on:
      keycloack:
        condition: service_healthy
    restart: unless-stopped
```

Figura 10. Docker Compose File AuthServer e Keycloak

Notiamo che esiste una dipendenza tra il server e Keycloak, ciò è dovuto principalmente alla necessità che quando verrà contattato il server per fare l'autenticazione ho necessità di avere anche Keycloak disponibile dato che senza di lui non sarà possibile rilasciare alcuna tipologia di token oppure di verificarlo. Partendo dalla configurazione di Keycloak, possiamo notare le seguenti variabili d'ambiente: KEYCLOAK_ADMIN e KEYCLOAK_ADMIN_PASSWORD indicano le credenziali che useremo per accedere alla admin dashboard; KC_FEATURES fornisce la possibilità di avere funzionalità aggiuntive (per esempio, la validazione del dominio) e KC_PROXY viene utilizzato per permettere a Keycloak di restare dietro a un proxy. Oltre all'opzione di "Edge", esistono ulteriori opzioni valide come "Passthrough" oppure

“rsencypt” ma il nostro interesse è semplice utilizzare il nostro proxy server come terminatore TLS per cui, seguendo la documentazione di Keycloak, la nostra scelta è corretta. Per fini di sviluppo, è stato creato un realm con utenti già predefiniti che viene importato con il comando “—import realm” e dato che mettiamo Keycloak dietro a un reverse proxy abbiamo necessità di codificare tutte le API di frontend messe a disposizione da Keycloak dietro a un path definito dal dominio in cui mi trovo in modo da non avere problemi per l’iterazione HTTP/HTTPS e avere sulla pagina di login/registrazione API che usano HTTPS per cui è stato aggiunto “—hostname-URL” per dire a Keycloak di indicare un path base in HTTPS per tutte le sue API. Analizziamo ora il server, esistono una serie di variabili d’ambiente che descriviamo nel seguito: “keycloackClientId” e “keycloackSecretId” sono le credenziali di accesso per usufruire delle API di Keycloak; “adminUser” e “adminPassword” sono le credenziali di un ADMIN che abbiamo predisposto e il suo uso verrà spiegato nel seguito; “domain” e “realmName” sono il dominio e il nome del realm su cui abbiamo gli utenti; redirectPath e logoutPath sono i path di redirect e di logout. Ai fini della disponibilità di Keycloak, si impone nel server una dipendenza con “depends on” in modo che venga fatto partire il server solo quando Keycloak è effettivamente disponibile (per noi è disponibile quando Keycloak sarà in grado di chiamare il comando “exit 0” nel suo terminale). La condizione di health check su Keycloak mi assicura che avrà finito tutti gli import e le configurazioni così che il server potrà contattarlo. In dettaglio, il processo di autenticazione sarà il seguente:

1. L’utente, per contattare Keycloak, chiamerà un API /auth/login del server che lo rimanda verso la pagina di autenticazione che poi dovrà compilare ;

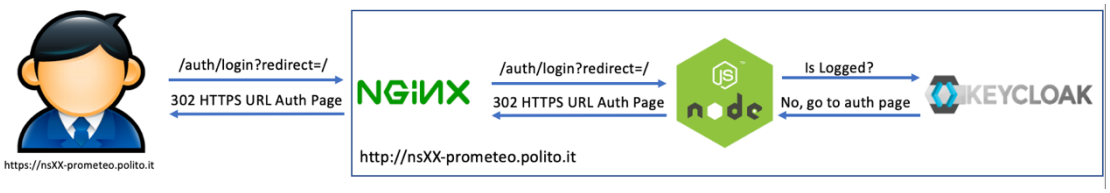

Figura 11. Flow di autenticazione parte 1

2. Dopo aver compilato il form e premuto il bottone di login, verrà rindirizzato verso la callback /auth/login/callback che avrà il compito di: rilasciare dei cookie e fare il redirect alla pagina precedente all’autenticazione;

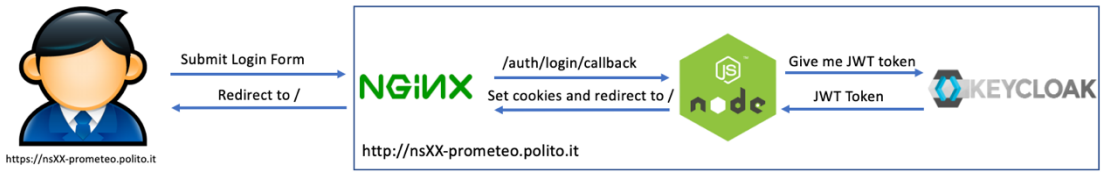

Figura 12. Flow di autenticazione parte 2

Quando l'utente vorrà scollegarsi, verrà chiamata l'API /auth/logout che revocherà il token in modo che non sia più usato. Andiamo ora ad analizzare il codice necessario per fare tutto questo. Innanzitutto, vengono fatte varie configurazioni come possiamo notare dal seguente codice:

```
function sleep(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}

var keycloakIssuer = null
while (keycloakIssuer == null) {
    try {
      keycloakIssuer = await Issuer.discover(`http://
                  keycloack:8080/realms/${realmName}`)
    } catch (error) {
        console.log("Waiting for Keycloak...")
        await sleep(10 * 1000)
    }
}

const client = new keycloakIssuer.Client({
    client_id: clientId,
    client_secret: clientSecret,
    redirect_uris: [redirectUri],
    post_logout_redirect_uris: [logoutUri],
});
```

Figura 13. Listato di codice 1

Al fine di rendere più robusta la soluzione ai comandi di tipo "Docker compose", si è deciso lo stesso di inserire una funzione di sleep che ha dato la possibilità al server di attendere l'avvio di Keycloak mediante l'URL per la configurazione del nostro issuer. In particolare, comandi del tipo "Docker compose up --build" aspettano l'avvio di Keycloak dato che lo stesso container dovrà fare tutte le operazioni di import e di build delle estensioni per poi abilitare il suo terminale ed eseguire "exit 0" ma nel caso di "Docker compose restart", per esempio, Keycloak è già avviato e non rifarà le stesse operazioni per cui il suo terminale è già attivo ma allo stesso non disponibile per il restarting per cui si aspettano 10 secondi e parte il server. In particolare, lasciamo un estratto del file JSON ricavato dal server:

```
{"issuer":"https://ns01-prometeo.polito.it/realms/AuthRealm",
"authorization_endpoint":"https://ns01-
prometeo.polito.it/realms/AuthRealm/protocol/openid-connect/auth",
"token_endpoint":"https://ns01-
prometeo.polito.it/realms/AuthRealm/protocol/openid-connect/token",
"introspection_endpoint":"https://ns01-
prometeo.polito.it/realms/AuthRealm/protocol/openid-
connect/token/introspect, ...}
```

Figura 14. URL di configurazione client per ns01-prometeo.polito.it

Come possiamo notare, stiamo usando HTTP per contattare direttamente Keycloak sull'URL "http://keycloack/realms/AuthRealm/.well-known/openid-configuration" per ottenere tutti gli endpoint che useremo (AuthRealm è il nome del realm già impostato). Le astrazioni messe a disposizione da PassportJS, ci consentono di accedere direttamente all'URL indicando solo l'entità da contattare e, in base alla risposta di quella richiesta, verrà avviato il server o si riproverà dopo dieci secondi.
Successivamente abbiamo la necessità di impostare alcuni parametri per utilizzare quegli endpoint per cui, una volta ritornato un oggetto Issuer che mantiene in ricordo tutti gli endpoint del JSON, istanzieremo un oggetto di tipo Client che dentro avrà: clientId e clientSecret per usare gli endpoint; il redirect_uri che verrà utilizzato come primo redirect al fine di impostare i cookie e il path per effettuare il logout. Una volta creato l'oggetto client potremo usare le API ricavate dal JSON con la sicurezza che siano considerati tutti i parametri indicati in fase di impostazione.
Per fare tutto ciò, però, devo impostare una strategia con cui usare PassportJS e la scelta è ricaduta sull'uso di OpenID Connect ovvero un protocollo di autenticazione di generazione OAuth che si basa sull'uso di un Auth Server e ci offre le astrazioni di cui abbiamo bisogno in passport. Riportiamo nel seguito il listato:

```
passport.use('oidc', new Strategy({ client }, async function (tokenSet, _,
done) {
    console.log('Checked Authentication....')
    const informationsUser = await
client.introspect(tokenSet.access_token)

    let providedData = {
        "user_name": informationsUser.username,
        "role": informationsUser.realm_access.roles[0],
        "token": tokenSet.access_token,
        "refresh_token": tokenSet.refresh_token,
        "id_token": tokenSet.id_token,
        "authenticated": true
    }
        done(null, providedData);
    })
)
```

Figura 15. Definizione della strategia di PassportJS

Come possiamo vedere, definiamo una strategia di autenticazione openid basata sul client che abbiamo definito e ci viene data la possibilità di accedere al tokenSet che altro non è che un oggetto JSON che contiene solo i token. Dato che Grafana ci fornisce possibilità di aggiungere Header aggiuntivi che ci consentono l'autenticazione, abbiamo deciso di rendere direttamente disponibili il nome e il ruolo dell'utente. Inoltre, dato che abiliteremo anche l'autenticazione basata su segreto vi è la necessità di contraddistinguere le due tipologie di autenticazione per cui viene aggiunta una proprietà "authenticated".

Ora analizzeremo gli endpoints necessari all'autenticazione per effettuare il login, in particolare ricordiamo i seguenti obiettivi:

- Dobbiamo capire se l'utente sia autenticato o meno prima di rindirizzarlo alla pagina di login
- Dopo il login, dobbiamo fare un redirect verso la pagina precedente (la pagina in cui l'utente ha acceduto al form di login)
- Rilasciare i cookies che verranno usati esternamente per capire che l'utente è autenticato

In base a tali obiettivi, sono state definiti i seguenti endpoints:

```
/* The first route redirects the user to the Provider, where they will
authenticate */
app.get('/auth/login', savePageSession, passport.authenticate('oidc'));

/* The second route processes the authentication response and logs the
user in,
when the OP redirects the user back to the app */
app.get('/auth/login/callback', passport.authenticate('oidc'), async
(req, res) => {
    const current_user_info = {
        "token": req.user.token,
        "refresh_token": req.user.refresh_token,
        "id_token": req.user.id_token,
        "authenticated": true
    }
    const path = cookieParser.JSONCookies(req.cookies).path
    res
        .cookie("user_info", current_user_info)
        .cookie("User", req.user.user_name)
        .cookie("Role", req.user.role)
        .clearCookie("path")
        .redirect(path)
});
```

Figura 16. Endpoints di login

Tralasciando il nostro problema di rilascio dei cookies, abbiamo necessità di mantenere due endpoints per fare il login. La motivazione è legata alle necessità del nostro Identity Provider; infatti, il primo path /login serve per capire se l'utente è autenticato (questo lavoro viene fatto da "passport.authenticate('oidc')") e ci possiamo trovare in due situazioni:

- L'utente non è autenticato, per cui viene fatto un redirect alla pagina di Login di Keycloak
- L'utente è autenticato per cui viene fatto un redirect verso /auth/callback e viene impostata una sessione con Keycloak

L'ausilio del secondo endpoint /auth/login/callback è necessario perché Keycloak riesca ad impostare una sessione con l'utente. Andando più nel dettaglio della funzione GET /auth/login, notiamo la presenza di una middleware definita nel seguente modo:

```
const savePageSession = (req, res, next) => {
const clientPath =`${basePath}${req.query.redirect}`
    res.cookie("path", clientPath)
    next()
}
```

Figura 17. Middleware savePageSession

Quest'ultima serve per salvare in un cookie "path", la variabile redirect definita all'atto della richiesta di login al server in modo da avere a disposizione il path completo a cui fare il redirect successivamente in /auth/login/callback. Passando poi al secondo endpoint di autenticazione, possiamo accedere alla sessione creata con la strategia (questo sarà l'unico caso in cui lo faremo perché dobbiamo ricavare le informazioni dell'utente per impostare i cookies) e lo possiamo fare perché, precedentemente, nella definizione della strategia abbiamo definito un oggetto JSON che rappresenta l'utente i cui campi sono accessibili con "req.user". Successivamente, prima di fare il redirect, puliamo il cookie "path" dato che ci servirà solo in questo contesto. Andando un po' più nel dettaglio ciò che ci rimarrà della sessione, sarà essenzialmente la presenza dei cookie che sono definiti nel seguente modo:

- user che rappresenta lo username dell'utente
- role che rappresenta il ruolo utilizzato dall'utente
- user_info che rappresenta un oggetto JSON contenente il token, il refresh_token, l'id_token e un flag sull'autenticazione

Per un client, è importante capire come riconoscere se un utente è autenticato o meno. In particolare, un client sviluppato in ReactJS spesso fa uso di rendering condizionato che può basarsi sull'uso di un tipico endpoint di /userInfo per comprendere se l'utente sia autenticato o meno. Per tale motivo, abbiamo messo a disposizione tale URL a cui fare riferimento:

```
app.get("/userInfo", async (req, res) => {
 const jsonFormatCookie = cookieParser.JSONCookies(req.cookies)
 if (jsonFormatCookie.user_info) {
       const response = await
client.introspect(jsonFormatCookie.user_info.token)
       if (response.active) { res.status(200).json(response)}
       else { res.status(401).end() }
} else {
      res.status(401).end()
    }
})
```

Figura 18. Definizione dell'endpoint per le informazioni dell'utente

Notiamo che una volta formattati i cookie in formato JSON grazie alla libreria cookie-parser, se è presente il token ed è valido (lo capiamo grazie all'oggetto Client che chiama l'API di introspect del token di Keycloak) allora l'utente è autenticato diversamente ritorneremo uno stato 401 cosicché il client capisca che l'utente non è autenticato. In caso il controllo vada a buon fine, ritorniamo un oggetto JSON che contiene il token JWT in modo che il client possa usarlo anche direttamente per ricavare più informazioni possibili sull'utente come l'e-mail per esempio.

Fornito il processo di autenticazione, soffermiamoci sul processo di logout. In particolare, l'obiettivo è quello di mettere a disposizione un endpoint sul nostro server che renda il token invalido in modo che non possa essere più utilizzato per cui, anche in questo caso, ci affideremo alle API di Keycloak che sono astratte dall'oggetto client, nel seguito riportiamo il codice:

```
app.post('/auth/logout', (req, res) => {
    const jsonFormatCookie = cookieParser.JSONCookies(req.cookies)
    req.logout(() => {
        res
            .clearCookie("User")
            .clearCookie("Role")
            .clearCookie('user_info')
            .redirect(client.endSessionUrl({
                post_logout_redirect_uri: logoutUri,
                id_token_hint: jsonFormatCookie.user_info.id_token
            }))
    })
})
```

Figura 19. Endpoint per effettuare il logout

Come possiamo notare dal codice definito, vengono ripresi i cookies perché abbiamo necessità di avere l'identificativo del token JWT da invalidare. Viene fatta quindi una pulizia dei cookies presenti sul browser e successivamente viene fatto un redirect verso il path definito per il logout disabilitando il token grazie al suo identificativo.

Cerchiamo ora di entrare più nel dettaglio degli obiettivi post autenticazione. Precedentemente abbiamo detto di voler garantire la visualizzazione dei grafici previa autenticazione, il risultato è raggiungibile grazie ai cookies che abbiamo predisposto e, al momento, ci basta sapere che i cookies di cui abbiamo necessità sono il nome dell'utente (User) e il suo ruolo (Role) che Grafana userà per capire che l'utente è stato autenticato. Il problema però è il seguente: Grafana non può basarsi su User e Role solamente per capire che l'utente è autenticato dato che chiunque potrebbe fare l'injection di quei cookies ed è qui che entra in gioco l'endpoint /auth/check. L'endpoint è stato costruito con l'intendo di garantire la validazione del token e, nel caso scadesse, di fare il refresh. Ovviamente si può supporre di avere un tempo di

validità del token molto ampio ma stiamo semplificando la vita agli attaccanti dato che più il token JWT è valido maggiori sono le possibilità che venga rubato e usato per cui abbiamo la necessità di fare il refresh. Riportiamo nel seguito il codice:

Il codice, per quanto possa sembrare complesso, è abbastanza semplice da seguire.

```
app.get("/auth/check", checkRole, async (req, res) => {
    const jsonFormatCookie = cookieParser.JSONCookies(req.cookies)
    try {
            const tokenInformations = await
      client.introspect(jsonFormatCookie.user_info.token)
        if (tokenInformations.active) { res.status(200).end()}
        else if (jsonFormatCookie.user_info.token && !tokenInformations.active){
            console.log("Refreshing...")
            const refreshing = await
      client.refresh(jsonFormatCookie.refresh_token)
            const new_user_info = {
                "token": refreshing.access_token,
                "refresh_token": refreshing.refresh_token,
                "id_token": refreshing.id_token,
                "authenticated": true
            }
            res
                .cookie('user_info', new_user_info)
                .status(200)
                .end()

        } else { res.status(401).end() }
    } catch(e) {
        res.status(401).end()
    }
})
```

Figura 20. Endpoint per la validazione e il refresh del token

Dato che potrebbe essere chiamata ovunque, potremmo non trovare il token nei cookies per cui viene fatto un controllo sull'errore dato che la introspect se ci sono valori undefined lancia un eccezione per cui nel caso ritorniamo 401. Se invece la stringa è valida, la controlliamo e se attivo il token ritorniamo 200 facendo passare le richieste verso Grafana. Diversamente useremo il refresh token per ottenere un nuovo token set e rimpostare i cookies. Così come il token JWT ha una scadenza anche il suo refresh token ha una scadenza, molto più lunga ma bisogna comunque considerarla. Nel caso il refresh token non fosse più valido, allora basterà ritornare una risposta 401 per dire che l'utente non è più autenticato. Sottolineiamo, però, l'importanza della durata del token JWT: in fase di sviluppo la sua durata è stata impostata a cinque minuti (durata minima fornita da Keycloak) ma si potrebbe aumentare con la conseguenza di rendere più facile la vita agli attaccanti.

Un altro obiettivo che ci siamo prefissati è quello di garantire l'autenticazione mediante segreto o chiave privata. In particolare, è stata solamente definita una logica di autenticazione con chiave privata lasciando ai posteri la possibilità di implementare completamente il processo dietro la validazione tramite controllo sul Database. Keycloak, di norma, non mette a disposizione API che ci consentono di utilizzare una chiave privata per autenticarsi ma mette a disposizione le cosiddette "REST Admin API" che possono essere utilizzate tramite un utente di Keycloak. A tale scopo, abbiamo definito all'interno del realm di development un utente Admin le cui credenziali vengono riportate all'interno del file di environnement. L'idea è rilasciare un token basilare per un utente anonimato dandogli il semplice ruolo di Viewer (ruolo più basso nella gerarchia di Grafana). Una volta fatto ciò, per dire a Keycloak che esiste un utente devo necessariamente chiamare la funzione req.login che fornisce possibilità di creare una nuova sessione.

```
app.get('/auth/key/:key', ensurePermissions, async (req, res) => {

    const url = process.env.keycloackUrl + '/protocol/openid-connect/token'
    const result = await fetch(url, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body:
`grant_type=password&client_id=${clientId}&client_secret=${clientSecret
            }&username=${adminUser}&password=${adminPassword}`
    })

    const resultBody = await result.json()

    const user_info = {
        "token": resultBody.access_token,
        "refresh_token": resultBody.refresh_token,
        "id_token": resultBody.id_token,
    }

    res
        .cookie('User', "Anonymous")
        .cookie('Role', "Viewer")
        .cookie('user_info', user_info)
        .json(resultBody.access_token)
});
```

Figura 21. Endpoint per login con segreto

In questo caso, lo scopo è quello di passare un link (es. tramite e-mail) per visualizzare i grafici. Si può utilizzare questa API per creare la sessione temporanea e creare i cookie necessari a far funzionare tutto. Come possiamo notare, la scelta è stata quella di evitare di usare "authenticated" dato che l'idea è usare una chiave privata che creerà un token nuovo ogni qualvolta ci si autentica tramite questa API. Insieme a questa API, è possibile avere anche un middleware associata ma, dato che si tratta di un obiettivo

secondario del progetto di Tesi, si è deciso solo di implementare la logica business e non il backend dietro alla presenza della chiave nel Database. Ne riportiamo il funzionamento nella figura seguente:

```
const ensurePermissions = async (req, res, next) => {

    // Pass the key throguh an header
    const passedKey = req.params.key
    // All the Database Logic
    // db.get(...)
    // Supposed it's en encrypted key returned
    var encryptedKey =
"$2b$10$6duFdqQAPVuqP0G7NX1BBegac/F9WNLWLZjYlrkxzk.KDNhCuzL0m"

    const isValid = await bcrypt.compare(passedKey, encryptedKey);
    if (isValid)
        next();
    else
        return res.status(401).send({ message: 'Invalid key', code: 401 });

};
```

Figura 22. Codice per il controllo della chiave privata

Grazie all'ausilio della libreria bcrypt, è possibile fare il confronto tra la chiave che ho passato tramite URL e la chiave cifrata che vado a prendere dal DB. Al momento, solo una chiave è stata sperimentata e nel caso fosse valida l'utente potrà ricevere il token, altrimenti si ritorna un errore 401 con messaggio "Invalid Key".

Configurazione di Keycloak

Fino ad ora ci siamo soffermati sulla configurazione del server di autorizzazione, estraniandoci da quello che succede su Keycloak. Prima di riassumere il processo di autorizzazione/autenticazione, cerchiamo di comprendere come Keycloak sia stato configurato. Riprendiamo quindi un analisi più dettagliata del file Docker compose dedicata a Keycloak:

```
version: "3"
services:
  keycloack:
    image: quay.io/keycloak/keycloak:latest
    environment:
      KEYCLOAK_ADMIN: ${KEYCLOAK_ADMIN}
      KEYCLOAK_ADMIN_PASSWORD: ${KEYCLOAK_ADMIN_PASSWORD}
      KC_FEATURES: declarative-user-profile
      KC_PROXY: edge
    volumes:
      - ./keycloak/realm/.:/opt/keycloak/data/import
      - ./keycloak/plugins/.:/opt/keycloak/providers
    command:
      - start-dev
      - --import-realm
      - --hostname-url=https://${DOMAIN}/
    restart: unless-stopped
    healthcheck:
      test: "exit 0"
```

Figura 23. Docker compose file Keycloak

Al fine di accedere alla console di Admin messa a disposizione da Keycloak per la gestione degli utenti, sono state definite delle variabili d'ambiente che permettono di definire il nome utente e la password. Inoltre, dato che noi siamo interessati a validare le emails durante il processo di registrazione, abbiamo aggiunto la feature denominata "declarative-user-profile". La gestione, sebbene fatta da Keycloak in maniera impeccabile, non offre un controllo completo sulla e-mail ma solo sul suo dominio ma il problema è che il controllo può essere fatto solamente tramite regex di Java per cui la funzionalità è abbastanza limitata. A tale scopo, insieme a Keycloak, è stato sviluppato un plugin aggiuntivo per il controllo delle e-mail in modo da permettere a un Admin di far registrare solo chi ne ha il diritto ovvero che si presenta all'atto della registrazione con una precisa e-mail. Il plugin di Keycloak è stato scritto in Java e viene caricato al suo avvio, per cui abbiamo mappato una cartella in cui inserire tutti i file .jar denominata "plugins". Per importare nuovi realm, basterà sostituire il file .json all'interno della cartella /realm. In particolare, dalla cartella /realm, Keycloak importerà tutti i dati di configurazione compresi gli utenti (qualora ce ne fossero).

Infine, dato che Keycloak si trova dietro a un proxy, abbiamo abilitato l'opzione "Edge" in cui il nostro proxy farà da terminatore SSL e Keycloak comunicherà usando HTTP. Tra i comandi, quello più interessante è sicuramente "—hostname-URL" perché ci permette di ottenere un file di configurazione degli URL di Keycloak dietro al proxy (senza quel comando, gli URL sarebbero http://Keycloak/<some> che non sarebbero raggiungibili direttamente). Cerchiamo prima di analizzare come è stato configurato Keycloak per garantire il processo di autenticazione e, nel seguito, verranno indicati i comandi per salvare il proprio realm. Innanzitutto, possiamo accedere a Keycloak mediante l'indirizzo "https://ns01-prometeo.polito.it/Keycloak" che ci porta alla seguente pagina:

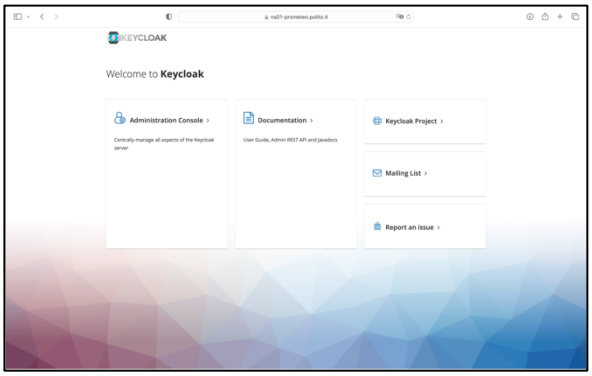

Figura 24. Pagina principale di Keycloak

Da qui è possibile accedere alla console di amministrazione, alla documentazione e poi a tutta una serie di pagine relative ai progetti Keycloak. Una volta cliccato su "Administration Console" verremo portati alla pagina di login per l'amministratore, all'interno della quale dovremo porre le nostre credenziali, riportiamo nel seguito la pagina in questione:

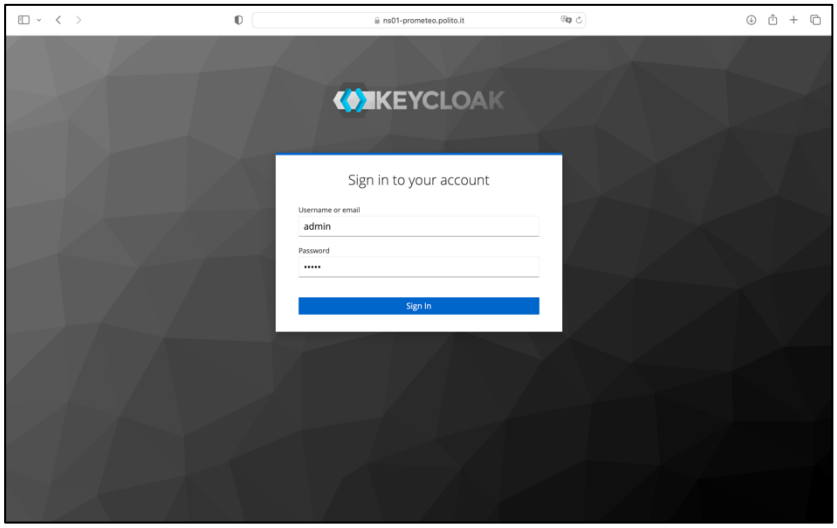

Figura 25. Administration Console di Keycloak

Una volta inserite le credenziali, si accederà alla pagina dedicata al master realm di default. Al momento, ai fini di sviluppo, abbiamo configurato Keycloak con un realm chiamato AuthRealm e, per passare da un realm all'altro, si clicca in alto a sinistra e si seleziona sul realm desiderato:

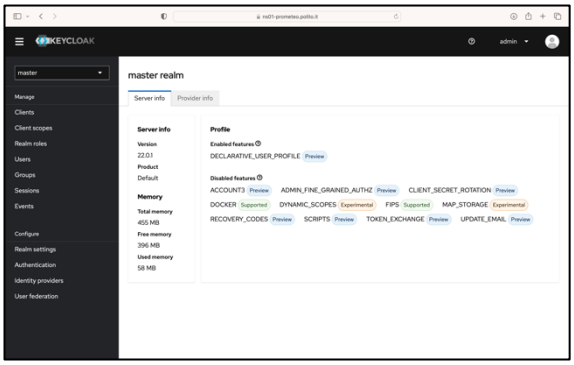

Figura 27. Master Realm Keycloak

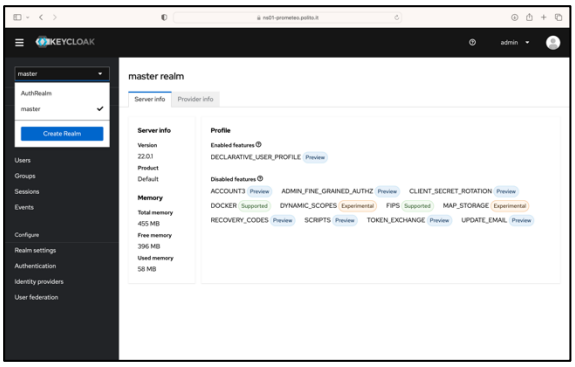

Figura 26. Selezione di AuthRealm

Come possiamo vedere dalla figura 26, è possibile creare un ulteriore realm, cliccando sul pulsante "Create Realm" e andando a configurare gli utenti e i "client" di cui noi avremo bisogno. Una volta passati a AuthRealm, possiamo visualizzare tutti i client disponibili che nel contesto di Keycloak sono le applicazioni che usufruiscono dei suoi servizi. Abbiamo deciso di rinominare il nostro client con "Grafana" e tramite click sul nome è possibile accedere a tutte le informazioni di configurazione:

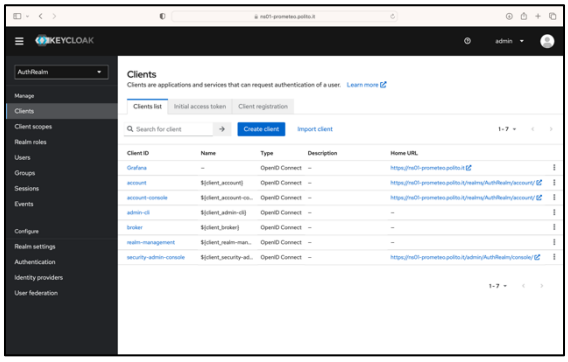

Figura 28. Lista dei clients Keycloak

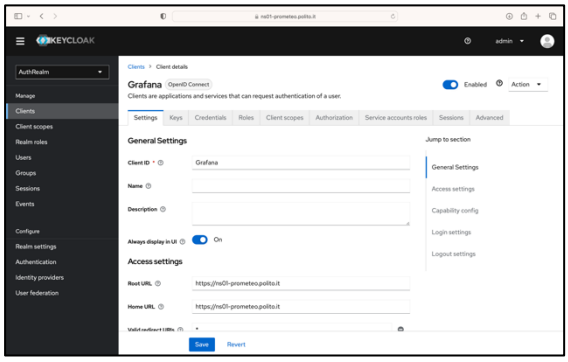

Figura 29. Informazioni del client Grafana in Keycloak

Come possiamo notare, ci sono una serie di impostazioni come il clientID che abbiamo definito nel Docker compose per il nostro server ed è possibile accedere al segreto

associato nel tab “Credentials”. La cosa interessante è che è possibile vedere le impostazioni create precedentemente in “Access settings”:

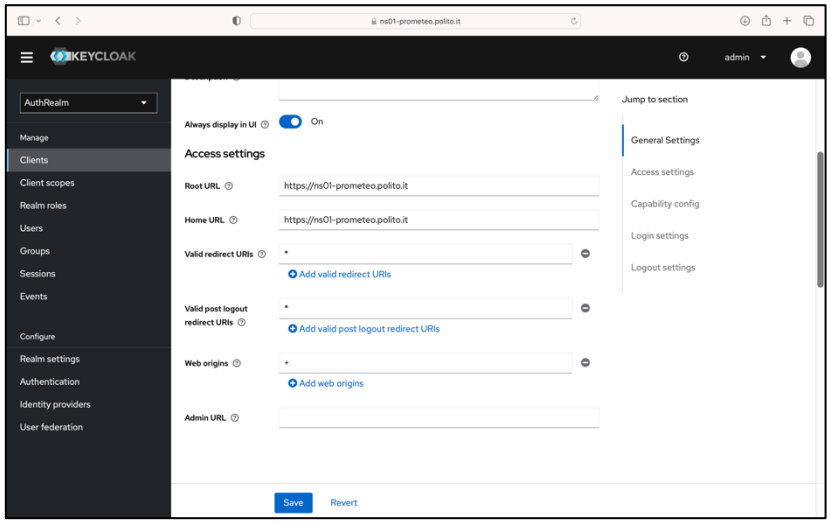

Figura 30. Settings di Grafana in Keycloak

Notiamo alcuni URL come il root URL e home URL che corrispondono all’URL base del sotto dominio, in particolare: root URL indica l’URL base a cui vengono aggiunti ulteriori path mentre l’home URL è quello da usare nel caso di redirect o di ritorno al client. Vengono poi definiti ulteriori URL come: “valid redirect URIs” e “valid post logout redirect URIs” che usano il simbolo “*” per indicare che qualsiasi path indicato rispetto all’URL va bene (ovviamente ciò viene definito nel server, Keycloak deve solo validare se l’URL usato sia valido o meno con le configurazioni). Infine, “web origins” serve per motivi di CORS, in particolare con “+” stiamo indicando che qualsiasi applicazione può comunicare con Keycloak.
Per arrivare a una configurazione di questo tipo è ovvio che bisogna passare per i pulsante “Create Client” che permette di creare un nuovo client per Keycloak e quindi di ottenere una serie di credenziali per accedere ai servizi. Ne riportiamo le tre schermate di creazione del client nel seguito:

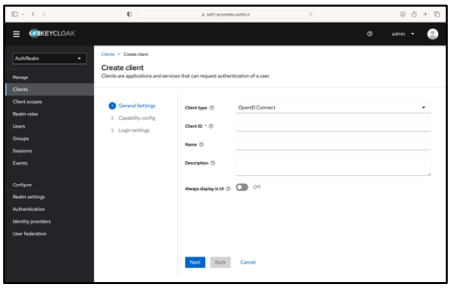

Figura 33. Configurazione Client Keycloak Schermata 1

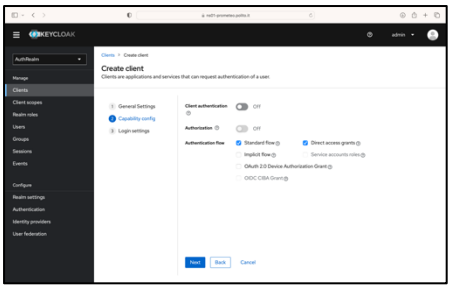

Figura 32. Configurazione Client Keycloak Schermata 2

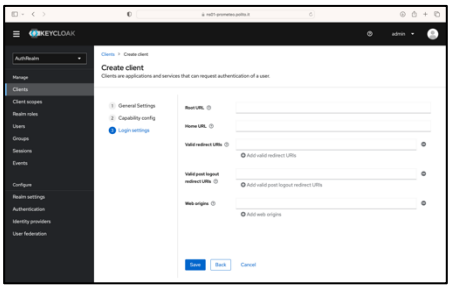

Figura 31. Configurazione Client Keycloak Schermata 3

Come possiamo notare, tutte le configurazioni che noi troviamo nella pagina dedicata al client vengono presentate in maniera coerente. Nella prima schermata, possiamo inizializzare il valore del Client ID (possiamo inserire qualsiasi cosa ma aiuta usare il nome dell'applicazione per riconoscerlo). Dalla seconda schermata, possiamo abilitare la autenticazione e l'autorizzazione. Nella terza schermata vengono inizializzati gli URL precedentemente analizzati. In ultimo, sono stati importati degli utenti precedentemente creati che è possibile visualizzare nel tab "Users" e si tratta di utenti che sono disponibili a tutti i client:

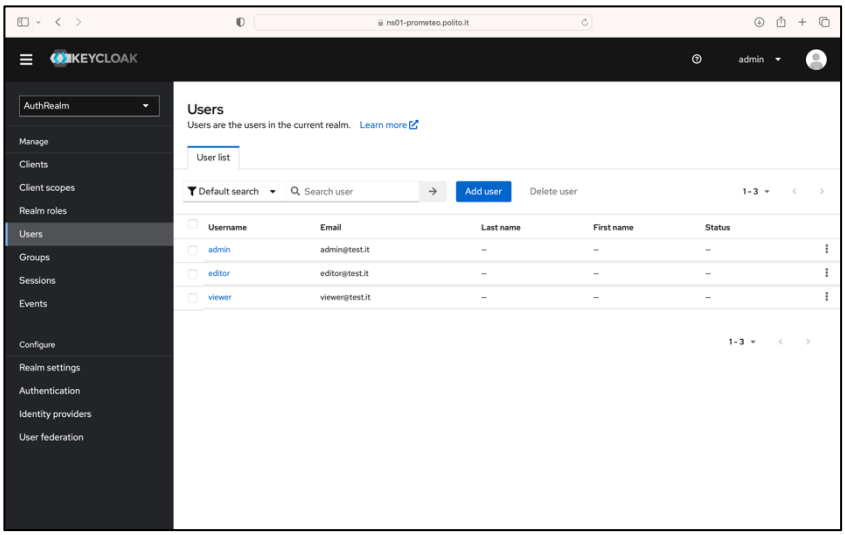

Figura 34. Schermata User Keycloak

Per ogni ruolo presente in Grafana, è stato possibile definire una serie di utenti. In particolare, si è definito anche l'utente admin che corrisponde sia per Keycloak che per Grafana. Inoltre, sempre a livello di Realm sono stati definiti i ruoli che sono quelli che noi troviamo in Grafana. La definizione dei ruoli avviene all'interno del tab "Realm roles" e per crearlo basta cliccare su "Create role" e aggiungere il nome del ruolo:

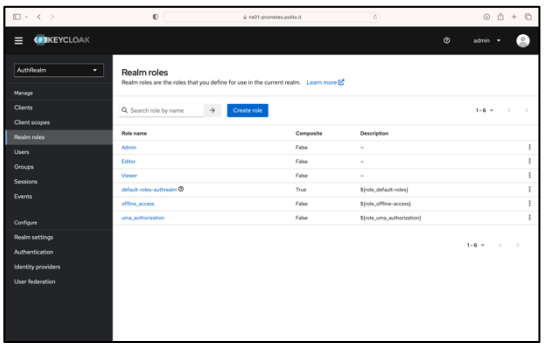

Figura 36. Create Role Schemata 1 Keycloak

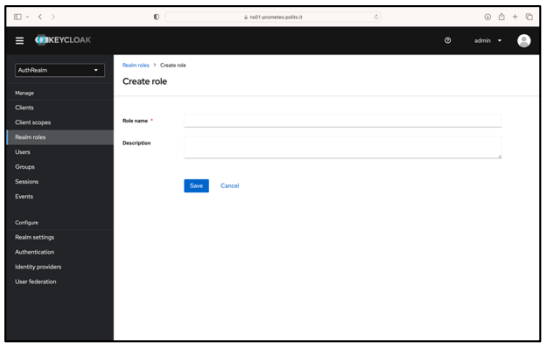

Figura 35. Create Role Schermata 2 Keycloak

Una volta creato il ruolo, è possibile assegnarlo all'utente mediante click sul nome dell'utente, "Role Mapping" e cliccando "Assign Role". Quest'ultimo aprirà un tab che ci permetterà di selezionare il ruolo che noi vogliamo tra quelli disponibili. La creazione di un utente, invece, avviene mediante il tab "Users" e poi facendo click su "Add user" in cui poi è possibile inserire tutte le informazioni relative al nuovo utente, nel seguito riportiamo la pagina di configurazione:

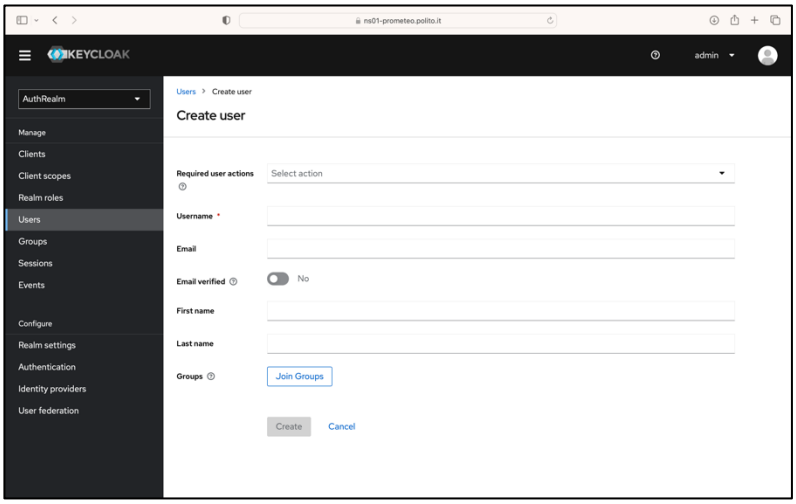

Figura 37. User Configuration Keycloak

Concludiamo il discorso Keycloak, indicando il procedimento di esportazione della configurazione. All'interno del container che viene messo a disposizione è possibile importare i dati, come abbiamo visto, mediante il comando "—import realm". L'importante è che il file JSON, che ci siamo costruiti, venga inserito in una cartella Keycloak dato che lì mapperemo il volume di import del container. Per esportare una configurazione di Keycloak basta andare nella cartella "/opt/Keycloak/bin" e successivamente usare il comando "/kc.sh export --file <file>" indicando al posto di <file> il nome ed automaticamente esporterà il file in formato JSON con tutte le configurazioni all'interno della stessa cartella dove abbiamo chiamato tale comando. Dal programma Docker, è possibile poi scaricare tale file cliccando sul container, andando in "files" e andare nel path "opt/Keycloak/bin".

Configurazione NGINX

Tutte le API che sono definite nel server di autorizzazione hanno il loro corrispettivo nel server NGINX. Questo perché sono accessibili all'interno dello stesso dominio, da un punto di vista dell'utente, come "https://nsXX-prometeo.polito.it". Tale server si occupa di effettuare il lavoro di reverse proxy ovvero quello di fare da "schermo" verso le API. L'idea è fornire una configurazione base che definiremo all'interno di una cartella e definire una serie di Header aggiuntivi oppure da far passare. Inoltre, dato che all'interno dello stesso dominio si parla HTTP, non avremo la necessità di configurare certificati di alcun tipo ed inoltre di esportare porte esterne perché non bisogna permettere all'utente di accedere usando HTTP ma HTTPS. In particolare, all'interno del file di Docker compose viene configurato nel seguente modo:

```
app:
    image: nginx
    volumes:
      - "./nginx/nginx.conf:/etc/nginx/nginx.conf"
      - "./nginx/includes/:/etc/nginx/includes/"
    depends_on:
      - keycloack
      - server
      - client
      - grafana
```

Figura 38. Docker compose file NGINX

Come possiamo notare mappiamo un file di configurazione da una cartella NGINX in locale e un ulteriore cartella /includes che mi permette di definire dei file .conf come Header di configurazione da usare in più parti del file di configurazione di NGINX. Inoltre, la presenza di tutte queste dipendenze è dovuta alla necessità di essere sicuri che il server NGINX faccia raggiungere servizi effettivamente disponibili e non ritorni un errore 502 Bad Gateway.
Nelle successive pagine, analizzeremo nel dettaglio le "location" che sono state definite soffermandoci solo su quelle che al momento sono necessarie lasciando il resto nei prossimi paragrafi.

La prima parte che analizzeremo sarà dedicata a Keycloak, questo perché vogliamo garantire la gestione dell'IdP all'Admin dietro alla connessione HTTPS rendendo così sicura la comunicazione. In particolare, seguendo la documentazione di Keycloak per una tipologia di proxy di tipo "Edge", ho necessità di fare il forwarding di tutti gli Header di tipo "X-Forwarded-*" perché non viene fatto in automatico dal reverse proxy e Keycloak non si prende in carico il lavoro di mantenere uno stato. Sulla base di ciò, abbiamo il seguente file .conf definito in /includes per il forwarding degli Headers:

```
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Host $host;
proxy_set_header X-Forwarded-Server $host;
```

Figura 39. File per il forwarding degli Header Keycloak.conf

Un'altra questione che rimane aperta è la seguente: se spostiamo Keycloak dietro a un reverse proxy, mi aspetto che tutte le risorse associate ad esso siano messe a disposizione dietro al reverse proxy. Per tale motivo, seguendo sempre la documentazione di Keycloak, sono state definite diverse "location" che rimanderanno alla dashboard admin, risorse di realm ecc. di Keycloak. Ne analizziamo in dettaglio il contenuto:

```
location /keycloak/ {
    proxy_pass http://keycloack:8080/;
    include /etc/nginx/includes/keycloak.conf;
}

location /realms/ {
    proxy_pass http://keycloack:8080/realms/;
    include /etc/nginx/includes/keycloak.conf;
}

location /admin/ {
    proxy_pass http://keycloack:8080/admin/;
    include /etc/nginx/includes/keycloak.conf;
}

location /resources/ {
    proxy_pass http://keycloack:8080/resources/;
    include /etc/nginx/includes/keycloak.conf;
}

location /js/ {
    proxy_pass http://keycloack:8080/js/;
    include /etc/nginx/includes/keycloak.conf;
}
```

Figura 40. Sezione Keycloak nel file di configurazione NGINX.conf

Come notiamo dalla configurazione, il path /Keycloak permette di raggiungere la schermata iniziale in cui è possibile accedere anche alla Admin Dashboard. Assieme ad esso, ci sono altre “location” che sono state configurare come /realms, /admin, /resources ecc. che servono per configurare totalmente Keycloak dietro al reverse proxy. Dato che usiamo NGINX come reverse proxy, possiamo aggiungere/eliminare Header a nostro piacimento e, in particolare, ci sono una serie di proxy_set_Header che verranno usati nella maggior parte dei casi:

- Host che serve per ricavare le informazioni di chi ha fatto la richiesta
- X-Real-IP serve come direttiva per impostare l'intestazione "X-Real-IP" nella richiesta proxy con l'indirizzo IP del client che sta effettuando la richiesta.
- X-Forwarded_For che per impostare l'elenco degli indirizzi IP intermedi attraverso cui la richiesta è passata. Questo è particolarmente utile quando ci sono più livelli di proxy tra il client e il server backend. In questo modo, il server backend può accedere all'indirizzo IP del client originale, anche se la richiesta è stata inoltrata attraverso più proxy.
- X-Forwarded-Host e X-Forwarded-Server indicano l’host originale e il server originale che hanno effettuato la richiesta

Per quanto riguarda il server, tutte le API vengono messe dietro al Reverse Proxy NGINX e, l’unica differenza interessante rispetto agli Header inoltrati con Keycloak.conf è la presenza di “proxy_set_Header Cookie $http_cookie;”. La motivazione è legata al fatto che noi stiamo rilasciando dei cookies al client per cui abbiamo necessità di inserire tale Header (come possiamo vedere nella figura 41).

```
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Forwarded-Host $host;
proxy_set_header X-Forwarded-Server $host;
proxy_set_header Cookie $http_cookie;
...
```

Figura 41. File per il forwarding degli Header proxy.conf

L’unica eccezione, per quanto riguarda le API del server, è fatta da “/auth/check” che viene utilizzata solamente per la gestione dei grafici per cui la analizzeremo più tardi.

Overview dell'autenticazione

In conclusione, cerchiamo di capire come funziona il processo di autenticazione complessivamente:

1. L'utente accede all'indirizzo https://nsXX-prometeo.polito.it venendo rindirizzato verso la pagina base dell'applicazione Client:

Figura 42. Pagina https://ns01-prometeo.polito.it

2. L'utente clicca sul bottone di login chiamando così l'API "/auth/login?redirect=/" messa a disposizione da NGINX dietro https://nsXX-prometeo.polito.it/auth/login. Viene così fatto un controllo su sé l'utente sia autenticato:

    a. Se l'utente è autenticato, verrà chiamata l'API di "/auth/login/callback" che creerà una sessione tra Auth Server e Keycloak momentanea con lo scopo di rilasciare dei cookies.

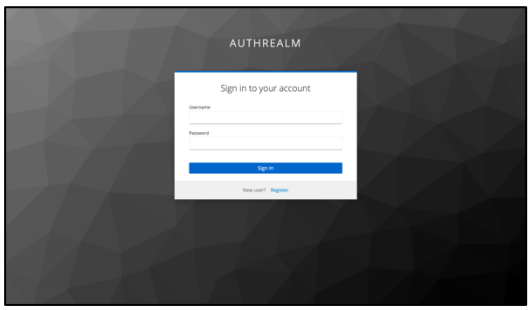

Figura 43. Pagina di autenticazione di Keycloak

    b. Se l'utente non è autenticato, verrà rindirizzato alla pagina di autenticazione di Keycloak in cui dovrà compilare un form. Successivamente, la chiamata "/auth/login/callback" farà il controllo sulla veridicità delle credenziali e setterà una sessione temporanea per il rilascio del token che verrà reso disponibile tramite i cookies.

3. Rilasciati i cookies, l'utente potrà navigare tranquillamente sul sito web partendo dal sito indicato tramite il parametro "redirect" nella "/auth/login". Allo stesso tempo il client React non dovrà occuparsi di mandare alcun Header di autenticazione dato che viene tutto gestito lato Auth Server che prenderà i cookie e controllerà che il token sia valido mediante "/userInfo".

Opzionalmente può essere usata l'autenticazione basata su chiave primaria ma in questo caso si tratta dello stesso procedimento (tranne per l'inserimento delle credenziali username/password) e viene fatto il redirect diretto verso la pagina di home. Nel seguito, viene messa a disposizione una rappresentazione dettagliata del processo di autenticazione in caso in cui l'utente non è autenticato precedentemente:

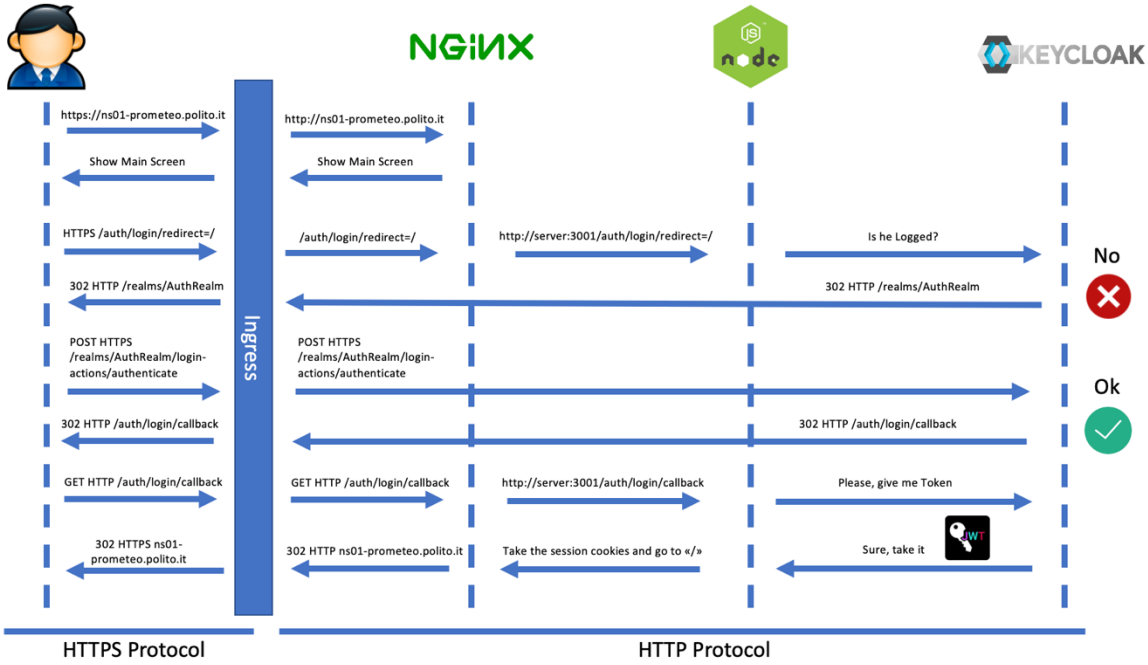

Figura 44. Autenticazione per la rete ns01-prometeo.polito.it

Rendering dei grafici

Come abbiamo esaminato precedentemente nel capitolo 2, l'obiettivo principale della Tesi è quello di garantire un processo di autenticazione che sia il più indipendente possibile rispetto all'applicazione Web e che funga da servizio esterno utilizzabile mediante alcune API. L'idea base è aggiungere tale processo ma poi andare nella direzione di fare un rendering grafico dei dati, raccolti mediante Grafana, previa autenticazione e soprattutto l'utente che vuole visualizzare ne deve avere i permessi. Esternamente ci siamo occupati del processo di autenticazione ma lasciamo alla piattaforma la possibilità di fare un controllo sugli accessi basati sul ruolo, in particolare Grafana introduce la possibilità di rendere visibile un grafico solo quando l'utente ha i permessi (ovvero ha un determinato ruolo oppure appartiene a un preciso team). Nel seguito, faremo riferimento alla definizione nel Docker compose di Grafana e al suo file di configurazione ma soprattutto cercheremo di capire come Grafana riesca, dopo aver effettuato l'autenticazione, a permettere di accedere ai grafici e anche alla sua dashboard.

Configurazione di Grafana

Andiamo a dare un'occhiata al file Docker compose:

```
grafana:
    image: grafana/grafana
    depends_on:
      - postgres
    volumes:
      - ./grafana/grafana_info:/var/lib/grafana
      - ./grafana/grafana.ini:/etc/grafana/grafana.ini
    environment:
      - GF_INSTALL_PLUGINS=marcusolsson-json-datasource
    restart: unless-stopped
postgres:
    image: postgres:latest
    volumes:
      - ./database_grafana/postgres-
data:/var/lib/postgresql/data
      - ./database_grafana/.:/docker-entrypoint-initdb.d
    environment:
      - POSTGRES_PASSWORD=password
      - PGDATA=/var/lib/postgresql/data/db
    restart: unless-stopped
```

Figura 45.Docker compose file di Grafana e PostgreSQL

Come possiamo notare, abbiamo deciso di mantenere il database di Grafana senza affidarlo a un DBMS esterno, in particolare abbiamo mappato sia il file di configurazione (Grafana.ini) che il database in locale all'interno di una cartella "Grafana". Insieme ad esso, abbiamo definito un servizio PostgreSQL che va a simulare i dati raccolti dai sensori con lo scopo di andare a definire grafici di test in Grafana.

Generazione dei dati di testing

Per la generazione dei dati di testing si è seguito un procedimento molto banale. Come prima cosa, Grafana mette a disposizione la possibilità di ricavare dei dati di testing per verificare che l'ambiente di sviluppo funzioni correttamente. Per utilizzare questi dati di testing bisogna seguire il seguente procedimento:

- Cliccare sul menu in alto a sinistra
- Cliccare su connessioni
- Cercare nella sezione "Add connections" la parola chiave "Test Data"
- Cliccare il pulsante di creazione dei dati di testing

Una volta fatto ciò sarà possibile definire un nome alla sorgente dei dati di testing e, successivamente, sarà possibile salvare oppure eliminare la sorgente. Per poter utilizzare tali dati, bisogna creare una nuova dashboard andando nel menù dedicato ad esse e successivamente è possibile creare dei grafici utilizzando come sorgente i nostri "Test Data" selezionando l'opportuna sorgente. Una volta creato un panel all'interno di Grafana, è possibile scaricare i dati in formato csv ed è possibile farlo cliccando sul menu posto in alto a destra, cliccando su inspect e poi data. Da qui apparirà un menu che permetterà di scaricare il file csv dei dati a cui possiamo aggiungere anche dei vincoli sulla formattazione e sul formato dei dati.

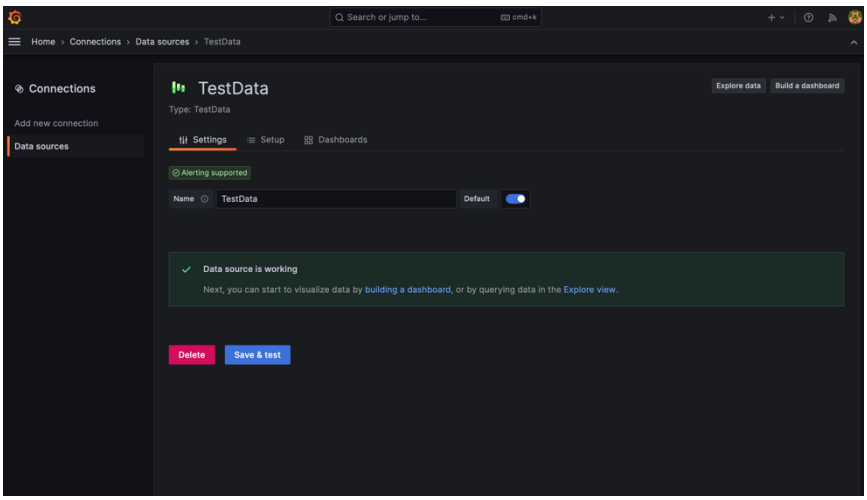

Figura 46. Test Data in Grafana

Configurazione di PostgreSQL

Come abbiamo visto all'interno del Docker compose è presente anche il servizio PostgreSQL che useremo come sorgente dati di testing. Per inserire all'interno di PostgreSQL dei dati, necessitiamo di uno script di inizializzazione che, partendo dal CSV, estragga le colonne e crei, almeno, due tabelle. La struttura delle tabelle sarà la seguente: la prima colonna indicherà il momento in cui è stata effettuata la misurazione; mentre la seconda indicherà il valore catturato. Quando verrà chiamato il comando "Docker compose up --build", a meno che non esista già, verrà inizializzato il database con le query SQL che troveremo nel file di inizializzazione. In particolare, diamo un'occhiata al file "init.sql" che contiene le query che verranno eseguite all'atto della creazione del DB:

```
create database "Grafana";
\c "Grafana"

drop schema if exists public cascade;
create schema public;

CREATE TABLE public.sensor1 (
    "time" NUMERIC NOT NULL,
    "A-series" double precision NOT NULL
);

CREATE TABLE public.sensor2 (
    "time" NUMERIC NOT NULL,
    "A-series" double precision NOT NULL
);

COPY public.sensor1("time", "A-series")
    FROM '/docker-entrypoint-
initdb.d/sensor_1.csv'
    WITH DELIMITER ',' CSV HEADER;

COPY public.sensor2("time", "A-series")
    FROM '/docker-entrypoint-
initdb.d/sensor_2.csv'
    WITH DELIMITER ',' CSV HEADER;
```

Figura 47. Script SQL per inizializzare il database

Come possiamo vedere, per ogni sensore abbiamo definito una tabella. Queste query SQL permettono di andare a prendere i dati csv che ci siamo scaricati da Grafana e metterli all'interno delle rispettive tabelle. Un singolo file .csv sarà composto da un Header corrisposto da due colonne: il momento in cui è stata fatta la misurazione e il valore. Una volta popolato il database con i dati dei sensori, quello che manca è connettere Grafana con questa sorgente per ricavare i dati. Allo stesso modo, per come abbiamo fatto per i Test Data, è possibile collegare Grafana con PostgreSQL

andando ad utilizzare la sua porta di default (5432). In particolare, una volta cercato la sorgente dati con la parola PostgreSQL, è possibile accedere alla sua pagina di configurazione:

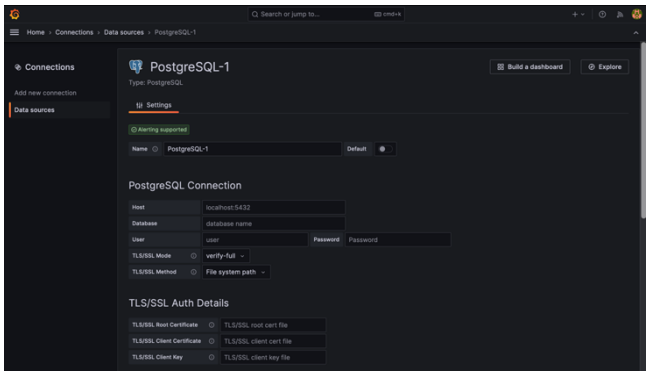

Figura 48. Pagina di configurazione di PostgreSQL

Dove ad host andremo ad inserire l'URL utilizzato per comunicare con il DBMS (nel nostro caso sarà "PostgreSQL:5432") mentre in "Database" metteremo Grafana (notiamo che nel file init.sql in figura 47 abbiamo creato il database di nome Grafana e inserito i dati lì dentro). Esistono poi le configurazioni legate all'utente che usa PostgreSQL, in particolare dal file di Docker compose abbiamo definito la password "password" e il nome usato per l'utente è lasciato di default. Fatto questo possiamo fare "Save & Test" e saremo sicuri che il nostro database funzionerà. Da qui in poi sarà possibile creare dei nuovi panel di visualizzazione basandoci sulla sorgente dati "PostgreSQL" selezionando, in base al sensore di interesse, la tabella.

Definizione del file di configurazione

Compresa la configurazione della sorgente dati, possiamo dare un'occhiata al file di configurazione di Grafana. Il nostro interesse è mettere Grafana dietro al reverse proxy in modo che sia accessibile mediante un path "/chart" all'interno del dominio. La prima parte del file "Grafana.ini" prende in considerazione questa esigenza, come possiamo notare dalla prima parte:

```
[server]
domain=ns01-prometeo.polito.it
root_url = %(protocol)s://%(domain)s:%(http_port)s/chart/
serve_from_sub_path = false
```

Figura 49. Sezione configurazione server in Grafana.ini

In particolare, prendiamo di riferimento il dominio "ns01-prometeo.polito.it". La particolarità, con le versioni successive di Grafana, è stato deprecato l'uso di "serve_from_sub_path" di cui non abbiamo bisogno ma si è deciso lo stesso di lasciare l'opzione a "false" nel caso in cui venga cambiata la sua versione di installazione. Nella

sezione successiva del file, ci siamo concentrati sull'autenticazione, in particolare come possiamo vedere dalla seguente figura:

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

[auth.proxy]
enabled = true
header_name = X-WEBAUTH-USER
header_property = username
auto_sign_up = true
sync_ttl = 60
enable_login_token = true
headers = Role:X-WEBAUTH-ROLE
```

Figura 50. Sezione configurazione autenticazione in Grafana.ini

Abbiamo disabilitato il form di login e il menu per effettuare il logout perché sono funzionalità totalmente legate all'applicazione Web. In questo caso, per l'autenticazione, ci siamo affidati al proxy. L'obiettivo è avere due Header: "X-WEBAUTH-USER" e "X-WEBAUTH-ROLE", la cui presenza assicura a Grafana che l'utente è autenticato. Infine, dato che vogliamo visualizzare i grafici sulla nostra applicazione React, dobbiamo abilitare l'esportazione in embeddings per cui aggiungeremo la seguente riga:

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

Figura 51. Sezione configurazione embeddings in Grafana.ini

Gestione degli iFrame

Analizziamo ora come vengono gestiti i grafici, in particolare cerchiamo di comprendere come visualizzare i grafici previa autenticazione. Normalmente in Grafana è possibile sempre visualizzare i grafici semplicemente esportandoli mediante l'opportuno menu presente nel panel come iFrame. Questi non sono altro che dei tag html che permettono di esportare immagini tipo png all'interno di una cornice. Il problema che si presenta però nell'esportare l'iframe è che normalmente non è possibile aggiungere ad essi Header aggiuntivi oppure da client React variare la richiesta, per cui ci dobbiamo affidare a NGINX il cui compito sarà:

1. Inserire degli Header X-WEBAUTH-USER e X-WEBAUTH-ROLE che faranno capire a Grafana che l'utente è autenticato

2. Gestire la validità del token e il suo refreshing

Dopo essersi autenticato l'utente avrà nel browser tre tipologie di cookie che ricordiamo essere user, role e user_info. Il server NGINX utilizzerà tali cookie ponendoli all'interno degli appositi Header, per quanto riguarda user e role, e utilizzerà user_info per fare validazione e refreshing. Andiamo ad analizzare la location definita per Grafana:

```
location /chart {

      rewrite ^/chart/(.*) /$1 break;
      auth_request /authServer;

      error_page 403 = @authorizationError;
      error_page 401 = @authenticationError;
      error_page 500 = @genericError;

      auth_request_set $new_cookie $sent_http_set_cookie;
      add_header Set-Cookie $new_cookie;

      include /etc/nginx/includes/grafana.conf;

      proxy_pass http://grafana:3000;
    }
```

Figura 52. Definizione della location per Grafana

Rispetto alle altre location viste precedentemente, questa appare diversa per varie ragioni. La prima cosa che andremo ad analizzare è la direttiva "auth_request" che, secondo la documentazione di NGINX, permette di definire una location non direttamente raggiungibile dall'esterno ma che permette di effettuare la chiamata a un server. Tale direttiva può ritornare due tipologie di stato: 401/403 oppure stati del tipo 2xx. Se tornasse 401 o 403, grazie alle direttive "error_page" sarà possibile fare il redirect verso la pagina di login. Superata la soglia del controllo del token, la direttiva "auth_request" può tornare un risultato per cui aggiorneremo i cookie (se ce ne sarà bisogno) e inseriremo i corrispettivi valori nell'Header. Cerchiamo ora di dare un occhiata più approfondita ai vari file e sezioni definite, iniziando proprio dalla location di /authServer usata dalla "auth_request". Come possiamo notare dalla figura 53,

l'idea è quella di chiamare un API del server che consenta di controllare i cookie e successivamente inizializzarli per mantenere attivo l'utente.

```
location = /authServer {
     internal;
     proxy_method GET;
     proxy_set_header X-Real-IP $remote_addr;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
     proxy_set_header X-Forwarded-Proto $scheme;
     proxy_set_header X-NginX-Proxy true;
     proxy_set_header Content-Type "application/x-www-form-urlencoded";
     proxy_set_header Cookie $http_cookie;
     proxy_pass http://server:3001/auth/check;
     proxy_set_header Cookie $http_cookie;
}
```

Figura 53. Sezione /authServer per il controllo del JWT

Precedentemente, si è scelto di non entrare nel particolare dell'analisi dell'API /auth/check del server per capirne meglio il funzionamento ora. Quando viene fatta richiesta al server di controllare il valore dei cookies, devono essere considerate diverse situazioni:

1. L'utente è autenticato e il token JWT è valido per cui non vi è necessità di aggiornare i cookies;
2. L'utente è autenticato ma il token JWT deve essere refreshato perché scaduto per cui dobbiamo aggiornare i cookies;
3. Il token JWT presentato non è valido perché inserito manualmente oppure non definito;

Sulla base di queste considerazioni è stata costruita la seguente API:

```
app.get("/auth/check", checkRole, async (req, res) => {
    const jsonFormatCookie = cookieParser.JSONCookies(req.cookies)
try {
            const tokenInformations = await
      client.introspect(jsonFormatCookie.user_info.token)

 if (tokenInformations.active) { res.status(200).end()}
 else if(jsonFormatCookie.user_info.authenticated && !tokenInformations.active){
      console.log("Refreshing...")
     const refreshing = await client.refresh(jsonFormatCookie.refresh_token)
     const new_user_info = {
               "token": refreshing.access_token,
               "refresh_token": refreshing.refresh_token,
               "id_token": refreshing.id_token,
               "authenticated": true
     }
      res
        .cookie('user_info', new_user_info)
        .status(200)
        .end()

    } else { res.status(401).end() }
} catch(e) { res.status(401).end() }})
```

Figura 54. API per il controllo del token JWT

Come possiamo notare, quello che viene fatto è ricavare i cookies in un formato JSON e, successivamente, viene chiamata l'API di Keycloak per il check della validità del token. Se questo fosse scaduto, verrebbe fatto il refresh diversamente significa che non è valido. Nel caso di refresh, rilasciamo la versione aggiornata di user_info e ciò motiva la scelta, nel server NGINX, di fare il forwarding dei cookies. Notiamo però la presenza di una middleware, in particolare di checkRole. La base su cui poggia la sua introduzione è la seguente: vogliamo garantire l'accesso alla dashboard con i suoi grafici ai soli admin. Quello che è segue è che se il ruolo dell'utente autenticato è Viewer oppure Editor non avrà i permessi per accedere a Grafana per cui ha senso lanciare un errore 403 (autenticato ma non autorizzato). Nella figura seguente possiamo dare un occhiata alla middleware:

```
const checkRole = (req, res, next) => {
    const jsonFormatCookie = cookieParser.JSONCookies(req.cookies)
    if (jsonFormatCookie.Role == "Editor"
            || jsonFormatCookie.Role == "Viewer") {
        res.status(403).end()
    } else {
        next()
    }
}
```

Figura 55. Middleware checkRole

Come sempre accediamo alle informazioni dell'utente mediante l'uso dei cookies. In sostanza, se l'errore è 403 l'utente viene rindirizzato verso "auth/login?redirect=/" e ciò significa controllare se è autenticato e rindirizzarlo verso il path base. Concludiamo questa parte analizzando il file "Grafana.conf" per capire le operazioni che vengono effettuate rispetto agli Header:

```
...
add_header X-WEBAUTH-USER "";
proxy_set_header X-WEBAUTH-USER $cookie_User;
add_header X-WEBAUTH-ROLE "";
proxy_set_header X-WEBAUTH-ROLE $cookie_Role;
```

Figura 56. File di configurazione per gli Header di Grafana

In sostanza, si tratta di aggiungere gli Header per far capire a Grafana che l'utente è autenticato ma ciò viene fatto solo dopo la validazione/refresh del token JWT per cui siamo sicuri che l'utente è autenticato (diversamente verrebbe rindirizzato verso la schermata di login).

