DIPARTIMENTO DI INGEGNERIA DELL'INNOVAZIONE

CORSO DI LAUREA IN: INGEGNERIA INFORMATICA

PROGETTO IN "GESTIONE DI BIG DATA"

TITOLO DEL PROGETTO:
"SMART ENERGY METER AS A SERVICE"

Studenti:

CARMINE ACCOGLI

MATTIA COTARDO

DAVIDE ROLLO

ANNO ACCADEMICO:

2022/2023

INDICE

PRESENTAZIONE E TRACCIA DEL PROGETTO

I progetto in questione è stato pensato, svolto ed eseguito grazie agli strumenti sia software che conoscitivi messi a disposizione da FIWARE, fondazione che fornisce strumenti implementativi Open Source per permettere uno sviluppo più semplice e rapido di soluzioni smart.

La traccia svolta è la seguente:

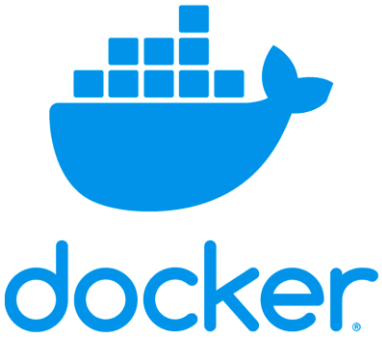
Smart Energy Meter as a Service:

"Design and implementation of a Smart Energy Meter as a Service using FIWARE's technology to collect and monitor home appliances daily power consumptions and inform the user on some best practices to optimize power usage through effective dashboards. Measurements and decisions must be made based on data streamed by a variety of IoT sensor devices, such as air quality sensors to keep track of fundamental variables in different room environments, multimeter sensors to measure the power absorbed by any home appliance and so forth.

Furthermore, the system shall also make provisions to trigger actuators (e.g. relays installed on electric plugs, servo motors to drive heater units valves, etc.) in order to actively optimize power consumptions."

SOFTWARE IMPIEGATI

Per la realizzazione di questo progetto è stato necessario scaricare ed utilizzare i seguenti software:



- **Docker**: Docker è una piattaforma per la creazione e l'esecuzione di container che facilitano lo sviluppo, la distribuzione e la gestione delle applicazioni. Permette di isolare le applicazioni dall'ambiente di esecuzione e di gestirle in modo semplice e scalabile. Un container Docker è un'unità di software che racchiude tutte le componenti necessarie per far funzionare un'applicazione, come il codice, le librerie, le variabili d'ambiente e il sistema operativo. I container vengono eseguiti in modo isolato l'uno dall'altro e dal sistema host, ma possono comunicare tra di loro e accedere a risorse condivise. Tutto questo rende più semplice e affidabile la distribuzione e la gestione delle applicazioni [1].

- **Postman**: Postman è un'applicazione per la creazione, l'invio e la ricezione di richieste HTTP (Hypertext Transfer Protocol), utilizzata principalmente per testare le API (Application Programming Interface), visualizzarne e analizzarne le risposte [2].

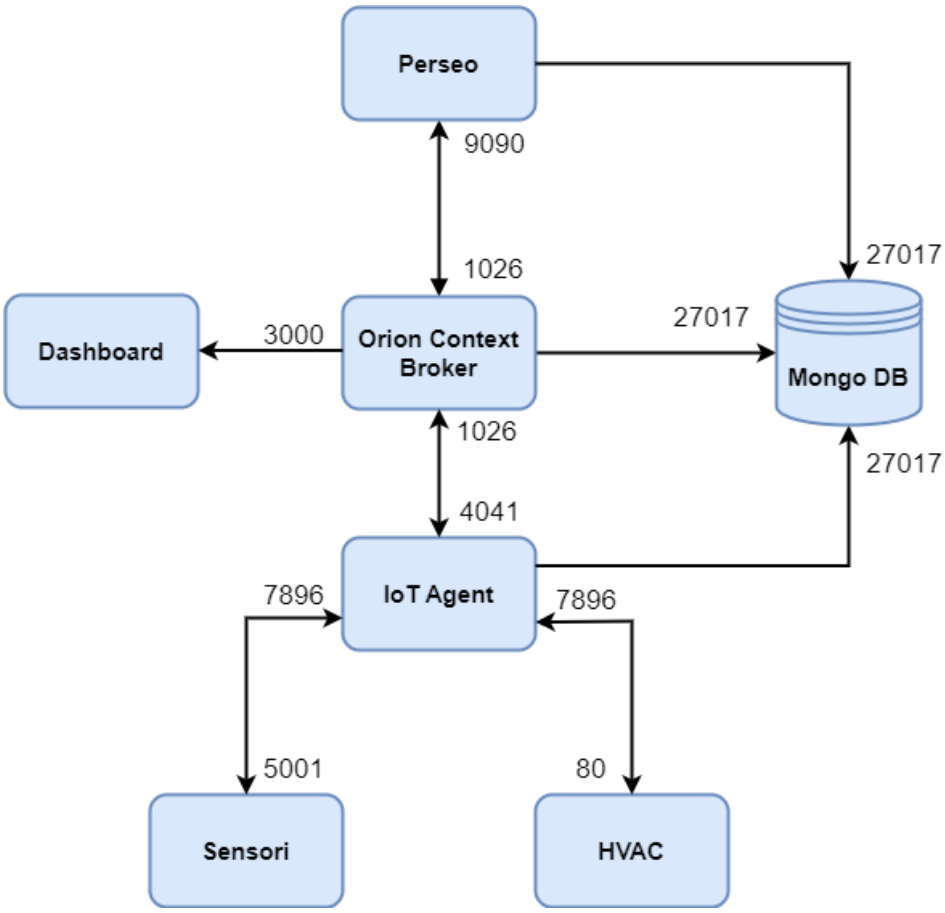
- **Flask:** è un microframework per la creazione di applicazioni web in Python. Flask fornisce un'interfaccia semplice per la gestione di richieste HTTP, la creazione di route e l'utilizzo di template per la visualizzazione delle pagine web [3].

- **Node.js:** è una piattaforma per la creazione di applicazioni web sviluppata in JavaScript. Include una vasta libreria di moduli per la creazione di server HTTP che la rendono una soluzione versatile per molti scopi, tra cui siti web dinamici, API REST e applicazioni real-time [4].

- **MongoDB Compass**: è un software che presenta un'interfaccia grafica user-friendly per la gestione e l'analisi dei dati presenti all'interno di un cluster MongoDB. Fornisce una visualizzazione intuitiva dei documenti, una serie di strumenti per creare e modificare schemi e indici, e una potente funzionalità di query. In sintesi, MongoDB Compass rende più semplice e veloce l'utilizzo di MongoDB per utenti con diversi livelli di esperienza [5].

ARCHITETTURA DEL PROGETTO E PRESENTAZIONE DELLE COMPONENTI

L'architettura adottata per la realizzazione del progetto è la seguente:



Le componenti utilizzate sono:

Orion Context Broker: Orion Context Broker consente di gestire l'intero ciclo di vita delle informazioni di contesto, inclusi aggiornamenti, query, registrazioni e Subscription. Esso riceve le richieste usando NGSI-v2 ("Next Generation Service Interfaces version 2"), ossia un modello informatico per la gestione delle informazioni di contesto, utilizzato in particolar modo per la comunicazione fra l'IoT Agent e il Context Broker che mira a raccogliere i dati ottenuti da dispositivi IoT [6]. Esso è in ascolto sulla porta 1026.

MongoDB: Database NoSQL documentale (che utilizza documenti con formato JSON) utilizzato da Orion Context Broker per conservare informazioni sui dati di contesto come entità e Subscription, da Perseo e dall'IoT Agent. Esso è in ascolto sulla porta 27017.

IoT Agent: Un IoT Agent è un componente che consente a un gruppo di device (sensori ed attuatori) di inviare i propri dati e di essere gestiti da un Context Broker utilizzando i propri protocolli nativi. Ogni IoT Agent è definito per un singolo formato di payload, sebbene possa essere in grado di utilizzare più protocolli di trasporto per quel payload [6]. Nel nostro caso specifico il formato utilizzato sarà di tipo JSON.
L'IoT Agent si occupa direttamente delle due tipologie di traffico principali che lo attraversano:

- **NorthBound Traffic**: richieste che vengono generate dai dispositivi IoT e che l'IoT Agent inoltra verso Orion. Queste informazioni sono anche dette "misure". L'IoT Agent riceve queste misure dai device in formato JSON e le trasforma in richieste di tipo NGSI-v2 da mandare direttamente ad Orion. Nel nostro caso le grandezze misurate saranno la temperatura e la qualità dell'aria. Per il NorthBound Traffic, l'IoT Agent è sempre in ascolto sulla porta 4041.
- **SouthBound Traffic**: comandi che l'IoT Agent riceve da Orion e che sono destinati agli attuatori. Essi vengono scatenati in relazione alle misure che sono state registrate e porteranno a un loro conseguente cambiamento. Nel nostro caso i comandi che verranno lanciati saranno relativi alle misurazioni effettuate di temperatura e di qualità dell'aria. Per il SouthBound Traffic, l'IoT Agent è sempre in ascolto sulla porta 7896.

Dispositivi IoT: sono dei device fisici in grado di connettersi a una network per scambiare dei dati. Nel nostro caso abbiamo tre tipologie di sensori (ossia device in grado di registrare grandezze del mondo fisico che li circonda) e una tipologia di attuatore (ossia un device in grado di eseguire dei comandi e alterare lo stato del mondo fisico che lo circonda con tali azioni, in risposta alle grandezze registrate dai sensori).

- **Sensori IoT**:
    - Sensore di temperatura: si tratta di un sensore che è in grado di registrare ciclicamente la temperatura corrente della stanza. Il suo comportamento verrà simulato da un'applicazione scritta in Python, la quale genera casualmente dei valori di temperatura plausibili e positivamente correlati fra di loro in base alla stagione che viene selezionata casualmente all'avvio del sistema (per semplicità si è limitata la selezione a due possibili stagioni: "Inverno" o "Estate"). All'avvio del sistema di riscaldamento/raffreddamento, (vedi Attuatore IoT più avanti) una funzione interverrà per permettere una regolarizzazione della temperatura a un valore considerato accettabile.
    - Sensore di qualità dell'aria: è un sensore in grado di registrare ciclicamente la quantità di CO2 presente nell'aria, la cui unità di misura è parti per milione (ppm). Il suo comportamento verrà simulato da un'applicazione scritta in Python che genera, attraverso l'impiego di una distribuzione di probabilità uniforme, un aumento della concentrazione di CO2 registrata. All'avvio del sistema di ventilazione (vedi Attuatore IoT più avanti) una funzione interverrà per permettere un progressivo decremento della CO2.
    - Sensore Multimetro: è un sensore in grado di registrare ciclicamente il consumo energetico cumulativo di tutti i dispositivi attivi all'interno della casa. Il suo comportamento verrà simulato da un'applicazione scritta in Python che controlla i device attivi in quell'istante e somma i loro valori di consumo di energia elettrica (si noti che si è supposto un consumo, seppur irrisorio, dei sensori che saranno sempre attivi L'attuatore, invece, si attiverà solo in caso di necessità).

