UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Elettronica

Progettazione e sviluppo di un sistema per la gestione di eventi nel contesto del water re-use mediante il framework FIWARE

Design and Development of a system to manage events in the context of water re-use using the FIWARE framework

Relatore:
Ch.mo Prof. Mancini Adriano

Laureando:
Castellano Rino

Anno Accademico 2019-2020

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Lo svolgimento di questo progetto è focalizzato sull'analisi ed implementazione del software di gestione dati FIWARE [7] per la creazione di un sistema di *Early-Warning* nel *Water Re-use*. Il fine del progetto è stato quello di studiare soluzioni smart, attraverso l'utilizzo di linguaggi di programmazione come Python e di *open-source application deployment projects* come Docker, per una gestione ottimale del sistema in un contesto di *Warning* nel rilevamento parametri.

1.2 Struttura della tesi

La tesi è strutturata in modo tale da esporre prima nozioni generiche del *framework* FIWARE, con esempi illustrativi, per poi trattare argomenti più specifici ed utili alla comprensione del progetto nella sua totalità. La tesi è strutturata come segue:

- nella prima parte saranno esposti gli strumenti ed i metodi utilizzati durante l'esperienza di tirocinio partendo dal software FIWARE con le sue funzionalità e le sue fondamentali estensioni, come Quantum-Leap[12]. Successivamente saranno esposti altri strumenti altrettanto essenziali come Docker, Docker-compose e Python[23].

- nella seconda parte sarà esposto lo schema di una WWTP (*Waste Water Treatment Plant*), la creazione dell'entità stessa, delle *subscription* e degli effetti sull'interfaccia utente attraverso l'utilizzo di Grafana.

- nella terza parte sarà esposta la progettazione di uno script Python che permetta l'inserimento automatico dei parametri appartenenti all'entità.

- nella quarta parte saranno esposte nozioni generiche del *framework* KNOWAGE integrato nel mondo FIWARE, delle sue potenzialità e delle difficoltà riscontrate nell'utilizzo che hanno successivamente causato il suo abbandono
- nella quinta parte sarà esposta la progettazione di uno script Python che, tramite *web-service*, controlla costantemente il trend dei parametri e, in presenza di anomalie, lo notifica l'utente.
- l'ultima parte conterrà considerazioni sul lavoro svolto e sull'importanza di un *framework* per la gestione dati, come FIWARE, nell'idea di *Smart Solutions*.

Capitolo 2

Strumenti e Metodi

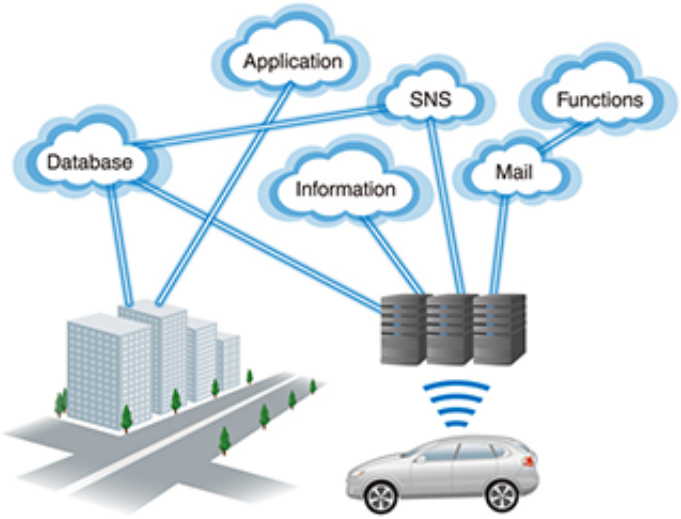
2.1 FIWARE

FIWARE è una piattaforma open-source per la gestione di ***"context data"***, ovvero il network di connessioni tra unità informative: esso contiene un set standard universale utile per la gestione di tutte le realtà *"smart"*, dalle *"smart industries"* alle *"smart cities"*. In questo paragrafo verranno analizzate tutte le sue caratteristiche e features che la contraddistinguono dalle altre piattaforme di management.

2.1.1 Smart Solutions

In un mondo sempre più diretto alla digitalizzazione delle informazioni, è risultato necessario uno standard comune per la gestione di dati che velocizzasse la comunicazione tra utenti e che fosse privo di in incongruenze di tipo lessicale-informatico. Una realtà sempre più prorompente nella gestione dei dati è quella delle ***"smart solutions"***, che può integrarsi in ambienti più estesi come gli *"smart office"*, le *"smart industries"*, le *"smart cities"* e così via in crescendo. Con *"smart solutions"* si intende una stretta collaborazione ottimizzata tra sensoristica hardware (IoT, *Internet of Things*) e software di carattere gestionale. L'idea di *"smart solutions"* si può applicare anche a realtà quotidiane come nel settore automobilistico. Con determinati sensori e dispositivi wireless, è permessa una gestione e visualizzazione facilitata di molte informazioni utili, tra cui lo stato di sicurezza dell'automobile, la propria rubrica telefonica oppure la propria mail.
I sistemi di *"smart solutions"* sono governati da determinate librerie e, poichè spesso le stesse case produttrici le creano, ognuna di esse è indirizzata verso il proprio settore. Un conseguente svantaggio di questa diversificazione di librerie è la loro carente versatilità.
FIWARE è una comoda soluzione per una gestione di dati generalizzata in numerosi campi, quali lo *Smart environment*, il *Waste Management*, *Weather* ecc.

Figura 2.1: Esempio di smart car: connessioni con diverse funzionalità come applicazioni, posta elettronica, informazioni di sensoristica [20]

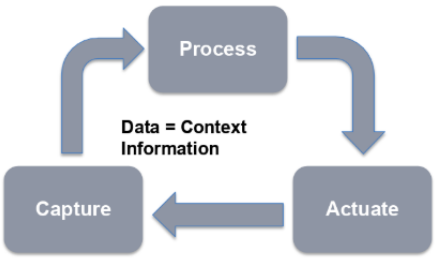


2.1.2 Ciclo di lavoro

FIWARE possiede un proprio processo di elaborazione dati che permette una gestione efficiente delle risorse:

- ***capture***: le componenti IoT, robot o sistemi terzi acquisiscono i dati di loro pertinenza che verranno poi gestiti di conseguenza dentro il "Core"
- ***process***: il dato acquisito viene processato
- ***actuate***: in base alle condizioni imposte dal progettista nella fase precedente, verranno effettuate delle operazioni sui sensori o sul sistema stesso al fine di eliminare eventuali anomalie oppure per ottimizzare determinati processi

Figura 2.2: Processo elaborazione dati FIWARE[24]

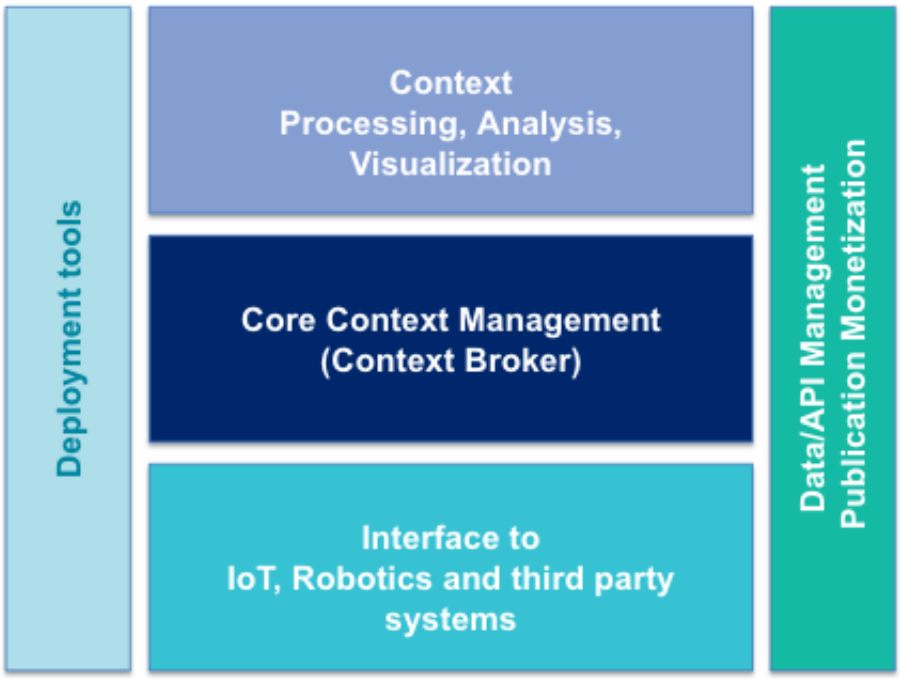


2.1.3 Architettura

La piattaforma FIWARE è organizzata secondo il seguente schema:

- **Orion Context Broker**: il cuore ed il motore del framework. Implementa NGSIv2 REST API[9], libreria per l'invio di richieste *HTTP* ad altre periferiche, sia di tipo *server* che *client*. Ogni comando inviato a ciascuna periferica del sistema deve passare prima per l'*Orion Context Broker*, che di conseguenza le processa e le invia ai destinatari nella forma più indicata. Il suo utilizzo risulta fondamentale nella creazione di entità, *subscriptions* ed invio di *queries*, argomenti che saranno sviluppati lungo la trattazione della tesi. [7]
- **IoT, Robotics and third party systems**: componenti che forniscono i dati da gestire o che ricevono comandi dall'*Orion Context Broker* nella fase di *Actuate*, ovvero dopo che è stato processato il dato [8]
- **Context Processing, Analysis, Visualization**: permette l'analisi dei dati, la sua visualizzazione ed eventualmente una loro modifica per una migliore gestione dell'environment. Ne fanno parte diverse applicazioni come *QuantumLeap*, che sarà analizzata di seguito.

Figura 2.3: Organizzazione generica FIWARE[24]



2.1.4 Data acquisition & processing

Per effettuare le varie richieste *HTTP* all'*Orion Context Broker*, viene utilizzata una libreria NGSIv2 che definisce un unico data model per le informazioni, il ***context entities***, ed un'unica interfaccia di context data per lo scambio di informazioni attraverso *query* e ***subscription***[9].

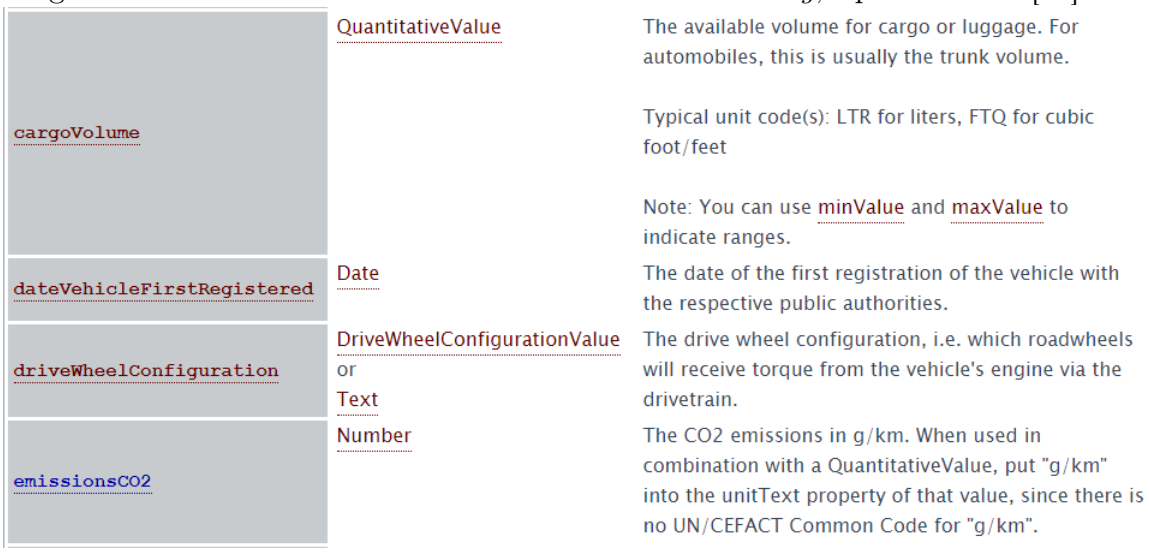
2.1.4.1 Context Entities

L'entità è una rappresentazione, in formato JSON, dell'ambiente da gestire in questione. Secondo l'utilizzo di un unico standard di informazioni esistono dei valori, obbligatori o meno, da inserire al fine di rendere l'entità riutilizzabile e standardizzata:

- ***id***: nome che identifica l'entità in questione. (nell'esempio di gestione di una casa con all'interno delle stanze, l'entità di una stanza può chiamarsi *"Room:1"*)
- ***type***: il tipo di entità che si sta utilizzando e che caratterizzerà la scelta dei successivi attributi. (nell'esempio della stanza il tipo sarà *"Room"*)
- ***attrs***: attributi caratterizzanti l'entità e di cui è possibile tracciarne la veridicità. Possono essere di tipo geografico (indirizzo, coordinate), di tipo testuale (nome della stanza, nome dell'industria), di tipo tempo (ultimo accesso, ultima modifica, anno di nascita) ecc. Il più frequente, lavorando spesso con sensori IoT, è il tipo numerico.

Gli attributi devono essere inseriti seguendo dei determinati criteri basati sul tipo di entità che si sta analizzando. A questo scopo è associato a FIWARE l'utilizzo di ***schema.org***, un vocabolario standardizzato per dati strutturati. È stato scelto *schema.org* per il suo motore di ricerca che "comprende" il significato del *web content* e per la sua completa affidabilità nella ricerca *client-side*. *schema.org* sopprime diverse ambiguità della classificazione dati come l'utilizzo di un unico URI per ogni istanza e l'ipergeneralizzazione. Grazie all'utilizzo dell'enorme vocabolario di cui dispone, la comprensione degli attributi più pertinenti da utilizzare è immediata e priva di ambiguità.

Figura 2.4: Interfaccia utente della libreria di *schema.org*, tipo *"Vehicle"*[18]



cargoVolume	QuantitativeValue	The available volume for cargo or luggage. For automobiles, this is usually the trunk volume. Typical unit code(s): LTR for liters, FTQ for cubic foot/feet Note: You can use minValue and maxValue to indicate ranges.
dateVehicleFirstRegistered	Date	The date of the first registration of the vehicle with the respective public authorities.
driveWheelConfiguration	DriveWheelConfigurationValue or Text	The drive wheel configuration, i.e. which roadwheels will receive torque from the vehicle's engine via the drivetrain.
emissionsCO2	Number	The CO2 emissions in g/km. When used in combination with a QuantitativeValue, put "g/km" into the unitText property of that value, since there is no UN/CEFACT Common Code for "g/km".

