POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Dashboarding per il monitoraggio di sistemi IoT

Relatori
prof. Antonio SERVETTI

Candidato
Michele MASIELLO

Anno Accademico 2023-2024

Indice

Elenco delle figure

Capitolo 1

Introduzione

La visualizzazione dei dati è uno degli strumenti essenziali all'interno di molteplici servizi per il monitoraggio di metriche o performance di un determinato sistema e/o sensore. I servizi di dashboarding rappresentano, quindi, strumenti essenziali per la loro visualizzazione e la loro analisi sottoforma di grafici. Questo perché è possibile unire in un'unica schermata più performance chiave in tempo reale per valutare l'efficienza del sistema in esame. Questi software sono utilizzati per generare visualizzazioni a partire da dati grezzi che derivano dal collegamento di fonti di dati. L'utilizzo di questi software porta a diversi vantaggi, alcuni esempi possono essere:

- Monitoraggio in tempo reale
- Controllo di dati provenienti da diverse fonti
- Viste semplificate per una migliore comprensione
- Automazione dei report

Esistono svariati software che offrono questo tipo di servizio.

Ogni software offre diverse funzionalità e integrazioni diverse con i database dal quale ricavare i dati da mostrare però tutti hanno in comune il modo di inserire il grafico o i grafici generati all'interno di applicazioni terze. Tutti usano il componente chiamato iframe.

L'iframe (inline frame) [1] è un elemento HTML che serve ad incorporare un'altra pagina web all'interno della pagina corrente. I suoi utilizzi sono molteplici: incorporamento di contenuti multimediali come animazioni o file audio, visualizzazione di pubblicità in pagine web o, nel caso preso in esame, la visualizzazione di grafici generati da software di dashboarding.

```
<iframe
        title={'titolo'}
        id={'id'}
        src={srcIframe}
        width="100%"
        height={800}>
 </iframe>
```

Figura 1.1: Codice iframe

Questa è la definizione di un iframe all'interno di React. Le proprietà sono le seguenti:

- title: nome che verrà dato all'iframe
- id: id da utilizzare per interagire con l'iframe
- src: rappresenta l'url del grafico da mostrare
- width/height: dimensioni che l'iframe dovrà avere

L'obiettivo di questa tesi è confrontare i diversi funzionamenti dei software di dashboarding. Nel prossimo capitolo verranno esplorati tre tra i principali software: Metabase, Kibana e Grafana. Il focus principale poi sarà su Grafana. Verranno esplorate nello specifico le sue funzionalità, come interagire con il suo l'iframe e come ampliare le sue funzionalità utilizzando i plugin. Per quanto riguarda l'iframe di Grafana, il focus riguarderà l'aumentare l'efficienza in termini di richieste effettuate e velocità di caricamento nel momento in cui si voglia aggiornare un iframe già presente in pagina. Per il raggiungimento di questo obiettivo verranno descritte due possibilità di implementazione andando prima a modificare il codice sorgente di Grafana e, successivamente, andando ad utilizzare i plugin che verranno descritti nei prossimi capitoli.

Capitolo 2

Sistemi di dashboarding

2.1 Metabase

Metabase [13] è uno strumento di dashboarding open source che permette di creare rapitamente dashboard interattive. È stato sviluppato per risultare semplice da utilizzare, consentendo l'utilizzo di query sui dati senza avere conoscenze specifiche di SQL. Le caratteristiche principali che possiede sono:

- Interfaccia semplice per la creazione di dashboard e grafici da visualizzare
- Compatibilità con vari database relazionali e non come MySQL, PostgreSQL e MongoDB
- Integrazione delle dashboard in applicazioni terze tramite l'utilizzo degli iframe

Metabase viene principalmente utilizzato per l'analisi dei dati e la generazione di report. Ma le sue implementazioni sono molteplici e variano a seconda delle esigenze dell'organizzazione che vuole utilizzarlo.

Dashboard in Metabase

Le dashboard in Metabase sono generate a partire dai dati presenti nei database che vengono associati ad esso. Per aggiungere un database all'interno di Metabase, si deve accedere al menù amministratore presente nelle opzioni in alto a destra della schermata iniziale, accedere alla sezione 'Database' e cliccare su 'Aggiungi un database'.

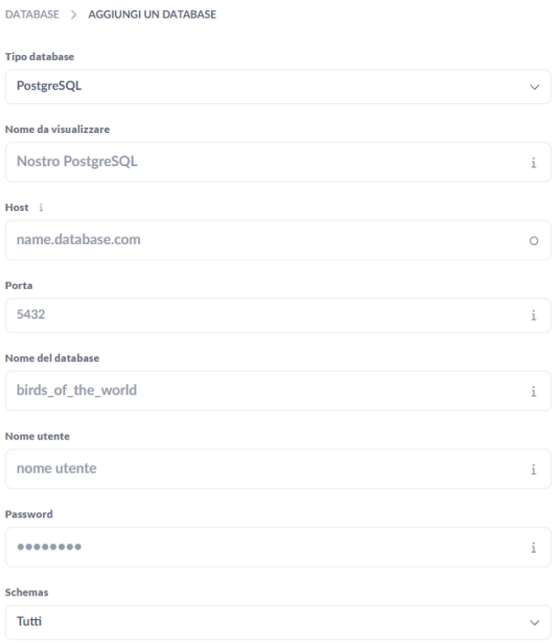

Figura 2.1: Menù configurazione database Metabase

In figura è presente il menù di configurazione del database dalla quale verranno poi utilizzati i dati per andare a generare le dashboard. I tipi di database che Metabase accetta sono molteplici, relazionali e non relazionali.

Una volta aggiunto il database, è possibile procedere con la creazione della dashboard tornando alla schermata home. Cliccando il pulsante 'Nuovo' in alto a destra è possibile accedere alla creazione di una dashboard con i dati appena aggiunti.

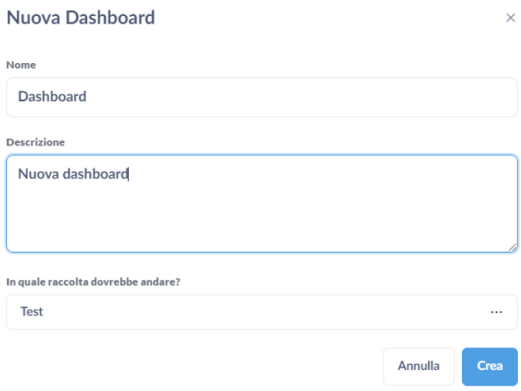

Figura 2.2: Menù salvataggio dashboard Metabase

È possibile definirne il nome e dare una breve descrizione di quello che andrà a contenere.

Una volta che la dashboard è stata creata, è possibile inserire le visualizzazioni. Il menù risulta essere molto semplice e permette la creazione di query SQL per la visualizzazione dei dati. Metabase offre, come supporto iniziale, una serie di data set di prova e delle dashboard già create piene di esempi in modo da facilitare la comprensione del suo funzionamento per permettere di essere ampiamente utilizzato anche da utenti non esperti di programmazione o del linguaggio SQL per le query.

Installazione

L'installazione di Metabase avviene attraverso l'utilizzo di Docker [19]. Bisogna prima pullare l'immagine di Metabase sul proprio computer/server utilizzando il comando:

```
docker pull metabase/metabase:latest
```

A seguire si deve creare un container a partire dall'immagine appena scaricata utilizzando il comando:

```
docker run -d -p 3000:3000 --name NOME metabase/metabase
```

Questo porterà alla creazione di un container di nome 'NOME' basato sull'immagine di Metabase.

2.2 Kibana

Kibana [11] è uno stumento di dashboarding open-source che fa parte dello stack ELK insieme a Elasticsearh e Logstash. Kibana, infatti, non può essere usato singolarmente ma si appoggia a Elastichsearh. All'interno di quest'ultimo vengono inserite tutte le connessioni ai database che si vuole utilizzare per le visualizzazioni mentre Kibana è progettato per utilizzare questi dati e creare delle visualizzazioni per l'utente. Le sue caratteristiche principali sono:

- Possibilità di creare dashboard dinamiche ed interattive
- Possibilità di utilizzare delle mappe reali per confrontare i dati su più regioni diverse con la funzionalità Kibana Maps
- Possibilità di eseguire automaticamente dei controlli su anomalie all'interno dei grafici con l'utilizzo del Machine Learning

Viene usato, ad esempio, nel caso in cui si debba controllare il funzionamento di molteplici sensori conoscendo il loro comportamento standard. Utilizzando la funzionalità di Machine Learning, l'utente viene a conoscenza se si sono presentate delle anomalie e può verificarla o segnalare un comportamento consono dovuto a fattori esterni.

Dashboard in Kibana

Come detto precedentemente, le dashboard in Kibana vengono create a partire dai dati inseriti all'interno di Elasticsearch. Una volta aggiunti, si può procedere con la creazione della dashboard che li visualizzerà. Partendo dalla home di Kibana, si può accedere alle dashboard andando nel menù laterale nella sezione 'Dashboards'. Cliccando 'Create dashboard' verrà aperta la nuova sezione dove sarà possibile inserire nuovi pannelli, utilizzare pannelli preesistenti creati in precedenza o generare nuovi grafici utilizzando i campi dei database aggiunti in precedenza.

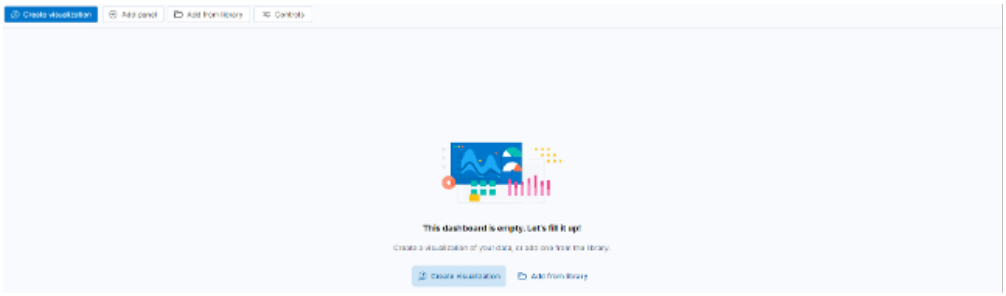

Figura 2.3: Dashboard Kibana

Una volta terminata la configurazione, in alto a destra è presente il pulsante 'Save' che consentirà il salvataggio, con la possibilità di dare un nome alla dashboard, una descrizione e un tag per una più facile ricerca successiva.

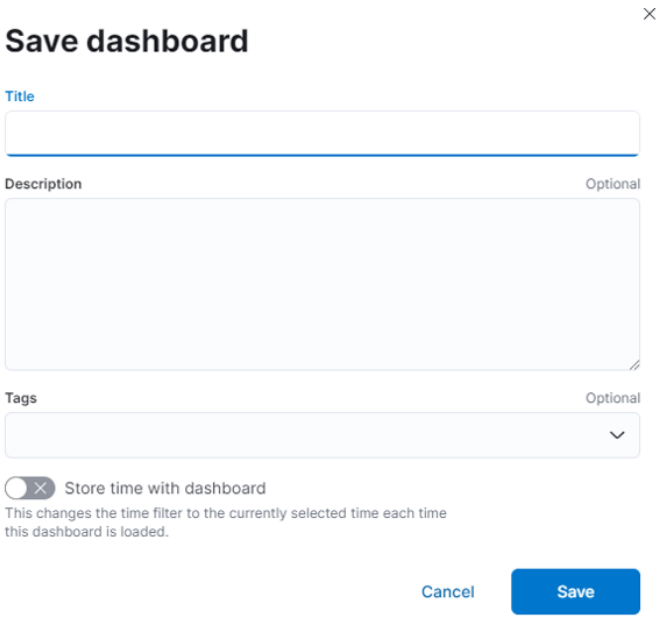

Figura 2.4: Menù salvataggio dashboard Kibana

Sono già presenti dei dati e delle dashboard connesse per un più semplice utilizzo e modellazione. Non meno importante, la creazione delle dashboard avviene attraverso l'utilizzo dell'interfaccia grafica permettendo ad utenti non esperti di utilizzarlo. Queste due caratteristiche sono comuni anche con Metabase.

Installazione

La sua installazione [9] utilizza i conteiner Docker. Come detto, Kibana funziona in coppia con Elasticsearch. Infatti, per prima cosa bisogna creare una network che connetterà i due container. Il comando da eseguire è:

```
docker network create elastic
```

Una volta creata la rete, si passa alla creazione del container di Elasticsearch con il pull della sua immagine:

```
docker pull docker.elastic.co/elasticsearch/elasticsearch:latest
```

E con la creazione di un container:

```
docker run --name es01 --net elastic -p 9200:9200 -it -m 1GB
docker.elastic.co/elasticsearch/elasticsearch
```

Dopo la creazione del container, verranno stampati a schermo due stringhe che rappresentano la password ed il token per accedere a Kibana. Una volta salvate le due stringhe, si passa al pull dell'immagine di Kibana:

```
docker pull docker.elastic.co/kibana/kibana:latest
```

Con successiva creazione del suo container:

```
docker run --name kib01 --net elastic -p 5601:5601
docker.elastic.co/kibana/kibana:8.14.1
```

All'avvio di Kibana nel browser alla porta 5601 verrà richiesto l'inserimento del token salvato in precedenza per la connessione con Elasticsearch. Infine, verranno richiesti username e password per l'accesso. Lo username da inserire è 'elastic' mentre la password è quella salvata nel precedente passaggio.

In caso di smarrimento del token o della password sono presenti due comandi che è possibile eseguire per rigenerarli entrambi:

- Token

  ```
  docker exec -it es01 /usr/share/elasticsearch/bin/
  elasticsearch-create-enrollment-token -s kibana
  ```

- Password

```
docker exec -it es01 /usr/share/elasticsearch/bin/
elasticsearch-reset-password -u elastic
```

2.3 Grafana

Grafana [6] è una piattaforma open-source che si occupa della visualizzazione e del monitoraggio di dati. La sua caratteristica principale è quella di integrarsi con molteplici database da utilizzare come fonte di dati. Questo consente agli utenti di utilizzarla senza problemi di compatibilità.