  Ulteriori dettagli implementativi sono descritti nella sezione a pag. 20.

- **Attuatore IoT:**
    - HVAC: Sigla che sta per "Heating, Ventilation and Air Conditioning". Si tratta di un attuatore le cui funzionalità verranno comandate in base alle misure che vengono registrate dai sensori. Ciò per permettere di

ottimizzare il consumo di energia elettrica e automatizzare il funzionamento del dispositivo. Il suo comportamento sarà simulato da un'applicazione scritta in Python, la quale presenta una serie di metodi per poter interagire con le sue funzionalità: accendere il sistema di raffreddamento o il sistema di riscaldamento se le temperature registrate raggiungono certe soglie prefissate (sono stati impostate rispettivamente i limiti di 35° e di 10°), spegnere il riscaldamento/raffreddamento nel momento in cui le temperature ritornano in un range ottimale (tra i 15° e i 25°), accendere il sistema di ventilazione nel caso di una concentrazione troppo elevata di CO2 nell'aria (soglia impostata a 1000 ppm), e spegnere il sistema di ventilazione nel momento in cui la suddetta concentrazione rientri in un intervallo ottimale (soglia impostata a 600 ppm). L'Attuatore è raggiungibile sulla porta 80.

Perseo: Perseo è un software CEP (Complex Event Processing) basato su Esper, progettato per essere completamente conforme a NGSI-v2. Utilizza infatti NGSI-v2 come protocollo di comunicazione per gli eventi e, pertanto, è in grado di lavorare senza soluzione di continuità e congiuntamente con Orion Context Broker. Perseo è in ascolto di Eventi provenienti dalle informazioni di contesto per identificare dei pattern descritti da dalle Regole: nel momento in cui un evento soddisfa un determinato pattern, la Regola a esso associata si innesca, scatenando uno o puù azioni che consentono di creare o aggiornare entità, interagire con sistemi esterni come server Web, e-mail o SMS [7]. Nel nostro caso l'azione andrà ad attivare uno dei possibili comandi dell'attuatore tramite interazione con le API presenti nell'applicazione HVAC.

La sua architettura è mostrata in figura 1 e si compone di due parti principali:

- **Perseo FrontEnd**: responsabile dell'elaborazione degli eventi e delle regole in entrata, dell'archiviazione delle regole e dell'esecuzione delle azioni. Raggiungibile sulla porta 9090.

- **Perseo Core**: controlla se gli eventi in arrivo soddisfano una Regola scritta in linguaggio EPL e invoca Perseo FE nel caso un'azione debba essere eseguita. Esso non ha porte esposte ma è raggiungibile unicamente da Perseo FE.

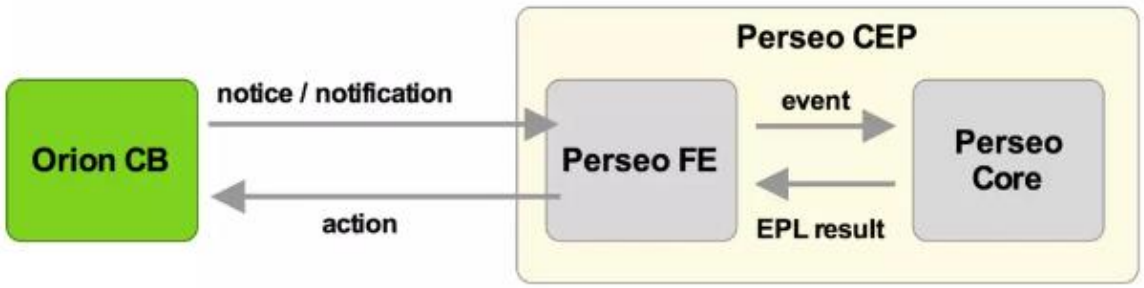


Figura 1: *Architettura di Perseo*

Come è descritto in figura 1, Orion invia una notifica a Perseo FE ogni volta che i valori di uno o più attributi di una entità a cui Perseo ha effettuato la Subscription vengono aggiornati. Perseo traduce tale Notifica in un Evento che viene inoltrato al suo Core. Il Core controlla se la condizione della Regola precedentemente definita è rispettata (per esempio, un sensore che registra una temperatura al di sotto di una determinata soglia) [8]. La Regola è stata definita tramite il linguaggio EPL (Event Processor Language) ossia un linguaggio SQL-like usato per la gestione di Eventi. Se la Regola vede la propria condizione soddisfatta, Perseo FE eseguirà l'Azione che era stata precedentemente definita associata a quella Regola (ad esempio: accensione della modalità Heating dell'HVAC).

FLUSSO DEI DATI

Nell'architettura precedentemente descritta vi è la presenza di un certo flusso di dati e contenuti che vengono scambiati fra i molteplici componenti del sistema. Riportiamo qui di seguito in forma tabellare le varie informazioni scambiate fra questi elementi costituitivi dell'architettura, dal momento in cui i container vengono avviati e le applicazioni entrano in esecuzione.

Numero dello step	Componente sorgente dell'informazione	Componente destinazione dell'informazione	Flusso dati scambiato
1 - POPOLAMENTO DEL CONTESTO			
1.a	Amministratore di sistema tramite file "services.sh"	Orion Context Broker	Vengono aggiunte tramite "curl" 2 entità Room
1.b	Amministratore di sistema tramite file "services.sh"	IoT Agent	Viene effettuato all'IoT Agent il Provisioning del Service Group, ossia viene definita una chiave di autenticazione per i device che verranno aggiunti e viene collegato l'IoT Agent ad Orion
1.c	Amministratore di sistema tramite file "services.sh"	IoT Agent	Vengono aggiunte tramite "curl" 2 entità HVAC (una per Room), 2 entità Sensori di Temperatura (una per Room), 2 entità Sensore di Qualità dell'aria (una per Room), 1 entità Sensore Multimetro.
1.d	Amministratore di sistema tramite file "services.sh"	Orion Context Broker	Viene aggiunta una Subscription agli attributi "temperatura" e "CO2_level" delle Room. Questa Subscription manderà delle notifiche a Perseo ogni

			volta che uno di questi attributi verrà aggiornato
1.d	Amministratore di sistema tramite file "services.sh"	Perseo FE	Vengono aggiunte 5 Regole (3 per la gestione del sistema di riscaldamento/raffreddamento, 2 per la gestione del sistema di ventilazione) [9]
2.a	IoT Agent	Orion Context Broker	I dispositivi IoT vengono aggiunti al contesto
2 - NORTHBOUND TRAFFIC (INVIO DELLE MISURE)			
1.a	Sensori IoT	IoT Agent	Le misure (temperatura, qualità dell'aria e consumo energetico cumulativo) vengono mandate ciclicamente all'IoT Agent
1.b	Sensori IoT	Orion Context Broker	Le misure (temperatura, qualità dell'aria) vengono mandate ciclicamente alle loro entità Room di riferimento
2.a	IoT Agent	Orion Context Broker	L'IoT Agent aggiorna il contesto di Orion con le misure appena ricevute
3 – INVIO DELLE NOTIFICHE			
1.a	Orion Context Broker	Perseo FE	Ogni volta che viene aggiornato l'attributo temperatura oppure l'attributo CO2_level dell'entità Room, grazie alla Subscription precedentemente effettuata, Orion manda una notifica a Perseo per informarlo del cambiamento appena avvenuto
2.a	Perseo FE	Perseo Core	Perseo FE riceve la notifica e crea un Evento che viene mandato a Perseo Core
4 – ESECUZIONE DELLA REGOLA			

1.a	Perseo Core	Perseo FE	Perseo Core per ogni Evento ricevuto esegue la query EPL specificata nel campo "text" della Regola precedentemente definita e invia il risultato a Perseo FE
2.a	Perseo FE	Orion Context Broker	Se la condizione della query EPL è stata soddisfatta (ad esempio la temperatura supera una certa soglia), viene eseguito l'azione contenuta nella regola specificata (PATCH all'entità hvac per la quale è stata ricevuta la notifica che ha innescato la Regola)
5 – SOUTHBOUND TRAFFIC (INVIO DEI COMANDI)			
1.a	Orion Context Broker	IoT Agent	Il comando contenuto all'interno del campo "action" della Regola viene mandato all'IoT Agent
2.a	IoT Agent	Attuatore IoT	Il comando viene inviato al relativo device HVAC che lo esegue (accensione di "heating" o "cooling", spegnimento di "heating" o "cooling", accensione di "ventilation", spegnimento di "ventilation")
3.a	Attuatore IoT	IoT Agent	Viene restituito all'IoT Agent il risultato del comando ("OK" per heating, cooling e ventilation)
4.a	IoT Agent	Orion Context Broker	Viene aggiornato il contesto con il risultato del comando

Lo stesso flusso di dati prima descritto è sintetizzato dal diagramma di flusso rappresentato in figura 2.