2.1.4.2 CRUD

Qualsiasi comando di inserimento, modifica e lettura entità avviene attraverso comandi di tipo *HTTP*. I comandi vengono compensati nel concetto di CRUD (*Create, Read, Update, Delete*) e si riassumono nei seguenti:

- **POST**: creazione di entità (http verb: `/v2/entities`), attributi (`/v2/entities/<entity-id>`)
- **GET**: lettura di tutte le entità(`/v2/entities`) o di una specifica (`/v2/entities/<entity-id>`) e di attributi (`/v2/entities/<entity-id>/attrs/<attribute>/value`)
- **PUT**: caricamento/sovrascrizione valori di attributi singoli (`/v2/entities/<entity-id>/attrs/<attribute>/value`)
- **PATCH**: analogo a PUT ma utile per blocchi di attributi
- **DELETE**: eliminazione di entità `/v2/entities/<entity-id>`) o di singoli attributi (`/v2/entities/<entity-id>/attrs/<attribute>`)

I comandi di tipo CRUD richiedono obbligatoriamente per ogni richiesta :

- il tipo di comando (POST,GET . . .)
- l'ambiente di lavoro dove eseguire il comando
- il formato dell'informazione che si vuole utilizzare, tipicamente JSON
- l'informazione stessa

Esempi espliciti verranno analizzati nella sezione 3.2

2.1.4.3 Subscription

Esistono dei parametri nella nostra entità che cambieranno nel tempo, subiranno delle variazioni di stato del tipo:

- **sincrono**, si aggiornano con una scandita ciclicità
- **asincrono**, possono variare in qualsiasi momento senza un periodo ben definito

Gli eventi più complessi da gestire sono quelli di tipo asincrono poiché spesso derivano da possibili anomalie o semplici eventi che necessitano di una repentina risposta del sistema. *Orion Context Broker* offre un meccanismo asincrono di notificazione che permette di sottoscrivere al cambiamento del *context information*. Il meccanismo di notificazione è fondamentale per una conoscenza immediata dell'utente sugli eventi di nostro interesse. Questo meccanismo si chiama ***"subscription"*** ed è una colonna portante nell'ecosistema FIWARE poiché non necessita di continui sondaggi o ripetizioni di richieste, fattore di elevata efficienza e snellezza. Il vantaggio più consistente è la diminuzione del traffico dati all'interno del network, quindi un aumento della risposta complessiva[1].

Come per le entità, le *subscriptions* possiedono un loro Id e degli attributi che li caratterizzano, i più fondamentali sono:

- **subject**: descrive l'entità, l'attributo in questione ed eventualmente una condizione che permetta la notificazione. Se quest ultimo è assente viene inviata una notifica ogni qual volta l'attributo cambia stato
- **notification**: tutte le informazioni inerente alla notifica (indirizzo destinatario, l'ultima volta che la notifica è avvenuta con successo o meno ecc).

Le *subscription* vengono gestite nell'*HTTP verb* `/v2/subscriptions` e si utilizzano gli stessi comandi CRUD del paragrafo Context Entities. Esempi espliciti verranno analizzati nel capitolo successivo sezione 3.3.

2.1.5 QuantumLeap

QuantumLeap è un REST service di immagazzinamento, richiesta e recupero via NGSIv2 di *time-series data*, sequenze di dati ordinati nel tempo. Lo scopo dell'applicazione è convertire il dato dal formato NGSI in un formato tabulato affine all'immagazzinamento in *Time-series Database*. Ciascun dato, trasformato in record, è quindi associato ad un indice di tempo e, se presente, una locazione geografica. È possibile successivamente poter recuperare un determinato dato attraverso un sistema di filtraggio *time-range* di tipo spaziale.
Le specifiche di libreria REST, soprannominate *NGSI-TDB*, sono state definite con lo scopo di creare un interfaccia di database agnostica all'immagazzinamento, richiesta e recupero di time-series data.

2.1.5.1 Crate

CrateDB è il *time-series database back-end* usato di default da *QuantumLeap*. Il suo utilizzo è intuitivo e risulta facile nella gestione grazie alla sua architettura *shared-nothing*. *CrateDB* è inoltre predisposto alla containerizzazione dell'applicazione, discorso che sarà sviluppato nel paragrafo *"Docker e Docker-Compose"*. L'enorme capacità di dati di *CrateDB* e la sua elevata velocità di accesso sono tali da rendere le risposte alle query in real-time; questa grossa qualità sarà essenziale per un utilizzo efficiente di Grafana, altra applicazione fondamentale nell'environment di QuantumLeap.

2.1.5.2 Grafana

Grafana è un tool di visualizzazione grafica di *time-series data*. Il suo utilizzo dipende dall'esisteza di un *time-series database* come *CrateDB* che gli permetta un adeguato accesso ai dati ed una sua rappresentazione grafica. Attraverso questo tool è inoltre possibile creare query per una visualizzazione più specifica e pertinente alle esigenze del *client*.
Grafana sarà molto utilizzato per l'implementazione grafica di esempi svolti nella tesi, soprattutto per rendere chiari determinati concetti come il *Data Ingestion & Visualization Workflow*. .[12]

2.1.5.3 Struttura QuantumLeap e ciclo di elaborazione dati

Tipicamente *QuantumLeap* acquisisce indirettamente l'IoT data da un *FIWARE IoT agent layer* combinato al sistema di subscription NGSI impostato in precedenza. L'entità NGSI entrante sarà convertita in un record di database e immagazzinata in uno dei *time-series database back-end* che tipicamente utilizza l'applicazione. Affinchè *QuantumLeap* possa ricevere i dati dall'*Orion Context Broker*, deve esistere una subscription che specifichi l'indirizzo di arrivo della notifica e la causa per cui è stata inviata.
Il diagramma mostra la creazione, da parte dell'utente, di una subscription direttamente con *QuantumLeap*:

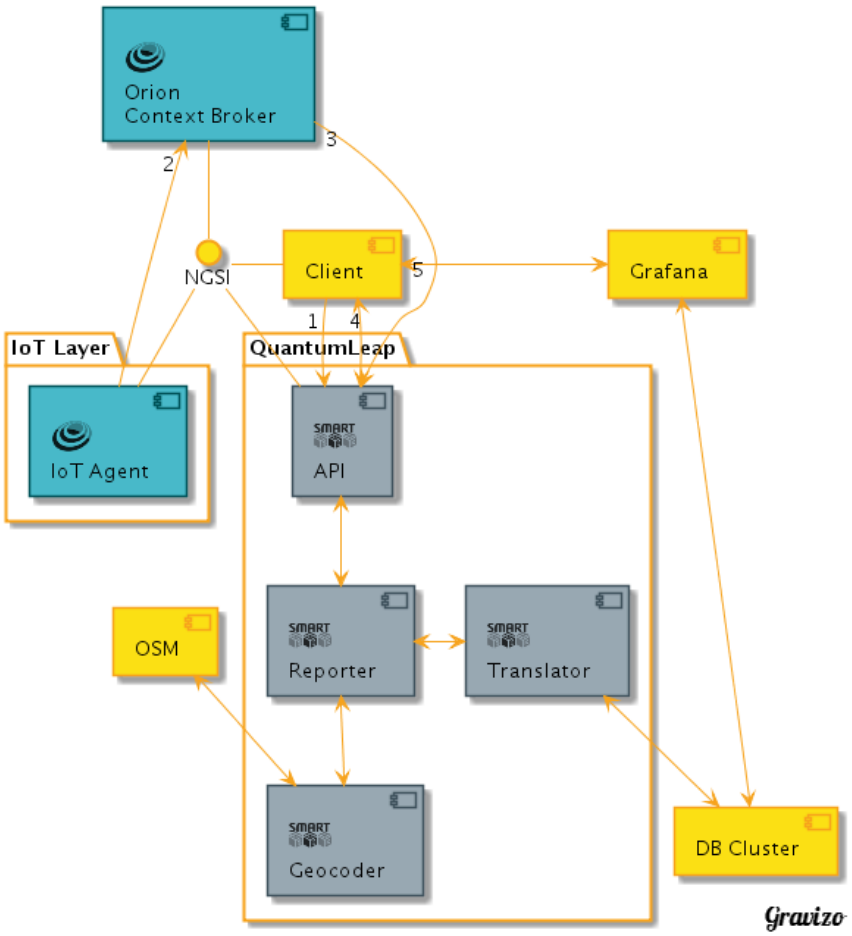
- il primo passaggio è solo un end point di convenienza nel *QuantumLeap REST API* dove il client inoltra la *subscription* all'*Orion*.
- l'*IoT Agent*, che può essere un sensore od un software esterno, invia il dato all'*Orion Context Broker*.
- se i dati si riferiscono ad un entità individuale, l'*Orion Context Broker* inoltra il data al *QuantumLeap* attraverso una richiesta POST (notifica). Il componente **Reporter** analizza e valuta il dato che ha causato la notifica. Se il geo-coding è configurato, il Reporter invoca l'utilizzo del **Geocoder** che, con l'ausilio di OpenStreetMap (OSM) ne identifica le informazioni

geografiche utili. Successivamente il Reporter passa le entità convalidate ed armonizzate al **Translator**. Il Translator converte l'entità NGSI in formato tabulare e lo indirizza come *time-series data* nel database.

- quando il *client* chiede al REST API di recuperare le entità NGSI, il *Reporter* ed il *Translator* interagiscono per inoltrare la domanda WEB in una SQL con clausole spazio temporali. Successivamente ottengono il record indicato e lo riconvertono in entità NGSI. Come notato prima, il meccanismo di *query* è semplice: *QuantumLeap* supporta il filtraggio attraverso *time range*, richieste geografiche e funzioni aggregate.

- infine *Grafana* mostra il database in modalità grafica per una possibile analisi da parte del *Web Client*.

Figura 2.5: Struttura di funzionamento QuantumLeap con Orion Context Broker [12]



2.2 Docker e Docker-compose

Nello sviluppo del progetto il primo problema in cui è possibile imbattersi è l'utilizzo di molteplici applicazioni che si appoggiano sullo stesso *server*. Una conseguenza di questa osservazione è l'annullanmento della sicurezza informatica delle applicazioni poichè usufruibili anche a utenti esterni.
Una possibile soluzione a questo problema è stato l'utilizzo di **Docker**, una piattaforma *open source* che automatizza il *deployment* di applicazioni all'interno di *container software*. Per container si intende l'applicazione stessa ma con funzioni di incapsulamento implementate che permettono l'isolamento dalle altre.
L'incapsulamento delle applicazioni risulta particolarmente efficiente rispetto all'utilizzo individuale di ognuno per diversi motivi:

- **maggiore portabilità**: può essere eseguito in diverse piattaforme come *VirtualBox, Amazon Web Services (AWS), Google Compute Platform (GCP)*
- **isolamento delle risorse**, utile per la rimozione delle app con maggiore efficienza
- **sicurezza**: le app eseguite sui container sono completamente separate le une dalle altre, per garantirne il completo controllo sul flusso del traffico. Questo vantaggio risulterà particolarmente utile nello sviluppo del progetto.

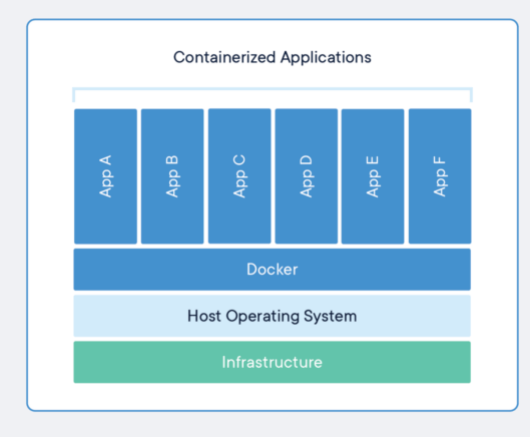
Ogni container possiede il proprio file di sistema privato, la cosiddetta *image*, che include il codice, *dependencies* ed altri file di sistema richiesti.
Docker-Compose è un tool di *Docker* che permette l'utilizzo di più container simultaneamente: attraverso un file di estensione "*.yaml*" è possibile configurare i servizi di applicazione e successivamente poter creare, avviare, stoppare, eliminare tutti i container con un unico comando. Durante lo svolgimento della tesi verranno utilizzati entrambe le applicazioni poiché risulterà necessario un utilizzo simultaneo di tutti i container ed al contempo anche un utilizzo singolo.

2.3 Python

Python è un linguaggio di programmazione di "alto" livello, molto vicino al mondo della programmazione ad oggetti. Possiede una sintassi intuitiva ed una tipizzazione dinamica tale da renderlo un linguaggio fortemente utilizzato nello scripting e nello sviluppo di rapide applicazioni. Nel corso della scrittura di codice, saranno utilizzate diverse librerie non ancora installate nell'ultimo aggiornamento e che quindi necessiteranno un download esterno: a questo scopo sarà utilizzato il classico comando via terminale

```
1 pip3 install <nome-libreria>
```

Figura 2.6: Esempio di organizzazione containerizzata nell'open-source Docker [26]



Python sarà utilizzato per la scrittura di due script fondamentali per il progetto:

- script per l'inserimento automatico di valori, attraverso comandi *http* all'interno di una entità FIWARE
- script per la creazione di un web-service adatto al controllo Alert nella rilevazione parametri dell'entità FIWARE di nostro riferimento.

Capitolo 3

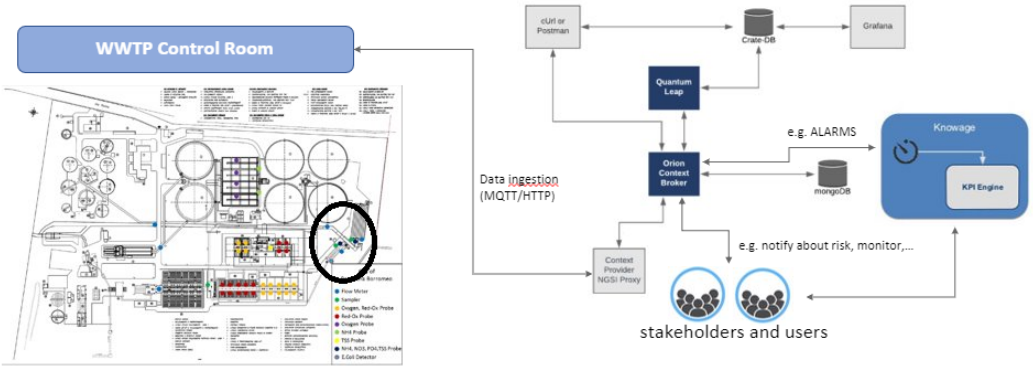
Sviluppo del progetto

L'idea di "smart solutions", come scritto precedentemente, può essere applicata a qualsiasi tipo di realtà digitale, dalla *"smart car"* alla *"smart city"*. In questo capitolo sarà sviluppata la creazione framework un'entità di *WasteWater Treatment Plant (WWTP)* firmata ***FIWARE*** nell'ambito del progetto europeo Digital Water City (DWC)[22].

