ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE "GUGLIELMO MARCONI"

CORSO DI LAUREA TRIENNALE IN INGEGNERIA BIOMEDICA

ELABORATO IN CALCOLATORI ELETTRONICI

Studio ed integrazione di Grafana per la visualizzazione di flussi di dati generati dalla piattaforma semantica SEPA

Relatore
Prof. Ing. Luca Roffia

Candidata
Adelina Vivaldo

Correlatore
Dr. Gregorio Monari
Dr.ssa Elisa Riforgato

Anno Accademico 2021–2022

A mi familia

Abstract

Questo elaborato mostra lo sviluppo di un *plugin* per la visualizzazione in Grafana di eventi provenienti dalla piattaforma semantica SEPA (SPARQL *Event Processing Architecture*).

La principale funzione svolta dal SEPA è quella di notificare in modo asincrono i propri *client* rispetto al cambiamento dei risultati di una *query* che interroga il sottostante grafo RDF. La piattaforma trova il suo utilizzo in quei contesti caratterizzati da dati dinamici, eterogenei e non strutturati e viene impiegata principalmente come strumento per abilitare l'interoperabilità in domini come per esempio l'*Internet of Things*. Nasce quindi l'esigenza di disporre di strumenti per il monitoraggio e la visualizzazione di dati *real-time*.

Grafana risulta in questo caso lo strumento ideale data la sua flessibilità, che affiancata alla sua natura *open source*, lo rende particolarmente interessante per lo sviluppo della soluzione proposta da VAIMEE, spinoff dell'Università di Bologna, ospitato presso il CesenaLab, luogo dove è stato svolto questo lavoro di tesi.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Ad oggi il mondo del monitoraggio dati offre una grande varietà di sistemi, alcuni dei quali sono *open source*. Un sistema di monitoraggio è costituito da una serie di moduli per la raccolta, l'elaborazione e la visualizzazione dei dati con lo scopo di fornire informazioni sulle prestazioni e sulla disponibilità di infrastrutture, servizi e applicazioni. In questo ambito si inserisce Grafana, un *tool* di monitoraggio *open source* che, indipendentemente dal formato in cui il *database* archivia le informazioni, permette di interrogare i dati utilizzando le *query* e di rappresentarli mediante le *dashboard*. Grafana offre all'utente un ambiente di visualizzazione completo e *human-friendly* con la possibilità di organizzare le informazioni provenienti dal *database* con un'implementazione grafica molto efficace. Le opportunità che tale piattaforma fornisce sono diverse, tra cui quella di collegare una qualsiasi *data source* grazie alla possibilità di sviluppare appositi *plugins*. L'obiettivo di questo elaborato è quello di poter visualizzare in Grafana i dati provenienti dalla piattaforma semantica SEPA (*SPARQL Event Processing Architecture*). Il SEPA mette a disposizione un particolare meccanismo di pubblicazione e sottoscrizione che ha reso possibile creare un'architettura per la visualizzazione *real-time* degli eventi in cui il *plugin* fa da ponte tra ciò che reperisce i dati, il SEPA, e l'ambiente di analisi e rappresentazione, Grafana.

Nel secondo capitolo verranno presentate tutte le tecnologie impiegate nell'implementazione del *plugin* di Grafana. Successiva-

mente, nel terzo capitolo saranno mostrati i principali *step* dello sviluppo del *plugin*, a seguire sarà descritto come è stato possibile rendere *open source* il *plugin* creato e verrà mostrato un esempio applicativo. Infine, nell'ultimo capitolo saranno suggerite possibili future implementazioni riguardanti l'ulteriore sviluppo del *plugin*.

Capitolo 2

Nozioni e strumenti principali

2.1 Grafana

Grafana "è uno strumento *open source* di *query*, visualizzazione e *alert* di serie di dati temporali sviluppato da Torkel Ödegaard nel 2014" [11]. Tale piattaforma supporta TSDBs (*Time Series Databases*)[1] come Graphite [41], Prometheus [7], OpenTSDB [8], InfluxDB [27] e database più classici come MySQL [37] e PostgreSQL [20]. Grafana non prevede un meccanismo di immagazzinamento dei dati ma solo di visualizzazione, ovvero la rappresentazione grafica dei parametri, degli eventi e dei logs[2].
Il software, utilizzato da numerose aziende tra cui Stack Overflow, PayPal o Tripadvisor [32], rientra tra i più conosciuti *Open Source Monitoring Tools*. La soluzione proposta è real-time e può prevenire eventi critici, tempi di inattività o migliorare la gestione del flusso di lavoro attraverso indicatori chiave di prestazione come, ad esempio, il consumo della CPU, l'utilizzo della RAM, lo spazio su disco, il traffico di rete [35].
Grafana può adattarsi a molteplici ambiti applicativi nei quali i

[1] Database specializzati nell'archiviazione e l'interrogazione di *time series data*, ossia dati costituiti da una serie di *timestamp* (o marche temporali) e valori corrispondenti [9]

[2] Record di eventi passati

computer, integrati con il mondo fisico attraverso dei sensori, registrano decine di migliaia di parametri. Di conseguenza, la valutazione dello stato di salute di questi sistemi fisici coinvolge un enorme flusso di dati per determinare efficienza, redditività e prestazione [43].
La piattaforma è in grado di gestire grandi volumi di dati permettendone la visualizzazione in grafici, diagrammi e segnali di allarme [35], che notificano potenziali problemi o anomalie. Questo ambiente risulta una scelta interessante grazie ad alcuni aspetti [43] come:

- la velocità: Grafana si dimostra performante nell'interrogazione delle sorgenti di dati;
- l'estendibilità: attraverso plugins per pannelli e *data source*;
- la versatilità: non è legato a una particolare tecnologia di *database*;
- è gratuito: è disponibile come software *open source* ma anche come Grafana Cloud e Grafana Enterprise, che tuttavia è a pagamento.

Grafana è scritto in linguaggio di programmazione Go (creato da Google) e Node.js LTS, e il suo codice sorgente è pubblico su GitHub [30].
Nel dettaglio, una tipica architettura dell'infrastruttura di Grafana ha questi componenti:

- *Data Producer*: elemento che genera i dati da visualizzare. Ad esempio: macchina virtuale o sensore;
- *Data Source*: database che, con modalità diverse (polling o interrupt), riceve dati dal *data producer*;
- *Grafana Dashboard*: elemento di *fronted* che, tramite una *query*, interroga il *data source*. Il risultato della *query* viene visualizzato nella dashboard.

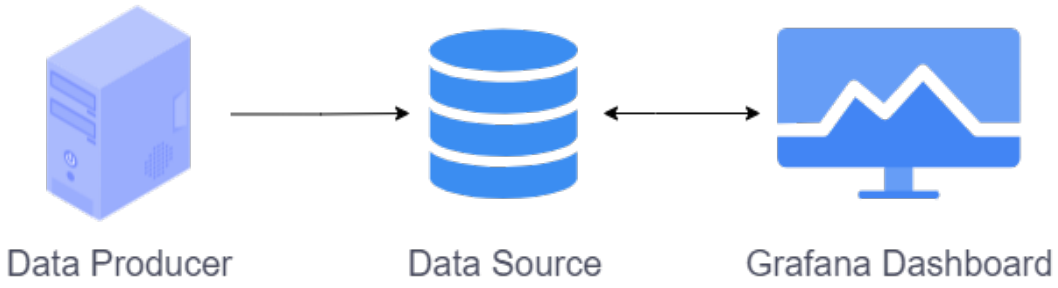

Figura 2.1: Architettura dell'infrastruttura Grafana composta da *Data Producer*, *Data Source* e dal sistema di visualizzazione

2.2 Windows PowerShell

Windows PowerShell è una *shell*, ovvero una componente fondamentale di un sistema operativo. Questo elemento ha lo scopo di mettere l'amministratore nelle condizioni di gestire tutti gli aspetti di un sistema Windows e le applicazioni che vengono eseguite [34]. Alcune operazioni fondamentali vengono presentate di seguito, riportando l'alias del comando:

- `cd -Path`: viene utilizzato per cambiare la directory (o cartella) di lavoro. La *path* indica il percorso specifico o indirizzo della cartella;
- `ls`: è utile per ottenere l'elenco di tutti i file e le cartelle all'interno di una cartella;
- `clear`: pulisce il display di PowerShell;
- `cat`: restituisce il contesto di un file.