Viene ampiamente utilizzata per creare dashboard interattive e personalizzabili che consentono di visualizzare, comprendere e monitorare metriche di sistema, prestazioni degli applicativi, dati di log e altro ancora.

Le dashboard possono essere formate da vari tipi di grafici, come grafici a linee, a barre, e altri ancora. Gli utenti possono interagire con i grafici in tempo reale.

Dashboard in Grafana

Le dashboard sono uno degli elementi fondamentali di questa piattaforma. Sono utilizzate per aggregare dati da diverse fonti di dati.

Le dashboard sono costituite da più strutture ma le principali sono i pannelli, le variabili e il query editor.

È possibile generare una dashboard attraverso l'apposito menù nella barra laterale 'Dashboards'.

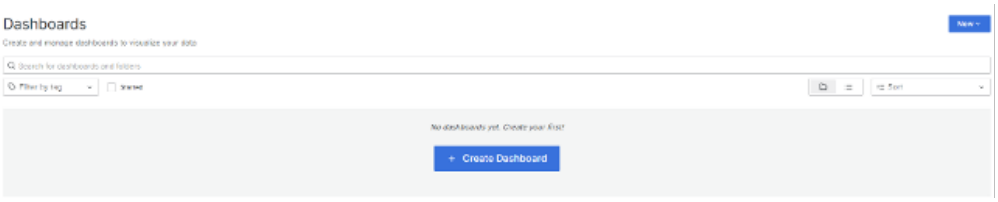

Figura 2.5: Dashboard Grafana

Con il pulsante 'Create Dashboard' verrà mostrata una dashboard bianca pronta per essere aggiornata.

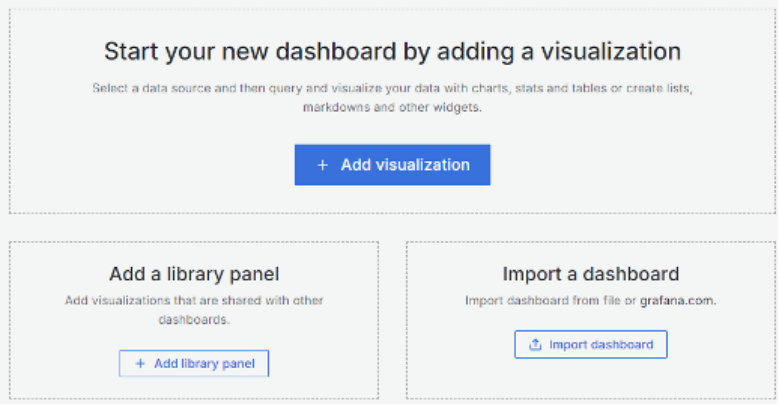

Figura 2.6: Configurazione dashboard Grafana

Tramite 'Add visualization' è possibile inserire i pannelli da visualizzare.

I pannelli sono i blocchi che costituiscono la visualizzazione finale dei dati sottoforma di grafici. Questi possono essere di vario tipo, alcuni esempi sono: grafici a linee, a barre, a torta e altri ancora.

Alla fine della configurazione della dashboard, per salvare si dovrà cliccare l'icona in alto a destra e si aprirà una schermata nella quale inserire il titolo della dashboard, una descrizione e in quale cartella salvare la dashboard.

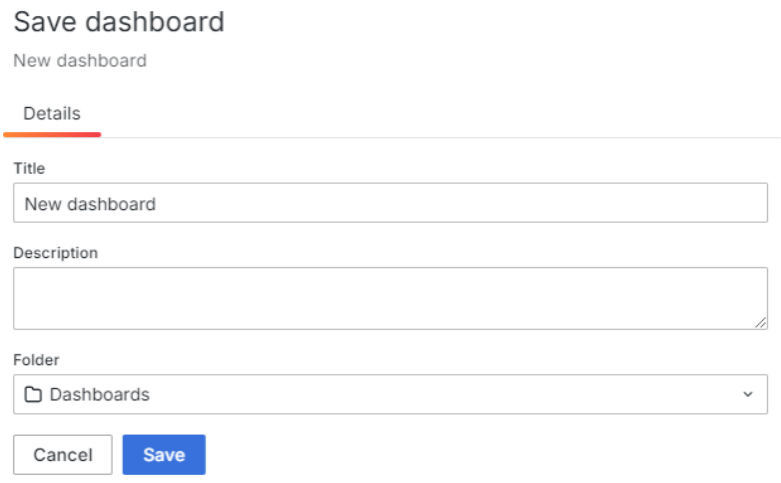

Figura 2.7: Menù salvataggio dashboard Grafana

Le variabili, invece, permettono di rendere le dashboard più dinamiche ed interattive. Queste possono essere usate dagli utenti per filtrare i dati mostrati secondo determinati criteri.

Possono essere di diversi tipi a seconda del contesto in cui vengono utilizzate. Alcuni esempi possono essere:

- Costanti Valori statici definiti dall'utente come nomi di host, di ambiente, ecc.
- Query Consentono all'utente di interrogare un database per ottenere dei valori in particolare. Si possono usare per ricavare delle liste di valori o di nomi.
- Intervalli di tempo Consentono di definire degli intervalli di tempo definiti come 'ultima ora', 'ultime X ore', ecc.

Le variabili possono essere configurare nella sezione 'Variabili' dalle impostazioni di Grafana. È possibile definirne il nome, il tipo, la query e l'ordine di visualizzazione in caso di più valori.

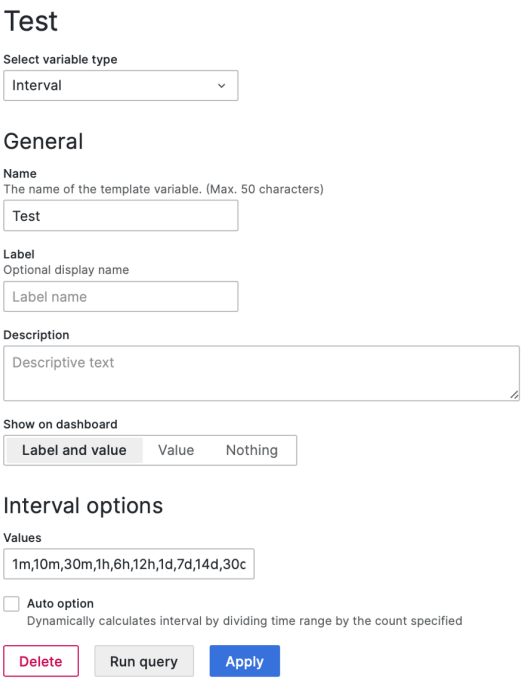

Figura 2.8: Menù configurazione variabili Grafana

Il query editor, infine, è l'area nella quale viene definita la tipologia di collegamento tra la datasource utilizzata e il grafico da visualizzare. È possibile definire il path verso cui effettuare la query, il metodo di richiesta (GET, POST, PUT, DELETE), i parametri che devono essere passati nel body della richiesta, gli headers se presenti e come Grafana deve interpretare i dati in ingresso per la loro corretta visualizzazione.

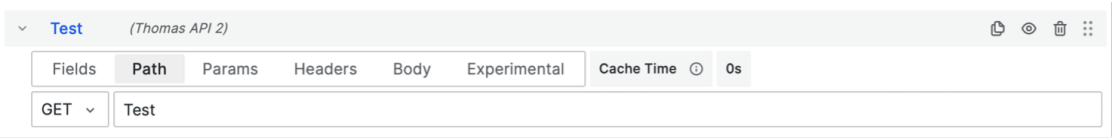

Figura 2.9: Menù configurazione query Grafana

Questo componente utilizza le data source che sono state inserite nel container Grafana per effettuare le query. Le data source sono di vario tipo e possono essere aggiunte al container attraverso l'apposita sezione presente nel pannello laterale sotto la voce 'Connections'.

Al primo avvio del container di Grafana, alcune data source sono già presenti e pronte all'uso mentre molte altre devono essere installate dopo l'avvio. Se, invece, all'avvio si conoscono già le tipologie di data source che verranno utilizzate, queste possono essere installate attraverso la modifica del file docker-compose.yml contenente la dichiarazione dei plugin da installare alla creazione del conteiner. La dicitura è da aggiungere sotto la voce environment.

GF_INSTALL_PLUGINS=plugin_name

In questo modo il plugin desiderato verrà installato all'avvio del container.

Una volta aggiunta, la data source è presente e può essere utilizzata in Grafana.

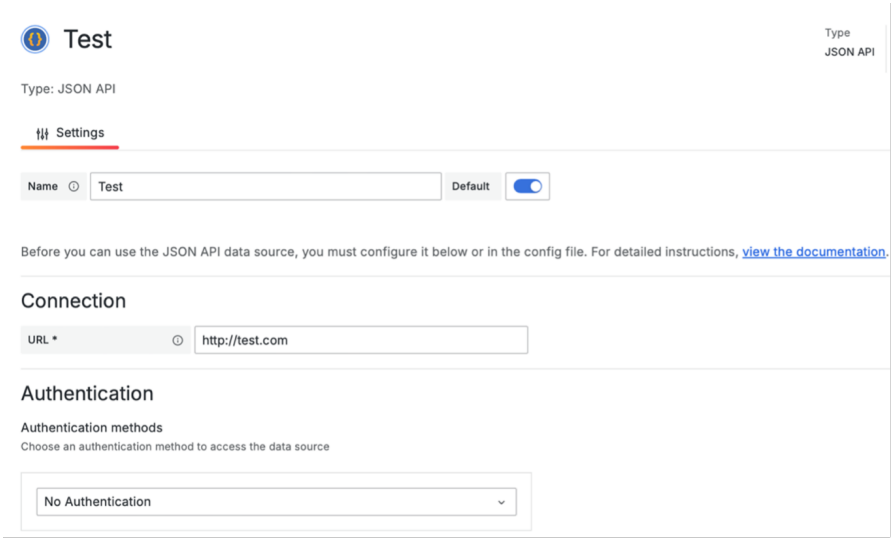

Figura 2.10: Menù configurazione datasource Grafana

Nella finestra di configurazione in Grafana è possibile scegliere il nome che verrà dato alla data source, l'url alla quale effettuare le richieste API e, se presente, il tipo di autenticazione che le richieste devono avere per poter accedere ad essa.

Come buildare un container Grafana [18]

Il codice di Grafana è open-source ed è possibile clonarne il contenuto per avere una base dal quale partire per apportare modifiche o inserire nuove funzionalità. Una volta clonato il repository usando il comando: git clone URL (link del repository GitHub di Grafana), si avrà una copia in locale dello stesso. Attraverso la linea di comando si accede alla cartella principale di Grafana che contiene il Dockerfile e si lancia il comando:

```
docker build -t esempio .
```

Questo comando indica che si verrà a creare un'immagine di Grafana con il nome 'esempio'. Il punto alla fine indica che il percorso del Dockerfile, in questo caso è la cartella attuale. Una volta creata l'immagine, è possibile creare un container usando Docker Desktop o si può usare il comando:

```
docker run -d -p x:x -name grafana-test esempio
```

Dove:

- x indica la porta sulla quale il container girerà (Es: 3000)
- –name grafana-test è il nome che verrà dato all'immagine creata, in questo caso grafana-test
- Esempio è il nome del container dal quale partire per creare l'immagine

Una volta seguiti questi passaggi, sarà possibile accedere a Grafana attraverso l'url:

```
http://localhost:x
```

Apportate le modifiche, buildato il container e testato le nuove funzionalità, si può effettuare il deploy dell'immagine su Docker Hub creando un account privato con un repository ed effettuando i seguenti passaggi:

- Eseguire docker tag per dare l'etichetta all'immagine che si vuole inserire su Docker hub:

  ```
  docker tag esempio utente/nome-repo:tag
  ```

  Dove utente è il nome utente di Docker hub, nome-repo è il nome dato al repository e tag è un flag per identificare la versione dell'immagine.

- Eseguire docker login da linea di comando per accedere all'account.
- Eseguire docker push per caricare l'immagine su Docker hub:

  ```
  docker push utente/nome-repo:tag.
  ```

Componenti utilizzati

In primo luogo si è dovuto pensare ad uno studio approfondito del codice sorgente di Grafana consultabile attraverso il repository Github [2]. Questo per comprendere come funziona, quali sono le tecnologie che implementa e come è strutturato per andare poi a implementare le modifiche desiderate.

Cercando nel codice di Grafana, si è risaliti ai vari file e componenti che rappresentano il flusso di dati per la creazione e gestione di un iframe contenente un grafico.

Il primo componente coinvolto è AppWrapper che gestisce la renderizzazione delle route. Alla fine della sua esecuzione, presenta nel return il componente presente in GrafanaRoute. Si occupa della ricerca della route desiderata e prepara la configurazione da visualizzare. Nel caso specifico di un iframe, il componente padre della route è situato in SoloPanelPage. In questo componente vengono caricate le informazioni circa la dashboard ed il panel all'interno della dashboard da visualizzare. Proprietà che vengono passate a DashboardPanel e le utilizza per caricare lo stile del pannello desiderato per poi arrivare al componente PanelStateWrapper. In questo componente è presente la logica effettiva di render del panel; quindi, nel suo campo return è presente il codice che andrà a generare il grafico dell'iframe scelto. In seguito una visione più dettagliata dei componenti e dei file appena citati.