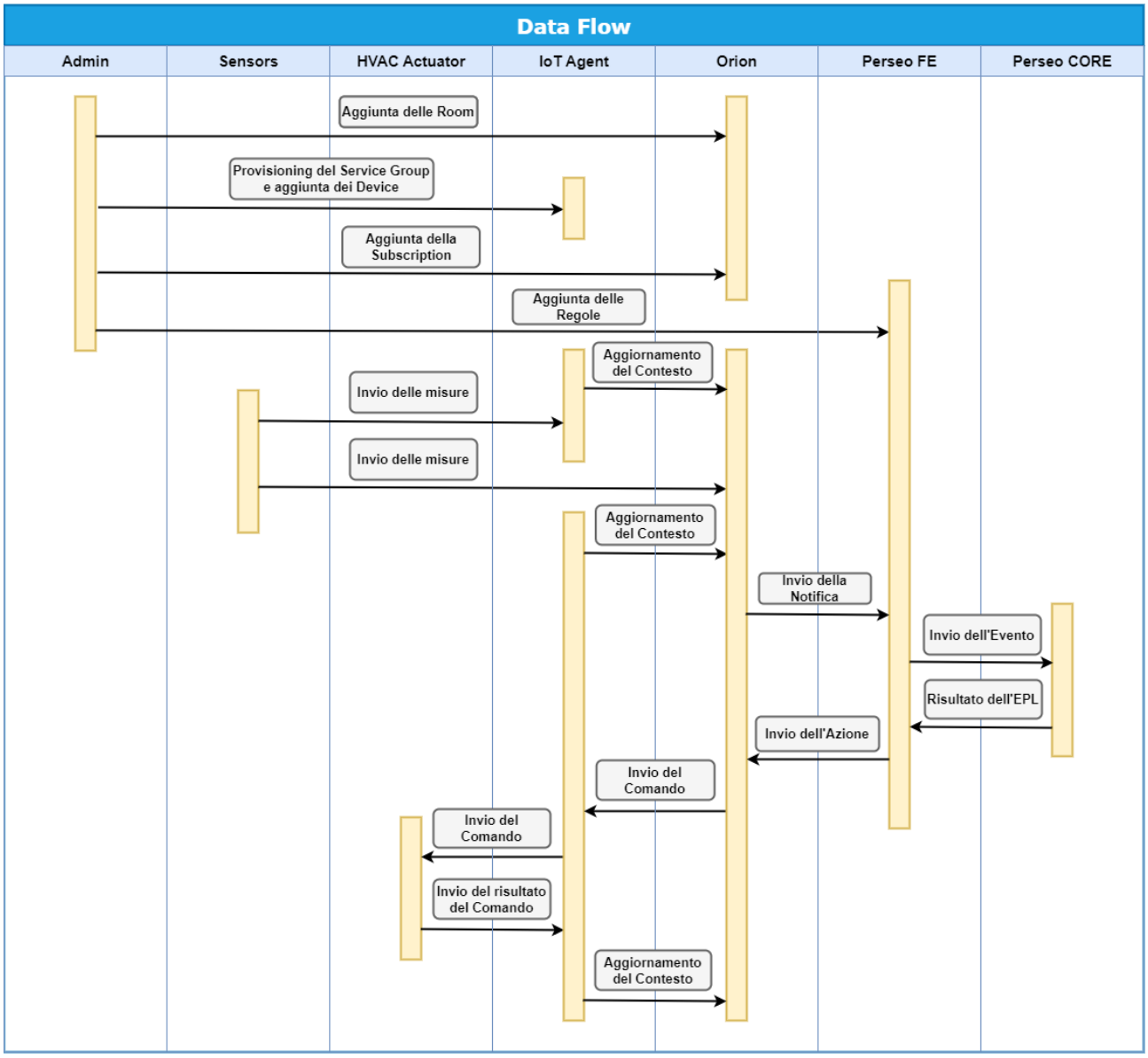


Figura 2: *Diagramma delle interazioni tra le varie parti del sistema*

MODELLO DATI DI MONGODB

Alcune parti del sistema (come Orion, Perseo, IoTAgent) necessitano di una funzionalità di persistenza dei dati. Per tale scopo è presente il servizio MongoDB sull'apposito container.

MongoDB è un sistema di gestione di database NoSQL orientato a documenti, che offre un'ampia flessibilità nella modellizzazione dei dati. Consente di archiviare grandi quantità di dati non strutturati e di effettuare query veloci e scalabili. È open source e supporta diversi linguaggi di programmazione.

Le seguenti figure sono state ottenute tramite MongoDB Compass, software che permette di connettersi a una distribuzione MongoDB in questo caso ospitata localmente e visualizzare i dati contenuti.

In figura 3 è riportata una schermata che mostra i databases contenuti con le relative collezioni.

Figura 3: *Lista di istanze di databases presenti nel sistema*

- **cep → executions**: collezione che raccoglie in ordine cronologico le azioni che sono state eseguite da Perseo. Ogni documento corrisponde a un'azione eseguita, nel nostro caso a un comando per il device HVAC. Nella figura *4* si può vedere come all'orario "13:51:06" sia stato eseguito il comando "ventilationON" e all'orario "13:51:57" sia stato eseguito il comando "ventilationOFF".

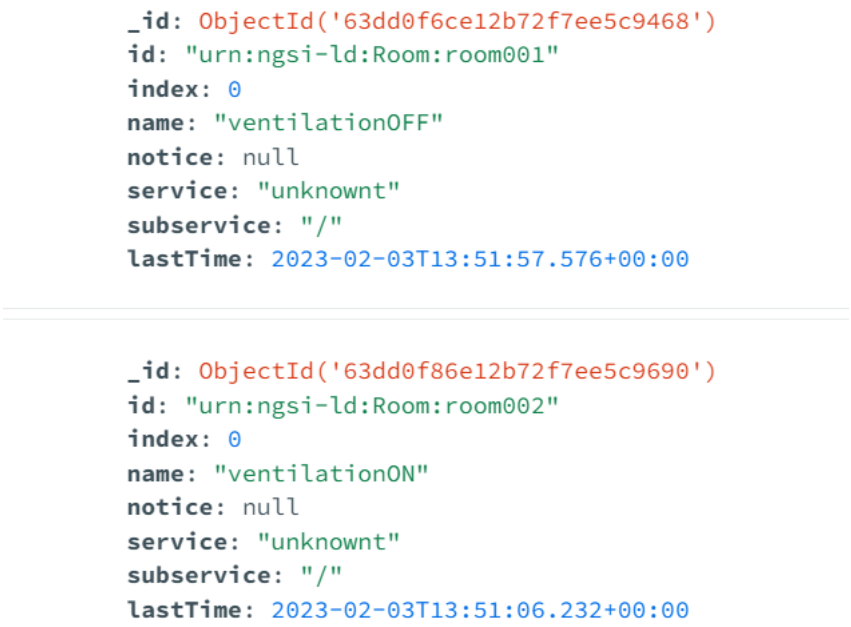


Figura 4: *Esempio di documenti presenti nella collezione "executions" dell'istanza "cep"*

- **cep → rules:** Collezione che raccoglie l'elenco di tutte le Regole di Perseo presenti nel sistema. Sono presenti pertanto 5 documenti. A scopo esemplificativo la seguente figura *5* mostra i due documenti relativi alle due Regole che attivano e disattivano il sistema di ventilazione.

Figura 5: *Esempio di documenti presenti nella collezione "rules" dell'istanza "cep"*

- **iotagentjson → devices:** Collezione che raccoglie tutti i device che abbiamo creato per il nostro sistema. Nel nostro caso essa contiene i documenti che rappresentano i due sensori di temperatura, due sensori di qualità dell'aria, un sensore multimetro e due attuatori hvac. A scopo esemplificativo, la figura 6 mostra un solo sensore per ogni tipologia, e un solo attuatore hvac (ossia tutti i dispositivi IoT relativi alla room001).

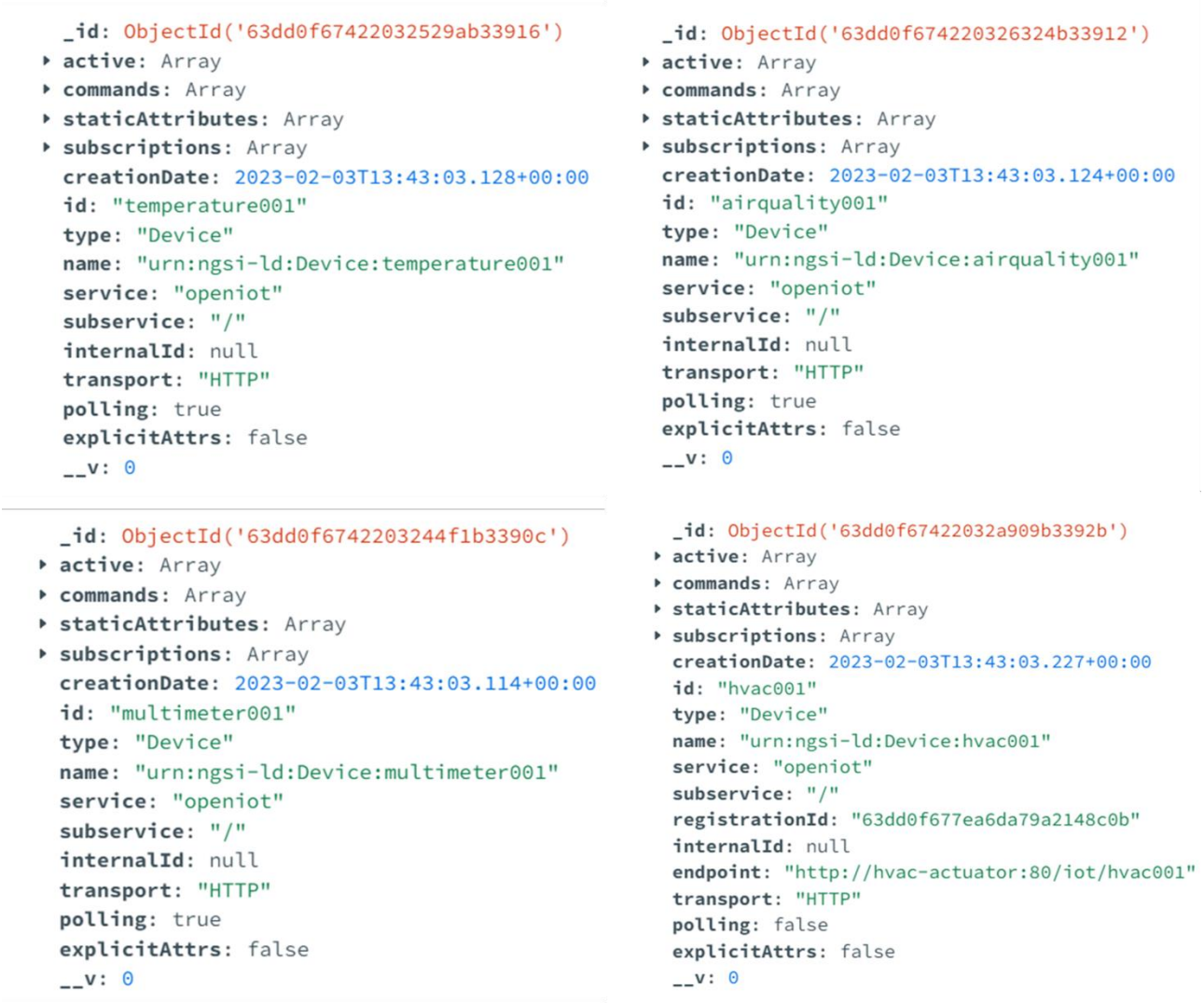


Figura 6: *Esempio di documenti presenti nella collezione "devices" dell'istanza "iotagentjson"*

