

UNIVERSITÀ DI PISA

CORSO DI LAUREA IN INFORMATICA

**Design e sviluppo di un framework per
la generazione di interfacce
multimodali in contesto industria 4.0**

Autore:

William SIMONI

Relatore:

Dr. Daniele MAZZEI

ANNO ACCADEMICO 2019/2020



Indice

1	Introduzione	7
1.1	Industria 4.0	7
1.1.1	L'internet delle cose	8
1.2	L'obbiettivo del progetto	8
1.3	Struttura del progetto	8
1.3.1	Zerynth Device Manager	9
1.3.2	Fetching dei dati dalla ZDM	10
1.3.3	Architettura generale	10
2	File di configurazione	12
2.1	Processo di creazione	12
2.1.1	La scelta del formato	12
2.2	Struttura della configurazione 1.0	13
2.2.1	Meta-dati del progetto	13
2.2.2	Viste e dati	14
2.2.3	Allarmi	17
2.2.4	Utenti	17
2.2.5	Tipi di dato	18
2.3	Generatore di configurazioni	19
3	Interfacce utente	20
3.1	login	20
3.2	Visualizzazione dei dati	20
3.2.1	Interfaccia grafica	20
3.2.2	Interfaccia linguaggio naturale	22
3.3	Tecnologie utilizzate	23
3.3.1	Interfaccia grafica	23
3.3.2	Interfaccia linguaggio naturale	23
4	FiBo for Filtering Border	24
4.1	Funzionalità	24
4.1.1	Architettura generale	24
4.1.2	Tecnologie utilizzate	26
4.2	Storage dei dati	28
4.2.1	Storage delle time series	28
4.2.2	Storage degli altri dati	30
4.3	Login	31

4.3.1	Interfaccia http	31
4.3.2	Implementazione	31
4.4	Logout	33
4.4.1	Interfaccia http	33
4.4.2	Implementazione	33
4.5	Validazione del JWT	33
4.6	Richiesta di dati	34
4.6.1	Interfaccia http	34
4.6.2	Implementazione	35
4.7	Creazione ed eliminazione di un progetto	39
4.8	Generazione degli allarmi	39
4.9	Notifica degli allarmi	40
4.9.1	Interfaccia socket.io	40
4.9.2	Implementazione	40
4.10	Organizzazione del codice	41
4.11	Test sulle richieste dei dati	42
4.11.1	Caching e memoizzazione	42
5	FiBo CLI	45
5.1	Implementazione	45
5.1.1	Tecnologie utilizzate	45
5.1.2	Funzionalità	45
5.2	Esperienza di utilizzo	46
5.2.1	Assistenza	46
5.2.2	Login e logout	46
5.2.3	creazione ed eliminazione del progetto	47
6	Conclusioni	48
6.1	Sviluppi Futuri di FiBo	48
6.1.1	Sicurezza	48
6.1.2	Clustering	48
6.1.3	Aggiornamento del token	49
6.1.4	Fetching continuo dei dati	49
6.2	Conclusioni	49
6.3	Ringraziamenti	50
A	Configurazione di esempio	51
B	Schemi del process database	54
C	Studio sui grafici	59
C.1	I grafici, elenco e regole di design	59
C.1.1	Icon chart e gauge chart	59
C.1.2	Column e bar chart	59
C.1.3	Line chart	60
C.1.4	Pie chart e donut chart	60
C.1.5	Area chart e versioni stacked di column chart e line chart	61

C.1.6 Scatter plot	63
C.1.7 Bubble chart	64
C.1.8 Histogram	65
C.2 Sviluppi	66
C.3 Fonti	66
Bibliografia	67
Bibliografia appendice C	69

Elenco delle figure

1.1	Architettura del sistema.	11
2.1	Generatore di configurazioni	19
3.1	login dell'utente	20
3.2	Pagina generata per la vista Umidità	21
3.3	Pagina generata per la vista Temperatura	21
3.4	Interazione dell'utente con il chatbot	22
3.5	Il chatbot nega l'accesso all'utente non autorizzato	23
4.1	Casi d'uso FiBo	25
4.2	Comunicazione tra i componenti di FiBo	26
4.3	Calcolo dei timestamp di una time series	29
4.4	Diagramma di attività login	32
4.5	Diagramma di attività logout	33
4.6	Esempio di aggregazione	38
4.7	Macchina a stati per la gestione di una room dell'alarm sender	41
4.8	Test 1 con 100 thread	44
4.9	Test 2 con 300 thread	44
5.1	Assistenza CLI	46
5.2	Login e logout	47
5.3	Errori nel file di configurazione	47
5.4	Creazione del progetto terminata con successo	47
5.5	Altri esempi con la CLI	47
6.1	Proxy inverso	48
B.1	Modello concettuale project database	54
B.2	Modello logico project database	55
C.1	Bar chart	60
C.2	Pie Chart distorto	61
C.3	Pie Chart 2D	61
C.4	Donut Chart 2D	61
C.5	Area chart e column chart	62
C.6	Regular area chart	62
C.7	Stacked line chart	62

C.8 100% stacked line chart	62
C.9 Scatter plot	63
C.10 Bubble chart 1	64
C.11 Bubble chart 2	65
C.12 Histogram	66

CAPITOLO

1 | Introduzione

Il lavoro descritto in questa tesi si inquadra in un progetto più ampio di cui ho fatto parte insieme ad altri miei compagni di corso, e sotto la supervisione del professore Daniele Mazzei. In questo capitolo introduttivo viene trattato l'obiettivo del progetto, l'architettura del sistema che abbiamo progettato e per concludere come ci siamo divisi il lavoro.

1.1 Industria 4.0

Nel 2011, alla Hannover Messe, una fiera sulle tecnologie industriali, Henning Kagermann, Wolf-Dieter Lukas e Wolfgang Wahlster pronunciarono il termine Zukunftsprojekt Industrie 4.0 riferendosi al piano industriale tedesco, poi concretizzato nel 2013, per riportare la manifattura tedesca ai vertici mondiali [1]. I risultati che ottenne la Germania portarono altri paesi a seguire questa politica; in Francia con il nome di Industrie du Futur, in Italia con il nome di Industria 4.0.

Nel 2013 la Germania iniziò quella che gli studiosi chiamano quarta rivoluzione industriale. In breve, un industria 4.0 è un'industria in cui i macchinari comunicano fra loro e sono collegati a un sistema che è in grado di visualizzare l'intera catena di produzione e compiere decisioni autonomamente [2]. Il termine inglese è *smart factory*.

Secondo uno studio del Boston Consulting Group le tecnologie, dette abilitanti, alla base dell'industria 4.0 sono [3][4]:

1. Big Data and Analytics;
2. Autonomous Robots;
3. Simulazioni;
4. Integrazione orizzontale e verticale;
5. **Industrial Internet of Things;**
6. Cloud;
7. Additive manufacturing;
8. Realtà aumentata;

9. Sicurezza informatica.

Di particolare interesse per il nostro progetto è il punto cinque; la "rete delle cose".

1.1.1 L'internet delle cose

L'internet delle cose (Internet of Things o IoT in inglese) è *l'estensione di internet al mondo degli oggetti e dei luoghi* [5]. Ogni oggetto, collegandosi e identificandosi in una rete, comunica dati su se stesso ed è in grado di accedere alle informazioni di altri oggetti [6]. In un industria 4.0, l'IoT consente non solo di avere macchine o linee di produzione in grado di eseguire determinate operazioni autonomamente, ma anche di poter accedere a una maggiore quantità di dati. Questi dati possono essere utilizzati per fare manutenzione predittiva, ottimizzare i processi o creare delle dashboard di controllo [7].

1.2 L'obiettivo del progetto

L'obiettivo del sistema è consentire a un qualsiasi tecnico esperto di IoT, ma non necessariamente di informatica, di creare dashboard industriali che siano accessibili sia attraverso un'interazione grafica che un'interazione in linguaggio naturale. Con la possibilità, in futuro, di poter aggiungere ulteriori modalità di interazione.

Per produrre il risultato finale bisognava mettersi nei panni di una persona tipo che rappresentasse l'utente finale del sistema, in inglese personas. Questa persona, che nel testo è indicata con il nome di **Iotu** (**I**nternet **O**f **T**hings **U**sers) è un esperto del mondo dell'**internet of things** la cui necessità è di dover creare delle interfacce attraverso le quali i suoi clienti possano visualizzare i dati prodotti dai vari oggetti IoT. Iotu vuole poter creare tali interfacce velocemente e semplicemente in modo da non preferire il copia-incolla alla creazione da zero.

Attualmente esistono vari servizi che permettono di gestire e visualizzare tramite GUI¹ i dati inviati dalla rete IoT. Per citarne alcuni: **Thingsboard.io** o **Ubidots**. D'altro canto non esistono servizi che permettano di creare dashboard interagibili tramite il linguaggio naturale.

La possibilità di poter accedere i dati in molteplici modi potrà quindi consentire ai fruitori della dashboard di richiedere i dati nel modo che ritengono più comodo e consono. Se vorranno osservare un trend probabilmente preferiranno l'interazione grafica. Se vorranno invece sapere la temperatura di un macchinario mentre sono impegnati in altre operazioni manuali difficilmente avranno un'altra mano per usare l'interfaccia grafica e quindi si rivolgeranno all'assistente virtuale.

1.3 Struttura del progetto

Definito l'obiettivo del progetto, è stata definita mano a mano che si parlava con il professore Mazzei la struttura del progetto:

¹L'interfaccia grafica (nota anche come GUI (dall'inglese Graphical User Interface), in informatica è un tipo di interfaccia utente che consente l'interazione uomo-macchina in modo visuale

1. L’utente che intende generare delle interfacce utilizzando il sistema deve generare un file di configurazione o a mano o utilizzando un tool di assistenza. Il file di configurazione deve essere strutturato in modo che sia leggibile e comprensibile per facilitare le modifiche e le letture future; inoltre, deve essere sufficientemente espressivo da supportare la generazione di sistemi multi-tenant. In altre parole, l’interfaccia deve cambiare in base ai dati a cui può accedere l’utente che la utilizza;
2. Questo unico file di configurazione deve poter permettere la generazione di interfacce grafiche o interfacce in linguaggio naturale. Quindi devono esistere un generatore di GUI e un generatore di interfacce in linguaggio naturale che leggendo il file di configurazione generano la rispettiva interfaccia;
3. L’interfaccia generata deve poter permettere all’utente che ne ha i diritti di accedere alle informazioni indicate nel file di configurazione.

Il tool di assistenza inizialmente doveva essere un semplice *wizard* a linea di comando, ma la creazione di un suo prototipo ha dimostrato che il processo di creazione era molto scomodo, per cui è stato creato un generatore di configurazioni più avanzato che assiste l’utente nella compilazione dei campi che compongono il file di configurazione [8].

1.3.1 Zerynth Device Manager

Il sistema si appoggia al servizio ZDM ²(Zerynth Device Manager) per la ricezione e memorizzazione dei dati inviati dai dispositivi IoT, per cui, ancor prima di generare il file di configurazione, l’utente deve creare un progetto sullo ZDM. La creazione di un progetto può avvenire o utilizzando un’interfaccia grafica web o utilizzando una CLI. Ogni progetto sulla ZDM si compone principalmente di tre elementi [9]:

- **workspace**, rappresenta il nodo principale nella gestione dei dispositivi Zerynth. Un workspace rappresenta un progetto contenente flotte (*fleet*) di dispositivi. Ogni workspace è assegnato a un id;
- **fleet**, rappresenta un insieme (una flotta) di device. Ogni fleet è assegnata a un id;
- **device**, rappresenta una periferica che genera dati. Ogni device è associato a uno o più **tag**. I device associati allo stesso tag dovrebbero avere una qualche caratteristica comune come la posizione o la funzione. Ogni device è assegnato a un id.

Questa organizzazione si è riflessa anche nella struttura del file di configurazione e consente in poche righe di fare riferimento a un’enorme mole di dati.

I dati ritornati da ogni device sono caratterizzati dai tag associati al device, dal timestamp in cui i dati sono stati rilevati, dall’id del device e da un **payload**. Il payload può contenere diversi valori. Ad ogni valore è associata una parola chiave

²Il servizio ZDM è stato sviluppato da Zerynth

il cui nome è **value**. Nel codice 1.1 il payload è composto da 3 valori associati rispettivamente ai value temp, humidity e pression.

Codice 1.1: esempio di dato inviato da un device

```
{
    "tag": "termometro",
    "timestamp_device": "2020-05-15T12:41:53",
    "device_id": "dev-esempio",
    "payload": {
        "temp": 19,
        "humidity": 14,
        "pression": 28
    }
}
```

1.3.2 Fetching dei dati dalla ZDM

Nello strutturare l’architettura era necessario decidere come le interfacce avrebbero ottenuto i dati memorizzati nei database di Zerynth. La soluzione è stata aggiungere un microservizio che facesse da intermediario tra l’interfaccia e lo ZDM. Il microservizio, che è stato chiamato **FiBo**, acronimo di Filtering Border, oltre a fare da intermediario si occupa di aggregare i dati e memorizzare per un certo periodo di tempo i risultati delle aggregazioni. L’aggiunta di questo microservizio consente quindi di:

- disaccoppiare il sistema dallo ZDM;
- evitare di inviare ad ogni interfaccia migliaia se non centinaia di migliaia di dati;
- ridurre il tempo di risposta grazie al caching dei dati aggregati.

1.3.3 Architettura generale

L’architettura è quindi composta da quattro moduli, ciascuno dei quali è stato progettato e scritto da un membro del team [8][10][11]. Io mi sono occupato della progettazione e dell’implementazione di FiBo, e di fare da collante tra i vari componenti del team. Il filo conduttore dei quattro moduli è il file di configurazione su cui abbiamo speso la maggior parte del tempo e che abbiamo continuato a modificare fino a poco prima della fine del progetto. I prossimi capitoli contengono una descrizione più approfondita dei vari moduli.

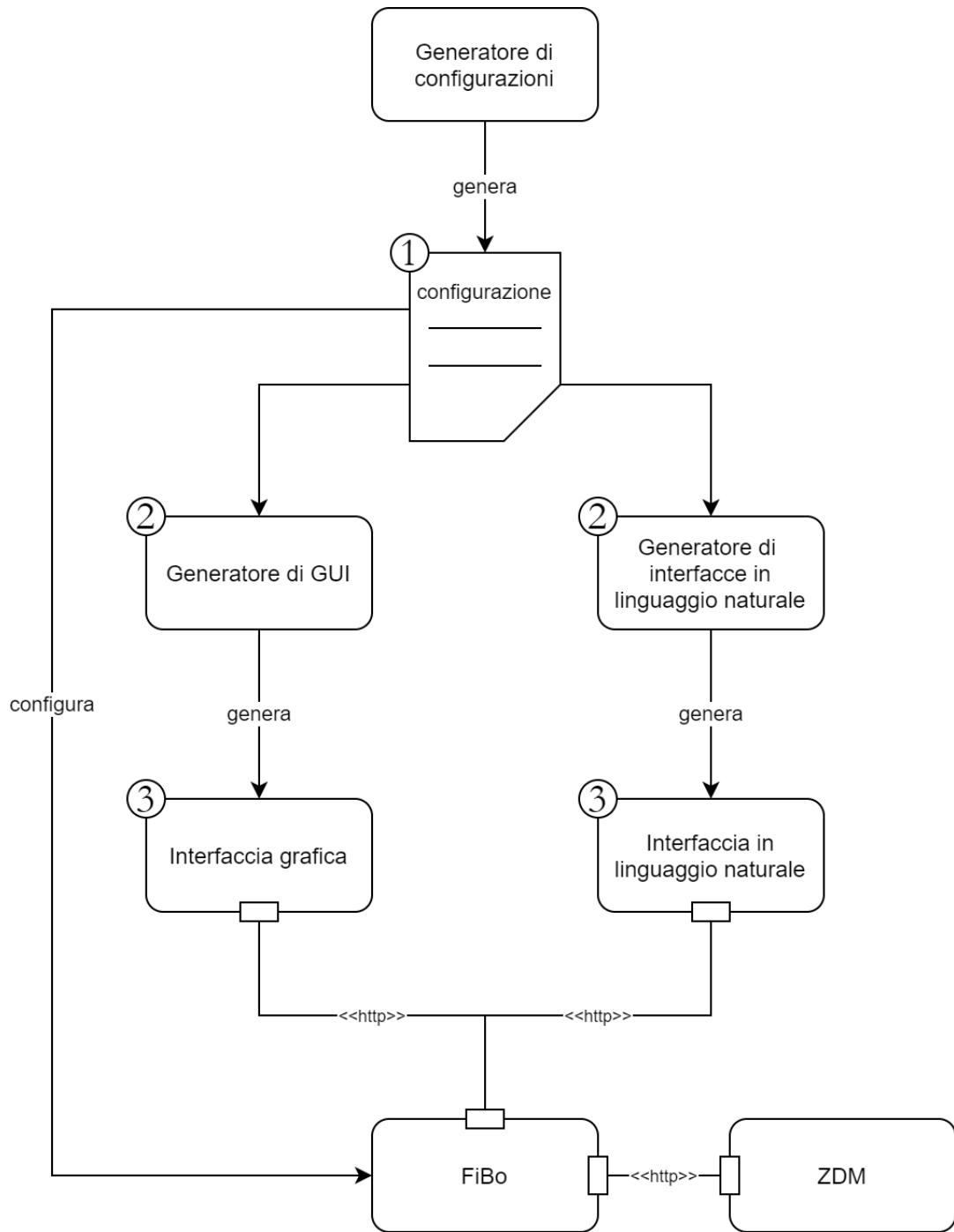


Figura 1.1: Architettura del sistema.

CAPITOLO 2

File di configurazione

2.1 Processo di creazione

Il primo obiettivo del progetto era strutturare il file di configurazione. Essendo alla base di tutto il progetto, la maggior parte del tempo è stato utilizzato per decidere come organizzarlo. Il processo di creazione della struttura è stato un processo iterativo che è possibile dividere in tre fasi:

1. Inizialmente il file di configurazione permetteva la creazione di un sito o di un bot altamente personalizzato. Per esempio l'utente poteva decidere dove e come posizionare il grafico all'interno di una pagina o quali domande il bot poteva riconoscere. Ma un approccio del genere avrebbe reso il processo di creazione del file molto lungo e scomodo;
2. Quindi il file di configurazione è stato snellito eliminando tutte le informazioni sul 'come'. Per esempio, nella scrittura del file di configurazione all'utente non doveva interessare come il grafico veniva posizionato all'interno della pagina ma solo cosa il grafico mostrava. Il risultato finale di questa fase è un file che *si concentra sul contenuto informativo* della dashboard riducendo all'osso la personalizzazione del sito o del bot;
3. La struttura ottenuta è stata periodicamente modificata mano a mano che venivano a galla alcune limitazioni o problemi. Per esempio, alla fine della fase due, nel file di configurazione si facevano riferimenti diretti a un determinato device. Ma successivamente, grazie all'utilizzo dello ZDM, abbiamo potuto utilizzare i concetti di tag e di fleet che semplificano la scrittura del file.

2.1.1 La scelta del formato

Il formato del file di configurazione è fondamentale. Un formato sbagliato avrebbe reso il file poco leggibile. Dato che la leggibilità era il parametro più importante è stato scelto il formato YAML. Lo YAML, acronimo ricorsivo¹ di "YAML Ain't Markup Language", è un formato di serializzazione leggibile da esseri umani. Il formato YAML è semplice da leggere sia per le persone che per la maggior parte dei linguaggi di programmazione. Sul sito yaml.org, che è scritto anch'esso in YAML,

¹un acronimo ricorsivo è un acronimo che contiene se stesso all'interno della propria scrittura per esteso [12]

sono elencati tutti i linguaggi di programmazione che hanno moduli o librerie che consentono la lettura del formato.