2.3 Docker

Docker è una piattaforma aperta per lo sviluppo, la fornitura e l'esecuzione di applicazioni [15]. Permette di impacchettare il programma in un *container*, in modo da poter distribuire rapidamente il software. Un *container* è un'unità standard che ospita il codice

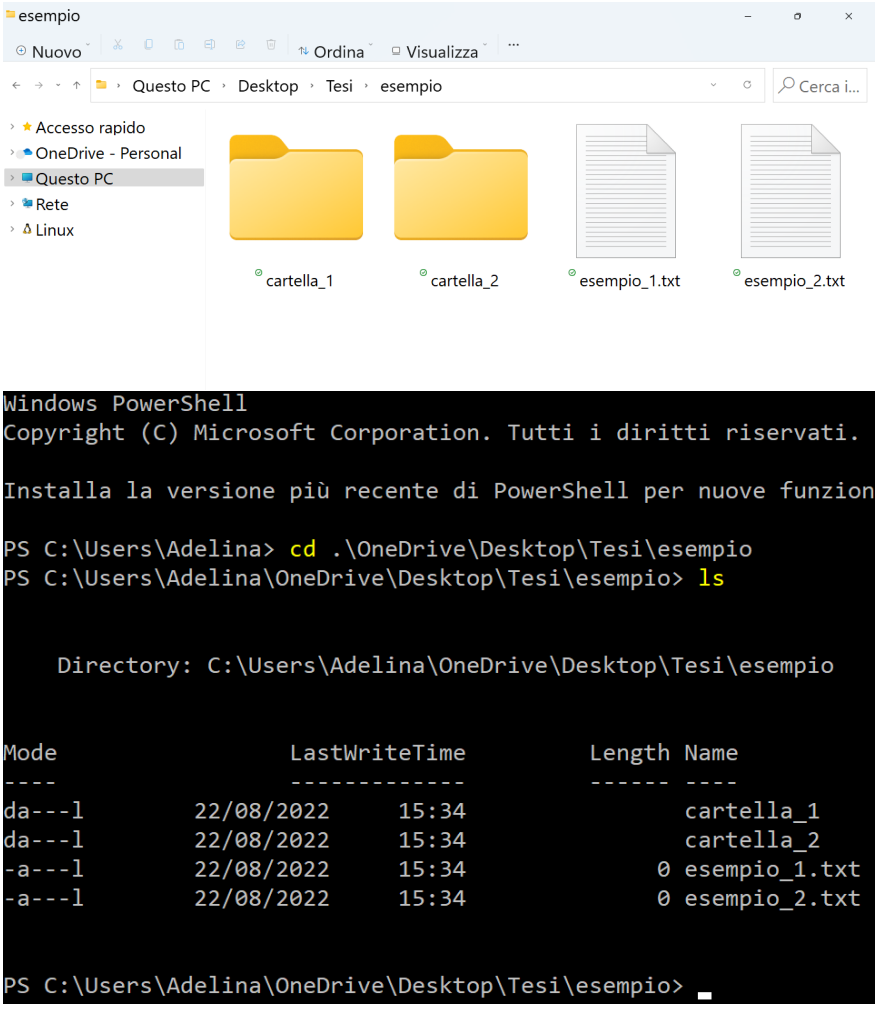

Figura 2.2: Esempio *directory* e Windows PowerShell. Come si può osservare il comando `cd` su Windows PowerShell permette di navigare fino alla cartella `esempio`, riportata nella prima figura, e `ls` di avere tutti gli elementi che si trovano al suo interno

sorgente e tutte le sue dipendenze in modo che possa essere eseguito in un qualsiasi ambiente informatico. Un *container* virtualizza il sistema operativo di un *server* come una macchina virtuale virtualizza la parte hardware (Figura 2.3). Un'immagine, in Docker, è un modello *read-only* con le istruzioni per la creazione di un *container*.

Docker utilizza un'architettura client-server. Il *client* comunica con il *daemon* Docker, che si occupa della costruzione, esecuzione e distribuzione dei *container*.
I comandi principali sono:

- `docker build [OPTIONS] PATH | URL | -`

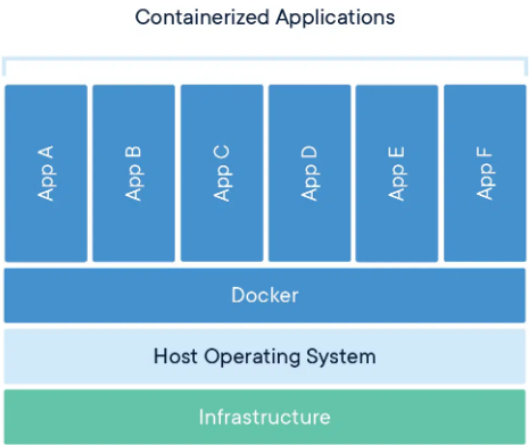

Figura 2.3: *Containers* Docker, fonte: `https://www.docker.com/resources/what-container/`. Come si può osservare i *containers* condividono il kernel del sistema operativo sull'*host*

- `docker push [OPTIONS] NAME[:TAG]`
- `docker pull [OPTIONS] NAME[:TAG|@DIGEST]`
- `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`

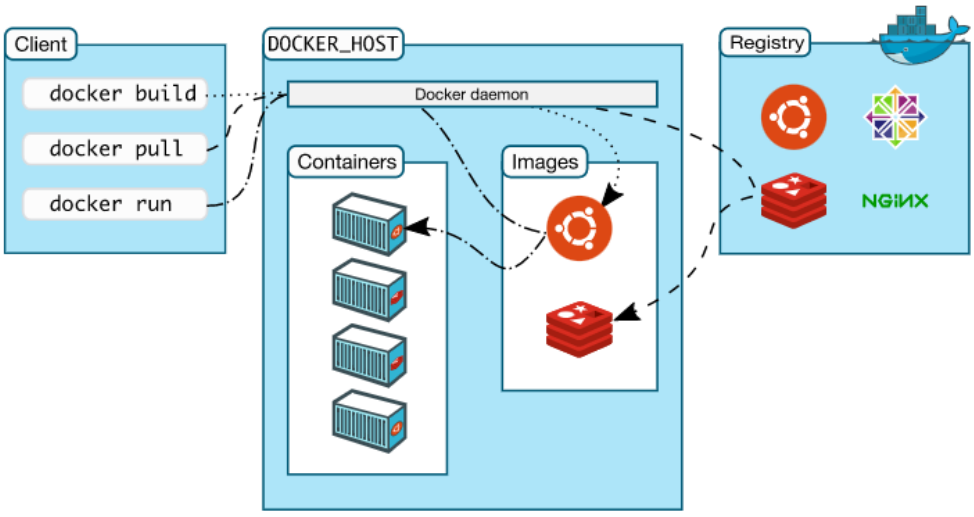

Figura 2.4: Architettura Docker, fonte: `https://docs.docker.com/get-started/overview/`

Il comando `docker build` [22] crea l'immagine Docker a partire da un Dockerfile e un contesto, ovvero l'insieme di files che si trovano nella path o nell'URL specificato. Dopo aver eseguito `docker`

`build` è possibile utilizzare `docker push` [25] per inviare l'immagine o la repository ad un registro, ovvero un luogo di archiviazione. Docker Hub[23], per esempio, è un registro pubblico. Il comando `docker pull`[24] permette di scaricare un'immagine da un registro. Infine, `docker run` [26] crea un nuovo *container* e lo avvia immediatamente.
Il montaggio di un volume è un meccanismo impiegato per lo *storage* di dati persistenti[3] generati e utilizzati dai *container* così da conservare i dati indipendentemente dal ciclo di vita del *container*. Collegare un volume permette di salvare i dati da un *container* e trasmetterli o scambiarli da uno all'altro. Per creare un *data volume* all'interno di un *container* è sufficiente spuntare il flag `-v` nel comando `docker create` o `docker run`. I volumi vengono quindi memorizzati in una parte del *filesystem host* gestito da Docker. In questo elaborato viene impiegato il meccanismo di creazione di un volume per salvare le modifiche apportate al contenuto del *container* di Grafana (Capitolo 3.1).

2.4 GitHub

GitHub [19] è una piattaforma per lo sviluppo collaborativo di progetti. É un servizio di *hosting* di *repository* sul cloud che permette agli utilizzatori di cooperare a distanza sulla stessa attività. Data la facilità di utilizzo, GitHub viene impiegato per svariati tipi di progetti, ma è principalmente adoperato da sviluppatori, che caricano il codice sorgente dei loro programmi e lo rendono scaricabile dagli utenti. É possibile, infatti, clonare una *repository* sul proprio computer, ovvero crearne una copia, e, successivamente, apportare delle modifiche al codice scaricato.

[3]La persistenza, in termini generali, indica la capacità di permane costantemente nel tempo. In ambito informatico, un dato persistente è un dato che sopravvive all'esecuzione del processo che lo ha creato, ovvero si conserva allo spegnimento del computer. Ciò è possibile mediante salvataggio dei dati in uno *storage* non volatile [33]

2.5 SPARQL Event Processing Architecture

2.5.1 Internet of Things

L'IoT (*Internet of Things*) è una rete di elementi interconnessi che possono comunicare e scambiare informazioni.

