



Università del Salento

Dipartimento di Ingegneria dell'Innovazione Big Data Management

Progetto IoT Simulator "Powered By Fiware"

Documentazione del Progetto

Docenti corso

Prof.sa Antonella Longo

Prof.re Marco Salvatore Zappatore

Membri del gruppo

Cezar Narcis Culcea

Francesco De Nunzio

Filippo Goffredo

ANNO ACCADEMICO 2022/2023

Indice

1	Traccia	4
2	Architettura	5
2.1	Architettura Logica	5
2.2	Architettura Fisica	6
3	Sviluppo	7
3.1	Data Models	7
3.2	Casi d'Uso	8
3.2.1	Generazione dati	8
3.2.2	Analisi dati	8
3.2.3	Esposizione risultati	8
3.3	Diagramma di Sequenza	9
3.3.1	Generazione dati	9
3.3.2	Analisi dati	9
3.3.3	Esposizione risultati	10
4	Componenti	11
4.1	Orion	11
4.2	MongoDB	11
4.3	Analysis	12
4.3.1	Main Process	12
4.3.2	Flask Server Process	14
4.4	Sensor	15
4.5	Node.js Web App	16
5	Docker Deployment	19
5.1	GitHub Repository	19
5.2	Docker-Compose Files	19
5.3	Services.sh	20

5.4	How to use it	21
5.4.1	Setup	22
5.4.2	Avvio	22
5.4.3	Container Sensor	22
5.4.4	Container Analysis	23
5.4.5	Container MongoDB	24
5.4.6	Container Node.js Web APP	24
5.4.7	Stop	26
5.4.8	Purge	26
5.4.9	Link Repository GitHub	26

Capitolo 1

Traccia

Progettazione e realizzazione di un prototipo software per la simulazione del comportamento di rilevamento di uno specifico sensore (ad esempio, contatore elettrico intelligente, **termo-igrometro** per interni/esterni, ecc.).

In particolare, il simulatore non dovrà essere utilizzato per attività complesse come il test funzionale, il test delle prestazioni o la simulazione della connettività, ma dovrà funzionare come fonte di flussi di dati del sensore adatti al contesto in esame (ad esempio, generazione di misure compatibili con la grandezza fisica monitorata, temporizzate e caratterizzate dalla possibilità di introdurre elementi di perturbazione deterministici o stocastici, ecc).

Deve essere definita l'**architettura logica e fisica** del sistema è richiesta la presenza di enabler generici e di enabler specifici dal catalogo del framework FIWARE.

Gli abilitatori FIWARE devono essere selezionati e integrati nell'architettura in modo coerente con l'obiettivo prefissato (ad esempio, deve essere utilizzato sicuramente il **Context Broker**).

L'architettura deve avere almeno un livello di **ingestione dei dati** e un livello di **elaborazione dei dati**.

Le soluzioni di archiviazione per l'ingestione dei dati e per le fasi successive all'elaborazione dei dati devono essere compatibili sia con i tipi di dati considerati sia con la modalità di generazione dei dati adottata, oltre che con i requisiti del framework FIWARE.

Per l'elaborazione dei dati può essere utilizzato Apache Spark o un'altra soluzione simile.

La scelta del linguaggio di programmazione da utilizzare nella fase di implementazione non è vincolante.

Capitolo 2

Architettura

2.1 Architettura Logica

Al fine di soddisfare le richieste, abbiamo ideato la seguente architettura logica:

- Sensore: un componente che genera i dati e li manda al Context Broker
- Context Broker: riceve i dati e notifica gli enabler
- NoSQL Database: supporto di memorizzazione dati
- Generic Enabler: riceve le notifiche dal Context Broker
- Motore di analisi dati: analisi dei dati
- Web Application: dashboard con i risultati

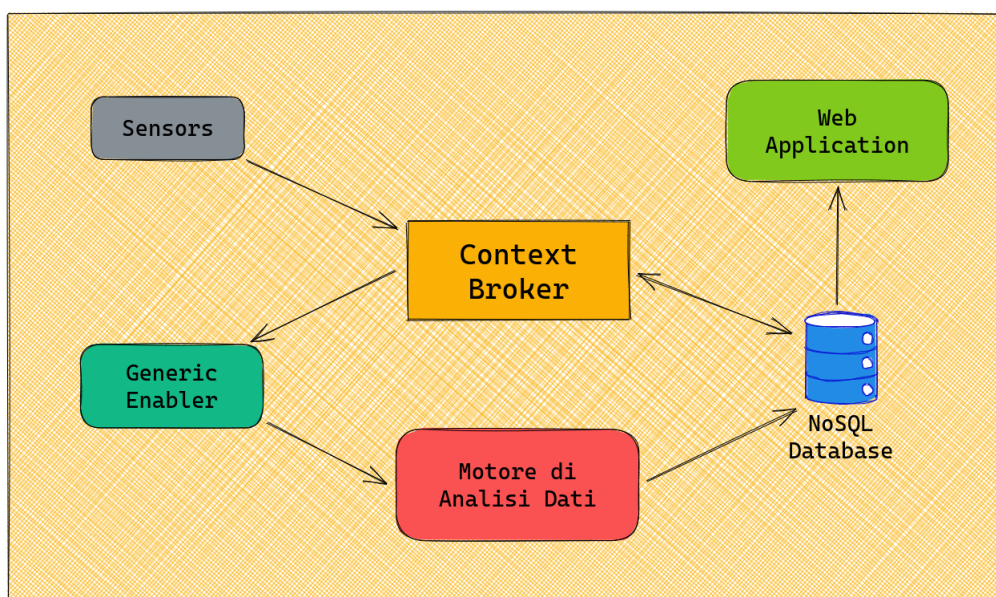


Figura 2.1: Architettura Logica

2.2 Architettura Fisica

L'implementazione dell'architettura logica è la seguente:

- Sensor.py: uno script python simula il comportamento dei sensori, inviandoli al Context Broker. In particolare genererà dati di **temperatura e livello** dell'acqua
- Fiware Orion CB: riceve i dati e notifica gli enabler, cuore di un applicazione "Powered By Fiware"
- MongoDB: un database NoSQL documentale semplice da accedere
- Analysis.py: un applicativo python che genera un Flask server, con funzione di Generic Enabler, ed un Main Process che si occupa dell'analisi
- Web Application Component: un insieme di Front-end (HTML/CSS/JS) e Back-end (Node.js) con lo scopo di visualizzare le elaborazioni