Inoltre lo YAML è un *superset* del formato JSON². In altre parole, il formato YAML è un'estensione del formato JSON. Quindi un file JSON è sempre convertibile in un file YAML ma non sempre vale il viceversa [13].

Codice 2.1: informazione serializzata nel formato YAML

Director:

```
name: Spielberg
Movies:
  - Movie:
    title: Jaws ,
    year: 1982
  - Movie:
    title: Jurassic Park
    year: 1993
```

2.2 Struttura della configurazione 1.0

In questa sezione, al fine di descrivere e far comprendere le varie parti che compongono il file di configurazione, si simulerà una sua creazione partendo da un file vuoto. Il file di configurazione ottenuto rappresenterà un progetto. La simulazione è divisa idealmente in quattro parti:

1. Scrittura dei meta-dati del progetto;
2. Definizione dei dati che compongono la dashboard;
3. Definizione degli allarmi;
4. Definizione degli utenti.

Nell'appendice A è mostrato il file di configurazione ottenuto in questo esempio. Prima di procedere alla creazione del file di configurazione Iotu (l'esperto di IoT) deve aver creato un progetto sullo ZDM. Nell'esempio Iotu ha creato un workspace con id *wks-esempio* contenente due fleet rispettivamente con id *flt-esempio1* e *flt-esempio2*. Ciascuna delle fleet contiene cinque dispositivi con tag **termometro** e **temperatura interna** e un dispositivo con tag **termometro** e **temperatura esterna**. Tutti questi dispositivi restituiscono un payload con un unico valore identificato dalla parola **temp**. La seconda fleet ha un ulteriore dispositivo con tag **umidità** e restituisce un payload contenente un unico valore identificato dalla parola **humidity**.

2.2.1 Meta-dati del progetto

Inizialmente Iotu deve indicare alcuni meta-dati che servono ai vari generatori per ottenere i dati da FiBo o personalizzare il risultato finale.:

²Javascript Object Notation

- Il **nome del progetto** nel campo *projectname*. Iotu sceglie il nome **esempio**.
- Nel campo *workspace*, l'id del **workspace ZDM** che l'utente intende assegnare al progetto. Questo campo è necessario per far capire a FiBo dove andare a recuperare i dati.
- Nel campo *fleets*, la **lista delle fleet** contenute all'interno del workspace. Ogni elemento di questa lista è caratterizzato da due campi, il primo contiene un nome mentre il secondo contiene l'id assegnato alla fleet nello ZDM.
- Il nome del **template** (o modello) che il generatore di GUI deve utilizzare per creare la dashboard grafica. La lista dei nomi dipende dal generatore. Nell'esempio Iotu sceglie di utilizzare il modello base di nome **standard**.
- L'**entry point** ai servizi Amazon e la **voce** dell'assistente virtuale nei campi *entry point* e *voice* entrambi contenuti nel campo *Lex*.

Codice 2.2: il risultato finale di questa fase

```
projectname: esempio
template: Standard
workspace: wks-esempio
fleets:
  - name: Fleet1
    id: flt-esempio1
  - name: Fleet2
    id: flt-esempio2
lex:
  region: eu-west-1
  voice: salli
```

2.2.2 Viste e dati

Il prossimo campo che Iotu deve riempire si chiama *views* e contiene la lista di tutte le viste che caratterizzano la dashboard. Una vista può essere interpretata come un contenitore di dati che possono essere ottenuti a partire da una o più fleet. Ogni singola vista è caratterizzata da:

- un **nome**. Ogni vista deve avere un nome univoco;
- una **lista di id** delle fleet che si intende usare all'interno della vista;
- La **lista dei dati** contenuti nella vista;
- La **lista degli allarmi** associati alla vista.

Iotu vuole creare due viste, una relativa alla temperatura e una relativa all'umidità. Quindi chiama la prima vista "Temperatura" e le assegna tutte le fleet mentre chiama la seconda vista "Umidità" e le assegna solo la seconda fleet. Iotu decide di assegnare alla seconda vista solo una delle due fleet perché intende inserirvi i dati con il tag

umidità e solo la seconda fleet ha a disposizione un dispositivo che genera dati con tale tag.

Codice 2.3: la struttura delle views

```
#codice sezione precedente
views:
  - viewname: Temperatura
    enabledfleet:
      - flt-esempio1
      - flt-esempio2
    data:
      #to be continued in Codice 2.4
    alarm:
      #to be continued in Codice 2.6
  - viewname: Umidita'
    enabledfleet:
      - flt-esempio2
    data:
      #to be continued in Appendice A
    alarm:
      #to be continued in Appendice A
```

A questo punto Iotu deve scrivere i dati che vuole siano contenuti nelle due viste. Per ogni dato deve indicare:

- il **nome del dato** nel campo *title*. All'interno di una vista i nomi dei dati devono essere univoci;
- il **tipo di dato** nel campo *type*. I tipi di dato sono definiti nel campo *datatype* che verrà trattato nella sezione 2.2.5. Ogni tipo di dato è associato a un nome e a un'unità di misura. Nel campo *type* è necessario inserire solo il nome;
- le **time series³** che **compongono il dato** nel campo *timeSeries*. Questo campo è un array di coppie con nome *tagvalue* e *aggregation* ognuna delle quali rappresenta una time series. Ogni time series può essere ottenuta a partire da una o più fonti di dati. Ognuna di queste fonti va scritta nel campo *tagvalue* indicando la coppia (*tag,value*) che le identifica. Se una time series ha più fonti allora è necessario indicare come aggregare nel campo *aggregation* che deve contenere il nome della funzione di aggregazione che l'utente intende usare. Le funzioni di aggregazione supportate sono sum (somma), mean (media), min (minimo) e max (massimo).
- il **nome della funzione di aggregazione** da utilizzare per aggregare i dati temporalmente in base a due ulteriori campi che sono *timerange* e *granularity*. Timerange indica **il periodo di tempo, a partire da "ora", da cui ricavare i dati** mentre granularity definisce **la dimensione degli intervalli** su cui applicare la funzione di aggregazione. Sia timerange che granularity sono espressi con

³Una time series (in italiano serie storica) è una serie di osservazioni indicizzate (o elencate o rappresentate) in ordine temporale

una stringa che contiene un numero e l'unità di tempo associata al numero (es: 1 ora). Le funzioni di aggregazione supportate sono sum (somma), mean (media), min (minimo) e max (massimo).

- **il grafico** che deve mostrare le serie di dati. Ogni grafico è identificato con un nome e può possedere proprietà specifiche. Per maggiori informazioni consultare l'appendice C che riassume il risultato di una breve indagine che ho svolto a proposito di questo argomento.

Iotu definisce i dati che vuole inserire nella prima vista (Codice 2.4):

1. Il primo dato deve contenere la temperatura interna media nell'ultimo minuto e quindi viene chiamato *temperatura interna media ultimo minuto*. Il dato contiene una sola time series con un'unica fonte identificata dalla coppia **temperatura interna - temp**. Dato che la time series ha una sola fonte non è necessario compilare il campo aggregation. Iotu quindi definisce la **media** come funzione di aggregazione e **1 minuto** come timerange e granularity. Infine sceglie di utilizzare un **Gauge Chart** per la rappresentazione del dato.
2. Il secondo dato deve contenere la temperatura media di ogni ora dell'ultimo giorno e quindi viene chiamato *temperatura media ultime 24 ore*. Il dato contiene una sola time series con due fonti, la prima è identificata dalla coppia **temperatura interna - temp** mentre la seconda dalla coppia **temperatura esterna - temp**. Iotu decide di usare la media per aggregare i dati provenienti dalle due fonti e quindi scrive nel campo aggregation del dato la stringa **mean**. Successivamente definisce la **media** come funzione di aggregazione temporale, **1 giorno** come timerange e **1 ora** come granularity in modo da dividere l'ultimo giorno in 24 ore. Infine sceglie di utilizzare un **Line Chart** per la rappresentazione del dato.

A entrambi i dati viene assegnato il tipo **temperature**.

Dato che nel campo *enabledfleet* della vista sono presenti due fleet, in totale sono stati definiti quattro dati, due per la prima fleet e due per la seconda.

Codice 2.4: il campo data nella prima vista

```

- title: temperatura interna media ultimo minuto
  type: temperature
  timeseries:
    - aggregation:
        tagvalue:
          - tag: temperatura interna
            value : temp
      aggregationfunction: mean
      timerange: 1 minute
      granularity: 1 minute
      chart:
        type: gauge
        min: 0
        max: 40
  
```

```

- title: temperatura media ultime 24 ore
  type: temperature
  timeseries:
    - aggregation: mean
      tagvalue:
        - tag: temperatura interna
          value : temp
        - tag: temperatura esterna
          value : temp
  aggregationfunction: mean
  timerange: 1 day
  granularity: 1 hour
  chart:
    type: line

```

La seconda vista viene creata nello stesso modo.

2.2.3 Allarmi

Il file di configurazione permette di impostare degli allarmi in modo che un utente dell’interfaccia venga notificato quando un valore è fuori dalla norma. Ogni allarme è una quadrupla composta da:

- **Tag** e **Value** che identificano la fonte di dati da monitorare.
- **Min** il minimo valore sotto il quale il dato è fuori norma.
- **Max** il massimo valore sopra il quale il dato è fuori norma.

L’utente può aggiungere uno o più allarmi inserendoli nel campo *alarm* di una vista. Come è possibile osservare nel codice 2.5, Iotu inserisce nella prima vista due allarmi che controllano rispettivamente la temperatura interna e la temperatura esterna.

Codice 2.5: il campo allarmi relativo alla prima vista

```

alarm:
  - tag : temperatura interna
    value: temp
    max: 60
    min: 40
  - tag: temperatura esterna
    value: temp
    max: 30
    min: 0

```

2.2.4 Utenti

All’interno del sistema esistono due categorie di utenti:

- Il **viewer** che può accedere solo ai dati inviati da alcune fleet del progetto;

- L'**admin** che può accedere ai dati inviati da tutte le fleet del progetto.

Per definire gli utenti del sistema un utente deve compilare il campo users. Ogni elemento in users rappresenta un utente ed è costituito dai seguenti campi:

- **Mail**, e-mail dell'utente necessaria per autenticarsi;
- **Pass**, la password con cui autenticarsi;
- **Role**, la categoria di utente che può essere viewer o admin;
- **Fleets**, lista delle fleet a cui l'utente ha accesso. Per il ruolo di admin può essere omesso.

Iotu crea un utente admin, un utente che può accedere solo ai dati della prima fleet e infine un utente che può accedere solo ai dati della seconda come mostrato nel codice 2.6.

Codice 2.6: il campo users

```
users:
  - mail: admin@esempio.it
    pass: admin
    role: Admin
  - mail: pino@esempio.it
    pass: 12345678
    role: Viewer
    fleets:
      - flt-esempio1
  - mail: gino@esempio.it
    pass: 12345678
    role: Viewer
    fleets:
      - flt-esempio2
```

2.2.5 Tipi di dato

Alla fine del file di configurazione è presente un ultimo campo chiamato *datatype* che deve contenere tutti i tipi di dato indicati nei dati delle varie viste. Quest'ultimo campo consente di associare a ogni dato un'unità di misura. Nel codice 2.7 viene mostrato il campo datatype creato da Iotu in questo esempio.

Codice 2.7: il campo datatype

```
datatype:
  - name: humidity
    unitofmeasure: kg / m^3
  - name: temperature
    unitofmeasure: C
```

2.3 Generatore di configurazioni

E questo è tutto. Il file di configurazione non richiede altre informazioni rendendo la creazione di un progetto molto veloce anche scrivendo tutto a mano. Il generatore di configurazioni ha lo scopo principale di insegnare all'utente principiante come strutturare correttamente il file di configurazione mostrando graficamente la struttura e indicando il significato delle varie parti che la compongono. Inoltre rende il lavoro più facile anche all'esperto permettendo di riempire i campi con informazioni prese direttamente dal workspace ZDM dell'utente e quindi evitando errori come id di fleet, tag o value scritti erroneamente o che non esistono. L'attuale generatore di configurazioni mette a disposizione dell'utente un insieme di tipi di dato predefiniti che l'utente può assegnare al dato e che vengono automaticamente aggiunti nel campo *datatype*.

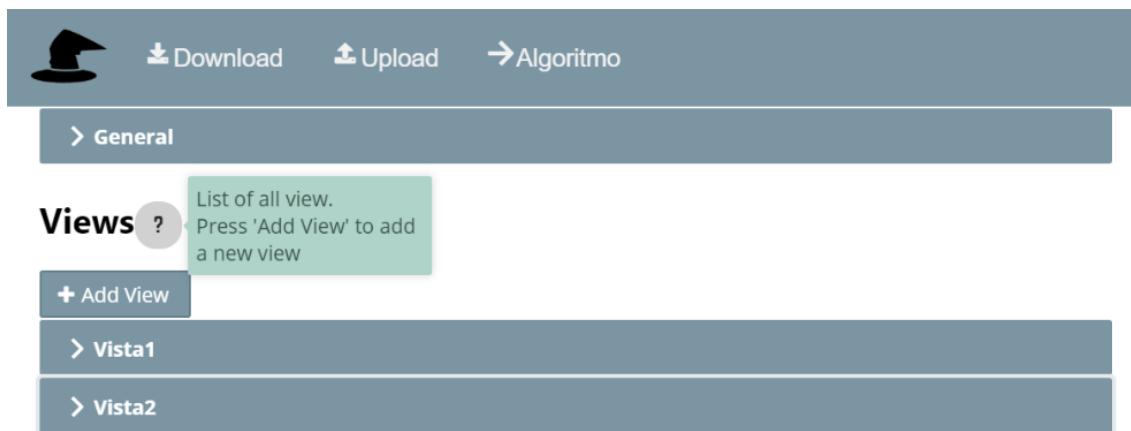


Figura 2.1: Generatore di configurazioni. Il generatore mostra la struttura e guida l'utente. Se l'utente non sa cosa fare può spostare il mouse sopra il punto interrogativo per avere qualche informazione in più.

CAPITOLO 3

Interfacce utente

Nel creare il file di configurazione era importante che ogni generatore avesse l’indispensabile per creare l’interfaccia finale. Questo capitolo descrive come gli elementi della configurazione sono utilizzati per creare e personalizzare l’interfaccia finale. I documenti [11] e [10] contengono maggiori informazioni su rispettivamente il generatore di interfacce vocali e il generatore di GUI.

3.1 login

Per accedere all’interfaccia (GUI o vocale) è necessario autenticarsi inserendo una email e una password fra quelle indicate nel file di configurazione. In figura 3.1 l’utente si autentica con l’account `pino@esempio.it` (riga 92 del file di configurazione contenuto nell’appendice A del testo). Ovviamamente avrebbe potuto accedere, conoscendo le credenziali, anche come `admin@esempio.it` o `gino@esempio.it`.



(a) login tramite interfaccia grafica

(b) login tramite interfaccia vocale

Figura 3.1: login dell’utente

3.2 Visualizzazione dei dati

3.2.1 Interfaccia grafica

All’interno della GUI una vista rappresenta un template (o modello) di una pagina web. Per ogni fleet associata ad una vista viene generata una pagina web diversa. Quindi, se una vista è associata a due fleet, allora viene creata una pagina per la prima fleet e una pagina per la seconda fleet, entrambe con la stessa struttura. Ogni pagina viene intitolata con il nome della vista che rappresenta. Per esempio, nella

figura 3.2, la pagina si intitola **Umidità**' perché così è stata chiamata la vista nel campo *viewname* alla riga 55 del file di configurazione. La pagina è costituita da un grafico per ogni dato contenuto nel campo *data* della vista (riga 58 del file di configurazione). Il titolo di ogni grafico è ottenuto leggendo il campo *title* del dato, mentre il tipo di grafico generato dipende dal campo *chart*. Il periodo di tempo mostrato e la distanza temporale tra i vari punti dipendono dai campi *timerange* e *granularity*.

L'utente che può accedere a più fleet deve, inizialmente, scegliere a quale fleet è interessato. Successivamente, come ogni utente, può accedere a una vista selezionandola dal menù laterale (o drawer in inglese).

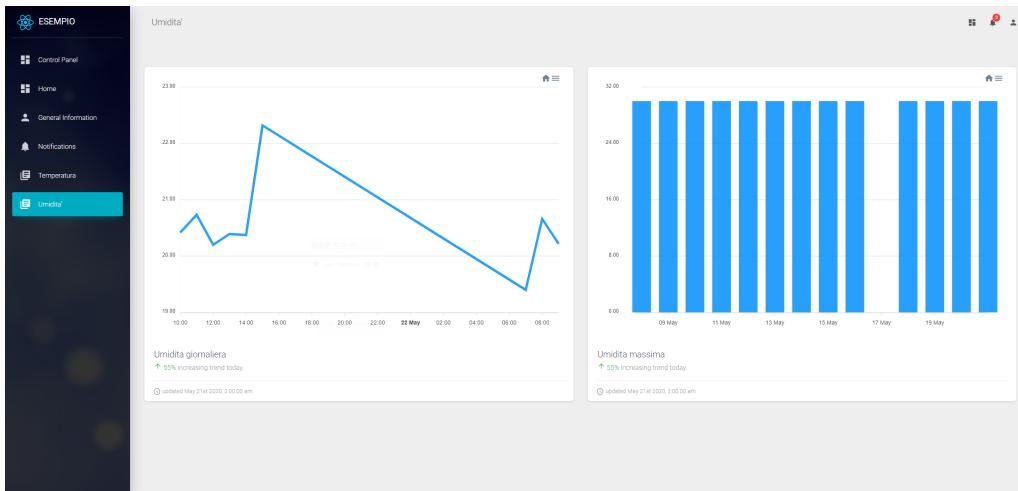


Figura 3.2: La pagina generata per la vista umidità.

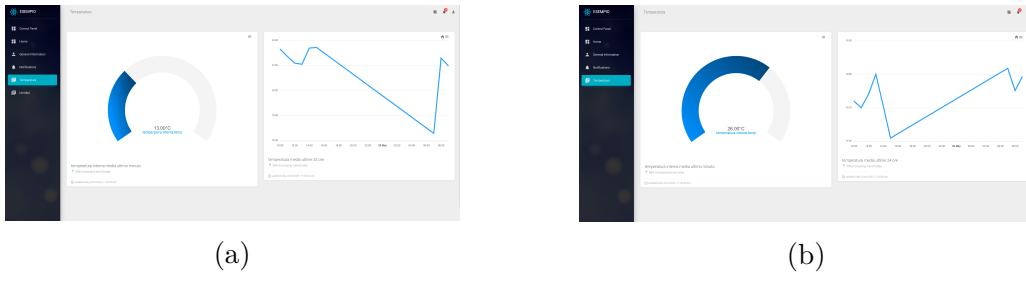


Figura 3.3: La vista Temperatura è associata a due fleet. Per cui il generatore crea due pagine con la stessa struttura. La figura 3.3a è relativa alla fleet con id `ftl-esempio2` mentre la figura 3.3b mostra i dati della fleet `ftl-esempio1`. Notate inoltre che la figura 3.3b ha un'opzione in meno nel menu laterale. Questo perché la figura 3.3a è visualizzata da un utente che si è autenticato come `pino@esempio.it` mentre la figura 3.3b da un utente che è acceduto come `gino@esempio.it`. Alla seconda vista (quella relativa all'umidità) può accedere solo chi ha accesso ai dati della fleet `ftl-esempio2`, quindi `pino@esempio.it` può accedervi mentre `gino@esempio.it` no perché può accedere esclusivamente ai dati della fleet `ftl-esempio1`.

Il titolo del progetto (riga 1 del file di configurazione) è mostrato in alto a sinistra nel menu laterale.