3.1 Stazione di WWTP

Il trattamento delle acque reflue è un processo di rimozione di sostanze dannose (inquinanti) per l'ambiente e per chi ne usufruisce al fine di poterle riutilizzare in altri settori o per essere reinserita in un ambiente naturale.
In base alla carica batterica presente nelle acque reflue ed al tipo di riutilizzo che ne verrà fatto, vengono utilizzati determinati processi depurativi fisici, chimici e biologici.
Nella fig.3.1 è possibile visualizzare un esempio di una WWTP, l'area di sensoristica, quindi pertinente al progetto, è quella cerchiata in basso a destra. La *"smart solution"* applicata alla *WasteWater Treatment Plant* è inerente alla rilevazione di parametri che determinino la "pulizia" delle acque, quindi mediante un utilizzo di un'attrezzata sensoristica. Successivamente alla rilevazione, i dati saranno inviati all'*Orion Context Broker* che a sua volta li invierà, sotto forma di subscription, al sistema del *QuantumLeap*.
Dovendo sviluppare un sistema di *early-warning*, un ramo fondamentale è quello della gestione simultanea di allarmi: nell'esempio in 3.1 questo ruolo è svolto da Knowage, un'applicazione di analisi dati sviluppata parallelamente a FIWARE. Dato lo scarso riscontro nell'utilizzo di esso, Knowage sarà sostituito da un web-server sviluppato ad hoc su Python.

Figura 3.1: Mappa di una WWTP e della sua implementazione con FIWARE



3.1.1 Parametri utilizzati

Per la creazione dell'entità sono state seguite le regole imposte dal vocabolario di *schema.org*, più precisamente nell'ambiente *WaterQualityObserved*.
Oltre ai classici attributi di *id*, *type*, indirizzo e di ultima modifica, i parametri usati per l'entità di *WWTP* sono i seguenti:

- *temperature*. *Default unit*: *Celsius Degrees*.
- *conductivity*. *Default unit*: *Siemens per meter (S/m)*.
- *conductance*. *Default unit*: *Siemens per meter at 25 °C (S/m)*.
- *tss* (*total suspended solid*). *Default unit*: *milligrams per liter (mg/L)*
- *tds* (*total dissolved solid*). *Default unit*: *milligrams per liter (mg/L)*.
- *turbidity*: ammontare di luce sparpagliata dalle particelle nella colonna d'acqua. *Default unit*: *Formazin Turbidity Unit (FTU)*.
- *salinity*. *Default unit*: *Parts per thousand (ppt)*.
- *pH* : livello di acidità o basicità nella soluzione acquosa. *Default unit*: *Negative of the logarithm to base 10 of the activity of the hydrogen ion*.
- *orp* (*Oxidation-Reduction potential*).*Default unit*: *millivolts (mV)*.

[6] I parametri sopra riportati possono a loro volta possedere dei "metadati", ovvero dei dati supplementari che consentono una comprensione più accurata del parametro in esame. Nel caso degli attributi di una WWTP l'unico metadata reso disponibile è il "timestamp", ovvero la data di ultima modifica dei dati stessi.

3.2 Creazione entità

Per creare l'entità è necessaria una richiesta di tipo POST via terminale con i seguenti prerequisiti:

- Descrizione del tipo di comando (`curl -iX POST`). Con *curl* si intende l'utilizzo dell'omonimo tool, utile al trasferimento di dati da o verso un server, usando i classici protocolli supportati (ad es. *HTTP*). I comandi *curl* sono progettati per lavorare senza interazione da parte dell'utente[2].
- L'indirizzo URI dove è indirizzato tale comando (`uri 'http://localhost:1026/v2/entities'`)
- Il formato dell'informazione che si vuole passare (`header 'Content-Type: application/json'`)
- L'informazione da passare. Nella creazione dell'entità è fondamentale l'inserimento dell'*id* e del *type*, tutte le altre informazioni possono essere inserite successivamente.

```
curl -iX POST \
--url 'http://localhost:1026/v2/entities' \
--header 'content-type:application/json'\
--data'{
"id":"WaterStation:1",
"type":"WaterStation",
"tds":{"type":"Double","value":1},
"temperature":{"type":"Double","value":0.13},
"tss":{"type":"Double","value":22},
"orp":{"type":"Double","value":56},
}'
```

Un esempio grafico è nella fig. 3.2.
Nel caso di modifica dell'entità le parti da cambiare del comando sono:

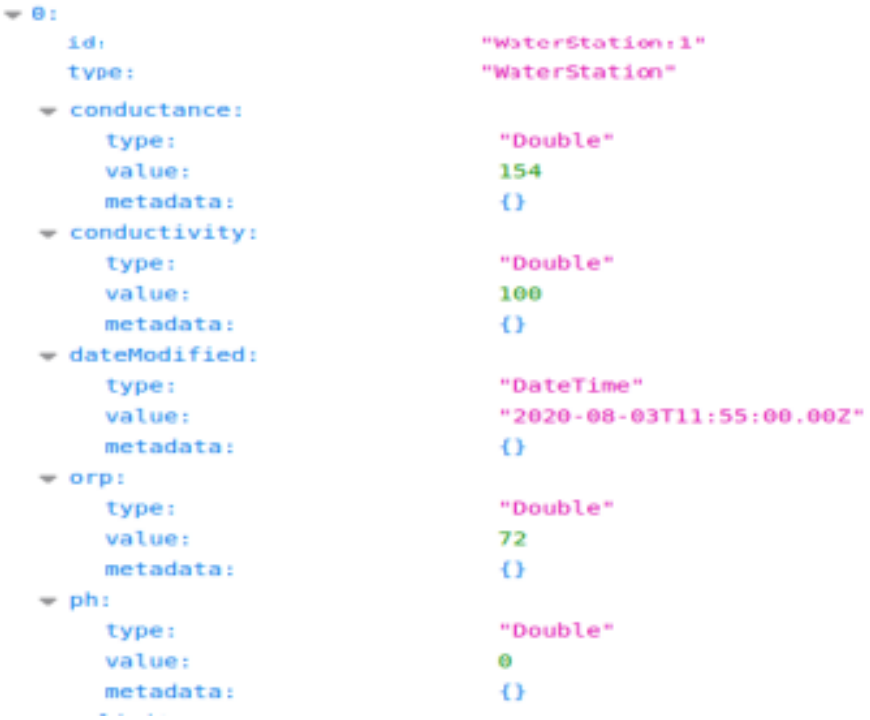
- Sostituire l'estensione `/entities` con `/op/update`
- Aggiungere nel `data` il tipo di azione da eseguire, in questo caso `"actionType":"append"`
- Rappresentare l'entità da modificare come un elemento dell'array `"entities"`

```
curl -iX POST \
--url 'http://localhost:1026/v2/op/update' \
--header 'content-type:application/json'\
--data'{
"actionType":"append"
"entities":[
{
"id":"WaterStation:1",
```

```
"type":"WaterStation",
"tds":{"type":"Double","value":24.5},
"temperature":{"type":"Double","value":28.0},
"tss":{"type":"Double","value":12},
"orp":{"type":"Double","value":35},
}
]}'
```

Eventualmente può essere aggiunto il metadata "*timestamp*" per ogni parametro[1].

Figura 3.2: Esempio di rappresentazione formato JSON dell'entità creata



3.3 Creazioni subscription

Volendo informare il *QuantumLeap* di ogni cambiamento di stato dei parametri, non è necessario l'inserimento di una condizione poiché di default il singolo cambiamento di stato risulta motivo di notificazione.
Per inserire una subscription è necessaria una richiesta POST via terminale con i seguenti requisiti:

- Descrizione del tipo di comando (`curl -iX POST`)
- L'indirizzo URI dove è indirizzato tale comando (`uri 'http://localhost:1026/v2/subscriptions'`)

[1]parte omessa poichè irrilevante al nostro utilizzo

- Il formato dell'informazione che si vuole passare (`header 'Content-Type: application/json'`)
- L'informazione da passare. I parametri da inserire obbligatoriamente sono i seguenti:
  - `description`: il motivo della notificazione
  - `subject`: tutte le informazioni inerenti l'entità da notificare e gli attributi che causano la notifica
  - `notification`: tutte le informazioni riguardanti il destinatario, ovvero il suo `http`, l'attributo che sarà inserito nel time-series database, ed eventuali metadata da considerare
  - `throttling`:limite per il tasso di richieste http che può ricevere il *QuantumLeap* tramite questa *subscriptions* [4]

Nel prossimo capitolo sarà analizzato il problema dell'inserimento automatico di valori temporizzati nel sistema *FIWARE*.
Per poter rendere possibile la temporizzazione dei dati in *CrateDB*, nelle *subscription* utilizzate è stato inserito in *(notification)(attrs)* il dato *"dateModified"*, che si modifica automaticamente ad ogni cambio di stato nell'entità.

```
curl -iX POST \
--url 'http://localhost:1026/v2/subscription' \
--header 'content-type:application/json'\
--data'{
"description":"change salinity",
"subject":{
"entities":[
{
"idPattern":"WaterStation:1",
"type":"WaterStation"
}
]
},
"notification":{
"http":{
"url":"http://localhost:8668/v2/notify"},
"attrs":[
"salinity","dateModified"],
},
"throttling":5}'
```

Nella fig. 3.3 è presente un esempio grafico di creazione subscription.

Figura 3.3: Rappresentazione formato JSON della subscription dopo la sua creazione

Capitolo 4

Progettazione e Sviluppo del Sistema

4.1 Data Ingestion & Visualization Workflow

Un sistema sensoristico di un WWTP necessita di un flusso di dati continuo da e verso la stazione affinché si possa istantaneamente ricorrere, nel caso di anomalie, ad azioni risolutive. Lavorando nell'ambiente digitale, una ricezione continua è pressocchè impensabile: una soluzione intuitiva risulta la discretizzazione dei valori nel tempo con un periodo sufficientemente basso per permettere il requisito scritto precedentemente. I processi caratterizzanti questo capitolo sono i seguenti:

- **Data Ingestion**: processo di cattura istantanea e di importazione dati per un loro utilizzo immediato o immagazzinamento in database. Se inviassimo una *HTTP request* per ogni dato ricevuto potrebbe risultare difficile, se non impossibile, permettere una trasmissione dati istantanea: per ottimizzare questo processo, saranno create *HTTP request* con all'interno tutti i valori dei parametri acquisiti in quel preciso istante.[19]
- **Visual Workflow**: tecnica di visualizzazione "dall'alto" di tutto il processo sotto esame, nel nostro caso l'inserimento dei dati automatizzato nel sistema.[21]

Gli strumenti utilizzati e l'algoritmo dello script Python creato per questo scopo, verranno illustrati nei seguenti paragrafi.

4.1.1 *script_inserimento_automatico.py*

4.1.1.1 Principio di funzionamento

Dato che non è stato possibile, per ovvi motivi, accedere ai sensori di rilevazione dati di una stazione WWTP, è stato scritto uno script Python per l'inserimen-

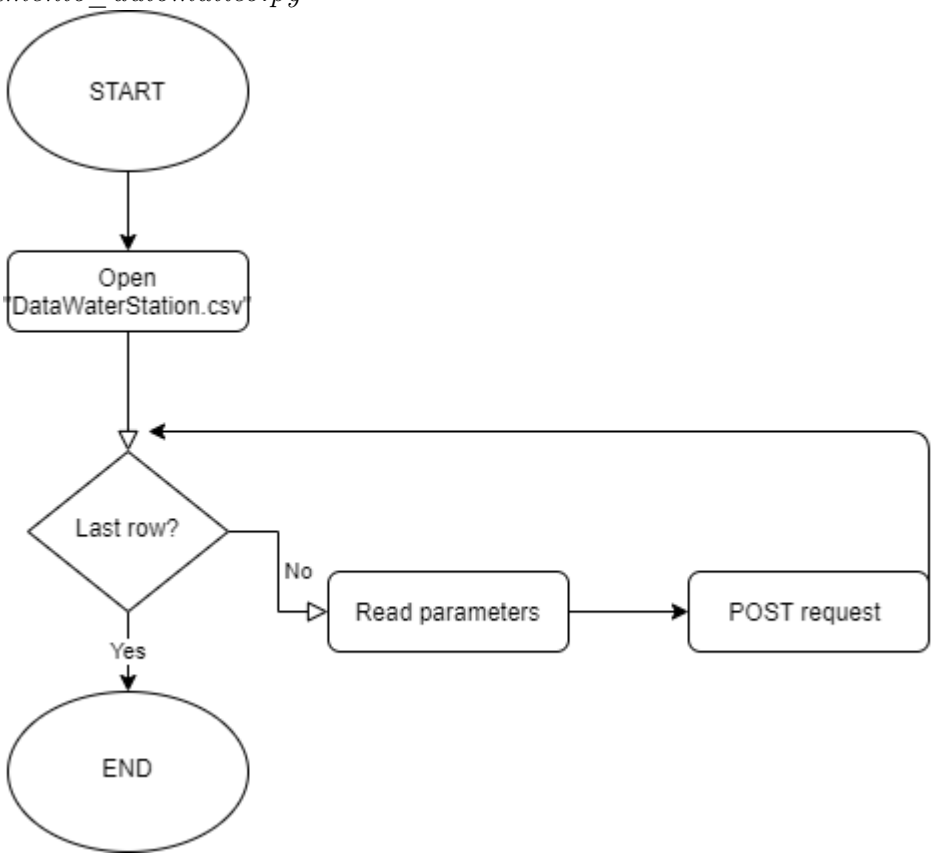
to automatico di valori temporizzati all'interno dell'entità "*WaterStation:1*", in modo tale da emulare un flusso continuo di dati.

Il principio di funzionamento è basato su pochi step ma fondamentali per una buona esecuzione:

- **apertura del file *DataWaterStation.csv*** , file contenente tutti i dati temporizzati da caricare su *CrateDB*
- **lettura riga per riga** dei dati attinenti ad un istante temporale
- **caricamento**, tramite richiesta di tipo **POST dei dati** letti nella riga seguente.

Per rendere il più realistico possibile il flusso di ingresso dati si è ipotizzata una temporizzazione discretizzata ad 1 minuto, anche se l'inserimento effettivo di tutto il file *.csv* è condensato in pochi secondi, a seconda del numero di righe da leggere.

