POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Politecnico
di Torino

Tesi di Laurea Magistrale

Progetto e implementazione di un'architettura server a supporto della gestione logistica dei processi produttivi

Relatore	Candidato
Prof. Giovanni MALNATI	Emanuele MUSCATELLO

Luglio 2021

Sommario

Con l'avvento dell'Industria 4.0, la gestione operativa all'interno del settore industriale ha subito un cambiamento radicale, andando sempre di più verso la digitalizzazione dei processi industriali. Tra i vari aspetti di questa innovazione grande importanza è rivestita dal concetto di *Industrial IoT*, che pone alle proprie basi l'utilizzo, all'interno delle aziende, di dispositivi smart in grado di connettersi ad una rete e di acquisire una controparte digitale grazie alla quale poter favorire l'interconnessione e la cooperazione dei dispositivi e agevolare la generazione e l'elaborazione di dati utili al miglioramento dell'efficienza dei processi produttivi. L'impiego di questi dispositivi all'interno di un'azienda rende possibile, tra le altre cose, la realizzazione di quella che in questo contesto viene chiamata *Smart Logistics*, ovvero la gestione del monitoraggio di una filiera produttiva industriale tramite l'utilizzo di tag RFID, codici a barre o qualunque altra tecnologia di identificazione. In questo caso i dispositivi leggono questi identificativi e producono dati di vario tipo sotto forma di eventi da spedire ad un server centrale. Dalla raccolta di questi dati nasce quindi l'esigenza di fornire, per ogni evento, dei comandi di risposta opportuni che possono essere sia azioni di controllo automatico da gestire localmente da opportuni dispositivi interni all'azienda, sia azioni legate a scelte tattico-strategiche che nascono da osservazioni a 360° su tutte le informazioni disponibili all'azienda e che arrivano "da fuori" o "dai piani alti".

L'obiettivo di questa tesi è quello di creare un supporto a livello server per l'autenticazione, l'amministrazione e il monitoraggio dei dispositivi sopracitati nel contesto di XTap, una piattaforma per il monitoraggio della supply chain, comprendendo anche la gestione a livello Docker dei servizi che compongono XTap e il loro deployment su Kubernetes. Nel corso dell'elaborato verranno spiegati nel dettaglio, dopo una prima parte introduttiva in cui verranno esposte alcune informazioni sul background del lavoro effettuato e sugli strumenti utilizzati durante lo sviluppo della soluzione, l'architettura che consente ai dispositivi di autenticarsi e di operare presso la piattaforma e i meccanismi che consentono il monitoraggio del comportamento di questi, inserendo anche delle brevi descrizioni riguardanti le modifiche effettuate ad XTap nell'ambito dell'avvio delle immagini Docker e della migrazione dei servizi verso

il GKE (Google Kubernetes Engine). A seguire verrà mostrato come i servizi aggiunti per favorire quanto descritto sono stati effettivamente programmati e quali scelte implementative sono state prese in fase di sviluppo, dando infine una valutazione finale del lavoro svolto ed elencando i possibili sviluppi futuri.

Indice

Elenco delle figure

Capitolo 1

Background

Di seguito verranno riportate le nozioni necessarie per comprendere il contesto in cui il lavoro della tesi è stato svolto. Verranno discussi i principi dietro ai concetti di Industria 4.0 e di Industrial IoT, su cui l'intera tesi è basata, focalizzandoci sulle migliorie da questi apportate. Successivamente verrà descritto lo standard EPCIS, che permette il monitoraggio dei prodotti di una filiera industriale e che viene utilizzato per la cattura e l'elaborazione degli eventi da XTap, un progetto per la creazione di una piattaforma che offre alle aziende degli strumenti che sfruttano questo standard e che descriveremo in questa sezione. Concluderemo infine con la descrizione dei dispositivi, spiegando cosa sono e come dovrebbero funzionare.

1.1 Industria 4.0 e Industrial IoT

Il processo di innovazione tecnologica ha toccato molti settori tra i quali vi è quello industriale, che sta vivendo una vera e propria rivoluzione, la quarta nella sua storia, che proprio per questo motivo prende il nome di *Industria 4.0* [1]. Le fabbriche di oggi infatti si dirigono sempre di più verso l'automazione dei processi industriali attraverso l'utilizzo di sistemi ciberfisici, cioè di macchine intelligenti che fungono da collegamento tra il mondo fisico e quello digitale e che consentono lo scambio di informazioni tra tutti gli elementi che compongono il processo produttivo, sia all'interno che all'esterno di un'azienda. Per fare in modo che queste macchine possano avere questo comportamento è stato introdotto il concetto di Internet of Things all'interno delle industrie, dando vita all'Industrial IoT (IIoT). L'IIoT rende possibile l'inserimento, all'interno dei processi industriali, di dispositivi fisici che presentano una controparte digitale in grado di operare all'interno di una rete. Questi dispositivi sono rappresentati da sensori, associati a macchine o ad elementi

infrastrutturali di un'industria, che sono in grado di raccogliere dati da spedire verso un'entità centrale in grado di processarli. Il processamento di questi dati permette di eseguire operazioni che fino a qualche tempo fa risultavano impossibili, come ad esempio effettuare in modo automatico diagnostica e risoluzione dei problemi, adattare il comportamento dei macchinari di produzione a real-time, simulare i processi produttivi per capirne problemi e vantaggi, seguire le varie fasi di sviluppo di un prodotto, sfruttare le potenzialità della realtà aumentata oppure effettuare la gestione dei dati aziendali su cloud. Tutte queste operazioni e altre ancora possono essere riunite in tre gruppi che rappresentano gli ambiti di utilizzo dell'IIoT, ovvero:

- *Smart Factory* - Tecnologie che coordinano tutti gli elementi di un processo produttivo.
- *Smart Lifecycle* - Elementi che gestiscono il ciclo di vita di un prodotto.
- *Smart Logistics* - Monitoraggio dei prodotti attraverso identificativi (tag RFID, codici a barre, codici QR, ecc.) [2].

1.1.1 Smart Logistics

All'interno di un'industria, la logistica riveste grandissima importanza ma allo stesso tempo può rappresentare una fonte di problemi la cui risoluzione può essere molto complessa. Una cattiva gestione della logistica può portare ad esempio ad avere un'organizzazione delle merci poco ottimizzata, causare errori nella movimentazione della merce o ritardare le spedizioni. Oltre a ciò, la logistica tradizionale non dispone solitamente di sistemi che garantiscono la tracciabilità dei prodotti in tempo reale lungo il loro percorso all'interno della filiera. Tutti questi problemi influenzano negativamente le aziende, causando perdita di ore di produttività e, di conseguenza, di profitti. La Smart Logistics offre una soluzione a tutto ciò cercando di automatizzare e semplificare tutte le procedure che compongono le operazioni di logistica. I prodotti ad esempio possono essere associati ad un tag RFID o ad un codice identificativo di natura simile in modo tale che, disponendo di un dispositivo smart con un lettore RFID integrato, lo stato del prodotto possa essere notificato, per ogni fase attraversata dal prodotto stesso all'interno della filiera, ad un server centrale in grado di elaborare le informazioni ricevute. Queste informazioni, il cui contenuto può essere standardizzato come ad esempio per EPCIS, possono essere utilizzate per monitorare il prodotto ed effettuare analisi riguardanti le varie fasi che questo attraversa durante il processo di produzione. La Smart Logistics tuttavia non si occupa soltanto del monitoraggio di un prodotto, ma anche di altri aspetti come ad

esempio il controllo remoto delle saracinesche di un sito di produzione per l'accesso ai camion, lo stoccaggio dei prodotti per mezzo di trasloelevatori automatici o il tracking dei mezzi di trasporto delle merci attraverso sistemi GPS [3].

1.2 EPCIS

EPCIS [4] è uno standard creato da GS1, un ente per lo sviluppo di standard per la comunicazione tra imprese, che consente di tracciare un prodotto all'interno di una filiera seguendolo durante tutte le fasi, da quella di produzione a quella di vendita, e notificando il suo stato in tempo reale attraverso la generazione di eventi. Lo standard include sia la definizione del data model da usare per la costruzione degli eventi, sia la descrizione delle interfacce da utilizzare per la cattura degli eventi (Capture Interface) e la conseguente richiesta di questi da parte delle applicazioni (Query Interface).

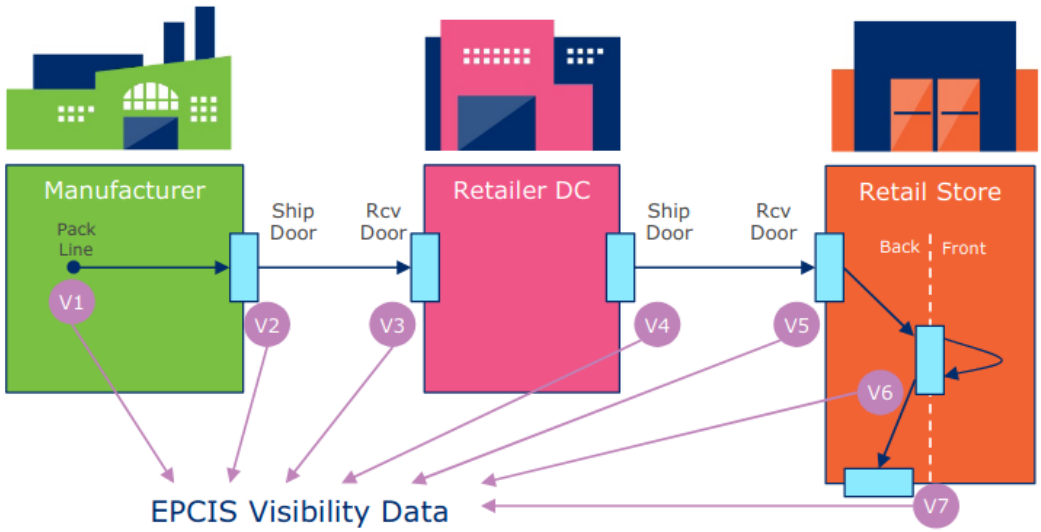

Figura 1.1: Esempio di generazione di eventi EPCIS.

Un evento EPCIS è una struttura generata ad ogni step eseguito dal prodotto all'interno della filiera e nella quale vengono specificate una serie di informazioni organizzate in quattro dimensioni:

- *What* - Identificativo del prodotto ricevuto.
- *When* - Data e ora della ricezione del prodotto.
- *Where* - Luogo in cui è stata effettuata la spedizione del prodotto.

- *Why* - Contesto del prodotto, come ad esempio collegamenti a fatture, indicazione di informazioni sul destinatario, ecc.

Gli eventi EPCIS possono inoltre essere categorizzati in quattro diverse tipologie:

- *ObjectEvent* - Descrive un evento avvenuto nei confronti di un oggetto, come la ricezione di un prodotto all'interno di uno stabilimento.

- *AggregationEvent* - Rappresenta un evento che riguarda l'aggregazione o la disaggregazione fisica di oggetti, come lo stoccaggio o la rimozione di un prodotto all'interno di un magazzino.

- *TransformationEvent* - Rappresenta un evento in cui degli oggetti forniti in ingresso vengono usati totalmente o parzialmente per la creazione di un nuovo oggetto, come la fabbricazione di un prodotto a partire dalle sue materie prime.

- *TransactionEvent* - Rappresenta un evento nel quale uno o più oggetti vengono associati o dissociati a/da una transazione commerciale, come l'associazione di un prodotto alla propria fattura.

Gli eventi vengono generati tramite l'utilizzo di dispositivi in grado di leggere codici a barre o tag RFID interagendo con un'applicazione di cattura che invia gli eventi alla Capture Interface, la quale si occupa del salvataggio degli eventi in un repository. A questo fa accesso la Query Interface, un'interfaccia definita come un web service che rende disponibili gli eventi catturati alle applicazioni definendo una serie di API utilizzabili da queste. La raccolta degli eventi da parte delle applicazioni consente a queste di effettuare operazioni che normalmente non sarebbero possibili, come ad esempio capire lo stato di un prodotto cercandone l'ultimo evento associato, capire quali sono le fasi che un prodotto attraversa all'interno della filiera, effettuare l'analisi degli eventi ricevuti nel tempo da parte di un particolare processo industriale o automatizzare l'avvio dei processi industriali attraverso la ricezione di un determinato evento.

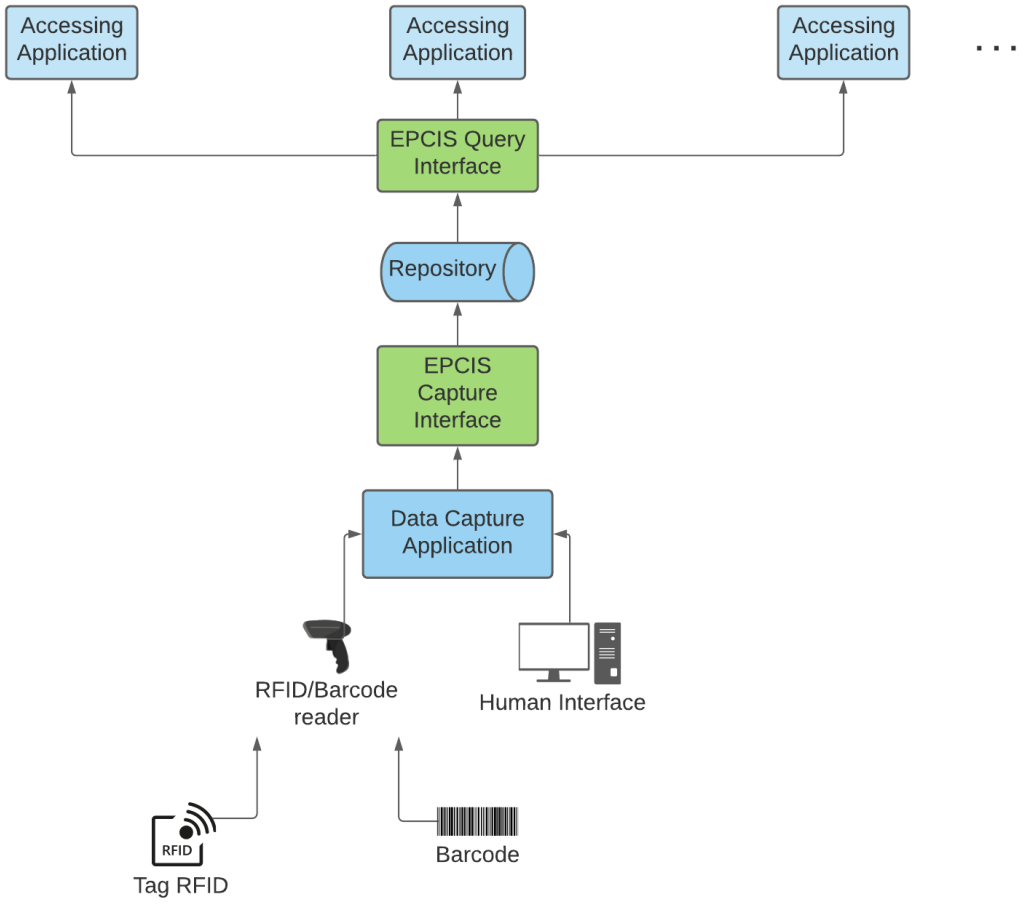

Figura 1.2: Catena della gestione degli eventi EPCIS.

1.3 XTap

XTap [5] è un progetto portato avanti da LINKS, un centro di ricerca del Politecnico di Torino, per lo sviluppo di una piattaforma che mette al servizio delle aziende degli strumenti che consentono di monitorare un'intera filiera produttiva. L'idea dietro ad XTap è quella di creare un network che colleghi i fornitori ai clienti e agli stakeholder, fornendo un sistema al quale le aziende possano registrarsi ed inviare aggiornamenti riguardanti le varie fasi di produzione ed elaborazione che i propri prodotti attraversano nella filiera tramite richieste HTTP contenenti degli eventi EPCIS. Una volta prodotti dalle varie aziende, ad esempio attraverso la lettura di tag RFID o di codici a barre da parte di un lettore o tramite creazione di un evento dall'interfaccia utente di XTap, gli eventi vengono inviati al server di XTap, inseriti in una coda, elaborati e mandati al set di aziende della filiera autorizzate a

riceverli, le quali possono visualizzare le informazioni riguardo agli eventi ricevuti nel cruscotto reso loro disponibile. Qui vengono visualizzate per ogni evento le quattro dimensioni che lo caratterizzano (What, Why, Where, When). Gli eventi riguardanti un singolo prodotto vengono raggruppati in modo tale da poter visualizzarne la storia mostrando l'elenco delle varie fasi attraversate da questo all'interno della filiera, con la possibilità di scegliere la granularità temporale di visualizzazione. Sono poi disponibili anche grafi per monitorare il flusso di produzione della filiera e altri strumenti che consentono di controllare i KPI (Key Performance Indicator) dell'azienda.

1.4 Requisiti funzionali dei dispositivi

Come abbiamo potuto constatare nelle sezioni precedenti, il concetto di IIoT e di Smart Logistics è profondamente legato alla presenza di dispositivi smart. Anche la maggior parte del lavoro svolto in questa tesi gravita attorno al principio di dispositivo, per questo motivo in questa sezione cercheremo di capire cos'è un dispositivo e quali sono le funzionalità che ci si aspetta che questo abbia. Un dispositivo è una macchina che abbia, come requisiti minimi, un sistema operativo e una scheda di rete che consentano a questa di comunicare con altre macchine ed effettuare richieste HTTP. Ciò significa che i dispositivi possono essere di varia natura e possono essere impiegati in diversi tipi di attività, ad esempio, come vedremo, un dispositivo può essere identificato in uno strumento Android per la lettura e scrittura di tag RFID o anche in un PC Linux/Windows per uso generale. Nonostante le grandi diversità che possono essere presenti tra i vari dispositivi, è necessario che questi abbiano una base di funzionalità comune a tutti rappresentata dalle operazioni di autenticazione, auto-configurazione e monitoraggio.

Solitamente, quando pensiamo alle modalità con cui può essere effettuata una procedura di autenticazione pensiamo subito all'inserimento di uno username e di una password all'interno di un form di login. Tuttavia questa modalità può essere molto scomoda nel caso dei dispositivi, i quali verranno utilizzati da personale che deve a questo punto memorizzare i dati di accesso di tutti i dispositivi e inserirli ogniqualvolta ce ne sia il bisogno. Per evitare ciò, i dispositivi devono essere quindi dotati di un certificato digitale che contenga le informazioni necessarie ad effettuare la procedura di autenticazione, che in questo modo diventerebbe un semplice controllo della validità di un certificato.

Affinché posseggano tutti gli strumenti necessari per effettuare le operazioni a cui sono destinati, i dispositivi devono essere capaci di auto-configurarsi scaricando una

configurazione dal server. Prendiamo ad esempio un lettore RFID Android: questo potrebbe non contenere le applicazioni necessarie per effettuare operazioni come, ad esempio, la registrazione di un prodotto o la verifica di un ordine. È necessaria quindi la presenza di una procedura che porti all'installazione delle applicazioni sopracitate in modo automatico e senza l'intervento dell'utilizzatore finale.

Infine i dispositivi devono essere in grado di fornire statistiche sull'utilizzo degli stessi e delle applicazioni su di essi installate in modo tale da controllarne il comportamento ed eventuali anomalie.