3.2.2 Interfaccia linguaggio naturale

L'utente che utilizza il chatbot ha la possibilità di richiedere i dati indicando le informazioni richieste, o con un'unica frase, o guidato dal bot (figura 3.4). Attualmente l'utente può richiedere dati con una sola time series costituita da una sola coppia di tag e di value. Quindi, riprendendo il file di configurazione di esempio, l'utente può richiedere la temperatura interna media o la temperatura esterna media, ma non può chiedere la media generale di entrambe come invece è indicato nel file di configurazione. Le informazioni che l'utente deve obbligatoriamente esplicitare sono:

- la **vista** in cui il dato è contenuto;
- il **tag** del dato;
- il **value** del dato;
- la **funzione di aggregazione** da utilizzare per aggregare il dato.

Ovviamente l'utente può indicare solo coppie (tag,value) contenute nei dati che compongono la vista che ha scelto. L'utente ha anche la possibilità di richiedere un periodo di tempo diverso rispetto a quello indicato nel file di configurazione indicando l'inizio e la fine del periodo a cui è interessato. Nel caso in cui l'informazione non sia specificata viene utilizzato il periodo scritto nel file di configurazione per il dato che contiene la coppia (tag,value) scelta.

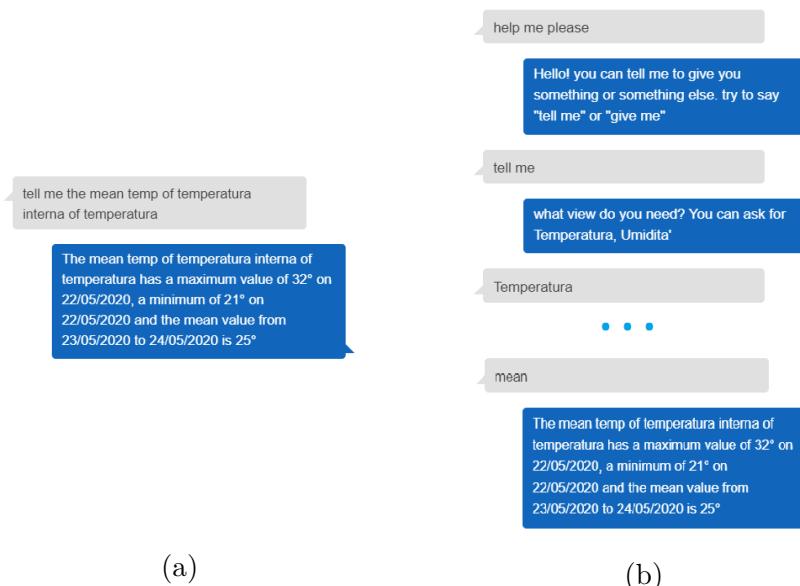


Figura 3.4: Interazione dell'utente con il chatbot.

Il chatbot si occupa anche dell'autorizzazione all'accesso dei dati. In figura 3.5 l'utente tenta di accedere a un dato a cui non può accedere.

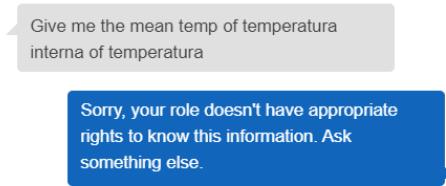


Figura 3.5: L’utente non può accedere al dato richiesto

3.3 Tecnologie utilizzate

3.3.1 Interfaccia grafica

L’interfaccia grafica è stata creata utilizzando React¹, una libreria javascript per la creazione di interfacce grafiche. Per strutturare alcuni elementi dell’interfaccia è stato utilizzato material-ui² mentre per generare i grafici è stato utilizzato il modulo apexCharts³.

3.3.2 Interfaccia linguaggio naturale

Il chatbot (o assistente virtuale) è stato realizzato utilizzando la tecnologia **Lex** creata da Amazon. Ogni chatbot Lex si basa su quattro elementi [14][11]:

- **intento:** l’operazione che l’utente vuole eseguire. Quindi in questo caso gli intenti potrebbero essere "io voglio autenticarmi" oppure "io voglio questo dato". Il bot del progetto è costituito da tre intenti che sono **Login** per l’autenticazione, **obtainInfo** per ottenere le informazioni e infine **Help** per assistenza nell’interazione;
- **utterances:** affermazioni di esempio che servono a istruire il modello di machine learning su come l’utente può interagire con il chatbot. Le utterances vengono generate automaticamente dal generatore;
- **slot:** i parametri di una richiesta. Per esempio, quando l’utente richiede i dati, gli slot sono la vista, il tag, il value, la funzione di aggregazione ed infine l’inizio e la fine di un periodo. Gli slot possono essere obbligatori o opzionali. Ogni slot è caratterizzato da un tipo di slot che viene utilizzato per addestrare il modello di machine learning a riconoscere i valori di uno slot;
- **funzione Lambda,** la funzione che viene chiamata quando il chatbot viene interpellato. Permette di personalizzare l’esperienza utente, di **convalidare l’input dell’utente** o **esaudire l’intento dell’utente**. In particolare quindi, nel chatbot generato, si occupa di autenticare o autorizzare l’utente o di validare l’input.

¹<https://it.reactjs.org/>

²<https://material-ui.com/>

³<https://apexcharts.com/>

CAPITOLO

4

FiBo for Filtering Border

Il capitolo inizia con una panoramica dei casi d'uso che FiBo attualmente implementa. Successivamente vengono mostrati i vari componenti che compongono il servizio evidenziando le tecnologie che sono state impiegate per la loro implementazione. In seguito viene descritto come sono stati implementati i casi d'uso e, infine viene mostrato il risultato di qualche test.

4.1 Funzionalità

Riprendendo il capitolo 1, FiBo è un microservizio che ha il compito di disaccoppiare le interfacce del progetto dallo ZDM. Il sistema deve servire due tipi di utente:

- **l'utente** delle interfacce generate a partire dal file di configurazione;
- **l'utente** che ha il compito di configurare il sistema. Questo utente viene chiamato **superuser** o super utente.

Entrambi gli utenti devono innanzitutto **autenticarsi**. L'utente standard autenticato può:

- **richiedere dei dati**;
- **sottoscriversi** a un canale per ricevere una notifica quando scatta un allarme.

Il super utente invece può:

- **creare un progetto** a partire da un file di configurazione. In altre parole, configurare il sistema per gestire il progetto descritto nel file di configurazione;
- **eliminare un progetto** che possiede.

Entrambi i tipi di utenti una volta autenticati possono eseguire il **logout**. Il diagramma in figura 4.1 schematizza quanto scritto in queste righe.

4.1.1 Architettura generale

Per poter garantire queste funzionalità FiBo è costituito da sette componenti che comunicano fra loro utilizzando **http**¹. Al momento quindi la comunicazione fra i vari componenti non è sicura. I componenti e le loro funzionalità sono:

¹http (Hypertext Transfer Protocol) è un protocollo per la trasmissione d'informazione sul web.

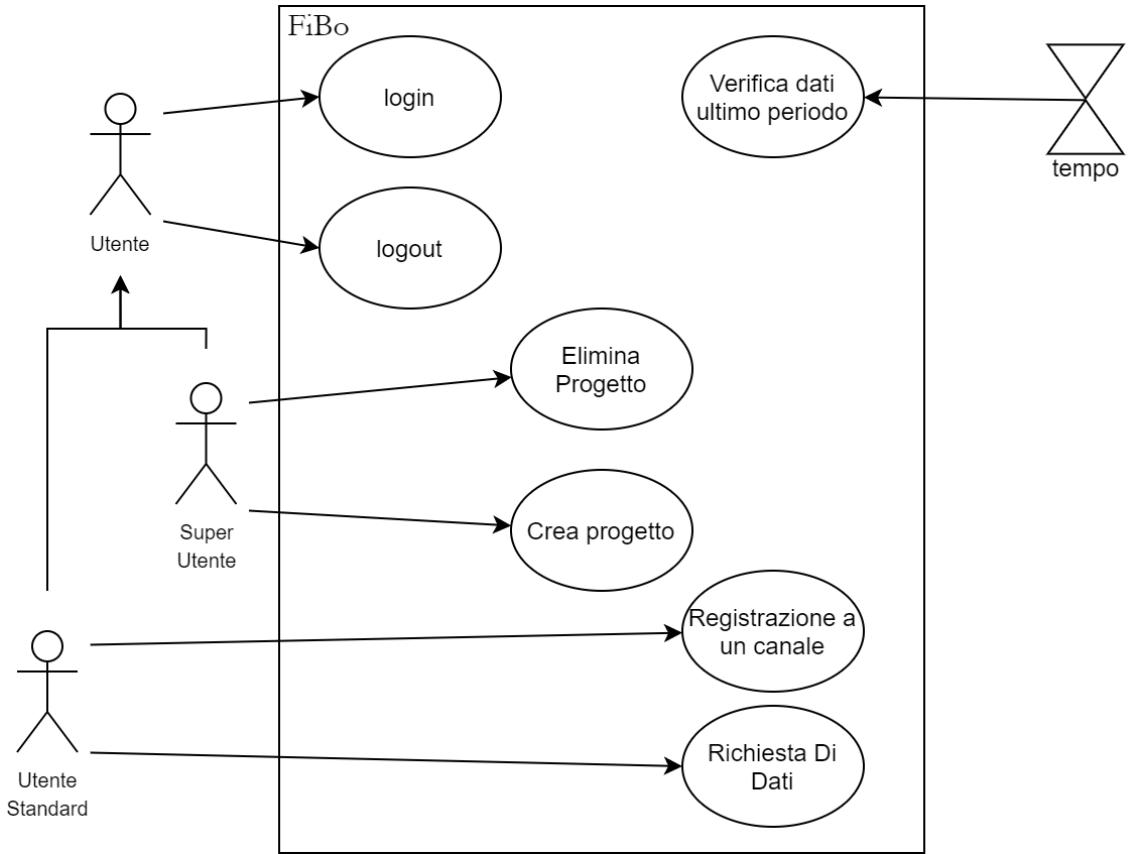


Figura 4.1: Casi d’uso di FiBo. Oltre ai due utenti esiste un terzo attore che svolge un compito fondamentale nell’analisi dei dati inviati dai dispositivi IoT per la generazione di allarmi. Il diagramma potrebbe contenere più casi d’uso ma per motivi di spazio ho preferito inserire solo i principali.

1. Lo **stub**. Questo componente ha il compito di gestire il login e il logout, svolgere le funzionalità di configurazione di un progetto e ricevere le richieste dei dati;
2. Il **calculator** viene utilizzato dallo stub quando un utente richiede dei dati. Raccoglie, su richiesta, i dati salvati nello ZDM e successivamente li aggrega in una o più metriche a seconda dei parametri della richiesta inviata dallo stub;
3. l'**alarm sender** avverte l’utente che ne ha fatto richiesta e che ne ha i diritti della registrazione di un allarme;
4. l'**alarm finder** si occupa di richiedere periodicamente (qui entra in gioco l’attore tempo in figura 4.1) i dati allo ZDM e di verificare che non siano anomali. Nel caso lo siano, genera un allarme;
5. Il **time series database** è un database in cui vengono memorizzati i dati delle time series. La memorizzazione è gestita dallo stub e dal calculator;
6. Le informazioni relative a un progetto, come allarmi o credenziali degli utenti o dei super utenti, vengono salvate in un secondo database chiamato **project**

database. Questi dati vengono utilizzati dallo stub, dal calculator, dall’alarm finder e dall’alarm sender;

7. L’alarm sender e l’alarm finder comunicano fra loro utilizzando un protocollo **publish/subscribe** implementato nel componente **pubsub**.

La figura 4.2 mostra come i vari componenti comunicano fra loro. L’utente, utilizzando un **client**, comunica con lo stub tramite il protocollo **http** mentre comunica con l’alarm sender servendosi di **socket.io**.

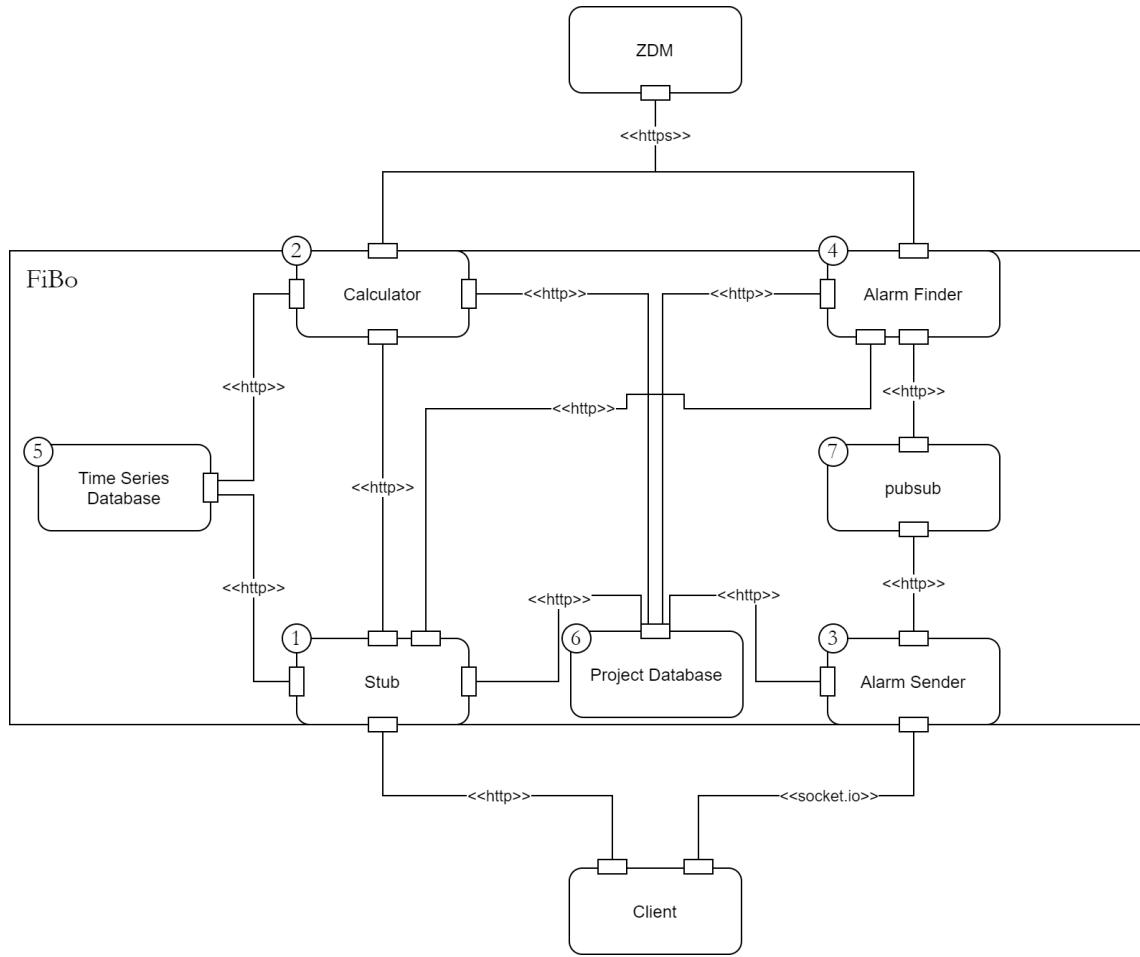


Figura 4.2: Come i vari componenti di FiBo comunicano. Il client utilizzato da un utente o da un super utente dovrebbe poter comunicare esclusivamente con lo stub e con l’alarm sender.

4.1.2 Tecnologie utilizzate

NodeJS

I primi quattro moduli sono stati sviluppati utilizzando **NodeJS**. Nodejs è una runtime Javascript costruita sul motore JavaScript V8 di Google Chrome. La scelta di NodeJS è principalmente dovuta ai seguenti motivi:

- **Semplicità.** Scrivere un programma in nodejs è molto semplice. Questo principalmente grazie ai migliaia di moduli che sono disponibili con licenza MIT su **npm**, il gestore di pacchetti di nodejs. In particolare, per questi quattro componenti è stato utilizzato il modulo **express** per gestire l'interfaccia http.
- La programmazione con Node.js è **event-driven**, ovvero il flusso del programma è determinato da eventi esterni, per cui l'accesso alle risorse del sistema operativo o le operazioni più lunghe possono essere gestite in modo asincrono consentendo al server di gestire con un singolo thread (Node.js ha un'architettura single thread) migliaia, se non milioni di connessioni contemporaneamente;
- Sia il generatore di interfacce grafiche che il generatore di configurazioni sono scritti in React che si basa su nodejs, per cui potrebbe essere possibile scambiare moduli tra questi tre componenti.

InfluxDB

Il time series database è un database InfluxDB. InfluxDB è un database noSQL specializzato nella memorizzazione di time series. Ogni elemento (**sample data**) all'interno di influxDB è caratterizzato da quattro proprietà [15]:

- **measurement:** concettualmente simile a una tabella dei database SQL, un measurement è un stringa che identifica un insieme di elementi. Ogni elemento è contenuto in un measurement;
- Il **tag set** è un insieme di coppie (tag key, tag value) che contengono i meta-dati. Sia le chiavi che i valori sono stringhe;
- il **field set** è un insieme di coppie (field key, field value) che contengono i dati. Le chiavi sono stringhe mentre i value possono essere stringhe, booleani, integer o float;
- **time** è il timestamp associato all'elemento.

In un database non possono esistere due elementi con lo stesso tag set e lo stesso timestamp, quindi se viene scritto un elemento con lo stesso tag set e lo stesso timestamp di un altro elemento già esistente, quest'ultimo viene sovrascritto. L'insieme degli elementi che hanno lo stesso tag set rappresenta una **time series**.

Nonostante sia un database noSQL, il linguaggio per fare query al database è molto simile al linguaggio SQL. I tag sono **indicizzati**, questo significa che le ricerche basate sui tag sono più veloci di quelle basate sui field.

InfluxDB permette di gestire abbastanza facilmente le **retention policies**. Una retention policy indica a InfluxDB quando eliminare un dato². Un database può contenere più retention policies. Se quando viene scritto un elemento non viene specificata una retention policy allora all'elemento viene associata la retention policy di default del database.

²Una retention policy serve anche per specificare **quante copie** del dato avere nei cluster. Questa proprietà non è stata considerata perché il clustering è una funzionalità che fa parte della versione a pagamento di InfluxDB (InfluxDB Enterprise)

time	project	fleet	value
2019-07-07T00:00:00Z	esempio	flt-esempio1	23
2019-07-07T01:00:00Z	esempio	flt-esempio1	12
2019-07-07T02:00:00Z	esempio	flt-esempio1	43
2019-07-07T00:00:00Z	esempio	flt-esempio2	21

Tabella 4.1: La tabella contiene quattro elementi. *Project* e *fleet* sono tag key mentre *value* è un field key. I primi tre elementi appartengono alla time series identificata dal tag set {(project : esempio),(fleet : flt-esempio1)} mentre l'ultimo elemento appartiene alla time series identificata dal tag set {(project : esempio), (fleet : flt-esempio2)}

PostgreSQL e Redis

Il project database è un database SQL molto diffuso chiamato **PostgreSQL**. Il componente pubsub invece utilizza **Redis**, un key-value database in-memory consigliato spesso su **StackOverflow** o **Medium** per la creazione di canali di comunicazione publish/subscribe.

4.2 Storage dei dati

4.2.1 Storage delle time series

Una time series in FiBo ha le stesse proprietà delle time series nel file di configurazione. Quindi ogni time series è caratterizzata da:

- Il **progetto** di appartenenza;
- La **fleet** da cui sono ottenuti i dati che la compongono;
- I **tag** e i **value** che identificano i dati di cui è composta;
- Le **funzioni di aggregazione** impiegate per aggregare i dati;
- La **granularità** (es: 1 ora, 2 minuti, ecc.), ovvero la distanza in tempo tra i suoi elementi.