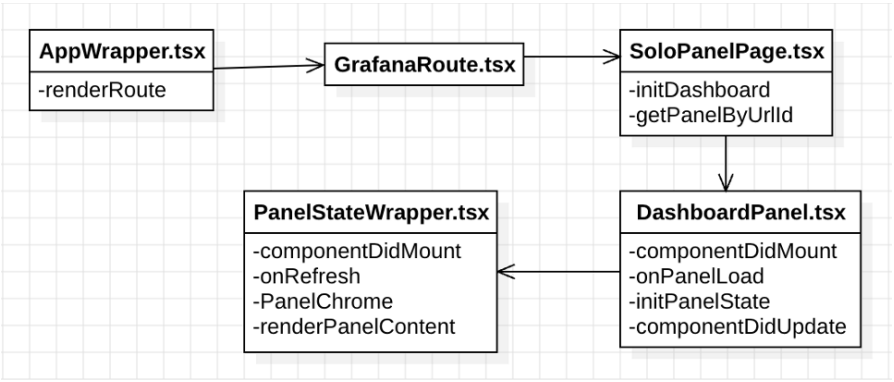

Figura 2.11: Flusso dati creazione grafico Grafana

Prima della creazione del grafico, Grafana utilizza tre componenti per la gestione del render delle pagine. Questi tre componenti sono: AppWrapper.tsx, GrafanaRoute.tsx e routes.tsx.

AppWrapper.tsx

Questo componente è situato in public/app.

Gestiste la renderizzazione delle route chiamando il componente GrafanaRoute.

La funzione principale che garantiste questo comportamento è:

- renderRoute (route: RouteDescriptor) Riceve come unico parametro in ingresso route che contiene al suo interno tutti i dati per il render della route richiesta.

GrafanaRoute.tsx

Questo componente è situato in public/app/core/navigation.

Riceve come parametro in ingresso le proprietà e la route da renderizzare.

Routes.tsx

Questo componente è situato in public/app/routes.

Contiene la lista delle route che Grafana può renderizzare. In ognuna di esse è presente il componente che contiene la logica per mostrare la pagina desiderata.

La route che verrà modificata per implementare le nuove funzioni è:

```
/d-solo/:uid/:slug
```

Questa route richiama il componente che descriveremo in seguito SoloPanelPage.tsx.

I componenti ricercati che quindi gestiscono la creazione dei grafici e la loro manipolazione sono principalmente tre: SoloPanelPage.tsx (già citato), DashBoardpanel.tsx e PanelStateWrapper.tsx.

SoloPanelPage.tsx

Questo componente è situato in public/app/features/dashboard/containers. È il primo richiamato dalla route e si occupa di recuperare la dashboard ed il panel che verrà poi mostrato in pagina.

Le funzioni principali in questo componente sono: initDashboard e getPanelByUrlId.

- initDashboard Questa funzione carica la dashboard usando come parametri quelli passati tramite url a Grafana.

- getPanelByUrlId Viene chiamata sulla dashboard appena caricata, come parametro in ingresso utilizza l'id del panel da cercare, passato come props al componente, e restituisce il panel, che viene salvato in uno stato locale. Nel caso in cui il panel sia inesistente, viene settato un flag 'notFound' a true e successivamente invece del panel viene mostrato il messaggio 'Panel with id id not found'.

Una volta caricati dashboard e panel, il componente in questione chiama DashboardPanel. Le props principali che vengono passate sono: dashboard, panel, timezone (recuperata dall'url dell'iframe se presente) e hideMenu (flag che rappresenta la presenza o meno di un menù a tendina utilizzato all'interno dell'editor di Grafana).

DashboardPanel.tsx

Questo componente è situato in public/app/features/dashboard/dashgrid. Si occupa di caricare ed inserire lo stile del panel.

L'unica funzione da citare in questo componente è: onPanelLoad.

- onPanelLoad Non ha nessun parametro in ingresso. Viene chiamata al mount del componente. Al suo interno è presente la chiamata a initPanelState che riceve come parametro il panel e gestite lo stile del panel modificato attraverso l'editor di Grafana.

Questo componente chiama poi PanelStateWrapper.

Le props principali che vengono passate sono le medesime con la differenza che il panel è stato aggiornato all'interno del componente.

PanelStateWrapper.tsx

Questo componente è situato in public/app/features/dashboard/dashgrid. Ultimo componente che si occupa del render del grafico, degli aggiornamenti che il grafico mostrato può subire come aggiunta/rimozione di annotazioni e zoom in/out.

Presenta molte funzioni al suo interno, le più importanti sono quelle presenti all'interno di componentDidMount dove il risultato finale è l'aggiunta delle funzioni che il panel deve avere una volta renderizzato. Altra funzione che verrà usata in seguito da nominare è la onRefresh.

- onRefresh Non riceve nessun parametro in ingresso e si occupa di ri-renderizzare il grafico in seguito alle modifiche effettuate. Il primo controllo che esegue è sul time range, se questo è modificato, il grafico viene riaggiornato di conseguenza. Il secondo riguarda i dati da visualizzare, effettua nuovamente la query per ricevere i dati e aggiornare il grafico di conseguenza.

Una volta montato il componente, il render finale che si occupa di mostrare il grafico è formato dalla coppia PanelChrome e dalla funzione renderPanelContent.

- PanelChrome Componente che fa parte della libreria ui di Grafana e gestisce la porzione di pagina che conterrà il grafico utilizzando weight e height come dimensioni.

- renderPanelContent Genera il grafico che verrà inserito nello spazio generato da PanelChrome utilizzando i dati caricati in precedenza ed il panel.

Ultimo componente cercato per gli step successivi di modifica è quello che gestisce le variabili di Grafana. Il componente fa parte della libreria runtime di Grafana ed è templateSrv.ts.

templateSrv.ts

Il codice è visualizzabile al percorso:

packages/grafana-runtime/src/services/templateSrv.ts.

Contiene la logica che gestisce le variabili e verrà utilizzato per gli aggiornamenti dell'iframe. La funzione principale utilizzata è getTemplateSrv che restituisce un array di oggetti che contengono le variabili e la loro configurazione.

Capitolo 3

Estensione di Grafana tramite plugin

L'utilizzo principale di Grafana è quello di visualizzazione e monitoraggio di dati temporali. Questo avviene attraverso la creazione di dashboard interattive che aiutano ad analizzare dati provenienti da diverse fonti. Le dashboard sono basate sull'utilizzo dei plugin. I plugin [3] rappresentano componenti aggiuntivi che si inseriscono all'interno di Grafana per estenderne le funzionalità per dare agli utenti svariati modi di interpretare e visualizzare i dati.

L'utilizzo di plugin all'interno di Grafana è utile agli sviluppatori per aggiungere nuove funzionalità senza andare a modificare il codice sorgente. Questa modularità rende più semplice l'aggiornamento della dashboard in caso di modifiche future.

3.1 Tipi di plugin

I possibili plugin che possono essere utilizzati all'interno delle dashboard di Grafana sono di diversi tipi. Ognuno è stato sviluppato per svolgere funzioni specifiche e aggiungere nuove capacità.

Panel Plugin

I panel plugin sono utilizzati per la visualizzazione dei dati sottoforma di grafici all'interno delle dashboard. Sono presenti svariati tipi come grafici a linee, grafici a barre, tabelle, ecc. I panel plugin possono essere personalizzati in base alle esigenze degli utenti.

Alcuni esempi possono essere:

- Graph Panel per visualizzare i dati in forma di grafici
- Table Panel per avere una visualizzazione in tabelle

Nel caso specifico preso in esame, il panel utilizzato è il TimeSeries Panel. Questo panel consente la visualizzazione dei dati in forma di grafico a linea. Questo tipo di Panel Plugin riceve i dati da mostrare attraverso delle richieste API alle datasource collegate al container Grafana.

Lo stile del pannello può essere modificato dall'utente attraverso il Panel Editor presente nella dashboard di Grafana.

Le modifiche possono riguardare:

- Le opzioni di base come il titolo del panel in questione e la descrizione
- Lo stile proprio del grafico come spessore della linea, colore di quest'ultima
- Possibilità di aggiungere annotazioni in determinate posizioni del grafico
- Modifica del tooltip che viene visualizzato quando si passa con il cursore sul grafico

Data Source Plugin

I Data Source Plugin consentono a Grafana di connettersi alle diverse fonti di dati per consentire poi la loro visualizzazione. Estendono le funzionalità di base che Grafana presenta consentendo agli utenti di integrare dati provenienti da diverse fonti all'interno delle dashboard. Alcuni esempi possono essere:

- Prometheus per il monitoraggio
- InfluxDV per database di serie temporali
- Elasticsearh per l'analisi dei dati

Nel caso specifico preso in esame, la datasource utilizzata è JSON API. Questo plugin viene utilizzato per ricavare dati utilizzando un URL che ritorna un JSON. Può essere utilizzato per ricevere dati da API rest o da file statici presenti nei server.

App plugin

Gli App Plugin vengono utilizzati per combinare più pannelli e fonti di dati all'interno della stessa visualizzazione per risolvere casi specifici. Possono includere pagine di gestione o dashboard preconfigurate.

Alcuni esempi possono essere:

- Kubernetes App per il monitoraggio e la gestione dei cluster Kubernetes
- AWS CloudWatch App per il monitoraggio delle risorse AWS tramite CloudWatch

App with scenes Plugin

Gli App With Scenes Plugin vengono utilizzati per creare dashboard dinamiche utilizzando pagine custom inserendo panel plugins e datasources svariate. Possono implementare interazioni avanzate, animazioni e interazioni che migliorano l'usabilità delle dashboard.

Alcuni esempi possono essere:

- Worldmap Panel per la visualizzazione di dati geografici
- Geomap Plugin per creare mappe geografiche avanzate e personalizzate

3.2 Come installare i plugin

I plugin in Grafana possono essere installati in svariati modi a seconda dell'esigenza e dell'ambiente di sviluppo in cui Grafana è presente [15]. Al primo avvio del container di Grafana sono presenti svariati plugin installati di default ma possono esserne aggiunti altri.

Grafana Marketplace

Il metodo più semplice è installarli utilizzando l'interfaccia utente di Grafana. Si accede al Marketplace cliccando sulla voce 'Plugin' del menù laterale.

All'interno del marketplace sono presenti tutti i plugin facenti parte della community di Grafana e sono tutti plugin verificati e controllati dal team di sviluppo. I plugin possono essere filtrati per categoria, raccomandati e più utilizzati. Una volta trovato il plugin desiderato, si può installare attraverso la sua schermata, che contiene le info su come utilizzarlo, cliccando sul tasto 'Install'.

Grafana CLI

Grafana CLI è un tipo di installazione che sfrutta la riga di comando per gestire i plugin e per effettuare anche la loro installazione. Diventa utile in caso si voglia avere installazioni automatizzate o se si predilige l'utilizzo della linea di comando.

Per effettuare l'installazione si utilizza il terminale sulla macchina in cui Grafana è installato e attivo. Si esegue il seguente comando:

```
grafana-cli plugins install PLUGIN
```

Dove PLUGIN deve essere sostituito con il nome del plugin che si vuole installare.

Successivamente si deve riavviare il container di Grafana per rendere effettiva l'installazione.

Manuale

Questo tipo di installazione risulta essere comodo se si vuole installare un plugin non verificato dal team di Grafana poiché si va a inserire il plugin direttamente nel codice sorgente di Grafana. I plugin possono essere ricercati dai siti ufficiali o da repository GitHub. Una volta scaricato e decompresso il contenuto del plugin, questo deve essere copiato al seguente percorso nel codice di Grafana:

```
/var/lib/grafana/plugins
```

Prima di riavviare il conteiner si devono verificare i permessi dei file appena copiati per consentire a Grafana di accedervi senza problemi.

Docker

Questo tipo di installazione può essere effettuato se Grafana è installato all'interno di un container Docker. Per installare i plugin si deve eseguire il seguente comando da terminale:

```
docker run -d -p PORT:PORT  \ --name=NAME \ -e
"GF_INSTALL_PLUGINS=PLUGIN" \ grafana/grafana
```

Dove:

- PORT è la porta sulla quale il container verrà installato
- NAME è il nome da dare al container
- PLUGIN è il nome del plugin da installare. È possibile installare più plugin con lo stesso comando separandoli con una virgola

Dopo aver seguito uno di questi metodi per l'installazione è possibile verificarne l'effettivo funzionamento andando a cercare il plugin appena installato nella lista dei possibili plugin da utilizzare per creare una dashboard o un panel.

3.3 Come sviluppare un plugin

Se tra i plugin presenti nel Marketplace e quelli già ideati e presenti su GitHub nessuno soddisfa le funzionalità che l'utente sta cercando, è possibile procedere con lo sviluppo di un nuovo plugin [5] che verrà poi inserito su Grafana.

Il Plugin è un'applicazione React, quindi, è possibile svilupparlo come se fosse una pagina web e poi inserirla all'interno di un Plugin Grafana. Le possibilità di implementazione sono infinite e qualsiasi utente può sviluppare il proprio plugin con le funzionalità che desidera integrare e sviluppare.

Lo sviluppo di un plugin parte da un plugin di base che può essere installato eseguendo uno dei seguenti comandi da terminale a seconda della tecnologia che si vuole utilizzare:

- npx @grafana/create-plugin@latest
- pnpm dlx @grafana/create-plugin@latest
- yarn create @grafana/plugin

Dopo aver eseguito uno di questi comandi, il terminale richiederà delle informazioni per la creazione del plugin e sono:

- Nome che si vuole dare al plugin
- Organizzazione al quale si appartiene
- Descrizione
- Tipo di plugin che si vuole andare a sviluppare (panel, datasource, app)

Una volta terminata la configurazione, verranno scaricati tutti i pacchetti che servono per la creazione del plugin ed a schermo ci saranno i comandi per la build e la creazione di un container Grafana con all'interno il plugin appena generato.

Come anticipato, per testare il funzionamento di un plugin appena sviluppato prima di pubblicarlo è possibile testarlo in un container di sviluppo Grafana. Questo avviene utilizzando degli script presenti nel package.json del plugin appena scaricato. In seguenza bisogna prima eseguire il download dei moduli presenti nel package.json, eseguire il build del plugin che porterà alla creazione di una cartella contenente la sua build e alla successiva creazione del container Grafana con innestato il plugin al suo interno. Per eseguire gli script è possibile utilizzare il menù laterale presente in Visual Studio Code o eseguire manualmente i comandi connessi da terminale. I tre script coinvolti durante questo procedimento sono:

- build: npm run build Installa i moduli utilizzati nel plugin
- dev: npm run dev Genera la build del plugin
- server: npm run server Stanzia un container Grafana con il plugin al suo interno

La cartella che viene generata contiene un plugin di base dal quale è possibile estendere le funzionalità a seconda del tipo scelto durante l'installazione. Ogni tipo avrà quindi un diverso albero di componenti all'interno.

