

Distributed Systems and Big Data

Relazione DSBD

Campione Salvatore - 1000056279

Cipriano Mario - 1000056257

My Weather

Descrizione progetto

Il progetto che si è voluto realizzare è quello di un'applicazione che è in grado di notificare ai vari utenti le condizioni meteorologiche delle città che desiderano.L'applicazione è costituita da ben 4 microservizi,di cui parleremo nel dettaglio più avanti .Tali microservizi sono gestiti tramite l'ausilio di Docker e comunicano tra di loro scambiandosi informazioni. Per due di essi ,ovvero i primi due è stato fatto l'utilizzo di Flask,mentre per gli ultimi due si è pensato di utilizzare un broker kafka per la comunicazione e quindi lo scambio di messaggi. Questa applicazione prevede a far sì che ogni utente ,dopo essersi registrato tramite un apposito form di registrazione o dopo essersi loggato, tramite un form di login, e quindi dopo aver inserito le proprie credenziali , possa sottoscriversi a una o più città di interesse e possa definire una serie di vincoli per esse. Per la realizzazione della nostra applicazione sono stati scelti come vincoli la temperatura massima ,la temperatura minima,la presenza di pioggia,espressa in millimetri e calcolata nel range di un'ora e la presenza di neve,anche essa espressa in millimetri e calcolata nel range di un'ora.Per poter reperire le informazioni relative ai dati meteorologici si è previsto l'utilizzo di un servizio di api esterne chiamate (<https://openweathermap.org/api>),che a intervalli regolari va ad essere interrogato e va prelevare le informazioni,per poi essere filtrate sulla base delle richieste effettuate dall'utente. Inoltre qualora dovessero

verificarsi le condizioni specificate dall' utente,il sistema restituisce un alert e di conseguenza l'utente verrà notificato per email.Una volta realizzata l'applicazione ,la fase successiva prevede di andare a configurare un server prometheus e di realizzare un'operazione di scraper su delle metriche scelte da noi ed esposte da un eventuale exporter.Una volta definite tali metriche la parte conclusiva del progetto consisteva nel realizzare un servizio che permetesse di definire un SLA con un set di metriche precedentemente collezionate da prometheus.

Microservizi

User_manager: Un microservizio che gestisce la fase di registrazione e login dell'utente permettendo il salvataggio dei dati all'interno di un database chiamato utenteDB.