Capitolo 2

Tecnologie e strumenti utilizzati

2.1 Docker

Docker [6] è un framework che offre un ambiente virtualizzato per lo sviluppo e il deployment di applicazioni all'interno di *container*, ovvero dei "contenitori" isolati che permettono di separare le applicazioni dall'infrastruttura delle macchine sulle quali queste vengono eseguite. Questa separazione semplifica il processo di sviluppo di un prodotto poiché le dipendenze di quest'ultimo, come ad esempio librerie o file di configurazione, vengono tutte gestite nel contesto del container grazie a delle immagini, dei pacchetti contenenti tutto ciò che è necessario per avviare uno specifico servizio, consentendo quindi la portabilità dell'applicazione su macchine anche molto differenti. Ogni container all'interno di Docker ha anche la possibilità di comunicare con gli altri container, attraverso la creazione di una rete locale, e di effettuare la persistenza dei dati, tramite la creazione di un volume associato al container. Docker è basato su un'architettura client-server: gli utenti utilizzano il client Docker per inviare richieste al server mediante l'utilizzo di comandi da CLI, i quali vengono ricevuti da un elemento del server chiamato *Docker deamon* che si occupa della ricezione delle richieste e della gestione di container, immagini, volumi e reti. Le immagini da eseguire nei container possono essere scaricate da un registry oppure costruite tramite un'operazione di build a partire da un'altra immagine il cui riferimento nel registry viene specificato, insieme ad altri parametri, all'interno di un file chiamato Dockerfile.

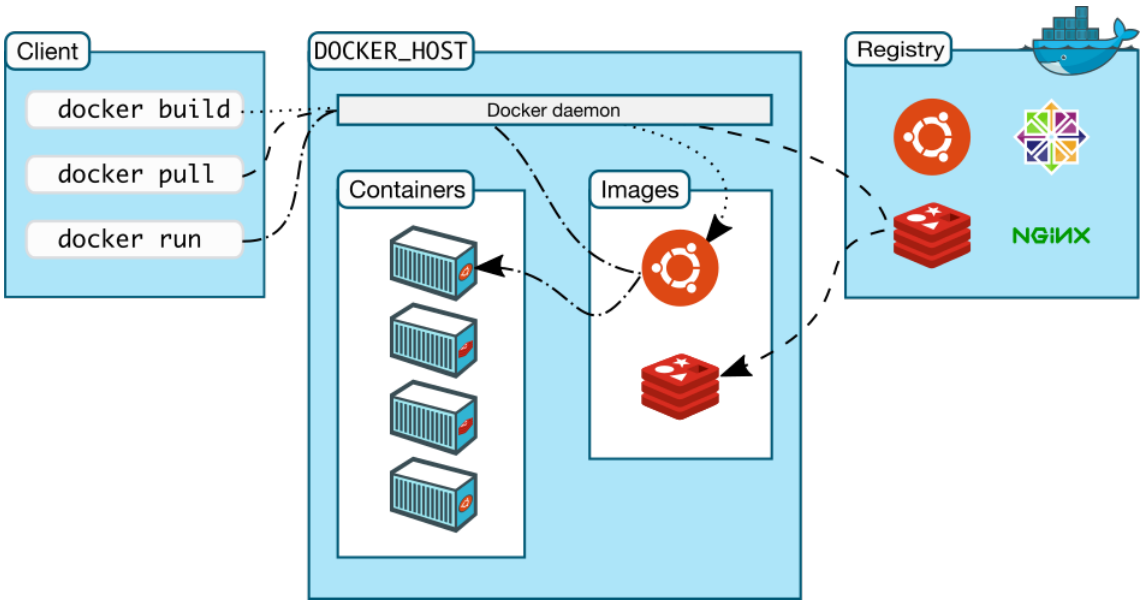

Figura 2.1: Architettura dei componenti Docker.

La logica alla base dei container deriva dal concetto di virtualizzazione, cioè la creazione di una macchina virtuale che sfrutta le risorse hardware della macchina reale sottostante. I container differiscono tuttavia dalle macchine virtuali poiché la virtualizzazione avviene solo a livello di sistema operativo e non a livello hardware. Questa differenza rende i container molto più leggeri e performanti rispetto alle macchine virtuali, diminuendo i tempi di avvio dei servizi e offrendone maggiore scalabilità. Nel momento in cui i servizi da gestire all'interno dei container siano tanti, Docker offre la possibilità di gestirli in modo semplice tramite il tool Docker Compose.

2.1.1 Docker Compose

Docker Compose [7] è un tool per la gestione di applicazioni Docker multi-container, che utilizza dei file YAML, chiamati file Compose, per descrivere i servizi da mandare in run all'interno dei container. Tra le varie operazioni offerte, le più importanti sono quelle di *build* e *up*, che consentono rispettivamente di buildare e avviare tutti i servizi elencati all'interno del file Compose specificato. Ogni servizio, una volta avviato, viene aggiunto alla rete locale di Docker con la possibilità di comunicare con tutti gli altri servizi containerizzati presenti nella rete. All'interno dei file Compose ogni servizio può inoltre specificare un file .env contenente delle variabili d'ambiente da poter leggere a runtime all'interno del container.

2.2 API REST e FastAPI

Le API (Application Programming Interface) sono degli strumenti che consentono ad un'applicazione di accedere ai dati e alle funzionalità di altre applicazioni. Per permettere ciò, le API fanno solitamente affidamento ad un protocollo o ad un'architettura software, una delle quali è rappresentata da REST (Representational State Transfer) [8]. REST è un sistema di regole che sfrutta le richieste HTTP, effettuate da un *client*, per gestire un insieme di risorse, contenute all'interno di un *server*. Le risorse sono rappresentate da un URI, un identificatore globale diverso per ogni risorsa, che può essere utilizzato per effettuare diverse operazioni sulla risorsa stessa. Le possibili operazioni che il client può effettuare dipendono dal metodo utilizzato nella richiesta HTTP, che può essere di tipo:

- POST, per richiedere la creazione di una risorsa;
- GET, per richiederne la lettura;
- PUT o PATCH, per richiederne l'aggiornamento rispettivamente totale o parziale;
- DELETE, per richiederne la cancellazione.

Attraverso le richieste, client e server si scambiano rappresentazioni della risorsa, ovvero documenti caratterizzati da uno specifico formato e contenenti le informazioni che costituiscono la risorsa stessa. Il formato usato per la rappresentazione della risorsa può essere di vario tipo e quello più utilizzato è JSON, un formato contenente coppie chiave-valore.

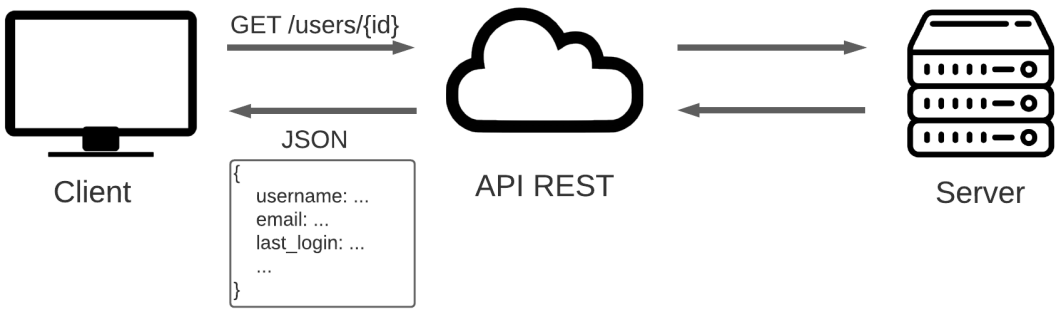

Figura 2.2: Esempio di richiesta di una risorsa user attraverso API REST.

Lo sviluppo della business logic dietro alle API può essere molto complesso, sopratutto nel caso di progetti contenenti un gran numero di risorse differenti e

tra loro correlate. Per questo motivo, per la creazione di un server, ci si appoggia solitamente a delle architetture logiche ausiliari come ad esempio FastAPI [9], un framework per lo sviluppo di server REST in Python. Caratteristica principale di FastAPI è la sua velocità, dovuta all'utilizzo da parte del framework della libreria *asyncio* di Python per la programmazione concorrente. FastAPI consente inoltre di sviluppare le API in modo facile, veloce e intuitivo permettendone la divisione logica attraverso un sistema di APIRouter, degli oggetti che antepongono all'URI associato agli endpoint delle API un prefisso che consente la divisione di queste in diverse categorie.

2.3 Certificati digitali X.509

Un certificato digitale è un documento che lega in modo univoco un'identità ad un chiave pubblica e che può essere utilizzato, attraverso questa sua caratteristica, nell'ambito di procedure di autenticazione dei dati e dei soggetti. Il contenuto di un certificato digitale dipende dallo standard a cui aderisce, il più comune dei quali è rappresentato dallo standard X.509 [10], che definisce diversi aspetti come la struttura, la richiesta e la revoca di un certificato. I certificati X.509 sono costituiti principalmente da tre elementi: il certificato vero e proprio, in cui vengono specificate informazioni come ad esempio il richiedente, l'ente che lo ha emesso, la validità o la chiave pubblica, l'algoritmo crittografico usato per la firma e la firma digitale della Certificate Authority che ha emesso il certificato. Per poter richiedere un certificato, è necessario che il richiedente invii, all'interno di una Certificate Signing Request, i dati identificativi necessari ed una chiave pubblica da associare al certificato ad una Certificate Authority, che si occuperà dell'emissione dello stesso apponendo la propria firma digitale. Nel momento in cui ce ne fosse il bisogno, un certificato può essere invalidato attraverso l'emissione di una revoca.

2.4 React Framework

React [11] è una libreria open-source JavaScript mantenuta da Facebook per la creazione di interfacce utente. Lo sviluppo di una UI in React passa attraverso l'utilizzo di componenti, pezzi indipendenti di codice riutilizzabili che accettano degli argomenti, chiamati *props*, e che ritornano un elenco di elementi da visualizzare a video e che costituiscono la struttura del componente. I componenti possono essere basati su funzioni o su classi: nel primo caso ritornano gli elementi da visualizzare

come una semplice funzione, nel secondo caso questo compito viene effettuato dal metodo *render()* contenuto nella classe.

```
function Hi(props) {
    return <h2>Hi, I am {props.name}!</h2>;
}

class Hi extends React.Component {
    render() {
        return <h2>Hi, I am {this.props.name}!</h2>;
    }
}
```

Listing 2.1: Esempio di componente basato su funzione (in alto) e su classe (in basso).

Per lo sviluppo dei componenti viene utilizzato JSX, un linguaggio di scrittura che combina JavaScript e XML e che consente di avere maggiore controllo sulla struttura del rendering dei componenti. Ogni componente in React ha un lifecycle definito e che passa attraverso tre fasi:

- *Mounting* - Gli elementi del componente vengono aggiunti al DOM e renderizzati.
- *Updating* - Il componente viene aggiornato e ri-renderizzato a causa di un cambiamento del suo stato o dei suoi props.
- *Unmounting* - Il componente viene rimosso dal DOM.

Durante la fase di update di un componente, il DOM di React non esegue l'aggiornamento dell'intera pagina ma vengono confrontate le differenze tra il nuovo stato e quello precedente all'aggiornamento, modificando soltanto i componenti interessati.

Grande importanza all'interno di React è rivestita dagli *hooks*, delle funzioni che permettono di agire sullo stato e sul ciclo di vita dei componenti e che sono stati aggiunti a React per sopperire alle difficoltà di gestione dello stato e di comprensione del codice dovuto alla presenza delle classi. Tra tutti gli hook principali menzioniamo quelli che verranno utilizzati nella tesi, ovvero *useState()*, che permette di avere delle variabili all'interno di un componente, e *useContext()*, che consente di creare dati comuni all'interno di un contesto a cui può accedere un'intera gerarchia di componenti.

2.5 MongoDB

MongoDB [12] è un database No-SQL opensource basato su documenti che consente l'archiviazione di grossi volumi di dati. Essendo un database No-SQL, MongoDB non effettua il salvataggio dei dati come un normale database relazionale attraverso l'utilizzo di tabelle, ma si serve del concetto di documento, una struttura composta da coppie chiave-valore in modo molto simile ad un JSON. I documenti sono raccolti all'interno di collezioni che a loro volta sono logicamente raggruppate in databases. Le collezioni sono l'equivalente delle tabelle all'interno di un database relazionale, le cui righe invece sono rappresentate dai documenti. In MongoDB tuttavia i documenti all'interno di una collezione possono differire strutturalmente sia per nome delle chiavi sia per numero di queste ultime. Ogni documento deve però obbligatoriamente includere il campo *_id*, contenente l'identificativo del documento, che assume la funzione di chiave primaria.

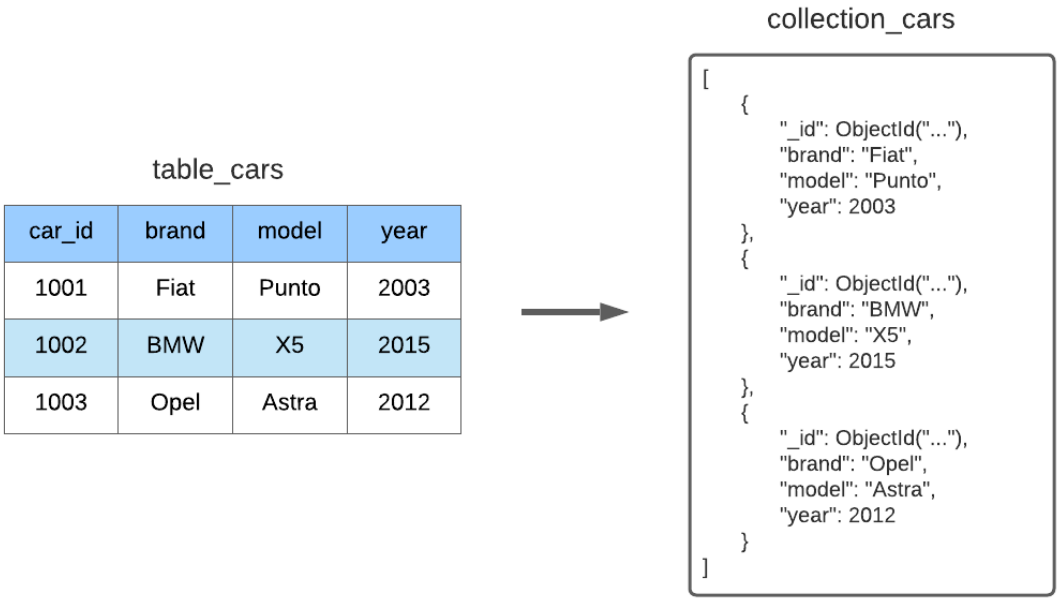

Figura 2.3: Esempio di differenza tra database relazionale e MongoDB.

Per poter gestire i dati, MongoDB mette a disposizione questa lista di operazioni CRUD [13] da effettuare sui documenti nel contesto delle collezioni:

- *.insertOne({ <field1>: <value>, <field2>: <value> ... })* - Inserimento di un documento.

- *.insertMany([<document 1> , <document 2>, ...])* - Inserimento di più documenti specificando un array di documenti.
- *.find(<field1>: <value>, <field2>: <value> ...)* - Richiesta di tutti i documenti contenenti determinate coppie chiave-valore.
- *.updateOne(<filter>, <update>)* - Aggiornamento dei campi definiti in <update> del primo documento contenente le coppie chiave-valore specificate in <filter>.
- *.updateMany(<fiter>, <update>)* - Aggiornamento dei campi definiti in <update> di tutti i documenti contenenti le coppie chiave-valore specificate in <filter>.
- *.replaceOne(<fiter>, <update>)* - Sostituzione del primo documento contenente le coppie chiave-valore specificate in <filter> con un documento contenente i campi definiti in <update>.
- *.deleteOne(<filter>)* - Eliminazione del primo documento contenente le coppie chiave-valore specificate in <filter>.
- *.deleteMany(<filter>)* - Eliminazione di tutti i documenti contenenti le coppie chiave-valore specificate in <filter>.

MongoDB dispone anche di configurazioni che consentono di aumentare la resilienza e la scalabilità orizzontale del database. Ciò è possibile attraverso l'utilizzo di *replica sets*, processi che possono essere mandati in run su più macchine e che consentono di gestire la duplicazione dei dati, e di meccanismi di *sharding*, attraverso i quali poter partizionare i dati su server multipli.

2.6 Processamento degli stream

2.6.1 Kafka e Zookeeper

Kafka [14] è una piattaforma sviluppata da Apache per l'elaborazione di stream di messaggi in tempo reale attraverso un meccanismo di tipo publish-subscribe. Essa permette lo streaming di informazioni mantenendo una latenza molto bassa e offrendo la possibilità di salvare i messaggi su storage prevenendone la perdita. La gestione dei dati passa attraverso diverse fasi che cominciano con la creazione di un messaggio da parte di uno o più *producer*, un'entità che genera dati, e il conseguente invio verso il cluster di Kafka. Il cluster contiene dei server, chiamati *broker*, che si occupano

della ricezione dei messaggi e del loro smistamento all'interno di *topics*, delle code che contengono tutti i messaggi appartenenti ad una determinata categoria. Per garantire la scalabilità del sistema, i topic vengono divisi in partizioni, ognuna delle quali viene affidata ai vari broker che compongono il cluster e può essere replicata in modo tale da garantire la reliability dei dati. I messaggi infine vengono mandati ai *consumer*, delle entità che ricevono e consumano tutti i messaggi a loro destinati sulla base dei topic a cui essi si sono precedentemente iscritti. Per consentire la sincronizzazione dei broker e l'accesso ai topic da parte dei producers e dei consumers, Kafka fa affidamento a Zookeeper [15], un server open-source per il coordinamento distribuito di applicazioni cloud.

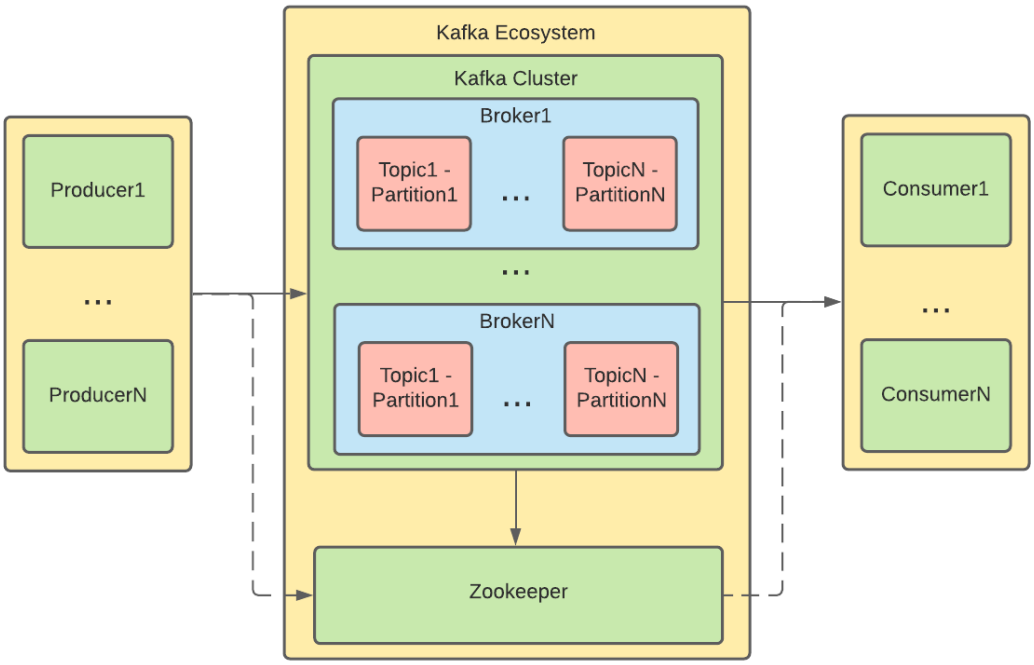

Figura 2.4: Architettura dell'ecosistema Kafka.