Le time series vengono divise in due macro categorie:

- **Standard.** Le time series standard sono time series caratterizzate da una granularità con numero uguale ad uno e chiave uguale a minuto, ora o giorno (1 minuto, 1 ora o 1 giorno) e da un'unica coppia di tag e di value. Vengono utilizzate per le aggregazioni delle time series non standard.
- **Non Standard o Particular.** Le time series non standard sono tutte le altre time series che non soddisfano le proprietà del punto precedente. Le time series non standard (eccetto quelle in cui la granularità è espressa in secondi) sono ottenute aggregando time series standard.

All'interno di FiBo ogni time series inizia idealmente dalla mezzanotte (UTC) del primo gennaio 1970. In altre parole, ogni time series ha un inizio assoluto che corrisponde a quando il timestamp unix valeva 0. In questo modo i timestamp associati agli elementi di una time series sono assoluti e non differiscono fra due richieste che avvengono a pochi secondi di distanza.

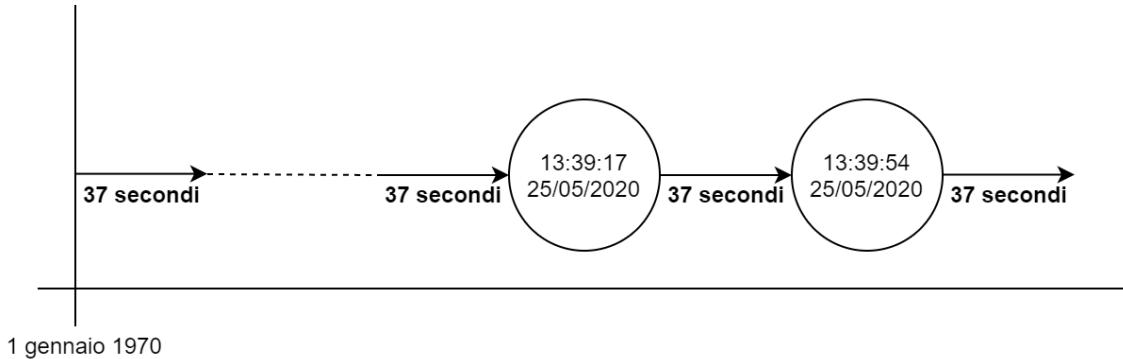


Figura 4.3: I timestamp dei dati della time series vengono calcolati a partire dal primo gennaio 1970. In questo caso la granularità della time series è 37 secondi.

Struttura dei dati

La categorizzazione delle time series si riflette nel database con due **measurement** (vedi sezione InfluxDB capitolo 4.1.2), uno chiamato **standard** che contiene tutte le time series standard e l'altro chiamato **particular** per le altre time series. Ogni elemento contenuto nel measurement particular ha il seguente *tag set*:

- **aggregationFunction**: funzione di aggregazione temporale;
- **internalAggregationFunction**: funzione di aggregazione applicata per aggregare le diverse fonti della time series;
- **fleet**: fleet di appartenenza;
- **projectName**: nome del progetto di appartenenza;
- **tag**: stringa che contiene tutti i tag concatenati in ordine alfabetico utilizzati per recuperare il dato;
- **unit**: stringa che contiene tutti i value concatenati in ordine alfabetico utilizzati per recuperare il dato;
- **granularityKey** e **granularityNumber**: contengono il numero e la chiave della granularità.

Gli elementi "standard" hanno un *tag set* identico eccetto per l'assenza di internalAggregationFunction e granularityNumber. In entrambi i measurement il *field set* è composto da due elementi:

- **value**: il valore associato al dato;

- **isValid:** un booleano che indica se l'elemento contiene un valore valido o meno. Un elemento è invalido se è relativo a un periodo di tempo in cui non sono stati trovati dati per la sua time series di appartenenza. Per esempio, in figura 3.2, si può notare che tra le 15 e le 7 la linea del primo grafico è continua. Questo avviene perché verso le 15 è stato spento il computer che si occupa della generazione dei dati di prova. In questo caso tutti gli elementi di quella time series in quel periodo di tempo sono marcati come invalidi (`isValid = false`).

Divisione dei dati

Il database è costituito da cinque diversi database, ognuno associato a una retention policy di default diversa. I database sono:

- **twelveHours** con una retention policy di **dodici ore** (significa che contiene esclusivamente elementi relativi alle ultime dodici ore). Nel database vengono memorizzati tutti gli elementi con granularità **in secondi**.
- **twentyFourHours** con una retention policy di **venticinque ore**. Contiene tutti gli elementi con granularità **in minuti**.
- **fiveWeek** con una retention policy di **cinque settimane** contiene gli elementi con granularità **in ore**.
- **fourYears** mantiene gli elementi con granularità **in giorni** per al massimo **quattro anni**.
- **longPeriods** con una retention policy senza limiti contiene tutti gli elementi con granularità **in settimane, mesi e anni**.

Quindi, maggiore è la granularità, maggiore è il periodo di tempo in cui gli elementi della time series possono essere salvati in memoria. Per esempio, in un periodo di un mese, possono essere salvate solo time series con granularità uguale o superiore a un'ora come 1 ora, 2 ore, 12 ore, 1 giorno e così via.

4.2.2 Storage degli altri dati

Tutti i dati relativi a un progetto come lista degli allarmi o credenziali degli utenti vengono memorizzati in un database PostgreSQL. Non è stato perso molto tempo per strutturare questo secondo database quindi lo schema è sicuramente migliorabile. In particolare ci sono diverse ridondanze che potrebbero essere eliminate in futuro. Lo schema riflette in buona parte la struttura del file di configurazione. In figura B.1, è mostrato lo schema concettuale del database, mentre in figura B.2, il relativo schema logico. La soluzione cerca in primis di rendere più veloci le operazioni più richieste che riguardano in particolare la lettura delle credenziali di un utente e delle fleet a cui ha accesso, la lettura degli allarmi associati a una fleet e la lettura della lista dei token in blacklist (vedi logout).

4.3 Login

4.3.1 Interfaccia http

Il client che intende autenticarsi come utente deve inviare all'endpoint **/login** dello **stub**, nel body di una richiesta POST, un oggetto JSON contenente username, password e progetto di appartenenza (vedi figura 4.1).

Codice 4.1: richiesta di login

```
{
  "username": "admin@esempio.it",
  "password": "admin",
  "projectName": "esempio"
}
```

Per autenticarsi come super utente la richiesta è la stessa ma l'endpoint è **/login/-superuser**. Se la procedura ha successo, il client deve inserire il JWT ottenuto, nel campo **Authorization** dell'header delle richieste successive.

4.3.2 Implementazione

Il processo di login è molto semplice e si basa sull'invio di un JWT (Json Web Token) al client che si è autenticato con successo. Questo JWT permette al client di accedere per un certo periodo di tempo alle funzionalità offerte da FiBo. Ogni JWT è costituito da tre parti [16]:

- L'**Header** che indica il tipo di token e l'algoritmo utilizzato per criptare la signature (in FiBo è **HS256**). L'header è criptato in **Base64url**.
- Il **payload** che contiene informazioni specifiche sull'utente che si è autenticato o sul token. Un payload è composto da **claims**. I claims possono essere pre-definiti (registered), pubblici o privati. I claims privati che FiBo memorizza nel payload sono l'username, l'id dell'utente (id della tabella users in figura B.1), il progetto di appartenenza e la lista delle fleet a cui l'utente può avere accesso. Il payload è criptato in **Base64url**.
- Una **signature** per validare il token. Questa firma è ottenuta concatenando con un punto come separatore i valori criptati in **Base64url** dell'header e del payload. La stringa ottenuta viene nuovamente criptata impiegando l'algoritmo indicato nell'header utilizzando come chiave crittografica la chiave privata del server. In FiBo la chiave privata è memorizzata nella variabile d'ambiente `JWT_KEY` per i login degli utenti, e nella variabile d'ambiente `JWT_KEY_SUPERUSER` per i login dei super utenti.

Queste tre parti sono infine concatenate con un punto come separatore³. Ogni JWT può avere una data di scadenza dopo la quale non ha più valore. Questa data viene memorizzata nel claim predefinito **exp** del payload. Il tempo di validità di un token

³Le tre parti del JWT sono divise da un punto: eyJhJ9.xNDc0fQ.DKGfngI

è determinato dalla variabile d'ambiente EXPIRATION_TIME ed è di ventiquattro ore.

Per l'autenticazione, lo stub cerca nel project database l'username inviato dal client. Se l'username viene trovato, allora la password inviata dal client viene confrontata con la password trovata nel database. Per motivi di sicurezza, nel database viene memorizzato l'hash della password concatenata con otto bit casuali⁴ e non la password in chiaro. L'hash viene calcolato utilizzando il pacchetto npm **bcrypt**. Anche il confronto fra le password avviene servendosi di un'altra funzione di questo pacchetto. Se i due valori coincidono l'autenticazione ha successo, per cui il JWT viene generato usando il pacchetto npm **jsonwebtoken** e successivamente inviato al client. Il processo è schematizzato in figura 4.4.

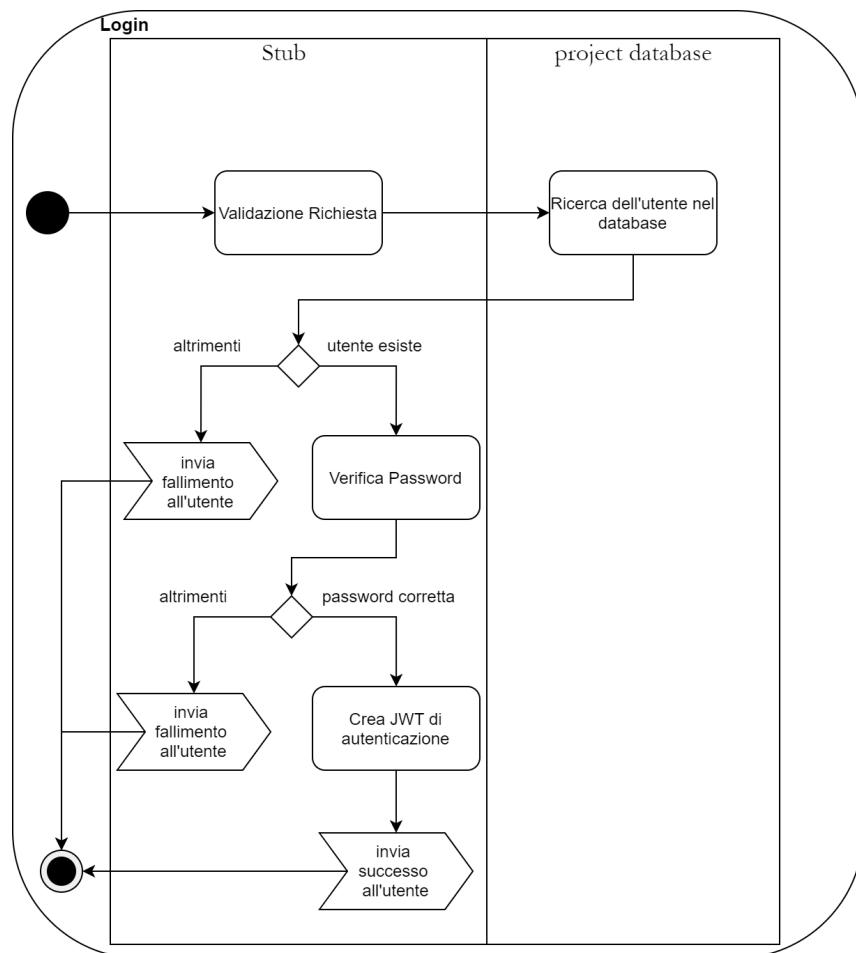


Figura 4.4: Diagramma di attività login.

La lista delle fleet memorizzate nel payload del token vengono utilizzate per gestire l'accesso ai dati delle fleet. Un client può accedere ai dati di una determinata fleet solo se l'id di tale fleet è contenuto nel JWT inviato con la richiesta.

⁴Questa tecnica è chiamata salting e ha l'obiettivo di evitare che venga memorizzato lo stesso hash per due password identiche.

4.4 Logout

4.4.1 Interfaccia http

Il client che intende eseguire il logout, deve inviare una richiesta POST all'endpoint **logout** dello **stub**, inserendo nel campo Authorization dell'header della richiesta un JWT valido.

4.4.2 Implementazione

Un JWT perde validità solo quando scade, per cui il JWT inviato dal Client viene inserito in una black list contenuta all'interno del project database (tabella tokensblacklist). I token inseriti nella black list mantengono la loro validità ma non permettono più l'accesso alle funzionalità di FiBo. Periodicamente, viene eseguito uno script che elimina i token scaduti contenuti nella black list.

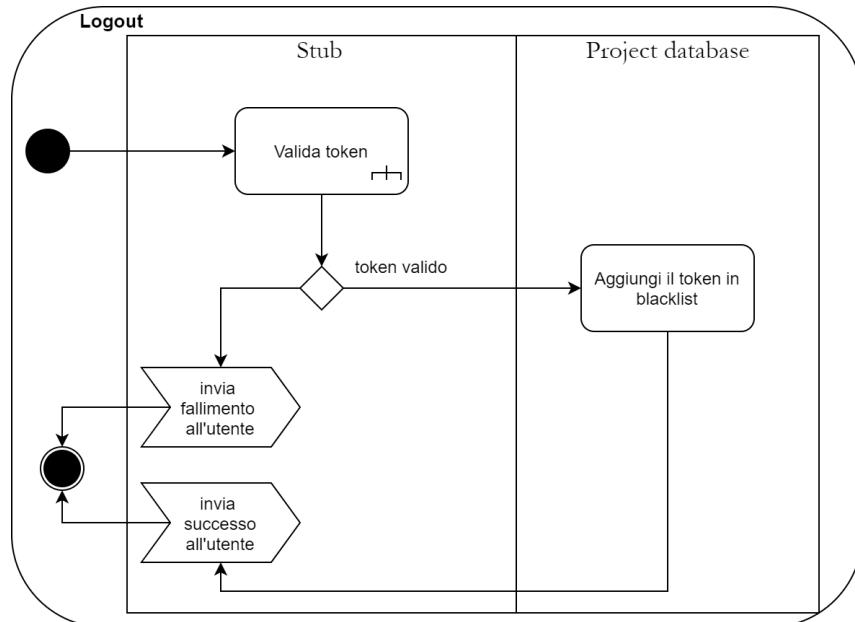


Figura 4.5: Diagramma di attività logout.

4.5 Validazione del JWT

Tutti gli endpoint verso l'esterno, eccetto il login e il logout, sono "sorvegliati" da un middleware che si occupa di autenticare e decriptare il JWT inviato dal Client servendosi del pacchetto npm **jswowebtoken**. Affinché una richiesta non venga bloccata, il JWT inviato dal client deve essere valido, non deve essere nella black list (tabella tokensblacklist) e deve contenere informazioni valide (l'id salvato nel JWT viene utilizzato per cercare l'utente nel database. Nel caso non venga trovata una corrispondenza la richiesta viene bloccata).

4.6 Richiesta di dati

4.6.1 Interfaccia http

Il client che intende richiedere dei dati di una time series deve inviare una richiesta POST all'endpoint `/get` dello **stub**. La richiesta deve contenere nel body un oggetto JSON come quello mostrato nel codice 4.2 (ci possono essere alcune variazioni).

Codice 4.2: esempio di richiesta di dati

```
{
  "projectName": "esempio",
  "timeSeries": [
    [
      {
        "tag": "termometro",
        "value": "temp"
      },
      {
        "tag": "umidita",
        "value": "humidity"
      }
    ],
    "fleet": "flt-esempio2",
    "aggregationFunction": "sum",
    "agrFunPerGroup": [ "sum" ],
    "timeRange": {
      "key": "day",
      "number": 1
    },
    "unit": "s",
    "granularity": {
      "key": "hour",
      "number": 1
    },
    "store": "true"
  }
}
```

La maggior parte dei campi come `timeSeries` o `projectName` sono abbastanza chiari già dal loro nome riferendosi ad elementi contenuti nel file di configurazione. Il campo `store` serve per far sì che le time series richieste vengano in ogni caso memorizzate nel time series database. Il periodo di tempo può essere specificato in due modi:

- indicando un **time range** come nel codice 4.2 dove si richiedono tutti i dati da un giorno fa fino ad adesso;
- indicando un **timestamp d'inizio periodo** e opzionalmente un **timestamp di fine periodo** nei campi `start` ed `end`. In questo caso il client deve anche specificare l'**unità di misura** del timestamp inserendola nel campo `unit`. Per esem-

pio, se i timestamp di start ed end sono in secondi, allora unit deve essere posto uguale ad s che sta per secondo. Le altre unità sono nell'ordine M per millisecondo, m per microsecondo e infine n per nanosecondo.

Se vengono inseriti entrambi i campi, timeRange non viene considerato. Il campo *aggrFunPerGroup*, corrispondente del campo *aggregation* nel file di configurazione, contiene i nomi delle funzioni di aggregazione che si intende utilizzare per aggregare le fonti delle time series. L'i-esimo elemento si riferisce all'i-esima time series.

Granularità minima

La granularità minima è **la minima unità di tempo** (secondo < minuto < ora < ecc.) utilizzabile come campo **key** della granularità. La distanza temporale tra l'inizio del periodo specificato nella richiesta e il momento in cui viene effettuata la richiesta è direttamente proporzionale al valore della granularità minima. Per esempio, se l'utente richiede i dati dell'ultimo giorno, la granularità minima è il minuto, mentre se richiede i dati dell'ultimo anno la granularità minima è il giorno. Questo significa che se nella richiesta l'utente ricerca dei dati dell'ultimo anno ogni quarantanove ore, dato che la granularità minima è il giorno, la granularità effettivamente utilizzata sarà due giorni.

La granularità minima determina anche quale time series standard utilizzare per aggregare una time series non standard. Tornando all'ultimo esempio, se la granularità della time series richiesta è due giorni, questa viene calcolata aggregando due a due gli elementi della time series standard con granularità un giorno.

Il calcolo della granularità minima si basa sulle retention policies dei cinque database del time series database.

4.6.2 Implementazione

Fase I

Quando lo stub riceve una richiesta, la **valida** e successivamente **determina il periodo in cui cercare i dati**. L'inizio del periodo non coincide necessariamente con quello richiesto dall'utente. Per esempio, se l'utente desidera una time series con una granularità di venticinque minuti a partire dalle 12:53:20 del 27 maggio fino alle 12:29:59 del 28 maggio, il periodo effettivamente considerato va dalle 12:45:00 del 27 maggio alle 12:29:59 del 28 maggio. Questo è dovuto al fatto che le time series hanno idealmente inizio il primo gennaio 1970, per cui l'inizio del periodo da considerare viene calcolato cercando il punto della time series con il massimo timestamp minore o uguale al momento indicato nella richiesta. Sempre in questa fase viene calcolata la **granularità minima** e convertita se necessario la granularità specificata dall'utente. Tutte queste operazioni vengono eseguite dalla funzione **definePeriods** nel file **time.js**.

Fase II

Lo stub **ricerca i dati** richiesti nel time series database. Nel caso vengano trovati tutti, l'elaborazione della richiesta termina e al client viene inviata la time series.

Altrimenti lo stub **contatta il calculator** all'endpoint **aggregate** inviando un POST contenente un oggetto JSON come quello nel codice 4.3. Alcuni campi coincidono con quelli della richiesta utente mentre altri sono leggermente diversi. Specificamente, periods contiene tutti i periodi in cui non sono stati trovati elementi per le time series indicate nel campo timeSeries mentre roundfactor è la granularità minima.