Figura 4.1: Diagramma di flusso del funzionamento dello script Python *script_ inserimento_ automatico.py*



4.1.1.2 *DataWaterStation.csv*

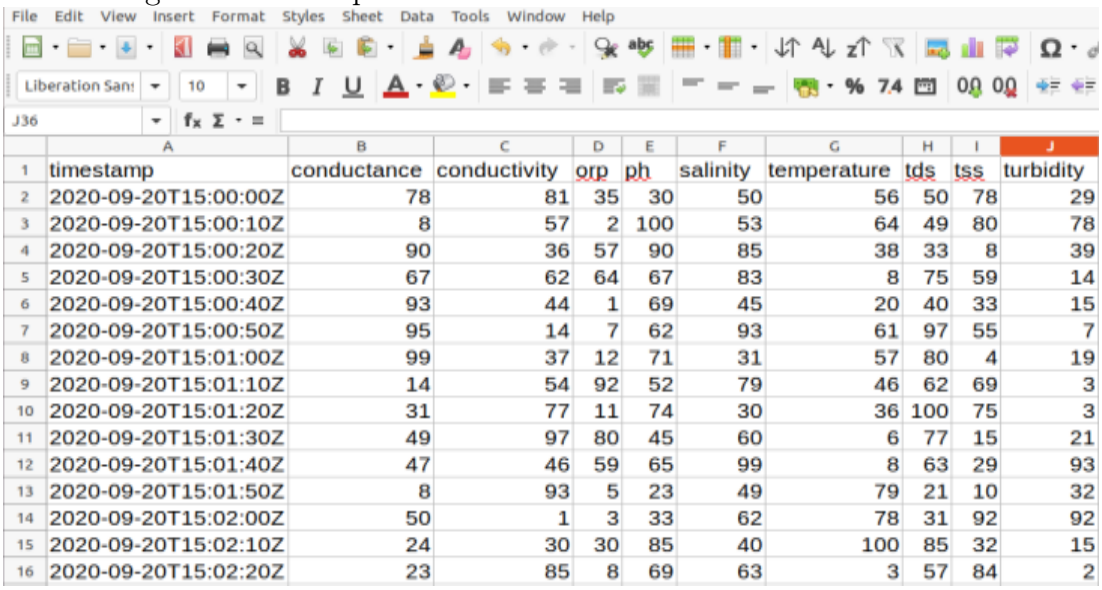
Il file *DataWaterStation.csv* simulerà l'ingresso costante di nuovi valori temporizzati nel tempo. Lo schema è facilmente intuibile e si può suddividere il contenuto in due macro-attributi:

- ***timestamp***: l'ipotetico istante temporale preso in considerazione per l'erogazione dati da parte dei sensori. Con l'ultimo aggiornamento della libreria *NGSIv2*, il formato di tipo Data è diventato
  YYYY-MM-DD'T'HH:mm:ss'Z'
  Risulta necessaria l'aggiunta della lettera 'Z' (*'Zulu time'*) , sostituta del più noto 'GMT' (*'Greenwich Mean Time'*).

- I **parametri** letti in quel preciso *timestamp* dai sensori, e poi inviati all'*Orion Context Broken*. A causa delle classiche regole di *upcasting*, risulta necessario un inserimento dei valori tipizzati correttamente onde evitare conseguenze indesiderate, soprattutto nel caso di parametri *Integer*. Per un corretto inserimento ci si può riferire alla propria entità, poiché nella registrazione iniziale del parametro è anche associato il tipo.

Non essendo stato richiesto uno studio di andamenti particolari dei valori, sono stati assunti per ogni istante di tempo dei numeri randomici da 0 a 100, come si può notare in fig. 4.1.1.2.

Figura 4.2: Esempio di contenuto del file *DataWaterStation.csv*



	timestamp	conductance	conductivity	orp	ph	salinity	temperature	tds	tss	turbidity
2	2020-09-20T15:00:00Z	78	81	35	30	50	56	50	78	29
3	2020-09-20T15:00:10Z	8	57	2	100	53	64	49	80	78
4	2020-09-20T15:00:20Z	90	36	57	90	85	38	33	8	39
5	2020-09-20T15:00:30Z	67	62	64	67	83	8	75	59	14
6	2020-09-20T15:00:40Z	93	44	1	69	45	20	40	33	15
7	2020-09-20T15:00:50Z	95	14	7	62	93	61	97	55	7
8	2020-09-20T15:01:00Z	99	37	12	71	31	57	80	4	19
9	2020-09-20T15:01:10Z	14	54	92	52	79	46	62	69	3
10	2020-09-20T15:01:20Z	31	77	11	74	30	36	100	75	3
11	2020-09-20T15:01:30Z	49	97	80	45	60	6	77	15	21
12	2020-09-20T15:01:40Z	47	46	59	65	99	8	63	29	93
13	2020-09-20T15:01:50Z	8	93	5	23	49	79	21	10	32
14	2020-09-20T15:02:00Z	50	1	3	33	62	78	31	92	92
15	2020-09-20T15:02:10Z	24	30	30	85	40	100	85	32	15
16	2020-09-20T15:02:20Z	23	85	8	69	63	3	57	84	2

4.1.1.3 Esecuzione script

Il codice Python necessiterà di due sole librerie supplementari a quelle già presenti di default:

- **csv**: comandi per lettura file di tipo *csv*
- **json**: comandi per lettura, scrittura in variabili *JSON*
- **requests**: comandi per invio *HTTP requests*

L'esecuzione del file si basa sui passi esposti nel paragrafo precedente e si può suddividere nei seguenti passaggi:

- apertura file *DataWaterStation.csv*

```
    with open("./DataWaterStation.csv", newline="",encoding="
ISO-8859-1") as filecsv:
            lettore=csv.reader(filecsv,delimiter=",")
            header=next(lettore)

```

- Lettura riga per riga del file, tramite un ciclo for che effettua comandi per ogni riga acquisita.

```
    for valori in dati:
        print(len(valori))
        url = "http://localhost:1026/v2/op/update"
        payload = f"{{\"actionType\":\"APPEND\",\"entities
\":[{{\"id\": \"WaterStation:1\", \"type\":\"WaterStation
\",\"conductance\":{{\"value\":{valori[1]},\"type\":\"
Integer\"}},\"conductivity\":{{\"value\":{valori[2]},\"type
\":\"Integer\"}},\"orp\":{{\"value\":{valori[3]},\"type\":\"
Integer\"}},\"ph\":{{\"value\":{valori[4]},\"type\":\"
Integer\"}},\"salinity\":{{\"value\":{valori[5]},\"type\":\"
Integer\"}},\"temperature\":{{\"value\":{valori[6]},\"type
\":\"Integer\"}},\"tds\":{{\"value\":{valori[7]},\"type\":\"
Integer\"}},\"tss\":{{\"value\":{valori[8]},\"type\":\"
Integer\"}},\"turbidity\":{{\"value\":{valori[9]},\"type
\":\"Integer\"}},\"dateModified\":{{\"value\":\"{valori
[0]}\",\"type\":\"DateTime\"}}}}]}}"
        y=json.loads(payload)
        payload=json.dumps(y)

```

Tramite l'utilizzo della tecnica di *f-string*, è possibile inserire variabili all'interno di una stringa per crearne una concatenata. Ad esempio, al posto di *{valori[0]}*, ci sarà il valore effettivo associato all'array di lettura per ogni riga.
Un accorgimento non indifferente è l'inserimento di *timeStamp* come parametro di *dateModified*, poiché nella *subscription* creata nella 3.3 è stato inserito il suddetto parametro come attributo di notifica.

- Caricamento, tramite *POST request*, dei valori temporizzati.

```
    headers = {'Content-Type': 'application/json'}

    response = requests.request("POST", url, headers=headers,
    data = payload)

```

L'esecuzione dello script consiste nella scrittura sul terminale del seguente comando:

```
python3 script_inserimento_autom.py
```

4.2 Esempi di Risultati

Per questo esempio verranno utilizzate le seguenti informazioni del file *DataWaterStation.csv*, rappresentate in figura 4.3:

Figura 4.3: Esempio di *DataWaterStation.csv*

	timestamp	conductance	conductivity	orp	ph	salinity	temperature	tds	tss	turbidity
2	2020-09-20T15:00:00Z	78	81	35	30	50	56	50	78	29
3	2020-09-20T15:01:00Z	8	57	2	100	53	64	49	80	78
4	2020-09-20T15:02:00Z	90	36	57	90	85	38	33	8	39
5	2020-09-20T15:03:00Z	67	62	64	67	83	8	75	59	14
6	2020-09-20T15:04:00Z	93	44	1	69	45	20	40	33	15
7	2020-09-20T15:05:00Z	95	14	7	62	93	61	97	55	7
8	2020-09-20T15:06:00Z	99	37	12	71	31	57	80	4	19
9	2020-09-20T15:07:00Z	14	54	92	52	79	46	62	69	3
10	2020-09-20T15:08:00Z	31	77	11	74	30	36	100	75	3

L'utilizzo dello script *script_inserimento_autom.py* ipotizza l'ingresso di un numero finito di http POST per istanti temporali ben definiti. É possibile immaginare l'esecuzione dello script come un ciclo for di *POST requests*, tutte del seguente tipo:

```
curl -iX POST \
url "http://localhost:1026/v2/op/update"\
header "content type:application/json"\
data "{
"actionType":"append",
"entities":
[
{
"id":"WaterStation:1",
"type":"WaterStation",
"orp":{"type":"Double","value":35},
"temperature":{"type":"Double","value":56},
"tss":{"type":"Double","value":78},
"tds":{"type":"Double","value":50},
```

```
"turbidity":{"type":"Double","value":29},
"salinity":{"type":"Double","value":50},
"conductance":{"type":"Double","value":78},
"conductivity":{"type":"Double","value":81},
"ph":{"type":"Double","value":30},
"dateModified":{"type":"DateTime","value":"2020-09-20T15:00:00Z"}}
]
}"
```

Terminata l'esecuzione dello script, o durante la stessa esecuzione, se il file *DataWaterStation.csv* possiede un numero particolarmente elevato di righe, è possibile visualizzare determinati risultati dell'operazione sulle applicazioni *Crate* e *Grafana*.

4.2.1 Crate

Analizzando le fig. 4.2.1, l'utilizzo dello script provoca delle conseguenze favorevoli o meno: l'inserimento dei valori ha avuto successo ma *CrateDB* non dispone in ogni istante di tempo tutti i valori dei parametri. Il *time-series DB* rileva l'inserimento dei valori come coppia *parametro - timestamp*, associando ad ognuno di essi una propria riga e lasciando per gli altri parametri un valore di tipo *"NULL"*.

Un esempio è evidente nelle Fig. 4.2.1: nella prima riga è presente solo il

Figura 4.4: Risultato esecuzione script sul time-series DB Crate

datemodified	fiware_servicepath	temperature
1600614190000 (2020-09-20T15:03:10.000Z)		NULL
1600614190000 (2020-09-20T15:03:10.000Z)		NULL
1600614190000 (2020-09-20T15:03:10.000Z)		NULL
1600614190000 (2020-09-20T15:03:10.000Z)		13

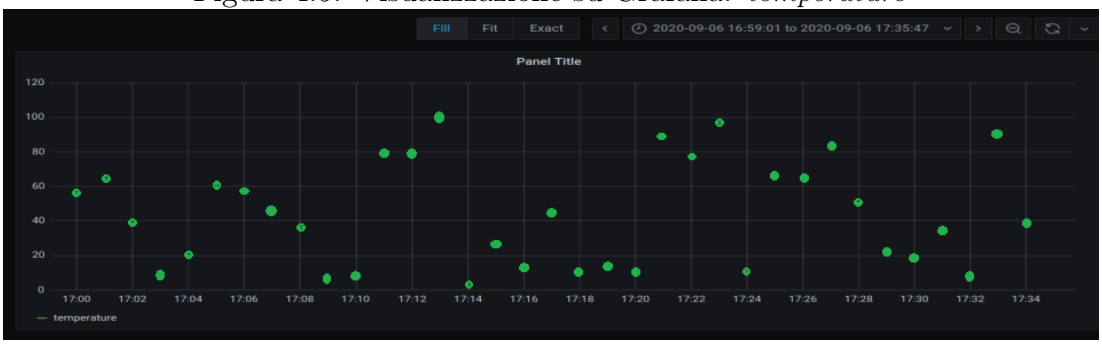
tss	tds	salinity
20	NULL	NULL
NULL	NULL	NULL
NULL	64	NULL
NULL	NULL	NULL

valore di *tss* associato all'istante temporale *2020-09-20T15:03:10.0000Z*, per vedere il risultato in *tds* nello stesso istante, è necessario visualizzare la terza riga. Questo è dovuto al fatto che ogni riga di DB si basa sulle subscriptions create nella 3.3, le quali hanno come unici attributi il singolo parametro e *dateModified*. Visto che con CrateDB risulterebbe complicato controllare il successo dello script precedente, verrà utilizzato Grafana, molto più affine alla visualizzazione grafica di dati.

4.2.2 Grafana

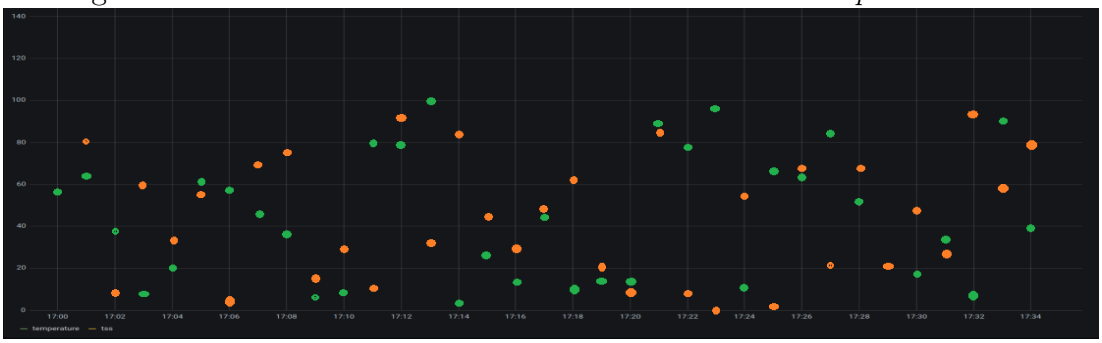
Nella fig. 4.2.2 è possibile osservare un risultato di inserimento automatico di dati, precisamente nel range di tempo *17:02-17.34*, filtrato per il parametro *temperature* e, in fig. 4.2.2, confrontato con *tss*.
La prima considerazione è la discretizzazione di un flusso che, visto da un range di tempo più esteso ed attraverso determinati metodi di interpolazione, risulterà continuo.
La seconda considerazione, ma non meno importante, è la possibilità di visualizzare l'andamento di un parametro in determinati range temporali, condizione implausibile per CrateDB.
Una stretta conseguenza di quest'ultima considerazione è la possibilità di evidenziare anomalie nell'andamento temporale, come ad esempio il superamento di una determinata soglia [1].