Per questioni di comprensione, riassumiamo il processo di autenticazione dietro Grafana mediante il seguente grafico:

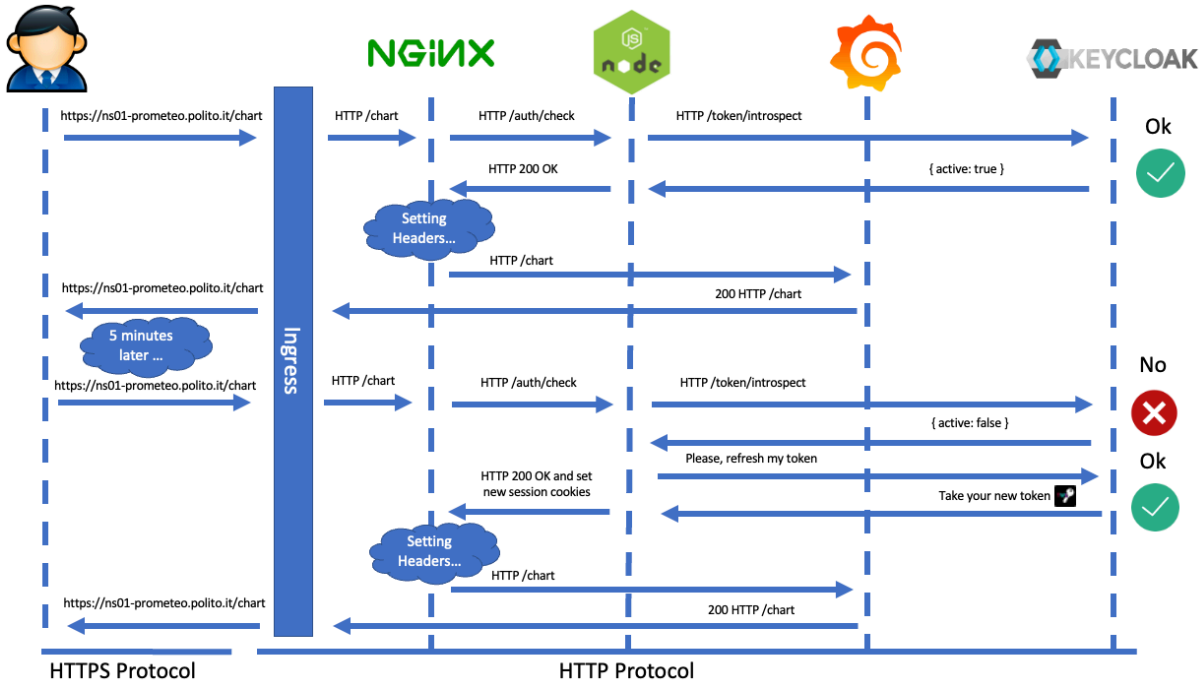

Figura 57. Validazione e Refresh del token per la visualizzazione dei grafici

Come possiamo vedere, il tempo di scadenza per il token è impostato a cinque minuti ma possiamo anche aumentarlo. Il funzionamento rispecchia i requisiti e obiettivi che ci siamo prefissati durante questo lavoro di Tesi.

Configurazione variabili in Grafana

Concludiamo il capitolo dedicato a Grafana andando ad analizzare la configurazione per le variabili. L'obiettivo è quello di effettuare delle query dinamiche grazie alle quali sia possibile fare il rendering di un grafico basandosi sul valore di una variabile, permettendo quindi di visualizzare dinamicamente, per esempio, il valore delle temperature misurate da un sensore anziché un altro. Per cui questo significa che tutti i grafici corrisposti da un URL del tipo "https://<domain>/chart/<some>&var-sensor=<sensorName>" dove: il "sensorName" corrisponde al nome del sensore di cui si vuole visualizzare i dati e "sensor" corrisponde alla variabile che abbiamo definito (Grafana aggiunge di default "var-<variableName>"). Per fare ciò dobbiamo innanzitutto definire una dashboard e accedere alle impostazioni. Tra i vari settings disponibile avremo la possibilità di selezionare "Variables" e cliccare "New Variable". Successivamente verrà data la possibilità di definire le proprietà della variabile e la sorgente dati corrispondente, in particolare la tipologia di variabile che ci interessa è

“Query” e potremmo così definire nella sezione “Query options”, la query SQL con cui useremo la nostra variabile (esempio, “Select * from sensor” dove “sensor” è il nome usato per la variabile). Fatto ciò, basterà creare un nuovo panel all’interno della dashboard e, successivamente, definire una query che usa il valore della variabile (esempio, “Select * from $sensor”). Riportiamo una configurazione del panel nella figura seguente:

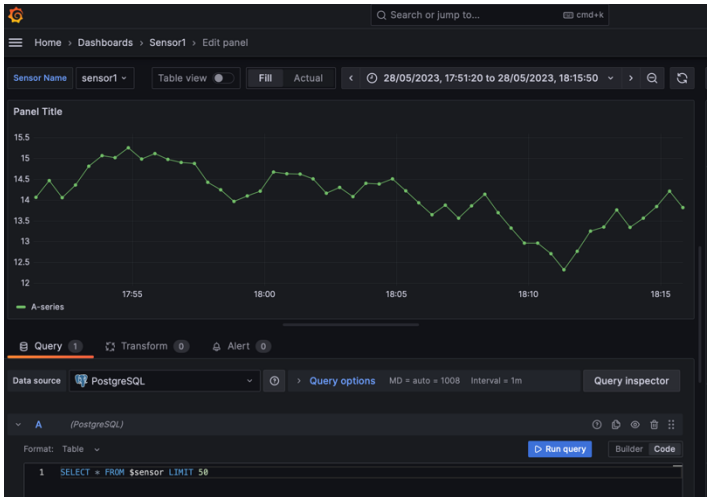

Figura 58. Custom Variable in Grafana

Una volta che viene creato il panel, sarà esportabile nel seguente modo: “https://<domain>/chart/<some>&var-sensor=<sensorName>” e potremo, lato client React attraverso l’uso di uno stato, pensare di cambiare sulla base della scelta dell’utente quali dati reperire dando la possibilità di scelta del sensore.

Registrazione dell'utente all'interno del dominio

Abbiamo analizzato fino ad ora come garantire l'autenticazione a un utente e, partendo da questa, garantire solo agli utenti autenticati e autorizzati la visualizzazione dei grafici. Come sancito dal capitolo 2, siamo interessati anche a garantire la registrazione dell'utente all'interno del dominio stesso e, in particolare, lo faremo mediante Keycloak da cui gestiremo tutto il processo di registrazione. In generale, durante la registrazione, possiamo richiedere la verifica del dominio della posta elettronica e della conferma della e-mail. Analizzeremo prima come Keycloak fornisce una risposta a questi due requisiti e successivamente analizzeremo come attivare e cosa succede durante il processo di registrazione.

Validazione dell'e-mail

In Keycloak, è possibile accedere al menu dedicato alla registrazione e, in particolare, all'insieme delle informazioni che vengono messe a disposizione. Per accedere alle informazioni presenti nel form di registrazione e aggiungere la validazione bisogna:

1. Accedere al Realm e andare sul tab "User Profile" di "Realm Settings"

2. Cliccare la proprietà che ci interessa, nel nostro caso sarà e-mail

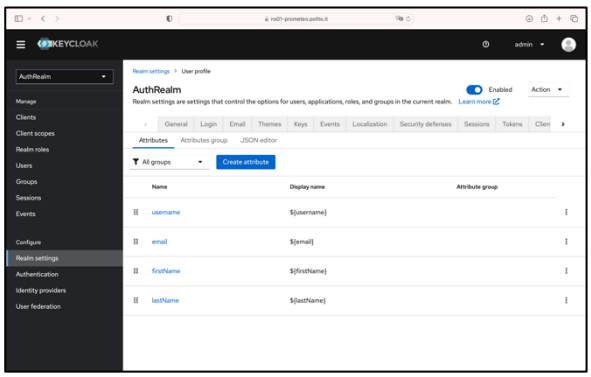

Figura 60. Schermata User Profile

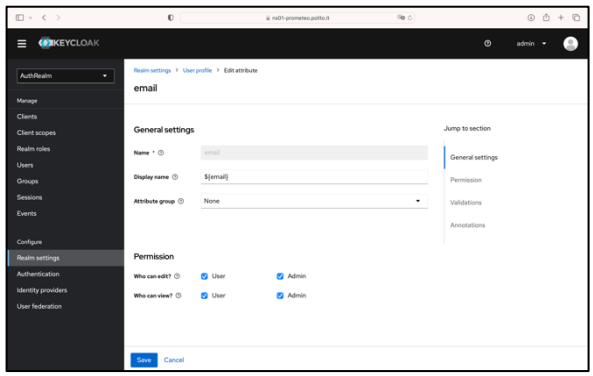

Figura 59. Schermata User Profile - E-mail

3. Andare nella sezione dedicata ai validators cliccando "Add Validator"

4. Siamo poi chiamati a scegliere la tipologia di validatore, nel nostro caso sceglieremo “pattern” perché ci permetterà di usare per la validazione le regex messe a disposizione di Java (nel nostro caso, per il dominio ns01-prometeo.polito.it inseriremo come pattern .+ns01-prometeo.polito.it)

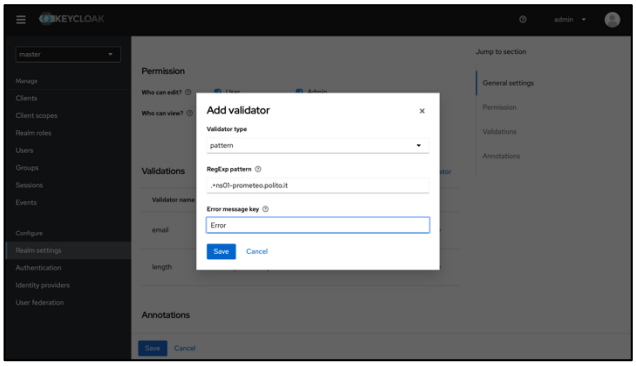

Figura 61. Schermata selezione pattern

5. Opzionalmente, è possibile anche inserire un messaggio di errore nel caso in cui l’utente inserisce una stringa che non sia coerente con il pattern di validazione

Si presuppone che venga attivato in Keycloak l’opzione per la gestione dei profili utente mediante la variabile di ambiente “KC_FEATURES” e il relativo menu di “User Profile” per il realm in questione.

Verifica dell’e-mail

Per quanto riguarda invece l’invio dell’e-mail di conferma, Keycloak di default prevede l’invio della e-mail per un determinato utente e, senza la conferma, non è possibile utilizzare l’account appena creato. Tutte le e-mail dovranno essere uguali al nostro dominio e si può configurare per Keycloak un server mail. Per l’invio ci si è affidati a un server esterno alla nostra rete messo a disposizione per tutti che si chiama “smtp-relay.sendinblue.com” che lavora sulla porta 587. Una volta registrato al sito, è possibile usufruire gratuitamente del servizio ottenendo delle credenziali associate ma che in Keycloak non useremo. Per configurare il server di posta da usare, si può accedere alle configurazioni del realm in questione e accedere alla sezione e-mail.

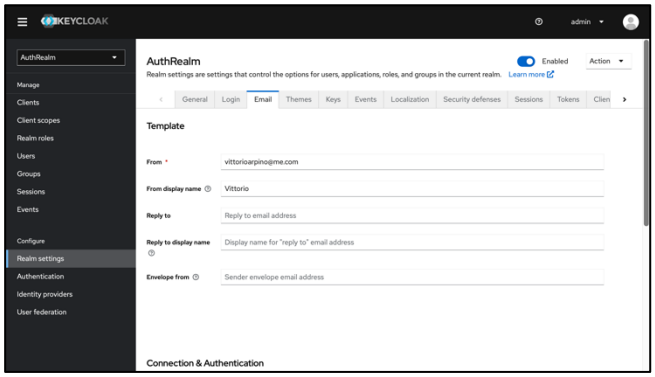

Figura 62. Schermata configurazione server mail 1

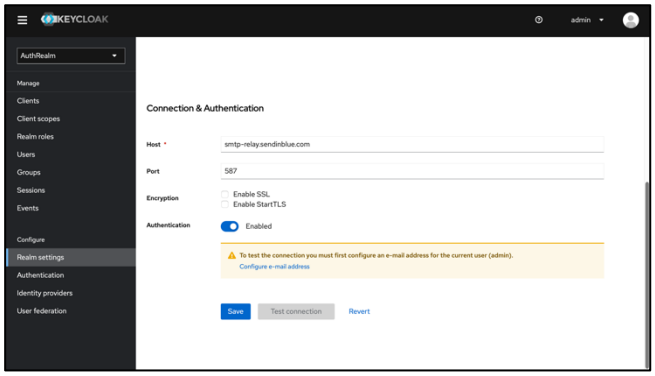

Figura 63. Schermata configurazione server mail 2

Come possiamo vedere dalle figure 62 e 63, è possibile in una prima sezione indicare chi è che ha effettuato l’invio della e-mail (per esempio si potrebbe definire una mail per l’amministratore) e il nome da mostrare all’invio. Successivamente è possibile configurare l’host e la porta su cui lavorare che sono informazioni date da “brevo”. Interessante, inoltre, è la possibilità di testare l’invio della e-mail configurando una e-mail di amministratore con lo scopo di verificare che tutto funzioni. Al momento non abbiamo effettuato una configurazione del messaggio inviato, per cui useremo quello di default che conterrà un link che rimanderà al nostro dominio e che ci permetterà di attivare l’account su Keycloak.