Codice 4.3: richiesta dello stub al calculator.

```
{
    "aggrFun": "sum",
    "projectName": "esempio",
    "timeSeries": [
        [
            {
                "tag": "termometro",
                "value": "temp"
            },
            {
                "tag": "umidita",
                "value": "humidity"
            }
        ],
        "fleet": "flt-esempio2",
        "aggrFunPerGroup": [ "sum" ],
        "periods": [
            { "start": 1590645600, "end": 1590717599 },
            { "start": 1590721200, "end": 1590731999 }
        ],
        "granularity": { "number": 1, "key": "hour" },
        "roundFactor": "minute"
    ]
}
```

Fase III

Il calculator **valida** la richiesta dello stub e successivamente **ricerca** nel time series database le time series standard che compongono le time series richieste dallo stub. Riprendendo il codice 4.3, la time series richiesta dallo stub è composta da due time series standard, una per ogni coppia (tag,value) diversa. La ricerca riguarda le time series standard con granularità pari alla granularità minima e con funzione di aggregazione il nome indicato nel campo *aggrFun* della richiesta. Successivamente il calculator controlla la risposta del database in modo da trovare periodi con dati mancanti i quali vengono richiesti allo ZDM utilizzando il metodo **getData** della classe **IoTData**. L'endpoint per richiedere i dati allo ZDM è <https://api.zdm.zerynth.com/v1/tsmanager/workspace/wks-esempio> dove viene anche specificato il workspace da cui richiedere i dati. Successivamente, nella parte di query dell'URL, viene richiesto che i dati vengano inviati dal più recente al più vecchio, scrivendo `?sort=-timestamp_device`, e che vengano restituiti

tutti i dati possibili con `&size=-1`. Infine è possibile filtrare i dati per tag, per fleet e per timestamp. Se, elaborando la richiesta 4.3, il primo periodo in *periods* non contenesse alcun dato per nessuna delle due time series standard, allora quest'ultima parte della query sarebbe: `&tag=termometro&tag=umidita&fleet=flt-esempio2&start=2020-05-28T06:00:00.000Z&end=2020-05-29T01:59:59.000Z`. Nel codice si è cercato di ridurre le chiamate allo ZDM inserendo nella stessa query tag relativi a time series diverse. Per ottenere i dati di un utente dello ZDM è necessario essere in possesso di un token relativo a quell'utente. Il token, come le credenziali dello ZDM utilizzate per aggiornarlo quando scade, sono memorizzate all'interno del project database. Queste informazioni sono criptate e decriptate utilizzando l'algoritmo AES256 implementato dal pacchetto npm **crypto-js**. Crypto-js si occupa automaticamente di generare un vettore di inizializzazione per evitare che due informazioni identiche vengano memorizzate con lo stesso valore. Attualmente le chiavi private utilizzate sono salvate nelle variabili d'ambiente che iniziano con ENCRYPT_KEY.

Fase IV

Il calculator inizia infine la fase di aggregazione dei dati. Dato che l'aggregazione può essere un processo molto lungo, al fine di non bloccare il processo su una singola richiesta, l'operazione viene svolta da un thread di un thread pool dinamico realizzato usando il pacchetto npm **poolifier**. Il thread applica la funzione specificata nel campo *aggrFun* sui nuovi dati raccolti dividendoli in periodi con granularità `{key: granularità minima, number: 1}` generando quindi i nuovi elementi delle time series standard. I nuovi elementi vengono uniti a quelli precedentemente letti dal time series database. Il risultato dell'unione viene nuovamente aggregato e diviso in periodi con la granularità indicata nella richiesta dello stub. Le nuove time series, sempre relative a una sola fonte di dati, vengono a seconda del caso aggregate fra loro utilizzando le funzioni specificate nel campo *aggrFunPerGroup* della richiesta per ottenere le time series costituite da più fonti come quella nel codice 4.3. Ogni aggregazione produce due time series, una per gli elementi validi e l'altra per gli elementi invalidi come mostrato in figura 4.6

Fase V

Una volta che il thread ha terminato, i nuovi elementi calcolati (validi o non validi) delle time series standard vengono memorizzati nel database⁵ e contemporaneamente allo stub vengono inviate le time series che aveva richiesto.

Una volta ricevuto il risultato, lo stub unisce i nuovi elementi con gli elementi precedentemente raccolti dal database. Successivamente calcola il **nextRequestTime**. Questo parametro indica al client quanti secondi attendere per richiedere la stessa time series e ottenere nuovi elementi. Lo scopo è ridurre il numero di volte in cui il client richiede la stessa time series al server. In seguito, invia al client un oggetto JSON come quello nel codice 4.4, in cui sono contenuti gli elementi validi delle time series richieste. Alla fine, se store nella richiesta utente è **true**, salva nel database i nuovi elementi delle time series non standard.

⁵se standard, viene memorizzata anche la time series richiesta dallo stub.

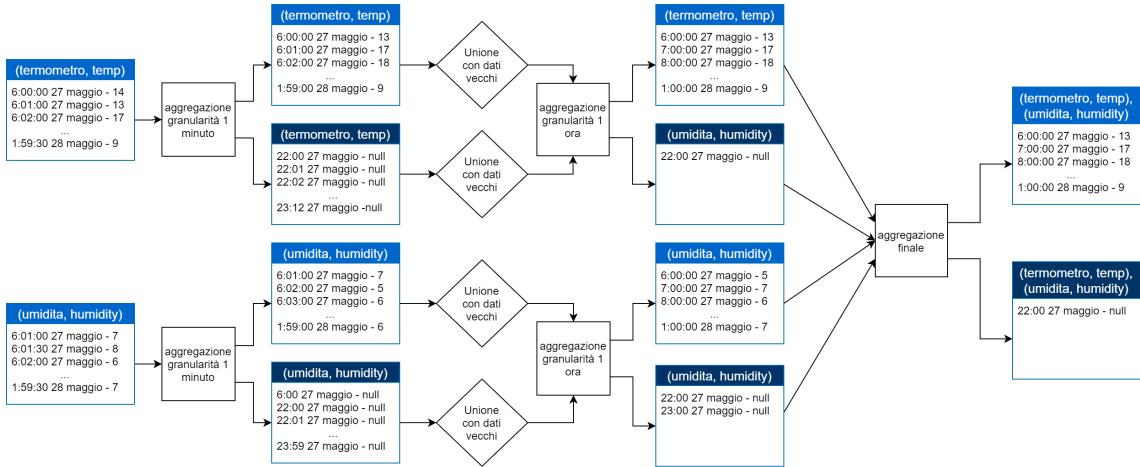


Figura 4.6: Esempio di aggregazione della richiesta 4.3 ipotizzando che il database non contenga le time series standard necessarie. In blu chiaro le time series valide, in blu scuro quelle non valide. In questo caso, dalle 22 alle 23:12:59 la ZDM non ha ricevuto alcun dato identificato dalla coppia (termometro,temp) mentre per i dati identificati da (umidita, humidity) il buco è tra le 22 e le 23:59:59. Aggregando poi i minuti in ore, la mancanza di dati per la prima coppia si traduce in una sola ora invalida in quanto l'ora dalle 23:00 alle 23:59:59 è costituita solo parzialmente da dati invalidi. Per la seconda coppia vengono invece prodotti due elementi invalidi perché i dati delle due ore fra le 22:00 e le 23:59:59 sono completamente invalidi. L'aggregazione finale delle due time series produce la time series richiesta dallo stub.

Codice 4.4: risposta di FiBo in seguito alla richiesta 4.2.

```
{
  "status": 200,
  "granularity": { "number": 1, "key": "hour" },
  "start": "2020-05-28T06:00:00.000Z",
  "end": "2020-05-29T05:59:59.000Z",
  "nextRequestTime": 2989,
  "result": [
    {
      "tags": [ "termometro", "umidita" ],
      "values": [ "temp", "humidity" ],
      "timeSeries": [
        { "time": 1590645600, "value": 9 },
        { "time": 1590649200, "value": 9 },
        { "time": 1590652800, "value": 9 },
        { "time": 1590656400, "value": 9 },
        { "time": 1590660000, "value": 9 },
        { "time": 1590717600, "value": 9 },
        { "time": 1590724800, "value": 9 },
        { "time": 1590728400, "value": 9 }
      ]
    }
  ]
}
```

{}

4.7 Creazione ed eliminazione di un progetto

Lo strumento utilizzato dal super utente per creare o eliminare un progetto è la CLI descritta nel prossimo capitolo. L'endpoint dello **stub** per la creazione di un progetto è **/config/createproject**. L'oggetto JSON, inviato nel body di una richiesta POST, contiene varie informazioni sul file di configurazione come le credenziali degli utenti o i dati e gli allarmi che fanno parte del progetto. L'inserimento di tutte queste informazioni avviene all'interno di un'unica transazione in modo da evitare che in caso di errore alcune informazioni rimangano salvate nel database. Una volta terminata la procedura di inserimento, lo stub avverte l'alarm finder della creazione di un nuovo progetto in modo che inizi a monitorare i nuovi dati. L'endpoint offre la possibilità di sovrascrivere un progetto già esistente. In questo caso, se il superutente possiede il progetto, questo viene eliminato e l'alarm finder informato della sua eliminazione.

L'eliminazione di un progetto avviene inviando un POST contenente un oggetto JSON all'endpoint **/config/deleteproject** dello **stub**. Nell'oggetto inviato è indicato il nome del progetto che il super utente intende eliminare. L'eliminazione avviene esclusivamente se il super utente possiede il progetto. Dopo che il progetto è stato eliminato, lo stub informa l'alarm finder dell'eliminazione in modo che non vengano più monitorati i dati del progetto rimosso.

4.8 Generazione degli allarmi

La generazione degli allarmi è l'ultima funzionalità implementata . L'alarm finder, che implementa questa funzionalità, inizialmente legge dal database tutti gli allarmi di tutti i progetti e li inserisce in una struttura dati in-memory chiamata **Alarm-Container**. Una volta terminata questa fase, viene creata una funzione per ogni coppia (fleet, progetto). Ognuna di queste funzioni controlla i dati inviati dai device della coppia (fleet, progetto) a cui è stata assegnata. I dati non vengono richiesti continuamente, ma ogni quanto di tempo. Questo tempo è gestito da un oggetto della classe **DelayCoordinator**. Il delayCoordinator indica alle funzioni quanto attendere per la prossima richiesta e fa in modo che le richieste siano dilazionate nel tempo. Le funzioni continuano ad essere eseguite finché la coppia (fleet, progetto) a cui sono associate è contenuta nell'alarmContainer. Anche questo componente utilizza la classe **IoTData** per interfacciarsi con lo ZDM.

Quando l'alarm finder è informato dallo stub della creazione di un nuovo progetto, cerca i nuovi dati nel project database, poi elimina, se esistono, gli elementi dell'alarmContainer associati a un eventuale vecchio progetto con lo stesso nome del nuovo progetto ed infine aggiunge i nuovi allarmi alla struttura dati. L'eliminazione degli allarmi invece consiste nell'eliminazione di elementi dall'alarmContainer.

Gli allarmi relativi a una determinata fleet e a un determinato progetto vengono pubblicati su un canale del componente pubsub con nome, la concatenazione del

nome del progetto e dell'id della fleet separati da un segno meno. Il codice 4.5 contiene un esempio di dato pubblicato sul canale *esempio-flt-esempio1* in seguito alla lettura di un dato inviato dal device con id *dev-esempio* appartenente alla flotta *flt-esempio1* che con un valore di 30 (*faultValue*) è ben al di sotto del limite minimo di 100 (*threshold*).

Codice 4.5: dato generato in seguito a un allarme.

```
{
  "tag" : "termometro",
  "value" : "temp",
  "device" : "dev-esempio",
  "fleet" : "flt-esempio1",
  "timestamp" : 1590649243,
  "threshold" : 100,
  "faultValue" : 30,
  "type" : "min"
}
```

4.9 Notifica degli allarmi

4.9.1 Interfaccia socket.io

Il protocollo di comunicazione utilizzato per far comunicare il client e l'alarm sender (il componente che si occupa della notifica degli allarmi) è socket.io. Socket.io consente di creare canali di comunicazione real-time, bidirezionali e event-based tra client e server [17]. Inizialmente socket-io crea una connessione **long-polling** per poi passare se possibile a un protocollo migliore come **websocket** [18]. La comunicazione tra client e server si basa su eventi attraverso cui è possibile inviare diversi tipi di dato che spaziano dal JSON al binario. Ogni canale di comunicazione tra un client e il server rappresenta un socket.

In fase di connessione il client deve inviare il JWT ottenuto in seguito alla procedura di autenticazione con lo stub. Il JWT viene validato da un middleware (vedi sezione 4.5) e se non è valido il server emette l'evento **error** e chiude la connessione.

Per ricevere gli allarmi relativi ad una fleet il client deve emettere l'evento **addFleet** attraverso cui invia all'alarm sender l'id della fleet di interesse e il progetto di appartenenza. La registrazione avviene esclusivamente se il client ha i diritti per accedere alla fleet. Per non ricevere più notifiche una volta registrato, il client deve emettere l'evento **rmFleet** in cui invia gli stessi dati che ha usato per registrarsi.

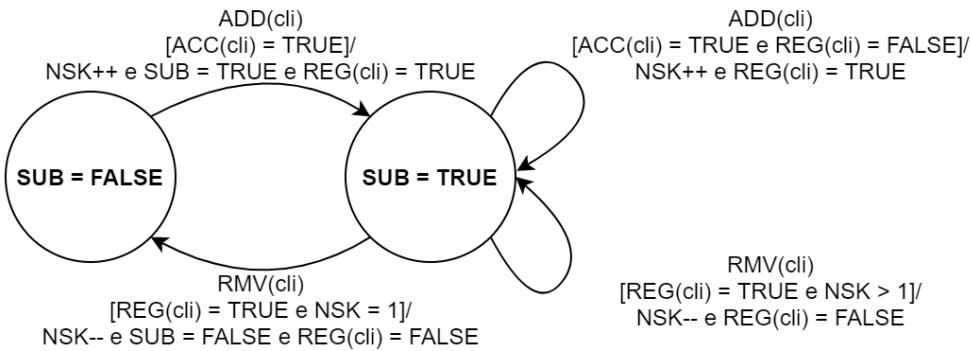
4.9.2 Implementazione

Socket.io consente di definire delle **room**. Ogni socket può unirsi e successivamente abbandonare una room. Il server può inviare messaggi ai socket registrati a una specifica room.

Quando un client emette l'evento **addFleet**, il suo socket viene aggiunto a una room con lo stesso nome del canale su cui vengono pubblicati gli allarmi per la fleet

e il progetto indicati con l'evento. Successivamente il server si registra a tale canale se non è ancora registrato. In questo modo, quando l'alarm finder trova un valore anomalo, il server viene informato e notifica tutti i socket registrati alla room con lo stesso nome del canale da cui è arrivato l'allarme. La notifica avviene emettendo l'evento **alarm**. Il dato che viene inviato ai client è lo stesso che l'alarm finder pubblica sul canale.

L'evento **rmFleet** provoca invece la rimozione del socket del client che lo ha emesso, dalla room relativa alla fleet e al progetto indicati con l'evento. Inoltre, il server si disiscrive dal canale del componente pubsub relativo a tale fleet e a tale progetto se la room, in seguito alla rimozione, rimane vuota.



NSK = **Numero di socket** registrati alla room relativa alla fleet **flt** e al progetto **prj**.
 SUB = **TRUE** se il server è sottoscritto al canale del componente pubsub relativo alla fleet **flt** e al progetto **prj**.
 ACC(**cli**) = **TRUE** se il client **cli** ha accesso al canale per i dati identificati dalla coppia (**flt,prj**).
 REG(**cli**) = **TRUE** se il client **cli** è registrato al canale per i dati identificati dalla coppia (**flt,prj**).
 ADD(**cli**) = il client **cli** emette l'evento addFleet per registrarsi alla room relativa alla fleet **flt** e al progetto **prj**.
 RMV(**cli**) = il client **cli** emette l'evento rmFleet per uscire dalla room relativa alla fleet **flt** e al progetto **prj**.

Figura 4.7: Macchina a stati per la gestione di una room dell'alarm sender.

4.10 Organizzazione del codice

La directory **database** contiene tutto ciò che è relativo alla comunicazione con il time series database e con il project database. Nel dettaglio:

- in `postgres.js` è implementata la classe **PostgresDB** che offre metodi per interfacciarsi con PostgreSQL utilizzando il pacchetto npm **pg**;
- in `influxdb.js` è implementata la classe **InfluxDB** che permette di fare query a InfluxDB servendosi del pacchetto npm **influxdb-nodejs**;
- in `db.js` è implementata la classe **Database** che è una super classe di InfluxDB e PostgreDB.

La directory **database script** contiene vari script per creare, eliminare o riempire i database.

La directory **custom modules** contiene moduli generici utilizzati da tutti i componenti:

- **delay** contiene una funzione per sospendere un processo in modo asincrono.;

- **errors** contiene varie classi di errore;
- **getIoTData** contiene la classe IoTData che permette di interfacciarsi con lo ZDM e di aggiornare il token di autenticazione quando scade;
- **safe** contiene le funzioni per criptare e decriptare con **crypto-js.**;
- **time** è il file più importante in quanto contiene varie classi che offrono metodi per manipolare le date. Inoltre contiene la funzione **definePeriods**. Tutti i metodi utilizzano il pacchetto npm **moment**.

Il file **.env** contiene le variabili d'ambiente in cui sono memorizzate:

- le informazioni per comunicare con gli altri componenti come numero della porta od host;
- le chiavi per criptare/decriptare;
- il tempo di scadenza di un JWT.

La lettura del file **.env** avviene usando il pacchetto npm **dotenv**.

A ogni componente è associata una directory chiamata con il nome del componente. Ognuna di queste directory è organizzata in questo modo:

- il file **index.js** è quello che deve essere eseguito per attivare il componente;
- tutti i **middleware** sono salvati nella directory **middleware**;
- tutte le **funzioni di gestione di un endpoint** sono salvate nella directory **routers**.

4.11 Test sulle richieste dei dati

La maggior parte dei test sono stati effettuati usando le interfacce generate dal generatore di interfacce grafiche [10] in quanto mostrano una rappresentazione grafica delle time series generate da FiBo, per cui è più semplice verificare che tutto funzioni correttamente. I dati sono generati e pubblicati sullo ZDM da uno script Python che viene eseguito su un PC acceso sempre eccetto la notte. Sono stati eseguiti anche dei test sul funzionamento delle funzioni di aggregazione e sulla tenuta del server.

4.11.1 Caching e memoizzazione

Oltre alla memorizzazione delle time series, un altro meccanismo di caching utilizzato dallo stub è la memoizzazione. La memoizzazione è *una tecnica di programmazione che consiste nel salvare in memoria i valori restituiti da una funzione in modo da averli a disposizione per un uso successivo* [19]. Nel caso dello stub, l'elaborazione della richiesta dei dati dalla fase II in poi è memoizzata. Questa tecnica è particolarmente efficace per gestire le richieste delle interfacce GUI mentre non si rivela particolarmente utile nel gestire le richieste più randomiche dell'interfaccia in linguaggio naturale. Il grafico 4.8 mostra le performance del server mentre serve cento

utenti in parallelo, ognuno dei quali fa cento richieste. I thread che rappresentano gli utenti vengono attivati nel giro di cento secondi (un thread al secondo). Allo scopo di simulare un sito internet le possibili richieste sono venti. Il throughput aumenta mano a mano che i dati vengono salvati nel time series database che inizialmente è vuoto. Il test è chiamato **jmeter_test** ed è eseguito usando **Jmeter**.

Un secondo test, chiamato **caching_test** e scritto in nodejs testa il tempo di risposta sia per richieste random in stile assistente virtuale che per la stessa richiesta ripetuta nel tempo. Il tempo medio restituito è in millisecondi. All'inizio di ogni test viene svuotato automaticamente il time series database.