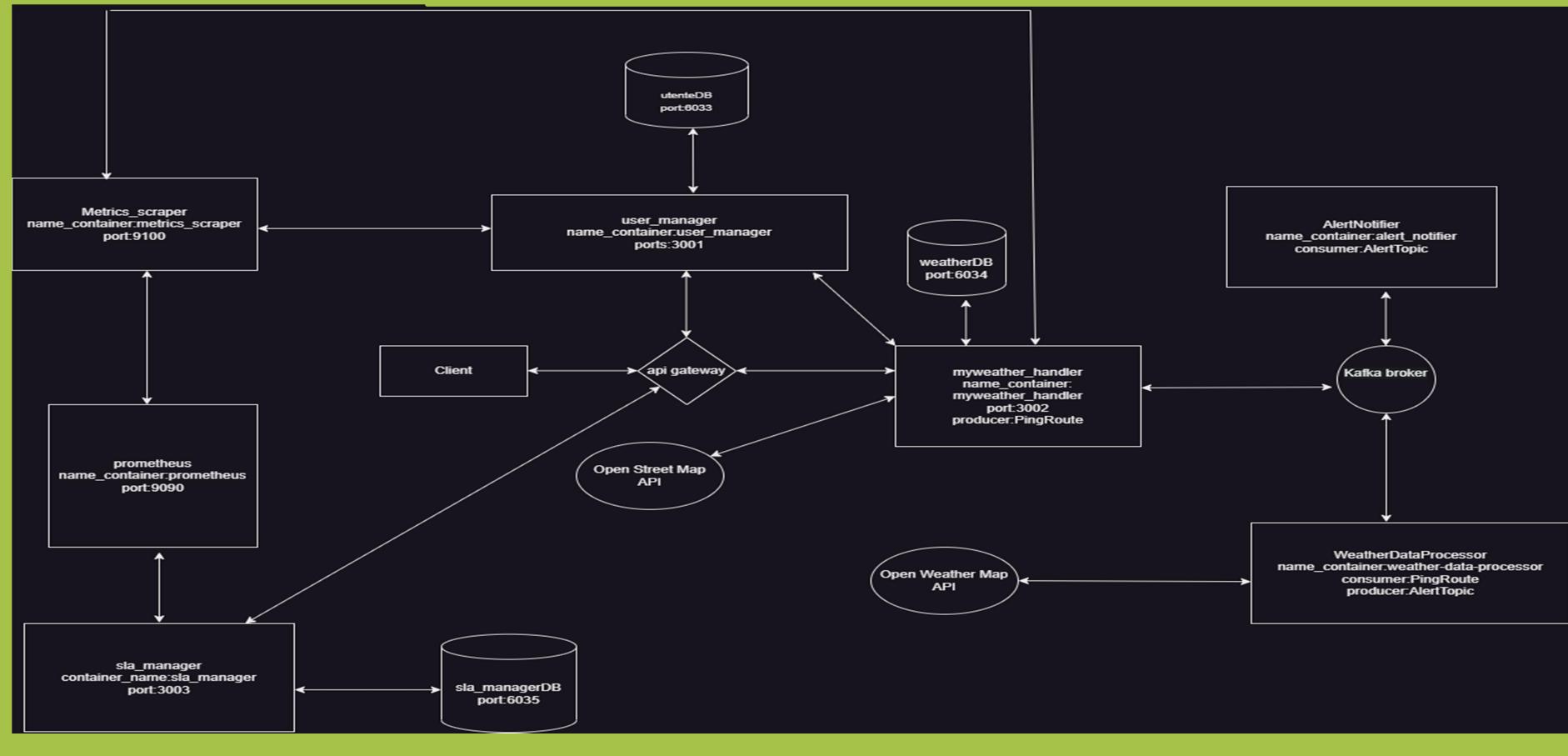
Myweather_handler: Un microservizio che si occupa di gestire i dati che compongono la sottoscrizione dell'utente ad una città,i dati saranno gestiti e inoltrati tramite un form quindi con una richiesta post e dopodichè salvati all'interno di un secondo database chiamato weatherDB.

WeatherDataProcessor: Un microservizio che riceve tramite un messaggio kafka le informazione che il microservizio precedente aveva salvato nel proprio db e sulla base di queste,dopo aver fatto le chiamate alle api confronta i dati e genera gli alert .

AlertNotifier: Un microservizio che una volta che sono stati generati gli alert dal microservizio precedente e quindi ci sono verificate le condizioni alle quali si era sottoscritto l'utente , invia a quest'ultimo una mail per notificarlo.

Schema Architetturale

N.B: Le connessioni fra client,user_manager,myweather_handler e sla_manager con l'api_gateway sono connessioni esterne verso nginx. Le connessioni fra myweather_handler,WeatherDataProcessor e AlertNotifier verso il broker Kafka sono connessioni di tipo publisher-subscriber. Le connessioni dei microservizi verso le API sono connessioni REST esterne,invece il resto sono connessioni REST interne tramite endpoint docker



User_manager

User_manager

Il primo microservizio come accenato sopra gestisce la registrazione e il login dell'utente. Per l'evenienza sono stati realizzati due form all'interno del quale il nostro utente potrà inserire i suoi dati e potrà registrarsi oppure potrà, se già registrato accedere alla homepage.

All'interno di esso è prevista la gestione dell'utente tramite il concetto di sessione, della quale si terrà traccia tramite l'utilizzo di un apposito cookie di sessione che verrà creato quando il nostro utente si registrerà o loggerà ed avrà una durata preimpostata. Grazie a tale cookie l'utente potrà rimanere loggato e rimanere all'interno della homepage fino a quando non scadrà il cookie o fino a quando non deciderà di effettuare il logout, con il quale la sessione e il cookie verranno eliminati. Per la gestione dell'utente inoltre sono stati utilizzati i concetti di `login_manager`, `login_manager_view`, le funzioni `login_user` e `logout_user` e il decoratore `login_required`, che viene utilizzato per verificare che l'utente sia autorizzato prima di accedere alla home. Ovviamente questi sono elementi forniti dal framework Flask.