- **iotagentjson → groups:** Collezione che contiene il Service Group di cui abbiamo effettuato il provisioning *[fig. 7]*.

```
_id: ObjectId('63dd0f664220326894b338e7')
resource: "/iot/json"
apikey: "4jggokgpepnvsb2uv4s40d59ov"
type: "Thing"
service: "openiot"
subservice: "/"
__v: 0
```


Figura 7: *Documento presente nella collezione "groups" dell'istanza "iotagentjson"*

- **orion → csubs:** Collezione che contiene la Subscription effettuata sulle entità Room per notificare Perseo-Fe della variazione degli attributi "temperature" e "CO2_level". Il documento indica anche il numero di notifiche inviate (attributo count) *[fig. 8]*.


```
_id: ObjectId('63dd0f677ea6da79a2148c0d')
expiration: 9223372036854775807
reference: "http://perseo-fe:9090/notices"
custom: false
timeout: 0
throttling: 0
maxFailsLimit: -1
servicePath: "/#"
status: "active"
statusLastChange: 1675431783.2371962
▸ entities: Array
▸ attrs: Array
▸ metadata: Array
blacklist: false
onlyChanged: false
covered: false
description: "Subscription to feed the CEP"
▸ conditions: Array
▸ expression: Object
▸ altTypes: Array
format: "normalized"
count: 690
lastNotification: 1675432671
failsCounter: 0
lastSuccess: 1675432671
lastSuccessCode: 200
```


Figura 8: *Documento presente nella collezione "csubs" dell'istanza "orion"*

- **orion → entities:** Collezione che contiene le informazioni sulle entità. Nel nostro caso essa contiene due entità Room. A scopo esemplificativo la seguente immagine mostra una sola room. *[fig. 9]*

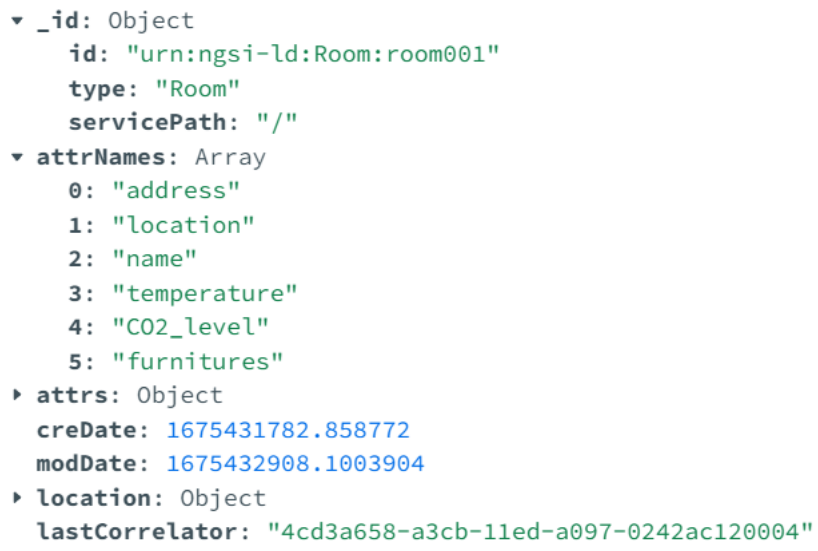


Figura 9: *Documento relativo all'entità room001 presente nella collezione "entities" dell'istanza "orion"*

IMPLEMENTAZIONE DEL CODICE

Il progetto è stato sviluppato attraverso i seguenti file presenti nella cartella "project_root":

```
|--project_root
   |--.env
   |--docker-compose.yml
   |--services.sh
   |--src
      |--hvac
         |--Dockerfile
         |--requiremets.txt
         |--hvac.py
      |--sensors
         |--Dockerfile
         |--requiremets.txt
         |--sensors.py
```

- ".env": contiene le variabili di ambiente necessarie per il completamento del file docker-compose. *[fig. 10]*

```
1   # Project name #
2   COMPOSE_PROJECT_NAME=fiware
3
4   # Orion variables #
5   ORION_PORT=1026
6   ORION_VERSION=3.7.0
7
8   # MongoDB variables #
9   MONGO_DB_PORT=27017
10  MONGO_DB_VERSION=4.4
```

Figura 10: *Schermata del file .env*

- "docker-compose.yml": File attraverso cui possiamo andare a definire e creare con un semplice comando una serie di container, i quali contengono le applicazioni, i servizi (un esempio è mostrato in figura 11), le immagini e i volumi necessari per il funzionamento del progetto. Questo rende più

semplice la distribuzione e la gestione delle applicazioni formate da molteplici componenti.

```yaml
version: "3.8" # Version for the docker compose #

# Definition of the Services #
services:

  # Orion is the context broker #
  orion:
    image: fiware/orion:3.7.0 # Loads this image from Docker Hub #
    hostname: orion # Create a hostname to be easier to find the container in the network #
    container_name: fiware-orion  # Create a container name to be easier to get information for it #
    depends_on:
      - mongo-db  # Database for persinting eantity information from orion #
    expose:
      - "1026"  # Expose this port to the network #
    ports:
      - "1026:1026" # match port 1026 on the machine to the one in the container #
    command: -corsOrigin __ALL -dbhost mongo-db -logLevel DEBUG -noCache
    healthcheck:  # check if orion is healthy #
      test: curl --fail -s http://orion:1026/version || exit 1
      interval: 10s
    networks:
      - default
```

Figura 11: *Schermata del file docker-compose.yml. Notare la presenza dell'opzione corsOrigin impostata a ALL per consentire a qualsiasi origine di effettuare richieste CORS (Cross Origin Resource Sharing) a Orion. Ciò è necessario per l'interazione con la dashboard creata.*

Si noti che il docker-compose.yml contiene altri servizi al suo interno oltre "orion", come "mongo-db", "iot-agent", "hvac-actuator", "sensors", "perseo-core", "perseo-fe" e "dashboard-react".

- "services.sh": File attraverso cui lanciamo i comandi per inizializzare il sistema, avviarlo e, eventualmente, arrestarlo. Esso inoltre fa il Provisioning del Service Group *[fig. 12]*, aggiunge al contesto i Device (temperature001, temperature002, airquality001, airquality002, multimeter001, hvac001 e hvac002), *[fig. 13]* le Entità (room001 e room002) *[fig. 14]*, la Subscription *[fig. 15]* e le Regole *[fig. 16]*.

```
#
# Provisioning del Service Group per la definizione di una chiave di autenticazione
#
curl -o /dev/null --silent -iX POST \
  'http://localhost:4041/iot/services' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: openiot' \
  -H 'fiware-servicepath: /' \
  -d '{
"services": [
  {
    "apikey":      "4jggokgpepnvsb2uv4s40d59ov",
    "cbroker":     "http://orion:1026",
    "entity_type": "Thing",
    "resource":    "/iot/json"
  }
]
}'
```

Figura 12: *Inserimento del Service Group tramite curl all'IoT Agent. Questo viene informato che verrà usato l'endpoint presente in "resource", che i devices si autenticheranno includendo il token in "apikey". Si fornisce inoltre la locazione del context broker in "cbroker"*

```
# Creazione delle entità sensore di temperatura 'temperature001' e 'tmperature002',
# sensore di air quality 'airquality001'
# e 'airquality002', e sensore Multimeter 'multimeter001'
#
curl -o /dev/null --silent -iX POST \
  'http://localhost:4041/iot/devices' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: openiot' \
  -H 'fiware-servicepath: /' \
  -d '{
"devices": [
  {
    "device_id":   "temperature001",
    "entity_name": "urn:ngsi-ld:Device:temperature001",
    "entity_type": "Device",
    "category": {
      "type": "Property",
      "value": ["sensor"]
    },
    "timezone":    "Europe/Berlin",
    "attributes": [
      {
        "object_id": "t",
        "name": "temperature",
        "type": "Number",
        "metadata": {
          "unitCode": {
            "type": "Text",
            "value": "CEL"
          }
        }
      }
    ],
    "static_attributes": [
      { "name":"refRoom", "type": "Relationship", "value": "urn:ngsi-ld:Room:room001"}
    ]
  },
```

Figura 13: *Inserimento del device "temperature001" tramite curl all'IoT Agent. In questo modo associamo il device con l'URN delle specifiche NGSI-LD, definiamo un mapping tra la misura t e l'attributo "temperature" in "attributes" e la relazione con la room001 in "static_attributes"*

```
# Creazione delle entità 'room001' e 'room002'
#
curl -o /dev/null --silent -iX POST \
  --url 'http://localhost:1026/v2/op/update' \
  --header 'Content-Type: application/json' \
  --data ' {
  "actionType":"append",
  "entities":[
        {
        "id":"urn:ngsi-ld:Room:room001","type":"Room",
        "address":{
          "type":"PostalAddress","value":{
           "streetAddress":"Viale Roma 12","addressRegion":"Puglia",
           "addressLocality":"Lecce","postalCode":"73100"
          }
        },
        "location":{
          "type":"geo:json","value":{
           "type":"Point","coordinates":[40.36060,18.20347]
          }
        },
        "name":{
          "type":"Text","value":"LivingRoom"
        },
        "temperature":{
          "type":"Number", "value": 0,
          "metadata": {"unitCode": {"type": "Text","value": "CEL"
            }
          }
        },
        "CO2_level":{
          "type":"Number", "value": 0,
          "metadata": {"unitCode": {"type": "Text","value": "ppm"
            }
          }
          },
        "refActuator":{
          "type":"Relationship","value":"urn:ngsi-ld:Device:hvac001"
          }
        },
        {
        "id":"urn:ngsi-ld:Room:room002","type":"Room",
        "address":{
          "type":"PostalAddress","value":{
           "streetAddress":"Viale Roma 12","addressRegion":"Puglia",
           "addressLocality":"Lecce","postalCode":"73100"
          }
        },
        "location":{
          "type":"geo:json","value":{
           "type":"Point",
           "coordinates":[40.36060,18.20347]
          }
        },
        "name":{
          "type":"Text","value":"Kitchen"
        },
        "temperature":{
          "type":"Number", "value": 0,
          "metadata": {"unitCode": { "type": "Text", "value": "CEL"
            }
          }
        }
        },
        "CO2_level":{
          "type":"Number", "value": "0",
          "metadata": {"unitCode": {"type": "Text", "value": "ppm"
            }
          }
          },
        "refActuator":{
          "type":"Relationship","value":"urn:ngsi-ld:Device:hvac002"
          }
        }
    ]
  }'
```