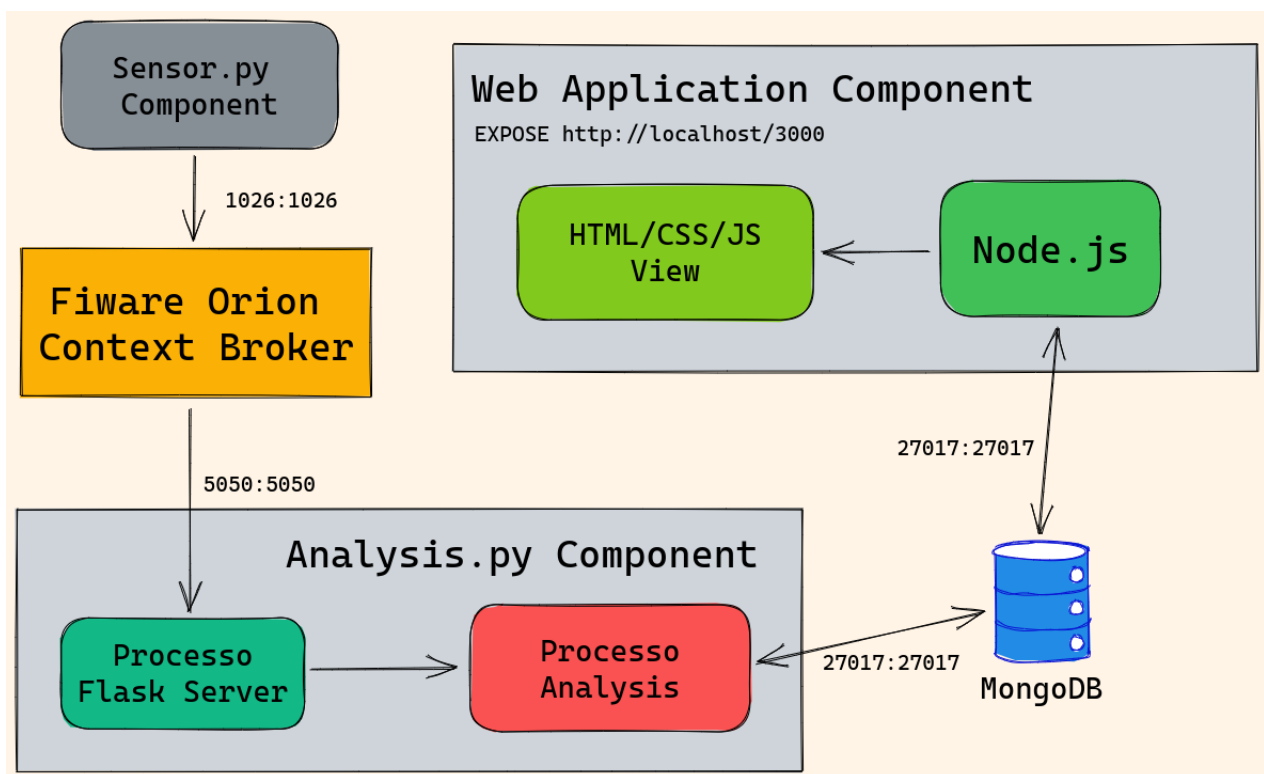


Figura 2.2: Architettura Fisica

Capitolo 3

Sviluppo

3.1 Data Models

Al fine di poter comunicare con **Fiware Orion CB** è stato necessario fare uso di specifici **Data Models**. Quest'ultimi sono stati selezionati attraverso una ricerca sulla repository pubblica degli Smart Data Models. Abbiamo selezionato il Data Model relativo all'Environment in quanto disponeva di tutti i campi necessari a salvare i dati dei nostri sensori.

Data Model Environment: <https://github.com/smart-data-models/dataModel.Environment>

```
1  {
2    "id": "urn:ngsi:WaterObserved:MNCA-001",
3    "type": "WaterObserved",
4    "dateObserved":
5    {
6      "type": "Property",
7      "value": {
8        "@type": "DateTime",
9        "@value": "2017-01-31T00:00:00Z"
10     }
11   },
12   "height":
13   {
14     "type": "Property",
15     "value": 80
16   },
17   "temperature":
18   {
19     "type": "Property",
20     "value": 50
21   },
22   "@context":
23   [
24     "https://raw.githubusercontent.com/smart-data-models/dataModel.Environment/master/context.jsonld"
25   ]
26 }
```

Figura 3.1: Esempio di file jsonld che rappresenta l'entità sensore

In particolare i campi di nostro interesse saranno:

- id: identificativo entità
- type: tipo di entità (indicate sui context.jsonld)
- dateObserved: data della rilevazione
- height: livello dell'acqua
- temperature: livello della temperatura
- context: link al context con informazioni sui campi

3.2 Casi d'Uso

3.2.1 Generazione dati

Attori: Sensor.py, Orion CB, MongoDB, Flask Process

Pre-test: Subscription effettuata dal Server Flask a Orion

1. Il sensore simula e manda i dati al Context Broker
2. Il Context Broker salva i dati su MongoDB
3. Il Context Broker notifica il server Flask inviandogli i dati

3.2.2 Analisi dati

Attori: processo Server Flask, processo di Analisi, MongoDB

Pre-test: Server Flask ha ricevuto i dati dal Context Broker

1. Il server Flask manda i dati al processo di Analisi
2. Il processo di Analisi analizza i dati ed imposta un flag al fine di distinguere gli outliers dai valori normali
3. Il processo di Analisi invia i dati analizzati a MongoDB

3.2.3 Esposizione risultati

Attori: Node.js, View, MongoDB

Pre-test: Web Application disponibile all'indirizzo <http://localhost:3000/>

1. Node.js effettua una query dei dati desiderati su MongoDB
2. Node.js invia i dati alla View
3. View organizza i dati e li presenta all'utente finale