3.4 Sviluppare un Panel Plugin

Come detto in precedenza, è possibile procedere con la stesura del codice per un Panel Plugin utilizzando uno dei tre comandi visti. Una volta creato il progetto di base, questo avrà la seguente struttura:

- provisioning
- dashboard: file per la configurazione di dashboard preimpostate
- datasources: file per la configurazione di datasource preimpostate
- src

  components: file tsx per la definizione dei componenti

  module.ts: contenente le opzioni che verranno mostrate nell'editor del panel appena creato

  types.ts: contenente l'interfaccia di dichiarazione delle opzioni
- package.json: contenente le dipendenze che verranno usate all'interno del plugin e gli script necessari al build, run e test del plugin
- cypress: suite integrata per effettuare e2e test prima dell'upload su Grafana Cloud
- Altri file di configurazione che però non verranno utilizzati durante lo sviluppo

Il componente principale nel quale definire il proprio panel sarà SimplePanel.tsx presente nella cartella components.

Sviluppare il panel

Se precedentemente, con l'utilizzo del codice sorgente di Grafana, bastava inserire la postMessage e utilizzarla con tutti i tipi di panel presenti, se si vuole creare il proprio plugin si deve sviluppare il proprio grafico partendo dal codice React.

Per quanto riguarda la ricezione dei dati da visualizzare, questi sono accessibili attraverso la proprietà data presente in SimplePanel.tsx ma il render è lasciato allo sviluppatore. Le librerie per generare dei grafici sono molteplici ma in questo esempio verrà utilizzata la libreria recharts [17].

Prima di utilizzarla deve essere aggiunta al progetto usando il seguente comando:

```
npm install recharts
```

Per semplicità verrà utilizzata la datasource di test di Grafana che fornisce dati random per la visualizzazione. In particolare, la query è chiamata 'Random Walk'.

Per prima cosa bisogna creare un nuovo file in components che conterrà la definizione del grafico. Nell'esempio LineChart.tsx.

```
import React from 'react';
import { LineChart, Line, XAxis, YAxis,
         CartesianGrid, ResponsiveContainer }
from 'recharts';
interface MyLineChartProps {
    data: any;
}
export function MyLineChart({ data }: MyLineChartProps) {
    return (
        <ResponsiveContainer width="100%" height={400}>
            <LineChart
                data={data}
                margin={{ top: 20,
                          right: 30,
                          left: 20,
                          bottom: 5 }}
            >
                <CartesianGrid strokeDasharray="3 3" />
                <XAxis dataKey='x' />
                <YAxis dataKey='y' />
                <Line type="monotone" dataKey="y" />
            </LineChart>
        </ResponsiveContainer>
    );
};
```

Figura 3.1: Esempio per creazione di un grafico TimeSeries

Nella figura mostrata è presente la definizione del componente utilizzando la libreria recharts installata in precedenza. L'interfaccia MyLineChartProps contiene il vettore con le coppie di oggetti che verranno mostrati sugli assi x e y del grafico.

La nuova definizione del componente SimplePanel sarà questa di conseguenza. In ingresso viene ricevuto l'oggetto data che contiene i valori da mostrare. Questo viene caricato da Grafana utilizzando la datasource scelta nell'apposita sezione all'interno dell'editor del panel. L'oggetto data presenta più campi, il principale utilizzato nell'esempio è series, questo contiene i risultati delle query o richieste api effettuate. Viene gestito come un array poiché possono essere presenti più richieste di dati all'interno dello stesso panel. Nell'esempio quindi viene utilizzato series[0] poiché si sta lavorando con una sola query. All'interno di series è presente la configurazione della query e, nell'array fields, i dati da visualizzare. Nell'esempio fields è un oggetto contenente due array che rappresentano le coppie che andranno sugli assi x ed y. Per questo x è preso da fields[0] mentre y da fields[1]. Questa manipolazione dei dati in ingresso porterà alla creazione di data2,

oggetto che conterrà tutte le coppie x/y da mostrare nel grafico.

Nelle figure successive verranno mostrati il risultato delle manipolazioni dei dati ed il nuovo panel appena descritto.

```
export const SimplePanel: React.FC<Props> = ({ options, data })
                                    => {
  const data2 = [];
  for (let i = 0;
         i < data.series[0].fields[0].values.length; i++) {
    data2.push(
    { x: data.series[0].fields[0].values[i],
    y: data.series[0].fields[1].values[i] });
  }
  return <MyLineChart data={data2} />;
};
```

Figura 3.2: Esempio per manipolazione data e creazione grafico

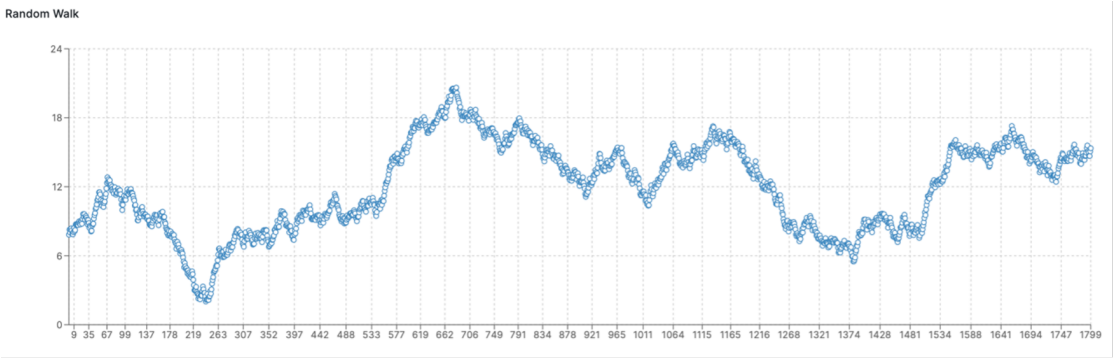

Figura 3.3: Grafico Panel Plugin

Aggiungere un opzione al panel editor

In Grafana è presente un'area della pagina dedicata solamente alla configurazione dello stile del panel. Questo facilita alcune semplici modifiche dato che è possibile configurare dei menù appositi durante lo sviluppo invece di andare a modificare e pubblicare un aggiornamento per ogni modifica che si vuole effettuare al panel. I menù che verranno creati saranno a disposizione nella sezione apposita per andare a modificare real time la configurazione di default scelta durante lo sviluppo.

Nel seguente esempio verrà mostrato come aggiungere un'opzione per la modifica del colore all'interno di un panel plugin che mostra un semplice cerchio.

Per aggiungere l'opzione all'editor del panel che riguarda quindi il CSS del panel, la dimensione del font o altro, si devono modificare più componenti.

Nell'esempio seguente verrà mostrato come creare un menù a tendina per modificare il colore di un panel contenente un cerchio.

La definizione dell'opzione in termini di codice React deve essere aggiunta all'interno della cartella components in un nuovo file tsx. Questo nuovo componente deve essere importato nel file module.ts per essere aggiunto all'user Interface aggiungendo un'etichetta per fare in modo che dal panel si riesca a risalire al nome ed al valore della option appena aggiunta.

Per utilizzare questa nuova proprietà, deve essere modificato il componente SimplePanel.tsx presente in components poiché è quello che si occupa del render del panel. Le opzioni appena create sono raggiungibili all'interno della proprietà options e possono essere usate nel componente per il render.

Di seguito un esempio:

```
import { StandardEditorProps } from "@grafana/data";
import { Select } from "@grafana/ui";
import React from "react";
interface ColorOption {
    label: string;
    value: string;
}
interface Settings {
    colors: ColorOption[];
}
type Props = StandardEditorProps<string, Settings>;
export const ColorEditor =
        ({ item, value, onChange }: Props) => {
    return <Select options={item.settings?.colors}
        value={value}
        onChange={
        (selectableValue) => onChange(selectableValue.value)}
        />;};
```

Figura 3.4: Esempio per creazione editor custom Grafana

Questa è la definizione di un componente per la gestione del colore all'interno del panel che si vuole creare, è stato definito nel file ColorEditor.tsx all'interno della cartella components.

ColorOption è un'interfaccia che rappresenta la coppia label/value dove label sarà la stringa mostrata a schermo nel panel editor (es: Red, Green, Blue) mentre value è il colore

corrispondente (es: red, green, blue). Il componente ColorEditor accetta in ingresso tre parametri: item, value e onChange. All'interno di item è presente il campo colors, value è il valore attuale mentre la onChange modifica il valore attuale ad ogni click. Questo componente genera un menù a tendina contenente i colori.

```
import { PanelPlugin } from '@grafana/data';
import { SimpleOptions } from './types';
import { SimplePanel } from './components/SimplePanel';
import { ColorEditor } from './components/ColorEditor';
export const plugin =
    new PanelPlugin<SimpleOptions>(SimplePanel)
    .setPanelOptions((builder) => {
    return builder
    .addCustomEditor({
      id: 'color',
      path: 'color',
      name: 'Color',
      editor: ColorEditor,
      settings: {
        colors: [
          {
            value: 'red',
            label: 'Red',
          },
          {
            value: 'green',
            label: 'Green',
          },
          {
            value: 'blue',
            label: 'Blue',
          },
        ],}}
);
```

Figura 3.5: Esempio aggiunta custom editor Grafana

Per aggiungere il componente appena creato alle proprietà e renderlo visibile si deve aggiungere la sua definizione nel file module.ts. Utilizzando addCustomEditor è possibile inserire gli editor appena creati al panel editor del plugin. I campi da inserire sono id, path e name. Id serve a diversificare gli editor aggiunti, path è il nome che viene dato alla variabile contenente il valore del colore mentre name è il nome che uscirà a schermo

nell'editor. Il campo editor rappresenta poi il componente da renderizzare, ColorEditor nel nostro caso mentre settings sono le coppie value/label che verranno mostrate a schermo.

```
export const SimplePanel: React.FC<Props> =
    ({ options, width, height }) => {
  const theme = useTheme2();
  let color = theme.visualization.getColorByName(options.color);
  return <svg
    width={width}
    height={height}
    xmlns='http://www.w3.org/2000/svg'
    xmlnsXlink='http://www.w3.org/1999/xlink'
    viewBox={`-${width / 2} -${height / 2} ${width} ${height}`}
  >
    <g>
      <circle style={{ fill: color }} r='100' />
    </g>
     </svg>;
};
```

Figura 3.6: Esempio per modifica colore utilizzando options

Quello mostrato in figura è il codice di un cerchio dove è possibile modificare il colore di riempimento attraverso il componente appena creato. L'opzione 'color' è accessibile tramite la proprietà 'options' in ingresso. Verrà utilizzata per recuperare la proprietà effettiva color attraverso il comando

```
let color = theme.visualization.getColorByName(options.color);
```

Trasformerà la stringa options.color nel color da inserire come proprietà al cerchio.

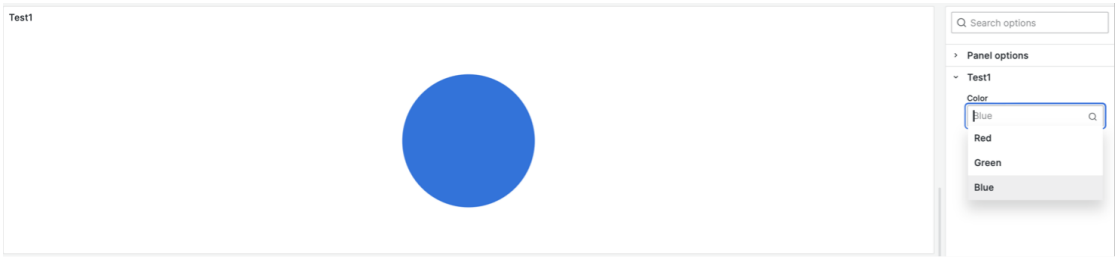

Figura 3.7: Esempio di un cerchio con editor custom

Questa è la schermata di Grafana se il nome del Plugin è Test1 e viene selezionato il colore Blue tra quelli presenti nel menù a tendina.

3.5 Sviluppare un App with scenes Plugin

Come detto in precedenza, è possibile procedere con la stesura del codice per un App with Scenes Plugin utilizzando uno dei tre comandi visti. A differenza del Panel Plugin, non è presente all'interno delle dashboard ma è un'opzione che viene aggiunta al menù laterale di Grafana nel momento in cui viene installata. Risulta essere quindi più complessa come struttura e possono essere aggiunte funzionalità più complesse realizzando una vera app all'interno di Grafana.

Le differenze principali con il Panel Plugin sono visibili all'interno della cartella src. Se nel primo caso presenta i file di configurazione per la visualizzazione del grafico, nell'App with Scenes Plugin contiene i file per la gestione delle routes, dei componenti da visualizzare all'interno di ogni singola route e un file per costanti per la configurazione delle datasource utilizzate o altri tipi.

Una visione più dettagliata della cartella src è la seguente:

- components

  App contenente i file sorgente dell'App che verranno utilizzati al build del plugin

  Routes contenente i file per la dichiarazione delle routes che verranno utilizzate nel plugin

- pages contenente i comonenti utilizzati per il render delle scene all'interno del plugin

- constants.js: file per la configurazione delle costanti. Queste possono riguardare ad esempio la dichiarazione delle datasource da utilizzare

I restanti file sono comuni con il Panel Plugin.