Figura 14: *Inserimento delle entità "room001", "room002" tramite curl a Orion*

```
# Creazione della Subscription Perseo-Orion sugli attributi 'temperature' e 'CO2_level' delle entità 'Room'
#
curl -o /dev/null --silent -iX POST \
'http://localhost:1026/v2/subscriptions' \
-H 'Content-Type: application/json' \
-d '{
  "description": "Subscription to feed the CEP",
  "subject": {
    "entities": [
      {
        "idPattern": ".*",
        "type": "Room"
      }
    ],
    "condition": {
      "attrs": ["temperature", "CO2_level"]
    }
  },
  "notification": {
    "http": {
      "url": "http://perseo-fe:9090/notices"
    },
    "attrs": ["temperature", "refActuator", "CO2_level"]
  }
}'
```

Figura 15: *Inserimento della subscription tramite curl a Orion, usata per rilevare cambiamenti agli attributi "temperature" e "CO2_level" sulle entità di tipo Room (attributo "condition") e inviare di conseguenza una notifica a Perseo-fe contenente solo gli attributi elencati in "attrs".*

```
# Creazione della regola 'coolingON': accendi in modalità 'cooling' quando la temperatura è > 35
#
curl -o /dev/null --silent -iX POST \
'http://localhost:9090/rules' \
-H 'Content-Type: application/json' \
-d '{
  "name":"coolingON",
  "text":"select *, refActuator? as ActuatorID, temperature? as Temperature
          from iotEvent where (cast(cast(temperature?,String),double)>35 and type=\"Room\")",
  "action":{
     "type":"post",
     "parameters":{
       "url": "http://orion:1026/v2/entities/${ActuatorID}/attrs",
       "method": "PATCH",
       "headers": {
           "Content-type": "application/json",
           "fiware-service": "openiot",
           "fiware-servicepath": "/"
       },
       "json": {
           "cooling": {
               "type" : "command",
               "value" : ""
               }
       }
     }
  }
}'
```

Figura 16: *Inserimento della regola su Perseo per eseguire l'accensione del sistema di raffreddamento relativo alla stanza nella quale è stata superata la soglia di temperatura di 35°.*

- "src/hvac/Dockerfile": Un Dockerfile specifica le operazioni che devono essere eseguite per configurare un'immagine Docker, come l'installazione delle dipendenze, la copia dei file di applicazione e la definizione delle variabili d'ambiente. Una volta che un Dockerfile è stato scritto, può essere utilizzato per creare un'immagine Docker che può essere eseguita come container in qualsiasi host che esegue Docker. In questo caso il Dockerfile definisce l'immagine per l'attuatore hvac. *[fig. 17]*

```
1   FROM python:3.11-alpine3.16
2
3   COPY requirements.txt .
4
5   RUN python3 -m pip install -r requirements.txt --no-cache-dir
6
7   COPY . .
8
9   EXPOSE 80
10
11  ENV FLASK_APP=hvac.py
12  CMD ["flask", "run", "--host=0.0.0.0", "--port=80"]
```

Figura 17: *Schermata del file Dockerfile relativa al web server Flask hvac*

- "src/hvac/requirements.txt": File che contiene i moduli necessari da importare nell'applicazione della quale generiamo l'immagine attraverso il relativo Dockerfile. *[fig. 18]*

```
1   flask
2   requests
```

Figura 18: *Schermata del file requirements.txt*

- "src/hvac/hvac.py": Web server Flask scritto in Python che simula il comportamento di un HVAC. Esso è composto dalle seguenti parti:
    - Definizione della classe HVAC con rispettive variabili e metodi per la gestione delle funzionalità *[fig. 19]*

```python
# Definizione della classe HVAC per simulare i sistemi di "Heating, Ventilation, Air Conditioning"
class HVAC:

    def __init__(self):
        self.cooling = 0
        self.heating = 0
        self.ventilation = 0
        self.consume = 0.5  # kW/h

    # Metodo per l'accensione della modalità riscaldamento #
    def heating_on(self):
        self.heating = 1
        self.cooling = 0

    # Metodo per l'accensione della modalità aria condizionata #
    def cooling_on(self):
        self.heating = 0
        self.cooling = 1

    # Metodo per lo spegnimento #
    def switch_off(self):
        self.heating = 0
        self.cooling = 0

    # Metodo per l'accensione della modalità ventilazione #
    def ventilationON(self):
        self.ventilation = 1

    # Metodo per lo spegnimento della modalità ventilazione #
    def ventilationOFF(self):
        self.ventilation = 0
```

Figura 19: *Classe usata per la simulazione di un sistema HVAC*

- Creazione delle due istanze della classe HVAC *[fig.20]*:

```python
# Creazione delle istanze per simulare 2 sistemi HVAC #
hvac001 = HVAC()
hvac002 = HVAC()
```

Figura 20: *Creazione di due device di tipo HVAC*

- API esposta dal sistema che consente di ottenere lo stato corrente dell'hvac001 o dell'hvac002 *[fig. 21]*.

```python
# GET per ottenere le informazioni sul sistema HVAC i-esimo #
@app.route('/iot/<hvac_id>', methods=['GET'])
def getStatus(hvac_id):

    response = {"cooling": getattr(eval(hvac_id), "cooling"),
     "heating": getattr(eval(hvac_id),"heating"), "ventilation": getattr(eval(hvac_id), "ventilation")}
    return Response(response=json.dumps(response), status=200, mimetype='application/json')
```

Figura 21: *API raggiungibile tramite localhost:80/iot/hvac_id*

- API esposta dal sistema che consente di ottenere il valore del consumo dei due device hvac. Per semplicità si è supposto che gli hvac avessero uguale consumo *[fig. 22]*.

```python
# GET per ottenere il valore di consumo di un sistema HVAC (tutti i sistemi hanno uguale consumo) #
@app.route('/iot/hvac001/consume', methods=['GET'])
def getConsume():
    response = {"consume": hvac001.consume}
    return Response(response=json.dumps(response), status=200, mimetype='application/json')
```

Figura 22: *API raggiungibile tramite localhost:80/iot/hvac001/consume*

- API esposta dal sistema che consente di modificare la modalità di funzionamento dei due hvac. Esso modifica gli attributi "status" e "info" dei relativi comandi dell'attuatore in relazione al comando che è stato invocato *[fig. 23]*.

```python
# POST per modificare la modalità di funzionamento del sistema HVAC i-esimo #
@app.route('/iot/<hvac_id>', methods=['POST'])
def executingCommand(hvac_id):

    # JSON ottenuto dalla richiesta #
    data = request.json

    # controllo errori e compatibilità con la tipologia di contenuto #
    if data is None or data == {}:
        return Response(response=json.dumps({"Error": "Incorrect type"}), status=400, mimetype='application/json')

    # modifica della modalità di funzionamento in base al comando invocato dalla richiesta #
    if list(data.keys())[0] == "cooling":

        eval(hvac_id).cooling_on()
        response = {"cooling": "ON", "heating": "OFF", "off": "false"}

    elif list(data.keys())[0] == "heating":

        eval(hvac_id).heating_on()
        response = {"cooling": "OFF", "heating": "ON", "off": "false"}

    elif list(data.keys())[0] == "off":

        eval(hvac_id).switch_off()
        response = {"cooling": "OFF", "heating": "OFF", "off": "true"}

    elif list(data.keys())[0] == "ventilationON":

        eval(hvac_id).ventilationON()
        response = {"ventilationON": "true", "ventilationOFF": "false"}

    elif list(data.keys())[0] == "ventilationOFF":

        eval(hvac_id).ventilationOFF()
        response = {"ventilationON": "false", "ventilationOFF": "true"}

    return Response(response=json.dumps(response), status=200, mimetype='application/json')
```

Figura 23: *API raggiungibile tramite localhost:80/iot/hvac_id*

- "src/sensors/Dockerfile": Ha lo stesso funzionamento del Dockerfile precendente. Esso definisce l'immagine per l'applicazione che gestisce i sensori. *[fig. 24]*

```
1   FROM python:3.11-alpine3.16
2
3   COPY requirements.txt .
4
5   RUN python3 -m pip install -r requirements.txt --no-cache-dir
6
7   COPY . .
8
9   EXPOSE 5001
10
11  ENV FLASK_APP=sensors.py
12  CMD ["flask", "run", "--host=0.0.0.0", "--port=5001"]
```

Figura 24: *Schermata del file Dockerfile relativa al web server Flask sensors*

- "src/sensors/requirements.txt": File che contiene i moduli necessari da importare nell'applicazione della quale generiamo l'immagine attraverso il relativo Dockerfile. Le dipendenze inserite sono le stesse viste in figura 18.
- "src/sensors/sensors.py": Web server Flask scritto in Python che simula il comportamento di un HVAC. Esso è composto dalle seguenti parti:
    - Inizializzazione delle variabili globali *[fig. 25]*. Si specifica che:
        - *freq* indica la frequenza di creazione dei valori di temperatura, di qualità dell'aria e quella di aggiornamento per il sensore che calcola il consumo. Per fini simulativi e per dimostrare il comportamento dell'applicativo, si è supposto che i "freq" secondi impostati corrispondano a 1 ora trascorsa nella realtà.
        - *increasingRatio* / *decreasingRatio* rappresentano il rapporto di incremento e decremento della temperatura in caso di accensione rispettivamente del sistema di riscaldamento e di condizionamento. Per scopi simulativi sono stati impostati pari a 1° (es. nel caso di accensione del sistema di riscaldamento, la temperatura aumenterà di 1° ogni *freq* secondi)

```
12  # Frequenza di aggiornamento della temperatura #
13  freq = 5
14
15  # Stagioni #
16  seasons = ["summer", "winter"]
17
18  # Generazione casuale stagione: la variabile globale permette alle istanze dei sensori
19  # temperatura di avere tutti la stessa stagione per la simulazione
20  seasonIN = random.choice(seasons)
21
22  # Rateo di incremento/decremento della temperatura #
23  increasingRatio = 1
24  decreasingRatio = 1
```