Figura 2.5: Rappresentazione dell'*Internet of Things*

L'IoT si propone l'obiettivo di rendere un sistema quanto più indipendente dall'interazione umana nel generare e gestire i dati, riducendo drasticamente l'errore dettato dalla minore accuratezza e attenzione umana. I computer possono quindi monitorare, rilevare e comprendere l'ambiente di lavoro grazie a diverse tecnologie, come RFID (*Radio Frequency Identification*) e sensori [6].

2.5.2 Semantic Web

Il *Semantic Web* (SW) è stato creato con l'idea che i computer potessero essere in grado di interpretare e usare le informazioni disponibili sul Web, fino ad ora accessibili principalmente solo all'uomo [10]. Il *Semantic Web* si basa sulle strutture ontologiche, che descrivono semanticamente i metadati di un dominio e forniscono dati interpretabili anche dalle macchine. Le ontologie costituiscono la base per la rappresentazione e condivisione standardizzata delle informazioni. Inoltre, mediante sistemi di *reasoning* permet-

tono di inferire informazioni non esplicite e di interrogare la base di conoscenza mediante *query* SPARQL [44] simili a quelle SQL. La semantica delle ontologie si basa sulla tripla RDF *(Resource Description Framework)* e sul linguaggio che le descrive, ossia l'OWL *(Web Ontology Language)* [14]. L'architettura del Semantic Web si presenta come un'insieme di livelli, come mostrato nella Figura 2.6 riportata di seguito.

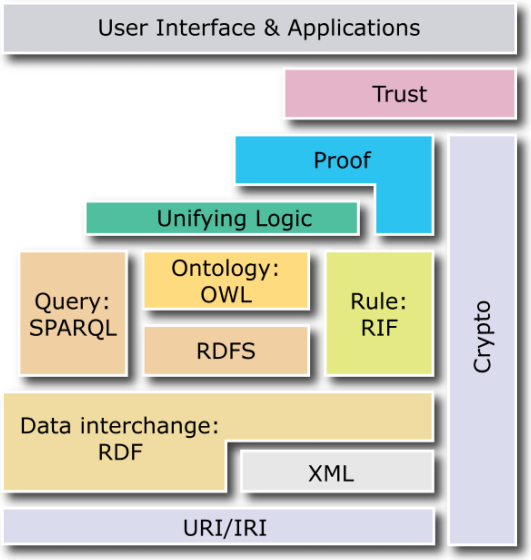

Figura 2.6: Architettura a livelli del Web Semantico, fonte: `https://it.wikipedia.org/wiki/Web_semantico`

L'RDF è uno standard basato su sintassi XML [5] e fornisce un modello per descrivere delle risorse, dove una risorsa è un qualsiasi oggetto identificabile con un URI. Mediante URI si identificano univocamente nelle ontologie le entità di uno specifico dominio di rappresentazione. La componente principale dell'RDF è la tripla. Una rappresentazione grafica di uno *statement* (o tripla) è di tipo Soggeto-Predicato-Oggetto (Figura 2.7).

La sorgente della relazione è il soggetto, l'arco il predicato e la destinazione della relazione l'oggetto.
Il soggetto o l'oggetto di una tripla possono essere anche il soggetto e/o l'oggetto di un'altra tripla, formando così una struttura chiamata grafo RDF.

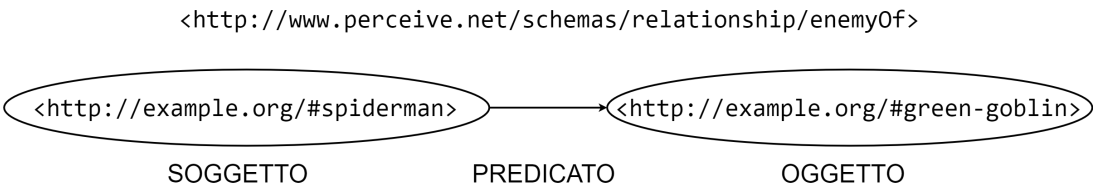

Figura 2.7: Tripla Soggetto-Predicato-Oggetto

2.5.3 SPARQL 1.1 Protocol

SPARQL *Protocol and RDF Query Language* (SPARQL) [21] è il linguaggio standard per l'interrogazione di dati RDF [39]. In questo caso le richieste possono essere di tipo *query*, per l'estrazione dei dati ricercati, o *update*, per apportare delle modifiche alla base di conoscenza.
La piattaforma semantica SEPA (Capitolo 2.5.4) sfrutta lo SPARQL 1.1 *Secure Event (SE) Protocol* [2], che estende lo SPARQL 1.1 *Protocol*, per supportare le *subscription* e garantire connessioni più sicure [2]. Questa estensione mette a disposizione, oltre alle due operazioni di *Update* e *Query*, una terza operazione, che è quella di *Subscribe*.

2.5.4 SEPA

Il SEPA (SPARQL *Event Processing Architecture*) [1] è un'architettura *software* che promuove e supporta lo sviluppo di applicazioni e servizi interoperabili, distribuiti e dinamici [42].

Figura 2.8: SPARQL Event Processing Architecture

Il SEPA offre un meccanismo di pubblicazione e sottoscrizione che permette agli *users* di ricevere i cambiamenti avvenuti in uno specifico contesto, ovvero l'aggiunta o la rimozione di dati, e inviare nuove informazioni. Tale piattaforma archivia i dati in modo semantico in grafi RDF e genera eventi che corrispondono ad "un

qualsiasi cambiamento in un archivio RDF" [42]. Il SEPA si compone dei seguenti elementi: lo SPARQL *Secure Event (SE) Protocol Service* (noto come SEPA broker), lo SPARQL 1.1 *Secure Event (SE) Protocol* [2] e lo SPARQL 1.1 *Subscribe Language* [3].

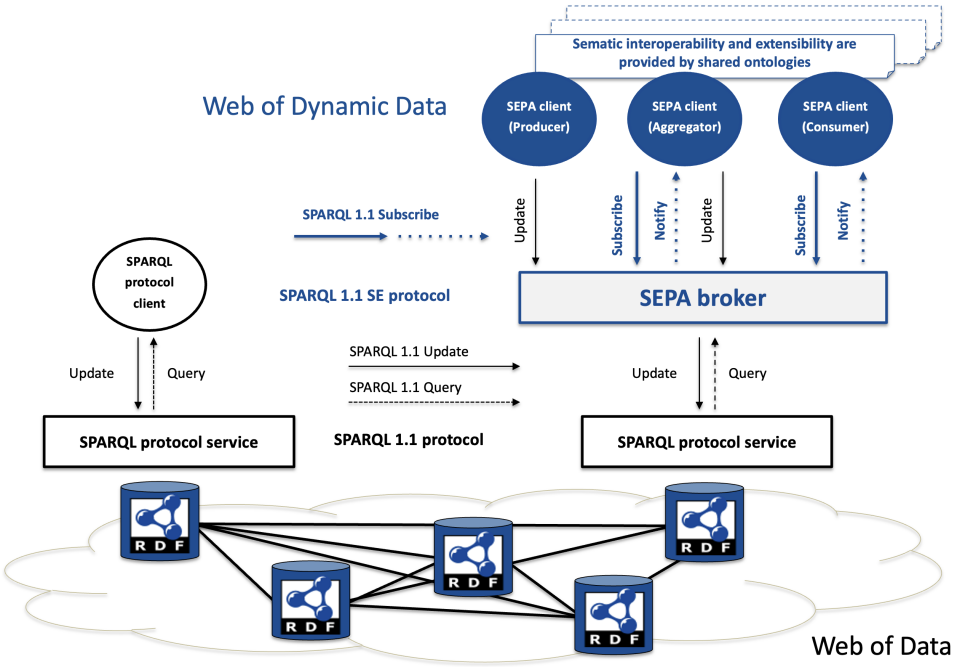

Figura 2.9: SPARQL *Event Processing Architecture (SEPA)*, fonte: `http://mml.arces.unibo.it/TR/sepa.html` . I *client* interagiscono con il SEPA *broker* attraverso lo SPARQL 1.1 *SE Protocol*, ossia mediante le richieste di *Query*, *Update* e *Subscribe*. Il SEPA *broker* sfrutta lo SPARQL 1.1 *Protocol* per relazionarsi con lo SPARQL *Protocol service*

Il SEPA broker è la componente server del SEPA e implementa i meccanismi di pubblicazione e sottoscrizione. Gli standard necessari al SEPA per comunicare sono: HTTP GET [48] per le *query*, POST per gli *updates* e *WebSocket* [4] per le *subscription* [46]. Una sottoscrizione SPARQL agisce come una *query* persistente. Il SEPA ad ogni evento restituisce solo le variazioni che avvengono nel grafo, in particolare sotto forma di cambiamenti nei risultati della query, ovvero i *bindings* aggiunti o rimossi. Questo prevede un consistente risparmio nel traffico di rete e nei costi di elaborazione dal

[4]Il protocollo WebSocket fornisce un canale di comunicazione bi-direzionale full-duplex [26]. Questa funzionalità consente una comunicazione interattiva tra il server e il client, senza che sia necessario il *polling* richiesto nelle implementazioni basate su HTTP

lato *client*, ossia non è necessario confrontare nuovi e vecchi dati per estrarre le modifiche avvenute. Nel Capitolo 3.2.1 è presente un esempio applicativo del meccanismo di sottoscrizione.