Codice 4.6: tempo medio di risposta di FiBo con la stessa richiesta.

```
*****Test1*****
tempo medio per 5 richieste inviate a blocchi di 1 : 100.8
Dopo 1 secondo
tempo medio per 5 richieste inviate a blocchi di 1 : 5.8
Dopo 15 secondi
tempo medio per 5 richieste inviate a blocchi di 1 : 55.2
Dopo 30 secondi
tempo medio per 5 richieste inviate a blocchi di 1 : 81.8
*****Test2*****
tempo medio per 250 richieste inviate a blocchi di 25 : 123.04
Dopo 1 secondo
tempo medio per 250 richieste inviate a blocchi di 25 : 56.108
Dopo 15 secondi
tempo medio per 250 richieste inviate a blocchi di 25 : 80.012
Dopo 30 secondi
tempo medio per 250 richieste inviate a blocchi di 25 : 86.808
```

Codice 4.7: tempo medio di risposta di FiBo con richieste scelte in modo random.

```
*****Test1*****
tempo medio per 5 richieste inviate a blocchi di 1 : 1517
Dopo 1 secondo
tempo medio per 5 richieste inviate a blocchi di 1 : 627
Dopo 15 secondi
tempo medio per 5 richieste inviate a blocchi di 1 : 1039.6
Dopo 30 secondi
tempo medio per 5 richieste inviate a blocchi di 1 : 713.4
*****Test2*****
tempo medio per 250 richieste inviate a blocchi di 25 : 664.404
Dopo 1 secondo
tempo medio per 250 richieste inviate a blocchi di 25 : 473.728
Dopo 15 secondi
tempo medio per 250 richieste inviate a blocchi di 25 : 499.94
Dopo 30 secondi
tempo medio per 250 richieste inviate a blocchi di 25 : 415.516
```

Nel codice 4.6 il tempo medio rilevato dopo un secondo dalle prime richieste è più basso rispetto alle altre rilevazioni grazie alla memoizzazione della richiesta.

I valori restituiti dallo stub vengono memoizzati per al massimo quindici secondi. Questo significa che la terza e la quarta iterazione dei test non utilizzano questo ulteriore livello di caching venendo eseguiti dopo quindici secondi. Le richieste hanno granularità 30 secondi.

Nel test 4.7 il flag *store* delle richieste è posto a **false** come è stato indicato di fare nelle richieste da parte di un chat bot, per cui le time series non standard non vengono salvate nel database. In questo caso la riduzione del tempo medio è probabilmente dovuta al riutilizzo di time series standard per calcolare time series non standard.

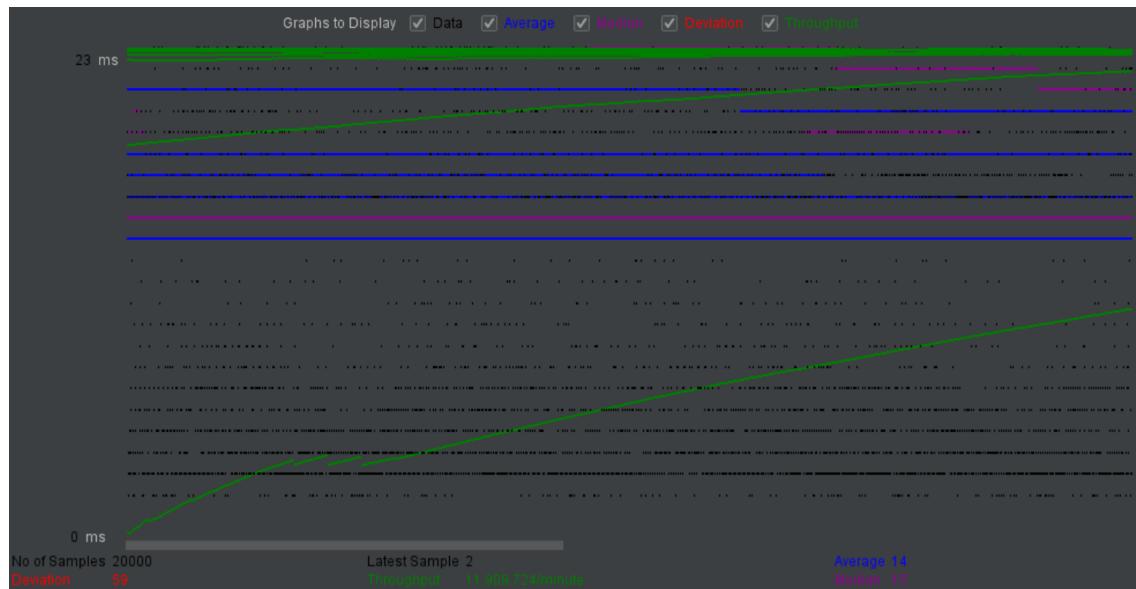


Figura 4.8: Test con 100 thread. Grafico generato da JMeter.



Figura 4.9: Test con 300 thread. Grafico generato da JMeter.

CAPITOLO

5

FiBo CLI

La CLI permette al super utente di creare ed eliminare progetti realizzati creando un file di configurazione. Il capitolo inizia con una panoramica dei pacchetti npm utilizzati per la generazione della CLI per poi mostrare un suo semplice flusso di utilizzo. Il numero di funzionalità può essere notevolmente aumentato, con per esempio, la possibilità di attivare o disattivare gli allarmi di una determinata fleet (motivo per cui la tabella alarm ha già il campo active in figura B.1)

5.1 Implementazione

5.1.1 Tecnologie utilizzate

Come per tutti i moduli che ho direttamente implementato, anche la FiBo CLI è scritta in nodejs. In questo caso i due moduli che permettono una creazione estremamente veloce di una CLI sono **commander** e **inquirer**.

Commander¹ permette di interpretare facilmente l'input dell'utente e di associare a un determinato comando una callback. **Inquirer**² permette invece di creare CLI interattive. Inoltre è stato utilizzato il plugin di inquirer **inquirer-command-prompt** per generare una cli con autocompletamento e history. Oltre a questi due, mi sono servito del pacchetto **chalk** per colorare la console, del pacchetto **figlet** per generare il titolo iniziale e del pacchetto **cli-spinner** per creare le animazioni di caricamento.

5.1.2 Funzionalità

Attualmente le funzionalità offerte dalla cli sono:

- il **login** con il comando `login` o `l`;
- il **logout** con il comando `logout`;
- la **creazione di un progetto** a partire da un file di configurazione con il comando `createProject` o `cp`;
- l'**eliminazione di un progetto** con il comando `deleteProject`;

¹<https://www.npmjs.com/package/commander>

²<https://www.npmjs.com/package/inquirer>

- la **lista di tutti i progetti che si possiede** con il comando `ls`.

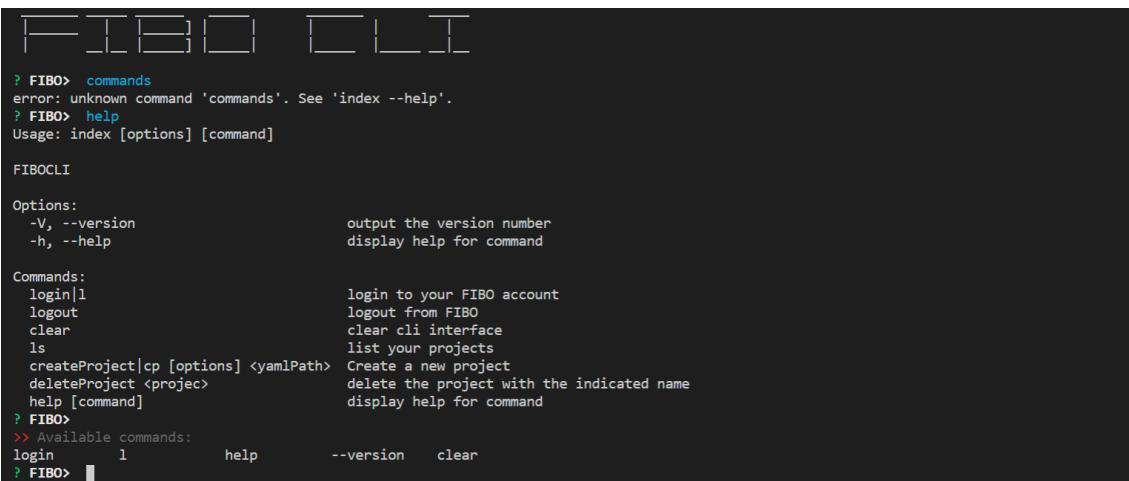
Quando un utente digita un comando interagisce con un prompt di inquirer. L'input inviato dall'utente viene parsato da un oggetto istanza della classe Command implementata utilizzando **Commander** il quale esegue, se possibile, la funzione corrispondente al comando richiesto.

Ogni funzione interagisce con FiBo utilizzando le funzioni della classe FIBO.

5.2 Esperienza di utilizzo

5.2.1 Assistenza

La prima cosa che potrebbe fare un utente principiante è scrivere una parola arbitraria che probabilmente non corrisponde a nessun comando. In questo caso la cli stampa l'errore suggerendo all'utente di utilizzare il comando `help`. Un'altra azione che può fare è premere il tasto `tab` in modo da attivare l'autocompletamento. Nella figura 5.1 l'utente non si è ancora autenticato, per cui l'autocompletamento non mostra i comandi per la creazione o l'eliminazione di un progetto. I comandi `help` e `version` sono generati automaticamente da Commander.



```
FIBO CLI
? FIBO> commands
error: unknown command 'commands'. See 'index --help'.
? FIBO> help
Usage: index [options] [command]

FIBOCLI

Options:
  -V, --version           output the version number
  -h, --help               display help for command

Commands:
  login|l                 login to your FIBO account
  logout                  logout from FIBO
  clear                   clear cli interface
  ls                      list your projects
  createProject|cp [options] <yamlPath> Create a new project
  deleteProject <project> delete the project with the indicated name
  help [command]          display help for command
? FIBO>
>>> Available commands:
login      l      help      --version    clear
? FIBO> █
```

Figura 5.1: Assistenza CLI. Il font del titolo è cyberlarge.

5.2.2 Login e logout

I comandi per il login sono `login` o `l`. La procedura richiede all'utente l'username e la password da superutente utilizzando nuovamente Inquirer.

Successivamente, per eseguire il logout, l'utente deve scrivere la parola completa `logout` e poi rispondere affermativamente o premere invio alla domanda della cli in cui gli viene chiesto se è sicuro di volersi scollegare.

```
? FIBO> login
? Insert your username William
? Insert your password [hidden]
Authentication successful!
? FIBO> logout
? are you sure you want to logout? Yes
logout successful!
```

Figura 5.2: Login e logout.

5.2.3 creazione ed eliminazione del progetto

Il compito principale della CLI nella creazione di un progetto in FiBo è guidare l'utente nella correzione degli errori contenuti nello YAML ed evitare che allo stub arrivi una configurazione errata. In caso di errore quindi indica dove si trova l'errore e in alcuni casi mostra i valori sbagliati.

```
? FIBO> cp filediconfigurazione
Yaml error: enabledfleet property in view called Temperatura contains id f1t-esempio which is not in fleets property
? FIBO> cp filediconfigurazione
Yaml error: user 2 is not valid. Password is empty
? FIBO> cp filediconfigurazione
Yaml error: two users with the same mail pino@esempio.it
```

Figura 5.3: Errori nel file di configurazione.

Se viene utilizzata l'opzione **-o** per sovrascrivere un progetto già esistente, all'utente viene chiesto se è sicuro dell'operazione. Per confermare l'utente deve scrivere yes o y mentre premere il tasto invio annullerà la procedura.

Alla fine della procedura l'utente deve inserire il suo username e la sua password per accedere allo ZDM.

```
? FIBO> cp filediconfigurazione
? Insert your ZDM email wilsimoni@gmail.com
? Insert your ZDM account password [hidden]
you have already created a project called esempio
use: cp -o filediconfigurazione.yaml to overwrite the old project
? FIBO> cp -o filediconfigurazione.yaml
? Are you sure you want to overwrite the esempio project? Yes
? Insert your ZDM email wilsimoni@gmail.com
? Insert your ZDM account password [hidden]
Project creation successful
```

Figura 5.4: Creazione del progetto terminata con successo.

Infine proviamo a eliminare il progetto appena creato. In questo caso non viene richiesta la conferma poiché per eseguire il comando è necessario scrivere il nome del progetto, quindi si può supporre che l'utente sia consapevole di ciò che sta eliminando. In figura 5.5 il comando **ls** ci permette di verificare se la creazione o l'eliminazione del progetto sono andate a buon fine.

<pre>Project creation successful ? FIBO> ls ? Your Projects name and workspaceuid: (Use arrow keys or type to search)■ > exit prova wks-4uriwnoxp70d esempio wks-4uriwnoxp70d</pre>	<pre>? FIBO> deleteProject esempio Project deleted ? FIBO> ls ? Your Projects name and workspaceuid: (Use arrow keys or type to search)■ > exit prova wks-4uriwnoxp70d</pre>
---	---

(a)

(b)

Figura 5.5

CAPITOLO

6 | Conclusioni

6.1 Sviluppi Futuri di FiBo

6.1.1 Sicurezza

Sicuramente uno degli aspetti più importanti che ho sorvolato nella progettazione di FiBo è la sicurezza delle comunicazioni in quanto praticamente tutto viene scambiato utilizzando il protocollo **http**. Per migliorare la sicurezza delle comunicazioni verso i client si potrebbe aggiungere tra i client e il server un **proxy inverso** che implementi **https** e poi inoltri le richieste al server utilizzando **http**. Il proxy inverso si potrebbe inoltre occupare della compressione delle risposte del server, lavoro che attualmente svolge direttamente lo stub utilizzando il pacchetto npm **compression**.

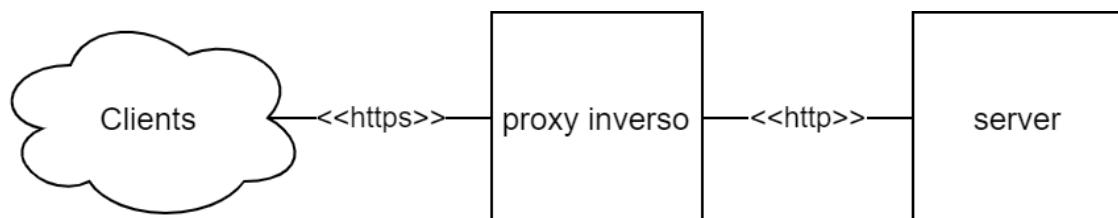


Figura 6.1: Procy inverso.

Anche le comunicazioni interne dovrebbero implementare SSL per supportare la possibilità di eseguire i vari componenti in ambienti diversi.

6.1.2 Clustering

Data la natura single thread di nodejs è stato quasi naturale dividere il sistema in diversi componenti. Tutti i componenti nodejs eccetto l'alarm finder possono essere clusterizzati senza troppi problemi permettendo quindi una buona scalabilità orizzontale. La creazione dei cluster può essere gestita utilizzando un process manager come **pm2**. L'alarm finder non può essere diviso in cluster perché, per come è implementato, ogni cluster gestirebbe tutti gli allarmi contenuti all'interno del database. Sarebbe quindi necessario, o un protocollo di comunicazione tra i cluster, in modo che questi si suddividano gli allarmi tramite uno scambio reciproco di messaggi, o un sistema centralizzato, che gestisca la distribuzione degli allarmi fra i vari cluster.

6.1.3 Aggiornamento del token

Per accedere ai dati di un utente dello ZDM è necessario che FiBo sia in possesso di un token di accesso relativo a quell'utente. Quando questo token scade, è necessario che il server riesegua la procedura di login per ottenere un nuovo token. Il numero di volte che il login viene eseguito potrebbe essere maggiore di uno. Questo perché l'aggiornamento del token è gestito all'interno della classe **getIoTData**, per cui sia l'alarm finder che il calculator potrebbero nello stesso momento eseguire una richiesta di login. Il modo più semplice per risolvere questo problema è creare un ulteriore componente il cui unico compito è effettuare login e aggiornare i token.

6.1.4 Fetching continuo dei dati

Attualmente i dati vengono presi dallo ZDM su richiesta. Potrebbe essere possibile potenziare l'alarm finder in modo che si occupi del fetching e dell'aggregazione di tutti i dati indicati nel file di configurazione. Per aggregare i dati si potrebbe utilizzare un'ulteriore funzionalità di InfluxDB che permette di effettuare aggregazioni periodiche sui dati (**continuous query**)¹.

6.2 Conclusioni

Partecipare a questo progetto è stata un'esperienza estremamente formativa e stimolante. Da una parte, è stato molto divertente lavorare in gruppo per riuscire a pensare e strutturare un sistema che raggiungesse, con i suoi limiti, l'obiettivo del progetto. Dall'altra progettare e implementare FiBo mi ha permesso di conoscere nuove tecnologie come nodeJS o InfluxDB e allo stesso tempo di poter conoscere alcune delle tecniche standard utilizzate nel mondo del web. L'utilizzo di nodeJS e InfluxDB si è rivelato per lo più una soluzione vincente grazie alla loro **flessibilità** e **semplicità**, caratteristiche fondamentali in quanto molto spesso il file di configurazione, e quindi la struttura dei dati, veniva cambiato.

Alcune parti del progetto sono sicuramente migliorabili. Per esempio, al momento il processo di creazione di un progetto consiste di quattro macro passi (1.creazione del file; 2.generazione della GUI; 3.generazione dell'assistente virtuale; 4.configurazione di FiBo) che potrebbero essere uniti e automatizzati. Un altro aspetto un po' tralasciato riguarda l'aggiornamento di un progetto (aggiungere un utente o una fleet). Anche questa operazione consta di quattro passi che potrebbero essere ridotti a uno solo.

Sono comunque soddisfatto del risultato finale. La generazione di una dashboard industriale risulta molto veloce grazie al fatto che il file di configurazione si concentra principalmente sul contenuto informativo. Ad esempio, la scrittura del file di configurazione del capitolo due ha richiesto poco più di dieci minuti, la generazione e la configurazione del progetto sono processi molto più veloci, per cui in totale il tempo perso per realizzare il progetto **esempio** è stato fra i quindici e i venti minuti.

¹Purtroppo questa strada non è ancora praticabile a meno di cambiare leggermente lo schema del database in quanto le continuous query non sono sufficientemente espresse per permettere una buona gestione delle aggregazioni con tale schema

6.3 Ringraziamenti

Concludo ringraziando i colleghi Alessandro Belfiore, Matteo Menghini e Sergio Attanzio con i quali è stato un piacere lavorare, anche in tempo di pandemia, a questo progetto. Ringrazio anche il professore Daniele Mazzei che ci ha guidati nella realizzazione del sistema.