Figura 25: *Variabili utilizzate per modellare il sensore di temperatura*

- Definizione della classe TemperatureSensor *[fig. 26]*. Il funzionamento della temperatura è regolato attraverso la selezione casuale di una delle due possibili stagioni ("Summer" oppure "Winter"). In base alla stagione selezionata ci sarà una configurazione differente della temperatura iniziale con una variazione della stessa attraverso l'impiego della funzione *random.uniform(),* in modo tale che simuli in maniera plausibile il clima della stagione selezionata. La generazione della temperatura è influenzata inoltre dallo stato della funzionalità di riscaldamento/raffreddamento degli hvac e pertanto è necessario conoscere tali parametri prima della generazione. In figura 31 è mostrato questo ultimo passaggio.

```python
# Definizione della classe TemperatureSensor che simula un sensore di temperatura #
class TemperatureSensor:

    def __init__(self):
        # Inizializzazion delle variabili #
        self.season = seasonIN
        self.cooling = 0
        self.heating = 0
        self.consume = 0.01     # kW/h

        # Temperature iniziali per le due stagioni #
        if self.season == "summer":
            self.temperature = 30
        elif self.season == "winter":
            self.temperature = 15

    # Funzione per ottenere la misura della temperatura #
    def get_temperature(self):

        if self.cooling == 1:
            self.temperature = self.temperature - decreasingRatio
        elif self.heating == 1:
            self.temperature = self.temperature + increasingRatio
        else:
            if self.season == "summer":
                # Generazione di un numero casuale da una distribuzione uniforme [-1,2]
                # e modifica della temperatura corrente #
                self.temperature = self.temperature + random.uniform(-1, 2)
            elif self.season == "winter":
                # Generazione di un numero casuale da una distribuzione uniforme [-2,1]
                # e modifica della temperatura corrente #
                self.temperature = self.temperature + random.uniform(-2, 1)

        return self.temperature
```

Figura 26: *Classe usata per simulare un sensore di temperatura*

- Definizione della classe AirQualitySensor, delle sue variabili e del suo metodo per ottenere il livello di CO2 *[fig. 27]*. Partendo da un valore iniziale, questo verrà variato in base allo stato del sistema di ventilazione.

```python
# Definizione della classe AirQualitySensor che simula un sensore di
# qualità dell'aria che misura la concentrazione di CO2 #
class AirQualitySensor:

    def __init__(self):
        # Inizializzazione delle variabili #
        self.level = 600        # livello iniziale di CO2 #
        self.ventilation = 0
        self.consume = 0.01     # kW/h

    # Funzione per ottenere la misura del livello di CO2 #
    def get_level(self):
        if self.ventilation == 0:
            self.level = self.level + random.uniform(-50, 150)
        else:
            self.level = self.level + random.uniform(-150, 0)

        return self.level

```

Figura 27: *Classe usata per simulare un sensore di qualità dell'aria*

- Creazione delle istanze dei sensori di temperatura e dei sensori di qualità dell'aria *[fig. 28]*

```
82   # Creazione delle istanze per simulare 2 sensori di temperatura #
83   temperature001 = TemperatureSensor()
84   temperature002 = TemperatureSensor()
85
86
87   # Creazione delle istanze per simulare 2 sensori di qualità dell'aria #
88   airquality001 = AirQualitySensor()
89   airquality002 = AirQualitySensor()
```

Figura 28: *Creazione di due device di tipo "Temperature Sensor" e due device di tipo "Air Quality Sensor"*

- Definizione della classe MultimeterSensor, con le sue variabili e con il metodo per calcolare il consumo energetico cumulativo. *[fig. 29]*

```
89   # Definizione della classe MultimeterSensor che simula un sensore per il calcolo
90   # della potenza assorbita dai dispositivi elettrici presenti nelle varie stanze #
91   class MultimeterSensor:
92
93       def __init__(self):
94           # Inizializzazione delle variabili #
95           self.consume = 0.01     # kW/h #
96           self.totalConsume = 0   # consumo totale #
97
98       # Funzione per ottenere il valore di consumo attuale #
99       def get_consume(self, cooling, heating, ventilation):
100
101          if (cooling == 1 or heating == 1) and ventilation == 0:
102
103              res = requests.get('http://hvac-actuator:80/iot/hvac001/consume')
104              stat = res.json()
105              hvac_consume = float(stat["consume"])
106
107          elif (cooling == 1 or heating == 1) and ventilation == 1:
108
109              res = requests.get('http://hvac-actuator:80/iot/hvac001/consume')
110              stat = res.json()
111              hvac_consume = float(stat["consume"]) * 2   # raddoppiamo il consumo poichè sono attive due modalità contemporaneamente
112
113          elif cooling == 0 and heating == 0 and ventilation == 1:
114
115              res = requests.get('http://hvac-actuator:80/iot/hvac001/consume')
116              stat = res.json()
117              hvac_consume = float(stat["consume"])
118
119          else:
120
121              hvac_consume = 0
122
123
124          return temperature001.consume + airquality001.consume + hvac_consume
```

Figura 29: *Classe usata per simulare un sensore per il calcolo del consumo di energia elettrica*

Considerando l'ipotesi descritta quando è stata introdotta la variabile *freq*, il calcolo del consumo prevede il controllo dei devices hvac attivi a intervalli di freq secondi. Pertanto è possibile modificare il loro stato solamente con tale risoluzione

(nell'ipotesi ogni ora). Il consumo totale viene allora calcolato sommando il consumo dei vari dispositivi attivi (ogni consumo è pari al prodotto dell'attributo consume, in kW, per l'ora di utilizzo ipotizzata).

- Creazione dell'istanza del sensore multimetro *[fig. 30]*

```
131    # Creazione del sensore 'multimeter001' #
132    multimeter001 = MultimeterSensor()
```

Figura 30: *Creazione di un device di tipo "Multimeter Sensor"*

- Definizione del loop infinito il cui contenuto viene eseguito per tutta la durata dell'esecuzione del sistema. Vengono ottenute attraverso delle "GET" lo stato dell'hvac, la misura della temperatura e quella della qualità dell'aria. *[fig. 31]*

```
136  # Loop infinito #
137  while(1):
138
139      #eseguiamo le successive operazioni per ogni sensore di temperatura e di qualità
140      #dell'aria presente (2 in questo caso) #
141      for i in range(1,3):
142
143          # Richiesta GET all'attuatore i-esimo per leggerne lo stato #
144          res = requests.get('http://hvac-actuator:80/iot/hvac00'+str(i))
145          stat = res.json()
146
147          tempId = "temperature00"+str(i)     # id del sensore di temperatura i-esimo
148          airId = "airquality00"+str(i)       # id del sensore di qualità dell'aria i-esimo
149          roomId = "room00"+str(i)          # id dell'entità room i-esima
150
151          # Impostazione degli attributi delle classi TemperatureSensor e AirQualitySensor
152          #necessari per la corretta simulazione delle misure #
153          # eval permette di trasformare una stringa in una variable Python
154          eval(tempId).cooling = int(stat["cooling"])
155          eval(tempId).heating = int(stat["heating"])
156          eval(airId).ventilation = int(stat["ventilation"])
157
158          # Lettura del livello di CO2 dal sensore di qualità dell'aria i-esimo #
159          lvl = eval(airId).get_level()
160
161          # Lettura della temperatura dal sensore di temperatura i-esimo #
162          temp = eval(tempId).get_temperature()
163
164          # Lettura della misura del consumo totale dal sensore di calcolo della potenza assorbita #
165          # Il consumo totale attuale sarà cumulativo, ogni freq secondi si aggiungerà il consumo
166          #dovuto ai vari dispositivi accessi nelle due stanze #
167          multimeter001.totalConsume = multimeter001.totalConsume + multimeter001.get_consume(eval(tempId).cooling,
168                                       eval(tempId).heating, eval(airId).ventilation)
```

Figura 31: *Prima parte del ciclo: lettura della temperatura e del livello di CO2 corrente dal sensore i-esimo, e aggiornamento del livello di consumo*

- Successivamente vengono inviate all'IoT Agent le misure ottenute dai sensori, con lo scopo di aggiornare i corrispettivi attributi nei device. Inoltre, queste stesse misure vengono inviate a Orion per aggiornare i corrispettivi attributi delle entità room. Tutto questo viene effettuato ciclicamente ogni "freq" secondi. [*fig. 32*]

```
162         # INVIO MISURE ALL'IOT-AGENT tramite richieste POST#
163
164         # Invio livello di CO2 all'entità del sensore di qualità dell'aria i-esimo #
165         body = {'CO2_level': str(round(lvl, 1))}
166         res = requests.post('http://fiware-iot-agent:7896/iot/json?k=4jggokgpepnvsb2uv4s40d59ov&i='+ airId, data = json.dumps(body),
167                             headers={'Content-Type': 'application/json'})
168
169         # Invio temperatura all'entità del sensore di temperatura i-esimo #
170         body = {'temperature': str(round(temp, 1))}
171         res = requests.post('http://fiware-iot-agent:7896/iot/json?k=4jggokgpepnvsb2uv4s40d59ov&i=' + tempId, data = json.dumps(body),
172                             headers={'Content-Type': 'application/json'})
173
174
175         # INVIO MISURE ALL'ENTITA' ROOM CORRISPONDENTE NEL CONTEXT BROKER tramite richieste PATCH #
176
177         # Aggiornamento del livello di CO2 generato dal sensore di qualità dell'aria i-esimo nell'entità room i-esima #
178         res = requests.patch('http://orion:1026/v2/entities/urn:ngsi-ld:Room:'+roomId+'/attrs?type=Room',
179                              data = json.dumps({"CO2_level":{"type":"Float", "value": str(round(lvl, 1))}}), headers={'Content-Type': 'application/json'})
180
181         # Aggiornamento della temperatura generata dal sensore di temperatura i-esimo nell'entità room i-esima #
182         res = requests.patch('http://orion:1026/v2/entities/urn:ngsi-ld:Room:'+roomId+'/attrs?type=Room',
183                              data = json.dumps({"temperature":{"type":"Float", "value": str(round(temp, 1))}}), headers={'Content-Type': 'application/json'})
184
185         # end for
186
187     # Aggiungo il consumo del multimetro #
188     multimeter001.totalConsume = multimeter001.totalConsume + multimeter001.consume
189
190     # Invio consumo totale attuale all'entità del multimeter sensor #
191     body = {'consume': str(round(multimeter001.totalConsume, 2))}
192     res = requests.post('http://fiware-iot-agent:7896/iot/json?k=4jggokgpepnvsb2uv4s40d59ov&i=multimeter001', data = json.dumps(body),
193                         headers={'Content-Type': 'application/json'})
194
195     # Attesa prima di una nuova misura pari a freq secondi #
196     time.sleep(freq)
```

Figura 32: *Seconda parte del ciclo: invio delle misure rilevate ai relativi device tramite l'IoT Agent e alle entità Room su Orion per aggiornare i rispettivi attributi*

PRESENTAZIONE DELLA DASHBOARD

Per la realizzazione della dashboard è stata utilizzata React, una libreria javascript frontend e open source per lo sviluppo di interfacce utente basata su componenti UI reattivi e scalabili [10].