Capitolo 3

Implementazione di un data source plugin per il SEPA

In questo capitolo verrà mostrata l'implementazione di un *data source plugin*, compatibile con la piattaforma semantica SEPA, in grado di effettuare una *subscription* per ricevere in modo asincrono gli eventi. Infine, i dati vengono inviati alla *dashboard* di Grafana in modo che essi siano visualizzati in *real-time*.

3.1 Ambiente di lavoro

Innanzitutto, è necessario configurare un ambiente di lavoro adatto allo sviluppo di un *plugin*. Una scelta vantaggiosa può essere quella di utilizzare la piattaforma Docker (Capitolo 2.3), così da non dover installare il software Grafana in locale. Per autorizzare un *plugin unsigned* [31] occorre configurare la modalità *developement*, in caso contrario *plugins* non firmati non saranno supportati dall'applicazione. Inoltre, per salvare dati persistenti generati e utilizzati dal *container*, è conveniente montare un volume nel *container* (si faccia riferimento al Capitolo 2.3).

```
docker run -d -p 3000:3000 -e "GF_DEFAULT_APP_MODE=development" -
    v "[FOLDER_PATH]/grafana:/var/lib/grafana" --name=grafana
    grafana/grafana
```

Una volta eseguita l'istruzione riportata sopra (comando 3.1) su Windows PowerShell (Capitolo 2.2), Grafana sarà accessibile alla porta 3000.

3.2 Creazione del plugin

Grafana-toolkit [36] è una CLI (*Command Line Interface*) [1] per lo sviluppo di *plugins*. L'obiettivo è quello di mettere a disposizione degli sviluppatori uno strumento che permetta loro di concentrarsi sul codice dei *plugins* piuttosto che sulla realizzazione della configurazione richiesta per svilupparli. Il comando di seguito riportato crea un *plugin* da un *template* [28].

```
npx @grafana/toolkit plugin:create datasource-plugin
```

Listing 3.1: Comando per la generazione di un *data source plugin* a partire da un *template*

Grafana supporta tre tipi di *plugins* [31]:

- *plugins* per pannelli: permettono di visualizzare i dati restituiti da una *query* ad un *data source*;
- *plugins* per *data sources*: vengono utilizzati per comunicare con fonti di dati esterne;
- *plugins* per app: sono una combinazione dei due soprastanti. Offrono un'esperienza di monitoraggio personalizzata.

La scelta, per questa tesi, è stata quella di sviluppare un *data source plugin*, poiché consente di connettersi con un *data source* esterno e restituire i dati in un formato [29] comprensibile alla piattaforma. Mediante i *plugins*, Grafana permette l'analisi di fonti eterogenee di dati. La sua efficace versatilità supera il confronto con altre applicazioni *software open source* per la visualizzazione

[1] L'interfaccia minima che un *software* fornisce all'utente [12]

dei dati, ad esempio Superset [18]. Superset richiede che le informazioni si trovino all'interno di un database SQL come MySQL [37], PostgreSQL [20] ed altri [17]. Dunque, per poter visualizzare dati RDF è necessario che prima essi vengano reindirizzati all'interno di un database SQL. Con Grafana, invece, le informazioni in RDF, ottenute con la *subscription* al SEPA, vengono fatte confluire dal *plugin* direttamente in un *data frame* [29] comprensibile alla piattaforma. Come mostrato in Figura 3.1, nel caso di Grafana non si ha bisogno di una manipolazione intermedia dei dati tra il SEPA e il sistema di visualizzazione. La gestione della *subscription* e del *data frame* sono approfonditi nel Capitolo 3.2.1.

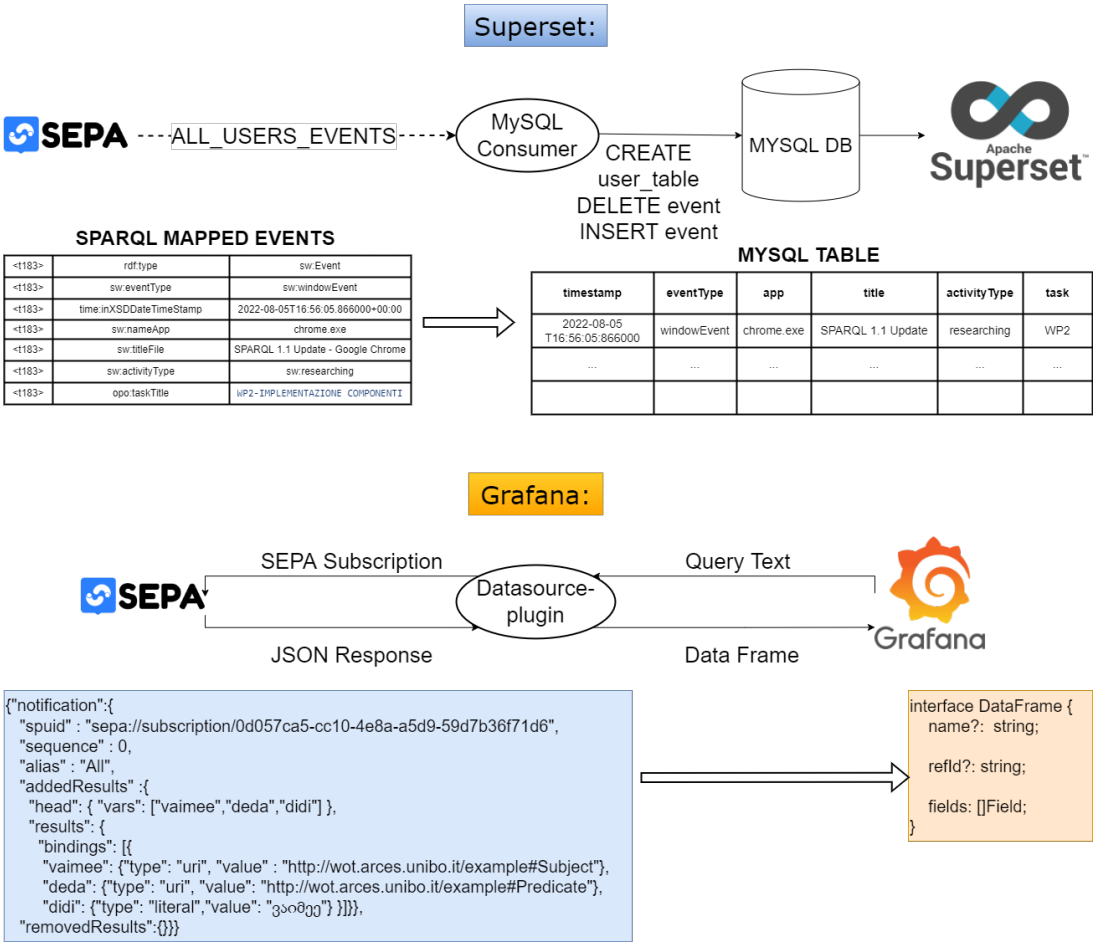

Figura 3.1: Confronto tra Superset e Grafana

Il *plugin*, come è evidente nella Figura 3.1, è l'elemento che si pone come intermediario tra l'architettura SEPA e la piattaforma di visualizzazione Grafana. Grafana richiede che i risultati della