3.3 Diagramma di Sequenza

3.3.1 Generazione dati

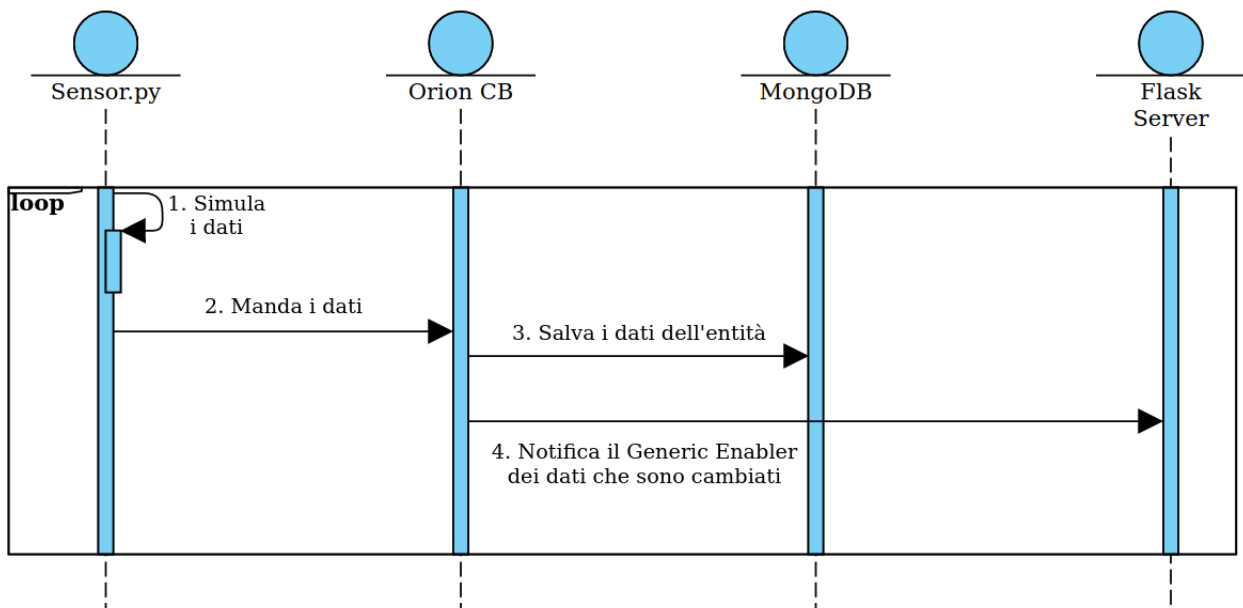


Figura 3.2: Diagramma di sequenza della generazione dei dati

3.3.2 Analisi dati

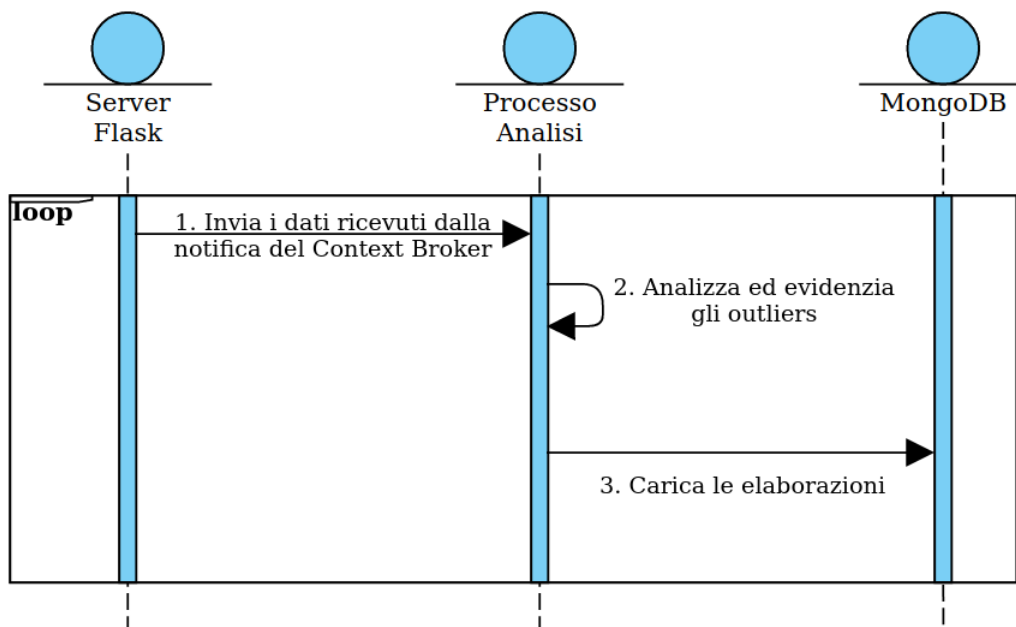


Figura 3.3: Diagramma di sequenza dell'analisi dei dati

3.3.3 Esposizione risultati

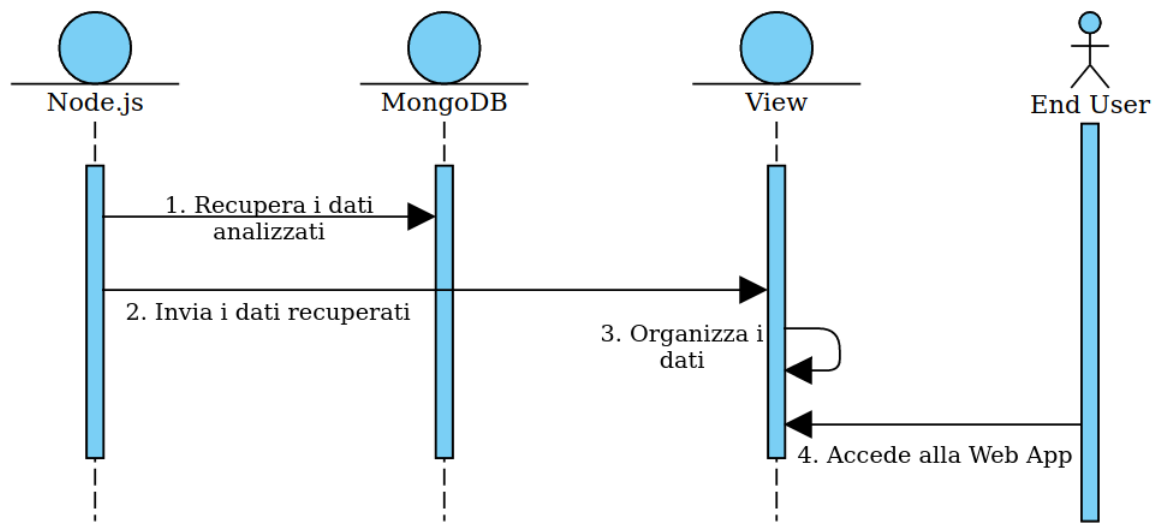


Figura 3.4: Diagramma di sequenza dell'esposizione dei dati

Capitolo 4

Componenti

4.1 Orion

Orion Context Broker è un software **open source** per la gestione delle **Context Information** nell'ambito dell'Internet of Things (IoT). Consente di memorizzare e recuperare informazioni (la posizione di un dispositivo, la temperatura di una stanza, ecc) al fine di renderle disponibili ad altre applicazioni attraverso interfacce standard.

Le API **NGSI-LD** sono un'estensione di Orion Context Broker che utilizza il modello di dati NGSI-LD (NGSI Linked Data), basato sugli standard **Linked Data** del W3C. Questo permette di integrare i dispositivi e le applicazioni IoT con il più ampio ecosistema Linked Data, consentendo capacità di elaborazione e interrogazione dei dati più potenti e flessibili.

Uno dei punti di forza di Orion sono le **Subscription**: esse consentono alle applicazioni di ricevere notifiche quando cambiano determinate informazioni sul contesto. Ciò consente alle applicazioni di essere più reattive ai cambiamenti dell'ambiente e di intraprendere azioni basate su questi. Le subscription possono essere basate su diversi criteri, come il valore di un attributo, la posizione di un'entità o una combinazione di entrambi.

Nel nostro caso la subscription notificherà il *Generic Enabler* per ogni nuova misura dei sensori.

4.2 MongoDB

MongoDB è un popolare sistema di gestione di database (DBMS) **open-source** orientato ai documenti che utilizza il formato BSON (**binary JSON**) per memorizzare i dati. È progettato per gestire grandi quantità di dati su più server, il che lo rende adatto all'uso in applicazioni web big data e in tempo reale. Una delle caratteristiche principali di MongoDB è il supporto per lo **scaling orizzontale**, che consente di gestire facilmente grandi quantità di dati distribuendoli su più server. Inoltre, il modello di dati orientato ai documenti di MongoDB consente una gestione dei dati più flessibile ed efficiente rispetto

ai database relazionali tradizionali, in quanto permette di memorizzare strutture di dati annidate e di interrogare e indicizzare campi specifici nei documenti.

MongoDB supporta anche un ricco linguaggio di query, indici secondari e ricerca full-text, e fornisce un supporto integrato per gli insiemi di repliche e lo sharding automatico.

Il motivo principale per cui è stato scelto è la semplicità di utilizzo dei file JSON e delle librerie Python che lavorano con MongoDB. Inoltre è bene considerare che Orion è estremamente ottimizzato per cooperare con MongoDB.