2.6.2 Faust

Faust [16] è una libreria Python per il processamento di stream Kafka. Essa fornisce una serie di decorator da associare a delle funzioni e che permettono la gestione dei messaggi Kafka e dei dati di processamento [17]. I decorator sono:

- *@app.agent* - Crea un agent, ovvero un consumer che riceve ed elabora i messaggi di un determinato topic Kafka.

- *@app.task* - Chiama la funzione associata una sola volta non appena l'applicazione Faust è operativa.
- *@app.timer* - Chiama la funzione associata alla scadenza di un timer la cui durata viene decisa dall'utente.
- *@app.crontab* - Chiama la funzione associata in uno specifico momento determinato da un Crontab.
- *@app.page* - Crea una pagina web associata ad un web server interno a Faust.
- *@app.command* - Crea un comando CLI associato all'applicazione Faust.

Le funzioni che utilizzano questi decorator vengono fatte partire all'interno di una o più istanze di Faust che prendono il nome di *workers* e che sono in grado di processare decine di migliaia di eventi al secondo.

2.7 Monitoraggio

2.7.1 Prometheus

Prometheus [18] è un sistema di monitoring open-source basato su metriche. Queste sono costituite da un nome, che consente di identificarle nel momento in cui si esegue una query, e da un certo numero di *labels*, coppie chiave-valore che includono informazioni aggiuntive col fine di differenziare le caratteristiche della metrica cui fanno riferimento.

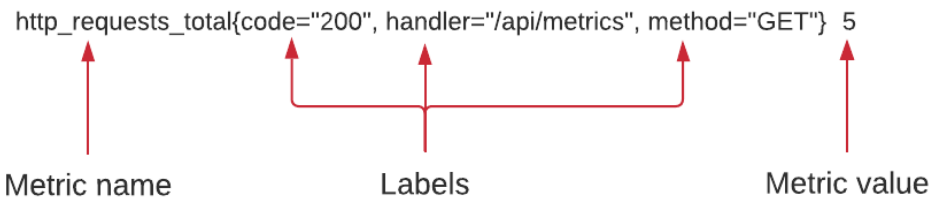

Figura 2.5: Esempio di metrica.

Le metriche vengono salvate come time-series e vengono lette attraverso una logica pull, ovvero scaricandole con delle richieste HTTP da dei servizi che le rendono disponibili a degli specifici endpoint e che vengono chiamati *jobs*. Prometheus si

occupa quindi dell'aggregazione di tutti i jobs, dando la possibilità di effettuare query sulle metriche da questi contenute. Le query vengono effettuate utilizzando PromQL, un linguaggio che consente agli utenti di selezionare e aggregare i dati delle time-series in modo molto sintetico. Prometheus fornisce quattro tipi di metriche che i jobs possono utilizzare per esporre i dati:

- Counter - Rappresenta un contatore monotonico crescente il cui valore può solo aumentare o essere settato a zero.
- Gauge - Rappresenta un valore che può arbitrariamente sia aumentare che diminuire.
- Histogram - Campiona il valore di un'entità e conteggia i valori ottenuti dividendoli in *buckets*.
- Summary - Come Histogram, ma offrendo la possibilità di calcolare i quantili dei valori campionati all'interno di una finestra temporale.

2.7.2 Grafana

Grafana [19] è una soluzione open-source per effettuare data analytics e monitoraggio di dati attraverso la visualizzazione di questi all'interno di dashboards. Le dashboards sono dei pannelli in cui sono contenuti grafici ed infografiche, i quali possono essere creati attraverso un'interfaccia query builder. Grafana offre anche un supporto API a livello HTTP per la gestione di vari aspetti, come ad esempio la creazione di dashboards e grafici, l'accesso degli utenti o l'aggiunta di data source. Grafana supporta nativamente come data sources una grande quantità di tecnologie, tra cui Prometheus e MongoDB.

2.8 Kubernetes

Kubernetes [20] è una piattaforma open-source per il deployment, la gestione e lo scaling di applicazioni containerizzate. Essa si occupa dell'orchestrazione dei container gestendo in modo automatico la creazione dei servizi, la loro scalabilità e il loro failover. Effettuando il deployment di Kubernetes si ottiene un cluster, ovvero un insieme di macchine, virtuali o fisiche, in cui i container sono eseguiti. Queste macchine vengono chiamate *nodes*, i quali possono essere di due tipi: i *Worker Nodes*, che eseguono i carichi di lavoro dell'utente, e i *Control Plane Nodes*, che coordinano l'esecuzione dei Worker Nodes. All'interno di un Worker Node sono presenti tre elementi:

- *kubelet* - Gestisce le risorse col nodo e comunica con il Control Plane Node.
- *kubeproxy* - Gestisce le regole del networking.
- *Container Runtime* - Esegue i container, i quali vengono raggruppati all'interno di *Pods*.

I Control Plane Nodes sono invece costituiti da questi componenti:

- *kube-apiserver* - Espone delle API che consentono di controllare il cluster Kubernetes. I client possono usufruire delle API attraverso l'utilizzo di *kubectl*, il tool da riga di comando di Kubernetes.
- *etcd* - Un database chiave-valore in cui viene salvato lo stato del cluster.
- *kube-scheduler* - Assegna i Pods ai nodi.
- *kube-controller-manager* - Gestisce dei controller più piccoli che eseguono operazioni riguardanti ad esempio il monitoring dei nodi o la replicazione dei Pods.
- *kube-cloud-manager* - Consente l'integrazione dei vari cloud provider con Kubernetes. [21]

Come abbiamo potuto vedere parlando dei Worker Nodes, i container vengono eseguiti all'interno di questi raggruppati in Pods, delle istanze che raccolgono tutti i container che condividono le stesse risorse e che rappresentano la più piccola unità di esecuzione in Kubernetes. I Pods vengono definiti all'interno di file YAML specificando un elenco di parametri come, ad esempio, il nome del Pod e l'elenco dei container da posizionare all'interno di questo. Tuttavia solitamente gli utenti non utilizzano direttamente i Pods per la gestione dei container, ma si affidano a dei concetti di livello più alto come ad esempio i *Deployment*. Questi vengono definiti esattamente come i Pods all'interno di file YAML e permettono di avere il controllo completo del ciclo di vita dell'applicazione containerizzata, con la possibilità di replicare l'applicazione specificando il numero di Pods da utilizzare e di effettuare l'upgrade dell'applicazione in modo automatico. Essendo i Pods delle risorse non permanenti, questi possono essere creati e distrutti all'interno dei Deployment in qualsiasi momento, rendendo difficile il tracciamento del loro indirizzo IP. Per favorire la connettività tra i container quindi, i Deployment vengono solitamente accompagnati da un Service, anch'esso definito attraverso uno YAML. I Service sono delle risorse in cui viene specificato il tipo di politica da usare per poter accedere ad un determinato Pod e che inseriscono un'associazione nome-indirizzo IP del Pod all'interno del DNS del cluster. Un esempio

di tipo di politica è *LoadBalancer*, grazie al quale il Service espone un solo indirizzo IP pubblico, valido per ogni Pod appartenente al Deployment, e che permette di effettuare load balancing tra i Pod.

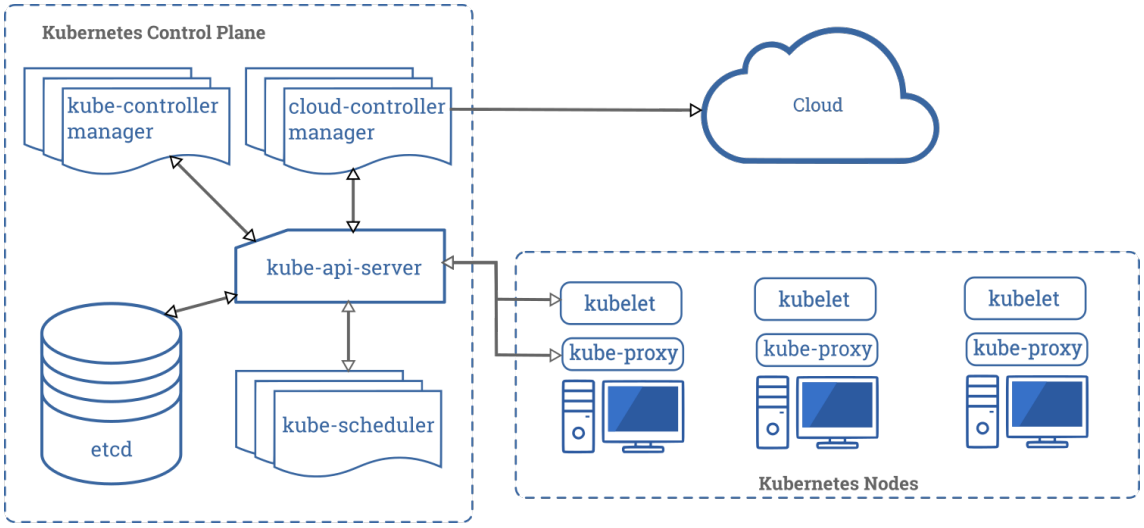

Figura 2.6: Componenti principali di Kubernetes.

Capitolo 3

Architettura della soluzione proposta

Come possiamo vedere dalla figura 3.1 la piattaforma di XTap è costituita da diversi componenti, eseguiti all'interno di container Docker. Il punto di partenza per l'avvio di questi componenti è costituito dallo script *xtap-cli.sh*, tramite il quale vengono effettuate le operazioni di build e run dei container, e dal file *local.env*, in cui sono contenute le variabili d'ambiente necessarie al corretto funzionamento dei vari componenti. Avviati i container, il primo servizio con cui si ha a che fare è il server REST: questo contiene le API che consentono di gestire gli eventi EPCIS, le aziende, gli utenti e tutte le risorse che possono essere inquadrate nel contesto di una filiera produttiva, le quali vengono poi salvate all'interno di MongoDB. Parte delle API vengono utilizzate dal frontend, in cui gli utenti possono loggarsi e avere una visualizzazione grafica delle informazioni e delle statistiche che riguardano le varie fasi di elaborazione dei prodotti della filiera. La gestione degli eventi invece viene effettuata attraverso il lavoro combinato di Kafka e Faust: il primo legge da MongoDB, attraverso un connector, gli eventi che arrivano al server e li salva all'interno di un topic, il secondo si occupa invece dell'elaborazione di questi attraverso degli agent che leggono gli eventi dai topic stessi.

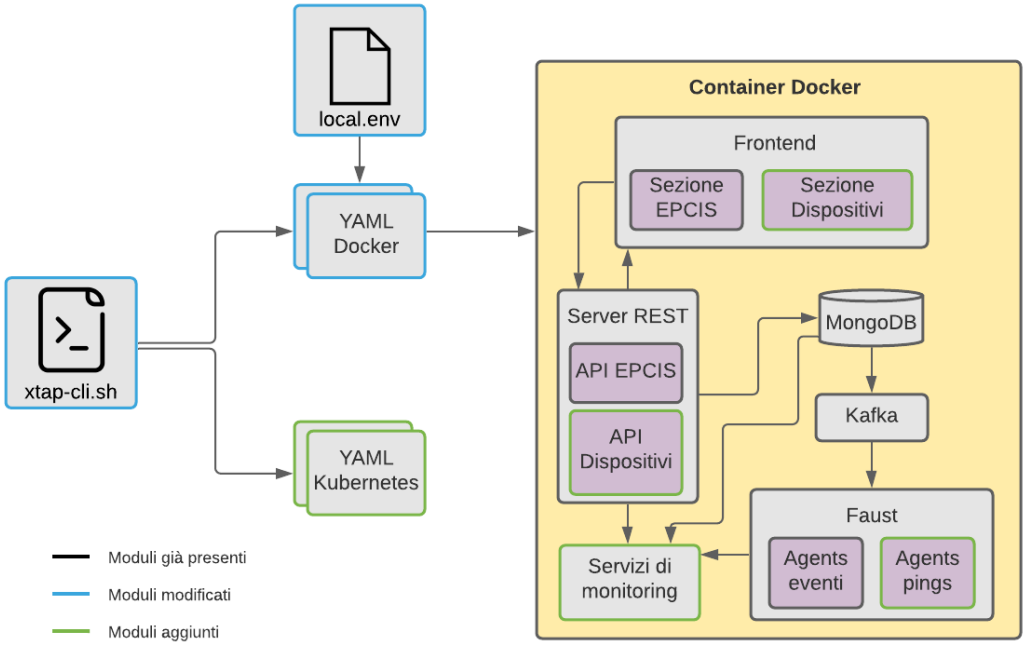

Figura 3.1: Architettura generale di XTap.

Dalla figura possiamo anche vedere quali sono le modifiche apportate nell'ambito del lavoro di questa tesi. Queste possono essere sintetizzate in:

- Aggiunta dei servizi di supporto ai dispositivi nel server e nel frontend.
- Aggiunta dei servizi per il monitoraggio del sistema.
- Modifica delle configurazioni Docker.
- Aggiunta delle configurazioni per il deployment della piattaforma su Kubernetes.

Tenendo conto di questa divisione, in questo capitolo andremo ad analizzare le componenti aggiunte ad XTap, focalizzandoci in primo luogo sull'architettura lato server del supporto alle funzionalità dei dispositivi introdotte nei capitoli precedenti. Sposteremo poi la nostra attenzione sui servizi che compongono la pipeline del sistema di monitoring, concludendo con la descrizione delle modifiche aggiunte nel contesto delle operazioni di build e avvio dei container Docker in cui sono racchiusi i servizi di XTap e il processo di migrazione degli stessi verso l'ambiente Kubernetes di Google.

3.1 Supporto dei dispositivi

Per avere il controllo totale sulla gestione dei dispositivi è necessario che gli elementi che costituiscono il supporto a questi siano mappati su delle API REST. Ciò significa che tutto quello che concerne i dispositivi, inclusi i dispositivi stessi, deve essere presente all'interno del server come risorsa da poter richiedere e modificare tramite richieste HTTP. Per raggiungere questo obiettivo sono state quindi individuate le seguenti risorse:

- **Device** - fa da interfaccia tra il server ed i dispositivi fisici e ne consente la gestione.
- **Certificate** - memorizza le informazioni riguardanti i certificati dei dispositivi.
- **Revocation** - memorizza le informazioni riguardanti le revoche dei certificati dei dispositivi.
- **Configuration** - consente il salvataggio delle configurazioni dei dispositivi.
- **Apk e Compose File** - consentono l'upload e il download di file di tipo APK e YAML.
- **Auth Request** - permette la gestione delle richieste di autenticazione provenienti dai dispositivi.
- **Ping** - consente il monitoraggio dei dispositivi e delle applicazioni installate su questi.
- **Grafana Data** - memorizza l'URL di embedding dei pannelli contenenti i grafici di Grafana per il monitoraggio dei dispositivi.

Queste risorse sono state quindi aggiunte al server REST di XTap, affiancandole a quelle già presenti, e sulla base di queste sono state create delle API che consentissero il controllo di tutti gli aspetti riguardanti i dispositivi e che vedremo di seguito.

3.1.1 API per la gestione delle risorse

Tutte le risorse che abbiamo appena introdotto, ad esclusione di Ping e Grafana Data che rappresentano un caso a parte, dispongono di API che ne consentono la gestione attraverso richieste HTTP secondo i principi REST. Le operazioni disponibili attraverso le API sono riassunte per ogni risorsa nella tabella in figura 3.2.

	Richiesta (GET)	Creazione (POST)	Modifica (PUT/PATCH)	Eliminazione (DELETE)
Device	✓	✓	✓	✓
Certificate	✓	✗	✗	✗
Revocation	✓	✓	✗	✗
Configuration	✓	✓	✓	✓
Apk e Compose FIle	✓	✓	✗	✓
Auth Request	✓	✓	✓	✗

Figura 3.2: Tabella delle operazioni REST sulle risorse.

Tra tutte le risorse, come possiamo vedere dalla tabella, Certificate rappresenta un'anomalia, poiché non dispone di nessun metodo al di fuori di quello che ne consente la visualizzazione. Questo è dovuto al fatto che la creazione di un certificato avviene soltanto durante il processo di creazione di una risorsa Device. La creazione di questa può infatti essere scomposta in:

- Creazione della risorsa Device.
- Creazione di un certificato X.509 e della relativa risorsa Certificate.
- Creazione di un utente XTap associato al dispositivo.

Così come per la creazione, lo stesso discorso vale per l'eliminazione, con la differenza che in questo caso l'oggetto Certificate non viene fisicamente eliminato dal server ma viene soltanto emessa una revoca permanente che, in quanto tale, non può essere eliminata tramite una richiesta DELETE. Anche le risorse Auth Request, di cui parleremo introducendo il concetto di autenticazione manuale, non possono essere eliminate tramite richiesta poiché vengono automaticamente rimosse dal server.

Tutte le operazioni sulle risorse sono disponibili nel cruscotto di XTap nella sezione *Device Admin*, nome che coincide con il ruolo che un utente deve possedere tra i suoi diritti per visualizzare la sezione ed effettuare qualsiasi operazione sulle risorse. Questa sezione rappresenta il punto focale della gestione dei dispositivi, poiché da qui possono essere effettuate azioni di cruciale importanza come, oltre a quelle di creazione e cancellazione delle risorse, settare una determinata configurazione per un dispositivo, cambiarne il certificato o accettarne o rifiutarne le richieste di autenticazione.

3.1.2 API per l'autenticazione

Come già discusso nei capitoli precedenti, i dispositivi vengono muniti di un certificato digitale X.509, emesso da una CA interna al server, nel quale sono contenute una serie di informazioni che costituiscono l'identità del dispositivo stesso. Per consentire l'accesso alle API di XTap all'utente associato al dispositivo, il certificato va presentato presso l'endpoint *auth/token_X509* in modo tale da iniziare la procedura di autenticazione. Questa può avvenire secondo due modalità che possono essere impostate per il singolo dispositivo nella sezione Device Admin nel frontend. La modalità può essere scelta tra:

- **Autenticazione standard** - Questo tipo di autenticazione segue un flusso molto basilare: quando un dispositivo presenta il proprio certificato all'endpoint di autenticazione, il server controlla che il certificato sia stato emesso dalla CA aziendale, controllandone la firma, e che non sia stato revocato. In caso di successo, il server risponde positivamente inserendo nell'header della risposta un token JWT da utilizzare da parte del dispositivo per poter autenticare le future richieste. In caso di fallimento, il server risponde con codice di errore 401 Unauthorized.