React è stata usata in combinazione con un server Node.js raggiungibile sulla porta 3000 che gestisce le logiche di backend.

La gerarchia dei file rilevanti necessari per l'implementazione è riportata di seguito:

```
|--react-docker
    |-- node-modules
    |-- dockerignorefile
    |-- Dockerfile
    |-- package.json
    |-- package-lock.json
    |-- public
        |--index.html
    |--src
        |--API
            |--API.js
        |--components
            |-- InfoMeasure.css
            |-- InfoMeasure.js
            |-- NavBar.css
            |-- NavBar.js
            |-- Room.css
            |-- Room.js
        |-- App.css
        |-- App.js
        |-- index.css
        |-- index.js
```

- "node-modules": è una directory che viene generata automaticamente all'esecuzione del comando "npm install" in un qualsiasi progetto basato su Node.js. Contiene tutte le dipendenze che sono stata installate per il corretto funzionamento dell'intero sistema.
- "dockerignorefile": è un file di configurazione che specifica quali file e cartelle non devono essere inclusi nella costruzione di un'immagine Docker. È stata esclusa la directory "node-modules" per ridurre le dimensioni dell'immagine. *[fig. 33]*

```
node_modules
npm-debug.log
build
.git
*.md
.gitignore
```

Figura 33: *Schermata del file .dockerignorefile*

- "Dockerfile": ha lo stesso funzionamento dei Dockerfile visti in precedenza per la creazione delle immagini relative ai sensori e all'hvac. Con il comando "npm start" viene avviato il server web che ospita l'applicazione che utilizza una configurazione specificata nel file "package.json". *[fig. 34]*

```
# Fetching the latest node image on alpine linux
FROM node:alpine AS development

# Declaring env
ENV NODE_ENV development

# Setting up the work directory
WORKDIR /react-app

# Installing dependencies
COPY ./package.json /react-app
RUN npm install
RUN apk add xdg-utils

# Copying all the files in our project
COPY . .

# Starting our application
CMD npm start
```

Figura 34: *Schermata del Dockerfile relativo al web server Node.js della dashboard*

- "public": è una directory che contiene file che devono essere resi pubblicamente accessibili dal server web. Include il file "index.html" che è il file principale che viene caricato quando l'utente accede all'applicazione.
- "src/API/API.js": è un file javascript che contiene la logica per interagire con le API messe a disposizione da Orion Context Broker per fornire i dati alla dashboard. Ogni funzione utilizza la sintassi "async/await" per gestire la richiesta in modo asincrono. Viene utilizzata poi la funzione "fetch" per effettuare la richiesta HTTP la quale restituisce una "Promise" (la risposta) che viene gestita tramite i metodi "then" e "catch". *[fig. 35]*

```
34  // API per ottenere le informazioni sullo stato degli HVAC presenti
35  async function getHVAC_status() {
36      const settings = {
37          method: 'GET',
38          headers : {
39              "fiware-service": "openiot",
40              "fiware-servicepath": "/"
41          }
42      };
43      const response = await fetch(URL, settings);
44      const hvacStatus = await response.json();
45      if (response.ok) {
46          return hvacStatus.filter(function (obj) {
47              return obj.id.includes("urn:ngsi-ld:Device:hvac")}).sort (function (a,b) {
48                  var collator = new Intl.Collator([], {numeric: true});
49                  return collator.compare(a.id, b.id)}).map((h) => (
50                      {id: h.id, cooling_info: h.cooling_info.value,
51                          heating_info: h.heating_info.value,
52                          ventilation_info: h.ventilationON_info.value, refRoom: h.refRoom.value}) )
53      } else {
54          throw hvacStatus;
55      }
56  }
```

Figura 35: *Esempio di funzione per ottenere lo stato dei device HVAC. La funzione ottiene tale informazione tramite una richiesta GET all'URL localhost:1026/v2/entities inserendo gli opportuni headers.*

- "components": è una directory in cui sono presenti tutti i componenti React utilizzati. Questi ultimi sono elementi modulari utilizzati per comporre l'interfaccia utente e sono costituiti da un insieme di elementi HTML e javascript [10]. Ogni file .js nella directory rappresenta un singolo componente a cui è associato il relativo foglio di stile .css. I componenti utilizzati sono tutti componenti funzionali, ovvero non hanno uno stato interno ma accettano proprietà ("props") dai componenti padre come argomenti.

- "components/InfoMeasure.js": è il componente adibito alla visualizzazione della stagione simulata corrente e del consumo totale a partire da quando è stato avviato il sistema. È stato creato un unico componente che esegue entrambe le funzioni grazie alla renderizzazione condizionale basata sulla proprietà "type" passata come argomento. *[fig. 36]*
  Viene evidenziato in verde in fig. 42.

```
function InfoMeasure(props) {

  //props passate al componente da App.js
  let type = props.info;
  let multiInfo = props.multimeterInfo;
  let currentSeason = props.currentSeason

  return (
    <div className="info--container">

      <div className="info--img--container">
        {type=="energy" ?
        <img className="info--img energy" src={multimeterSensorICON}></img>
        : currentSeason ==="summer" ? <img className="info--img season-sun" src={summerICON}></img> :
        <img className="info--img season-winter" src={winterICON}></img>
      }
      </div>


      <div className="info--text--container">
        {type=="season" ?
          <p className="info--text">Current season: {currentSeason} </p>
          : <p className="info--text">Total Consumed: {multiInfo.totalConsume} {multiInfo.unitCodeConsume} </p>
        }
      </div>


    </div>
  )
}
```

Figura 36: *Implementazione del componente InfoMeasure tramite renderizzazione condizionale*

- "components/Room.js": è il componente adibito alla visualizzazione delle informazioni relative alla singola entità "Room" presente su Orion come le misure di temperatura e qualità dell'aria con relativa unità di misura e lo stato corrente dell'attuatore HVAC. *[fig. 37]*
  Viene evidenziato in rosso in fig. 42.

```
function Room(props) {

  //props passate al componente da App.js
  let entityInfo = props.entityInfo;
  let roomNumber = props.roomNumber;
  let hvacStatus = props.hvacStatus;
  let tempDeviceInfo =props.tempDeviceInfo;
  let airDeviceInfo =props.airDeviceInfo;

  return (
    <div className='room'>

      <section className="room--header">
        <h1>ROOM {roomNumber}</h1>
        <h3>{entityInfo.name}</h3>
      </section>

      <div className="room--sensors">
        <p className="room--index">Sensors:</p>
        <div className="room--sensors--temperature">
          <img className="room--sensors--icon" src={tempSensorICON}></img>
          <p className="room--sensors--text">TEMPERATURE: <br></br> {entityInfo.temperature} {tempDeviceInfo?.unitCode === "CEL" ? "°" : "F" }</p>
        </div>
        <div className="room--sensors--airquality">
          <img className="room--sensors--icon" src={airqualitySensorICON}></img>
          <p className="room--sensors--text">CO2 LEVEL: <br></br> {entityInfo.CO2_level} {airDeviceInfo?.unitCode}</p>
        </div>
      </div>
      <div className="room--actuators">
        <p className="room--index">Actuators:</p>
        <div className="room--actuators--hvac">
          <img className="room--actuators--hvac--icon" src={hvacICON}></img>
          <div className="room--actuators--hvac--status">
            <div className="room--actuators-hvac-status-info">
              <p>COOLING:</p>
              {hvacStatus?.cooling_info === "ON" ? <img src={onICON}></img> : <img src={offICON}></img> }
            </div>
            <div className="room--actuators-hvac-status-info">
              <p>HEATING:</p>
              {hvacStatus?.heating_info === "ON" ? <img src={onICON}></img> : <img src={offICON}></img> }
            </div>
            <div className="room--actuators-hvac-status-info">
              <p>VENTILATION:</p>
              {hvacStatus?.ventilation_info === "true" ? <img src={onICON}></img> : <img src={offICON}></img>}
            </div>
          </div>
        </div>
      </div>
    </div>
  )
}
```

Figura 37: *Implementazione del componente Room per la visualizzazione delle misure rilevate dai sensori*

- App.js'': è il file che contiene il codice che renderizza il componente principale dell'applicazione, la logica di business e la definizione degli altri componenti utilizzati. Sono stati implementati due "hooks" (funzioni per accedere a determinate features di React):
    1. *useState*: mantiene lo stato interno del componente, ovvero un oggetto che memorizza informazioni relative all'applicazione e inoltre fornisce una funzione per aggiornare tale stato. Questo è stato utilizzato per tenere traccia dei dati ricevuti dalle API [10]. *[fig. 38]*

2. *useEffect*: permette di eseguire "effetti collaterali" all'interno di un componente, ovvero azioni che vengono eseguite al di fuori del normale flusso di rendering. Questo è stato utilizzato per eseguire le chiamate alle API ogni 5 secondi, cioè ad ogni variazione delle informazioni di contesto. Tale comportamento è stato implementato attraverso un array che viene aggiornato con la stessa frequenza innescando una nuova renderizzazione dei componenti [10]. *[fig. 39]*

La renderizzazione dei componenti "Room", è stata effettuata mediante un ciclo "map" sull'array di elementi memorizzati nello stato avvalorato dalla chiamata alla API "getEntities".Inoltre viene utilizzato l'attributo "refRoom" presente nei devices (che descrive la relazione con l'entità room nella quale il singolo device è installato) per poter filtrare per ciascuna stanza i dispositivi presenti. Gli stati avvalorati vengono poi passati ai componenti che li usano come proprietà in modo immutabile. *[fig. 40]*

```
//stato contenente l'array che viene restituito dalla API getEntities(): informazioni sulle entità rooms
const [entitiesInfo, setEntitiesInfo] = useState([])

//stato contenente le informazioni sul multimetro che vengono restituite dalla API getConsume(): consumo e unità di misura
const [multimeterInfo, setMultimeterInfo] = useState([])

//stato contenente l'array che viene restituito dalla API getHVAC_status(): stato degli HVAC
const [hvacStatus, setHvacStatus] = useState({})
```