La nostra istanza di MongoDB contiene i dati del progetto nel database denominato **orion**. Esso inoltre ospita le seguenti collezioni:

- **csubs**: collezione usata dal Context Broker per memorizzare le Subscriptions
- **entities**: collezione usata dal Context Broker per tenere traccia delle entità
- **measurements**: collezione usata dal componente di Analisi per memorizzare i risultati delle elaborazioni
- **means**: collezione in cui vengono salvati i JSON con i valori medi di temperatura e livello dell'acqua che ci si aspetta nelle diverse ore della giornata

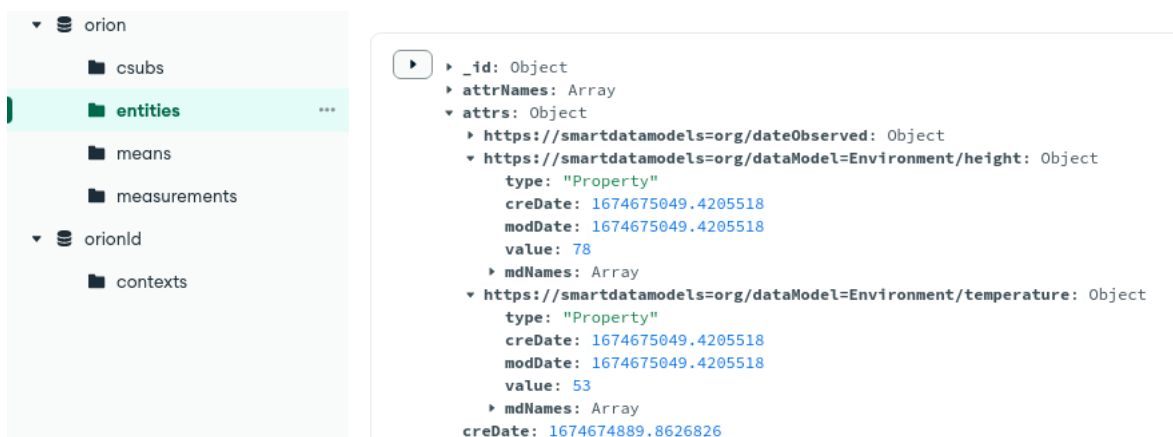


Figura 4.1: Collezioni interne al DB orion mostrate su Compass

4.3 Analysis

Lo script Python Analysis.py svolge diverse funzioni, originando due processi distinti al suo avvio (Main Process e Flask Server)

4.3.1 Main Process

Il primo compito del Main process è eseguire con successo la **subscription** al Context Broker, effettuando una POST ad Orion.

Nel JSON della richiesta POST si distinguono i seguenti campi:

- entities: indica il tipo di entità a cui ci si riferisce
- watchedAttributes: indica a quali attributi si è interessati, in modo da ricevere la notifica in caso di cambiamento
- uri: indica l'endpoint a cui inviare la notifica, nel nostro caso il Flask Server
- context: il contesto a cui fa riferimento l'entità

```
29 def subscribe_to_orion(orion_url):
30     headers = {'Content-type': 'application/ld+json'}
31     subscription = {
32         "description": "Notifica cambiamento dati di temperatura e livello acqua",
33         "type": "Subscription",
34         "entities": [{"type": "WaterObserved"}],
35         "watchedAttributes": ["dateObserved"],
36         "notification": {
37             "format": "keyValues",
38             "endpoint": {
39                 "uri": "http://analysis:5050/notification",
40                 "accept": "application/json"
41             },
42         },
43         "@context": "https://raw.githubusercontent.com/smart-data-models/dataModel.Environment/master/context.jsonld"
44     }
45     response = requests.post(orion_url, json=subscription, headers=headers)
```

Figura 4.2: Metodo di creazione della Subscription

Una volta completata la subscription, il main process estrae i valori medi da MongoDB, conservandoli in un dizionario.

Successivamente il Main process avvia in parallelo il Server Flask, e crea una **pipe** sulla quale si mette in attesa di ricevere dati dal server.

Quando il server Flask riceve i dati, li invia sulla pipe al Main process, che procederà con l'**analisi**.

Lo scopo del processo di analisi è identificare gli outliers, impostando il relativo flag:

- 0: se il valore misurato rispecchia la media
- 1: se il valore misurato si discosta più del 10% rispetto alla media

```
93 logging.info("----- Data: {} -----".format(date))
94 # controllo se sono outliers
95 height_flag = 0
96 if height <= means[0]/100*90 or height >= means[0]/100*110:
97     height_flag = 1
98     logging.info("OUTLIER | Livello - Valore: {} Media: {} - Differenza: {}".format(height, means[0], height-means[0]))
99 else:
100     logging.info("NORMALE | Livello - Valore: {} Media: {} - Differenza: {}".format(height, means[0], height-means[0]))
101
102 temp_flag = 0
103 if temp <= means[1]/100*90 or temp >= means[1]/100*110:
104     temp_flag = 1
105     logging.info("OUTLIER | Temperatura - Valore: {} Media: {} - Differenza: {}".format(temp, means[1], temp-means[1]))
106 else:
107     logging.info("NORMALE | Temperatura - Valore: {} Media: {} - Differenza: {}".format(temp, means[1], temp-means[1]))
108 logging.info("-----")
```

Figura 4.3: Logica dell'identificazione degli outliers

Infine il Main process provvederà ad aggiungere dati relativi all'analisi della misura ed a caricarli su MongoDB.

```
57     json_file = {
58         "id": "urn:ngsi:WaterObserved:MNCA-001",
59         "type": "Water",
60         "height": {
61             "type": "Property",
62             "value": height
63         },
64         "heightFlag": {
65             "type": "Property",
66             "value": height_flag
67         },
68         "observedDate": {
69             "type": "DateTime",
70             "value": date
71         },
72         "temperature": {
73             "type": "Property",
74             "value": temp
75         },
76         "temperatureFlag": {
77             "type": "Property",
78             "value": temp_flag
79         }
80     }
```

Figura 4.4: JSON risultante dall'analisi di ogni misura generata dal sensore

4.3.2 Flask Server Process

Il Server Flask ha lo scopo di ricevere le notifiche mandate dal Context Broker e di inviare i dati tramite pipe al processo principale.

Il server Flask è disponibile alla porta 5050. I seguenti **endpoint** sono implementati:

- `http://localhost:5050/notification`: mostra la lista delle ultime 24 notifiche ricevute
- `http://localhost:5050/measure`: mostra la lista delle ultime 24 analisi effettuate

La logica che lo gestisce è visibile nel seguente snippet di codice.

```
145 def server_process(conn, measurements):
146     stored_data = []
147
148     @app.route('/notification', methods=['GET', 'POST'])
149     def handle_notification():
150         if request.method == 'POST':
151             data = request.get_json() # legge il json del post
152             stored_data.append(data) # salva i json ricevuti
153
154             if len(stored_data) == 25:
155                 del stored_data[0] # fa in modo che salvi solo gli ultimi 24 json
156                 conn.send(data) # manda il json al main process tramite pipe
157                 return "Data received\n"
158             elif request.method == 'GET':
159                 if len(stored_data) == 0:
160                     return "Nessun dato ricevuto :("
161                 else:
162                     return jsonify(stored_data[::-1])
163
164     @app.route('/measure', methods=['GET'])
165     def handle_measure():
166         if len(measurements) == 0:
167             return "Nessun dato analizzato :("
168         else:
169             return jsonify(measurements[::-1])
170
171     app.run(host='analysis', port=5050)
```

Figura 4.5: Codice del processo Server Flask

4.4 Sensor

Lo script `Sensor.py` ha il compito di simulare il comportamento dei sensori. All'avvio genera l'entità sensore effettuando una richiesta al Context Broker.