Svilupare il panel

Una seconda differenza con il Panel Plugin si riscontra nella possibilità di utilizzare i panel preimpostati per il render dei dati. La query, invece, deve essere presente nel codice invece di utilizzare l'editor di Grafana poiché non presente in questo tipo di plugin. Il codice che rende possibile il render del pannello è possibile visionarlo in scenes.tsx presente in pages/Home/scenes.tsx.

```
return new EmbeddedScene({
    $timeRange: timeRange,
    $variables: new SceneVariableSet({ variables: [ts] }),
    $data: queryRunner,
    body: new SceneFlexLayout({
      children: [
        new SceneFlexItem({
          body: PanelBuilders.timeseries()
            .setTitle('test')
            .build(),
        }),
      ],
    }),
    controls: [
      customButton,
    ],
  });
```

Figura 3.8: Codice creazione panel App with scenes Plugin

Questo è il componente che contiene il render del grafico. I parametri presenti sono:

1. timeRange contenente i campi from e to che indicano i due estremi del grafico
2. variables contenente le variabili utilizzate per il render del grafico
3. data contenente i dati per effettuare la query verso la datasource
4. body contenente il render del grafico, è possibile scegliere il tipo di panel (timeseries nella figura) e settare il titolo del grafico
5. controls contenente i menù di modifica che il grafico deve avere allegati

Alcuni esempi dei componenti appena citati sono i seguenti:

1. const timeRange = new SceneTimeRange(from: 'now-90m', to: 'now',); From e to sono gli estremi del grafico durante il primo render
2. const ts = new CustomVariable(name: 'ts', label: 'Time Range', value: '90m', query: '90m, 3h, 6h, 12h, 1d, 3d, 7d',); Variabile contenente il nome, la label rappresentante l'etichetta, value che è il valore iniziale assunto da essa e query sono i possibili valori che questa variabile può assumere. Verrà mostrato un esempio di creazione di variabile nel paragrafo successivo.

3. const queryRunner = new SceneQueryRunner(datasource: DATASOURCE-REF, queries: [refId: 'A', ,]); La query verso una datasource dichiarata nel componente costants.ts per ricevere i dati.

Aggiunta del box di modifica del timeRange

Una delle possibili modifiche da effettuare, risulta essere la modifica del timeRange del grafico attraverso l'utilizzo di una variabile. Per il raggiungimento di questo scopo, si devono modificare più file all'interno del progetto. In linea generale i passi da seguire risultano essere la creazione di un oggetto rappresentante la variabile ed i suoi possibili valori, la creazione di un'interfaccia grafica da inserire nel render e l'inserimento di quest'ultima nella visualizzazione del grafico.

In primo luogo, quindi, si procede con la creazione di un oggetto custom contenente la lista dei possibili valori da dare al campo from poiché è quest'ultimo che rappresenta l'estremo sinistro del timeRange. Questo viene fatto nel file CustomSceneObject.tsx.

```
export interface CustomSceneButtonState
            extends SceneObjectState {
  from: string;
}
export class CustomSceneButton extends
SceneObjectBase<CustomSceneButtonState> {
  static Component = CustomSceneButtonRenderer;
  onValueChange = (value: string) => {
    this.setState({ from: value });
  };
}
const times = ["90m", "3h", "6h", "12h", "1d", "3d", "7d"];
function CustomSceneButtonRenderer({ model }:
SceneComponentProps<CustomSceneButton>) {
  return times.map((time) => (
    <Button
      key={time}
      onClick={() => { model.onValueChange(time);}}>
      {time}
    </Button>
  ));
}
```

Figura 3.9: Esempio per creazione bottone custom

Bisogna creare un'interface contenente il campo from e utilizzarla per la classe CustomSceneButton. Bisogna infine codificare il componente che conterrà i bottoni, nell'esempio CustomSceneButtonRenderer. Questo aggiungerà all'interfaccia grafica una lista di bottoni contenente i campi presenti in times.

Una volta costruito questo componente, bisogna aggiungere l'interazione fra il click dei bottoni e l'aggiornamento del timeRange del grafico mostrato. Questo avviene nel file in cui è presente il render del grafico.

```
const customButton = new CustomSceneButton({
    from: '3h',
  });
  timeRange.addActivationHandler(() => {
    const sub = customButton.subscribeToState((newState) => {
      timeRange.setState({
        from: 'now-' + newState.from,
        to: 'now',
      });
      timeRange.onRefresh();
    });
    return () => {
      sub.unsubscribe();
    };
  });
```

Figura 3.10: Esempio per aggiunta handler bottone

Il codice mostrato va aggiunto in scenes.tsx.

Bisogna inizializzare un CustomSceneButton con il valore from. Successivamente viene aggiunto al timeRange un handler con la funzione addActivationHandler che verrà triggerata ogni volta che un bottone verrà cliccato.

Il risultato del click del bottone sarà l'aggiornamento del campo from del timeRange con successivo re-render del grafico in seguito alla chiamata della onRefresh sul timeRange.

La unsubscribe verrà richiamata il unmount del componente per eliminare questo handler appena creato.

Figura 3.11: Grafico di esempio con bottoni custom

Il grafico finale utilizzando la datasource di test di Grafana sarà quello mostrato in figura con l'aggiunta dei bottoni creati utilizzando i custom object appena descritti.

3.6 Pubblicare un plugin

Una volta che il plugin è stato codificato secondo le funzionalità prestabilite in fase di sviluppo, è possibile sottoporlo a review da parte del team di Grafana per aggiungerlo in futuro al catalogo dei Plugin certificati che è possibile installare attraverso i metodi visti in precedenza.

La pubblicazione si svolge attraverso vari step elencati qui di seguito [16].

Prima di iniziare

Prima di sottoporre il plugin alla revisione, come nel caso della pullRequest, ci sono dei check da effettuare riguardo il codice. Le linee guida per farlo sono pubbliche e visionabili all'interno del blog grafana.com.

Alcuni esempi possono riguardar l'uso che verrà fatto del plugin, se privato, pubblico o a fini commerciali (a pagamento). Non è possibile ricavare dati o informazioni degli utenti che utilizzano il plugin e non devono esserci file nascosti che effettuano queste operazioni.

Altri controlli riguardano le restrizioni. Il plugin, ad esempio, non deve essere il fork di un pluin già esistente e non può essere l'embedding di più plugin insieme.

Se queste regole non vengono rispettate, Grafana ha la possibilità di rifiutare la pubblicazione del plugin anche se verrà usato per scopi privati.

Successivamente è possibile procedere con il creare il file sotto forma di archivio ZIP per inserirlo nella richiesta di pubblicazione.

Effettuare la richiesta

Dopo aver creato l'archivio, si deve accedere al sito Grafana Cloud e creare un account o accedere con un account di Google, GitHub, Microsoft o Amazon.

Nel menù laterale, sotto 'Org Settings', è presente la dicitura 'My Plugins'. Nella nuova schermata si deve cliccare su 'Submit New Plugin' e si aprirà un form da compilare per effettuare la richiesta. Il form contiene i seguenti campi:

- Scelta fra 'Single' o 'Multiple' che riguarda la capacità del plugin di essere eseguito su una o più architetture
- URL che contiene lo ZIP creato in precedenza
- URL al repository GitHub che contiene il codice sorgente del plugin
- SHA1 per inserire l'hash del plugin
- Una breve descrizione degli step da eseguire per installare e configurare il plugin in modo da eseguirlo correttamente in fase di test
- Un checkbox se all'interno del plugin è presente un provisioning per effettuare dei test in modo da facilitare il compito del team di Grafana

Dopo aver compilato il form, non resta che cliccare Submit e mettersi in attesa.

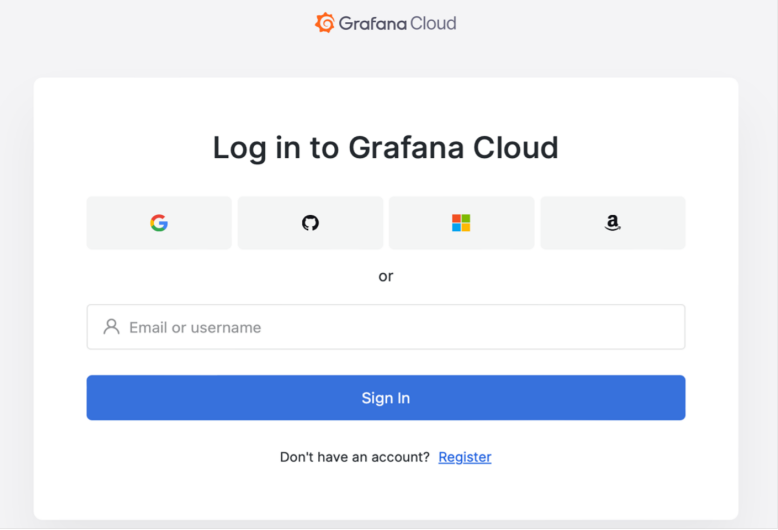

Figura 3.12: Schermata login Grafana Cloud

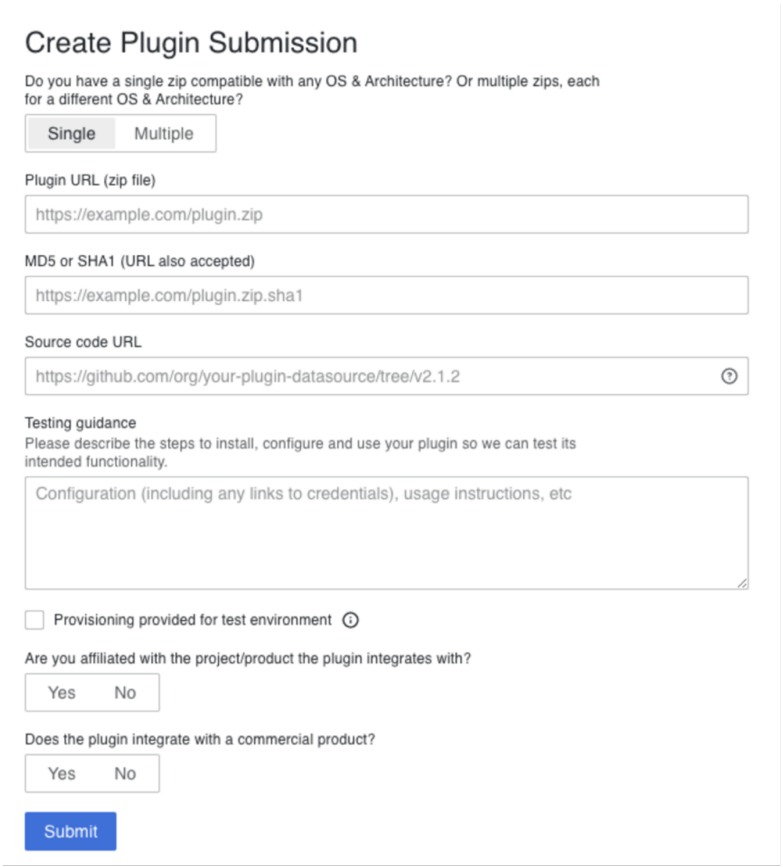

Figura 3.13: Form richiesta plugin

Controlli

Dopo aver effettuato la richiesta, vengono effettuati dei controlli automatici per verificare che la qualità e le misure di sicurezza siano superate.

Solo dopo aver passato questi controlli il plugin passa al controllo umano. Viene eseguita una code review approfondita e vengono svolti dei test in un'istanza di Grafana per verificare il corretto funzionamento. In questo caso è possibile che si venga contattati per aiutare il team ad installare e configurare correttamente il plugin. Questo ambiente di sviluppo creato in questo momento verrà utilizzato anche per i test futuri in caso di aggiornamento, da parte dell'utente, de plugin.

Aggiornare il plugin

Se si vuole aggiornare un plugin già esistente, si devono seguire gli stessi step mostrati in precedenza per pubblicarlo ma con l'eccezione di utilizzare 'Submit Update' nella schermata invece di 'Submit' per aggiungere questo nuovo aggiornamento al plugin già

esistente in modo da facilitare gli sviluppatori nel controllo poiché hanno già predisposto l'ambiente di test nel passaggio precedente.

Dopo che il team ha verificato la correttezza del codice, il funzionamento del plugin e il rispetto delle linee guida, il plugin viene reso pubblico ed accessibile attraverso le dichiarazioni data in fase di submit.

Inserire un plugin in un container Grafana senza pubblicarlo

In alternativa, se non si vuole attendere la revisione del plugin appena sottomesso a controllo, è possibile inserire il proprio plugin all'interno di un container come plugin 'unsigned' quindi non presente nel marketplace di Grafana. Questo avviene senza modificare il codice sorgente di Grafana ma andandolo ad inserire durante la build di Grafana. Quella che verrà aggiunta al container Grafana è la build del plugin. Per crearla si devono eseguire due script all'interno della cartella contenente il codice del plugin. I due script sono:

- `npm install`

  Per installare tutte le dipendenze presenti nel file di configurazione package.json

- `npm run dev`

  Per creare la cartella 'dist' contenente la build del plugin che andrà inserita nel container Grafana

Per copiare questa cartella in maniera automatica, bisogna modificare il file docker-compose.yml. Le cartelle da inserire nella cartella dove è presente il docker-compose.yml sono due: dist e .config. La prima contiene la build del plugin mentre la seconda la configurazione che Grafana deve adottare per accettare un plugin unsigned. Un esempio di docker-compose.yml per l'aggiunta di un plugin è il seguente:

```
grafana:
  build:
    context: ./.config
    args:
      grafana_image: grafana
  container_name: grafana
  ports:
    - "3000:3000"
  volumes:
    - ./dist:/var/lib/grafana/plugins/nomeplugin
```

Figura 3.14: Codice di esempio docker-compose.yml

Nomeplugin deve essere modificato con il nome del plugin. Con questa configurazione il plugin creato sarà presente nella lista dei possibili pannelli durante la creazione delle dashboard.

Capitolo 4

Embedding in applicazioni terze

4.1 Metabase

Dopo aver creato la dashboard, per generare l'url dell'iframe è possibile seguire due diverse strade: una più semplice per creare un iframe pubblico di semplice visualizzazione ed una più complessa che incorpora la possibilità di inserire nell'iframe anche le variabili utilizzate per la visualizzazione. Questa seconda opzione è utile se si vuole inserire la possibilità di aggiornare i dati visualizzati all'interno dell'applicazione in cui verrà inserito l'iframe.