Figura 4.5: Visualizzazione su Grafana: *temperature*



[1] Nel caso preso in analisi questo fattore non incide poiché i valori dei parametri sono stati presi con un metodo randomico, quindi non attinente alla realtà.

Figura 4.6: Visualizzazione su Grafana: confronto tra *temperature* e *tss*



Capitolo 5

Knowage

Il seguente capitolo esporrà brevemente le caratteristiche, l'utilità del *framework Knowage* e la sua integrazione con un entità firmata *FIWARE* ©. Infine saranno presentate le difficoltà riscontrate nel suo utilizzo e le motivazioni del suo abbandono nello sviluppo del progetto.

5.1 Open-Source Business Intelligence

Un obiettivo del percorso di tirocinio è stato quello di trovare un sistema di allarme efficiente che avvisasse in tempo reale l'utente di tutte le anomalie presenti.
A questo scopo inizialmente è stato scelto il *framework Knowage*, una *suite open source* professionale per analisi di tipo *modern business*.
La suite è composta da diversi moduli, ciascuno concentrato per uno specifico dominio analitico, che utilizzati individualmente attraverso risorse tradizionali e sistemi di *big data*, possono assicurare soluzioni a determinati obiettivi[14].

I sotto prodotti delle suite sono tutti inerenti al mondo della visualizzazione / *dashboarding* di dati, citando i più importanti:

- *BD (Big Data)*, per analizzare data immagazzinati in big data cluster o NoSQL database
- *SI (Smart Intelligence)*, business intelligence su dati strutturati ma più orientato a capacità *self-service*
- *LI (Location intelligence)*, per offrire business data con informazioni spaziali/geografiche

Knowage supporta una visione moderna delle *data analytics*: offre nuove capacità *self-service* che diano autonomia all'*end-user*, ora capace di costruire le sue analisi ed esplorare il suo *data space*, anche combinando i dati provenienti da differenti sorgenti.

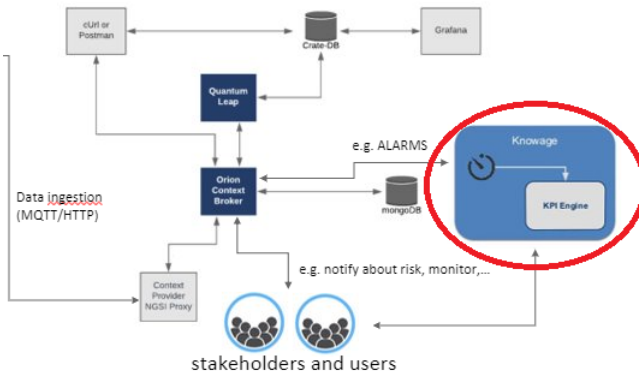
Figura 5.1: Esempio di interfaccia Knowage per il data visualization [25]

5.2 Integrazione con FIWARE

Osservando il dettaglio dello schema della stazione WWTP sulla sezione inerente a FIWARE (fig. 5.2), già descritto nel Cap.3, la parte cerchiata in rosso rappresenta Knowage.

Esso si pone, nella gestione degli **Alarm** esattamente in mezzo tra l'Orion

Figura 5.2: Mappa della implementazione con Knowage di FIWARE

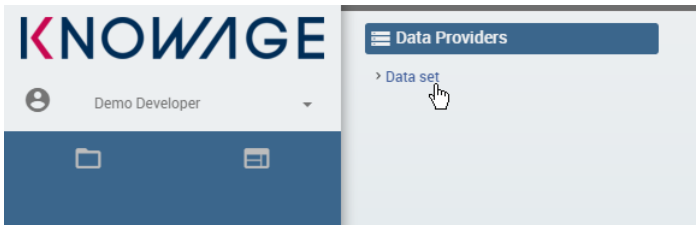


Context Broker e l'utente bi-direzionalmente:

- ***Southbound Traffic*** *(Traffico "verso sud")*: una comunicazione che va dall'*Orion Context Broker*, che invia i dati a *Knowage*, all'utente, che riceve da *Knowage* eventuali allarmi da gestire

- ***Northbound Traffic*** *(Traffico "verso nord")*: inversamente, una comunicazione che va dall'utente, che attua modifiche al sistema tramite *Knowage*, all'*Orion Context Broker*, che riceve tali modifiche e le gestirà di conseguenza

Per permettere una comunicazione tra l'utente e l'*Orion* sono necessari dei passaggi preliminari[1].
Prima di tutto è necessaria la creazione di un *DataSet* su *Knowage* inerente all'entità di nostro interesse: si clicca nella homepage su *Data set*.
Successivamente, cliccando sul '+' in alto a destra, si procede nella creazione



del nuovo *DataSet*.

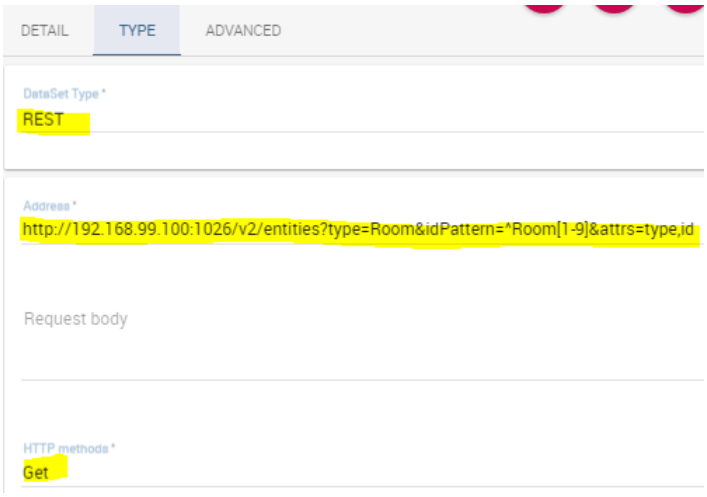
Le caratteristiche fondamentali ed obbligatorie sono nella finestra '*Tab*' e riguardano:

- L'URL dell'*Orion Context Broker* con le specifiche di interesse per l'utente. Se l'utente vuole focalizzarsi sull'analisi della temperatura dell'acqua l'URL sarà composto anche da `'...?attrs=temperature'`
- Il tipo di *HTTP request*.
- Checking della NGSIv2 box, permette la creazione di subscription nell'*Orion*

5.3 Interfaccia utente

Seguiti i precedenti passaggi è possibile passare alla creazione del **cockpit**, presente sul *My Workspace*. La sua creazione è intuitiva e basata sull'utilizzo di

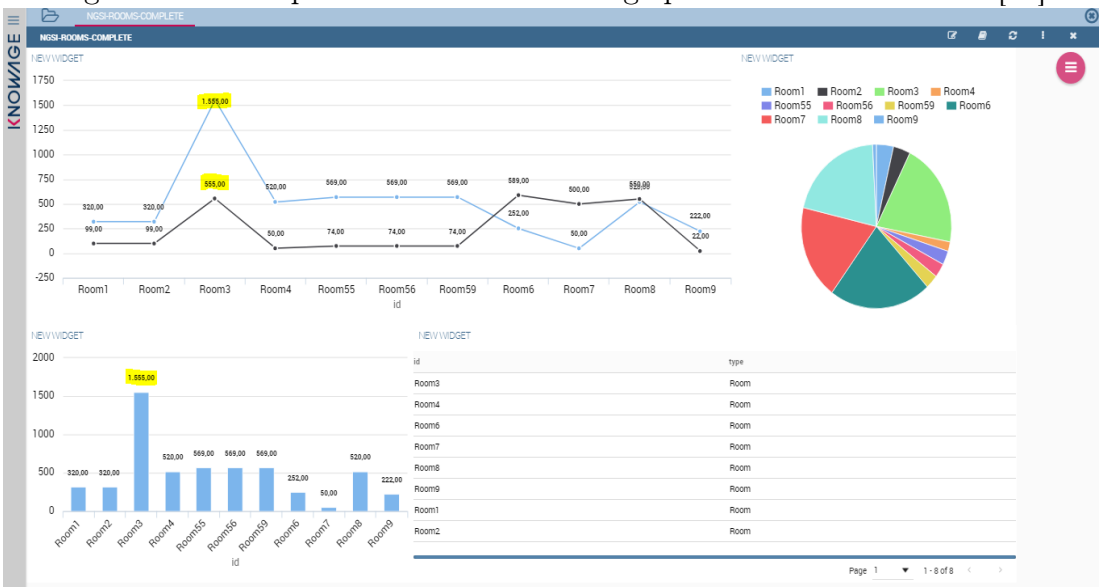
[1]Le immagini che seguono sono state prese dal tutorial di Knowage-FIWARE[15]



grafici e statistiche modificabili a discrezione dell'utente, sia da un punto di vista di intuitività grafica che di filtraggio analitico (selezione di range temporali, parametri, average/max/min value).

Infine è possibile creare delle *KPI Alert* (*Key Performance Indicator*) che identificano l'andamento di un determinato processo e, nel caso di conferma delle condizioni sottoscritte tramite subscription, inviino di allarmi all'utente.

Figura 5.3: Esempio di interfaccia Knowage per il data visualization [15]



5.4 Problemi riscontrati

Pur riconoscendo le potenzialità e le funzionalità avanzate del *framework*, dopo ripetuti tentativi di un suo utilizzo, l'applicazione è stata messa in secondo piano a causa di alcune problematiche irrisolvibili nel breve tempo che avrebbero impedito il raggiungimento dell'obiettivo.
In seguito sono elencate le problematiche riscontrate in ordine di importanza crescente:

- ***Connessione con l'Orion Context Broker*** poco intuitiva e ***resettata ogni qual volta venisse spenta la macchina***. Questa problematica si è riscontrata anche nello sviluppo dello script del Cap.6, ma poi risolto lavorando all'interno del codice: l'*Orion Context Broker* cambia il suo indirizzo IP ogni qual volta che la *Virtual Machine* viene riaccesa. Purtroppo *Knowage* non ha la funzionalità di trovare il nuovo indirizzo, l'unica soluzione è che l'utente stesso cerchi, tramite comandi via terminale, il nuovo indirizzo IP per poi cambiarlo con quello obsoleto nel *DataSource*.

- ***Tempi di attesa*** per l'avvio di *Knowage* ***troppo elevati (720.000 ms= 12 min)***. Probabilmente questo problema è anche dovuto al fatto che è stata utilizzata una *Virtual Machine* aggravata anche dalla pesantezza de suo codice JAVA.

- costanti ***problemi nella connessione tra i container 'Knowage' e 'KnowageDB'***.
  Knowage, non potendo connettersi direttamente a *CrateDB*, e quindi rilevarne le time-series data, si appoggia ad un DB integrato di tipo MySQL. Lavorando assieme al professore giorni interi su questo problema, non si è riusciti a raggiungere nessuna connessione tra le due applicazioni, quando essa stessa sarebbe dovuta essere automatizzata nell'installazione dell'applicazione.

5.5 Considerazioni

Vista la macchinosità del suo utilizzo e vista la carenza di dati a disposizione, solamente quelli salvati sull'*Orion*, si è stati costretti ad abbandonare l'utilizzo di *Knowage* e del suo sistema di *KPI Alert*.
Per poter comunque raggiungere l'obiettivo, ovvero creare un sistema di *Early-Warning*, è stata scelta la creazione di un web-service che avesse in linea di principio le stesse funzionalità di gestione *Alert*.
Tale idea sarà sviluppata nel capitolo seguente tramite una creazione di uno script Python ben mirato ed ottimizzato.

Capitolo 6

Gestione di Warning ed Allarmi

Date le considerazioni scritte in conclusione nel Cap.5, è risultato necessario cercare un'alternativa a *Knowage* che svolgesse lo stesso compito di gestione *Alert*. Avendo già lavorato durante il progetto con la scrittura di script Python, una possibile soluzione è risultata la creazione di un *Web Server* tramite script Python, che fosse in costante ascolto sull'*Orion Context Broker* per gestire eventuali *Alert*.
Nel seguente capitolo sarà descritto, dopo una breve presentazione dello strumento Flask[10] e del concetto di *Web Server*, il processo di creazione di tale script e di tutte le funzionalità implementate in esso.

6.1 Flask

6.1.1 Web Server

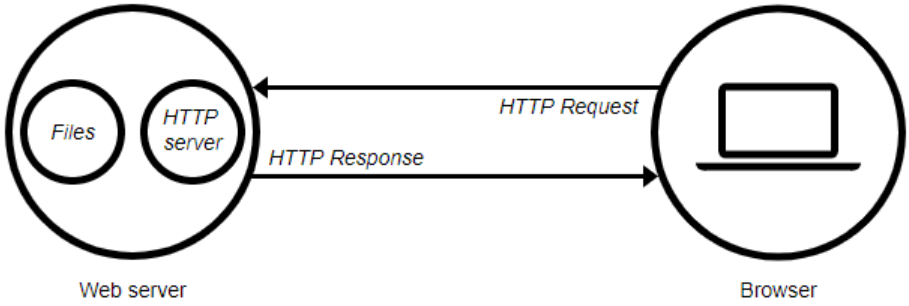
Per ***Web Server*** si intende un'applicazione software che, in esecuzione su un server, è in grado di gestire le richieste di trasferimento pagine web del *client*, tramite l'utilizzo di determinati protocolli HTTP e porte *TCP (Transmission Control Protocol)*[1].
Per rendere più chiara la definizione è possibile descrivere un esempio: ogni volta che un browser necessita di un file ospitato in un *web server*, il browser effettua una richiesta *HTTP*. Quando la richiesta raggiunge il corretto web server (nel senso di porta TCP), esso accetta la richiesta, trova il file e lo inoltra al browser, sempre tramite protocolli *HTTP*[3].
Per ogni richiesta *HTTP* inviata, il web browser restituisce un codice di risposta. I codici più comuni sono:

- 200, la richiesta ha avuto esito positivo;

[1]La porta TCP più comune è la 8080, ma ciò non nega il fatto che si possano usare altre porte; nel nostro caso infatti utilizzeremo la porta 5000

- 400, richiesta non valida;
- 404, oggetto non trovato;
- 500, errore interno del server.