La richiesta non è altro che una **POST** effettuata all'indirizzo del Context Broker Orion (`http://orion/1026/ngsi-ld/v1/entities/idEntità`). Il payload è un JSON con la stessa struttura del Data Model predentemente illustrato.

Successivamente lo script procede con l'aggiornamento degli attributi effettuando una richiesta **PATCH** all'indirizzo `http://orion:1026/ngsi-ld/v1/entities/idEntità/attrs/`, includendo nel payload un JSON contenente solo i campi da aggiornare.

```
16 def updateEntity(temp, water_level, date_time):
17     headers = {"Content-Type": "application/json"}
18     data = {
19         "https://smartdatamodels.org/dateObserved":
20         {
21             "type": "Property",
22             "value": {
23                 "@type": "DateTime",
24                 "@value": date_time
25             }
26         },
27         "https://smartdatamodels.org/dataModel.Environment/height":
28         {
29             "type": "Property",
30             "value": round(water_level*100)
31         },
32         "https://smartdatamodels.org/dataModel=Environment/temperature":
33         {
34             "type": "Property",
35             "value": round(temp)
36         }
37     }
38     # invio a orion
```

Figura 4.6: Struttura del payload della richiesta PATCH

I dati sono simulati a partire da un valore iniziale (valore atteso). Successivamente, seguendo una distribuzione **gaussiana**, si effettua una variazione del valore precedente: le misure risultano quindi **correlate**. A scopo simulativo si prevede, con una probabilità del 5%, l'esistenza di un outlier.

```
113 # Water level sensor simulation
114 previous_water_level = None
115 def water_level_sensor(current_time):
116     global previous_water_level
117     if previous_water_level is None:
118         water_level = 0.9
119     else:
120         water_level = previous_water_level + random.normalvariate(0, 0.02)
121         water_level = min(max(water_level, 0.6), 1.0)
122     previous_water_level = water_level
123     # Add outlier with 5% probability
124     if random.random() <= 0.05:
125         water_level += random.uniform(0.1, 0.3)
126
127     return water_level
```

Figura 4.7: Metodo di generazione della misura del livello dell'acqua. La temperatura è generata in maniera simile

4.5 Node.js Web App

La Web Application su cui visualizzare le dashboard con i valori provenienti dai sensori, dopo essere stati analizzati, si basa su Node.js.

Node.js è un runtime di JavaScript open source **multiplatforma**, nato nel 2009, che permette agli sviluppatori di utilizzare questo linguaggio anche nel back-end. Node.js utilizza l'architettura **“Single Threaded Event Loop”** per gestire più client allo stesso tempo. Essa si contrappone ad altri runtime che gestiscono le richieste tramite molte thread concorrenti.

In un “normale” modello di richiesta-risposta multithread, per ogni richiesta inviata da ciascun client, il server assegna una thread differente, allo scopo di gestire le chiamate in maniera concorrente. Node.js, invece, mantiene un pool di **thread limitato** e, quando arriva una richiesta, la mette in una coda: verrà assegnato un thread solo alle richieste che necessitano di operazioni di I/O bloccanti. In questo modo Node.js utilizza un minor numero di risorse, rendendo l'applicazione più **leggera**.

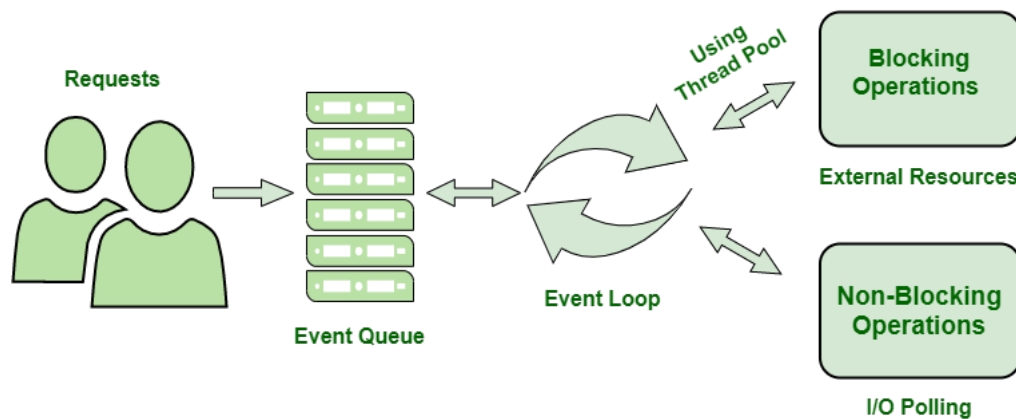


Figura 4.8: Architettura di Node.js

Queste caratteristiche rendono Node.js perfetto per applicazioni **realtime**, come quelle nell'ambito dell'IoT. In genere in queste applicazioni sono presenti molti sensori che generano “piccoli” dati, ma con grande frequenza, rendendo necessario sia la velocità che la “leggerezza” dell'applicazione.

Il front-end, ovvero l'interfaccia utente dell'applicazione è scritto in HTML, CSS e JavaScript. La libreria alla base della UI è **Chart.js**, che permette di creare diversi tipi di grafici personalizzati con caratteristiche interessanti.

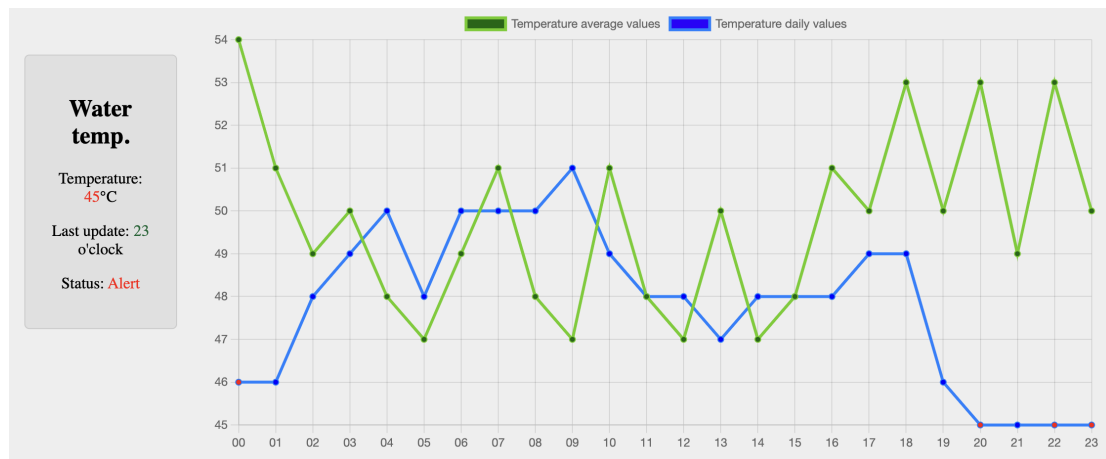


Figura 4.9: Esempio di utilizzo di Chart.js

L'applicazione web mostra all'utente finale la dashboard con i valori misurati dai sensori e analizzati dalla componente di analisi. Node.js gestisce richieste HTTP da parte del front-end e dell'utente finale; un semplice esempio è la richiesta GET per ottenere il dato più recente (analizzato) dal database:

```
app.get('/live_temp', (req, res) => {
  Sensor1.findOne({'heightFlag.value': { $ne: -1 } }).sort({'observedDate.value': -1}).exec((error, result) => {
    if (error) return console.error(error);
    res.json(result);
  });
});
```

Figura 4.10: Logica dell'endpoint "/live_temp"

L'applicazione risponde all'endpoint /live_temp effettuando una query a Mongo DB, estraendo il valore con "observedDate" più recente, e verificando che questo sia stato analizzato (controllando che il flag relativo allo status sia diverso da -1).