Il primo microservizio sarà lanciato dal dockerfile, una volta che questo è stato accuratamente configurato. Per la gestione del primo microservizio abbiamo fatto utilizzo di Flask, mettendo in ascolto tale microservizio sulla porta 3001. Per quest'ultimo sono state definite le seguenti route:

User_manager

/register: questa route è stata definita per permettere all'utente di registrarsi e quindi solo dopo aver compilato i campi del form senza errori, ovvero dopo che tutti i controllo sui campi sono stati validati, di accedere alla homepage.

/login: questa route è stata definita per permettere all'utente di loggarsi e accedere alla homepage, dove aver ovviamente effettuato i controlli sull'email e la password che sono stati inseriti nei campi del form dall'utente.

/home: questa route non fa altro che ospitare l'utente una volta che esso si è loggato. Qui in particolare viene utilizzato il decoratore login_required di cui abbiamo parlato sopra.

/logout: questa è invece la route che permette all'utente loggato di uscire dalla sessione e di eliminare il cookie di sessione. Questa route permette all'utente di essere reindirizzato nuovamente alla pagina di login. Qui viene richiamato l'utilizzo della funzione logout_user e del decoratore login_required.

User_manager

/MySubscription: questa è l'ultima route che andiamo a definire per il primo microservizio. Essa restituisce una pagina dedicata alle sottoscrizioni effettuate dall'utente. Permette di accedere ad una sezione della home chiamata le mie sottoscrizioni, dove vengono salvate le sottoscrizioni effettuate dall'utente, che vengono costruite dinamicamente in un file javascript, andando a recuperare i dati dal database e facendoseli restituire sottoforma di json, tramite una fetch.

MyWeather_handler

MyWeather_Handler

Il secondo microservizio invece come visto in precedenza si occupa di tenere traccia delle varie sottoscrizioni che un utente può fare alle diverse città.L'idea è stata quella di realizzare un form che il nostro utente può compilare ,inserendo i valori relativi ai vincoli specificati per quella determinata città. I dati una volta che sono stati inseriti,vengono salvati all'interno di un database mysql che prende il nome di weatherDB. Per garantire anche che le sottoscrizioni siano sempre visibili,senza andare a consultare il database,si è scelto di realizzare una sezione della homepage dove si vanno a inserire dinamicamenete ,tramite javascript quelle che sono le sottoscrizione,costruendole a partire dai dati del database ,che vengono ritornati in formato json.Accanto all'utilizzo del form è stata previsto anche l'utilizzo di una mappa Interattiva,che al riempimento dei campi del form relativi alla città e al paese,positonava un marker sulla mappa e permetteva di visualizzare un piccolo popup con le condizioni meteo previste per quella città.Tale meccanismo è stato realizzato sfruttando le api di Open Street Map.Il secondo microservizio prevedeva anche che periodicamente esso doveva andare a interrogare il nostro database per poter prelevare i dati relativi alla sottoscrizione e poter con essi costruire un messaggio da inoltrare al terzo microservizio. A proposito di questo nel secondo microservizio,oltre ad usare nuovamente utilizzato Flask, andando ad esporre il servizio sulla porta 3002,si è fatto l'utilizzo del broker

MyWeather_handler

kafka,mappato in un apposito container sulla porta 9093.Kafka è semplicemente un broker,che permette di scambiare messaggi fra loro ai vari microservizi. A tal proposito per l'utilizzo di kafka è stata prevista la realizzazione di due producer e di un consumer. Il ruolo del primo produce viene svolto dal secondo microservizio. Abbiamo deciso a tal proposito di andare a realizzare un thread che periodicamente (60 sec) va a interrogare il nostro database e va a prelevare i dati che costituiscono la nostra sottoscrizione.Sulla base dei dati recuperati va costruire un messaggio kafka che quindi ogni 60 secondi verrà inviato al terzo microservizio che svolgerà il ruolo di consumer.

Anche in questo caso il secondo microservizio viene avviato tramite il dockerfile e quindi gestito tramite l'ausilio di docker e dei container.Così come nel primo microservizio anche qui le route che sono state esposte su Flask sono state mappate su un file configuration.conf e gestite tramite un api gateway(nginx).Le route che vengono utilizzate in tale microservizio sono le seguenti:

MyWeather_handler

/SendRequest: questa route viene utilizzata per andare a inviare i dati del form e di salvarli all'interno del database weatherDB.