La generazione dell'url da inserire avviene nella pagina dell'iframe attraverso l'apposito menù superiore 'Condividi'. Come anticipato, verrà richiesto se creare un link pubblico per usi statici ed uno detto 'incorporato' per utilizzi più complessi.

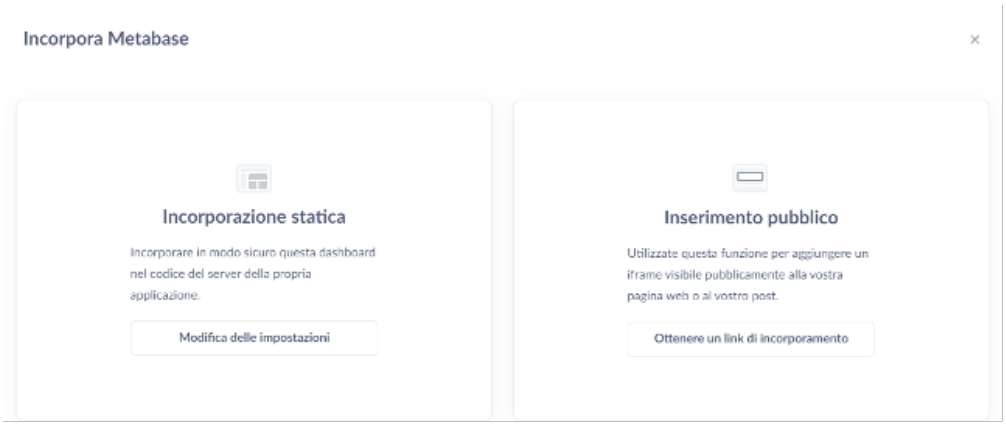

Figura 4.1: Menù creazione url iframe Metabase

Il link pubblico viene subito generato e sarà così formato:

```
http://domain/public/uid
```

Dove uid è l'id della dashboard da visualizzare.

Per quanto riguarda il link 'incorporato' bisogna prima effettuare una configurazione per poterlo utilizzare.

Utilizzando questa tipologia di iframe è possibile integrare in esso anche dei menù a tendina generati da Metabase per poter aggiornare la visualizzazione dei grafici. Per farlo, si deve selezionare la voce 'Parametri' presente nel menù e selezionare quali sono i parametri di query che devono essere mostrati. Le opzioni sono tre:

- Disabilitato se si vuole solamente mostrare i grafici senza possibilità di aggiornamento
- Modificabile se si vuole dare possibilità all'utente di modificare la visualizzazione
- Bloccato se si vuole mostrare all'utente i parametri di query ma senza dare possibilità di modifica

Si deve cliccare il pulsante 'Pubblicato' per rendere possibile l'embed della dashboard con le modifiche ai parametri desiderate e si deve effettuare una nuova pubblicazione ad ogni modifica effettuata. Nell'applicazione in cui si decide di inserire bisogna installare il modulo jsonwebtoken [10] con il comando:

```
npm install jsonwebtoken
```

Ed aggiungere il seguente codice per inserire l'iframe:

```
var jwt = require("jsonwebtoken");
var METABASE_SITE_URL = "http://domain";
var METABASE_SECRET_KEY = "secretKey";
var payload = {
  resource: { dashboard: N },
  params: {},
  exp: Math.round(Date.now() / 1000) + (10 * 60)
};
var token = jwt.sign(payload, METABASE_SECRET_KEY);
var iframeUrl = METABASE_SITE_URL + "/embed/dashboard/"
    + token + "#bordered=true&titled=true";
```

Figura 4.2: Esempio configurazione iframe Metabase

I parametri utilizzati sono:

- Domain che rappresenta il dominio di Metabase

- secretKey generata da Metabase
- N generato da Metabase univoco per ogni dashboard

iframeUrl sarà l'url da inserire nell'iframe per la sua corretta visualizzazione.

4.2 Kibana

Come per Metabase, il primo passaggio da seguire è la creazione e salvataggio della dashboard. Una volta creata, si può procedere con la creazione dell'iframe che, anche in questo software, può avvenire in due diversi modi. Modi che però portano a conclusioni diverse. Nel caso di Kibana si possono generare url statici (Snapshot) che salvano al loro interno lo stato attuale della dashboard. Quindi in caso di modifica della dashboard, la sua visualizzazione non verrà aggiornata nell'applicazione in cui viene inserito. La seconda strada consente la dinamicità dell'iframe (Saved object) quindi ogni aggiornamento della dashboard si rifletterà nella sua visualizzazione.

Per poter creare l'url, è presente il pulsante 'Share' all'interno della pagina contenente la dashboard. Il menù a tendina che si aprirà consentirà di scegliere l'opzione appena descritta.

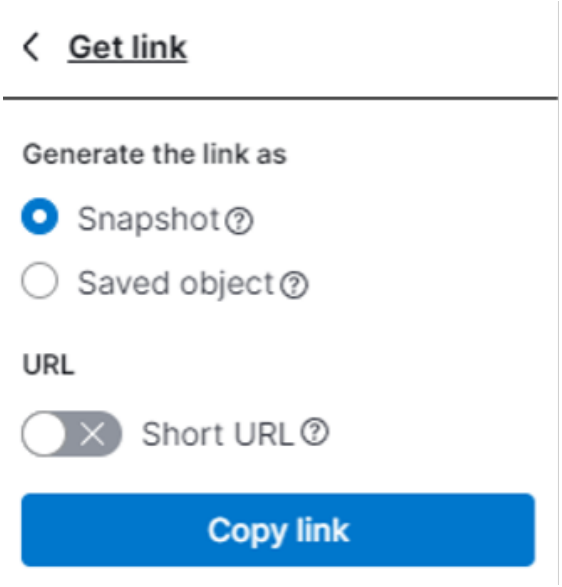

Figura 4.3: Menù creazione url iframe Kibana

L'opzione 'Short URL' consente di generare un url compresso utile se si vuole utilizzare il link non come sorgente in un iframe ma come una pagina web poiché alcuni browser presentano limiti di lunghezza per quanto riguarda gli url.

L'url che verrà generato sarà differente a seconda della configurazione scelta. La parte comune sarà:

`http://domain/app/dashboards#/view/uid?`

Il proseguo invece dipenderà dal tipo di url scelto e conterrà la configurazione e l'intervallo temporale dei grafici da visualizzare.

4.3 Grafana

Come già detto, la funzione principale di Grafana è quella di creare grafici personalizzati per poi andarli ad inserire all'interno di applicazioni web. L'integrazione viene fatta attraverso più passaggi di seguito elencati.

Configurazione

Per iniziare bisogna creare una dashboard e creare il panel che si vuole visualizzare utilizzando la datasource desiderata e la visualizzazione scelta. In questo passaggio è possibile personalizzare lo stile del panel come colore, dimensione del grafico, scala del grafico, range temporale e altro ancora.

Generare l'URL di embed

Una volta creato il panel con il grafico, si passa alla generazione dell'url che conterrà il grafico. Si procede aprendo le opzioni del panel scelto e si clicca su 'Share'. Il menù che si aprirà conterrà le opzioni per l'embed. È possibile copiare solamente l'URL per avere una pagina contenente solo il grafico o, selezionando la scheda 'Embed' in alto, è possibile copiare il codice HTML contenente l'iframe per il render del grafico.

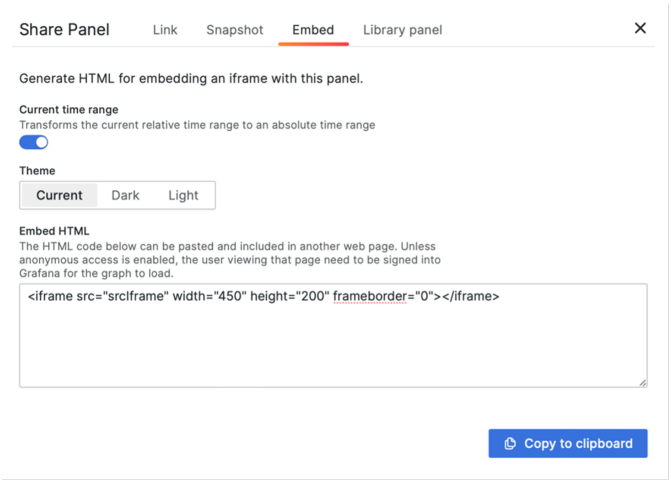

Figura 4.4: Menù creazione url iframe Grafana

In questa schermata è possibile selezionare il tema che dovrà essere dato all'iframe. Le scelte sono 'Current' se si vuole utilizzare il tema utilizzato nel Grafana di appartenenza o 'Dark' e 'Light' se si vuole avere un tema differente. Cliccando su 'Copy to clipboard' il codice HTML sarà copiato e sarà possibile utilizzarlo in applicazioni terze.

L'url che viene generato conterrà le proprietà che servono a Grafana per renderizzare il panel scelto. Un esempio di url contenuto in un iframe è il seguente:

```
https://domain/d-solo/UID/
        prometeo?orgId=1&from=X&to=Y&panelId=1&var-Z=A
```

- domain rappresenta l'url di base per accedere a Grafana
- d-solo sta per dashboard solo ed indica che il grafico rappresenta un singolo panel
- UID è il codice associato alla dashboard ed è generato da Grafana
- Prometeo?ordId=1indica l'organizzazione di appartenenza del container Grafana
- From e to sono i campi che contengono i due estremi del range temporale del grafico da visualizzare
- panelId indica l'id del panel da mostrare
- Infine, è presente la dichiarazione delle variabili che verranno utilizzate all'interno del grafico per il render

Configurazioni aggiuntive

Per rendere possibile l'embed in iframe, si deve modificare la configurazione di base di Grafana che di default vieta l'embed. Per aggiungere questa funzionalità sono presenti due vie differenti:

- È possibile modificare il codice sorgente andando a modificare il file defaults.ini presente nella cartella conf. Il flag da modificare è chiamato allow-embedding. Deve essere true per permettere l'embed
- Questa modifica può essere fatta anche non andando a modificare il codice sorgente ma si possono inserire, in fase di build del container, delle modifiche al file di configurazione. Nella cartella contenente il docker-compose.yml si aggiunge un file chiamato grafana.ini in cui vengono inseriti i flag da modificare. In questo caso si aggiunge la riga:

  ```
  allow_embedding = true
  ```

Questo file sarà inserito nel campo volumes con questa dicitura:

```
./grafana.ini:/etc/grafana/grafana.ini
```

In questo modo le modifiche aggiunte in grafana.ini saranno presenti nel container appena creato.

Nel caso in cui i pannelli di Grafana richiedano autenticazione, questa dovrà essere configurata attraverso l'utilizzo di token di autenticazione. Se, invece, si vuole rendere pubblico l'accesso ai panel di Grafana si deve aggiungere la modifica ad un secondo flag chiamato [auth.anonymous] che si default è false ma deve essere modificato a true utilizzando uno dei due modi elencati in questo paragrafo.

Aggiornare l'iframe

L'url, quindi, contiene la configurazione per il render di un singolo grafico. Se si vuole mostrare un nuovo grafico all'interno dello stesso spazio, l'url deve essere aggiornato di conseguenza modificando i campi riguardanti il panelId o le variabili utilizzate per il grafico. Questo può essere fatto inserendo uno stato locale nell'applicazione web di nome srcIframe e modificarlo di conseguenza in seguito al click o alla presenza di una nuova configurazione.

```
const [srcIframe, setSrcIframe] = useState(calcolaUrl('1h'));
const [ts, setTs] = useState('1h');
useEffect(() => {
setSrcIframe(calcolaUrl(ts));
}, [ts]);
return <>
<Button onClick={() => setTs('3h')}>3h</Button>
<iframe
    title={'grafana'} id={'grafana'} src={srcIframe}
    width="100%" height={800}>
</iframe>
</>;
```

Figura 4.5: Esempio aggiornamento url iframe Grafana

In questo esempio la funzione calcolaUrl viene richiamata la prima volta utilizzando 1h come valore di ts quindi il range temporale del grafico sarà di 1h. Al click del bottone 3h, la funzione viene chiamata una seconda utilizzando il nuovo valore di ts aggiornando il grafico mostrato di conseguenza con un range temporale di 3h.

Limiti

La configurazione appena vista consente, ad ogni aggiornamento delle variabili, di renderizzare un nuovo grafico in pagina. Questo viene eseguito attraverso numerose richieste

verso Grafana che andrà ad appesantire la rete. Molte richieste sono superflue dato che il grafico è già presente nell'applicazione web e l'unica richiesta utile ai fini del nuovo render è quella verso la datasource per ricevere i nuovi dati da visualizzare. Una situazione nella quale non è un problema avere svariate richieste si presenta in piccole applicazioni web dove i grafici da mostrare sono ridotti. In caso di grandi database o svariati sensori, alleggerire il carico sulla rete diventa fondamentale. Nei seguenti capitoli verranno mostrate più soluzioni a questa problematica appena descritta.

Capitolo 5

Comunicazione con iframe

La postMessage [20] è un metodo presente nell'interfaccia Window di Javascript. Viene utilizzato prevalentemente nel momento in cui si voglia scambiare informazioni tra più finestre o con una finestra che contiene un iframe come nel nostro caso. Questo viene chiamato sull'iframe.

Può essere considerata una soluzione efficiente al problema del caricare un nuovo iframe poiché consente di interagire dall'esterno con il codice interno che lo ha generato. Questo si riflette in un minor consumo di potenza di calcolo e maggior efficienza poiché l'iframe non viene ricaricato totalmente ma, se correttamente utilizzata, consente di ricaricare solo i dati ed aggiornare la visualizzazione del panel in pagina.

5.1 Come funziona

Il primo passaggio è effettuato recuperando l'iframe di destinazione utilizzando il metodo document.getElementById(name) con parametro name che rappresenta l'id dell'iframe. Successivamente viene chiamato il metodo in questione che accetta due parametri principali: message e targetOrigin.

- Message L'effettivo messaggio da inviare alla destinazione. Questo può essere qualsiasi cosa presente in Javascript come un numero, una stringa o un oggetto serializzato.