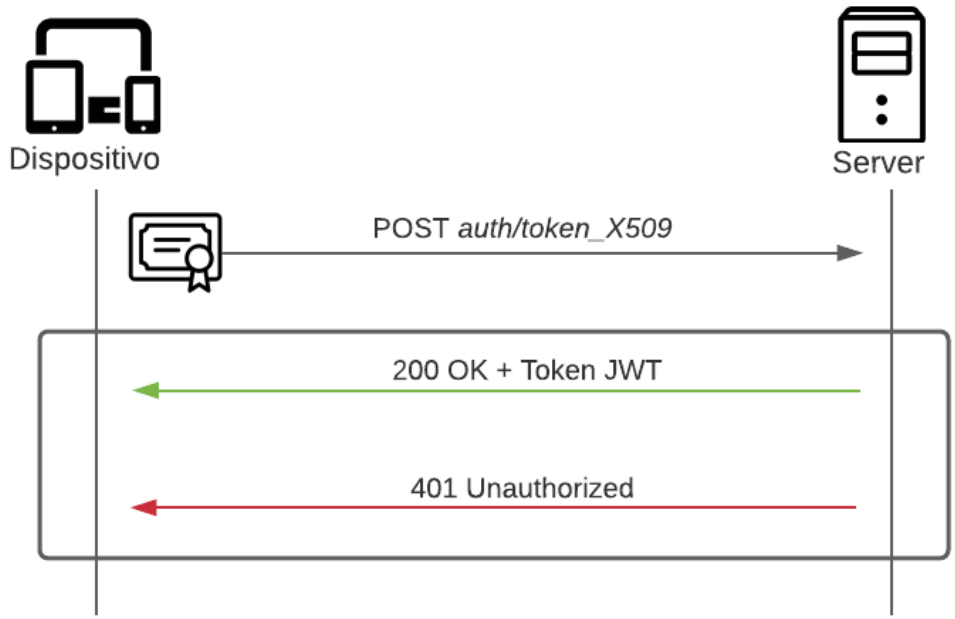

Figura 3.3: Diagramma di sequenza dell'autenticazione standard.

- **Autenticazione manuale** - Nel caso dell'autenticazione manuale, prima di eseguire la vera e propria autenticazione da parte del server, vengono eseguite una serie di operazioni volte ad inserire un livello di sicurezza in più rispetto all'autenticazione standard nel caso si operasse con dispositivi a rischio di duplicazione del certificato. In questo caso infatti il dispositivo, supponendo che sia in polling presso l'endpoint di autenticazione, non subisce immediatamente il processo di autenticazione ma la sua richiesta viene inoltrata, creando una risorsa di tipo AuthRequest, al Device Admin che può decidere in tempo reale se accettarla o meno dal frontend del server. In caso affermativo, si procede come nell'autenticazione standard. In caso negativo, il server risponderà al dispositivo con codice di errore 401 Unauthorized.

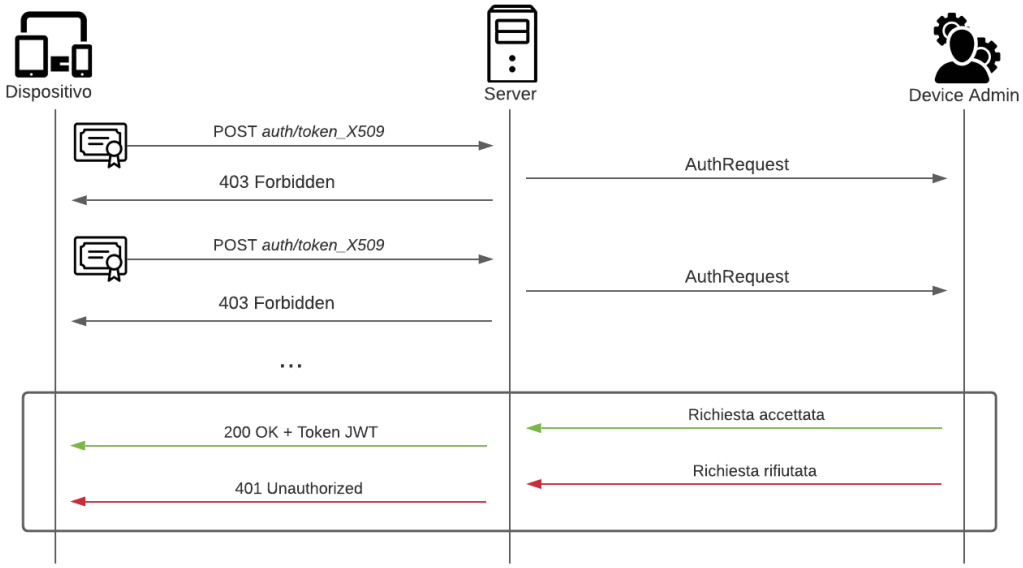

Figura 3.4: Diagramma di sequenza dell'autenticazione manuale.

3.1.3 API per l'auto-configurazione

Per favorire il processo di auto-configurazione dei dispositivi, a questi è stato reso disponibile l'endpoint *api/devices/get_configuration/*, tramite il quale richiedere, dopo aver effettuato l'autenticazione, una configurazione precedentemente settata per il singolo Device all'interno della sezione Device Admin nel frontend. Nel nostro contesto, una configurazione non è altro che un file JSON in cui sono indicati diversi parametri di cui natura e numero dipendono dal sistema operativo del dispositivo. È lecito pensare che tra i parametri di una configurazione ci possa essere la lista di

applicazioni da installare sul dispositivo, difatti, allo stato attuale, il server supporta l'inserimento di configurazioni per dispositivi che abbiano o un sistema operativo Android o che siano in grado di avviare delle immagini Docker, specificando per questi all'interno delle proprie configurazioni rispettivamente la lista dei nomi degli APK da installare o il nome del file Compose contenente l'elenco delle immagini Docker da avviare. APK e file Compose, oltre a disporre di API che ne consentono l'upload e il download dal server, forniscono infatti una API, che i dispositivi possono utilizzare durante il processo di auto-configurazione, per richiedere una risorsa specificandone il nome.

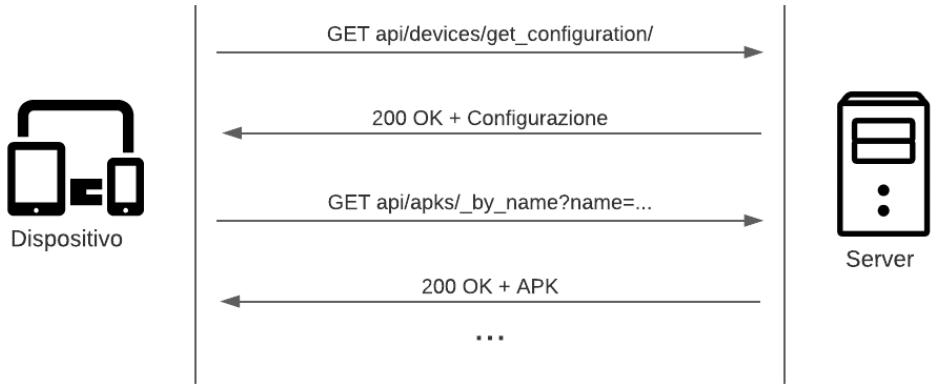

Figura 3.5: Diagramma di sequenza dell'auto-configurazione di un dispositivo.

3.1.4 API per il monitoraggio

Per fare in modo che l'uso dei dispositivi e delle applicazioni su di questi installate possa essere monitorato è stata resa disponibile agli stessi una API di *ping* e *heartbeat*, ovvero un endpoint, situato in *monitoring/ping* per la ricezione e la gestione di segnali singoli, nel caso dei ping, o periodici, nel caso degli heartbeat, inviati dai dispositivi e in cui sia eventualmente specificata l'applicazione in uso da questi ultimi. I dispositivi inviano i ping e gli heartbeat tramite delle richieste contenenti una risorsa di tipo Ping e nel momento in cui una di queste viene ricevuta dal server, la stessa viene salvata all'interno di MongoDB in un'apposita collezione. Successivamente un connector MongoDB-Kafka [22] si occupa della conversione dell'operazione di inserimento del documento all'interno del database in un record Kafka, che viene poi inserito in una coda all'interno di un topic in cui vengono raccolti tutti i record di tutti i ping e heartbeat ricevuti da ogni dispositivo registrato ad XTap. Questi record vengono consumati da un agent Faust incaricato di ispezionare ogni singolo ping o

heartbeat ricevuto e di aggiornare di conseguenza una serie di metriche di Prometheus. Queste metriche vengono utilizzate da Grafana per la visualizzazione di grafici in cui vengono mostrate statistiche sull'utilizzo dei dispositivi e delle applicazioni. I grafici in questione possono essere consultati o connettendosi a Grafana come admin, nel caso si volessero controllare le statistiche riguardanti tutte le aziende, o dalla sezione Device Admin nel frontend, in cui nell'apposita sottosezione sono disponibili i pannelli con le statistiche per la singola azienda.

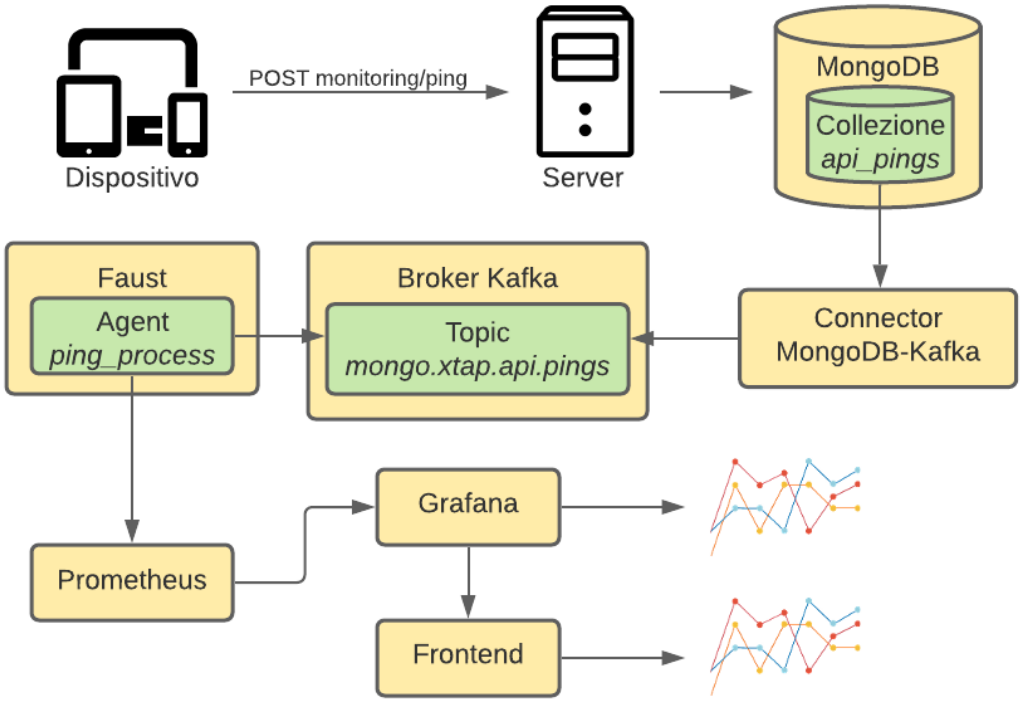

Figura 3.6: Catena di gestione dei ping e degli heartbeat.

3.1.5 Altre API rilevanti

Cambio chiave di cifratura dei certificati

Siccome i certificati digitali dei dispositivi devono poter essere letti dal frontend è opportuno che questi vengano salvati sul server in modo sicuro, per questo motivo vengono prima cifrati e poi salvati sul database. Se per qualsiasi motivo la chiave di crittografia utilizzata venisse esposta, ci sarebbe il chiaro rischio di furto e duplicazione dei certificati. Per evitare ciò, è stata resa disponibile una API per

il cambio della chiave: l'API è raggiungibile all'endpoint *admin/certificates/change_encryption_key/* ed è disponibile solo agli utenti che dispongono dei diritti di super_admin.

Simulazione di eventi EPCIS

XTap dispone di un generatore di eventi EPCIS con cui simulare l'invio di eventi da dispositivi appartenenti ad aziende diverse, tuttavia gli eventi vengono direttamente mandati al database senza passare attraverso le API. Per questo motivo è stata aggiunto l'endpoint *epcis_events/from_generator/* che consente al generatore di eventi di passare prima dal server.

3.2 Servizi per il monitoring

In XTap è stato aggiunto un blocco contenente dei servizi per il monitoring che è necessario non solo per seguire l'utilizzo delle applicazioni e dei dispositivi come visto nella sezione precedente, ma anche per monitorare l'utilizzo di MongoDB, delle API del backend e degli eventi EPCIS ricevuti dal server.

3.2.1 Le metriche

Il blocco di monitoring è costituito da diversi sottocomponenti che cooperando consentono il recupero delle informazioni da monitorare, la memorizzazione su storage delle stesse e la loro visualizzazione grafica. Affinché tutto ciò sia possibile è però necessario prima di tutto identificare delle sorgenti di dati da cui poter prelevare questo tipo di informazioni sotto forma di dati grezzi da elaborare successivamente. Questi dati grezzi sono rappresentati dalle *metriche* e le sorgenti non sono altro che dei servizi, che nel nostro specifico caso sono:

- Faust, per il monitoraggio degli eventi e, come abbiamo visto, dei dispositivi e delle applicazioni;
- MongoDB, per il monitoraggio dell'efficienza del database;
- Server Rest, per il monitoraggio dell'utilizzo delle API.

Se per Faust le metriche sono nativamente disponibili grazie ad un sensore interno che le espone ad uno specifico endpoint, lo stesso non vale per MongoDB e per il server REST. Andando in ordine, per poter esportare i metadati e le statistiche di MongoDB e renderle disponibili a Prometheus è stato necessario creare un exporter di

metriche chiamato *mongo-query-exporter*, ovvero uno script col compito di estrapolare informazioni, creare ed aggiornare delle specifiche metriche ed esporre un endpoint da cui prelevare queste ultime. Per quanto riguarda il server invece, creazione e aggiornamento delle metriche vengono effettuati in un *middleware* di FastAPI, cioè una funzione che effettua pre-processamento delle richieste prima che queste vengano effettivamente ricevute e gestite dal server, mentre l'esposizione delle metriche viene effettuata sfruttando lo stesso principio delle API.

3.2.2 Descrizione e funzionamento dei componenti

Una volta resi disponibili tutti gli endpoint dei servizi da cui prelevare le metriche, gli altri componenti entrano in gioco: ad interfacciarsi direttamente con i servizi è Prometheus, che controlla la raggiungibilità degli endpoint e permette la lettura delle metriche da questi ultimi con la possibilità di effettuare delle query mirate in linguaggio PromQL. L'indirizzo Ip degli endpoint da cui effettuare la lettura delle metriche viene comunicato a Prometheus tramite un file di configurazione che contiene, oltre alle informazioni sugli endpoint, anche l'indirizzo Ip del database su cui effettuare lo storage a lungo termine dei dati delle metriche lette. La soluzione solitamente adottata per Prometheus per lo storage è TimescaleDB che però, essendo un database SQL, non mantiene la possibilità di usare PromQL come linguaggio per le query. Per porre rimedio a questo problema, a Prometheus è stato affiancato Promscale [23], una soluzione che contiene al suo interno TimescaleDB [24] e un connector. Come possiamo vedere in figura 3.7, la presenza del connector che si frappone tra Prometheus e TimescaleDB fa sì che i dati delle metriche memorizzati siano accessibili ai client esterni sia tramite query SQL che PromQL. A questo punto i dati memorizzati vengono letti tramite query PromQL da Grafana, che si occupa della visualizzazione grafica delle metriche attraverso una dashboard accessibile al suo indirizzo e visualizzabile una volta loggati come admin. Anche Grafana, così come per Prometheus, dispone di file di configurazione che consentono il provisioning delle dashboard e il settaggio della sorgente da cui prelevare i dati, che nel nostro caso è appunto il connector di Promscale.

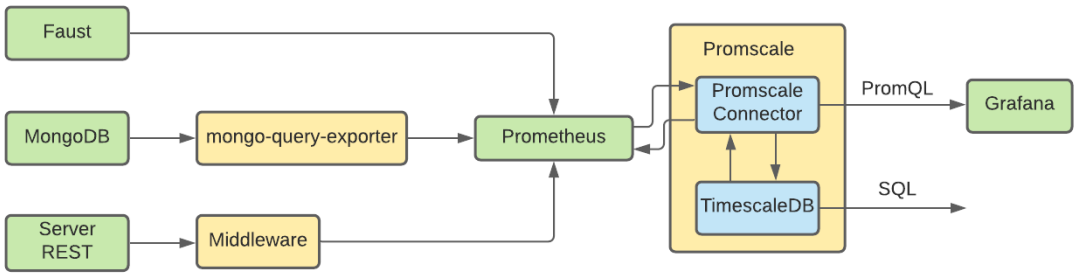

Figura 3.7: Architettura del blocco di monitoring.

3.3 Gestione delle immagini Docker

Fulcro della gestione di tutti i servizi in XTap è *xtap-cli*, uno script che permette l'esecuzione di varie operazioni sulle immagini Docker in modo semplice ed automatico a partire dai file Compose e dai Dockerfile dei servizi di XTap. Tra le varie operazioni importanza maggiore è rivestita da quelle di build e run, che consentono appunto di buildare o avviare tutti i container Docker allo stesso tempo con un unico comando. Con l'aggiunta dei nuovi servizi per il monitoring, queste operazioni hanno subito un notevole appesantimento, impattando quindi sullo sviluppo di XTap. Per questo motivo, per poter avere il controllo sulla scelta delle immagini su cui effettuare le operazioni di build e run, nel contesto di queste operazioni è stata aggiunta la possibilità di specificare una modalità tra:

- *base*
- *base-develop*
- *monitor*
- *monitor-develop*

La modalità *base* comprende i servizi già presenti precedentemente all'interno di XTap, mentre la modalità *monitor* seleziona anche le immagini contenenti il blocco di monitoring. Il suffisso *-develop* aggiunto alle modalità sopra descritte consente di lanciare alcune delle immagini esponendo delle porte in locale, in modo tale da poter essere raggiunte dall'esterno della rete locale Docker e facilitare il testing dei servizi in fase di sviluppo.

3.4 Migrazione a Kubernetes

Configurando Docker nel modo opportuno ed effettuando del port forwarding è possibile esporre i container all'esterno rendendoli raggiungibili da un host remoto, tuttavia questa soluzione, oltre ad essere scarsamente adottata in produzione, ha molte limitazioni in fatto di scalabilità ed affidabilità. Per questo motivo la soluzione adottata per il deployment di XTap è Kubernetes, o più precisamente l'engine Kubernetes offerto da Google (GKE) [25]. Disponendo dei file YAML contenenti le configurazioni Kubernetes dei servizi di XTap, le operazioni per la gestione delle immagini su Kubernetes vengono effettuate tramite lo script xtap-cli esattamente come per le immagini Docker in locale, a patto di disporre di uno user Google con cui si abbia effettuato la procedura di autenticazione sulla macchina da usare per il deployment e che abbia i giusti permessi per operare sul cluster di XTap. Effettuati questi passi preliminari, lo script xtap-cli permette di effettuare le operazioni di upload e run dei container su Kubernetes attraverso due comandi: *push-images* e *run-on-k8s*. Il primo comando effettua il build delle immagini Docker, selezionate specificando una modalità tra base e monitor, e ne esegue il push verso il container registry di Google associato al progetto. Il secondo comando invece si occupa della creazione dei servizi Kubernetes sul cluster effettuando l'upload delle configurazioni di questi, includendo le variabili d'ambiente contenute all'interno del file local.env.