Figura 6.1: Esempio di utilizzo di un Web Server[3]



Flask è un *micro web server* scritto in Python[5]. Non richiedendo particolari *tool* o librerie, con conseguente perfetta maneggiabilità per il programmatore, è classificato come *microframework*. Può essere integrato con qualsiasi libreria di Python e non pone vincoli nelle decisioni operative[11]. Flask è stato creato per essere semplice nel suo utilizzo ed estensione con altre applicazioni: l'idea che pone le radici in Flask è la costruzione di una solida fondazione per applicazioni web di differente complessità, che possa applicarsi ad ogni estensione noi utile.
La scelta di questo specifico *microframework* è anche dovuta al supporto integrato estremamente veloce di debugging, al contrario di Knowage, ed una piena libertà concessa all'utente nella costruzione di moduli[17].

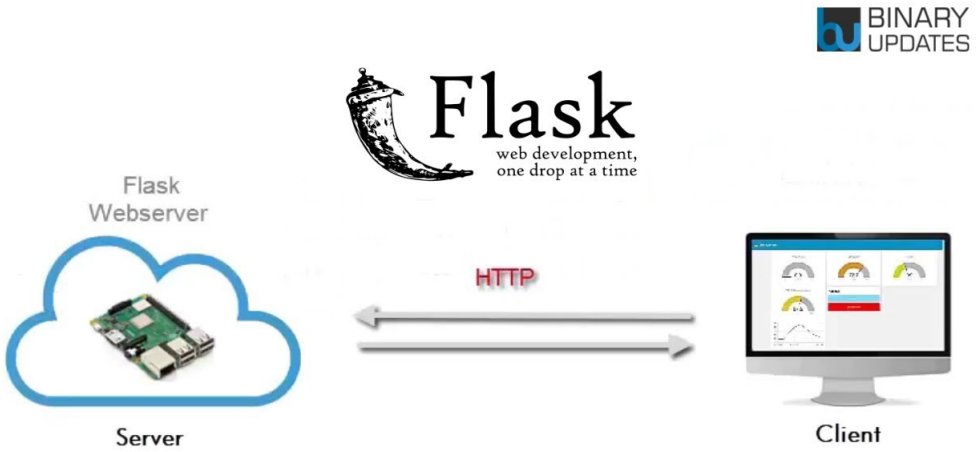
6.1.2 Integrazione con Docker

Lo scopo di Flask come Web Server è quello di sostituire nello schema della stazione WWTP integrata a FIWARE, il ruolo di Knowage, ovvero:

- ***Gestione Alert*** provenienti da *Orion* tramite *subscription*
- ***Rappresentazione grafica*** utile per la visualizzazione dell'utente

Affinchè il *Web Server* possa interfacciarsi con l'*Orion* è necessario un prerequisito fondamentale: anche il web server creato deve essere containerizzato nello stesso *Docker Network* di tutto l'universo FIWARE utilizzato.
Il file di configurazione/creazione dei container è il *docker-compose.yml*: all'interno si trovano tutte le caratteristiche dei container, tra cui anche la porta TCP di riferimento.

Figura 6.2: Esempio precedente di Web Server ma di tipo Flask [13]



Per poter utilizzare Flask correlato alle altre applicazioni, si aggiunge nel file[2]:

```
web:
  build: .
  ports:
    - "5000:5000"

redis:
  image: "redis:alpine"
```

Dopo aver creato lo script Python inerente al Web Server, che analizzeremo nel prossimo paragrafo, con il comando

```
sudo docker-compose up -d
```

seguirà la creazione del container e della *image* dell'applicazione Flask secondo le istruzioni scritte dentro il proprio *Dockerfile*.
Tra i vari comandi scritti nel *Dockerfile*, i più rilevanti sono:

```
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
```

Il file *requirements.txt* è un file contenente l'elenco delle librerie esterne utilizzate sul proprio script Python. Con questi due comandi Docker deve leggere il file *requirements.txt* ed installarne il contenuto nel futuro script di Python containerizzato. Questa operazione enfatizza le ottime caratteristiche di sicurezza che possiede Docker: possono essere effettuate modifiche

[2]Da notare la scelta della porta che non coincide con il classico 8080

da esterni su un file containerizzato ma non ne subisce effettive variazioni se tali modifiche avvengono durante l'esecuzione.

- `ENV FLASK_APP=dockers.py`

  L'ambiente di esecuzione dell'applicazione è rappresentato dal file *dockers.py*

- `CMD [ "python3" , "tutorial.py"]`

  Prima di lanciare il Flask, viene eseguito lo script *tutorial.py*: esso ha lo scopo di ottenere l'indirizzo IP di "tirociniofiware_web_1", il mio container dove sarà salvato il *Web Server*. Il nuovo indirizzo IP sarà poi scritto sul file *my_ip.txt* e sarà visibile anche agli occhi dell'utente poichè il container non è ancora in esecuzione.
  Tale script è utile al fine di tenere il mio *web server* sempre aggiornato sulle possibili variazioni dell'indirizzo IP e di conseguenza di aggiornarsi automaticamente durante l'esecuzione senza ricorrere ad un intervento umano.

Nel prossimo paragrafo sarà descritto il principio di funzionamento dello script Python *dockers.py*, delle sue funzionalità e potenzialità.

6.2 Alarm Visualization & Management

Le librerie utilizzate per l'esecuzione dello script sono le seguenti:

- ***redis***: libreria di gestione di DBMS (*Database Management System*) di tipo NoSQL
- dalla libreria Flask: ***Flask***, ***jsonify***(permette di restituire in formato *HTML* un contenuto in *JSON*)
- ***json***
- ***csv***
- ***Crate.client***: implementa comandi di tipo *query* nel CrateDB
- ***datetime***
- dalla libreria flask_mail: ***Mail, Message***

La procedura per avviare il *Web Server* è la seguente:

```
app=Flask(__name__)
mail=Mail(app)
cache=redis.Redis(host='redis', port=6379)
if __name__=="__main__":
app.run(debug=True)
```

In sequenza: creazione di una istanza di tipo Flask (*linea 1*), creazione di una istanza di tipo Mail (*linea 2*), creazione di una istanza Redis, collegata tramite porta TCP 6379(*linea 3*), start dell'applicazione (*linea 4-5*).
Nel Web Server creato ad hoc per il progetto esistono tre pagine che possono ricevere entrambe le richieste *HTTP* di tipo POST e GET. Nei prossimi paragrafi saranno esaminate tutte le pagine e le risposte per ogni richiesta ricevuta.

6.2.1 /main

La pagina *main* è la colonna portante del Web Server sviluppato: possiede un ruolo di segnalazione anomalie ad una lista di utenti. Le anomalie considerate nel progetto sono solo di tipo ***threshold overtaking***, ovvero di superamento soglia(*threshold*) imposta dall'utente. La pagina sviluppata permette una visualizzazione di due caratteristiche possibili di anomalia:

- ***Alert***: l'ultimo valore ricevuto[3] supera un determinato *threshold* scelto dall'utente. Inoltre è possibile determinare se l'anomalia è un caso singolare o conseguenza di un trend precedente, studiando gli ultimi x valori e calcolando quante volte è stata superato il *threshold*.

- ***Wait_ exit_ alert***: l'ultimo valore ricevuto del parametro non supera la soglia massima ma negli ultimi x valori y lo superano. Per uscire da tale condizione è necessario avere un numero n di valori negli ultimi x che sia minore di y[4].

L'utente può solo visualizzare la pagina poichè la richiesta di POST è gestita dall'*Orion*: qualora avvenga una *subscription*, l'*Orion* invierà una notifica al *Web Server*.
Il primo passo per un utilizzo corretto del *main* è la creazione di adeguate *subscription* che interagiscano con il *Web Server*. Un esempio è rappresentato in fig. 6.2.1: gli attributi che meritano considerazione sono i seguenti:

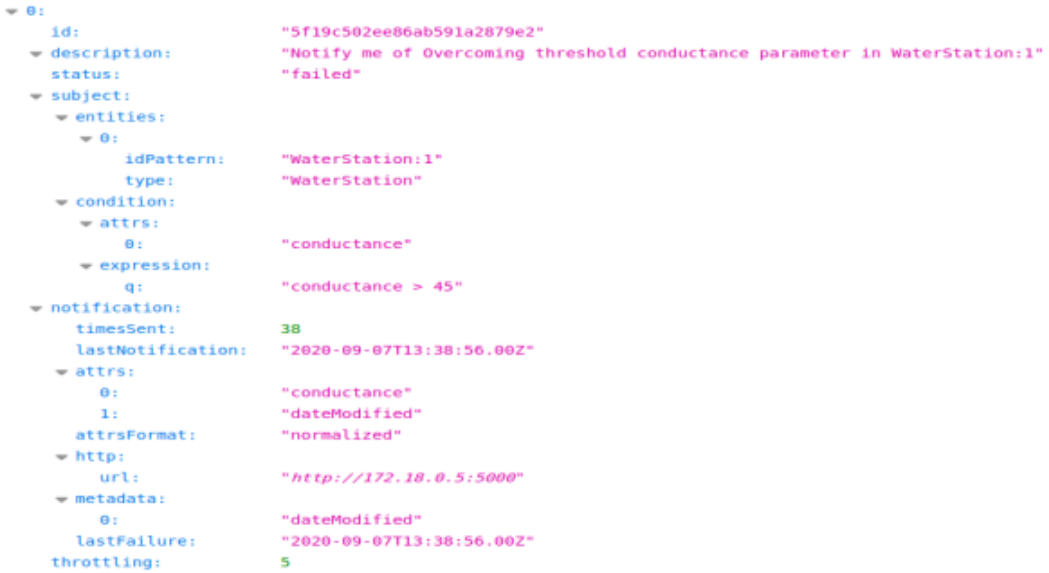
- ***expression***: lo script è in grado di correggere l'equazione ogni qual volta vengono modificate le configurazioni nel file *Threshold_ WaterStation.csv*

- ***url***: lo script è in grado di correggere automaticamente l'IP ogni qual volta cambia

[3]Per ultimo si intende leggendo il *dateModified* più recente, poichè è possibile inserire anche valori indietro nel tempo tramite opportuna digitazione

[4]$n < y$, con $y \leq x$

Figura 6.3: Esempio di subscription creata per il Web Server



6.2.1.1 Threshold_WaterStation.csv

L'intero */main* si sviluppa attorno al file *Threshold_ WaterStation.csv*, contenente informazioni essenziali per l'analisi di possibili anomalie della rilevazione parametri. Per ognuno di essi esistono 4 valori, modificabili dall'utente stesso nel file:

- ***threshold*** *(linea 1)*: la soglia che un parametro non dovrebbe superare per evitare una situazione di Alert
- ***limit*** *(linea 2)*: la quantità di valori che si prendono in esame per determinare l'uscita dalla *Wait_ exit_ alarm* descritta precedentemente.
- ***counter*** *(linea 3)*: quantità di valori all'interno di *limit* che devono superare la soglia per confermare il *trend* di Alert. Questo attributo è considerato soltanto quando l'ultimo valore di un parametro è maggiore della *threshold*
- ***counter_ exit*** *(linea 4)*: quantità di valori all'interno di *limit* che non devono superare la *threshold* per confermare l'uscita dalla situazione di *Wait_ exit_ alarm*.Questo attributo è considerato soltanto quando l'ultimo valore di un parametro è minore della *threshold*

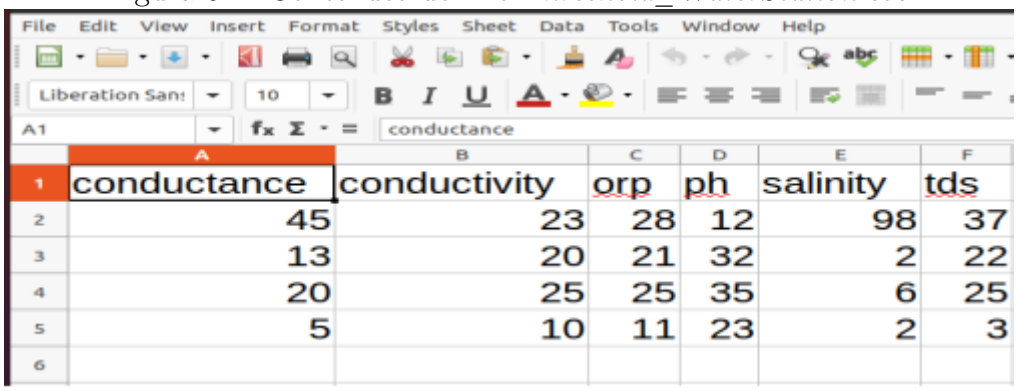
Questi valori possono essere modificati prima di avviare l'applicazione poiché successivamente anch'essi saranno protetti da un sistema di sicurezza imposto da Docker.
Considerando che i tempi di start per l'applicazione sono esigui rispetto all'inserimento dati[5], questo sistema è considerato accettabile seppur necessita di un

[5]Circa 1 min

riavvio tramite i comandi via terminale

```
sudo docker stop tirociniofiware_web_1
sudo docker start tirociniofiware_web_1
```

Figura 6.4: Contenuto del file *Threshold_ WaterStation.csv*



	A	B	C	D	E	F
1	conductance	conductivity	orp	ph	salinity	tds
2	45	23	28	12	98	37
3	13	20	21	32	2	22
4	20	25	25	35	6	25
5	5	10	11	23	2	3
6						

6.2.1.2 get_sub

Prima del controllo effettivo della richiesta *HTTP* ricevuta, esistono delle funzioni che vengono eseguite esternamente.
Una di queste si chiama *get_ sub* ed effettua un filtraggio tra tutte le subscription nell'Orion prelevandone solo quelle con le *expression*, poichè le altre subscription sono di comune variazione valore e che quindi non posseggono una *expression*. Il fulcro della funzione si riassume nelle seguenti righe:

```
for n in range(len(data)):
    if 'expression' in data[n]['subject']['condition']:


      header.append(data[n]['subject']['condition']['attrs'][0])
      if 'timesSent' in data[n]['notification']:
        timesSent.append(data[n]['notification']['timesSent'])
      else:
        timesSent.append(0)
    else:
      array_del.append(n)
```

Il codice di cui sopra consiste nel controllo in ogni *data*, ovvero subscription, della presenza di una *expression* e se è stata mai inviata una notifica (*timesSent*). Se è presente la *expression*, tale subscription sarà salvata in un array temporaneo di subscription. Se non sono presenti *timesSent*, ne sarà assegnato uno di default, ovvero 0.