query provenienti da un qualsiasi *data source* siano consolidati in una struttura chiamata `data frame` [29]. Non è possibile collegare direttamente il SEPA a Grafana poiché la risposta alla *subscription* dal SEPA non ha un formato come quello del `data frame` ma è un file JSON (*JavaScript Object Notation*)[2]. Di conseguenza si utilizza un *plugin* per incanalare i dati provenienti dal SEPA in un formato adatto per Grafana.
Di seguito viene descritto il funzionamento del *plugin* a partire dall'avvio di Grafana fino alla visualizzazione dei dati nella *dashboard*. Per prima cosa Grafana scansiona la *directory* del *plugin* (Figura 3.3) alla ricerca del file `plugin.json` poiché contiene tutte le informazioni fondamentali sul *plugin*. Successivamente, Grafana carica il file `module.ts` che espone la tipologia di *plugin* che si sta implementando [28]. A questo punto un utente di Grafana potrebbe voler creare un *panel* in una *dashboard*[3] e selezionare come *data source* il *plugin* caricato. L'utente per poter visualizzare una specifica informazione può scrivere una *query* per interrogare la base di dati. Il testo della *query* viene inviato da Grafana al *plugin* in un JSON. Se l'utente non inserisce nessuna *query* il *plugin* ne utilizza una di *default*. A questo punto il *plugin* usa la *query* ricevuta, oppure quella di *default*, per effettuare una sottoscrizione al SEPA. Da questo momento in poi il *plugin* riceverà i JSON di risposta dal SEPA da cui estrarre le informazioni da visualizzare. Infine, per poter inviare i dati a Grafana il *plugin* imposta un `data frame`. In sostanza, un *data frame* è una raccolta di *fields*, ogni *flieds* corrisponde a una colonna e, a sua volta, è costituito da una raccolta di valori [29]. Un esempio di `data frame` viene riportato in Figura 3.2. Quando i nuovi eventi arrivano al *plugin* i dati vengono inseriti nel `data frame` e inviati a Grafana perché siano visualizzati nella *dashboard*. Mediante questi passaggi è stato possibile realizzare il nuovo *plugin* designato per Grafana.

[2]JSON è un formato di testo per l'archiviazione e il trasporto di dati. É autodescrittivo e facile da capire [49]. Un oggetto JSON può essere definito come un insieme di proprietà

[3]La *dashboard* è l'ambiente di visualizzazione di Grafana ed è costituita da *panels* indipendenti tra loro

DATA FRAME:

time	temperature
2020-01-02 03:04:00	25.0
2020-01-02 03:05:00	27.0
2020-01-02 03:06:00	25.0

Figura 3.2: La tabella rappresenta un `data frame` con due *fields*, tempo e temperatura [29]

3.2.1 Implementazione del plugin

Come anticipato nel Capitolo 3.1 gli obiettivi per l'implementazione del *plugin* sono principalmente tre:

1. effettuare una *subscription* al SEPA affinché il *plugin* riceva in maniera asincrona i nuovi eventi;
2. impostare un `data frame` per accogliere i dati da inviare a Grafana;
3. incanalare i dati nel `data frame` e inviarli alla *dashboard*.

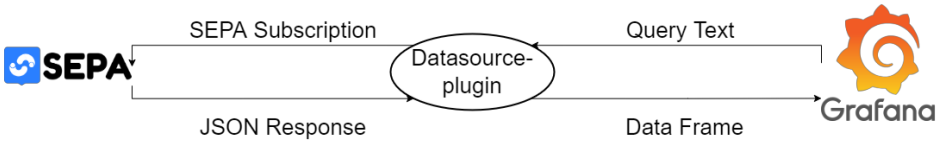

Figura 3.3: Architettura della connessione SEPA - *datasource-plugin* - Grafana. Partendo da sinistra, Grafana invia al *plugin* la *query* impostata dall'utente, il *plugin* effettua la *subscription* al SEPA e, successivamente, invia la risposta ottenuta dal SEPA a Grafana

Innanzitutto, è necessario che il *plugin* sia visualizzabile da Grafana, a questo scopo deve essere riavviato il *container* Docker. Un altro passaggio preliminare è quello di aggiungere al progetto la *repository* di GitHub (Capitolo 2.4) `arces-wot/SEPAjs`[4] affinché

[4] `https://github.com/arces-wot/SEPA-js`

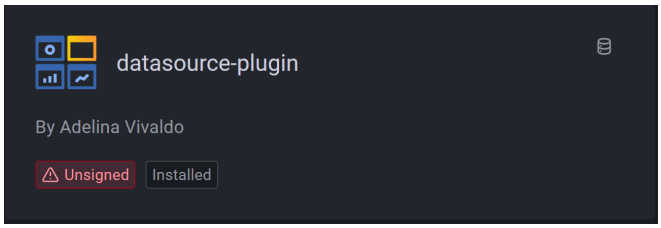

Figura 3.4: Sezione *Plugins* di Grafana

sia possibile utilizzare le funzioni in JavaScript che questa libreria fornisce per la creazione di un *client* SEPA.

```
yarn add @arces-wot/sepa-js
```

Listing 3.2: Comando per l'aggiunta della libreria `arces-wot/SEPAjs` al progetto

Per implementare il *plugin* è necessario modificare il suo codice che si trova nel *folder plugins* dentro alla cartella di Grafana. Al suo interno ci sono i files generati dal *template*, fornito da *grafana-toolkit*, tra cui `module.ts`, `plugin.json`, `datasource.ts`, `ConfigEditor.tsx`, `QueryEditor.tsx` e `type.ts` (Listing 3.3).

```
grafana
|__ alerting
|__ csv
|__ png
|__ grafana.db
|__ plugins
    |__ datasource-plugin
          |__ .github
          |__ coverage
          |__ dist
          |__ node_modules
          |__ src
              |__ img
              |__ ConfigEditor.tsx
              |__ datasource.ts
              |__ module.ts
              |__ plugin.json
              |__ QueryEditor.tsx
              |__ types.ts
```

Listing 3.3: *Directory* della cartella grafana

In questa sottosezione si farà riferimento al codice scritto in linguaggio TypeScript [45] riportato nel Listing 3.4. Tale frammento

di codice è il *core* del *plugin* e si occupa di portare a termine gli obiettivi di *subscription* al SEPA, impostazione del `data frame` e invio dei dati. Di seguito verranno analizzati nel particolare gli elementi e le funzioni presenti nel codice.

Il primo termine della prima linea di codice è `query`. `Query` è un metodo della classe `DataSource`[5]. Il file `datasource.ts` (Listing 3.3) fornisce già la classe `DataSource` e il metodo `query`, di conseguenza basterà occuparsi del solo sviluppo del metodo affinché esegua i comandi desiderati. Come è stato anticipato nel Capitolo 3.1, *grafana-toolkit* mette nelle mani dello sviluppatore un *template* che gli permette di concentrarsi sullo sviluppo di singole parti del codice e non sulla configurazione globale. Il frammento di codice (Listing 3.4) riporta l'implementazione del metodo `query`.

```
query(options: DataQueryRequest<MyQuery>): Observable<
    DataQueryResponse> {
  const observables = options.targets.map((target) => {
  const query = defaults(target, defaultQuery);

  //DEFAULT_JSON CONFIGURATION CHANGED WITH VALUES TAKEN FROM THE
     GRAFANA FRONTEND:
  default_json.host = this.host;
  default_json.sparql11protocol.port = this.http_port;
  default_json.sparql11seprotocol.availableProtocols.ws.port =
    this.ws_port;

  return new Observable<DataQueryResponse>((subscriber)=>{
  const frame = new CircularDataFrame({
    append: 'tail',
    capacity: 1000,
   });

//------------------
//SEPA SUBSCRIPTION
//------------------
var i=0;
let client = new SEPA(default_json);
const sub = client.subscribe(query.queryText);

sub.on("subscribed", console.log)

```

[5]TypeScript è un linguaggio Object Oriented, quindi permette l'utilizzo delle classi. Una classe definisce un tipo di oggetto. I metodi sono funzioni contenute nelle classi.

```
25 sub.on("notification", (not: SparqlBindings) => {
26     var bindings = extract_bindings(not);
27
28     //FIELDS:
29     for (; i<1; i++){
30         frame.refId = query.refId;
31             for (var key in bindings[0]){
32     frame.addField({ name: key, type: FieldType.string });
33         }
34     }
35
36     //FRAME:
37     var jsonData: any = {};
38     bindings.forEach((element: any) => {
39     for (var key in element){
40       jsonData[key]=element[key].value;
41     }
42         frame.add(jsonData);
43   });
44
45     subscriber.next({
46               data: [frame],
47               key: query.refId,
48     });
49
50 });
51
52 sub.on("error",console.error);
53 });
54 });
55
56 return merge(...observables);
57 }
```