Estensioni di Keycloak

Come abbiamo già detto, Keycloak è una piattaforma per la gestione delle identità e degli accessi offrendo una vasta gamma di funzionalità pronte all'uso in ambito di sicurezza. Tuttavia, la flessibilità di Keycloak non si limita solo a quelle funzionalità predefinite. Viene data la possibilità agli sviluppatori di ampliare e personalizzare ulteriormente il comportamento del sistema mediante la progettazione di estensioni. Queste rappresentano moduli aggiuntivi sviluppate da terze parti o personalizzati internamente che offrono funzionalità extra o comportamenti specifici per le esigenze di un'applicazione o di un ambiente particolare. Le estensioni permettono di adattare Keycloak ai requisiti specifici del nostro progetto, fornendo soluzioni diverse su misura per scenari di autenticazione, autorizzazione e gestione delle identità.
Da un punto di vista dei nostri obiettivi, potrebbe essere interessante: customizzare i temi per il Login/Registrazione e permettere la registrazione solo per determinate e-mails predefinite anticipatamente. Per il primo obiettivo introdurremmo "Keycloakify" con il suo funzionamento mentre per il secondo obiettivo definiremo direttamente un progetto JAVA utilizzando le librerie messe a disposizione da Keycloak.

Keycloakify

Normalmente per definire un tema di Keycloak, vi è necessaria la definizione di file di tipo ".ftl". Questi file vanno inseriti all'interno della cartella chiamata "themes" e, in base alla pagina che vogliamo implementare, diamo un nome (per esempio nel caso del login il file si chiamerà "login.ftl") (Keycloak). La difficoltà principale però è lo sviluppo di file con estensione ".ftl", infatti ricordiamo che sono modelli FreeMarker, un motore di modellazione per la creazione di pagine web, documenti testuali e altro (FREEMARKER). Questo implica che bisogna imparare la sintassi di FreeMarker con tutte le direttive ed espressioni con una curva di apprendimento molto alta fin da subito. Tra l'altro ci sono anche ulteriori rischi come: mescolare logica di business e presentazione rendendo il codice difficile da leggere, avere template troppo complessi con l'introduzione di code smells ecc. La soluzione a tutto questo si chiama Keycloakify (Keycloakify) che è un progetto open source in React che fornisce la possibilità di introdurre pagine scritte in React per il login/registrazione come un eseguibile "jar"

generato tramite un comando. Cerchiamo di capire a grandi linee il funzionamento del progetto open source. Come possiamo vedere dalla figura seguente:

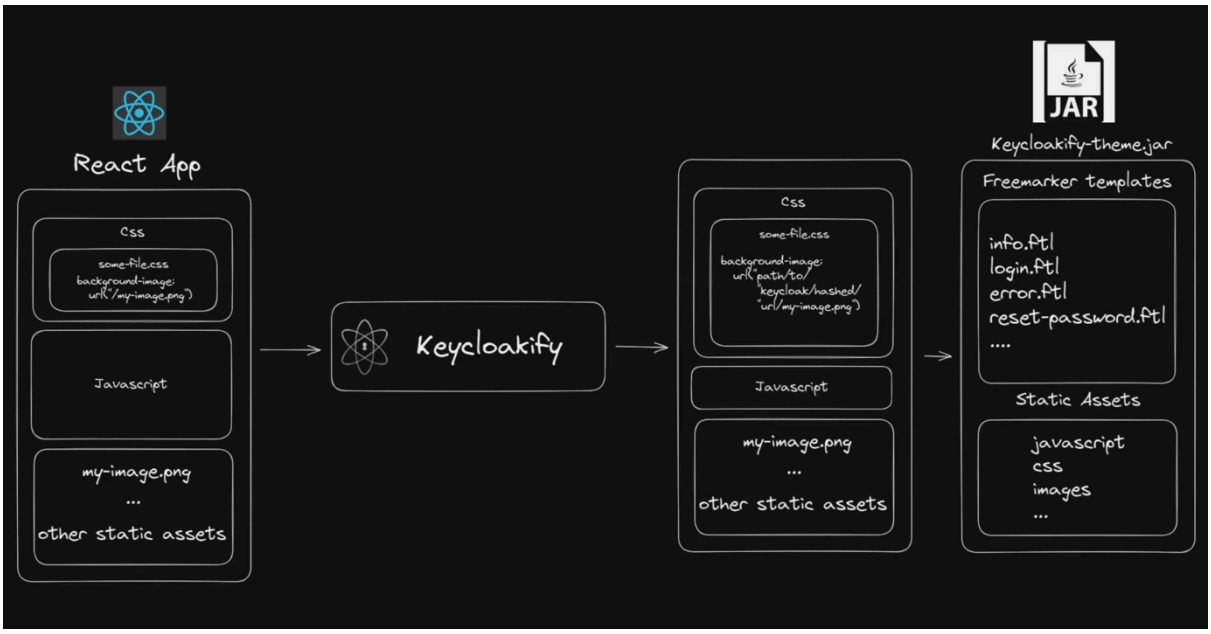

Figura 64. Keycloakify all'interno del processo di building

L'idea è partire da un'applicazione React e utilizzare Keycloakify per ottenere dei file statici che abbiano un URL associato ai temi chiamati da Keycloak (ricordiamo che Keycloak recupera i temi utilizzando URL). Successivamente, tramite comando da terminale è possibile generare un JAR eseguibile da Keycloak che contiene vari temi e che possono essere settati dalla dashboard di Admin. La customizzazione non è un obiettivo primario della Tesi per cui non è stato fatto un grande cambiamento dei temi ma semplicemente si sono adattati i colori all'applicazione Web di appoggio.

Figura 66. Nuova schermata di login

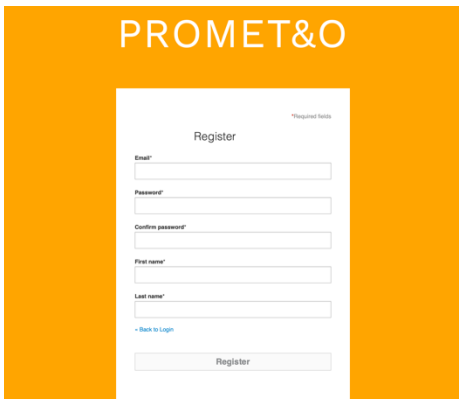

Figura 65. Nuova schermata di registrazione

In particolare, quello che è stato fatto è partire dal progetto base di Keycloakify che è possibile trovare facilmente su GitHub (https://github.com/codegouvfr/Keycloakify-

starter). Partendo da questo, è stato cambiato il background della pagina e poi è stato lanciato il comando “yarn build-Keycloak-theme” che crea il file JAR per Keycloak.

Validazione delle e-mails

Come abbiamo già detto, Keycloak offre la possibilità di inserire la validazione solo per il dominio delle e-mails attraverso l’uso di validators. La soluzione offerta è molto debilitante perché non ci consente di creare una mail whitelisting ovvero una di emails che è possibile usare in fase di registrazione. Per fare ciò è stata quindi scritta un estensione in JAVA utilizzando le librerie del tipo “org.Keycloak.*” e librerie di JAVA. Si è partiti con il definire una classe astratta “RegistrationProfileDomainValidation” che è possibile estendere fornendo una validazione per gli indirizzi e-mail. Tale classe ha due proprietà statiche:

- domainListConfigName: rappresenta una lista che mantiene la lista dei domini a cui è consentita la registrazione;

- DEFAULT_DOMAIN_LIST: mantiene la lista dei domini di default a cui viene fatta la validazione

- DOMAIN_LIST_SEPARATOR: mantiene il valore di un separatore che servirà per separare le e-mail all’interno del form di Keycloak

Dato che stiamo lavorando sul profilo di registrazione degli utenti, la nostra classe astratta dovrà estendere RegistrationProfile che definisce in modo astratto campi che vengono visualizzati agli utenti quando si registrano in Keycloak e le regole di convalida che devono essere soddisfatte. Le classi che estendono RegistrationProfile implementano una logica per convalidare i dati forniti dagli utenti. Ad esempio, si potrebbe implementare la nostra logica whitelisting oppure una logica blacklisting (blocco una serie di e-mail per la registrazione). Inoltre, partendo da RegistrationProfile, dobbiamo implementare la logica di validazione e lo possiamo fare andando a fare l’override del metodo “validate()” e il suo compito è controllare la validazione ritornando false e un messaggio di errore oppure true, nel caso in cui la validazione vada a buon fine. Dato che vogliamo avere la possibilità di estendere più volte la classe, in modo da garantire più tipologie di checking future da sviluppare in maniera semplice, definiamo un metodo astratto “isEmailValid()” che verrà implementato dalle sottoclassi e utilizzato nel metodo “validate()”.

Lasciamo quindi un estratto listato del codice che è stato sviluppato:

```
public abstract class RegistrationProfileDomainValidation extends
RegistrationProfile {

   protected static String domainListConfigName;
   protected static final String DEFAULT_DOMAIN_LIST = "example@example.org";
   protected static final String DOMAIN_LIST_SEPARATOR = "##";

   @Override
    public boolean isConfigurable() {
        return true;
   }

   @Override
   public void validate(ValidationContext context) {
      MultivaluedMap<String, String> formData =
context.getHttpRequest().getDecodedFormParameters();

      List<FormMessage> errors = new ArrayList<>();
      String email = formData.getFirst(Validation.FIELD_EMAIL);
      AuthenticatorConfigModel mailDomainConfig =
context.getAuthenticatorConfig();
      String eventError = Errors.INVALID_REGISTRATION;
      if(email == null){
         context.getEvent().detail(Details.EMAIL, email);
         errors.add(new FormMessage(RegistrationPage.FIELD_EMAIL,
Messages.INVALID_EMAIL));
         context.error(eventError);
         context.validationError(formData, errors);
         return;
      }
      String[] domainList =
mailDomainConfig.getConfig().getOrDefault(domainListConfigName,
DEFAULT_DOMAIN_LIST).split(DOMAIN_LIST_SEPARATOR);
      boolean emailDomainValid = isEmailValid(email, domainList);
      if (!emailDomainValid) {
         context.getEvent().detail(Details.EMAIL, email);
         errors.add(new FormMessage(RegistrationPage.FIELD_EMAIL,
Messages.INVALID_EMAIL));
      }
      if (errors.size() > 0) {
         context.error(eventError);
         context.validationError(formData, errors);
      } else {
         context.success();
      }
   }

   public abstract boolean isEmailValid(String email, String[] domains);
}
```

Figura 67. Classe RegistrationProfileValidation

Come possiamo vedere, otteniamo i parametri mediante una MultivaluedMap dal form di registrazione e ricavo tutte le informazioni mediante il riferimento con nome del campo di registrazione. Alla fine del codice, troviamo la chiamata al metodo

isEmailValid() che, come abbiamo già detto, lasciamo astratto per dare la possibilità di implementare tecniche di validazione differenti in maniera semplice. Successivamente è stata definita la classe per la validazione delle emails "RegistrationProfileWithMailDomainCheck" che implementa diversi metodi tra cui:

- getDisplayType(): restituisce il nome del provider di validazione
- getId(): restituisce un identificativo del provider di validazione
- getHelpText(): restituisce la descrizione del provider di validazione
- getConfigProperties(): restituisce l'elenco delle proprietà di configurazione del provider di validazione
- buildPage(): costruisce la pagina di registrazione con i campi necessari per la validazione
- isEmailValid(): verifica se l'indirizzo e-mail fornito è valido

In particolare, le proprietà che sono state definite all'interno del file sono statiche:

```
private static final List<ProviderConfigProperty> CONFIG_PROPERTIES=new
ArrayList<>();

   static {
      domainListConfigName = "validEmails";
      ProviderConfigProperty property;
      property = new ProviderConfigProperty();
      property.setName(domainListConfigName);
      property.setLabel("Valid emails");
      property.setType(ProviderConfigProperty.STRING_TYPE);
      property.setHelpText("List mail authorized to register, separated
by '##'");
      CONFIG_PROPERTIES.add(property);
   }
```

Figura 68. Proprietá definite nella classe RegistrationProfileWithMailDomainCheck

e vengono ottenute con il metodo getConfigProperties() di cui è fatto l'override grazie alla classe RegistrationProfile. Come già detto, implementiamo il metodo isEmailValid nel seguente modo:

```
@Override
   public boolean isEmailValid(String email, String[] emails) {
      for (String e : emails) {
         if (email.equalsIgnoreCase(e)) {
            return true;
         }
      }
      return false;
   }
```

Figura 69. Validazione dell'e-mail

Ottenendo le emails tramite form separate dallo splitter “##” e facendo poi il controllo con la e-mail ottenuta dal form di registrazione.