6.2.1.3 control_threshold

Altra funzione fondamentale è la *control_threshold* che verifica la modifica di parametri nel *Threshold_WaterStation.csv* e dell'indirizzo IP, salvato su *my_ip.txt*, rispetto alle ultime subscription salvate.
Il controllo viene effettuato sull'array di subscription scritto precedentemente e, se esistono delle variazioni, saranno modificate di conseguenza. L'array modificato sarà infine sovrascritto sulle precedenti subscription, presenti nell'Orion Context Broker, tramite una *HTTP request* di tipo PATCH, poiché una richiesta di tipo POST rischierebbe di creare nuove subscription mantenendo anche quelle errate in vita, quindi annullandone l'effetto richiesto.

```
1 if (threshold[p] not in data[n]['subject']['condition']['expression'
    ]['q'] or data[n]['notification']['http']['url']!=my\_ip):
2
3         \#modify new sub
4         url = f"http://tirociniofiware\_orion\_1:1026/v2/
    subscriptions/{data[n]['id']}"
5         payload = f"{{\"description\":\"Notify me of Overcoming
    threshold {header[n]} parameter in WaterStation:1\",\"subject\":
    {{\"entities\": [{{\"idPattern\": \"WaterStation:1\", \"type\":
    \"WaterStation\"}}],\"condition\": {{\"attrs\": [\"{header[n
    ]}\"],\"expression\": {{\"q\": \"{header[n]} > {threshold[p
    ]}\"}}}}}},\"notification\": {{\"http\": {{\"url\": \"{my\_ip
    }\"}},\"attrs\":[\"{header[n]}\",\"dateModified\"],\"metadata
    \":[\"dateModified\"]}},\"throttling\": 5}}"
6         y=json.loads(payload)
7         payload=json.dumps(y)
8         headers = {'Content-Type' :'application/json'}
9         response = requests.request("PATCH", url, headers=headers,
     data = payload)
```

In questa funzione viene utilizzato solo un valore del file *Threshold_WaterStation*, ovvero *threshold*, poiché lo studio degli allarmi sarà effettuato soltanto nella richiesta di tipo GET.

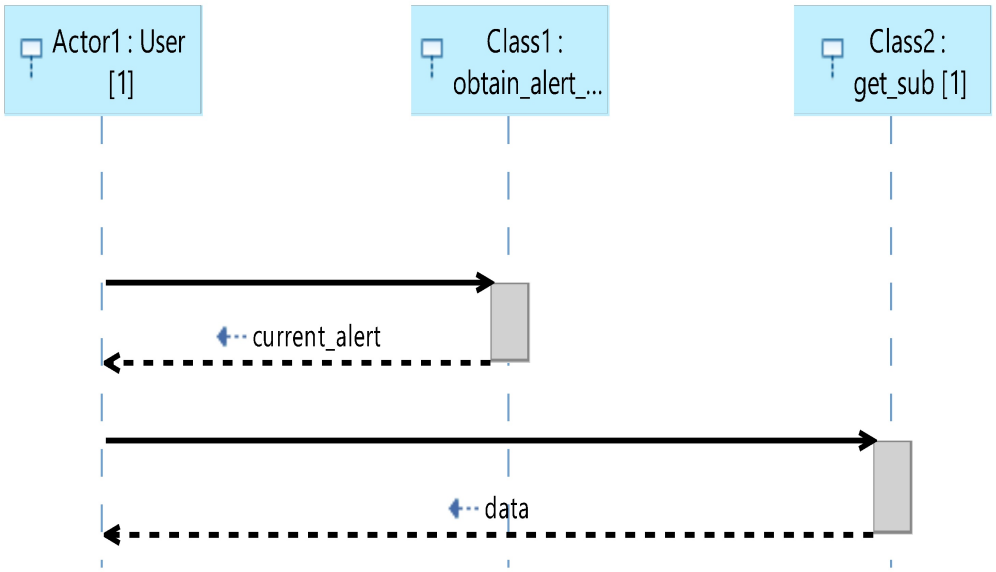
6.2.1.4 create_wait_alert

Questa funzione è fondamentale soltanto per la prima esecuzione dello script, poiché nell'utilizzo di esso risulta ininfluente allo sviluppo.
Tale funzione controlla ancor prima di ricevere le richieste se esistono dei possibili Alert o *Wait_exit_alert* che possano essere già pubblicati nell'entità di riferimento. Effettuando un controllo parametro per parametro su CrateDB, in base alle condizioni descritte sopra ed ai valori all'interno del file *Threshold_WaterStation.csv*, si ottengono le informazioni richieste e vengono postate nell'entità.

6.2.1.5 GET request

I diagrammi delle sequenze presenti all'interno dello script *dockers.py* sono essenzialmente corti e si riassumono in poche funzioni per diagramma. Il diagramma delle sequenze riferito alla richiesta GET nel *main* non fa eccezione:

Figura 6.5: Diagramma delle sequenze del metodo GET riferito all'estensione *main*



- ***obtain_alert_wait***: funzione che ottiene, leggendo l'entità di interesse, tutte le situazioni di Alarm o Wait che sono già presenti. La selezione delle informazioni di tipo Alarm è stata effettuata nel seguente modo:

```
1     array={k: v for k, v in data.items() if k.startswith('Alert
      ')}
2   for alert_id, alert_info in array.items():
3
4     string=alert_info['value']
5     date=alert_info['metadata']['dateModified']['value']
6
7     current_alert.append(f"Alert: {alert_id}, What: {string}.
      Last time notify: {date}. \n")
8
```

Riassumendo: (*linea 1*) sono stati selezionati tutti gli attributi presenti nell'entità che hanno come prima parola *'Alert'*, successivamente ne vengono filtrate le informazioni come il nome dell'Alarm esistente (*alert_id*, *linea 2*), la sua descrizione (*string*, *linea 4*), e la data in cui è apparso tale

allarme (*date*, *linea 6*). La sequenza di comandi descritta precedentemente, viene iterata allo stesso modo anche per le Wait filtrando in questo caso gli attributi con una stringa differente. La funzione infine ritorna l'array di Alert e Wait ordinati, ora utili per essere pubblicati a video.

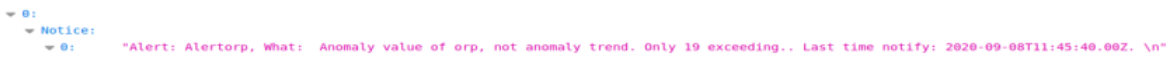
- ***get_ sub***, già descritto precedentemente. Risulta utile soltanto per un possibile refresh di informazioni se il sistema, alla GET request, rimane in caricamento per un periodo esteso.

Il contenuto ritornato da *obrain_ alert_ wait*, dopo essere stato trasformato in formato *JSON*, sarà utilizzato come messaggio di ritorno all'utente tramite la linea di codice

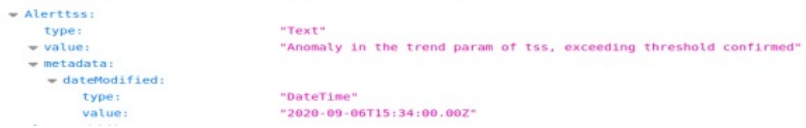
```
1 return jsonify(ok)
```

dove *'ok'* è l'array di Alert e Wait.
Come descritto precedentemente, ci sono 3 possibili stati di Alert/Wait riferiti ad un parametro:

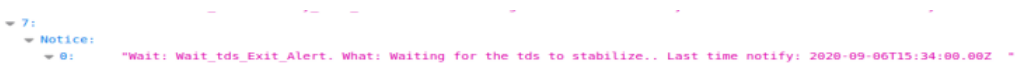
- ***Alert singolare***, poichè non presente un trend negativo di Alert



- ***Alert con trend negativo***



- ***Attesa di uscita*** dalla situazione di pericolo

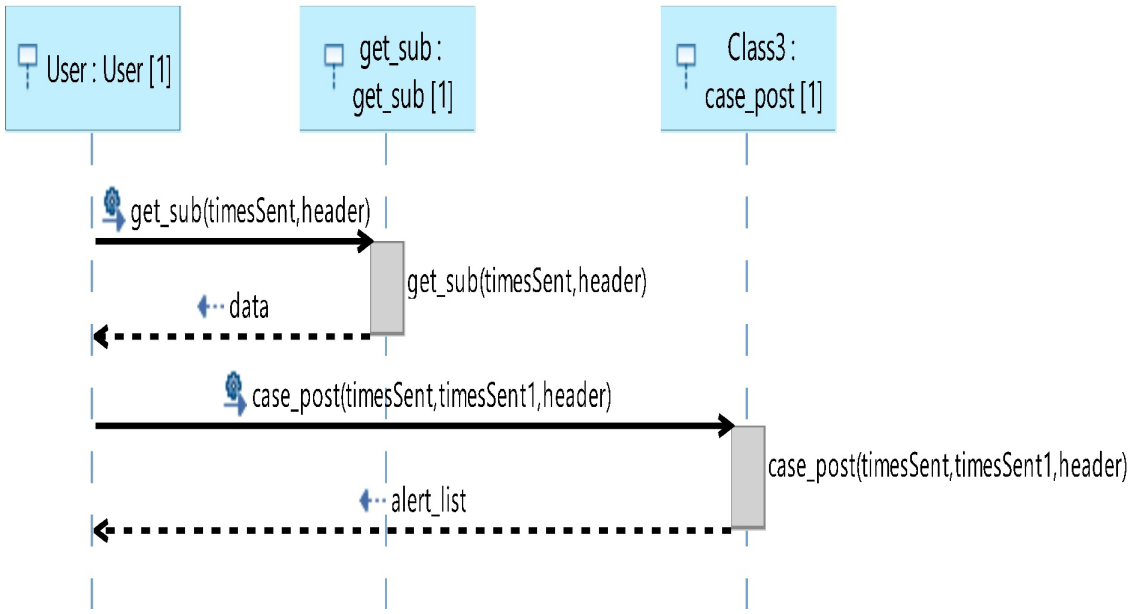


6.2.1.6 POST request

Come riportato precedentemente, il comando di tipo POST in questo caso lo effettua l'*Orion Context Broker* sotto forma di notifica, come conseguenza della *subscription* creata precedentemente. Il diagramma delle sequenze in questo caso è composto dalle seguenti funzioni:

- ***get_ sub***, utilizzato per gli stessi motivi per cui è utilizzato nel caso di GET

Figura 6.6: Diagramma delle sequenze del metodo POST riferito all'estensione *main*



- ***case_post***: ogni parametro sarà controllato attraverso le specifiche nominate nel file *Threshold_WaterStation.csv*.
  In base al valore di *limit*, da CrateDB saranno estratti un determinato numero di valori ed il codice controllerà prima l'ultimo valore per verificare o meno la condizione di Alert, poi, in base all'esito, controllerà i *limit* parametri precedenti per poi confrontare i numeri di superamento soglia con *counter* o *counter_exit*.

```
    #utilizzo contatore per verifica futura su "counter" e "counter\_exit"
  for x in parametri:

    if int(x[1]) > int(threshold[header1.index(name_param)]):
                cont+=1
 #se il valore pi  recente supera la soglia -> studio Alert

    if int(parametri[0][1]) > int(threshold[header1.index(name_param)]):

    #controllo sul trend parametri. Tale controllo    utile ai fini del contenuto dell'alert
     if cont > int(limit[header1.index(name_param)]):

       text=f"Anomaly in the trend param of {name_param}, exceeding threshold confirmed"
       alert_list.append(text)
```

```
14          email_notification(text)
15
16      else:
17          text=f"Anomaly value of {name_param}, not anomaly trend.
        Only {cont} exceeding."
18          alert_list.append(text)
19          email_notification(text)
20
```

In modo complementare, un controllo analogo è effettuato per verificare se esiste o meno la condizione di uscita dal Wait, se ancora il trend non è uscito da una situazione di pericolo la condizione di Wait sarà aggiunta nell'*alert_list*.

- ***email_notification***: funzione che implementa l'invio di una mail con contenuto l'Alert/Wait creato. Le mail destinatarie potranno essere scritte dall' user nell'estensione */settings*, descritto nei prossimi paragrafi.

La stringa di ritorno dalla funzione *case_post* sarà utilizzato come messaggio di ritorno dall'intero caso POST; può essere un "*not ok*" se non ci sono stati Alert/Wait, oppure ritorna l'intero array di Alert/Wait.

6.2.2 /analysis

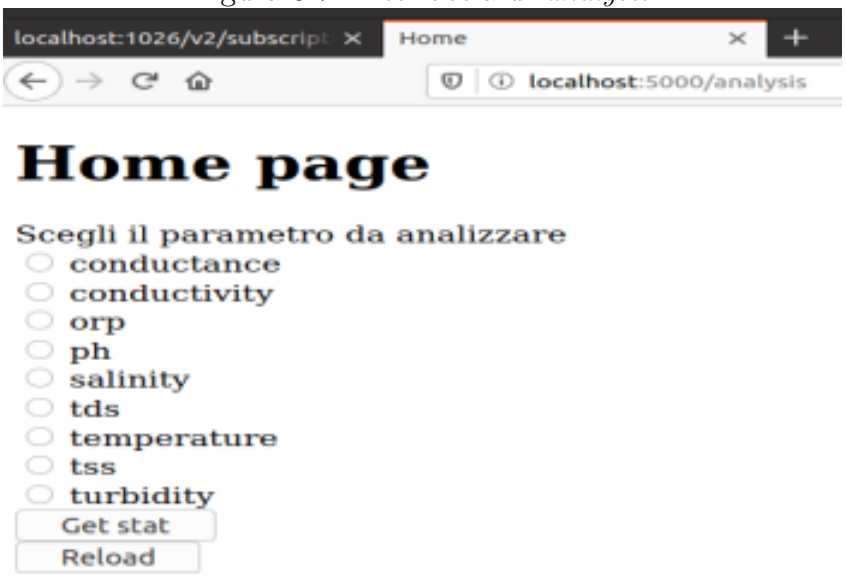
Knowage possiede delle ottime potenzialità nell'analisi statistica/grafica di valori numerici spazio-temporali, con cui è possibile compiere medie, creare istogrammi, grafici a torta e così via.
Dato che il *Web Service* creato è più focalizzato sulla gestione degli Alert e di notificazione utente, è stata implementata una semplice ed intutiva pagina web che calcolasse dati statici per ogni parametro.
L'estensione */analysis* permette all'utente di scegliere il parametro da controllare e restituisce valori statistici, riferiti agli ultimi *limit*, come:

- valor medio (*Avg*)
- valore massimo (*Max*)
- grado di emergenza nel controllo valori (*status*)

6.2.2.1 GET request

La richiesta di tipo GET in questo caso non richiama nessuna funzione esterna poiché l'unica azione effettuata è la creazione di un codice *HTML* che rappresenta graficamente una visuale più intuitiva possibile per l'utente. É stata inserita la possibilità di ricaricare la pagina, tramite il tasto *Reload*, oppure di visualizzare le statistiche di ogni parametro, selezionando il parametro dall'elenco e premendo il tasto *Get stat*.