CAPITOLO

A

Configurazione di esempio

Codice A.1: file di configurazione creato nell'esempio del capitolo 2

```
1 projectname: esempio
2 template: Standard
3 workspace: wks-esempio
4 fleets:
5   - name: Fleet1
6     id: flt-esempio1
7   - name: Fleet2
8     id: flt-esempio2
9 lex:
10    region: eu-west-1
11    voice: salli
12 views:
13   - viewname: Temperatura
14     enabledfleet:
15       - flt-esempio1
16       - flt-esempio2
17     data:
18       - title: temperatura interna media ultimo minuto
19         type: temperature
20         timeseries:
21           - aggregation:
22             tagvalue:
23               - tag: temperatura interna
24               value : temp
25         aggregationfunction: mean
26         timerange: 1 minute
27         granularity: 1 minute
28         chart:
29           type: gauge
30           min: 0
31           max: 40
32           - title: temperatura media ultime 24 ore
33             type: temperature
34             timeseries:
35               - aggregation: mean
```

```
36          tagvalue:
37              - tag: temperatura interna
38                  value : temp
39              - tag: temperatura esterna
40                  value : temp
41      aggregationfunction: mean
42      timerange: 1 day
43      granularity: 1 hour
44      chart:
45          type: line
46      alarm:
47          - tag : temperatura interna
48              value: temp
49              max: 60
50              min: 40
51          - tag: temperatura esterna
52              value: temp
53              max: 30
54              min: 0
55      - viewname: Umidita'
56      enabledfleet:
57          - flt-esempio2
58      data:
59          - title: Umidita giornaliera
60          type: humidity
61          timeseries:
62              - aggregation:
63                  tagvalue:
64                      - tag: umidita
65                          value : humidity
66          aggregationfunction: mean
67          timerange: 1 day
68          granularity: 1 hour
69          chart:
70              type: line
71          - title: Umidita massima
72          type: humidity
73          timeseries:
74              - aggregation: max
75                  tagvalue:
76                      - tag: umidita
77                          value : humidity
78          aggregationfunction: max
79          timerange: 2 week
80          granularity: 1 day
81          chart:
82              type: column
83      alarm:
84          - tag : umidita
```

```
85             value: humidity
86             max: 30
87             min: 0
88 users:
89     - mail: admin@esempio.it
90         pass: admin
91         role: Admin
92     - mail: pino@esempio.it
93         pass: '12345678'
94         role: Viewer
95     fleets:
96         - flt-esempio1
97     - mail: gino@esempio.it
98         pass: '12345678'
99         role: Viewer
100    fleets:
101        - flt-esempio2
102 datatype:
103     - name: humidity
104         unitofmeasure: kg / m^3
105     - name: temperature
106         unitofmeasure: C
```

CAPITOLO

B

Schemi del process database

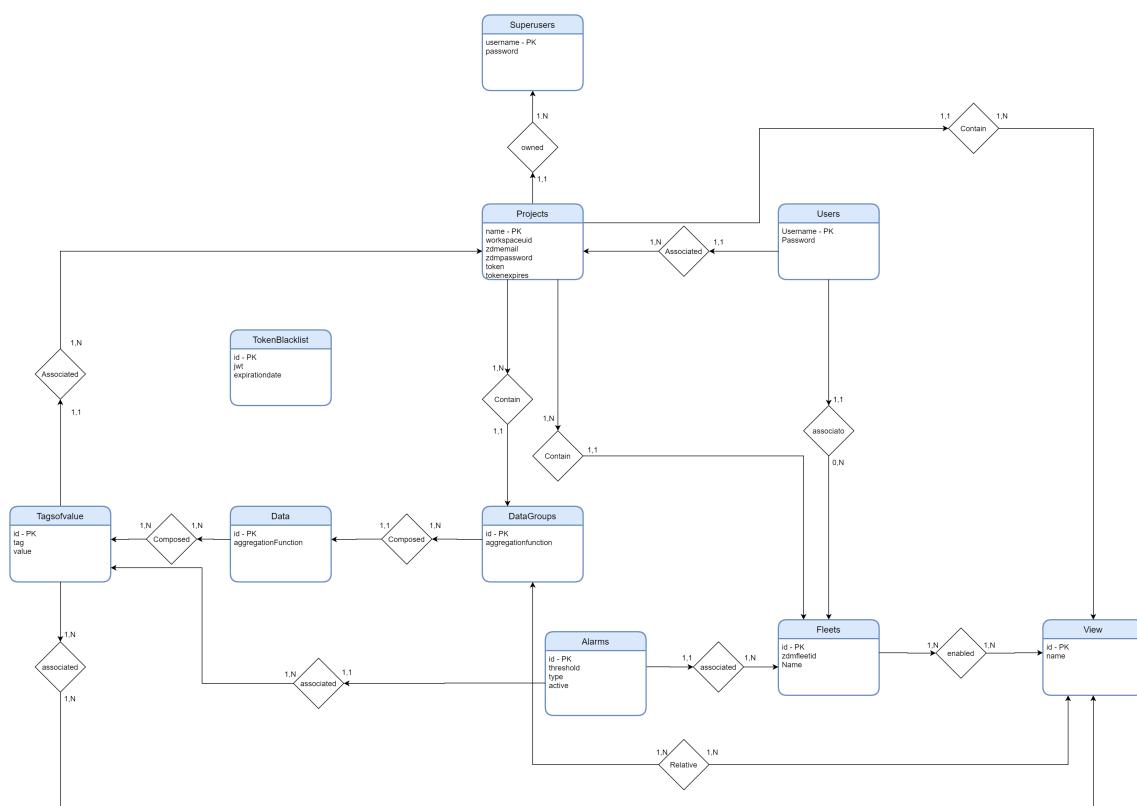


Figura B.1: Modello concettuale project database

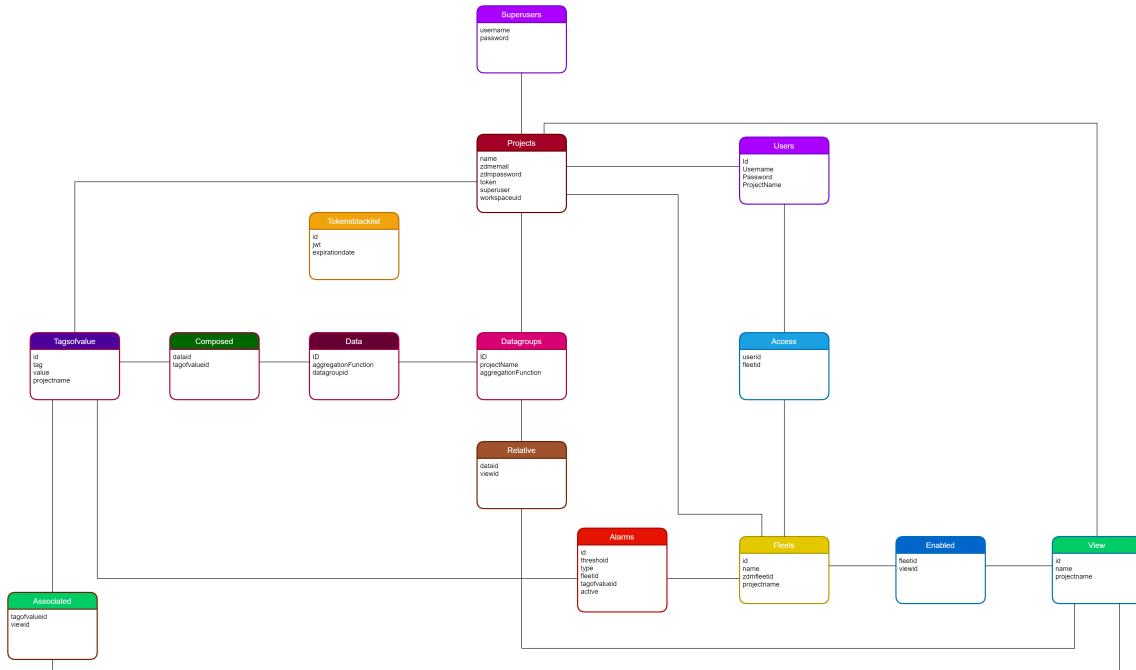


Figura B.2: Modello logico project database

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
username	text	✓	✓		
password	text	✓			

Tabella B.1: superuser

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
name	text	✓	✓		
token	text	✓			
tokenexpires	bigint	✓			
workspaceuid	text	✓			
zdmemail	text	✓			
zdmpassword	text	✓			
superuser	text	✓		✓	

Tabella B.2: project.

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
name	text	✓			
zdmfleetid	text	✓			✓
projectname	text	✓		✓	✓

Tabella B.3: project.

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
name	text	✓			✓
projectname	text	✓		✓	✓

Tabella B.4: views

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
fleetid	uuid	✓	✓	✓	
viewid	uuid	✓	✓	✓	

Tabella B.5: enabled

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
aggregationfunction	text	✓			
projectname	text	✓		✓	

Tabella B.6: datagroups

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
aggregationfunction	text	✓			
datagroupid	text	✓		✓	

Tabella B.7: data

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
tag	text	✓			
value	text	✓			
projectname	text	✓		✓	

Tabella B.8: tagsofvalue

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
tagofvalueid	uuid	✓	✓	✓	
dataid	uuid	✓	✓	✓	

Tabella B.9: composed

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
threshold	text	✓			
type	alarmtype	✓			
fleetid	uuid	✓		✓	
tagofvalueid	uuid	✓		✓	
active	boolean	✓			

Tabella B.10: alarms. Alarmtype può essere "min" o "max".

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
username	text	✓			✓
password	text	✓			
projectname	text	✓		✓	✓

Tabella B.11: users

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
userid	uuid	✓	✓	✓	
fleetid	uuid	✓	✓	✓	

Tabella B.12: access

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
datagroupid	uuid	✓	✓	✓	
viewid	uuid	✓	✓	✓	

Tabella B.13: access

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
id	uuid	✓	✓		
token	text	✓			
expirationdate	bigint	✓			

Tabella B.14: tokensblacklist

nome attributo	tipo	NOT NULL	PK	FK	UNIQUE
tagofvalueid	uuid	✓	✓	✓	
viewid	uuid	✓	✓	✓	

Tabella B.15: access

CAPITOLO C | Studio sui grafici

Durante i mesi di lavoro ho svolto una breve indagine sul come e quando rappresentare i grafici. I risultati della ricerca sono riassunti in questa appendice e sono alla base di alcune scelte che sono state fatte nella struttura del file di configurazione. Prima di procedere, è importante sottolineare che i grafici 4.8 e 4.9 sono un buon esempio di come NON creare un grafico.

C.1 I grafici, elenco e regole di design

Nel seguito si utilizzerà il termine **variabile** riferendosi al concetto di **timeSeries** del file di configurazione (vedi sezione 2.2.2). Quindi una variabile è caratterizzata da una o più fonti di dati (coppie di tag e di value) e da una funzione di aggregazione che le aggrega.

C.1.1 Icon chart e gauge chart

Quando si vuole **evidenziare** il valore di un **unica** variabile in un **determinato** momento il modo migliore per farlo è utilizzando:

- un **icon chart**, quindi un grafico in cui viene mostrato il valore della variabile con di fianco un icona che dovrebbe riferirsi al tipo di variabile mostrato;
- un **gauge chart**, ovvero un grafico con un arco circolare sul quale viene disegnata una barra progressiva che avanza o arretra in base alla distanza del valore della variabile da una soglia massima fissata. Il gauge chart può avere un ago che punta al valore del dato (es: **tachimetro**).

C.1.2 Column e bar chart

I column chart e i bar chart sono grafici in cui i dati di una variabile sono rappresentati per mezzo di colonne verticali o orizzontali. Questi due tipi di grafico sono abbastanza interscambiabili ma in generale:

- il **column chart** è migliore se si ha intenzione di mostrare l'andamento nel tempo di una o due variabili;

- il **bar chart** è migliore se si vuole confrontare i valori di più variabili (massimo quindici) relativi ad un unico periodo di tempo, se le label¹ sono molto lunghe e se il valore delle variabili può essere negativo.

Se in un bar chart le colonne non devono essere ordinate in qualche modo specifico (es: in base al tempo) allora potrebbero essere ordinate in base alla lunghezza (vedi figura C.1).

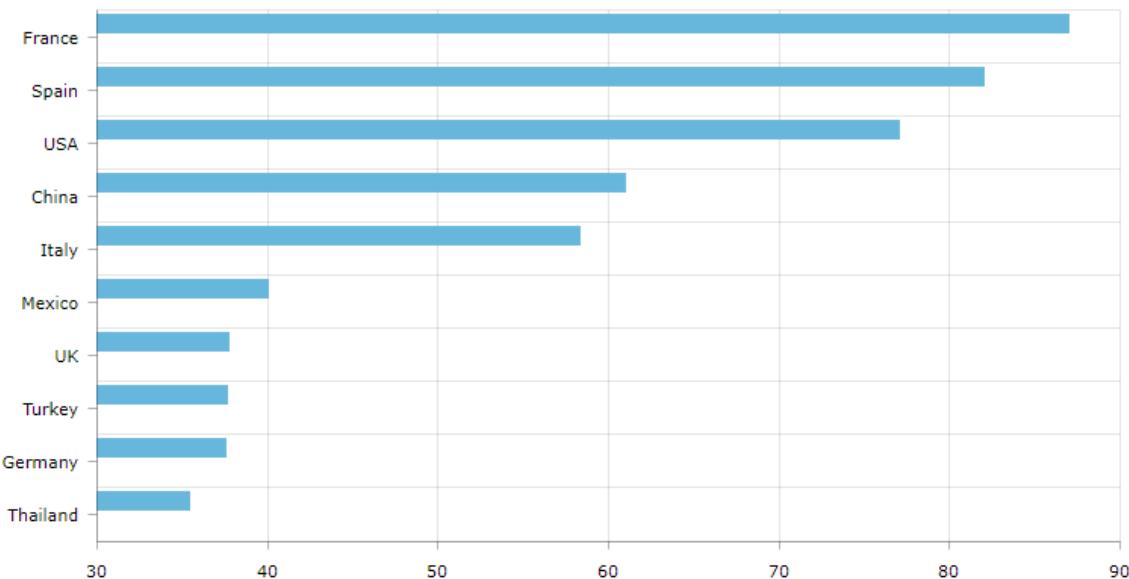


Figura C.1: Bar chart che mostra il numero di turisti relativi alle dieci nazioni più visitate nel mondo nel 2019. Dati raccolti da <https://www.travel365.it/paesi-piu-visitati-al-mondo.htm>.

C.1.3 Line chart

Il line chart è un grafico in cui i valori delle variabili sono rappresentati per mezzo di linee. Questo grafico è un ottimo strumento per evidenziare l'entità del cambiamento nel tempo di una o più variabili.

In un line chart bisogna cercare di evitare di mostrare più di quattro linee in modo da non generare confusione.

C.1.4 Pie chart e donut chart

Il pie chart è un grafico circolare diviso in fette. Ogni fetta rappresenta una parte di un intero. Più grande è la fetta, maggiore è il suo contributo sul totale.

Il pie chart permette di mostrare il contributo, in percentuale, delle parti sul totale in un periodo di tempo. Nel creare un pie chart è importante che siano considerate tutte le parti che compongono l'intero, altrimenti il grafico potrebbe mostrare un'informazione sbagliata.

Alcune regole per creare un buon pie chart sono:

¹etichette di asse

- evitare più di sette variabili nello stesso pie chart;
- la prima fetta dovrebbe partire dalle ore dodici;
- le fette dovrebbero essere ordinate dalla più grande alla più piccola;
- non vanno usati nel caso in cui i valori delle variabili siano negativi;
- evitare i pie chart inclinati o 3d in quanto potrebbero distorcere la visualizzazione dei dati. Per esempio, in figura C.2 il 19% della Apple sembra molto più grande del 21% di Other.

Un donut chart è un pie chart a forma di ciambella in cui al centro spesso viene mostrata la sommatoria dei valori delle variabili che compongono il grafico. Con i donut chart è più difficile distorcere la visualizzazione dei dati.

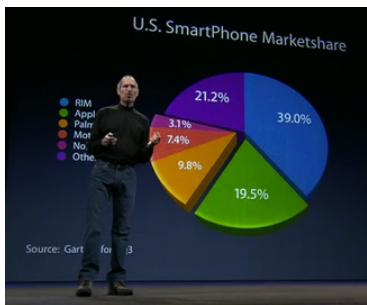


Figura C.2: Steve Jobs mentre mostra la quota di Apple nel mercato smartphone in USA.



Figura C.3: Lo stesso pie chart non distorto.



Figura C.4: Gli stessi dati in un donut chart.

C.1.5 Area chart e versioni stacked di column chart e line chart

Riprendendo l'esempio del pie chart, si vuole mostrare le quote del mercato smartphone mondiale nei quadrimestri del 2019. Quando si vuole mostrare come le parti di un intero cambiano nel tempo, come in questo caso, i grafici da utilizzare sono l'**area chart** e lo **stacked column chart**. L'area chart è essenzialmente un line chart in cui l'area sotto la linea è colorata. Esistono tre tipi di area chart:

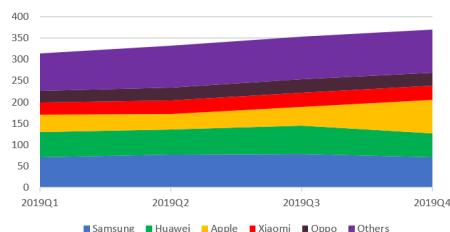
- **regular**;
- **stacked** in cui le linee sono impilate l'una sopra l'altra in modo da mostrare il contributo di ogni parte sul totale;
- **100% stacked** in cui le linee sono impilate l'una sopra l'altra in modo da mostrare il contributo, espresso in percentuale, di ogni parte sul totale.

Anche il column chart ha tre varianti:

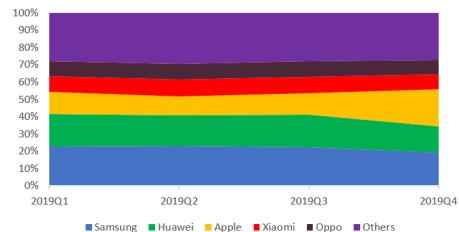
- **regular** discussa nella sezione C.1.2;

- **stacked** in cui i valori delle diverse variabili relativi allo stesso periodo vengono impilati in un'unica colonna in modo da mostrare il contributo di ogni parte sul totale;
- **100% stacked** in cui i valori delle diverse variabili relativi allo stesso periodo vengono impilati in un'unica colonna in modo da mostrare il contributo, espresso in percentuale, di ogni parte sul totale;

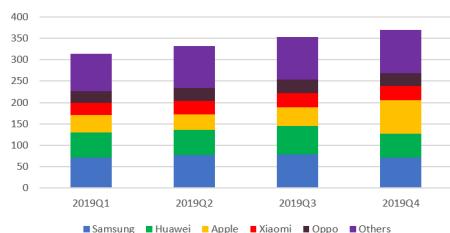
In generale, la versione stacked andrebbe utilizzata quando si vuole mostrare oltre al valore delle parti anche il valore del totale. La versione 100% stacked è invece migliore quando si vuole indicare in che misura le parti contribuiscono sul totale.



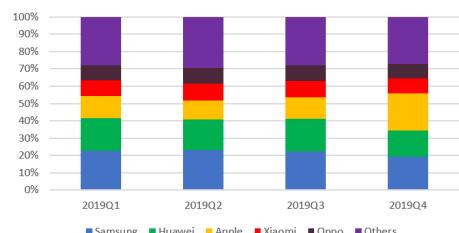
(a) Stacked area chart.



(b) 100% stacked area chart.



(c) Stacked column chart.



(d) 100% stacked area chart.

Figura C.5: Nell'esempio viene mostrato il numero, in milioni, di smartphone venduti nel mercato mondiale nel 2019. I dati sono stati presi da **Canalys**. I grafici sono stati realizzati con excel.

La versione **regular** dell'area chart andrebbe evitata come dimostra la figura C.6. Se non si ha intenzione di usare le versioni stacked dell'area chart allora è meglio scegliere un semplice line chart. Come per l'area chart, anche il line chart ha una sua versione stacked e 100% stacked. Queste due versioni del line chart andrebbero evitate. Per convincersene guardate le figure C.7 e C.8.

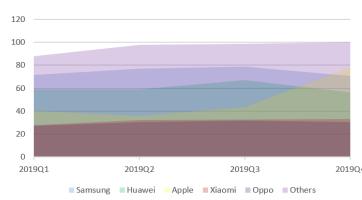


Figura C.6: Regular area chart. L'utente non riesce a capirci niente.

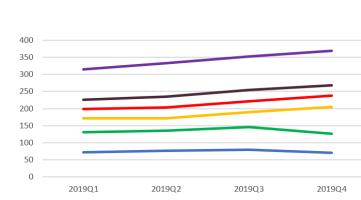


Figura C.7: Stacked line chart.