Installazione delle estensioni

Per avere le estensioni su Keycloak basta inserire i file JAR generati all’interno della cartella “opt/Keycloak/providers” ed è per questo motivo che abbiamo mappato la cartella in locale come “/Keycloak/plugins” come si evince dal Docker compose file nella figura 23. Una volta inseriti nella cartella, facendo partire Keycloak verranno caricati i JAR e saranno configurabili dalla dashboard di admin. Per quanto riguarda i temi custom creati con Keycloakify, basterà entrare nel menu del Realm e selezionare i temi in base a ciò che abbiamo sviluppato. Riportiamo la schermata di configurazione dei temi per Keycloak:

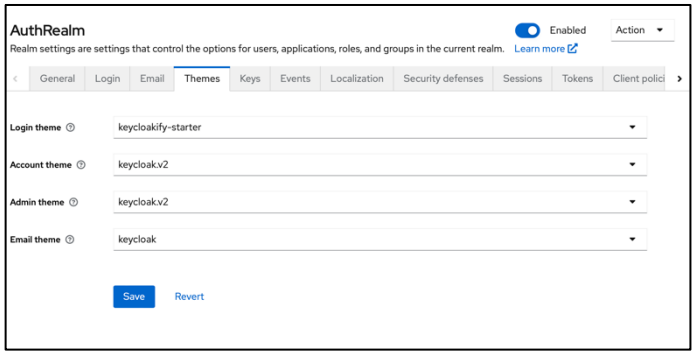

Figura 70. Configurazione Temi Keycloak

Come possiamo vedere, per il tema di login possiamo selezionare il tema “Keycloakify-starter”. Più complessa è la configurazione per la validazione delle emails, in particolare bisogna accedere al menu Authentication e duplicare, inizialmente, il flow di registrazione. Successivamente sarà possibile aggiungere step aggiuntivi:

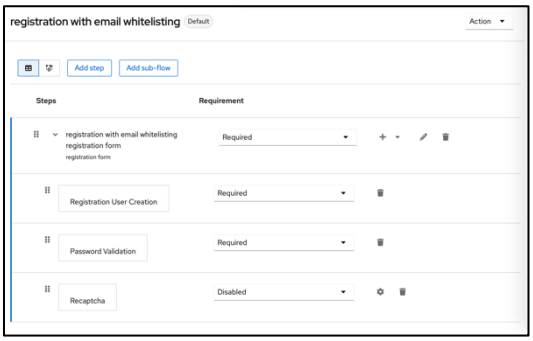

Figura 72. Duplicazione Registration Flow

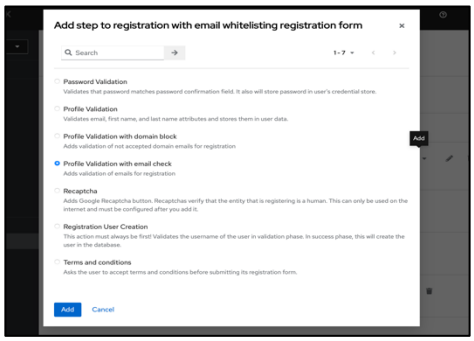

Figura 71. Aggiunta Step

Il nome dato allo step è “Profile Validation with Email Check” ma è comunque possibile cambiarlo all’interno del progetto JAVA. Successivamente abbiamo spostato lo step più in alto, in modo che venga fatto anticipatamente rispetto agli altri step e abbiamo configurato le emails valide del dominio mediante le impostazioni dello step.

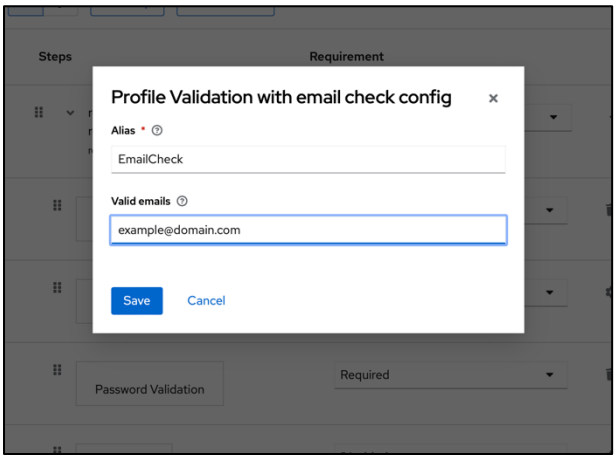

Figura 73. Configurazione della whitelisting

Processo di Registrazione

Analizziamo ora come avviare il processo di registrazione per un determinato utente. La prima cosa da fare è accedere all’applicazione client e cliccare sul pulsante di login per cui apriremo il relativo form per autenticarsi. Al di sotto di tale form, è possibile notare “New user? Register” e cliccandoci su, verremo rindirizzati alla seguente pagina:

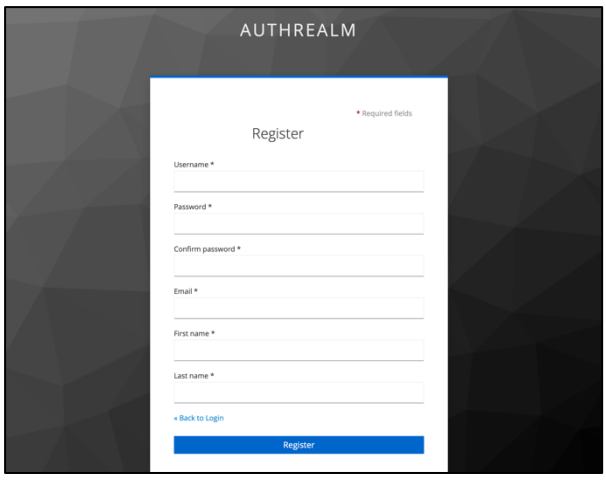

Figura 74. Form di registrazione

Una volta inserite tutte le informazioni e cliccato su “Register”, Keycloak, a meno di errori di validazione sulla mail, ci rimanderà a un ulteriore pagina che ci dirà che il link

di validazione è stato mandato e la validità di tale link al momento è di cinque minuti ma è possibile aumentarlo. Una volta inviata la mail sarà possibile andare nella propria casella di posta elettronica per la verifica e, cliccando sul link, verremmo rindirizzati alla root dell'applicazione già autenticati e con il profilo verificato.

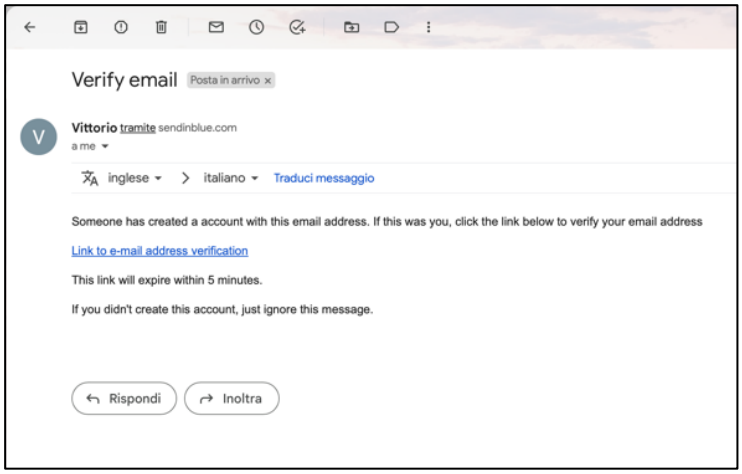

Figura 75. Esempio di mail di verifica

In conclusione, riportiamo uno schema del funzionamento del processo di registrazione:

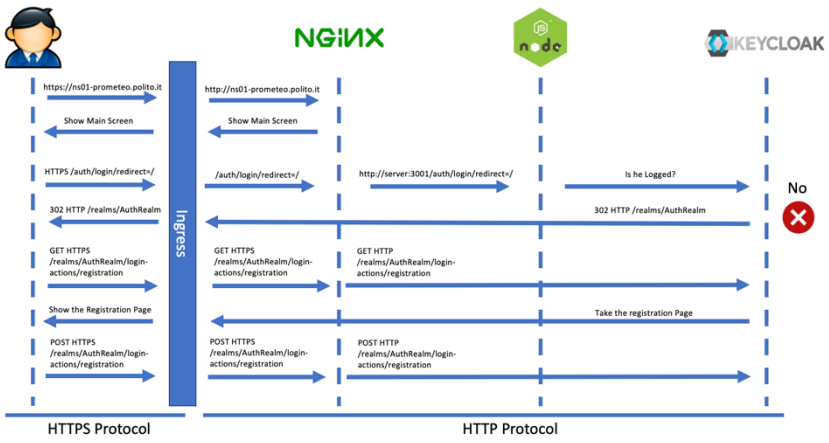

Figura 76. Schema di registrazione per ns01-prometeo.polito.it

Realizzazione dell'Architettura completa

Come già accennato, abbiamo posto i nostri obiettivi in un ambiente multi-dominio in cui ci sono più sottoreti che vengono indirizzate grazie all'ausilio di un server di Ingress. In particolare, rispetto al Docker compose file che abbiamo definito, non è stata definita alcuna porta da esporre questo perché non siamo interessati ad esporre dalla macchina delle porte interne del dominio, in modo che tutti i sottodomini non vadano in collisione tra loro. L'idea è stata quella quindi di costruire più sottoreti con il file di Docker compose presentato precedentemente ed utilizzare i nomi delle sottoreti esposte all'interno di un file NGINX che funge da server di Ingress, ne riportiamo il contenuto di una sezione:

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

        server_name ns01-prometeo.polito.it;
        resolver 127.0.0.53;

        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 ns01-prometeo.polito.it;
        ssl_certificate /etc/ssl/certs/prometeo_polito_it.pem;
        ssl_certificate_key /etc/ssl/certs/prometeo_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://ns01-app-1;
        }

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

Figura 77. Configurazione server Ingress per ns01

Come possiamo notare dal file di configurazione, abbiamo contraddistinto diverse sezioni per server diversi mediante commenti diversi per una facile lettura. Al momento prendiamo di riferimento la configurazione per la rete ns01-prometeo.polito.it e qualsiasi richiesta che viene indirizzata verso http viene riscritta in https, in particolare abbiamo una mappatura dei certificati utilizzati per essere terminatore SSL e del file di configurazione. Tutto ciò rende ogni dominio non direttamente accessibile ma per accedervi dobbiamo passare sempre per il server Ingress e qualsiasi richiesta, in base alla sottorete, raggiungerà il server NGINX del dominio in questione (in questo caso, ns01-app-1 corrisponde al server NGINX del dominio ns01-prometeo.polito.it). Abbiamo detto che ogni Docker compose crea una propria rete. Questo significa che gli elementi all'interno di un Docker compose sono accessibili solo tra loro. Se vogliamo collegare la rete di Ingress con le sottoreti del dominio, dobbiamo utilizzare una soluzione offerta da Docker. Questa soluzione ci consente di collegare più reti tra loro, formando un'unica rete. In questo modo, sarà possibile fare riferimento ai container presenti in altre sottoreti. Per comprensibilità riportiamo il file di Docker compose del server di Ingress:

```
version: '3.5'
services:
  ingress:
    image: ingress_in_keycloack
    build:
      context: ./
      dockerfile: ./Dockerfile
    healthcheck:
      test: curl --fail http://ns01-app-1; && curl --fail  http://ns02-app-1
      interval: 1m30s
      timeout: 10s
      retries: 10
      start_period: 40s
    networks:
      - ns01
      - ns02
    ports:
      - 80:80
      - 443:443
    restart: unless-stopped

networks:
  ns01:
    name: ns01_default
    external: true
  ns02:
    name: ns02_default
    external: true
```

Figura 78. Docker compose file del server Ingress

Come possiamo notare dal file, ai fini di sviluppo, abbiamo definito tre sottoreti che corrispondono a ns01 e ns02 e, inoltre, abbiamo esposto le porte 80 e 443 dove

qualsiasi richiesta con http sulla porta 80 verrà rindirizzata su https verso la porta 443. Per collegare il server con quello di Ingress, bisogna definire le reti con “networks” associandovi il nome indicando che è esterna. Inoltre, al fine di far partire il server Ingress solo quando tutte le sottoreti sono disponibili, è stato fatto un “health check” sui server NGINX delle sottoreti e, finché non saranno attivi tutti, verrà fatto un continuo restart del server Ingress. Per quanto riguarda l’avvio di tutti i file di Docker compose sarà possibile utilizzare il comando “Docker compose up --build” in maniera indifferente da ogni cartella ma l’importante che vengano create le reti ns01 e ns02 prima di chiamare il comando per il server di Ingress dato che ci sarà un controllo sulla presenza delle reti.

Conclusioni

Nel corso di questa Tesi, si è studiato e approfondito diverse tipologie di soluzione con lo scopo di raggiungere gli obiettivi che sono stati prefissati e realizzando un prototipo della soluzione per visualizzare grafici previa autenticazione. In particolare, sono state analizzate diverse soluzioni e metodologie durante lo sviluppo ma alla fine si è scelto di utilizzare tecnologie che sono ampiamente utilizzate con lo scopo di rendere il lavoro rivedibile e migliorabile in un prossimo futuro. Per quanto riguarda il processo di autenticazione si è cercato di dare peso a delle soluzioni che si basassero sul semplice uso del token JWT che potrebbe essere fatto oppure si potrebbe pensare di usare direttamente la sessione. Questa differenziazione ha permesso di rendere le chiamate ai diversi servizi disponibili più veloci a seconda delle esigenze (per esempio, non è possibile usare la sessione per Grafana e visualizzare il grafico ma bisogna utilizzare soluzioni meno pesanti come il JWT). L’uso di Grafana ha certamente imposto delle limitazioni sullo sviluppo dato che l’intero processo di autenticazione e autorizzazione si è dovuto basare su di esso ma, come detto precedentemente, la soluzione è molto diffusa non solo nel caso del monitoring delle temperature tramite sensori ma anche nel caso, per esempio, del monitoring di un server. Un ulteriore sviluppo che potrebbe essere promosso nella nostra soluzione è l’inclusione di un server mail interno al dominio che permetta di facilitare l’invio della mail e che sia completamente configurabile all’interno del dominio stesso dando più affidabilità. Su questa soluzione ci sono però degli svantaggi come la necessità di rendere il server mail affidabile da un punto di vista esterno e ciò richiede tempo. Una futura revisione della soluzione potrebbe riguardare, inoltre, la predisposizione di una pagina ad-hoc

sia per la registrazione che per il login per andare poi ad utilizzare direttamente le API di Keycloak. Ciò renderebbe la soluzione più personalizzabile ed è effettivamente possibile se si lavora con la configurazione di Keycloak. Il lavoro di Tesi ha posto la base per ulteriori sviluppi che devono proseguire con lo scopo di migliorare l'affidabilità della soluzione da un punto di vista della sicurezza e della semplicità di uso.

Bibliografia

Auth0. *Introduction to JSON Web Tokens*. Tratto da https://jwt.io/introduction

ENTRUST. Tratto da https://www.entrust.com/it/resources/faq/what-is-oauth

FREEMARKER. *What is Apache Freemarker?* Tratto da https://freemarker.apache.org

Keycloak. *Server Developer*. Tratto da https://www.Keycloak.org/docs/latest/server_development/#_themes

Keycloakify. Tratto da https://docs.Keycloakify.dev

LINFO. (2006, Gennaio 29). *Authentication Definition*. Tratto da https://www.linfo.org/authentication.html

Open Identity Platform. *Stateful vs Stateless Authentication*. Tratto da https://www.openidentityplatform.org/blog/stateless-vs-stateful-authentication