Un altro esempio di GET request è quello in cui, fissando una specifica data, si estraggono dal database tutte le misure del sensore del relativo giorno.

```
app.get('/date/:date', (req, res) => {
  const date = req.params.date;
  const startOfDay = moment.utc(`${date}T00:00:00Z`, 'YYYY-MM-DDTHH:mm:ssZ').startOf('day').toISOString();
  const endOfDay = moment.utc(`${date}T23:59:59Z`, 'YYYY-MM-DDTHH:mm:ssZ').endOf('day').toISOString();

  Sensor1.find({
    "observedDate.value": {
      $gte: startOfDay,
      $lt: endOfDay
    }
  }).sort({ "observedDate.value": 1 }).exec((error, result) => {
    if (error) return console.error(error);
    res.json(result);
  });
});
```

Figura 4.11: Logica dell'endpoint "date"

La "view" sfrutta gli endpoint descritti allo scopo di mostrare all'utente finale i dati a cui può essere interessato. I grafici presenti nell'applicazione sono dei "line-chart" che rappresentano le misure

di **temperatura** e del **livello** dell'acqua di una cisterna.

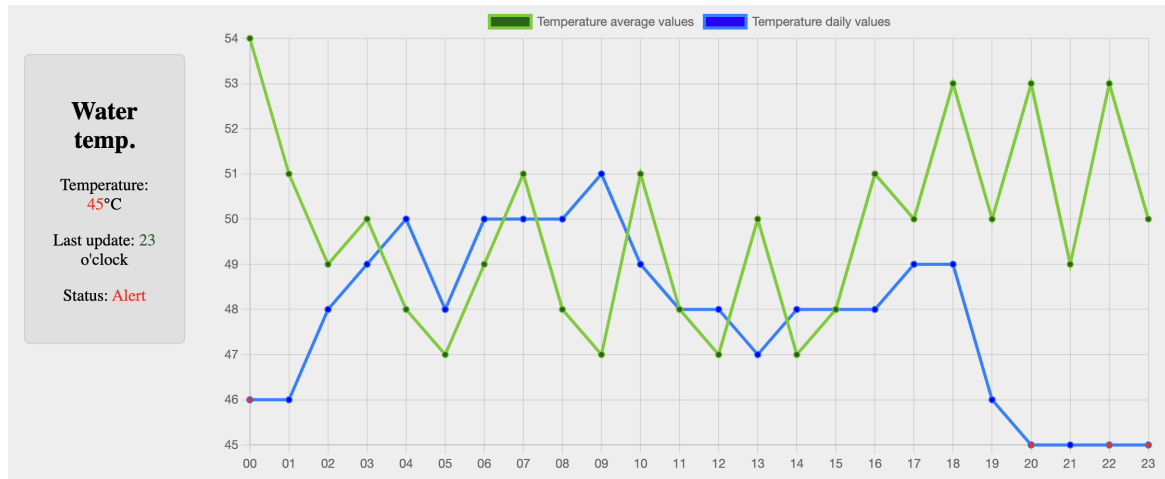


Figura 4.12: Grafico della temperatura

Ciascun grafico è composto da una rappresentazione dei valori medi in verde (basata su precedenti misure) e dai valori in tempo reale in blu, campionati ogni ora.

I valori sono rappresentati solo in seguito all'analisi, allo scopo di illustrare al meglio situazioni in cui sono presenti degli “**outlier**”, ovvero misure anomale al di fuori della media.

In un grafico in cui sono presenti valori che si discostano molto dalla media, si possono notare infatti dei campioni evidenziati in **rosso**, che evidenziano le anomalie. E' anche possibile visualizzare solo la parte del grafico a cui siamo interessati, isolando i valori medi oppure quelli in tempo reale. Questo è reso possibile dall'interazione con i pulsanti rettangolari di colore verde e blu.

Capitolo 5

Docker Deployment

5.1 GitHub Repository

L'intero progetto si trova al seguente indirizzo e si può scaricare liberamente tramite il comando:

```
git clone https://github.com/francesco-denu/Fiware_IoT_sensor_analysis.git
```

La struttura delle directory è organizzata nel seguente modo:

- analysis: directory contenente il codice del container che svolge le funzioni di **analisi**, i suoi requirements.txt ed il suo Dockerfile
- application: directory contenente il codice del container che svolge le funzioni **recupero ed esposizione** dei risultati, le sue dipendenze ed il suo Dockerfile
- docker-compose: directory contenente i file docker-compose.yml necessari al funzionamento dei container
- sensor: directory contenente il codice del container che svolge le funzioni di **simulazione dati**, i suoi requirements.txt ed il suo Dockerfile
- services.sh: script bash che si occupa di gestire con comodità l'intero progetto

5.2 Docker-Compose Files

La directory docker-compose include due file: docker-compose.yml e dc.yml

Per mantenere ordinato il tutto, abbiamo deciso di gestire i due file nel seguente modo:

- docker-compose.yml : contiene le informazioni utili al fine del Pull delle immagini **Fiware** e della costruzione della **network** (docker-compose_default) che ospiterà l'intero progetto.
- dc.yml : contiene le informazioni utili al build delle immagini dei container (Analysis, Sensor, Web App) che abbiamo realizzato al fine di essere usati in cooperazione con le componenti Fiware

```

1  version: "3.8"
2  services:
3    analysis:
4      build: ../analysis
5      container_name: analysis
6      expose:
7        - "5050"
8      ports:
9        - "5050:5050"
10     networks:
11       - fiware_default
12
13    sensor:
14      build: ../sensor
15      container_name: sensor
16      depends_on:
17        - analysis
18      networks:
19        - fiware_default
20
21    app:
22      build: ../application
23      container_name: application
24      depends_on:
25        - sensor
26      expose:
27        - "3000"
28      ports:
29        - "3000:3000"
30      networks:
31        - fiware_default
32
33    networks:
34      fiware_default:
35        external: true
36

```

Figura 5.1: Struttura del file dc.yml

Nella figura precedente notiamo l'uso delle seguenti labels:

- build : indica la posizione del Dockerfile per il build dell'immagine
- container_name : il nome che avranno i nostri container
- expose : la porta alla quale potremmo collegarci dal **localhost** per accedere ai nostri container
- ports : il mapping delle porte su docker
- networks : la rete alla quale i container sono connessi

5.3 Services.sh

Questo shell script è il cuore del progetto. Il suo scopo è chiamare i comandi docker per avviare, fermare, costruire e rimuovere le immagini dei container.