Capitolo 4

Implementazione della soluzione

Nel corso di questo capitolo verrà spiegato come i servizi aggiunti ad XTap, dei quali sono stati descritti composizione e funzionamento nel capitolo precedente, sono stati effettivamente programmati, quali scelte sono state prese nel corso dello sviluppo e quali migliorie sono state apportate alla piattaforma. Il capitolo verrà diviso in 5 macrosezioni che comprenderanno lo sviluppo del server lato API Rest e lato frontend, del blocco di monitoring, delle configurazioni Docker e della migrazione a Kubernetes, precedute da una breve descrizione dei linguaggi di programmazione utilizzati.

4.1 Linguaggi utilizzati

Durante lo sviluppo sono stati utilizzati diversi linguaggi di programmazione tra cui:

- Python per la scrittura del backend e di alcuni script per il monitoring di MongoDB e il testing delle API di XTap.
- JSX per la scrittura del frontend col framework React.
- Bash per la scrittura dello script di gestione delle immagini Docker di XTap.
- YAML per la scrittura dei file di configurazione per la parte di monitoring e Kubernetes.
- PromQL per effettuare query a Prometheus da Grafana.

Non c'è stata una vera e propria scelta dei linguaggi da utilizzare poiché questi sono subordinati alla natura degli strumenti utilizzati. L'unica possibile scelta è stata quella effettuata per l'esecuzione delle query a Prometheus, la cui alternativa era rappresentata da SQL (come descritto anche nella sezione 3.2.2): PromQL fornisce infatti una migliore capacità di sintesi delle query e maggiore facilità di aggregazione dei dati rispetto ad SQL nel contesto delle metriche.

4.2 Sviluppo delle API Rest

4.2.1 Struttura generale delle API

Il blocco principale del backend di XTap è costituito dal server FastAPI e le risorse aggiunte a quest'ultimo hanno mantenuto la stessa natura di quelle già presenti al suo interno. Questo significa che Device, Configuration e le altre risorse sono state inserite seguendo la già presente divisione interna in modelli, servizi ed API.

Modelli

I modelli sono delle classi che contengono i dati delle risorse su cui la soluzione si basa. Nello caso specifico di XTap, per i modelli sono stati usati dei *pydantic models*, ovvero delle classi che effettuano parsing e validazione dei dati che arrivano al server. Prendiamo ad esempio il codice sottostante:

```
1  class Device(MyBaseModel):
2      name: str
3      object_owner: Optional[str] = None
4      configuration: Optional[str] = None
5      manual_authorization: bool = False
6
7      @validator('name')
8      def check_name_not_empty(cls, v):
9          assert v != '', 'Name cannot be empty'
10         return v
```

Listing 4.1: Pydantic model di Device

Il pydantic model impone che alcuni campi siano obbligatoriamente di un determinato tipo, come per *name* per il quale vengono soltanto accettate stringhe, o che siano opzionali, come *configuration*. C'è anche la possibilità di inserire un validatore, cioè una funzione con cui è possibile effettuare un controllo più accurato sui singoli campi che compongono il modello: il validatore a riga 7 ad esempio controlla che il campo

name non sia vuoto. Nel caso in cui i dati in arrivo al server da una richiesta non soddisfino il modello pydantic associato, il server risponderà con codice di errore 422: Unprocessable Entity.

Servizi

L'accesso ai dati avviene tramite i servizi, degli oggetti che consentono di effettuare operazioni di lettura e scrittura dei dati presenti nei modelli attraverso delle query a MongoDB. Ogni classe contenente un servizio eredita da CRUDBase, una classe già presente in XTap che contiene dei metodi con le operazioni basilari da effettuare sul DB, come ad esempio lettura, scrittura o aggiornamento di un documento.

```
class CrudConfiguration(CRUDBase[Configuration]):

    def __init__(self, db=settings.MONGO_DB):
        super().__init__(Configuration, 'api_configurations')

    def by_name(self, name: str, owner: str = None) -> Optional[
Configuration]:
        if owner is not None:
            doc = self._coll.find_one({"name": name, "object_owner"
: owner})
        else:
            doc = self._coll.find_one({"name": name})
        return doc if doc is None else Configuration(**doc)
```

Listing 4.2: Servizio di Configuration

Come possiamo vedere dal codice, CrudConfiguration eredita da CRUDBase e ne specializza il comportamento aggiungendo il metodo *by_name*, che cerca e ritorna un documento contenente una configurazione filtrando per nome di quest'ultima.

API

Le API costituiscono gli endpoint da raggiungere e a cui effettuare le richieste. Nel momento in cui una richiesta viene ricevuta, sia essa una GET, una POST o qualunque altro tipo supportato dal server, questa viene servita chiamando in causa un servizio che ritornerà i dati richiesti o effettuerà altre operazioni di vario tipo. Prima che il servizio entri in gioco può esserci un processo di controllo della validità dei dati ricevuti nel body della richiesta o dei diritti posseduti dall'utente che l'ha effettuata.

```python
@router.get("/certificates/{certificate_id}")
def read_certificate(certificate_id: str, current_user: User =
    Depends(get_current_active_user)):
     if "device_admin" in current_user.rights:
         mdb = crudCertificates.read(certificate_id, current_user.
    main_domain)
         if mdb is None:
             certificate_not_found_exception = HTTPException(
                 status_code=status.HTTP_404_NOT_FOUND,
                 detail="Certificate not found",
                 headers={"WWW-Authenticate": "Bearer"},
             )
             raise certificate_not_found_exception
         return CertificateReq.parse_obj(mdb)
     else:
         credentials_exception = HTTPException(
             status_code=status.HTTP_401_UNAUTHORIZED,
             detail="Could not validate credentials - User is not a
    device admin",
             headers={"WWW-Authenticate": "Bearer"},
         )
         raise credentials_exception
```

Listing 4.3: Esempio di richiesta di un certificato tramite Id

4.2.2 Meccanismi delle operazioni sulle risorse

La maggior parte delle operazioni sulle risorse offerte dai servizi derivano dalla classe CrudBase, tuttavia alcune di queste meritano di essere analizzate nel dettaglio poiché sono caratterizzate da un comportamento unico che le contraddistingue e che andremo quindi a vedere di seguito.

Creazione dei Device

Come spiegato nella sezione 3.1.1, la creazione di un Device si articola nella creazione di tre elementi distinti. Questa procedura deve avvenire in modo atomico affinché, se qualcosa andasse storto durante una delle operazioni, lo stato del server possa rimanere inalterato e non subire modifiche parziali ai dati. Per ottenere questo comportamento ci si è avvalsi delle *transaction* di MongoDB, che garantiscono atomicità durante l'esecuzione di operazioni su più documenti e su più collezioni. Per

consentire l'uso delle transaction si sono rese necessarie delle modifiche alla classe CRUDBase:

```
    def create(self, obj: ModelType, session=None) -> str:
        doc = obj.dict()
        if "_id" in doc and doc["_id"] == None:
            del doc["_id"]
        if session is not None:
            return self._coll.insert_one(doc, session=session).
    inserted_id
        return self._coll.insert_one(doc).inserted_id
```

Listing 4.4: Metodo create di CRUDBase

tra i parametri del metodo *create* è stato aggiunto *session*, un oggetto di tipo ClientSession che identifica la sessione associata alla transazione. Nel caso in cui nel parametro session venga specificato un oggetto di questo tipo, esso viene aggiunto anche tra i parametri dell'operazione su MongoDB in modo tale da poter effettuare al termine della transazione il commit o, in caso di errore, l'abort di questa. In questo modo tutti i servizi che ereditano da CRUDBase possono, tramite il parametro session, chiamare il metodo create, e anche altri che vedremo, nell'ambito di una transazione.

È possibile richiedere la creazione di un Device tramite una richiesta POST verso la API /api/devices/, che al suo interno richiama il metodo *create_device()* del service per la gestione dei Device. La creazione dei tre elementi (l'oggetto Device, lo user associato e il certificato X.509) avviene in questo metodo: tramite una clausola *with*, come possiamo vedere nel codice sottostante, i metodi per la creazione del Device e del certificato vengono chiamati specificando lo stesso identificativo di sessione.

```
    with client.start_session() as new_session:
        with new_session.start_transaction():
            crudUsers.create(user, session=new_session)
            crudCertificates.create(certificate, session=
    new_session)
            return self._coll.insert_one(doc, session=new_session).
    inserted_id
```

Listing 4.5: Creazione degli elementi del Device

Il metodo *create* del servizio crudCertificate esegue una serie di operazioni volte alla creazione di un certificato X.509: prima crea una coppia di chiavi RSA 4096 bit e successivamente crea una CSR, settando il campo COMMON NAME col nome

associato al dispositivo e specificato durante la sua creazione e inserendo una delle chiavi precedentemente generate come chiave pubblica. A questo punto viene eseguita la generazione del certificato a partire dalla CSR e la firma con la chiave privata della CA interna ad XTap. Il certificato viene poi cifrato con Fernet [26], un tool per la crittografia simmetrica disponibile nella libreria Python *cryptography*, e salvato all'interno di MongoDB.

```
        # create public/private key
        key = PKey()
        key.generate_key(TYPE_RSA, 4096)

        # Generate CSR
        client_req = X.509Req()
        client_req.get_subject().CN = doc["common_name"]
        client_req.set_pubkey(key)
        client_req.sign(key, 'sha256')

        # Generate certificate
        ca_key = crypto.load_privatekey(FILETYPE_PEM, settings.
    CA_KEY, settings.CA_PASSPHRASE.encode('ascii'))
        ca_cert = crypto.load_certificate(FILETYPE_PEM, settings.
    CA_CERT)
        client_cert = crypto.X.509()
        client_cert.gmtime_adj_notBefore(0)
        client_cert.gmtime_adj_notAfter(5 * 365 * 24 * 60 * 60)  #
    5 years
        client_cert.set_issuer(ca_cert.get_subject())
        client_cert.set_subject(client_req.get_subject())
        client_cert.set_pubkey(client_req.get_pubkey())
        client_cert.set_serial_number(cert_serial_int)
        client_cert.sign(ca_key, 'sha256')
        encryptor = Fernet(settings.ENCRYPTION_SECRET_KEY)
        doc["certificate"] = encryptor.encrypt(crypto.
    dump_certificate(FILETYPE_PEM, client_cert))
        if session is not None:
            return self._coll.insert_one(doc, session=session).
    inserted_id
        return self._coll.insert_one(doc).inserted_id
```

Listing 4.6: Generazione del certificato X.509

Autenticazione dei Device

Per eseguire l'autenticazione presso XTap, i dispositivi devono effettuare una richiesta POST all'endpoint */auth/token_X509* inserendo nel body della richiesta un contenuto JSON come questo

```
{
    "certificate": <certificato in formato PEM>
}
```

Il comportamento del dispositivo durante la fase di autenticazione dipende dal fatto che l'autenticazione manuale sia attiva o meno, ovvero:

- Se l'autenticazione manuale è attiva, il dispositivo si metterà in polling sull'endpoint di autenticazione.
- Se l'autenticazione manuale non è attiva, il dispositivo eseguirà una sola richiesta all'endpoint di autenticazione.

```
1    ...
2    user = cache.get(doc["certificate"])
3    if user is None:
4        user = authenticate_user_X.509(crypto.load_certificate(
   FILETYPE_PEM, doc["certificate"]))
5        if not user:
6            raise HTTPException(
7                status_code=status.HTTP_401_UNAUTHORIZED,
8                detail="Invalid certificate",
9                headers={"WWW-Authenticate": "Bearer"},
10           )
11       cache[doc["certificate"]] = user
12
13   device = crudDevices.by_name(user.dict()["username"], user.dict
   ()["company_prefix"])
14   ip_address = request.client.host
15   if device.dict()["manual_authorization"]:
16       manual_authorization_check(user.dict()["username"],
   ip_address, user.dict()["company_prefix"])
17   ...
```

Listing 4.7: Autenticazione del Device

Ricevuta la richiesta il server controlla tramite un ExpiringDict [27], ovvero una struttura dati che funziona come una cache temporanea, se il dispositivo abbia

effettuato richieste nell'ultimo minuto: questo evita di effettuare, per ogni eventuale richiesta in polling da parte del dispositivo, il controllo della validità del certificato essendo un'operazione onerosa. Effettuata questa azione, appurato che il dispositivo abbia effettuato solo una prima (ed eventualmente unica) richiesta, viene effettuato il controllo sulla validità del certificato e sull'esistenza dell'utente e la verifica che l'autenticazione manuale sia abilitata o meno. Nel caso questa non fosse abilitata l'iter seguito sarebbe quello descritto nel capitolo 3.1.2 nel caso di autenticazione standard. Se invece fosse abilitata, verrebbe chiamato il metodo *manual_authorization_check* che crea un oggetto di tipo AuthRequest e lo salva, attraverso il servizio per le AuthRequest, nella collezione *api_auth_requests*. Gli elementi di questa collezione rappresentano delle richieste di autenticazione e scadono dopo 60 secondi se non refreshati, questo per evitare che le richieste rimangano memorizzate dopo che un dispositivo abbia smesso di effettuare il polling sull'endpoint di autenticazione. Gli oggetti di tipo AuthRequest contengono, tra i vari campi, un campo *state* che indica lo stato della richiesta tramite una *enum* di questo tipo:

```
1    class RequestStateEnum(IntEnum):
2        pending = 0
3        authorized = 1
4        unauthorized = 2
```

Listing 4.8: Enum per il campo *state*

Alla prima richiesta da parte del dispositivo, la richiesta viene salvata con lo stato settato a pending e il server risponde con codice 403 Forbidden. Dalla seconda richiesta in polling del dispositivo in poi sono tre gli scenari che si possono presentare:

- La richiesta è ancora in stato pending, si esegue il refresh della richiesta e il server risponde con codice 403 Forbidden.

- Il Device Admin ha rifiutato la richiesta di autenticazione e il server risponde quindi con codice di errore 401 Unauthorized.

- Il Device Admin ha accettato la richiesta di autenticazione e il server risponde inviando un token JWT coi dati dell'utente.

Nel momento in cui la richiesta viene accettata o rifiutata, questa viene di conseguenza eliminata dal database.

Richiesta e modifica della configurazione

La configurazione del dispositivo può essere settata con una richiesta PATCH all'endpoint *api/devices/update_configuration/{device_id}*, inserendo nel body un JSON con questa struttura

```
{
    "name": <nome del dispositivo>
    "configuration": <nome della configurazione>
}
```

Il dispositivo, una volta autenticato, può poi scaricare la configurazione in automatico dall'endpoint *api/devices/get_configuration/*

Cambio del certificato

Nel caso in cui il certificato di un dispositivo venisse compromesso è possibile effettuare revoca e creazione di un nuovo certificato in un'unica operazione. Per fare ciò, bisogna effettuare una richiesta GET all'endpoint *api/devices/{device_id}/change_certificate*: da qui, il server effettua una serie di controlli su permessi dell'utente che fa la richiesta, esistenza del dispositivo e presenza di un certificato già revocato associato al dispositivo. Il lavoro effettivo è quindi svolto dal metodo *change_device_certificate* del servizio dei Device il quale, usando una transazione, crea un oggetto Revocation associato al certificato, se questo non fosse già stato precedentemente revocato, e ne crea uno nuovo associandolo al Device.

Cancellazione del Device

La cancellazione è un'operazione molto simile a livello strutturale all'operazione di creazione. Si parte da una richiesta DELETE all'endpoint *api/devices/{device_id}* che, dopo aver effettuato controlli sui permessi dell'utente e sull'esistenza del dispositivo, chiama il metodo *delete_device_data* del service dei Device: come per le altre operazioni, anche qui viene usata una transazione per eliminare lo user associato, creare un oggetto Revocation associato al certificato, nel caso questo non fosse già stato revocato, ed eliminare l'oggetto Device.

Cambio della chiave di cifratura dei certificati

L'API per il cambio di chiave di cifratura è raggiungibile all'endpoint *admin/certificates/change_encryption_key/* ed è disponibile solo agli utenti che dispongono dei diritti di *super_admin*. Dall'API viene chiamato il metodo *change_encyption_key*

del servizio dei certificati che si occupa della decifratura e ricifratura con nuova chiave di tutti i certificati salvati sul database avvalendosi di una transazione. Al termine dell'operazione, la vecchia chiave viene sovrascritta con quella nuova.

```
    def change_encyption_key(self):
        new_key = Fernet.generate_key()
        decryptor = Fernet(settings.ENCRYPTION_SECRET_KEY)
        encryptor = Fernet(new_key)
        cert_list = list(self._coll.find({}))
        client = get_client()
        with client.start_session() as session:
            with session.start_transaction():
                for doc in cert_list:
                    doc["certificate"] = decryptor.decrypt(doc["
certificate"])
                    doc["certificate"] = encryptor.encrypt(doc["
certificate"])
                    self._coll.update_one({"_id": ObjectId(doc["_id
"])}, {"$set": {"certificate": doc["certificate"]}},
                                          session=session)
        settings.ENCRYPTION_SECRET_KEY = new_key
```

Listing 4.9: Metodo per il cambio di chiave di cifratura dei certificati

Upload e download di APK e file Compose

Nelle configurazioni scaricate dai dispositivi possono essere incluse, come descritto nel capitolo 3.1.3, liste di applicazioni da installare o di immagini Docker da mandare in run. Per rendere disponibile tutto ciò ai dispositivi, sono state create delle API per l'upload e il download di APK e file Compose. Questi vengono inviati al server tramite una richiesta POST agli endpoint *api/apks/* e *api/compose_files* specificando nel body un contenuto rispettivamente del tipo *application/vnd.android.package-archive* e *application/x-yaml*. Qui i file vengono caricati su MongoDB tramite GridFS [28], un tool che permette la memorizzazione di file di ogni tipo superando la restrizione imposta da MongoDB per i documenti in formato BSON di 16MB. La gestione dei file avviene tramite un oggetto di tipo Uploads, che al suo interno contiene delle crud per l'upload e il download di file tramite GridFS. I file, indistintamente dal tipo, vengono tutti salvati all'interno della collezione fs.files in MongoDB, con la possibilità di preporre un suffisso al nome del file per specificarne il tipo: nel nostro caso i suffissi possibili sono apk/ e compose_file/. Anche se upload e download sono

gestiti dall'oggetto di tipo Uploads, APK e file Compose sono forniti di servizio che consente l'eliminazione dal database e l'elencazione.

```
    ...
    if "device_admin" in current_user.rights:
        if file.content_type == "application/x-yaml":
            uploader = Uploads("compose_file")
            res = uploader.write_file(
                name=file.filename,
                data=await file.read(),
                content_type=file.content_type,
                owner=current_user.main_domain)
            return {"id": res}
    ...
```

Listing 4.10: Upload di un file Compose

4.2.3 Script per il testing

Per poter testare in modo semplice e automatico le API per l'autenticazione manuale e per il ping dei Device sono stati creati degli script usando *Click* [29], una libreria Python che facilita la creazione di script per command-line.

device_auth.py

Con questo script è possibile testare il funzionamento dell'autenticazione manuale, avendo disponibili uno user con i diritti di Device Admin e una risorsa Device già creata. Lo script può essere avviato con questo comando

```
python3 device_auth.py {user_name} {device_name}
```

specificando il nome dello user e il nome del Device da utilizzare per l'autenticazione. Lo script manda delle richieste in polling all'endpoint di autenticazione e aspetta che il Device Admin accetti o rifiuti la richiesta, mostrando a video l'esito della scelta.

ping.py

Questo script può essere usato per testare le funzionalità di ping e heartbeat dei dispositivi e può essere eseguito con

```
python3 ping.py {user_name}
```

dove user_name è il nome di uno user coi diritti di Device Admin, utilizzato per la creazione dei Device da utilizzare per mandare i segnali di ping e heartbeat. Il comando può anche essere lanciato usando le opzioni –n_pings per specificare il numero di ping da inviare al server (default=20), –n_devices per specificare il numero di devices da simulare (default=5) e –n_apps per specificare il numero di app da simulare (default=5).

gen.py

Già precedentemente presente all'interno di XTap, questo script permette di inviare eventi al server passando direttamente dal servizio per gli eventi, bypassando le API. Per evitare ciò è stata inserita l'opzione *–user* che consente di specificare lo username dell'utente da utilizzare per l'invio degli eventi verso le API del server.