- targetOrigin Rappresenta l'url di destinazione del messaggio. Fondamentale per la sicurezza. Nel caso in cui questo non è presente, la postMessage manda il message a qualsiasi destinazione.

```
let localUrl //Url presente nell'iframe
let frame = document.getElementById(id);
if (iframe != null) {
                const message = {
            // panelId o variables
        }
        iframe.contentWindow.postMessage(message, localUrl);
}
```

Figura 5.1: Esempio postMessage

Questo per quanto riguarda lato mittente del messaggio. All'interno dell'iframe invece bisogna implementare la gestione del messaggio in entrata. Questo avviene aggiungendo un listener alla pagina usando window.addEventListener('message',function(event)). All'interno di event sono presenti due campi fondamentali: origin e data.

- Origin Contiene l'url del mittente del messaggio. Viene utilizzato per verificare che il messaggio proviene dalla fonte che ci aspettiamo.
- Data Contiene il valore o oggetto che è stato spedito dal mittente.

```
const reveiveMessage = (event: any) => {
        // event.data: messaggio che arriva dalla postMessage
        // event.origin: l'url di origine del messaggio
        };
        window.addEventListener('message', receiveMessage);
        return () => {
          window.removeEventListener('message', receiveMessage);
        };
```

Figura 5.2: Esempio listener postMessage

5.2 Implementazione

Una volta capito come funzionano questi componenti e come sono collegati fra di loro, si è deciso di duplicarli per non andare ad intaccare la route principale durante la fase di test e sviluppo.

Per la modifica e l'aggiornamento del panel all'interno dell'iframe verrà utilizzata la postMessage.

L'implementazione è stata svolta seguendo diversi step di implementazione partendo da casi più semplici fino al generare la funzionalità completa finale. La eventListener che

gestisce l'arrivo dei messaggi è stata inserita nella componentDidMount del componente PanelStateWrapper.tsx che, come detto prima, si occupa del render effettivo del grafico. La componentDidMount è la funzione che viene chiamata nel momento del render del componente quindi aggiungendo la listener in questa funzione, il canale di ascolto viene generato al primo render e rimane aperto fin quando il componente non viene eliminato dalla visualizzazione. Gli step di modifica sono stati: modifica asse temporale usando i campi from e to, modifica asse temporale usando una variabile ts, modifica grafico utilizzando le variabili nella loro interezza.

Modifica asse temporale usando from e to

Questa prima modifica ha coinvolto la ricerca della funzione all'interno di Grafana che gestisce il timeRange del grafico mostrato.

La funzione in questione è onChangeTimeRange(timeRange: AbsoluteTimeRange) presente in PanelStateWrapper.tsx.

```
onChangeTimeRange = (timeRange: AbsoluteTimeRange) => {
    this.timeSrv.setTime({
      from: toUtc(timeRange.from),
      to: toUtc(timeRange.to),
    });
  };
```

Figura 5.3: Codice funzione aggiornamento timeRange Grafana

Il parametro timeRange che accetta in ingresso la funzione è composto da due campi: from e to. Questi due campi rappresentano gli estremi dell'asse temporale da mostrare nel grafico.

```
const sendMessageToGrafana = () => {
      const message = {
        from: from,
        to: to,
      };
      const grafanaIframe = document
                    .getElementById('grafana-iframe');
      grafanaIframe.contentWindow
                    .postMessage(message, srcIframe);
    };
```

Figura 5.4: Esempio sender postMessage

All'interno del message, quindi, si è inserito un oggetto contenente questi due campi e, chiamando la funzione sopracitata, l'asse temporale del grafico viene aggiornato correttamente.

Modifica asse temporale usando una variabile ts

Dato che l'obiettivo finale di questo lavoro è la modifica del grafico senza ricaricare tutto l'iframe, il passo successivo ha riguardato l'utilizzo delle variabili di Grafana.

Come primo caso si è passato dalla modifica specifica dei campi from e to all'utilizzo di una variabile ts che rappresenti la lunghezza dell'asse temporale. La variabile in questione viene aggiunta ai parametri del grafico come relative time diventando il valore da rappresentare partendo dall'istante di render del grafico andando indietro del valore di ts (Es: 3h, 6h, 1d, 1w, 1m).

Le variabili in Grafana sono salvate come var-x dove x è il nome della variabile. Per raggiungere questo obiettivo, il message da mandare con la postMessage presente nella pagina padre è stato modificato di conseguenza. L'oggetto è stato sostituito con un array di coppie chiave/valore dove chiave è il nome della variabile da aggiornare e valore è il nuovo valore.

```
const sendMessageToGrafana = () => {
      const message = {
        variables:
          { key: "ts", value: ts }
      };
      const grafanaIframe = document
                    .getElementById('grafana-iframe');
      grafanaIframe.contentWindow
                    .postMessage(message, srcIframe);
    };
```

Figura 5.5: Esempio sender iframe Grafana

Per quanto riguarda il container Grafana, si è cercato come le variabili vengono create, aggiornate e gestite all'interno del codice. La listener viene aggiornata di conseguenza:

- Usando la getTemplateSrv si accede al template che contiene le variabili.
- Con la getVariables si ottiene il vettore che contiene le variabili.
- Successivamente si cerca, se presente, la variabile ts.
- Grafana utilizza il campo current per il valore attuale della variabile, si aggiorna questo valore con il nuovo valore ricevuto dalla postMessage.

- Si chiama la funzione init passando come parametro il nuovo oggetto con la variabile aggiornata e si chiama la funzione onRefresh per far aggiornare il grafico renderizzato.

```
const srv = getTemplateSrv();
const variables = srv.getVariables();
let tmp = variables.find((v) => v.name === 'ts');
let newV = { ...tmp };
newV.current = { ...tmp.current, value: key };
srv.init(newV);
this.onRefresh();
```

Figura 5.6: Aggiornamento variabile codice sorgente Grafana

Modifica usando solo le variabili Grafana

Una volta compreso come poter manipolare una variabile ts, si è creata una funzione in grado di ricevere qualsiasi coppia chiave/valore e gestire questo array per aggiornare tutte le variabili ricevute. L'array di variabili ricevuto dalla getVariables è read-only quindi da questo si crea una copia modificando i valori current delle variabili e poi, come fatto nel precedente caso, si chiamano init e onRefresh.

```
const srv = getTemplateSrv();
const variables = srv.getVariables();
const newVariables: TypedVariableModel[] = [];
let tmp: (VariableWithOptions & TypedVariableModel) | undefined;
let newV: VariableWithOptions & TypedVariableModel;
change.forEach((c) => {
tmp = variables.find((v) => v.name === c.key)
                    as VariableWithOptions & TypedVariableModel;
if (tmp !== undefined) {
newV = { ...tmp };
newV.current = { ...tmp.current, value: c.value };
newVariables.push(newV);
}});
if (newVariables.length > 0) {
srv.init(newVariables);
this.onRefresh();
}
```

Figura 5.7: Aggiornamento più variabili codice sorgente Grafana

Modifica del panelId

Una seconda modifica che si è scelto di implementare, oltre alla modifica delle variabili con conseguente aggiornamento del grafico mostrato, è quella di renderizzare un panel all'interno dello stesso iframe senza ricaricare tutto l'iframe.

Così come la modifica delle variabili, la modifica è avvenuta utilizzando una postMessage come mezzo di comunicazione fra pagina esterna e iframe di Grafana. La listener in ricezione all'interno di Grafana è stata inserita all'interno di GrafanaRoute.tsx. Per garantire che la listener venga eseguita ed il canale venga creato, è stata inserita all'interno di una useEffect priva di proprietà in modo da venire eseguita solamente al primo render della route.

```
useEffect(() => {
    const receiveMessage = (event: {
      data: {
        panelId?: number;
        variables?: Array<{ key: string; value: string }>;
        timeRange?: AbsoluteTimeRange;
      };
    }) => {
      if (event.data.panelId !== undefined) {
        setPanelId(event.data.panelId);
      }
    };
    window.addEventListener('message', receiveMessage);
    return () => {
      window.removeEventListener('message', receiveMessage);
    };
  }, []);
```

Figura 5.8: Listener panel id codice sorgente Grafana

Il message che viene ricevuto contiene solamente il panelId del nuovo pannello da mostrare. Questo valore viene usato per modificare uno stato PanelId aggiunto al componente.

Questo stato interno è stato aggiunto alle proprietà passate al componente SoloPanelPage.tsx in modo da propagare la modifica. All'interno del componente, la funzione chiamata nel momento di nuove proprietà in ingresso è componentDidUpdate. In questa funzione è stato aggiunto il controllo sul panelid confrontando il vecchio id, presente nelle vecchie proprietà, con quello nuovo appena ricevuto. Nel caso in cui questi due siano diversi, viene richiamata la funzione getPanelByUrlId sulla dashboard passando come

parametro il panelid. La funzione ritorna il panel se presente o in caso di errore viene mostrato all'utente la dicitura: Panel with id id not found.

```
if (!prevProps.queryParams ||
        prevProps.queryParams.panelId !== queryParams.panelId)
    {
      const panel = dashboard
                .getPanelByUrlId(this.getPanelId().toString());
      if (!panel) {
        this.setState({ notFound: true });
        return;
      }
      if (panel) {
        dashboard.exitViewPanel(panel);
      }
      this.setState({ panel, notFound: false });
      dashboard.initViewPanel(panel);
    }
```

Figura 5.9: Aggiornamento panel codice sorgente Grafana

Il nuovo panel viene passato come proprietà ai componenti successivi e la visualizzazione viene aggiornata di conseguenza.

Dopo un primo test, questa nuova funzionalità risulta essere funzionante ma il nuovo panel che si viene a generare è privo di stile (colori, linee, ecc.). L'errore riscontrato è nel caricamento di alcune proprietà che il panel deve avere. Queste proprietà vengono inserite all'interno del componente DashboardPanel con la funzione onPanelLoad. La soluzione è stata quella di aggiungere la funzione componentDidUpdate, in modo che ad ogni cambio di proprietà in ricezione venga richiamata. All'interno di quest'ultima, è stata inserito, come in SoloPanelPage, un controllo per verificare che il panel precedente e quello nuovo siano differenti. Se risultano diversi, viene richiamata la funzione onPanelLoad che aggiunge le proprietà mancanti al panel risolvendo il problema di visualizzazione riscontrato durante il primo test.

```
componentDidUpdate(prevProps: Props) {
  const { panel } = this.props;
  if (!prevProps.panel || prevProps.panel.id !== panel.id) {
    this.props.panel.isInView = !this.props.lazy;
    if (!this.props.lazy) {
      this.onPanelLoad();
    }
  }
}
```

Figura 5.10: Aggiornamento stile panel codice sorgente Grafana

Per far coesistere le due receiver collocate in PanelStateWrapper e GrafanaRoute, si è deciso di dare nei nomi specifici agli oggetti da inserire nella postMessage per evitare comportamenti non voluti da parte di Grafana. Questo accade perché singola postMessage attiva entrambe le receiver presenti. Il controllo consiste nel verificare quale modifica l'utente vuole effettuare sull'iframe. Nel caso in cui si voglia modificare le variabili dell'iframe, il nome dell'oggetto da inserire è 'variables' mentre se si vuole modificare il panel mostrato, l'oggetto prende il nome di 'panelId'.

Capitolo 6

Aprire una Pull Request in Grafana

Una volta completata l'implementazione delle modifiche e verificato la loro completa funzionalità, il passo successivo è stato quello di aprire una Pull Request sul repository pubblico di Grafana in modo da implementare questa modifica nella versione ufficiale e renderla disponibile a tutti.

Per effettuare questa operazione si sono svolti diversi passaggi di seguito elencati.

6.1 Fork del repository

Per creare una Pull Request bisogna prima creare una copia del repository originale attraverso la funzione di Fork offerta da GitHub. Questo porta alla creazione di un clone nel quale inserire le proprie modifiche in modo che siano segnalate quando verrà creata la Pull Request.

6.2 Apportare le modifiche

Nel repository appena creato bisogna inserire le proprie modifiche seguendo le linee guida che Grafana [4] mette a disposizione per avere uniformità nella stesura del codice. Queste linee guida includono ad esempio la scelta di nomi da dare alle variabili utilizzate, alle funzioni e alla formattazione del codice da inserire. Successivamente effettuare un commit delle modifiche inserendo come commento un breve riassunto di quest'ultima.

6.3 Creare la pull request

Una volta effettuato il push delle modifiche su GitHub, si può creare la Pull Request.

Questo viene fatto attraverso la pagina del repository su github.com. Cliccando su 'Contribute' si apre un menù a tendina contenente la dicitura 'Open pull request'. La nuova pagina che verrà aperta conterrà il menù di modifica e creazione della pull request. Nello specifico caso di Grafana, il messaggio iniziale contiene una lista di domande preimpostate al quale bisogna rispondere per dare delle informazioni al team di Grafana che andrà a revisionare la pull request per verificare che sia una modifica correttamente implementata. Alcune delle domande sono le seguenti:

- Eseguire i test prima di aprire la pull request
- Modificare la documentazione se necessario
- Aprire una pull request come 'Draft PF' se la funzionalità è ancora in sviluppo e non funzionante
- Effettuare un rebase se non è possibile eseguire un merge automatico delle modifiche
- Dare un nome che contenga la categoria di appartenenza della pull request per una più facile comprensione degli sviluppatori. È possibile visionare le varie categorie nella documentazione di Grafana
- Se si sostiene che debba essere inserita come modifica importante, si deve aggiungere 'add to what's new' in modo da inserire l'etichetta corrispondente
- Inserire una spiegazione della propria modifica in maniera dettagliata per aiutare gli sviluppatori a comprenderne il funzionamento, le possibilità che offre e i motivi per il quale è necessario inserirla

Sono state citate le etichette. Sono degli aiuti che vengono dati agli sviluppatori per filtrare le varie pull request create su GitHub. A seconda del titolo alcune vengono già aggiunte in automatico. Un esempio è dato dalla categoria che viene data alla pull request, nel caso di modifiche al frontend verrà aggiunta l'etichetta 'area/frontend'. Un altro esempio è dato dal contenuto del messaggio di supporto dato agli sviluppatori. Nella sua stesura si deve aggiungere o negare la volontà ad essere aggiunti come aiutanti allo sviluppo, l'etichetta associata 'add to changelog' verrà aggiunta in maniera manuale da uno degli sviluppatori di Grafana.