Segue il procedimento necessario per avviare l'applicazione (comando **start**).

```

176  "start")
177    export $(cat .env | grep "#" -v)
178    stoppingContainers # fermiamo i container attivati
179    echo -e "Starting containers: \033[1;33mOrion\033[0m and a \033[1;34mMongoDB\033[0m database."
180    echo ""
181    ${dockerCmd} -f docker-compose/docker-compose.yml up -d --remove-orphans --renew-anon-volumes # avviamo i container Fiware
182    waitForMongo # aspettiamo che Mongo sia online
183    loadData # creiamo le collection e carichiamo i dati dei valori medi
184    addDatabaseIndex # aggiungiamo le collection di orion e gli indici
185    waitForOrion # aspettiamo orion sia online
186    echo -e "Starting containers: \033[1;35mAnalysis\033[0m, \033[1;31mSensor\033[0m, \033[1;36mNode.js Web APP\033[0m"
187    ${dockerCmd} -f docker-compose/dc.yml up -d --renew-anon-volumes # avviamo i container realizzati per il progetto
188    displayServices # mostriamo i servizi disponibili
189    ;;

```

Figura 5.2: Snippet dello script services.sh che avvia il progetto

5.4 How to use it

In questa sezione si illustrerà nel dettaglio il deploy del progetto in un ambiente di sviluppo.

Per il testing sono stati usati i seguenti sistemi:

- ParrotOS Electric Ara 5.1 (debian-based)
 - pip : versione 20.3.4
 - python3 : versione 3.9.2
 - docker : versione 20.10.5+dfsg1
 - docker-compose : versione 1.25.0
 - docker compose : versione 2.14.0
- MacBook Pro M1
 - pip : versione 21.2.4
 - python3 : versione 3.9.2
 - docker : versione 20.10.22
 - docker compose : versione 2.15.1

Nei seguenti passaggi è stato fatto uso della distribuzione linux ParrotOS.

```
git clone https://github.com/francesco-denu/Fiware_IoT_sensor_analysis.git  
cd Fiware_IoT_sensor_analysis/bigData_fiware_latest/
```

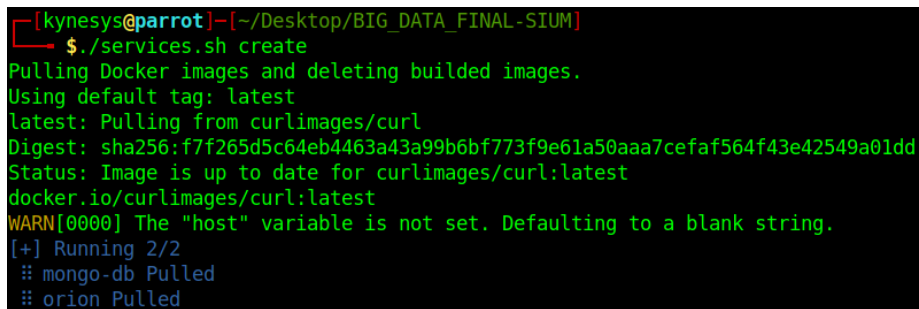
Lo script `services.sh` accetta i seguenti comandi:

- `help` : Mostra i comandi disponibili.
- `create` : Ferma i container. Inizia il pull delle immagini Fiware. Rimuove le build delle immagini personali.
- `start` : Ferma i container. Esegue il build delle immagini personali. Avvia i container.
- `stop` : Ferma i container.
- `purge` : Rimuove tutte le immagini ed i pull eseguiti per il progetto.

5.4.1 Setup

Una volta all'interno della directory di progetto, dopo aver reso eseguibile lo script `services.sh`, si procede con il seguente comando.

```
./services.sh create
```



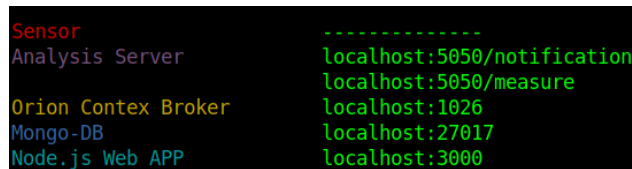
```
[kynesys@parrot]~/Desktop/BIG_DATA_FINAL-SIUM]
$ ./services.sh create
Pulling Docker images and deleting builded images.
Using default tag: latest
latest: Pulling from curlimages/curl
Digest: sha256:f7f265d5c64eb4463a43a99b6bf773f9e61a50aaa7cefaf564f43e42549a01dd
Status: Image is up to date for curlimages/curl:latest
docker.io/curlimages/curl:latest
WARN[0000] The "host" variable is not set. Defaulting to a blank string.
[+] Running 2/2
  :: mongo-db Pulled
  :: orion Pulled
```

Figura 5.3: Output del comando `create`

5.4.2 Avvio

Se il pull è stato eseguito con successo, si può procedere con l'avvio del progetto.

Il comando successivo è `./services.sh start`.



```
Sensor -----
Analysis Server      localhost:5050/notification
                    localhost:5050/measure
Orion Context Broker localhost:1026
Mongo-DB             localhost:27017
Node.js Web APP      localhost:3000
```

Figura 5.4: Indirizzi a cui sono reperibili i container

5.4.3 Container Sensor

Questo container si occupa della simulazione dei sensori, quindi non espone alcun servizio. Tuttavia è possibile accedere ai suoi file di log con il seguente comando: `docker logs sensor`

```

[kynesys@parrot]~/Desktop/BIG_DATA_FINAL-SIUM
$ docker logs sensor
INFO:root:Date and time: 2023-01-26T12:04:28Z Temperature: 50.00C, Water Level: 90.00%
INFO:root:Entità [ID: urn:ngsi:WaterObserved:MNCA-001] creata con successo su Orion Context Broker.
INFO:root:Date and time: 2023-01-26T13:04:28Z Temperature: 51.88C, Water Level: 86.97%
INFO:root:Entità [ID : urn:ngsi:WaterObserved:MNCA-001] aggiornata.
INFO:root:Date and time: 2023-01-26T14:04:28Z Temperature: 49.38C, Water Level: 89.59%
INFO:root:Entità [ID : urn:ngsi:WaterObserved:MNCA-001] aggiornata.
INFO:root:Date and time: 2023-01-26T15:04:28Z Temperature: 52.01C, Water Level: 89.73%
INFO:root:Entità [ID : urn:ngsi:WaterObserved:MNCA-001] aggiornata.

```

Figura 5.5: Log del container sensor

Nella figura precedente si nota la creazione dell'entità e l'aggiornamento dei suoi valori.

5.4.4 Container Analysis

Il container di Analisi espone varie informazioni.

All'indirizzo `http://localhost:5050/notification` sono visualizzate le ultime 24 notifiche ricevute dal Context Broker.

All'indirizzo `http://localhost:5050/measure` sono visualizzate le ultime 24 analisi effettuate.