4.3 Sviluppo del frontend

Le modifiche ad XTap riguardanti il frontend coincidono con l'aggiunta della sezione Device Admin nella homepage di XTap, in cui sono stati inseriti tutti gli strumenti di gestione delle risorse dei dispositivi. Per aggiungere la nuova sezione, alcuni moduli preesistenti hanno subito delle modifiche, in particolare *App.js* e *index.js*. Il primo è il modulo principale del frontend di XTap, contenente le operazioni di decodifica del token JWT ricevuto dopo la fase di autenticazione di un utente e la dichiarazione di un componente <BrowserRouter>, in cui vengono specificate tutte le route disponibili e che consente la navigazione tra le varie pagine che costituiscono il frontend. Questo componente viene dichiarato all'interno del context provider AuthContext, che fornisce i dati di autenticazione presenti nel token JWT a tutte le route dichiarate.

```
1     <AuthContext.Provider value={authentication}>
2     ...
3         <BrowserRouter>
4         <CssBaseline/>
5         <Layout>
6     ...
7         <PrivateRoute path="/AuthRequests" isAuthenticated=
8             {isAuthenticated} component={AuthRequests}/>
9         <PrivateRoute path="/Devices" isAuthenticated=
10            {isAuthenticated} component={Devices}/>
11        <PrivateRoute path="/DeviceControlPanel" isAuthenticated=
12            {isAuthenticated} component={DeviceControlPanel}/>
```

```
13          <PrivateRoute path="/Certificates" isAuthenticated=
14              {isAuthenticated} component={Certificates}/>
15      ...
16          </BrowserRouter>
17          </ThemeContext.Provider>
18          </StateProvider>
19      </AuthContext.Provider>
```

Listing 4.11: Struttura di BrowserRouter e AuthContext in App.js

Il secondo modulo invece è quello contenente il layout della homepage di XTap, in cui le sezioni disponibili sono state disposte all'interno di un componente <Accordion>, un menù "a fisarmonica" annidata, che contiene altri Accordion in cui sono presenti i collegamenti alle pagine che costituiscono le sottosezioni. Per aggiungere la sezione Device Admin e tutte le sue sottosezioni, le modifiche di questi due file hanno riguardato l'aggiunta delle nuove route all'interno del componente <BrowserRouter> in App.js e l'inserimento dei collegamenti alle pagine delle sottosezione di Device Admin all'interno del componente Accordion in index.js. Oltre a queste modifiche sono stati aggiunti dei moduli che vedremo singolarmente nel corso di questa sezione.

4.3.1 Visualizzazione della sezione *Device Admin*

La sezione Device Admin contiene strumenti accessibili soltanto da utenti che abbiano i diritti di Device Admin, per questo motivo è stato necessario trovare un modo per limitare la visibilità della sezione solo agli utenti autorizzati. La lista dei diritti degli utenti è disponibile all'interno del token JWT decodificato in App.js, il cui contenuto, come detto poche righe sopra, viene salvato all'interno del context provider AuthContext. Se tra i diritti dello user è presente quello di Device Admin, in AuthContext viene salvata la variabile isAdmin settata a True, viceversa viene settata a False. Tramite l'hook useContext(), AuthContext e quindi la variabile isAdmin vengono resi disponibili in index.js, in modo tale da mostrare l'Accordion contenente la sezione Device Admin solo nel caso in cui isAdmin fosse settata a True.

```
1   const authContext = useContext(AuthContext);
2   ...
3   {
4       authContext.isAdmin &&
5       <Accordion expanded={state.drawerSectionsExp.deviceAdmin}
6   ...
7       </Accordion>
8   }
```

Listing 4.12: Gestione della visualizzazione della sezione Device Admin in index.js

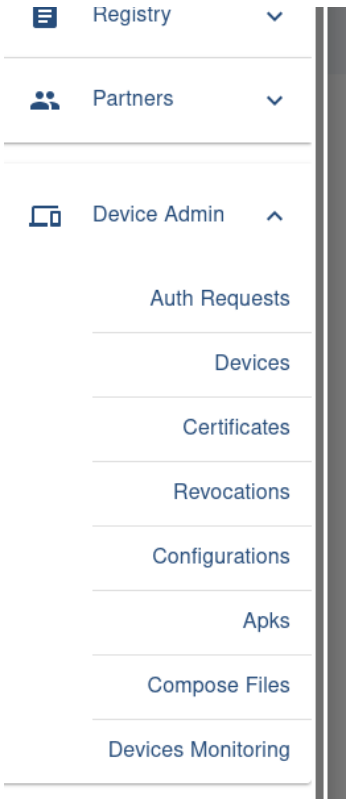

Figura 4.1: Accordion contenente la sezione Device Admin.

4.3.2 Descrizione delle sottosezioni

Le sottosezioni presenti in Device Admin, ad esclusione di Device Monitoring, sono accomunate dalla stessa struttura. All'interno di ogni sottosezione infatti viene eseguita una richiesta alle API del server attraverso *axios* [30], un componente che permette di effettuare richieste HTTP, in modo tale da richiedere la lista delle risorse a cui la sottosezione è dedicata. Se la lista viene ricevuta correttamente, gli elementi vengono visualizzati all'interno di una una MaterialTable, un componente che permette di costruire una datatable in stile Material Design. Per le MaterialTable delle varie sottosezioni vengono specificati dei props, tra i quali i più importanti sono:

- *columns* - In cui specificare la lista delle colonne della tabella con la loro composizione.

- *data* - In cui immettere i dati ricevuti dal server a fronte della richiesta dell'elenco degli oggetti contenuti in una risorsa.
- *onRowClick* - In cui specificare l'operazione da effettuare nel momento in cui viene effettuato un click su un elemento della lista.
- *actions* - Introduce una colonna aggiuntiva nella MaterialTable in cui poter inserire una serie di operazioni da effettuare sulla singola risorsa.

Sono poi presenti anche altri props in cui vengono specificate informazioni che permettono la paginazione dell'elenco delle risorse.

Nel contesto delle operazioni eseguite sulle risorse, molto spesso viene usato un dialog di conferma o di inserimento di informazioni aggiuntive ai fini del completamento dell'operazione. I dialog in questione sono solitamente costituiti dalla stessa struttura, ovvero da un componente di tipo <Dialog> in cui sono presenti dei bottoni che scatenano delle richieste verso il server REST e che restituiscono l'esito dell'operazione in un componente di tipo <SnackBar>, una finestra di notifica temporanea.

```
...
function deleteConfiguration() {
    axios.delete("api/configurations/"+props.configurationId)
        .then((res) => {
            console.log('res', res)
            props.onClose()
            handleSuccessClick()
            props.afterDelete()
        })
        .catch((err) => {
            console.log('err', err)
            props.onClose()
            handleErrorClick()
        })
}

return (
    <div>
        <Dialog
            open={props.open}
            onClose={props.onClose}
            aria-labelledby="alert-dialog-title"
            aria-describedby="alert-dialog-description"
```

```
24              >
25      ...
26              <Snackbar open={successOpen} autoHideDuration={6000}
    onClose={handleSuccessClose}>
27                  <Alert onClose={handleSuccessClose} severity="
    success">
28                      Operation done successfully
29                  </Alert>
30              </Snackbar>
31              <Snackbar open={errorOpen} autoHideDuration={6000}
    onClose={handleErrorClose}>
32                  <Alert onClose={handleErrorClose} severity="error">
33                      Error during operation
34                  </Alert>
35              </Snackbar>
36          </div>
37      );
38      ...
```

Listing 4.13: Esempio di dialog per l'eliminazione di una configurazione

Visti gli elementi comuni, vediamo adesso quelli che differiscono prendendo in esame ogni singola sottosezione.

Auth Requests

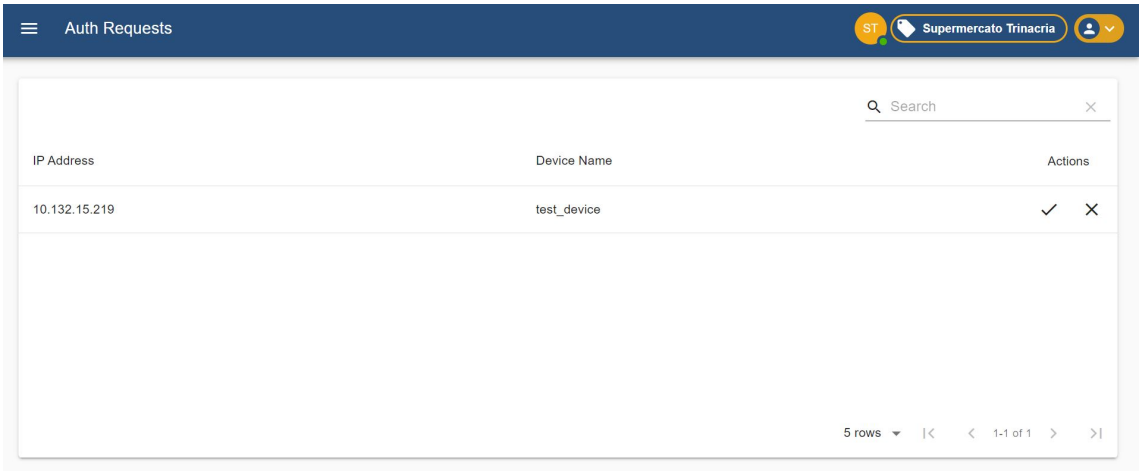

Figura 4.2: Lista delle richieste di autenticazione.

In questa sottosezione vengono mostrate le richieste di autenticazione effettuate al server da parte dei dispositivi. La lista delle richieste all'interno della MaterialTable viene refreshata ogni 20 secondi e vengono visualizzati indirizzo Ip e nome del dispositivo, con la possibilità di accettare o rifiutare singolarmente ogni richiesta attraverso le actions disponibili. Quando si effettua un click su una delle azioni disponibili, accettazione o rifiuto della richiesta in questo caso, viene aperto un dialog di conferma a cui viene comunicato, tramite il prop *operation*, il tipo di operazione effettuata. Confermando l'operazione quindi il client effettuerà una richiesta PATCH accettando o rifiutando la richiesta di autenticazione.

Devices

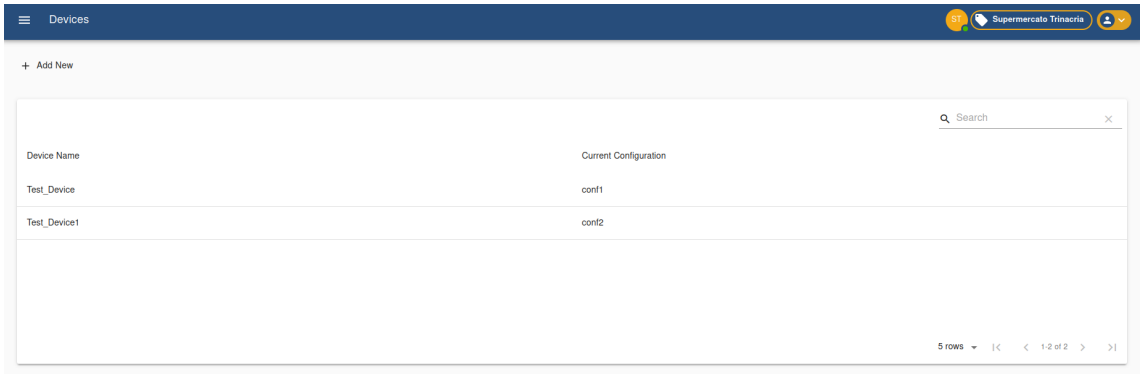

Figura 4.3: Lista dei dispositivi.

Qui viene mostrata la lista dei Device visualizzando per ognuno di essi nome e configurazione corrente, con la possibilità di crearne uno nuovo cliccando su *Add New*. In questo modo viene aperto un dialog in cui inserire il nome del dispositivo, scegliere la configurazione attraverso un menù a tendina ricavato tramite un componente di tipo <Select> e decidere attraverso un interruttore se attivare o meno l'autenticazione manuale per il Device da creare. Immessi i dati, cliccando *Create* viene inviata una richiesta POST all'endpoint per la creazione del Device, il quale a questo punto comparirà nella lista. Cliccando su uno degli elementi della lista è possibile accedere al pannello di controllo del Device che offre una serie di operazioni, ovvero:

- **Change configuration**: apre un dialog in cui è presente un componente <Select> che consente di scegliere una configurazione da una lista di quelle disponibili e una volta scelta invia una richiesta PATCH all'endpoint */api/devices/update_configuration/{device_id}*.

- **Change certificate**: apre un dialog di conferma e invia una richiesta GET all'endpoint */api/devices/{device_id}/change_certificate/*.

- **Configuration**: apre la pagina della configurazione associata al Device.

- **Certificate**: apre la pagina del certificato associato al Device.

- **Manual Authentication**: invia una richiesta PATCH all'endpoint */api/-devices/update_manual_authorization/{device_id}* per attivare o disattivare l'autenticazione manuale.

- **Delete Device**: apre un dialog di conferma e invia una richiesta DELETE all'endpoint *api/devices/{device_id}*.

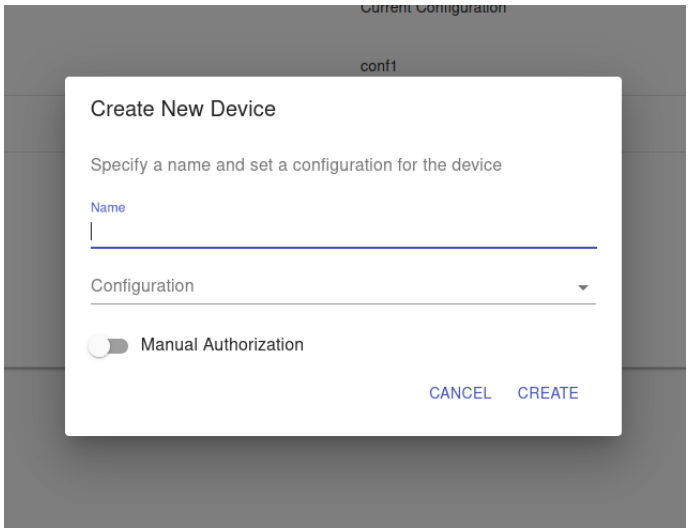

Figura 4.4: Dialog di creazione di un nuovo Device.

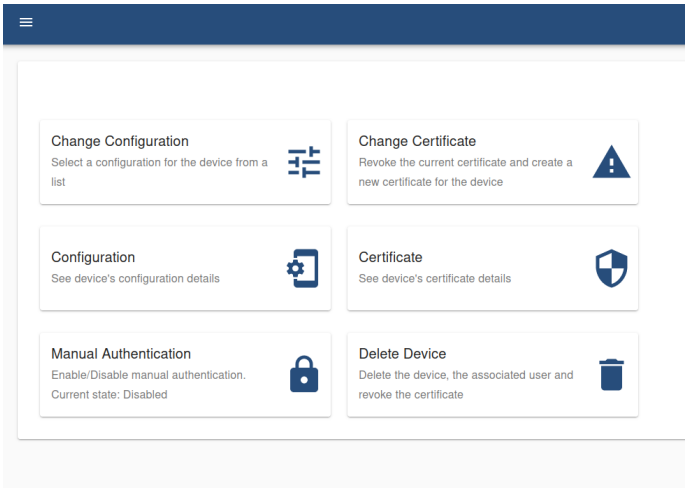

Figura 4.5: Pannello di controllo del dispositivo.

Certificates

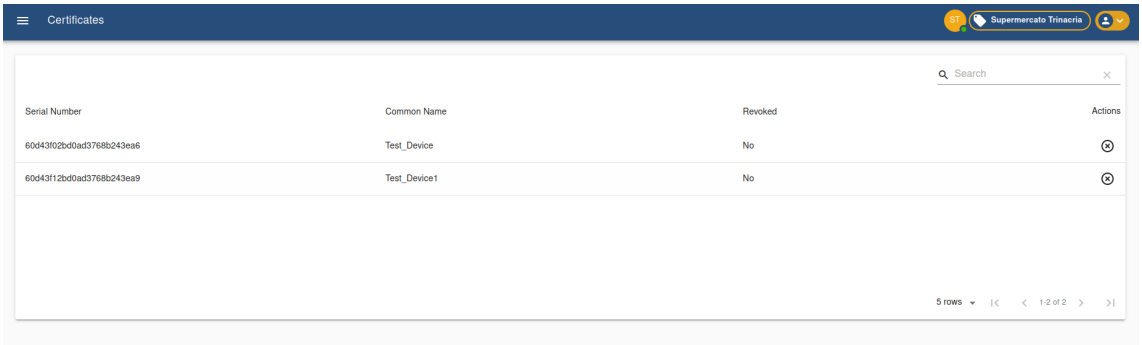

Figura 4.6: Lista dei certificati dei dispositivi.

Quando un Device viene creato, il suo certificato viene inserito nella lista presente in questa sottosezione specificandone il numero seriale, il common name e se sia stato revocato o meno, con la possibilità di revocare qualunque dei certificati presenti nella lista tramite l'apposita action, la quale apre un dialog in cui specificare la ragione di revoca e scatena una richiesta verso il server. Cliccando su un elemento della lista si accede alla pagina contenente le informazioni specifiche del certificato, il quale viene visualizzato in formato PEM in una textarea corredata da un bottone per copiarne il contenuto nella clipboard.