Figura 6.7: Interfaccia di *analysis*



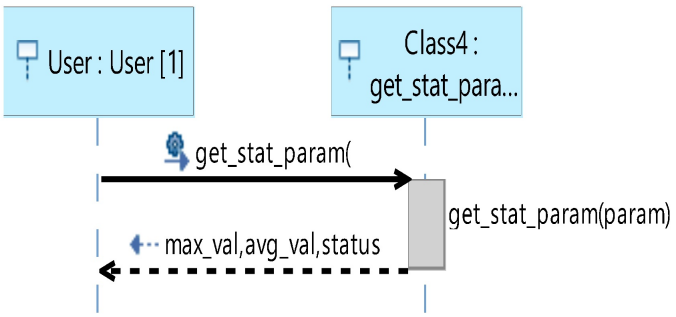
6.2.2.2 POST request

Il comando sopracitato (*Get stat*) è l'input per la richiesta di tipo POST: tramite il comando

```
param=request.form.get('parameter') #i parametri sono associati
    tutti alla variabile 'parameter'
```

è possibile risalire alle statistiche di quel parametro.
Successivamente è richiamata la funzione *get_stat_param* che ottiene le stati-

Figura 6.8: Diagramma delle sequenze di tipo POST per l'estensione *analysis*



stiche inerenti al parametro richiesto tramite classici algoritmi di calcolo.
Infine lo script ricerca nell'entità se esiste un attributo che inizi per 'Alert{name_param}' o per 'Wait_{name_param}'; a seconda della risposta restituirà un valore di status con un determinato colore associato:

- ***"Sotto controllo"*** colorato in verde: non è stato trovato nessun attributo con caratteristiche compatibili
- ***"Attesa uscita"*** colorato in giallo: è stato trovato un attributo che inizia per 'Wait_name_param'
- ***"Alarm!"*** colorato in rosso: è stato trovato un attributo che inizia per 'Alert{name_param}'

Figura 6.9: Interfaccia di *analysis* dopo la richiesta POST per il parametro *tss*

6.2.3 /settings

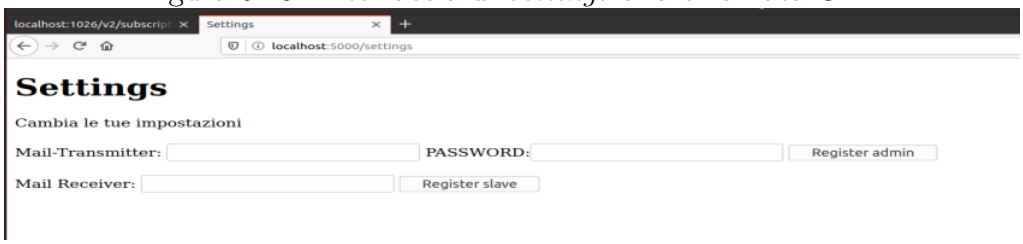
Quest ultimo script risulta fondamentale per la notificazione via mail: come è stato già spiegato in */main*, se ci si trova in una situazione di allarme il *Web Server* invierà una e-mail di Alert/Wait. In questa sessione è possibile decidere quali saranno il mittente ed il destinatario di tali e-mail.

6.2.3.1 GET request

Come per */analysis*, non ci sono funzioni richiamate dallo script poiché l'esecuzione consiste in un semplice codice *HTML* dove è possibile inserire due tipi di e-mail:

- ***Admin***: del mittente si deve conoscere sia la e-mail che la password poiché in Python per inviare una e-mail è necessario effettuare l'accesso nell'account per terze parti. Ogni volta che viene inserito un nuovo account *Admin*, questo sovrascriverà quello precedentemente salvato.
- ***Slave***: del destinatario basta conoscere la e-mail

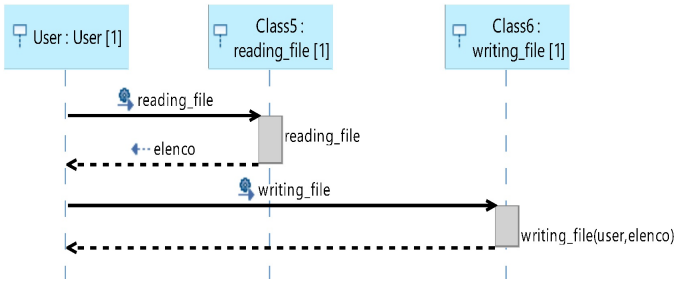
Figura 6.10: Interfaccia di *settings* alla chiamata GET



6.2.3.2 POST request

Anche in questo caso, come per */analysis* la richiesta di tipo POST la si riceve dall'utente all'invio delle informazioni account.

Figura 6.11: Diagramma delle sequenze della richiesta POST di *settings*



Le funzioni richiamate, che sia l'invio di un account "Admin" o "Slave", sono le seguenti:

- ***reading_file***: lettura del file *role_settings.txt*. Il contenuto del file è un array di "Oggetti", tutti composti da 3 variabili: *account* (nome dell'account), *password*, *role*. L'attributo *role* risulta fondamentale per l'invio della mail: se c'è scritto "*admin*", l'account associato sarà il mittente, se c'è scritto "*slave*", l'account associato sarà il destinatario. Il *role* sarà automaticamente assegnato dallo script in base a quale tasto è stato premuto, se *Register slave* o *Register admin*. L'attributo *password* di un account *slave*, sarà sostituito da un '-' di default.

- ***writing_file***: l'attributo *user*, ovvero il nuovo "oggetto" inserito dall'utente, sarà inserito all'interno dell'elenco utenti ed eventualmente sovrascriverà l'account *admin* se il suo *role* è l'omonimo. Lo script infine sovrascriverà l'elenco nel file *role_settings.txt*, affinchè possa essere riletto per una eventuale notifica da inviare.

Figura 6.12: Esempio di elenco di e-mail, sia user che admin

La scrittura sul file è protetta dal sistema di sicurezza di Docker: lavorando in container interconnessi tra loro tramite un *docker network*, le modifiche effettuate con Docker su file di qualsiasi tipo rimangono salvate all'interno del network e non vengono visualizzati dagli utenti che hanno accesso al computer. Per poter visualizzare il contenuto del file è necessario entrare dentro il container ed effettuare il comando

```
cat role_settings.txt
```

Il *Web Service* creato risulta graficamente molto spartano ma consente di realizzare quanto prefissato.
Ovviamente Knowage sarebbe risultato una soluzione migliore in quanto completo e pieno di risorse di sviluppo, ma lo script creato in termini di efficienza correlata all'efficacia è risultato più che adatto ai nostri scopi, soprattutto per la sua elevata velocità di calcolo e snellezza.

Capitolo 7

Conclusioni e sviluppi futuri

7.1 Conclusioni

Lo sviluppo del Web Service tramite l'utilizzo di uno script Python è avvenuto con successo. Analizzando FIWARE, nel suo organismo e nelle sue varie implementazioni nel mondo *smart* è risultato lampante sin da subito che il prodotto offerto è di un valore notevole soprattutto per la sua portabilità in tutte le varie realtà *smart*.
Il problema riscontrato nel percorso di tesi è stato l'utilizzo del framework Knowage che, idealmente, avrebbe implementato all'ennesima potenza l'idea di mondo *smart* e della versatilità di FIWARE. Purtroppo i tentativi di risolvere, o ridurre, i problemi di Knowage sono risultati per lo più vani e con tempistiche troppo estese per un periodo di tirocinio, quindi la scelta più ovvia è stata progettare ad hoc uno script che avesse la stessa funzione di Knowage.
Il risultato, come scritto precedentemente, è stato un successo se si esclude il fatto che la scrittura di uno script ad hoc limita completamente la versatilità che propone FIWARE.

7.2 Sviluppi futuri

7.2.1 Risoluzione bug e-mail

L'unico bug riscontrato nell'utilizzo dello script Python per la costruzione di un Web Service, è inerente all'invio delle e-mail da parte di un admin: anche se l'admin è stato registrato nell'elenco di *role_ settings.txt*, nell'invio di e-mail nel caso di anomalie di parametri, tali messaggi non vengono inviati ai corrispondenti destinatari. Si suppone che questo problema sia dovuta alla containerizzazione dei contatti ed anche delle e-mail prodotte dallo script: probabilmente un Web Service containerizzato non può accedere ad altre pagine Web non containerizzate.

Questo bug, trovato durante l'esecuzione dello script, sarà analizzato in futuro e risolto al fine di completarne la funzionalità del Web Service creato.

7.2.2 Ottimizzazione grafica

Non risultando un prerequisito fondamentale per il progetto, è stato lasciato in secondo piano l'implementazione grafica delle analisi statistiche. L'obiettivo, nel futuro, è rendere più "gradevole" l'interfaccia di visualizzazione per l'utente ovvero:

- aggiungere la possibilità di **rappresentazione grafica temporale** dello sviluppo dei parametri
- permettere di effettuare più **statistiche sui dati** dei parametri
- permettere un **confronto tra più parametri** e dei loro andamenti, normalizzati rispetto ai loro valori massimi. Un esempio lampante di utilità potrebbe essere nel confrontare il *tss* ed il *tds*.

7.2.3 Più funzionalità nel Web Server

Un altro aspetto futuro risulta l'implementazione di ulteriori funzionalità nel Web Server, che girino intorno al mondo del WWTP e della sua gestione, come ad esempio:

- aggiungere la possibilità di poter visualizzare direttamente dalla pagina Web le varie *threshold* dei parametri, ed eventualmente modificarle in diretta
- aggiungere una pagina di ingresso che richieda il login o la registrazione per poter accedere a tutti i contenuti forniti
- aggiungere la possibilità di poter ricevere notifiche sul proprio computer in caso di Alert con trend negativo

Appendice A

Comandi inizializzazione software

Di seguito sono illustrati i comandi da effettuare via terminale per poter avviare in modo corretto il progetto sviluppato. Tutti i comandi illustrati successivamente sono svolti sul terminale Ubuntu 18.04.

A.1 Installazione framework FIWARE

Per un discorso di minor complessità, i comandi saranno tutti effettuati tramite il *Docker-Compose*, che in modo omogeneo, lavorerà su tutti i container in egual modo. Per installare le componenti base di FIWARE, ovvero l'*Orion Context Broker*, *Redis*, il DB statico *MongoDB* e tutto il pacchetto di *Quantum-Leap* occorre effettuare i seguenti passaggi:

- scaricare dal sito citato [16], il file "*docker-compose .yml*"
- accedere tramite terminale alla cartella dove è contenuto il file *.yml*, tramite il comando

```
1    cd {folder}
2
```

- installare tramite *docker-compose* il completo pacchetto di applicazioni.

```
1    docker-compose -f docker-compose-dev.yml up -d
2
```

  Il comando "-d" sta solo a significare che non sarà visualizzato a schermo l'intero processo di installazione delle applicazioni.

A.2 Comandi sulle applicazioni

Attraverso il comando

```
sudo docker-compose ps
```

è possibile visualizzare tutte le applicazioni che in quel momento sono attive, come illustrato di seguito.
La sigla *sudo (switch user do)* è di fondamentale importanza perchè permette, in quanto amministratore, di poter effettuare determinati comandi, come ad esempio lavorare con Docker. Esistono dei comandi che possono essere effettuati su tutti

IMAGE	COMMAND	CREATED	STATUS
smartsdk/quantumleap	"/bin/sh -c 'python …"	2 minutes ago	Up 2 minutes
fiware/orion:1.13.0	"/usr/bin/contextBro…"	2 minutes ago	Up 2 minutes
grafana/grafana	"/run.sh"	2 minutes ago	Up 2 minutes
crate:1.0.5	"/docker-entrypoint.…"	2 minutes ago	Up 2 minutes
mongo:3.2	"docker-entrypoint.s…"	2 minutes ago	Up 2 minutes
redis	"docker-entrypoint.s…"	2 minutes ago	Up 2 minutes

i container contemporaneamente come i classici comandi di start e stop delle applicazioni.

```
sudo docker-compose start
sudo docker-compose stop
```

Alcuni comandi non risultano ottimizzabili con *docker-compose*, il più importante tra questi è *log*: tramite esso è possibile controllare tutto il *log* dei comandi effettuati sull'applicazione ed eventualmente anche degli errori riscontrati.
Un esempio che è risultato molto utile nel progetto è stato con il container CrateDB che aveva spesso dei problemi di occupazione di memoria RAM e quindi nel *log* è stato possibile trovare l'errore. Questo comando è scritto nel seguente modo:

```
sudo docker log tirociniofiware_crate_1
```

dove *tirociniofiware_crate_1* è il container che contiene l'applicazione *CrateDB*.

Bibliografia

[1] http://documenter.getpostman.com/view/513743/fiware-subscriptions/rw1dhetr.

[2] https://curl.haxx.se/docs/manpage.html.

[3] https://developer.mozilla.org/en-us/docs/learn/common_questions/what_is_a_web_server.

[4] https://docs.aws.amazon.com/it_it/apigateway/latest/developerguide/api-gateway-request-throttling.html.

[5] https://en.wikipedia.org/wiki/flask_(web_framework).

[6] https://fiware-datamodels.readthedocs.io/en/latest/environment/waterqualityobserved/doc/spec/index.html.

[7] https://fiware-orion.readthedocs.io/en/master/.

[8] https://fiware-tutorials.readthedocs.io/en/latest/iot-agent/index.html.

[9] https://fiware.github.io/specifications/ngsiv2/stable/.

[10] https://flask.palletsprojects.com/en/1.1.x/.

[11] https://flask.palletsprojects.com/en/1.1.x/foreword/.

[12] https://github.com/fiware/tutorials.time-series-data.

[13] https://ke.linkedin.com/company/binaryupdates-com.

[14] https://knowage.readthedocs.io/en/latest/.

[15] https://knowage.readthedocs.io/en/latest/user/ngsi/readme/index.html.

[16] https://quantumleap.readthedocs.io/en/latest/admin/.

[17] https://quintagroup.com/cms/python/flask.

[18] https://schema.org/car.

[19] https://whatis.techtarget.com/definition/data-ingestion.

[20] https://www.clarion.com/xe/en/corp/technology/solutions/.

[21] https://www.comindware.com/blog-what-is-visual-workflow-management-and-how-can-it-help-your-business/.

[22] https://www.digital-water.city/.

[23] https://www.docker.com/why-docker.

[24] https://www.fiware.org/developers/.

[25] https://www.slideshare.net/ow2/knowage-open-source-business-analytics-suite-presented-at-ow2con19-june-1213-2019-paris.

[26] http://www.giuneco.tech/application-container/.

Elenco delle figure