Listing 3.4: Metodo `query` della classe `DataSource`

A partire dalla riga 16 del codice (Listing 3.4) si scrivono le funzioni per la sottoscrizione e la gestione dei dati provenienti dal SEPA. Come prima cosa, la funzione `client.subscribe()` fornita dalla libreria `arces-wot/SEPAjs` permette al nuovo *client* SEPA, creato nella riga 20, di effettuare una *subscription*. In questo caso, il nuovo *client* è il *plugin*. Se la richiesta di sottoscrizione viene elaborata correttamente, il SEPA risponde con un messaggio come mostrato nel Listing 3.5 [3]. Il messaggio è un JSON e, una volta ricevuto dal *plugin*, è contenuto all'interno della variabile `not` (riga 25 del Listing 3.4). Una proprietà dell'oggetto *not* sono i *bindings*. I

bindings sono degli *array* che contengono un numero variabile di elementi che in quel momento condividono tutti la stessa struttura, ad esempio quella riportata nel Listing 3.6. Gli elementi dell'*array* sono i risultati della *query* e sono a loro volta dei JSON. Nell'esempio riporato nel Listing 3.5 l'*array* possiede un solo elemento.

```
{"notification":{
    "spuid" : "sepa://subscription/0d057ca5-cc10-4e8a-a5d9-59
   d7b36f71d6",
    "sequence" : 0,
    "alias" : "All",
    "addedResults" :{
      "head": { "vars": ["soggetto","predicato","oggetto"] },
      "results": {
        "bindings": [{
         "soggetto": {"type": "literal", "value" : "t263"},
         "predicato": {"type": "uri", "value": "http://
   hospital_example.org/name"},
         "oggetto": {"type": "literal","value": "Francesco"}
   }]},
    "removedResults":{}
    }
}
```

Listing 3.5: SEPA response

```
{"soggetto": {"type": "literal", "value" : "t263"},
"predicato": {"type": "uri", "value": "http://hospital_example.
   org/name"},
"oggetto": {"type": "literal","value": "Francesco"}}
```

Listing 3.6: Singolo elemento dell'array *bindings*

Una volta che il *plugin* acquisisce il messaggio di risposta del SEPA estrae i *bindings* con la funzione `extract_bindings()` (Listing 3.7).

```
//EXTRACT BINDINGS FROM JSON RESPONSE:
function extract_bindings(msg: SparqlBindings){
  var bindings=msg.addedResults.results.bindings;
  return bindings;
}
```

Listing 3.7: Funzione `extract_bindings()`

A questo punto il *plugin* ha già effettuato la *subscription*, ha ricevuto il messaggio dal SEPA e possiede i risultati della *query* dentro alla variabile `bindings` (riga 26 del Listing 3.4). Prima di poter

inviare i nuovi dati alla *dashboard* di Grafana è necessario impostare un formato comprensibile alla piattaforma. All'atto pratico consiste nel creare i *fields* e aggiungerli al `data frame`, di cui un esempio è riportato in Figura 3.2. Il *plugin* crea i *fields* con un ciclo *for* (riga 29 del Listing 3.4) e poi li aggiunge tramite la funzione `frame.addField()` al *data frame*.

```
1 { name : soggetto, type : FieldType.string }
2 { name : predicato, type : FieldType.string}
3 { name : oggetto, type : FieldType.string }
```

Listing 3.8: *Fields* ottenuti dal SEPA *response* e aggiunti al `data frame`

Come ultimo passaggio la funzione `frame.add()` prende i valori contenuti nell'oggetto `jsonData` e li dispone nei *fields* corrispondenti (riga 42 del Listing 3.4). Ad esempio, dal `jsonData` (Listing 3.9) il valore "t263" sarà aggiunto al *field* `soggetto`, l'URI "http://hospital_example.org/name" al *field* `predicato` e il nome "Francesco" al *field* `oggetto`.

```
1 {"soggetto":"t263","predicato":"http://hospital_example.org/name"
    ,"oggetto":"Francesco"}
```

Listing 3.9: Struttura del jsonData

soggetto	predicato	oggetto
t263	http://hospital_example.org/name	Francesco

Figura 3.5: Rappresentazione del un `data frame` con tre *fields* `soggetto`, `predicato` e `oggetto`

Infine, il `frame` viene inviato a Grafana per consentirne la visualizzazione. Si ricorda che una *subscription* restituisce solo i nuovi cambiamenti della base di conoscenza interrogata (Capitolo 2.5.4), quindi nel sistema di visualizzazione appariranno inizialmente i dati già presenti nel *database*, se ci sono, e poi, nel momento in cui avvengono, verranno aggiunti i nuovi eventi.

La funzione `client.subscribe()` (riga 20 nel Listing3.4) accetta in ingresso il parametro `query.queryText` che contiene l'espressione della *query* SPARQL. Il valore di `query.queryText` può essere direttamente modificato dall'utente nella fase di *set up* del *panel* su Grafana (Figura 3.6). La *query* `"select * where  ?s ?p ?o"` è quella che si è deciso di impostare come di *default*.

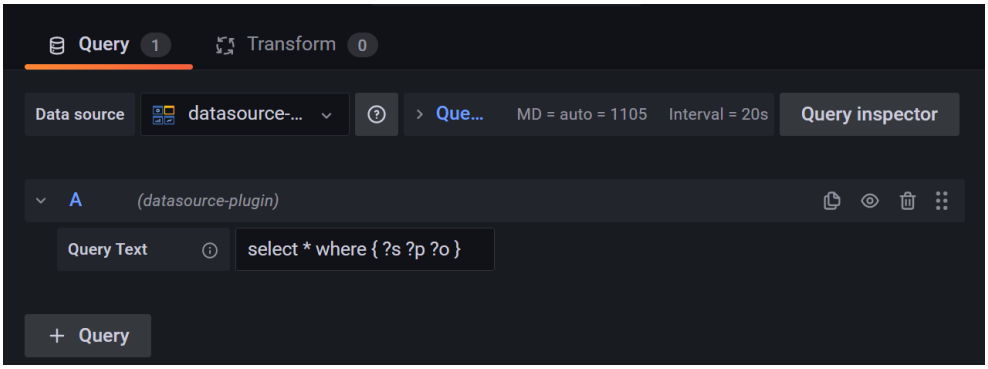

Figura 3.6: *Set up* del *Panel* di Grafana

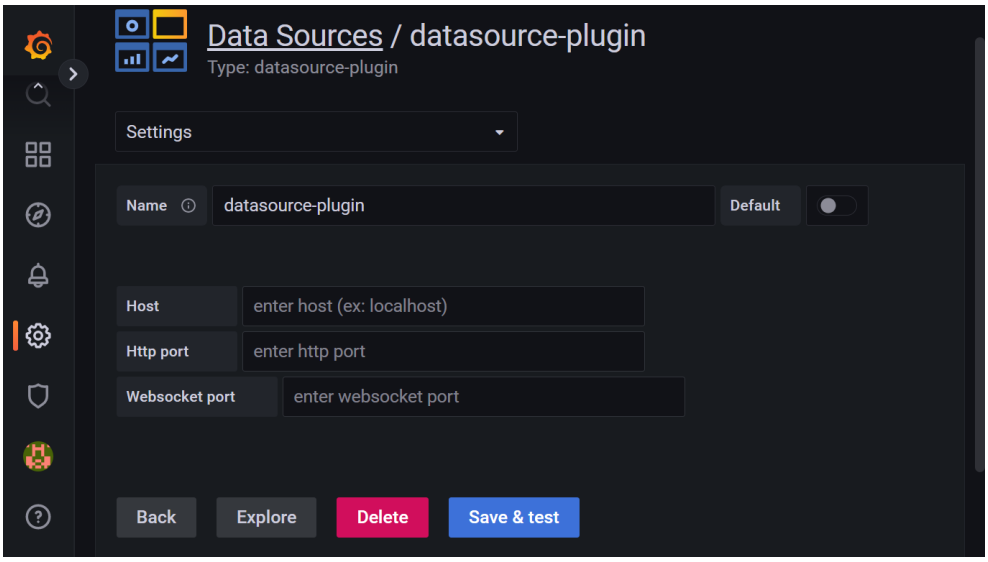

Figura 3.7: Configurazione *data source plugin*

Agendo sui files `ConfigEditor.tsx`, `QueryEditor.tsx` e `type.ts` è stato possibile modificare altri elementi di *frontend* come `Query Text` [28]. Ad esempio sono stati aggiunti i blocchi di `Host`, `HTTP port` e `Websocket port` nella sezione *Settings* del *plugin* (Figura 3.7). Questi elementi sono modificabili poiché possono variare in

base alla configurazione del SEPA sul proprio computer. É previsto di *default* che la porta per la comunicazione *HTTP* sia la 8600 e per la comunicazione *WebSocket* la 9600. Tali informazioni sono contenute nel `default_json` (Listing 3.10), utilizzato dalla funzione `SEPA()` per creare un nuovo *client* (riga 20 del Listing 3.4).

```
1  //DEFAULT JSON:
2  var default_json = {
3    "host": "localhost",
4    "oauth": {
5      "enable": false,
6      "register": "https://localhost:8443/oauth/register",
7      "tokenRequest": "https://localhost:8443/oauth/token"
8    },
9    "sparql11protocol": {
10     "protocol": "http",
11     "port": 8600,
12     "query": {
13       "path": "/query",
14       "method": "POST",
15       "format": "JSON"
16     },
17     "update": {
18       "path": "/update",
19       "method": "POST",
20       "format": "JSON"
21     }
22   },
23   "sparql11seprotocol": {
24     "protocol": "ws",
25     "availableProtocols": {
26       "ws": {
27         "port": 9600,
28         "path": "/subscribe"
29       },
30       "wss": {
31         "port": 9443,
32         "path": "/secure/subscribe"
33       }
34         }}}
```

Listing 3.10: JSON *default_json* contenete le informazioni di configurazione

3.3 Creazione del container Docker per la distribuzione del plugin

Come elemento conclusivo dell'elaborato si è deciso di creare un *container* Docker di Grafana con all'interno il *plugin* implementato per la connessione al SEPA. L'immagine del *container* è stata caricata su Docker Hub per renderla disponibile e utilizzabile da altri utilizzatori[6]. In questo modo basterà scaricare l'immagine e creare il corrispettivo *container* Docker sul proprio computer per avere un sistema di visualizzazione *realt-time* compatibile con l'architettura SEPA.