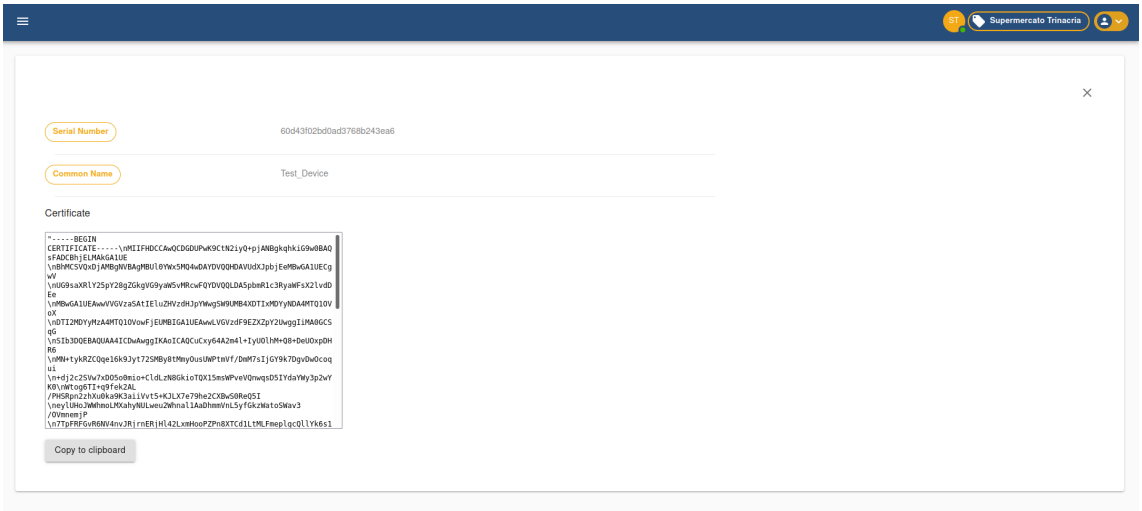

Figura 4.7: Pagina dei dettagli di un certificato.

Revocations

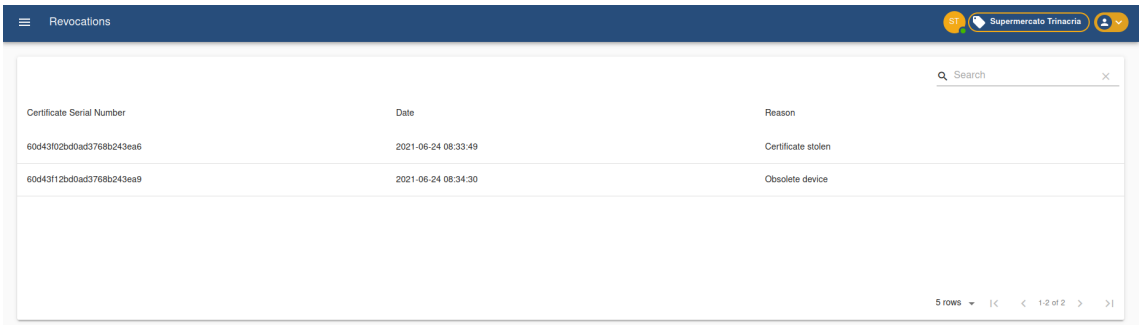

Figura 4.8: Lista delle revoche dei certificati.

I certificati revocati vengono visualizzati in questa sotto-sezione, per ognuno dei quali vengono mostrati il numero seriale, data e ragione di revoca.

Configurations

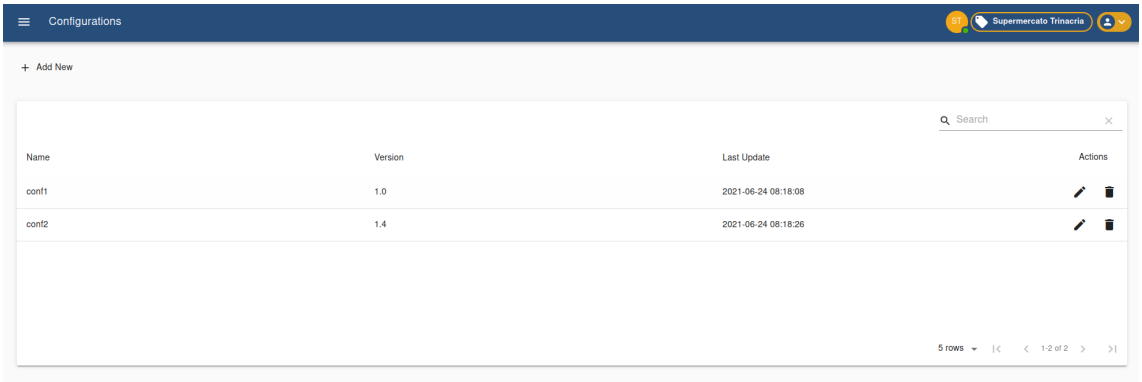

Figura 4.9: Lista delle configurazioni.

Questa sotto-sezione permette la visualizzazione delle configurazioni disponibili specificando per ognuna di esse il nome, la versione e la data di ultimo aggiornamento. Come per i Device è presente un bottone per la creazione, tramite il quale viene effettuata una navigazione verso un form in cui inserire nome, versione e il JSON contenente la configurazione vera e propria. Quest'ultima va inserita in una textarea che effettua la validazione del contenuto, infatti è possibile procedere con la creazione della configurazione soltanto inserendo un JSON valido. Inseriti correttamente tutti i campi, vi è una schermata di riepilogo da cui controllare le informazioni immesse prima di effettuare la richiesta al server.

Per gli elementi presenti nella MaterialTable sono disponibili delle actions che permettono, per ogni configurazione, di effettuarne la cancellazione, preceduta da un dialog di conferma, e la modifica, molto simile strutturalmente alla creazione e da cui differisce soltanto per il fatto che il form viene compilato con le informazioni preesistenti e il nome della configurazione non può essere modificato. Cliccando su uno degli elementi della MaterialTable, si accede alle informazioni della configurazione come per i certificati.

Figura 4.10: Form di modifica/creazione di una configurazione.

Apks e Compose Files

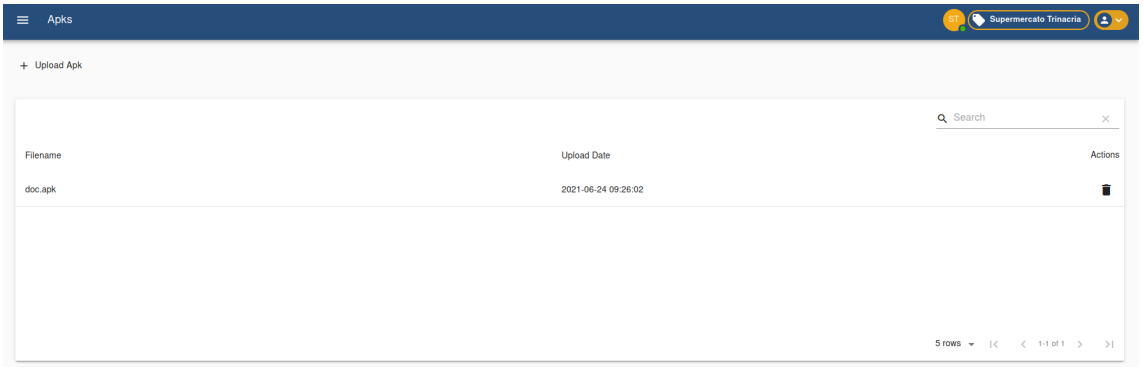

Figura 4.11: Lista degli Apk.

Entrambe le sottosezioni presentano la lista degli elementi, ognuno dei quali provvisto di azione per la cancellazione del file dal server. Per poter caricare un Apk o un file Compose è disponibile un bottone per mezzo del quale viene aperto un dialog contenente un uploader che effettua il caricamento dei file attraverso un componente di tipo <input>. L'uploader è uguale in entrambe le sottosezioni, a differire sono soltanto il Media-Type del file accettato e l'endpoint a cui effettuare l'upload.

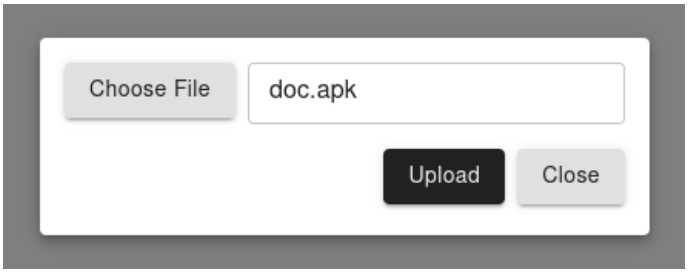

Figura 4.12: Uploader di Apk/file Compose.

Device Monitoring

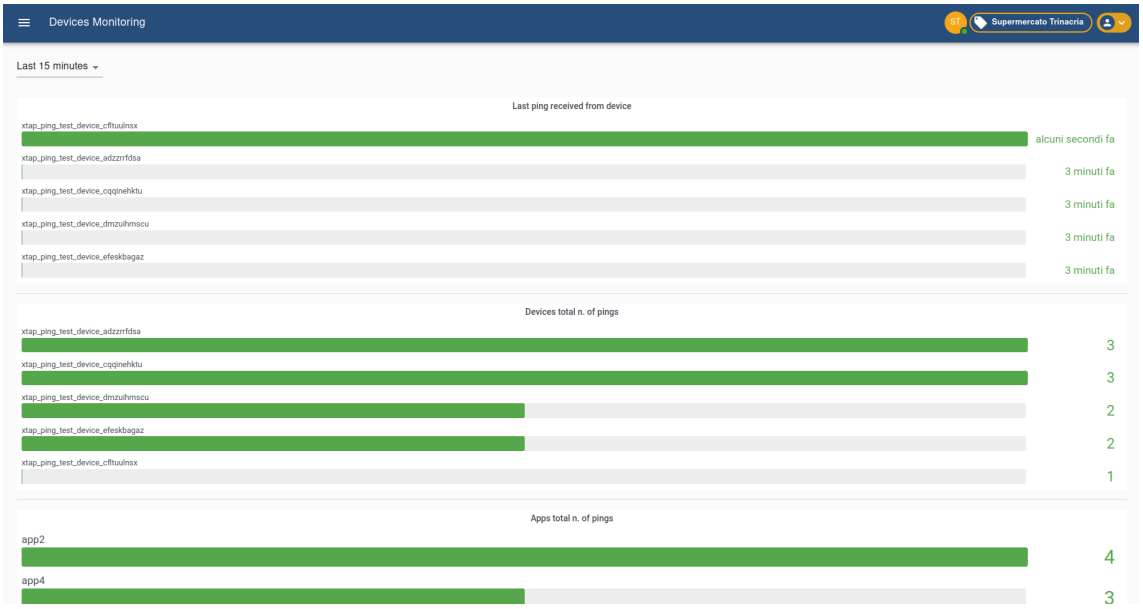

Figura 4.13: Pannelli contenenti le statistiche dei dispositivi e delle applicazioni.

Le informazioni per il monitoring dei Device e delle applicazioni vengono visualizzate qui grazie a dei componenti <iframe> che consentono l'embed dei grafici di Grafana tramite URL, con la possibilità di scegliere la finestra temporale di visualizzazione dei dati da una lista contenuta in un componente <Select>. Scegliendo una finestra temporale, l'URL del grafico viene modificato per soddisfare la richiesta e il componente <iframe> viene aggiornato.

4.4 Sviluppo del blocco di monitoring

4.4.1 Esposizione delle metriche

Per controllare il comportamento dei servizi che compongono XTap è necessario che questi espongano delle metriche in un formato leggibile da Prometheus. Nel momento in cui XTap viene avviato in modalità *monitor*, nella rete interna di Docker vengono resi disponibili al prelievo delle metriche questi tre endpoint

- backend:8000/monitoring/metrics
- faust:6066/metrics
- mongo-query-exporter:2205/metrics

che andremo singolarmente a descrivere nel dettaglio per vedere come questi effettuino la creazione e l'aggiornamento delle metriche.

backend:8000

Dall'endpoint esposto dal servizio contenente il server REST vengono prelevate le metriche riguardanti l'utilizzo delle API. Qui il monitoring viene effettuato attraverso un middleware che consente di ispezionare ogni singola richiesta in arrivo al server. Sono diversi i parametri tenuti in considerazione, ma quelli di maggiore importanza sono costituiti dal numero di richieste ricevute, gestito da un oggetto di tipo Counter, e dal tempo di processamento di ogni richiesta, gestito da un oggetto di tipo Histogram. Viene anche eseguita una categorizzazione di questi parametri grazie all'inserimento di alcune label che dividono le richieste per metodo utilizzato, URL associato, nome dell'utente e dell'azienda che le effettua. Per ogni richiesta in arrivo, il token JWT contenuto, se presente, viene decodificato e successivamente salvato in una cache per evitare che richieste provenienti dallo stesso utente subiscano ogni volta l'oneroso processo di decodifica del token. Prelevate da questo le informazioni necessarie, viene salvato il timestamp corrente in una variabile e la richiesta viene passata al server. Processata la richiesta viene nuovamente salvato il timestamp corrente in un'altra variabile e infine le metriche vengono aggiornate tramite i metodi offerti dagli oggetti Counter e Histogram: per il primo viene chiamato il metodo *inc()* che incrementa di 1 la metrica associata, per il secondo viene invocato il metodo *observe()* che tiene traccia del tempo di processamento della richiesta passando come parametro la differenza dei timestamp precedentemente memorizzati.

```
1    ...
```

```
2         if "authorization" in request.headers:
3             if request.headers["authorization"].split(" ")[0] == "
    Bearer":
4                 token = request.headers["authorization"].split(" ")
    [1]
5                 payload = cache.get(token)
6                 if payload is None:
7                     payload = jwt.decode(token, settings.
    AUTH_SECRET_KEY, algorithms=[settings.AUTH_ALGORITHM])
8                 cache[token] = payload
9                 owner = payload.get("company_prefix")
10                user = payload.get("sub")
11     ...
12     REQUESTS.labels(method=method, path_template=path_template,
    owner=owner, user=user).inc()
13     try:
14         before_time = time.perf_counter()
15         response = await call_next(request)
16         after_time = time.perf_counter()
17     ...
18         REQUESTS_PROCESSING_TIME.labels(method=method,
    path_template=path_template, owner=owner, user=user).observe((
    after_time - before_time))
19     ...
```

Listing 4.14: Aggiornamento delle metriche nel middleware

faust:6066

Tramite Faust vengono monitorati sia gli eventi EPCIS inviati al server, sia i ping e gli heartbeat dei dispositivi, le cui metriche vengono rese disponibili a Prometheus tramite un sensore interno. Per quanto riguarda gli eventi, le metriche prese in considerazione sono il numero di eventi totali, catturati tramite un oggetto di tipo Counter, e il tempo di elaborazione di ogni evento, tramite un oggetto di tipo Histogram, entrambi caratterizzati da label in cui specificare il dominio dell'azienda dello user che li ha generati. Le metriche vengono dichiarate all'interno del context manager OwnerMonitor, un oggetto che presenta un metodo *___enter___*, in cui viene memorizzato il timestamp corrente, e un metodo *___exit___*, in cui viene memorizzato di nuovo il timestamp corrente e vengono aggiornate le metriche con le operazioni di *inc()* e *observe()* allo stesso modo di quanto avviene nel backend. OwnerMonitor viene chiamato all'interno del già esistente agent per la gestione degli

eventi nel contesto di una clausola *with*, in modo tale da eseguire automaticamente ___*enter*___ all'entrata e ___*exit*___ all'uscita di with.

```
class OwnerMonitor:
def __init__(self, owner: str):
    self.owner = owner

def __enter__(self):
    self.t0 = time.time()
    return self

def __exit__(self, exc_type, exc_value, exc_traceback):
    capture_events_total.labels(owner=self.owner).inc()
    capture_events_runtime_ms.labels(owner=self.owner).observe(
        (time.time() - self.t0)
    )
```

Listing 4.15: Definizione di OwnerMonitor

Per i ping invece si sono rese necessarie alcune modifiche: precedentemente Faust veniva avviato come modulo standalone, quindi tutti gli agent disponibili erano contenuti in un unico file. Per avere la suddivisione tra gestione degli eventi e gestione dei ping e rendere possibile l'aggiunta di altri eventuali moduli Faust in modo semplice e senza modificare quelli preesistenti, gli agent sono stati divisi in file differenti contenuti in moduli Python distinti secondo questa gerarchia

```
+ xtap/
    - __init__.py
    - __main__.py
    - app.py

    + events/
        -   __init__.py
        -   agents.py
        -   models.py
        -   ...

    + pings/
        -   __init__.py
        -   agents.py
        -   models.py
```

```
-   prometheus_wrappers.py
-   ...
```

con il file *app.py* contenente l'applicazione Faust con la funzione di autodiscovery dei moduli attiva.

La catena per il monitoraggio dei ping, descritta nel capitolo 3.1.4, parte dalla API */monitoring/ping/* a cui i ping vengono inviati tramite richiesta POST contenente nel body un JSON del tipo

```
{
    "type": <tipologia_di_ping>
    "app_name": <nome_dell_app>
}
```

dove type può assumere solo i valori:

- 0 per segnalare l'accensione del dispositivo (da implementare).
- 1 ping di avvio di un'applicazione.
- 2 heartbeat del dispositivo.

Gli oggetti che gestiscono le metriche dei ping sono contenuti nel file *prometheus_wrappers()*: all'interno di questo sono infatti presenti tre Counter, che contano il numero di volte che un'applicazione è stata aperta e il numero di ping totali inviati da una singola applicazione o da un singolo Device, e un Gauge, in cui viene salvato il timestamp dell'ultimo heartbeat ricevuto da un Device. Per ogni metrica vengono specificate delle label contenenti il nome dell'applicazione, del dispositivo e il dominio dell'azienda di quest'ultimo. Arrivati a Faust, i ping vengono letti da un agent e le metriche vengono aggiornate a seconda della tipologia di ping che è stato ricevuto.

```
1     @app.agent(ping_topic)
2     async def ping_process(stream):
3         async for ping in stream:
4             opt_type, event_id, doc = parseConnectorItem(ping)
5             if opt_type == MongoOperationType.INSERT:
6                 if doc["type"] == PingTypeEnum.booting:
7                     return  # TODO
8                 elif doc["type"] == PingTypeEnum.starting_app:
9                     APP_OPENED_COUNTER.labels(app_name=doc["
    app_name"], owner=doc["object_owner"]).inc()
10                elif doc["type"] == PingTypeEnum.heartbeat:
```

```
11                      APP_PINGS_COUNTER.labels(app_name=doc["app_name
    "], owner=doc["object_owner"]).inc()
12                      DEVICE_PINGS_COUNTER.labels(device_name=doc["
    device_name"], owner=doc["object_owner"]).inc()
13                      DEVICE_LAST_PING_GAUGE.labels(device_name=doc["
    device_name"], owner=doc["object_owner"]).set(int(doc["timestamp
    "]))
```

Listing 4.16: Agent Faust per il processamento delle risorse Ping.

mongo-query-exporter:2205

mongo-query-exporter è un esportatore di metriche basato su uno script Python contenente un piccolo server HTTP che consente il monitoring di MongoDB attraverso la lettura della collection *system.profile*, contenente i metadati di ogni operazione effettuata sul database. La collezione viene letta una volta al secondo e nel momento in cui un'operazione sul db viene effettuata l'exporter ne legge i metadati, estraendone le informazioni necessarie all'aggiornamento delle metriche. Queste vengono rese disponibili mediante degli Histogram che monitorano, per ogni operazione, il numero di millisecondi di durata di un'operazione, di indici esaminati e di documenti ritornati. Servendosi di un oggetto di tipo HTTPServer, offerto dalla libreria Python di Prometheus, le metriche vengono automaticamente esposte alla porta specificata, 2205 in questo caso. L'exporter è stato quindi dockerizzato, creando un Dockerfile contenente un'immagine Linux che avvia lo script Python e che viene fatta partire esponendo la porta scelta all'interno della rete locale Docker.