/get_subscriptions/<email>: questa route invece serve per poter effettuare la fetch lato javascript per poter ottenere i dati dal database in formato json e creare dinamicamente le sottoscrizioni, per poi aggiungerle alla sezioni “le mie sottoscrizioni”

/delete/<id>: questa è una semplice route che all'occorenza dell'utente gli permette di eliminare le sottoscrizioni dalla sezione delle homepage e quindi dal database.

WeatherDataProcessor

WeatherDataProcessor

Il terzo microservizio invece abbiamo deciso di non realizzarlo utilizzando Flask, ma abbiamo deciso di realizzarlo come un normale script python,in quanto abbiamo deciso che quest'ultimo dovesse comunicare con il quarto microservizio solo tramite l'utilizzo di messaggi kafka. Il terzo microservizio svolge il ruolo sia di consumer che di producer. Il ruolo di producer perchè vogliamo che quest'ultimo una volta che ha ricevuto i dati tramite il messaggio kafka inoltratogli dal secondo microservizio vada ad effettuare e processare le richieste alle nostre api esterne e sulla base dei dati ricevuti da quest'ultima effettui dei confronti con i dati del messaggio kafka e generi così gli alert che poi verrano inviati al nostro AlertNotifier. Il ruolo invece di consumer,come già detto prima verrà svolto perchè quest'ultimo riceverà e interpreterà i dati che sono stati inoltrati dal secondo microservizio.

WeatherDataProcessor

Il producer che abbiamo definito prende il nome di “AlertTopic” mentre il consumer si chiama “SubscriptionTopic” e sono stati così definiti:

```
from kafka import KafkaConsumer, KafkaProducer
import json
import requests
import logging

logging.basicConfig(level=logging.INFO) # Imposta il livello di log a INFO o superiore
logger = logging.getLogger(__name__)

# Configurazione del consumatore
bootstrap_servers_consumer = 'kafka:9092'
group_id = 'gruppo-di-consutatori'
auto_offset_reset = 'earliest'
# Chiave api
api_key = '109a274e47fac26e28be9c3d15bbc8e6'

# Creazione del consumatore
consumer = KafkaConsumer(
    'SubscriptionTopic',
    bootstrap_servers=bootstrap_servers_consumer,
    group_id=group_id,
    auto_offset_reset=auto_offset_reset,
    consumer_timeout_ms=5000 # Tempo di attesa in millisecondi
)

bootstrap_servers_producer = 'kafka:9092'
producer = KafkaProducer(bootstrap_servers=bootstrap_servers_producer)
alert_topic = 'AlertTopic'
```

WeatherDataProcessor

La parte relativa invece alle richieste che vengono effettuate alle api esterne viene definita dalla funzione get_weather_data nel seguente modo:

```
def get_weather_data(city, country):
    base_url = 'http://api.openweathermap.org/data/2.5/weather'
    params = {'q': f'{city},{country}', 'appid': api_key, 'units': 'metric'}

    response = requests.get(base_url, params=params)
    if response.status_code == 200:
        return response.json()
    else:
        logger.error(f'Errore nell\'ottenere i dati meteorologici per {city}, {country}. Codice di stato: {response.status_code}')
        return None
```