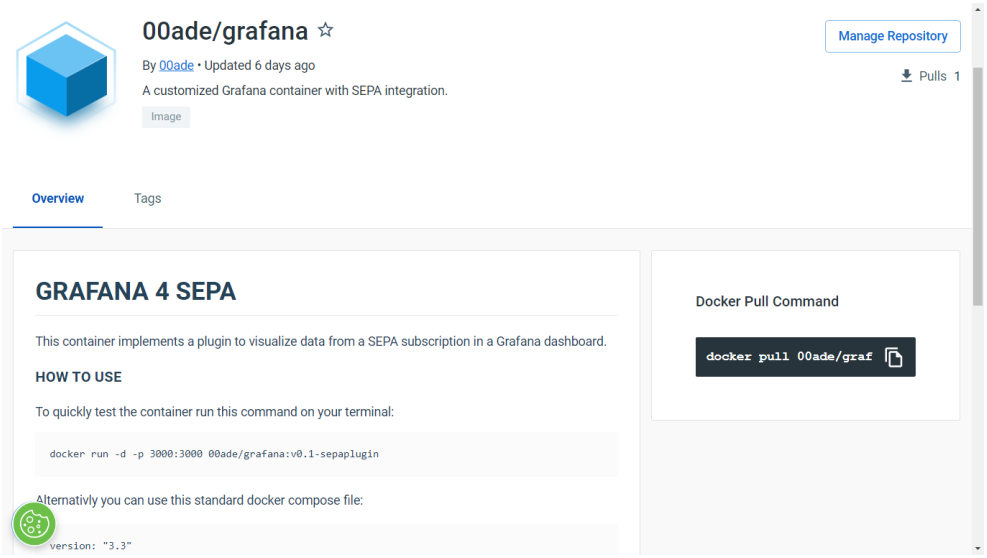

Figura 3.8: GRAFANA 4 SEPA

3.4 Esempio applicativo

Per comprendere la potenzialità di questa tesi viene riportato un esempio applicativo. L'esempio in Figura 3.9 simula un pannello di un centro di controllo di un ospedale. In alto, sotto alla scritta "*RED CODES*", sono riportati i nomi, i reparti e le patologie dei pazienti con codice rosso, così che siano immediatamente visibili.

[6] `https://hub.docker.com/r/00ade/grafana`

Al centro è possibile leggere l'elenco dei pazienti nei reparti 1, 2 e 3 e le corrispondenti malattie e codice colore. Infine, in basso sono mostrati l'insieme delle patologie, delle età e dei codici colore di tutti i pazienti.

Non appena un dato è inserito nel *data source* questo viene immediatamente visualizzato nella *dashboard* senza bisogno che il sistema richieda se ci sono nuovi dati.

Una *dashboard* è costituita da *panels* indipendenti tra loro (un *panel* è un riquadro) e ogni *panel* è il risultato di una specifica *query* SPARQL. Un esempio viene riportato nel Listing 3.11. Conoscendo i dati che si posseggono è possibile estrarre e visualizzare altre informazioni. Ad esempio, conoscendo le dottoresse e i dottori che lavorano in uno specifico reparto si potrebbe pensare di aggiungere un nuovo *panel* con il personale di riferimento per ogni reparto.

```
PREFIX schema: <http://schema.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX hosp: <http://hospital_example.org/>

SELECT ?Name ?Ward ?Disease WHERE {
  GRAPH <http://hospital_example.org/patients> {
    _:bnode rdf:type schema:Person;
            hosp:name ?Name;
            hosp:ward ?Ward;
            hosp:disease ?Disease;
            hosp:colour "red"
  }
}
```

Listing 3.11: *Query* SPARQL per l'estrazione dei codici rossi

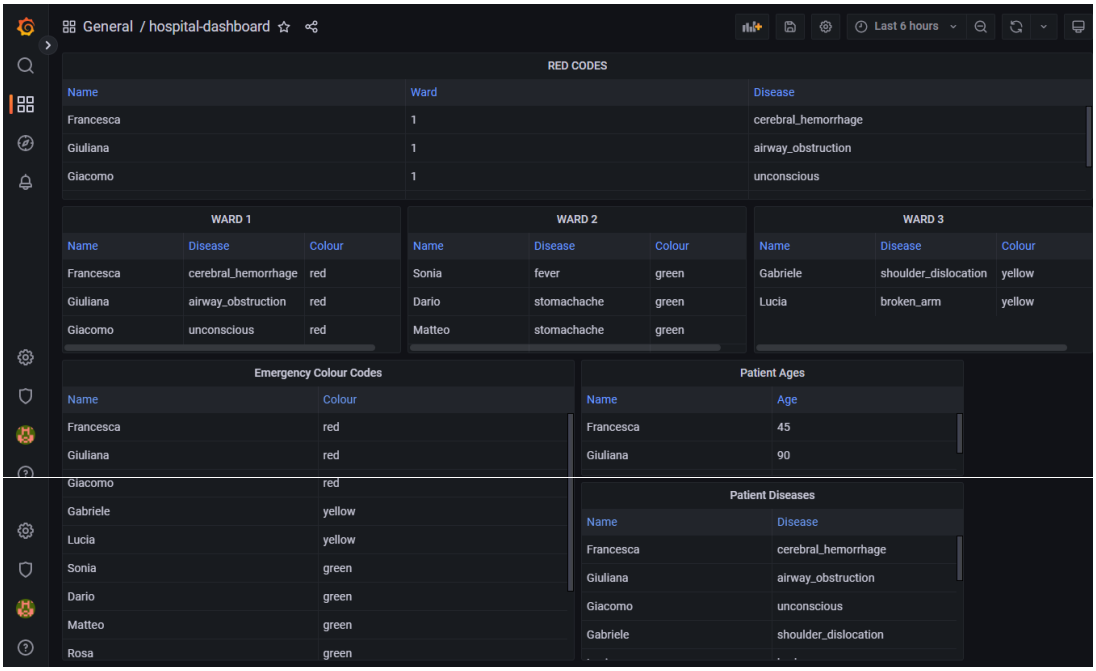

Figura 3.9: *Hospital dashboard* costituita dai *panels* RED CODES, WARD 1, WARD 2, WARD 3, Emergency Colour Code, Patient Ages e Patient Diseases.

Capitolo 4

Conclusioni

Questo elaborato è una soluzione innovativa che lascia pertanto spazio a future e nuove implementazioni. L'obiettivo di questa tesi è quello di mostrare come è stato ottenuto un sistema di visualizzazione *real-time* in grado di rappresentare dati RDF provenienti dalla piattaforma SEPA. I *plugins* in Grafana permettono di collegare *data source* molto grandi ed eterogenei tra loro, garantendo la visualizzazione dei dati indipendentemente dal formato con cui sono stati salvati, in questo caso grafi RDF. Alcune soluzioni analoghe, ad esempio *Superset*, oltre ad essere a *polling*, ossia non *real-time*, richiedono una manipolazione dei dati per cui si perde l'utilità stessa di averli salvati in grafi RDF. Grafana permette l'implementazione di diverse opzioni di visualizzazione che in questo elaborato non sono state tratte, poiché non rientravano nello scopo della tesi, ma possono essere approfondite. Tra le numerose modalità di visualizzazione disponibili, in questo elaborato è stata impiegata solo quella tabellare, come si può vedere in Figura 3.9, ma può essere interessante esplorarne di nuove come la forma "*Node Graph*" utilizzabile per la rappresentazione delle ontologie ad esempio. A livello tecnico, invece, sarebbe possibile sviluppare ulteriormente il *plugin*. In primo luogo, potrebbe essere implementata una gestione dei dati rimossi. Fino a questo momento la proprietà `removedResults` del JSON di risposta proveniente dal SEPA (Figura 3.5) è stata ignorata, quest'ultima contiene gli elementi che sono stati eliminati dalla base di conoscenza. Per potere togliere in *real-time* gli elemen-

ti eliminati anche dalla *dashboard* sarebbe necessario scansionare i dati che si possiedono, confrontarli con il dato eliminato estratto da `removedResults` e eliminare quel `frame`. Infine, per alleggerire il codice risulterebbe comodo poter acquisire dall'esterno il file JSAP (*JSON SPARQL Application Profile*)[1]. Per ora il codice contiene al suo interno la variabile `default_json` (Figura 3.10) che racchiude tutte le informazioni necessarie per la creazione di un *client* SEPA.