4.4.2 Configurazione di Prometheus e provisioning di Grafana

Prima di far partire le immagini Docker è necessario configurare Prometheus e Grafana in modo tale da consentire la corretta gestione dei dati dei servizi da monitorare. Prometheus legge la sua configurazione dal file *prometheus.yaml*, il quale viene copiato nel container durante la fase di build delle immagini Docker. In questo file vengono specificati gli intervalli temporali di lettura delle metriche, i singoli job, descritti precedentemente, definendone l'endpoint da cui prelevare le metriche e l'URL dello storage da cui effettuare lettura e scrittura di queste, il quale nello specifico è quello del connector di Promscale. Anche Grafana necessita di alcuni file da copiare in fase di build dell'immagine Docker per il provisioning delle data source e delle dashboard. In particolare i file sono:

- *dashboard.yaml*: vengono specificate le cartelle da cui prelevare i file JSON contenenti le dashboard.

- *dashboard.json*: file JSON contenente una dashboard per il monitoring generale dei servizi di XTap.

- *datasource.yaml*: viene inserita la lista delle sorgenti di dati che nel nostro caso contiene solo il connector di Promscale.

- *grafana.ini*: contiene impostazioni generali per Grafana come ad esempio l'attivazione dell'autenticazione anonima, che consente l'embedding e la visualizzazione di alcuni grafici agli utenti non autenticati.

4.4.3 Dashboards

La visualizzazione dei dati raccolti ed elaborati da Prometheus avviene per mezzo di grafici contenuti nei pannelli messi a disposizione da Grafana e raggruppati in dashboards, le quali possono essere divise in due tipologie: dashboard di base e dashboard per il monitoring dei Device.

Dashboard di base

Questa dashboard contiene pannelli per la visualizzazione di informazioni generiche riguardo allo stato dei servizi di XTap ed è normalmente disponibile fin dall'avvio del blocco di monitoring. Si accede a questa direttamente dall'endpoint di Grafana loggandosi con credenziali di admin e selezionandola dalla lista delle dashboard disponibili col nome di *xtap basic dashboard*. Le informazioni accessibili tramite i pannelli riguardano il tempo di processamento delle API di XTap e delle query a MongoDB, il numero di chiavi esaminate e di documenti ritornati da quest'ultimo, la quantità e il tempo di runtime degli eventi EPCIS e il numero di ping ricevuti dai dispositivi.

Figura 4.14: Schermata della dashboard di base visualizzabile in Grafana.

Dashboard per il monitoring dei Device

Nel momento in cui un'azienda viene aggiunta ad XTap, per ogni suo dominio viene creata una dashboard contenente dei pannelli con le informazioni sull'uso dei dispositivi e delle applicazioni. La creazione delle dashboard viene effettuata tramite le API di Grafana per mezzo di una richiesta POST all'endpoint */api/dashboards/db* contenente i parametri grafici e le query PromQL da effettuare a Prometheus e ricevendo in risposta l'identificativo della dashboard appena creata. Questo viene utilizzato per costruire l'URL di embedding dei pannelli e successivamente salvato su MongoDB tramite un oggetto GrafanaData, che contiene l'associazione dominio dell'azienda-URL della dashboard. Per visualizzare i pannelli di un'azienda occorre andare nella sezione *Device Admin* del frontend e cliccare su *Device Monitoring*: la pagina effettua una richiesta GET all'endpoint */api/grafana_data/get_data/* ottenendo l'URL dei pannelli precedentemente settato e usando quest'ultimo per la visualizzazione dei grafici come descritto nella sezione 4.3.2 Device Monitoring.

4.5 Sviluppo delle configurazioni Docker Compose

4.5.1 Organizzazione dei file Compose

Come già specificato precedentemente, XTap usa Docker Compose per il build e l'avvio dei servizi in ambiente multi-container. In precedenza, all'interno di XTap era presente un solo file Compose col quale avviare indistintamente tutti i servizi, senza una vera e propria possibilità di selezionare una piccola porzione di questi da esporre in locale per agevolare il testing, se non lanciando una serie di comandi aggiuntivi ed effettuando diverse operazioni. Ciò ha evidenziato, specialmente con la sopraggiunta dei servizi per il monitoring, la necessità di un sistema che consentisse di scegliere quali servizi far partire e/o quali di questi esporre in locale. Per raggiungere questo obiettivo ci si è avvalsi delle proprietà di merge e override dei file Compose: queste proprietà consentono di specificare, durante le operazioni effettuate tramite il tool da linea di comando *docker-compose*, una lista di file Compose di cui eseguire il merge nell'ordine in cui vengono specificati. In pratica, sfruttando queste proprietà, sono stati aggiunti questi file da specificare in quest'ordine:

1. docker-compose.yaml
2. docker-compose-dev.yaml
3. docker-compose-monitor.yaml
4. docker-compose-monitor-dev.yaml

Specificando uno o più di questi file si può avere il comportamento descritto nella sezione 3.3, più precisamente la modalità *base* verrà avviata specificando solo il primo file, *base-develop* i primi due, *monitor* i primi tre, *monitor-develop* tutti e quattro. I file Compose sono così articolati: docker-compose.yaml e docker-compose-monitor.yaml contengono rispettivamente le configurazioni per i servizi base di XTap e quelle per i servizi di monitoring; i Compose file con suffisso -dev contengono la lista di servizi con l'indicazione delle porte da esporre per essere raggiungibili da localhost.

4.5.2 Modifiche allo script *xtap-cli*

Tutte le operazioni concernenti i container Docker partono dallo script bash xtap-cli, modificato nel corso del tempo per essere uniformato alle feature aggiunte. A questo

è stata infatti aggiunta l'opzione *-m* con la quale specificare una modalità tra quelle precedentemente descritte. Le operazioni a partire da questo script saranno quindi eseguite lanciando un comando del tipo

```
./xtap-cli.sh -m {modalità} build|run
```

La scelta della modalità modifica una variabile interna allo script chiamata *LOCAL_CONF* che contiene la lista dei file Compose da specificare come argomento nel comando *docker-compose*, in modo tale da poter successivamente eseguire le operazioni di build e run dei container utilizzando all'interno dello script il comando *docker-compose* nel formato

```
docker-compose $LOCAL_CONF build|up
```

4.6 Sviluppo delle configurazioni Kubernetes

4.6.1 Conversione tramite Kompose

Il primo problema affrontato durante la migrazione dei servizi Docker sull'environment Kubernetes di Google è stata la creazione dei file YAML contenenti le configurazioni dei singoli pod a partire dai file Compose: questa operazione è stata parzialmente effettuata tramite il tool Kompose [31], per mezzo del quale è possibile effettuare esattamente una conversione di questo tipo. Kompose ha operato in questo modo:

- I container sono stati tutti trasformati in Deployment, per ognuno dei quali è stato creato un Service.
- Le variabili d'ambiente presenti in local.env sono state copiate in una ConfigMap.
- Per i container dotati di un volume, è stato creato anche un PersistentVolume associato al container.
- Per i container che espongono porte verso l'esterno il relativo Service è stato dotato di un LoadBalancer, che offre al container un indirizzo Ip esterno in fase di deployment.

Kompose tuttavia tiene conto soltanto di ciò che è presente all'interno dei file Compose, non considerando i Dockerfile di alcune immagini che necessitano di essere buildate. Per questo motivo in alcuni Deployment l'URL da cui prelevare l'immagine è costituito dal build tag dell'immagine Docker e in fase di run su Kubernetes questo non può essere altro che un problema nel momento in cui viene effettuato il pull-down

delle immagini. Per questo motivo, le immagini Docker in questione vanno buildate in locale e caricate su un container registry, che nel caso di XTap è quello di GKE, e il link di queste va inserito nello YAML del Deployment a cui fanno riferimento.

4.6.2 Automazione dei processi di upload e deploy

Avendo a disposizione gli YAML di tutti i servizi, effettuare il deployment su Kubernetes necessita, nel più semplice dei casi, dell'utilizzo di un solo comando kubectl, il tool da linea di comando di Kubernetes, ovvero:

```
kubectl apply -f <cartella contenente i file YAML>
```

Tuttavia nel nostro caso le operazioni da effettuare prima del deployment sono numerose e tengono conto di molteplici aspetti e per questo motivo, per automatizzare il processo, come già spiegato nel capitolo 3.4, sono state aggiunte allo script xtap-cli delle funzionalità per l'upload e il deployment dei servizi su Kubernetes tramite due nuovi comandi: *push-images* e *run-on-k8s*. Col primo comando le immagini dei servizi di XTap, selezionati a seconda della modalità specificata, vengono pushate verso il container registry, ma prima di effettuare la vera e propria operazione di push le immagini vengono buildate e viene modificato il build tag di queste con il link a cui effettuare il caricamento delle immagini, prelevando automaticamente l'Id del progetto, necessario nella costruzione del link, e modificando gli YAML dei Deployment delle immagini coi nuovi link appena costruiti.

```
1    PROJECT_ID=$(gcloud config get-value project 2> /dev/null)
2    ...
3    sed -i "s/image:.*/image: gcr.io\/$PROJECT_ID\/xtap_faust:
     $BUILD_TAG/" faust-deployment.yaml
4    docker tag xtap_faust:$BUILD_TAG gcr.io/$PROJECT_ID/xtap_faust:
     $BUILD_TAG
5    docker push gcr.io/$PROJECT_ID/xtap_faust:$BUILD_TAG
6    ...
```

Listing 4.17: Push delle immagini verso il container registry

Tramite *run-on-k8s* viene effettuato il deployment delle configurazioni Kubernetes su GKE e a seconda della modalità scelta le operazioni eseguite differiscono: scegliendo la modalità base, viene prima di tutto creata la ConfigMap contenente le variabili d'ambiente a partire dal file local.env e successivamente vengono caricate tutte le configurazioni dei container base di XTap su GKE. Con la modalità monitor, effettuate queste stesse operazioni ma soltanto sulle immagini che compongono il blocco di monitoring, lo script aspetta che il service di Grafana, di tipo LoadBalancer,

finisca di configurarsi ed esponga l'indirizzo Ip esterno in modo tale da inserirlo nella ConfigMap in aggiunta alle altre variabili d'ambiente presenti nel file local.env, poiché è necessario per la visualizzazione dei pannelli con le statistiche dei dispositivi nel frontend. Successivamente vengono anche caricate le configurazioni delle immagini base.

```
1    ...
2    sed -i 's/USE_PROMETHEUS=.*/USE_PROMETHEUS=1/' $ENV
3    kubectl create configmap local-env --from-env-file=$ENV
4    cd ./kubernetes
5    kubectl apply -f monitor
6    cd ..
7
8    GRAFANA_HOST=$(kubectl get svc grafana -o jsonpath='{.status.
   loadBalancer.ingress[0].ip}')
9
10   while [[ -z $GRAFANA_HOST ]]
11   do
12       echo "Waiting for Grafana external Ip..."
13       GRAFANA_HOST=$(kubectl get svc grafana -o jsonpath='{.
   status.loadBalancer.ingress[0].ip}')
14       sleep 5
15   done
16   GRAFANA_IP=$(kubectl get svc grafana -o jsonpath='{.status.
   loadBalancer.ingress[0].ip}:{.spec.ports[0].targetPort}')
17   sed -i "s/GRAFANA_EXTERNAL_HOST=.*/GRAFANA_EXTERNAL_HOST=
   $GRAFANA_IP/" $ENV
18   kubectl create configmap local-env --from-env-file=$ENV --dry-
   run=client -o yaml | kubectl apply -f -
19   cd ./kubernetes
20   kubectl apply -f base
21   ...
```

Listing 4.18: Deployment delle configurazioni Kubernetes su GKE

Capitolo 5

Conclusioni

Sebbene tutte le funzionalità descritte in questa tesi siano state fondamentalmente sviluppate nell'ottica di XTap, la soluzione proposta è valida anche per progetti di altra natura. I dispositivi supportati infatti sono di tipo *general purpose*, la loro funzione e il loro scopo finale possono andare oltre il concetto di Smart Logistics ed essere di qualunque tipo, purché vengano forniti i mezzi necessari per effettuare le operazioni di autenticazione, auto-configurazione e monitoraggio. Discorso simile può essere fatto per i servizi di monitoring: Prometheus, Grafana e TimescaleDB nella soluzione con Promscale rappresentano di fatto uno standard per il monitoring dei servizi e lo stesso schema descritto in questa tesi può essere utilizzato per ogni progetto.

Per quanto riguarda XTap invece possiamo dire che, grazie al lavoro svolto in questa tesi, è stato fatto un passo in avanti nella trasformazione di XTap da *progetto* a *prodotto*. Le funzionalità aggiunte sono state infatti create mantenendo un mindset che tenesse conto sia degli utenti finali, cercando di rendere l'utilizzo dei servizi di XTap semplice ed intuitivo, sia degli sviluppatori, semplificando le operazioni di avvio, controllo e deployment della piattaforma. L'aggiunta dei servizi per la gestione e autenticazione dei Device ad esempio ha aggiunto un livello di praticità ad XTap: se prima l'invio di eventi EPCIS poteva essere effettuato solo a livello di test attraverso script o direttamente dal browser usando la pagina Swagger messa a disposizione da FastAPI, adesso questa stessa operazione può essere effettuata da un vero dispositivo contenente, ad esempio, un lettore RFID, dando la possibilità a questo, una volta munito di un certificato X.509 valido, di autenticarsi e operare sulla piattaforma in modo strutturato e controllato. Anche l'aggiunta del blocco di monitoring ha portato diversi vantaggi, legati sopratutto alla fase di sviluppo del server. L'accoppiata Prometheus-Grafana permette infatti il controllo completo di tutte le funzionalità di

XTap, consentendo di effettuare un'analisi del comportamento del server, stimare il carico computazionale delle singole API e dei vari servizi e cercare soluzioni per alleggerire o potenziare il sistema. La creazione delle configurazioni Kubernetes e l'automazione del deployment hanno invece reso possibile l'avvio di XTap in un ambiente che sia più simile a quello di produzione piuttosto che a quello di sviluppo, portando alla luce nuove tipologie di problematiche ma anche di prospettive per lo sviluppo di nuove funzionalità che gestiscano i carichi di lavoro dei servizi presenti all'interno dei container.

5.1 Lavoro futuro

Quanto aggiunto ad XTap ha aperto le porte allo sviluppo di nuove funzionalità e al potenziamento di altre. L'aggiunta dei Device e di tutto quello che è a loro correlato, ad esempio, rende necessario lo sviluppo di un'architettura lato dispositivo che consenta di utilizzare le API messe a disposizione. Una grossa fetta del lavoro futuro è poi rappresentata dalle applicazioni Android e delle immagini Docker contenute nelle configurazioni dei dispositivi e che consentono a questi di effettuare operazioni sulle risorse e sugli eventi EPCIS. Lato monitoring si ha invece la possibilità di aggiungere a Prometheus dei nuovi esportatori di metriche per consentire il monitoraggio di altri aspetti, come ad esempio la memoria occupata dalle collezioni di MongoDB, eventualmente divisa per azienda, o lo stato dei container di Kubernetes/Docker. Altro aspetto interessante è costituito dalla possibilità di avere su Kubernetes più repliche della stessa istanza di un servizio di XTap. Questo può portare grandi vantaggi sopratutto prendendo in considerazione MongoDB e Kafka, poiché avendo più repliche aumenta sia la reliability dei dati memorizzati e sia, per quanto riguarda Kafka, la possibilità di gestire i topic in modo parallelo tra i vari consumer.

Bibliografia

[1] *Industria 4.0*, URL: https://it.wikipedia.org/wiki/Industria_4.0.

[2] G. Salvadori, *L'evoluzione dell'Industrial IoT in Italia: un confronto tra PMI e grandi imprese*, 2020, URL: https://blog.osservatori.net/it_it/industrial-iot-in-italia-confronto-pmi-grandi-imprese.

[3] J. Condemi, *Smart logistics: cos'è e quali sono le tecnologie della logistica 4.0*, 2021, URL: https://www.internet4things.it/iot-library/smart-logistics-cose-e-quali-sono-le-tecnologie-della-logistica-4-0/.

[4] *EPCIS and CBV Implementation Guideline*, 2017, URL: https://www.gs1.org/docs/epc/EPCIS_Guideline.pdf.

[5] *XTap*, URL: https://xeliontech.com/it/XTAP.

[6] *Docker*, URL: https://docs.docker.com/get-started/overview/.

[7] *Docker Compose*, URL: https://docs.docker.com/compose/.

[8] *Representational State Transfer*, URL: https://it.wikipedia.org/wiki/Representational_State_Transfer.

[9] *FastAPI*, URL: https://fastapi.tiangolo.com/.

[10] *X.509*, URL: https://it.wikipedia.org/wiki/X.509.

[11] *React*, URL: https://it.reactjs.org/.

[12] *MongoDB*, URL: https://www.mongodb.com/.

[13] *MongoDB CRUD*, URL: https://docs.mongodb.com/manual/crud/.

[14] *Kafka*, URL: https://kafka.apache.org/.

[15] *Zookeeper*, URL: https://zookeeper.apache.org/.

[16] *Faust*, URL: https://github.com/robinhood/faust.

[17] *Tasks, Timers, Cron Jobs, Web Views, and CLI Commands*, URL: https://faust.readthedocs.io/en/latest/userguide/tasks.html.

[18] *Prometheus*, URL: https://prometheus.io/.

[19] *Grafana*, URL: https://grafana.com/.

[20] *Kubernetes*, URL: https://kubernetes.io/.

[21] *I componenti di Kubernetes*, URL: https://kubernetes.io/it/docs/concepts/overview/components/.

[22] *MongoDB Kafka Connector*, URL: https://docs.mongodb.com/kafka-connector/current/.
[23] *Promscale*, URL: https://github.com/timescale/promscale.
[24] *TimescaleDB*, URL: https://www.timescale.com/.
[25] *Google Kubernetes Engine*, URL: https://cloud.google.com/kubernetes-engine.
[26] *Fernet*, URL: https://cryptography.io/en/latest/fernet/.
[27] *ExpiringDict*, URL: https://pypi.org/project/expiring-dict/.
[28] *GridFS*, URL: https://docs.mongodb.com/manual/core/gridfs/.
[29] *Click*, URL: https://click.palletsprojects.com/en/8.0.x/.
[30] *axios*, URL: https://www.npmjs.com/package/axios.
[31] *Kompose*, URL: https://kompose.io/.

Ringraziamenti

Al termine di questo lavoro di tesi, vorrei ringraziare il Prof. Giovanni Malnati, per avermi dato la possibilità di lavorare su questo progetto così interessante e per la professionalità e la gentilezza mostratemi sia nei panni di relatore che di professore.

Ringrazio il Dott. Fabio Forno, per essere stato la mia guida durante il lavoro di tesi, per essere sempre stato disponibile e per avermi fatto imparare tanti nuovi concetti.

Ringrazio la mia famiglia, per il loro sostegno incondizionato e per avermi sempre fatto sentire una persona speciale.

Ringrazio Alessia, per esserci sempre stata, per essere stata la compagna di viaggio perfetta e per avermi continuamente supportato e sopportato.

Ringrazio tutti i miei amici, per aver reso le pene di questi anni più leggere.

E infine vorrei ringraziare me stesso, per aver creduto in me e per non essermi mai arreso.

Ad maiora semper.