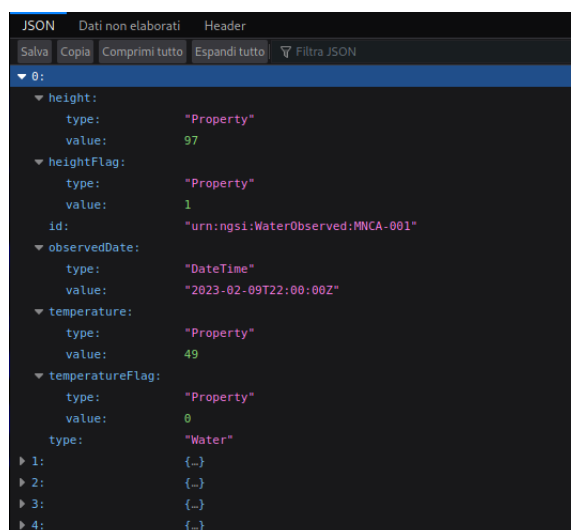


Figura 5.6: Esposizione delle analisi effettuate

E' possibile anche accedere ai file di log con il seguente comando: `docker logs analysis`. Esso mostrerà alcune informazioni come le richieste POST e GET, lo stato della subscription, le analisi che effettua e gli invii dei dati a MongoDB.

```

[kynesys@parrot]~/Desktop/BIG_DATA_FINAL-SIUM]
$docker logs analysis
INFO:root:Subscription created successfully
INFO:root:Server Flask pronto a ricevere le notifiche del Context Broker
* Serving Flask app 'analysis' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
server instead.
* Running on http://analysis:5050
INFO:werkzeug:Press CTRL+C to quit
INFO:root:Valori medi di riferimento estratti.
INFO:root:----- Data: 2023-01-26T13:17:43Z -----
INFO:root:NORMALE | Livello - Valore: 86 Media: 88 - Differenza: -2
INFO:root:NORMALE | Temperatura - Valore: 48 Media: 50 - Differenza: -2
INFO:root:-----
INFO:werkzeug:172.18.1.3 - - [26/Jan/2023 12:17:47] "POST /notification?subscriptionId=urn:ngsi-ld:Subscription:7007
277a-9d73-11ed-85d0-0242ac120103 HTTP/1.1" 200 -
INFO:root:JSON Analizzato inviato a MongoDB
INFO:root:----- Data: 2023-01-26T14:17:43Z -----
INFO:root:NORMALE | Livello - Valore: 86 Media: 87 - Differenza: -1
INFO:werkzeug:172.18.1.3 - - [26/Jan/2023 12:17:50] "POST /notification?subscriptionId=urn:ngsi-ld:Subscription:7007
277a-9d73-11ed-85d0-0242ac120103 HTTP/1.1" 200 -
INFO:root:NORMALE | Temperatura - Valore: 51 Media: 47 - Differenza: 4
INFO:root:-----
INFO:root:JSON Analizzato inviato a MongoDB

```

Figura 5.7: Log dei processi Analisi e Server Flask

5.4.5 Container MongoDB

Il database è ospitato all'interno di questo container ed espone la porta 27017. Esso è quindi esplorabile comodamente facendo uso del software ufficiale **MongoDB Compass**.

L'indirizzo a cui è presente il container è `mongodb://mongo:27017`.

5.4.6 Container Node.js Web APP

L'applicazione è contenuta all'interno di questo container, raggiungibile all'indirizzo `http://localhost:3000`.

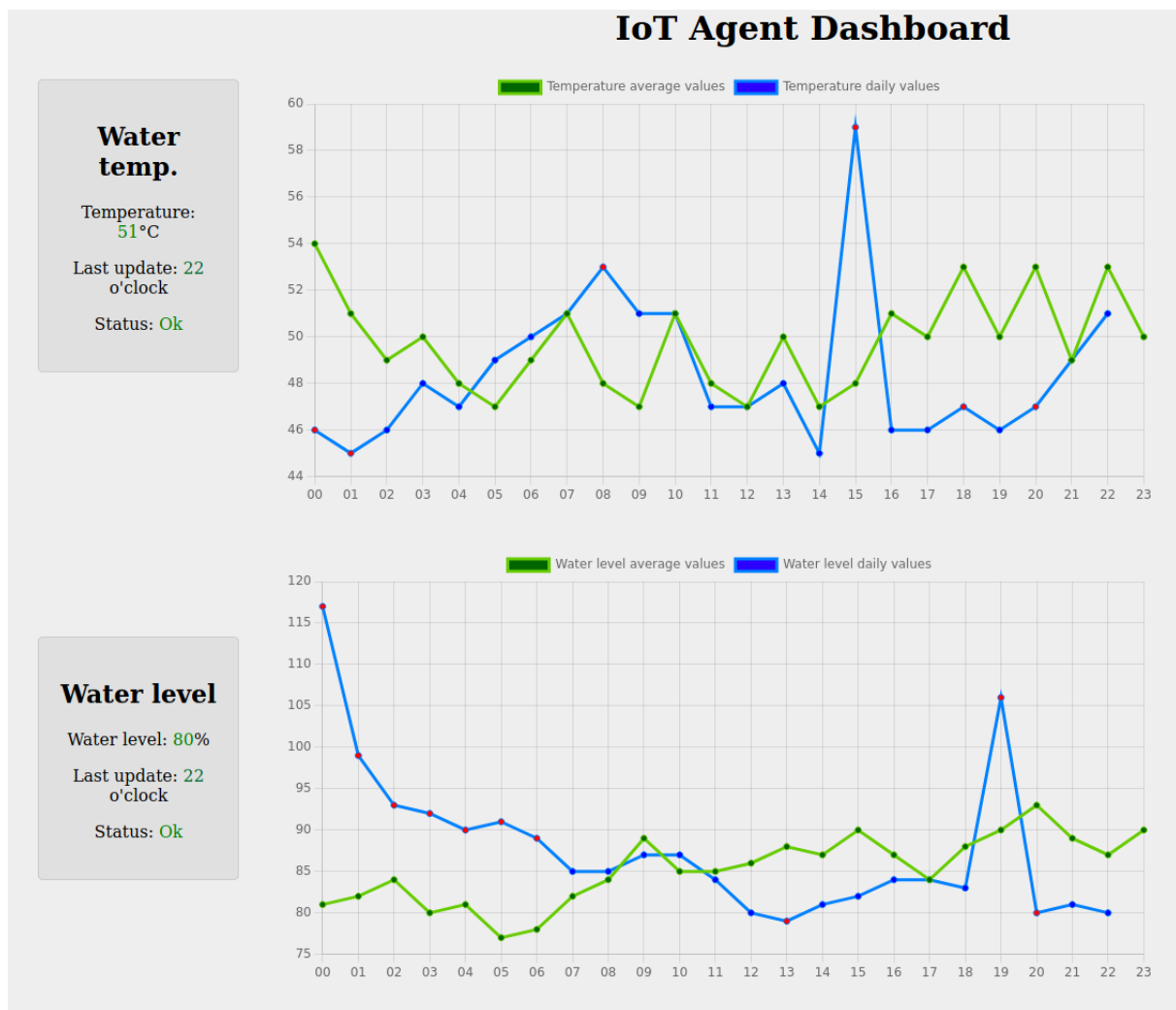


Figura 5.8: Rappresentazione delle misure di temperatura e livello dell'acqua, ora per ora, paragonati alle medie

5.4.7 Stop

Al fine di fermare tutti i container è possibile utilizzare il comando: `./services.sh stop`

```
[kynesys@parrot]~[~/Desktop/BIG_DATA_FINAL-SIUM]
$ ./services.sh stop
Stopping sensor container
done
Stopping Analysis container
done
Stopping Web App container
done
Stopping Fiware containers
done
Removing old volumes
done
Removing tutorial networks
done
```

Figura 5.9: Output del comando di stop

5.4.8 Purge

Per ripulire il sistema ed eliminare le immagini è possibile utilizzare il comando: `./services.sh purge`

5.4.9 Link Repository GitHub

https://github.com/francesco-denu/Fiware_IoT_sensor_analysis.git



Figura 5.10: QR Code per raggiungere la Repository