Figura 38: *Esempi di useState*

```
//per aggiornare lo stato entitiesInfo ogni 5 secondi
useEffect(() => {
  const prova = async() => {
      await API.getEntities()
          .then( (entities) => {setEntitiesInfo(entities);}
          )
          .catch( err => console.log(err))
  }
  prova()
},[time])
```

Figura 39: *Esempio di useEffect*

```
return (
  <div className="app">
    <Navbar />


    <div className="measureContainer">
      <InfoMeasure info={typeSeason} currentSeason={currentSeason} />
      <InfoMeasure info={typeConsume} multimeterInfo={multimeterInfo} />
    </div>

    <div className="room--container">
      {entitiesInfo.map( (entity,index) => (
                <Room
                key={entity.id}
                entityInfo={entity}
                roomNumber={index+1}
                hvacStatus={hvacStatus.filter( function(hvac) {return hvac?.refRoom == entity.id})[0]}
                tempDeviceInfo={tempDevice.filter( function(temp) {return temp?.refRoom ==entity.id})[0]}
                airDeviceInfo={airDevice.filter( function(air) {return air?.refRoom == entity.id})[0]}
                 />
              ))}
    </div>

  </div>
);
```

Figura 40: *Implementazione della renderizzazione del componente principale App. Si noti l'uso dell'attributo di tipo relationship presente nei devices per filtrare quali dispositivi sono presenti in ciascuna stanza.*

- "index.js": è il file principale dell'applicazione che contiene il codice per avviare l'applicazione React utilizzando il metodo "ReactDOM.render" per renderizzare l'applicazione nell'elemento HTML con ID "root". *[fig. 41]*

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Figura 41: *Schermata del file index.js*

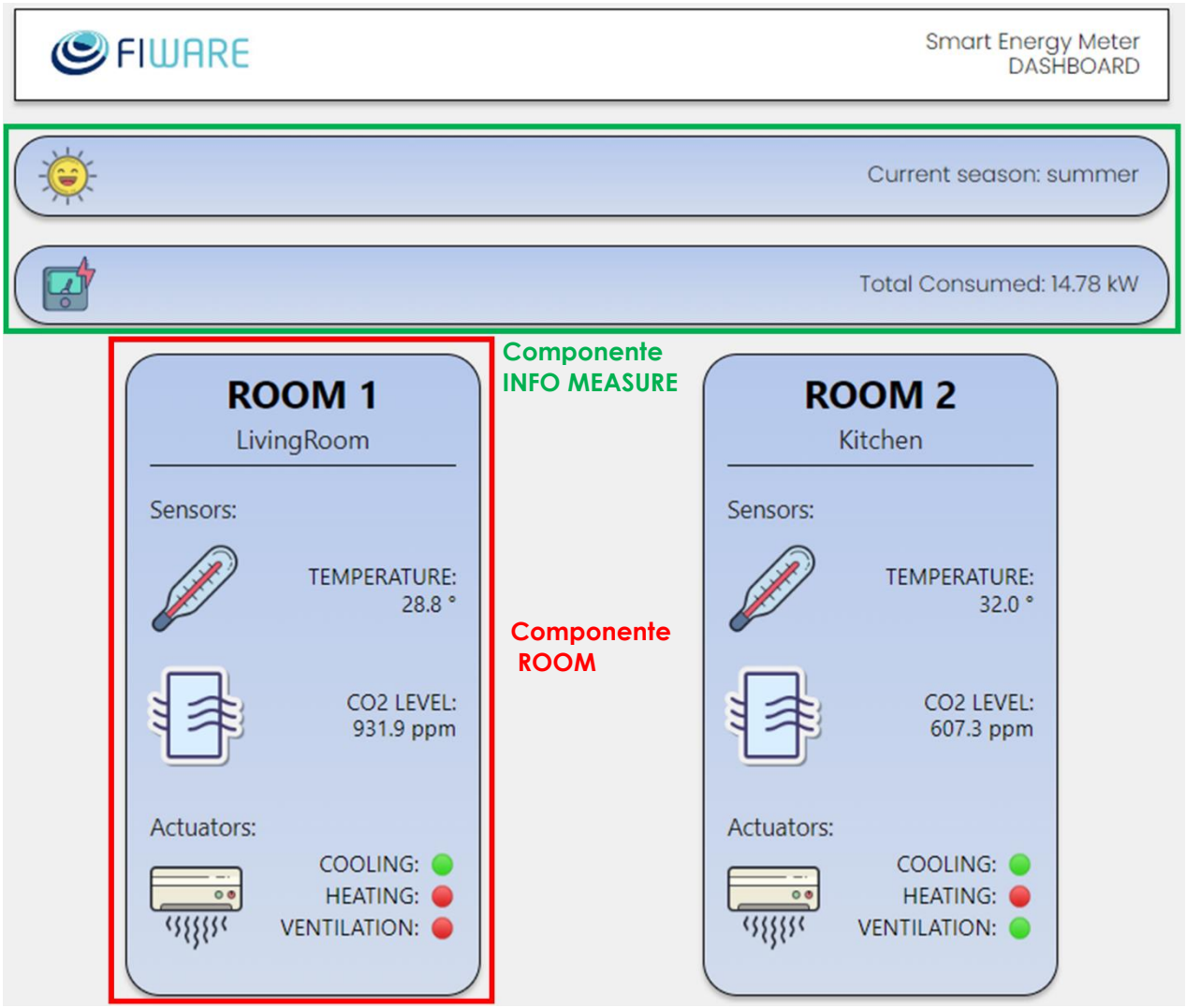


Figura 42: *Schermata della dashboard in esecuzione, raggiungibile su localhost:3000*

FLUSSO DI LAVORO E ORGANIZZAZIONE DEL TEAM

Il progetto è stato svolto nell'arco di circa 30 giorni e il lavoro è stato suddiviso e organizzato nel seguente modo:

Data	Tasks	Ore di lavoro
17-01-2023	Pianificazione della suddivisione del carico di lavoro	3
18-01-2023	Introduzione ai concetti e alle tecnologie FIWARE.	4
19-01-2023	Introduzione ai concetti e alle tecnologie FIWARE. (Tutorial IoT Sensors)	4
20-01-2023	Introduzione ai concetti e alle tecnologie FIWARE. (Tutorial IoT Agent Ultralight)	4
21-01-2023	-	-
22-01-2023	-	-
23-01-2023	Introduzione ai concetti e alle tecnologie FIWARE. (Tutorial IoT Agent JSON)	4
24-01-2023	Introduzione ai concetti e alle tecnologie FIWARE. (Tutorial Perseo)	4
25-01-2023	Definizione della struttura di base del progetto	2
26-01-2023	Definizione dell'architettura del progetto e aggiunta delle prime entità nel sistema tramite Postman	5
27-01-2023	Scrittura del file "Docker-compose" e relativi test	3
28-01-2023	Pianificazione delle funzioni e delle API necessarie per il funzionamento dei server Flask	3
29-01-2023	-	-
30-01-2023	Scrittura dei file "sensors.py" e "Dockerfile" e relativi test	5
31-01-2023	Scrittura dei file "hvac.py" e "Dockerfile" e relativi test	4

01-02-2023	Test tramite Postman delle Subscription, delle Regole e dei comandi	4
02-02-2023	Aggiornamento di “sensors.py” e “hvac.py” (aggiunta delle API)	4
03-02-2023	Debugging	3
04-02-2023	Debugging	4
05-02-2023	-	-
06-02-2023	Scrittura del file “services.sh” e relativi test	3
07-02-2023	Inserimento di nuovi device con relativi test	3
08-02-2023	Debugging	4
09-02-2023	Introduzione ai sistemi di dashboard e alle interfacce utente per Docker	5
10-02-2023	Scrittura dei file necessari per la dashboard	4
11-02-2023	Proseguimento dei lavori per la dashboard e relativi test	3
12-02-2023	-	-
13-02-2023	Scrittura della documentazione	5
14-02-2023	Scrittura della presentazione	4
15-02-2023	Correzioni e rifiniture	3

LAVORI E PROGETTI FUTURI

Come conseguenza delle informazioni e delle tecnologie apprese durante l'esecuzione di questo progetto, il team di lavoro mira a effettuare in futuro un conseguente miglioramento dello stesso. I miglioramenti che si intende effettuare sono:

- Implementazione di un numero superiore di stanze e, eventualmente, la possibilità di poter gestire anche più case contemporaneamente.
- Implementazione di sensori per rilevare altre grandezze, come sensori per la rilevazione dell'umidità nell'aria.
- Implementazione di attuatori specifici relativi ai nuovi sensori introdotti, come un device IoT in grado di attivare un sistema di deumidificazione.
- Implementazione di diverse tipologie di dispositivi smart, come lampadine e prese elettriche gestibili a distanza dall'utente.
- Implementazione di un algoritmo in grado di fare un'analisi dei consumi dei dispositivi accesi e segnalare all'utente delle *best practices* per ridurre ulteriormente i consumi.
- Ulteriore rifinitura dell'interfaccia grafica, in maniera tale che fornisca ancora più informazioni potenzialmente utili all'utente.
- Introduzione di una storicizzazione dei consumi utilizzando un database NoSQL di tipo chiave-valore collegato a Fiware tramite l'ulteriore componente Cygnus, oppure utilizzando STH-COMET basato su MongoDB per la creazione di una vista storica dei dati e il calcolo di indici di sintesi.

FONTI

[1] https://docs.docker.com/

[2] https://learning.postman.com/docs/getting-started/introduction/

[3] https://flask.palletsprojects.com/en/2.2.x/#user-s-guide

[4] https://nodejs.org/en/docs/guides/

[5] https://www.mongodb.com/docs/compass/current/

[6] https://fiware-tutorials.readthedocs.io/en/latest/

[7] https://fiware-perseo-fe.readthedocs.io/en/latest/

[8] https://www.slideshare.net/flopezaguilar/creating-a-contextaware-solution-complex-event-processing-with-fiware-perseo

[9] https://github.com/telefonicaid/perseo-fe/tree/master/examples

[10] https://reactjs.org/