6.4 Attendere la revisione del merge

Una volta inserita la pull request, GitHub assegnerà alla pull request appena creata una lista di possibili revisori che andranno a verificare la possibilità di inserire la nuova funzione nella successiva release di Grafana.

6.5 Checks in Grafana

Dopo che un revisore ha preso in carico il controllo della pull request, vengono sbloccati una serie di controlli che vengono effettuati per verificare la validità delle nuove implementazioni che si vogliono inserire nella nuova release di Grafana. In questo caso specifico i controlli riguardano i seguenti parametri:

- Correttezza del codice per quanto riguarda i tipi che vengono dati alle nuove variabili inserite quindi int/string/ecc. Devono tutti essere espliciti e non devono esserci ambiguità

- Ordine alfabetico all'interno della sezione degli import per una più facile ricerca e leggibilità del codice

- Corrette spaziature all'interno delle funzioni

- Uso del camelCase per i nomi che vengono assegnati alle variabili utilizzate

Questi controlli vengono effettuati in automatico ad ogni push sul branch del repository utilizzato per creare la pull request. Per verificare che il codice inserito sia corretto, prima di effettuare il commit, è possibile utilizzare dei comandi da terminale per verificare che tutti i controlli siano superati. Questi comandi sono:

- Yarn run prettier:check che verifica la correttezza delle spaziature all'interno del codice

- Yarn run typecheck che verifica se I tipi dati alle variabili siano corretti con il loro utilizzo successivo e verifica che ogni variabile abbia un tipo e non sia del tipo 'any'

Dopo che tutti i controlli risultano superati con successo, il revisore che ha preso in carico la pull request può decidere se effettuare il merge nel main branch di Grafana o meno.

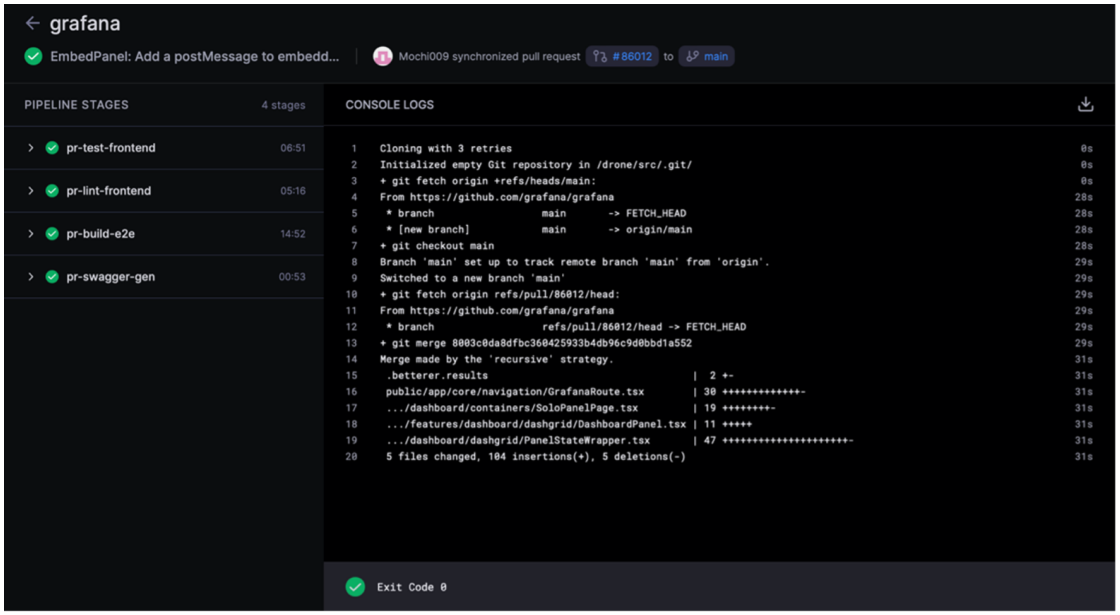

Figura 6.1: Risultato build pull request Grafana

Questa è la schermata di Github dove sono presenti i check che vengono fatti sul repository inserito nella pull request prima che venga valutata dagli sviluppatori del team Grafana.

Capitolo 7

Comunicazione con iframe e plugin

Dopo aver esplorato nei dettagli il processo che ha portato all'inserimento della postMessage all'interno del codice sorgente di Grafana per raggiungere l'obiettivo desiderato, il focus passa su una seconda modalità per raggiungere lo stesso scopo utilizzando i plugin di Grafana definiti in precedenza.

Prima di definire le modifiche effettuate, è utile descrivere il flusso di richieste che arrivano a Grafana per la creazione del grafico da inserire nell'iframe.

I passaggi ad alto livello eseguiti da sono i seguenti:

- Ricerca della dashboard contenente il panel da visualizzare
- Ricerca del panel all'interno della dashboard
- Viene effettuata la query dei dati da passare al panel
- Viene restituito il grafico appena generato utilizzato i dati presi dalla query

Da questa spiegazione è facile immaginare come sia impossibile inserire del codice nel panel per aggiornare i dati da visualizzare. Questo perché al panel arrivano già i dati da visualizzare, non potendo richiederne di nuovi come fatto durante lo sviluppo all'interno del codice sorgente di Grafana. Per aggirare questo ostacolo è stata utilizzata nuovamente la postMessage per interagire con l'iframe dall'esterno con le opportune modifiche rispetto al suo inserimento nel codice sorgente di Grafana.

Per raggiungere l'obiettivo desiderato, si deve effettuare una fetch per i dati all'interno del plugin. La sua realizzazione si compone di più passaggi. Viene utilizzata una libreria di Grafana per risalire all'identificativo della datasource da utilizzare per la query, si crea

l'url di destinazione della query aggiungendo le variabili interessate al fondo, si manipolano i dati ricevuti dalla fetch e si utilizzano per aggiornare la visualizzazione del panel.

La libreria di Grafana coinvolta è @grafana/runtime. Contiene al suo interno il metodo getDataSourceSrv che viene utilizzato per risalire alla configurazione della datasource con il comando:

```
const d = await getDataSourceSrv().get('nome');
```

Per 'nome' si intende il nome dato alla datasource. Verrà utilizzato il campo uid di d all'interno dell'url della query per i nuovi dati. I parametri di richiesta dell'api dovranno essere inseriti come stati all'interno del codice. La funzione di fetch, invece, verrà inserita in una useEffect che ha come parametri, appunto, i parametri di richiesta per eseguirne una nuova ad ogni aggiornamento di essi attraverso la ricezione della postMessage.

La postMessage, quindi, riceverà come 'message' i nuovi parametri ed aggiornera gli stati locali collegati. Di seguito un esempio di listener:

```
useEffect(() => {
    const receiveMessage = (event: {
      data: {
        ts: string;
      }
    }) => {
      setTs(event.data.ts);
    }
    window.addEventListener('message', receiveMessage);
    return () => {
      window.removeEventListener('message', receiveMessage);
    };
  });
```

Figura 7.1: Listener postMessage Panel plugin

In questo caso la postMessage manderà un oggetto data contenente il campo ts. L'aggiornamento dello stato ts andrà ad attivare una seconda useEffect contenente la nuova fetch dei dati con successivo loro aggiornamento. L'url che dovrà essere utilizzato per la fetch sarà così formato:

```
http://domain/api/datasources/proxy/uid + d.uid + /api/?ts=3h
```

d.uid è l'uid della datasource recuperato precedentemente con l'utilizzo della libreria di Grafana. 3h è un esempio del valore che può avere il parametro ts.

La funziona per effettuare la fetch dei dati sarà così strutturata:

```
async function fetchData(ts: string) {
  try {
   const d = await getDataSourceSrv().get('nome');
const dataTmp = await
        fetch('http://domain/api/datasources/proxy/uid/'
        + d.uid + '/api/?ts=' + ts)
        .then(response => response.json());
    return dataTmp;
  }
  catch (error) {
    return [];
  }
}
```

Figura 7.2: Funzione per il fetch dei dati Panel plugin

La useEffect, invece, incaricata dell'aggiornamento dei dati sarà la seguente:

```
useEffect(() => {
  const newData = fetchData(ts);
  setData(newData);
}, [ts]);
```

Figura 7.3: useEffect per effettuare una nuova fetch dei dati

I campi presenti in data saranno differenti dai dati che Grafana offre per la generazione dei grafici e saranno differenti a seconda del tipo di datasource che si utilizza.

Questa soluzione può essere conveniente nel caso in cui venga utilizzato lo stesso plugin per più visualizzazioni ma risulta essere poco efficiente se i tipi di panel da utilizzare sono molteplici poiché si traduce nel creare n nuovi plugin quanti sono i panel da utilizzare e, in caso di aggiornamento delle datasource in Grafana, tutti i plugin devono essere modificati e ripubblicati/ricaricati di conseguenza per proseguirne il loro funzionamento ed utilizzo.

Capitolo 8

Conclusioni

In questo lavoro di tesi si sono esplorati alcuni tra i più importanti servizi di dashbarding gratuiti disponibili sul mercato: Metabase, Kibana e nello specifico Grafana. Il loro utilizzo diventa fondamentale nel campo di visualizzazione e analisi dei dati salvati in database. Questo facilita gli utenti nel momento in cui vogliano monitorare metriche aziendali o sensori.

Metabase si distingue per la sua facilità d'uso anche da parte di team non esperti nel campo delle query poiché è possibile creare dashboard interattive utilizzando la sola interfaccia grafica molto intuitiva e semplice. Risulta essere uno strumento adatto a piccole e medie imprese.

Kibana, facente parte della suite Elastic, offre grandi capacità di ricerca e analisi che si dimostrano utili nella realizzazione di dati di log e monitoraggio. La sua integrazione con Elastic permette di eseguire ricerche avanzate o di visualizzare dati in tempo reale. Questo risulta utile per l'analisi di dati di grandi dimensioni e per il monitoraggio real time delle applicazioni.

Grafana, invece, risulta il più versatile poiché è l'unico che può integrarsi con una vasta gamma di fonti di dati con una personalizzazione più dettagliata delle dashboard. Una delle caratteristiche che lo differenziano, infatti, è la possibilità di effettuare l'embed di un singolo grafico invece che di un'intera dashboard in modo da integrarsi facilmente in applicazioni terze. La sua vasta gamma di plugin permette di ampliarne le sue funzionalità. Come visto, è anche possibile creare un nuovo plugin in base alle proprie esigenze. Questo tipo di approccio aumenta la flessibilità di Grafana andando ad aumentare anche il tempo di risposta nel momento di nuove esigenze da parte degli utenti finali.

Il passo successivo affrontato è stato aumentare l'efficienza di aggiornamento di un iframe Grafana. Il risultato ottenuto utilizzando la postMessage è stato positivo ed efficace. La trasmissione di messaggi ha portato il numero di richieste ad una singola richiesta verso la datasource per i nuovi dati da visualizzare. L'utilizzo all'interno del codice sorgente

di Grafana o internamente a dei nuovi plugin varia a seconda delle scelte di implementazione. Il primo caso risulta essere efficace se i pannelli generati utilizzando Grafana sono di diverso tipo o con stili di rappresentazione diversi. Lo sviluppo di un plugin con la postMessage al suo interno è una scelta migliore se i pannelli da visualizzare sono in numero ridotto in modo da non dover creare molteplici plugin.

In conclusione, questa analisi ha evidenziato le differenze fra i diversi servizi di dashboarding andando a descrivere come ogni software ha utilizzi specifici nonostante facciano parte della stessa categoria. Invece, l'obiettivo prefissato è stato ampiamente raggiunto mostrando con una riduzione drastica dei tempi di aggiornamento con una crescita in termini di efficienza. Queste due opzioni risultano essere valide in contesti di utilizzo diversi da parte dell'utente finale.

Bibliografia

[1] *Che cos'è l'iframe e come lo usi nel tuo sito web?* URL: https://it.siteground.com/kb/cose-iframe/.

[2] *Codice Sorgente Grafana.* URL: https://github.com/grafana/grafana.

[3] *Connect Grafana to integrations, apps, and more.* URL: https://grafana.com/grafana/plugins/.

[4] *Create a pull request.* URL: https://github.com/grafana/grafana/blob/main/contribute/create-pull-request.md.

[5] *Get started.* URL: https://grafana.com/developers/plugin-tools/.

[6] *Grafana.* URL: https://grafana.com.

[7] *Grafana Crash Course.* URL: https://volkovlabs.io/grafana/.

[8] *Grafana documentation.* URL: https://grafana.com/docs/grafana/latest/.

[9] *Install Kibana.* URL: https://www.elastic.co/guide/en/kibana/current/install.html.

[10] *jsonwebtoken.* URL: https://www.npmjs.com/package/jsonwebtoken.

[11] *Kibana.* URL: https://www.elastic.co/kibana.

[12] *Kibana Guide.* URL: https://www.elastic.co/guide/en/kibana/current/index.html.

[13] *Metabase.* URL: https://www.metabase.com.

[14] *Metabase documentation.* URL: https://www.metabase.com/docs/latest/.

[15] *Plugin management.* URL: https://grafana.com/docs/grafana/latest/administration/plugin-management/.

[16] *Publish or update a plugin.* URL: https://grafana.com/developers/plugin-tools/publish-a-plugin/publish-a-plugin.

[17] *recharts.* URL: https://www.npmjs.com/package/recharts.

[18] *Run Grafana Docker image*. URL: https://grafana.com/docs/grafana/latest/setup-grafana/installation/docker/.

[19] *Running Metabase on Docker*. URL: https://www.metabase.com/docs/latest/installation-and-operation/running-metabase-on-docker.

[20] *Window: postMessage() method*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.