WeatherDataProcessor

```
while True:
    for kafka_msg in consumer:
        # Elaborare il messaggio ricevuto
        try:
            logger.info("sono nel try")
            message_value_str = kafka_msg.value.decode('utf-8')
            logger.info('Messaggio ricevuto: %s', message_value_str)
            # Decodifica il messaggio JSON
            decoded_message = json.loads(message_value_str)

            # Estra i dati dalla struttura del messaggio
            subscription_id = decoded_message.get('subscription_id')
            email = decoded_message.get('email')
            city = decoded_message.get('city')
            country_code = decoded_message.get('countrycode')
            minTemp = decoded_message.get('minTemp')
            maxTemp = decoded_message.get('maxTemp')
            rain = decoded_message.get('rain')
            snow = decoded_message.get('snow')

            # Fai qualcosa con i dati estratti
            logger.info('Subscription ID: %s', subscription_id)
            logger.info('Email: %s', email)
            logger.info('City: %s', city)
            logger.info('Country Code: %s', country_code)
            logger.info('Temp Min: %s', minTemp)
            logger.info('Temp Max: %s', maxTemp)
            logger.info('rain: %s', rain)
            logger.info('snow: %s', snow)

            # Ottieni dati meteorologici dalla API di OpenWeatherMap
            weather_data = get_weather_data(city, country_code)

            if weather_data:
                # Confronta i dati estratti con quelli dell'API
                api_min_temp = weather_data['main']['temp_min']
                api_max_temp = weather_data['main']['temp_max']
                api_rain = weather_data['rain'][1h] if 'rain' in weather_data and '1h' in weather_data['rain'] else 0
                api_snow = weather_data['snow'][1h] if 'snow' in weather_data and '1h' in weather_data['snow'] else 0

                # Stampa i dati estratti dall'API
                logger.info('Dati estratti dall\'API di OpenWeatherMap:')
                logger.info('Temperatura Minima: %s°C', api_min_temp)
                logger.info('Temperatura Massima: %s°C', api_max_temp)

                if 'rain' in weather_data and '1h' in weather_data['rain']:
                    logger.info('Pioggia nell\'ultima ora: %s mm', weather_data["rain"]["1h"])
                else:
                    logger.info('Pioggia nell\'ultima ora: 0 mm')

                if 'snow' in weather_data and '1h' in weather_data['snow']:
                    logger.info('Neve nell\'ultima ora: %s mm', weather_data["snow"]["1h"])
                else:
                    logger.info('Neve nell\'ultima ora: 0 mm')

                # Esegui confronti e stampa messaggi di avviso se necessario
                if minTemp != api_min_temp:
                    logger.warning('Avviso: Differenza nella temperatura minima. Messaggio: %s, API: %s', minTemp, api_min_temp)

                if maxTemp != api_max_temp:
                    logger.warning('Avviso: Differenza nella temperatura massima. Messaggio: %s, API: %s', maxTemp, api_max_temp)

                if rain != api_rain:
                    logger.warning('Avviso: Differenza nella pioggia. Messaggio: %s, API: %s, rain, api_rain')

                if snow != api_snow:
                    logger.warning('Avviso: Differenza nella neve. Messaggio: %s, API: %s, snow, api_snow')
```

WeatherDataProcessor

```
if api_min_temp < minTemp:
    alert_message = {'type': 'MinTempAlert', 'message': f'Avviso temperatura minima per {city}: {api_min_temp}°C', 'email': email}
    producer.send(alert_topic, json.dumps(alert_message).encode('utf-8'))

if api_rain > rain:
    alert_message = {'type': 'RainAlert', 'message': f'Avviso pioggia per {city}: {api_rain} mm', 'email': email}
    producer.send(alert_topic, json.dumps(alert_message).encode('utf-8'))

if 'snow' in weather_data and '1h' in weather_data['snow']:
    alert_message = {'type': 'SnowAlert', 'message': f'Avviso neve per {city}: Nevicata di {weather_data["snow"]["1h"]} mm', 'email': email}
    producer.send(alert_topic, json.dumps(alert_message).encode('utf-8'))

except json.JSONDecodeError as e:
    logger.error('Errore nel decodificare il messaggio JSON: %s', e)
```

AlertNotifier

AlertNotifier

Il quarto microservizio è stato pensato anche lui senza l'utilizzo di Flask perchè sfrutta l'utilizzo di kafka ,in quanto come detto prima quest'ultimo svolgerà la funzione di consumer(AlertTopic) ,ovvero dovrà ricevere quelli che sono gli alert generati dal terzo microservizio e sulla base di questi ultimi andare a generare la mail con la quale notificare l'utente.A tal proposito la mail dell'utente che deve essere notificato abbiamo deciso di inserirla nel database relativo al secondo microservizio per poter tenere traccia delle sottoscrizioni dell'utente e di conseguenza,quando il secondo microservizio va a costruire il messaggio da inoltrare al terzo anche la mail viene passata come parametro. In tal modo quando poi il terzo microservizio comunicherà l>alert al quarto,verrà passata anche la mail dell'utente,in modo tale che possa essere notificato correttamente.Per realizzare il meccanismo delle mail si è utilizzata la libreria smtplib.