[1]Struttura che si basa sul JSON per la configurazione dei *client* che si connettono al SEPA, contiene parametri come HTTP *port* e *WebSocket port*

Bibliografia

[1] Luca Roffia et al. *arces-wot/SEPABins*. 2022. URL: https://github.com/arces-wot/SEPABins.

[2] Luca Roffia et al. *SPARQL 1.1 Secure Event Protocol*. Ott. 2018. URL: http://mml.arces.unibo.it/TR/sparql11-se-protocol.html.

[3] Luca Roffia et al. *SPARQL 1.1 Subscribe Language*. Ott. 2018. URL: http://mml.arces.unibo.it/TR/sparql11-subscribe.html.

[4] Luca Roffia et al. *SPARQL Event Processing Architecture (SEPA)*. Ott. 2018. URL: http://mml.arces.unibo.it/TR/sepa.html.

[5] Stefania Costantini Antonella Dorati. *Approcci al Web Semantico*. URL: http://www.websemantico.org/articoli/approcciwebsemantico.php.

[6] Kevin Ashton et al. «That 'internet of things' thing». In: *RFID journal* 22.7 (2009), pp. 97–114.

[7] Prometheus Authors. *Overview | Prometheus*. 2022. URL: https://prometheus.io/docs/introduction/overview/.

[8] The OpenTSDB Authors. *OpenTSDB Overview*. 2022. URL: http://opentsdb.net/overview.html.

[9] Andreas Bader, Oliver Kopp e Michael Falkenthal. «Survey and comparison of open source time series databases». In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017).

[10] Tim Berners-Lee, James Hendler e Ora Lassila. «The semantic web». In: *Scientific american* 284.5 (2001), pp. 34–43.

[11] Mainak Chakraborty e Ajit Pratap Kundan. «Grafana». In: *Monitoring Cloud-Native Applications*. Springer, 2021, pp. 187–240.

[12] Command Line Interface CLI. «Software User Interface Design». In: (1997).

[13] S. Decker, P. Mitra e S. Melnik. «Framework for the semantic Web: an RDF tutorial». In: *IEEE Internet Computing* 4.6 (2000), pp. 68–73. DOI: 10.1109/4236.895018.

[14] Li Ding et al. «Using ontologies in the semantic web: A survey». In: *Ontologies*. Springer, 2007, pp. 79–113.

[15] Inc. Docker. *Docker overview*. 2022. URL: https://docs.docker.com/get-started/overview/.

[16] Inc. Docker. *What is a container? - Docker*. 2022. URL: https://www.docker.com/resources/what-container/.

[17] The Apache Software Foundation. *Install Database Drivers*. 2022. URL: https://superset.apache.org/docs/databases/installing-database-drivers/.

[18] The Apache Software Foundation. *Welcome | Superset - The Apache Software Foundation!* 2022. URL: https://it.overleaf.com/project/622dffe24432f41c01ddd0dd.

[19] Inc. GitHub. *Hello World - GitHub Docs*. 2022. URL: https://docs.github.com/en/get-started/quickstart/hello-world.

[20] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2022. URL: https://www.postgresql.org/.

[21] The W3C SPARQL Working Group. *SPARQL 1.1 Overview*. 2013. URL: https://www.w3.org/TR/sparql11-overview/.

[22] Docker Inc. *Docker build*. 2022. URL: https://docs.docker.com/engine/reference/commandline/build/.

[23] Docker Inc. *Docker Hub*. 2022. URL: https://hub.docker.com/.

[24] Docker Inc. *Docker pull*. 2022. URL: https://docs.docker.com/engine/reference/commandline/pull/.

[25] Docker Inc. *Docker push*. 2022. URL: https://docs.docker.com/engine/reference/commandline/push/.

[26] Docker Inc. *Docker run*. 2022. URL: https://docs.docker.com/engine/reference/commandline/run/.

[27] InfluxData Inc. *InluxDB*. 2022. URL: https://www.influxdata.com/.

[28] Grafana Labs. *Build a data source plugin*. 2022. URL: https://grafana.com/tutorials/build-a-data-source-plugin/.

[29] Grafana Labs. *Data frames | Grafana documentation*. 2020. URL: https://grafana.com/docs/grafana/latest/developers/plugins/data-frames/.

[30] Grafana Labs. *grafana/grafana - GitHub*. 2022. URL: https://github.com/grafana/grafana.

[31] Grafana Labs. *Plugin management*. 2022. URL: https://grafana.com/docs/grafana/latest/administration/plugin-management/#plugin-catalog.

[32] Grafana Labs. *Success stories and case studies*. 2022. URL: https://grafana.com/success/.

[33] Wikipedia L'enciclopedia libera. *Persistenza (informatica)*. Lug. 2020. URL: https://it.wikipedia.org/wiki/Persistenza_(informatica).

[34] Karl Mitschke et al. *Windows PowerShell 2.0 Bible*. John Wiley & Sons, 2011.

[35] Bashir Mohammed, Mariam Kiran e Bjoern Enders. «NetGraf: An End-to-End Learning Network Monitoring Service». In: *2021 IEEE Workshop on Innovating the Network for Data-Intensive Science (INDIS)*. IEEE. 2021, pp. 12–22.

[36] Inc npm. *@grafana/toolkit - npm*. 2022. URL: https://www.npmjs.com/package/@grafana/toolkit.

[37] Oracle. *MySQL*. 2022. URL: https://www.mysql.com/it/.

[38] Axel Polleres Paula Gearon Alexandre Passant. *SPARQL 1.1 Update*. Mar. 2013. URL: https://www.w3.org/TR/sparql11-update/.

[39] Jorge Pérez, Marcelo Arenas e Claudio Gutierrez. «Semantics and complexity of SPARQL». In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), pp. 1–45.

[40] Victoria Pimentel e Bradford G. Nickerson. «Communicating and Displaying Real-Time Data with WebSocket». In: *IEEE Internet Computing* 16.4 (2012), pp. 45–53. DOI: 10.1109/MIC.2012.64.

[41] The Graphite Project. *Graphite Overview*. URL: https://graphiteapp.org/#overview.

[42] Luca Roffia et al. «Dynamic linked data: A SPARQL event processing architecture». In: *Future Internet* 10.4 (2018), p. 36.

[43] Eric Salituro. *Learn Grafana 7.0: A beginner's guide to getting well versed in analytics, interactive dashboards, and monitoring*. Packt Publishing Ltd, 2020.

[44] Andy Seaborne Steve Harris. *SPARQL 1.1 Query Language*. Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[45] Microsoft TypeScript. *TypeScript: JavaScript With Syntax For Types*. 2022. URL: https://www.typescriptlang.org/.

[46] Fabio Viola et al. «Mapping the NGSI-LD Context Model on Top of a SPARQL Event Processing Architecture: Implementation Guidelines». In: *2019 24th Conference of Open Innovations Association (FRUCT)*. 2019, pp. 493–501. DOI: 10.23919/FRUCT.2019.8711888.

[47] W3C. *RDF 1.1 Concepts and Abstract Syntax*. Feb. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[48] W3Schools. *HTTP Methods GET vs POST*. 2022. URL: https://www.w3schools.com/tags/ref_httpmethods.asp.

[49] W3Schools. *JSON - Introduction*. 2022. URL: https://www.w3schools.com/js/js_json_intro.asp.

Ringraziamenti

Gracias Mami, Papi, Didi y Tita porque si pase que me doy vuelta detrás de las espaldas para ver si ustedes están, ya sé que ustedes van a estar ahí

Grazie Anna, Ludo, Sara e Vale per aver condiviso con me anche gli anni dell'università. Grazie per le camminate, le chiamate, i caffè, le serate sul divano e le cene col vino bianco frizzante

Grazie alle mie amiche e amici a Sangio per esserci dimostrati che l'amicizia supera la distanza

Grazie alle mie amiche e amici a Cesena per aver reso questa città e questi tre anni indimenticabili

Grazie al Prof. Roffia, a Greg e ad Elisa per avermi seguito passo passo in questo percorso e avermi mostrato come è possibile lavorare in un ambiente dinamico e entusiasmante