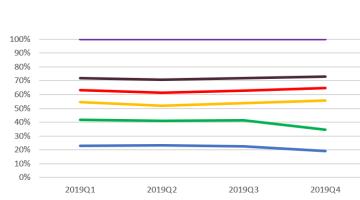


Figura C.8: 100% stacked line chart.

Alcune regole che si potrebbero seguire per un area chart:

- usare colori trasparenti;
- inserire le variabili più instabili in alto;
- non superare le cinque variabili.

C.1.6 Scatter plot

Uno scatter plot è un grafico in cui i singoli valori delle variabili sono riportati come punti in uno spazio cartesiano. Sono generalmente utilizzati per mostrare che relazione intercorre tra due variabili o per mostrare la distribuzione dei valori di due variabili.

Per esempio, supponiamo che come azienda di trasporti pubblici, si vuole osservare la relazione **chilometri percorsi in un giorno/ costo carburante in un giorno** degli autobus che si possiede. Gli autobus a disposizione funzionano con due carburanti, il carburante verde e il carburante arancione. In altre parole, l'informazione che si vuole mostrare è composta da quattro variabili:

- x_{cb} : chilometri percorsi dal mezzo nel giorno z utilizzando il carburante **cb**;
- y_{cb} : costo per rifornire il mezzo nel giorno z utilizzando il carburante **cb**.

Dove z assume i valori da ieri fino a 1 settimana fa.

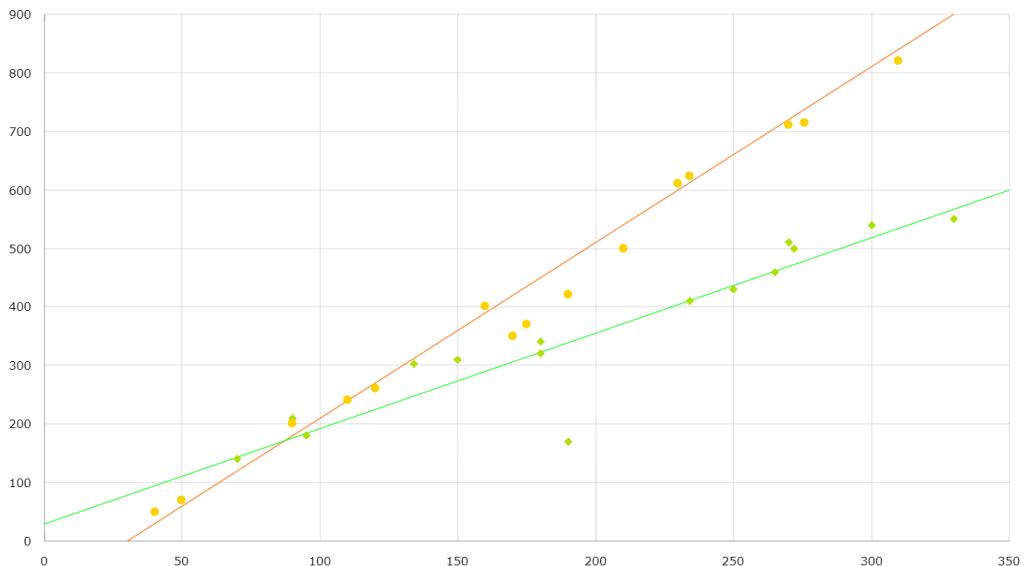


Figura C.9: Sull'asse x vengono indicati i chilometri percorsi dal mezzo nel giorno z , mentre sulla y il costo per farlo funzionare nel medesimo giorno. L'utente può subito ottenere tre informazioni: 1) più sono i chilometri percorsi da un autobus, maggiore è il costo per farlo funzionare, 2) i veicoli che funzionano con il carburante arancione costano meno quando i chilometri percorsi sono pochi (circa meno di 100 km) mentre costano di più quando i chilometri percorsi sono molti, 3) C'è un outlier (in italiano un dato anomalo) per quanto riguarda i veicoli che funzionano con il carburante verde, infatti un punto è molto distante dalla linea che mostra il trend del carburante verde.

Lo scatter plot è utile quando le due variabili sono logicamente correlate e ogni variabile assume solo valori numerici. Nel grafico C.9, si dice che le variabili sono correlate positivamente perché lo sciame di punti tende verso l'alto. Se lo sciame avesse teso verso il basso, allora le variabili sarebbero state correlate negativamente. Se le variabili non sono correlate, il risultato sarà un insieme di punti disposti casualmente all'interno del piano cartesiano.

Lo scatter plot non va usato per osservare l'andamento nel tempo delle variabili.

Alcune regole di design per lo scatter plot:

- far partire l'asse y da 0;
- Usare al massimo due linee per mostrare il trend dello sciame di punti.

C.1.7 Bubble chart

Tornando all'azienda dell'esempio precedente, oltre alle due informazioni già descritte, si vuole mostrare anche il guadagno di un autobus nel giorno z. Questo terzo dato si può mostrare usando la dimensione del punto nel piano cartesiano. Più è grande il punto più è stato alto il guadagno.

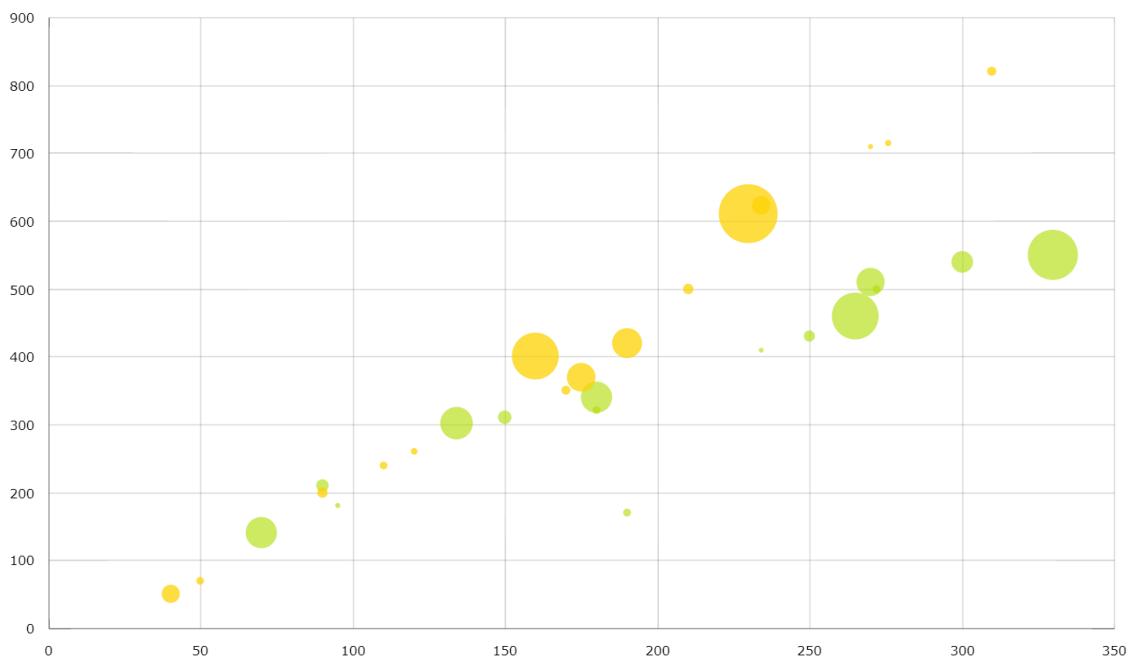


Figura C.10: Bubble chart.

Il grafico mostrato si chiama bubble chart e può essere utilizzato quando si vuole mostrare la relazione tra tre variabili in qualche modo correlate. Quindi ai due assi vengono associate due variabili mentre la terza variabile è associata alla dimensione del punto. I bubble chart come gli scatter plot permettono all'utente di trovare velocemente dati anomali.

I bubble chart vengono usati spesso insieme a una mappa. Ad esempio: sono stati posizionati, all'interno di un parco naturale, dei sensori di prossimità che inviano un allarme ogni volta qualcosa si trova nelle immediate vicinanze, presumibilmente

animali. Ogni volta che il sensore rivela un movimento invia un segnale a un server il quale memorizza l'ora dello spostamento e il device che ha inviato il segnale. Si vuole osservare, tramite una mappa, i punti in cui sono stati rilevati più movimenti e i punti in cui ne sono stati rilevati meno negli ultimi sette giorni, in modo da mappare gli spostamenti degli animali all'interno del parco.

All'asse X e all'asse Y vengono associati la posizione dei device sulla mappa(che conosciamo e sono costanti). La dimensione del punto sarà invece associata alla variabile "numero di allarmi per il device con id #".

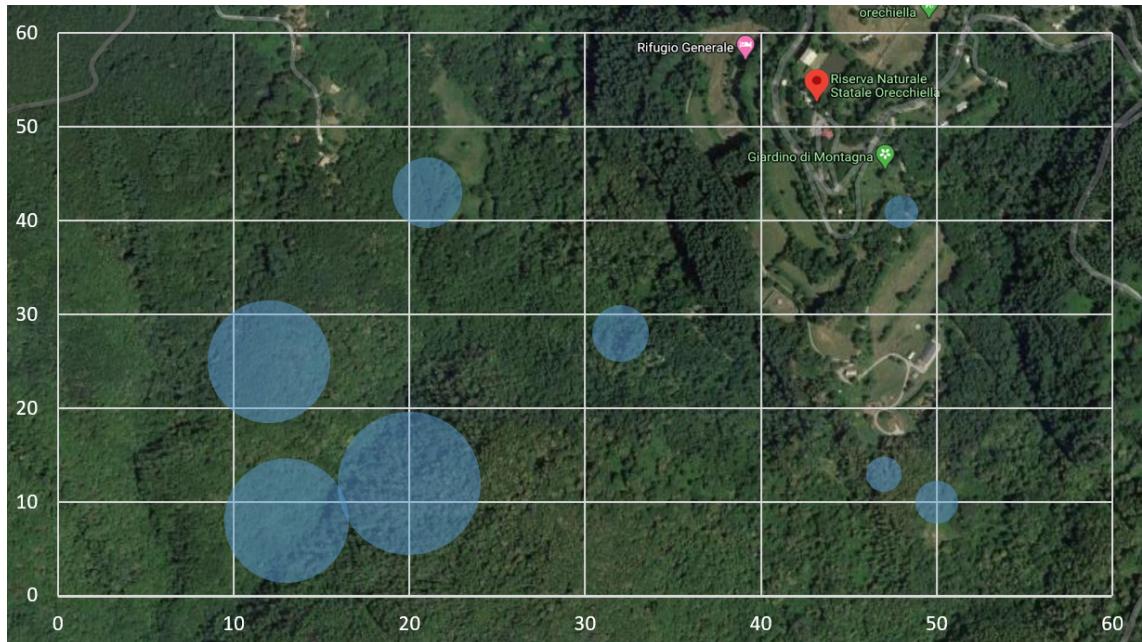


Figura C.11: La mappa mostra i luoghi del parco più frequentati dagli animali. I dati sono casuali.

Alcune regole di design per il bubble chart:

- usare solo cerchi;
- il colore di riempimento dei cerchi dovrebbe essere semi trasparente.

C.1.8 Histogram

Si vuole osservare con che frequenza la temperatura media oraria registrata da un dispositivo in ventiquattro ore rientra in vari **range** di valori:

- temperatura inferiore a 0 gradi Celsius;
- temperatura compresa tra 0 e 14 gradi Celsius;
- e così via, in range di quindici gradi, fino alle temperature superiori a 75 gradi Celsius.

Per mostrare questo genere di informazione, di solito si utilizza il grafico mostrato in figura C.12.

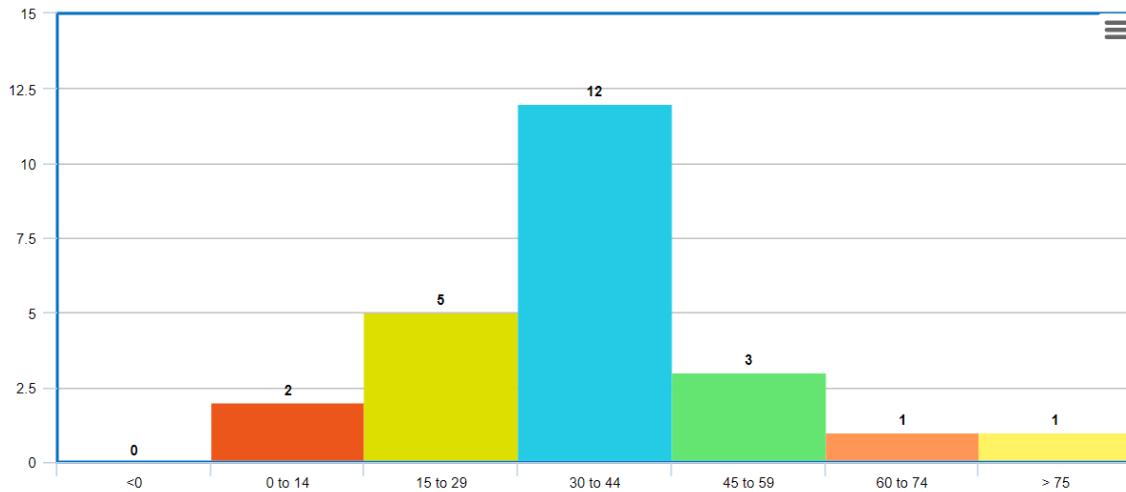


Figura C.12: Nelle ultime ventiquattr'ore, il device ha registrato una temperatura minore di 0 gradi Celius 0 volte, fra 0 e 14 gradi Celsius 2 volte, ecc.

Il grafico C.12 si chiama histogram. Gli histogram consentono di mostrare la distribuzione dei valori di una variabile. Nel farlo, vengono definiti dei contenitori a ognuno dei quali è assegnato un predicato. Ogni contenitore contiene tutti i valori che soddisfano il suo predicato. Maggiore è l'altezza della colonna associata ad un contenitore, maggiore è il numero di valori che soddisfano il suo predicato.

C.2 Sviluppi

I risultati di questa breve ricerca sono stati utilizzati per:

- definire i grafici che il generatore di GUI avrebbe dovuto generare²;
- definire le peculiarità di ogni grafico, e quindi le proprietà del campo **chart** nel file di configurazione;
- creare l'algoritmo di scelta del grafico utilizzato nel generatore di configurazioni.

Il documento intitolato **charts** [9], contiene una prima versione dell'algoritmo di scelta del grafico e una lista delle caratteristiche di tutti i grafici elencati in questa appendice.

C.3 Fonti

Le fonti utilizzate sono elencate nella bibliografia dell'appendice C. Tutti i grafici sono stati realizzati utilizzando excel o live.amcharts.com. Alcuni esempi sono stati presi dal documento **charts** mentre altri sono inediti.

²Lo scatter plot, il bubble chart, l'histogram e l'area chart non sono supportati al momento della stesura della tesi

Bibliografia

- [1] LuigiDCapra. *Origini del termine Industrie 4.0.* consultato 11 maggio 2020.
URL: http://luigidcapra.altervista.org/it/Fabbrica_40/Origini_del_termine_Industrie_4.0.html.
- [2] Bernard Marr. *Why Everyone Must Get Ready For The 4th Industrial Revolution.* 2016. URL: <https://www.forbes.com/sites/bernardmarr/2016/04/05/why-everyone-must-get-ready-for-4th-industrial-revolution/#42ca3bd63f90>.
- [3] Gizem Erboz. *how to define industry 4.0: The Main Pillars of Industry 4.0.* 2017. URL: https://www.researchgate.net/publication/326557388_How_To_Define_Industry_40_Main_Pillars_Of_Industry_40.
- [4] Governo Italiano. *Piano nazionale industria 4.0.* consultato 11 maggio 2020.
URL: https://www.mise.gov.it/images/stories/documenti/Piano_Industria_40.pdf.
- [5] Zerounoweb. *definizione IoT.* consultato 11 maggio 2020. URL: https://www.zerounoweb.it/tag/iot/?_ga=2.189722027.1820229467.1589182970-772114962.1589182970.
- [6] Wikipedia. *Internet delle cose.* consultato 11 maggio 2020. URL: https://it.wikipedia.org/wiki/Internet_delle_cose.
- [7] Zerounoweb. *L'Internet delle cose (IoT): cos'è e come rivoluzionerà prodotti e servizi.* 2020. URL: <https://www.zerounoweb.it/analytics/big-data/internet-of-things-iot-come-funziona/>.
- [8] Matteo Menghini. *Progettazione e sviluppo di un sistema di configurazione guidata per interfacce multi-modali.* 2020.
- [9] Zerynth. *Zerynth Device Manager documentation.* consultato 15 maggio 2020.
URL: https://docs.zerynth.com/latest/official/core.zerynth.toolchain/zdevicemanager/docs/docs_cli_total.html#.
- [10] Alessandro Belfiore. *Design e sviluppo di un framework per la generazione di dashboard in contesti Industria 4.0.* 2020.
- [11] Sergio Attanzio. *Progettazione e sviluppo di un generatore di interfacce in linguaggio naturale per applicazioni in contesti industria 4.0.* 2020.
- [12] Wikipedia. *Acronimo.* consultato 15 maggio 2020. URL: https://it.wikipedia.org/wiki/Acronimo#Acronimo_ricorsivo.

- [13] Ingy döt Net Oren Ben-Kiki Clark Evans. *Yaml documentation*. 2009. URL: <https://yaml.org/spec/1.2/spec.html#id2759572>.
- [14] Amazon. *Lex documentation*. 2020. URL: https://docs.aws.amazon.com/it_it/lex/latest/dg/what-is.html.
- [15] InfluxDB. *InfluxDB documentation*. 2020. URL: https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/.
- [16] jwt.io. *JSON Web Token Introduction*. 2020. URL: <https://jwt.io/introduction/>.
- [17] Socket.io. *doc*. 2020. URL: <https://socket.io/docs/>.
- [18] Socket.io. *doc*. 2020. URL: <https://socket.io/docs/client-api/>.
- [19] Wikipedia. *Memoizzazione*. 2020. URL: <https://it.wikipedia.org/wiki/Memoizzazione>.

Bibliografia appendice C

- [1] 365 Data Science. *Choosing the right chart: Selecting among 14 chart types.* 2018. URL: <https://365datascience.com/chart-types-and-how-to-select-the-right-one/>.
- [2] Lindy Ryan. *Choosing the Right Chart: Being Successful With Data Visualization.* 2017. URL: <https://www.dbta.com/BigDataQuarterly/Articles/Choosing-the-Right-Chart-Being-Successful-With-Data-Visualization-120616.aspx>.
- [3] Midori Nediger. *How to Choose the Best Types of Charts For Your Data.* 2019. URL: <https://venngage.com/blog/how-to-choose-the-best-charts-for-your-infographic/>.
- [4] Jami Oetting. *Data Visualization 101: How to Choose the Right Chart or Graph for Your Data.* 2018. URL: <https://blog.hubspot.com/marketing/types-of-graphs-for-data-visualization>.
- [5] Microsoft. *Available chart types in Office.* 2020. URL: <https://support.office.com/en-us/article/available-chart-types-in-office-a6187218-807e-4103-9e0a-27cdb19afb90>.
- [6] Umberto Santucci. *Grafico a dispersione.* consultato nel 2020. URL: <http://www.umbertosantucci.it/grafico-a-dispersione/>.
- [7] Sandra Durcevic. *Designing Charts and Graphs: How to Choose the Right Data Visualization Types.* 2019. URL: <https://www.datapine.com/blog/how-to-choose-the-right-data-visualization-types/>.
- [8] Corsair's Publishing. *Perspective FAIL — A Visual Primer.* 2016. URL: <https://charting-ahead.corsairs.network/perspective-fail-a-visual-primer-dcd8a435ac0a>.
- [9] William Simoni. *Charts.* 2020. URL: <https://github.com/WilliamSimoni/MMI40/blob/master/charts.pdf>.