Metrics_scrapers

Metrics_scraping

Nella seconda fase del progetto viene gestita la qualità del servizio della nostra applicazione. In merito a questo doveva essere stato configurato un server prometheus per fare lo scraping delle metriche. Si è deciso dopo aver configurato il server prometheus e aver creato un apposito file yml chiamato prometheus.yml, si sono scelte le metriche da esporre. Tali metriche sono state esposte da un exporter che nel nostro caso prende il nome di metrics_scraping (hostato sulla porta 9100). Inoltre prometheus viene containerizzato, avviato da dockerfile e hostato sulla porta 9090.

Come metriche si sono scelte le seguenti:

- Utilizzo della CPU a livello di container
- Utilizzo della RAM a livello di container
- Utilizzo della rete a livello di container
- Utilizzo della rete a livello di nodo
- Storage a livello di nodo
- Tempo di query al database a livello applicativo

Per la gestione di tali metriche si è pensato alla creazione di due thread, uno che viene utilizzato per la gestione delle metriche a livello di nodo e container e che ogni 15 secondi va ad eseguire lo scraping delle metriche. Il secondo thread ha lo stesso funzionamento, ma viene utilizzato per la gestione delle metriche a livello applicativo.

Metrics_scrapers

La scelta delle metriche è stata pensata ad hoc. Infatti la metrica sull'utilizzo della cpu a livello di container è utile per svariati motivi, di seguito eccone alcuni:

- Performance dell'applicazione: Monitorare l'utilizzo della CPU consente di valutare le prestazioni complessive dell'applicazione all'interno del container. Se l'utilizzo della CPU è troppo elevato, potrebbe verificarsi una degradazione delle prestazioni, e potrebbe essere necessario apportare modifiche all'applicazione o alle risorse assegnate al container.
- Scalabilità: Conoscere l'utilizzo della CPU aiuta a pianificare e gestire la scalabilità delle applicazioni containerizzate.
- Efficienza delle risorse: Misurare l'utilizzo della CPU aiuta a garantire un'allocazione efficiente delle risorse. Se un container utilizza più risorse di quelle necessarie, si potrebbe risparmiare risorse redistribuendo o ridimensionando il container in base alle sue reali esigenze.
- Pianificazione delle risorse: Con una metrica accurata sull'utilizzo della CPU, è possibile pianificare in modo più efficace l'allocazione delle risorse nel cluster di container. Ciò consente di evitare sovraffollamenti e garantisce che ogni container abbia accesso alle risorse di cui ha bisogno.
- Rilevamento di anomalie e problemi: Ad esempio quando vi sono picchi improvvisi di utilizzo della cpu dovuti a carichi elevati.

Metrics_scraping

Stesso discorso vale anche per le altre metriche. Per l'utilizzo della Ram a livello di container alcune delle motivazioni sono le seguenti:

- Prestazioni dell'applicazione: L'utilizzo eccessivo della RAM può causare rallentamenti delle prestazioni dell'applicazione
- Ottimizzazione delle risorse: Allo stesso modo in cui si ottimizza l'utilizzo della CPU, monitorare l'utilizzo della RAM aiuta a garantire un'allocazione efficiente delle risorse. Se un container utilizza più memoria di quella necessaria, è possibile ridimensionare o riconfigurare le risorse per evitare sprechi e garantire l'efficienza complessiva del sistema.
- Scalabilità: Se l'applicazione richiede più memoria per gestire un carico di lavoro più elevato, è importante essere in grado di rispondere in modo adeguato, distribuendo più risorse o ridimensionando i container.
- Evitare esaurimenti di memoria: L'esaurimento di memoria può portare a comportamenti imprevisti e a errori nell'esecuzione delle applicazioni. Monitorare costantemente l'utilizzo della RAM consente di identificare e prevenire situazioni in cui i container possono esaurire la memoria disponibile.
- Rilevamento di perdite di memoria

Metrics_scraping

Per quanto riguarda le metriche sull'utilizzo della rete a livello di nodo e a livello di container i vantaggi sono i seguenti:

A LIVELLO DI NODO

- Identificazione dei Colli di Bottiglia: Monitorare le metriche di rete consente di individuare i punti di congestione o i colli di bottiglia nella rete del nodo. Questa informazione è essenziale per ottimizzare le prestazioni del sistema e garantire una distribuzione uniforme del traffico.
- Rilevamento di Anomalie: Le metriche di rete possono aiutare a identificare anomalie nel traffico di rete. Un improvviso aumento o diminuzione del traffico può indicare problemi di configurazione, attacchi informatici o altre situazioni anomale che richiedono attenzione immediata.
- Gestione del Traffico: Monitorare le metriche di rete consente di gestire meglio il traffico all'interno del cluster di container. Questo è particolarmente importante in ambienti dinamici in cui i container vengono distribuiti o ridimensionati automaticamente. Una gestione efficiente del traffico aiuta a evitare sovraccarichi e migliorare le prestazioni complessive.
- Ottimizzazione delle Risorse
- Sicurezza della Rete: Il monitoraggio delle metriche di rete aiuta a rilevare attività sospette o comportamenti non autorizzati. –
- Pianificazione e Scalabilità: Le metriche di rete forniscono informazioni utili per la pianificazione delle risorse e la scalabilità del sistema. Comprendere come il traffico di rete varia nel tempo consente di pianificare in modo efficiente l'espansione o la contrazione delle risorse in base alle esigenze del carico di lavoro.

Metrics_scrapers

A LIVELLO DI CONTAINER

- Isolamento delle Prestazioni: Monitorare l'utilizzo di rete a livello di container consente di isolare e identificare le prestazioni della rete specifiche di ciascun container. Questo è particolarmente utile in ambienti in cui diversi container condividono la stessa macchina fisica.
- Ottimizzazione delle Risorse: Con le metriche di utilizzo di rete, è possibile ottimizzare l'allocazione delle risorse. Se un container utilizza più larghezza di banda di quanto necessario, è possibile ridimensionare o configurare le risorse di rete per garantire un'allocazione più efficiente.
- Scalabilità Dinamica: Nel contesto di orchestrazione dei container, come Kubernetes, il monitoraggio delle metriche di rete è cruciale per la scalabilità dinamica. Le piattaforme di orchestrazione possono utilizzare queste informazioni per distribuire o ridimensionare i container in base alle esigenze del traffico di rete.
- Bilanciamento del Carico: Le metriche di utilizzo di rete possono essere utilizzate per implementare strategie di bilanciamento del carico tra i container. Questo è particolarmente importante in scenari in cui alcuni container potrebbero ricevere un carico di rete più elevato rispetto ad altri.
- Analisi delle Prestazioni End-to-End: Le metriche di utilizzo di rete possono essere utilizzate per analizzare le prestazioni end-to-end di un'applicazione containerizzata, consentendo di identificare eventuali ritardi o problemi di comunicazione tra i vari servizi.

Metrics_scraping

Per quanto riguarda la metrica relativa allo storage invece è stata scelta per questi motivi:

- Capacità e Allocazione delle Risorse: Monitorare l'utilizzo dello storage a livello di nodo consente agli amministratori di sistema di valutare la capacità complessiva dello storage disponibile sul nodo. Queste informazioni sono fondamentali per garantire un'allocazione efficiente delle risorse e prevenire situazioni di esaurimento dello spazio.
- Prevenzione di Esaurimenti di Spazio a Livello di Nodo: Il monitoraggio dello spazio disponibile a livello di nodo è essenziale per evitare esaurimenti di spazio che potrebbero influire su tutti i container o servizi in esecuzione sul nodo. Un esaurimento di spazio potrebbe portare a malfunzionamenti dei servizi o addirittura al blocco del nodo.
- Ottimizzazione dello Storage Condiviso: In ambienti condivisi, come cluster di container, il monitoraggio delle metriche sull'utilizzo dello storage a livello di nodo aiuta a garantire che la distribuzione delle risorse di storage tra i vari container o servizi sia equa ed efficiente.
- Sicurezza e Conformità:
- Backup e Ripristino: Il monitoraggio dello spazio di archiviazione disponibile a livello di nodo è fondamentale per la pianificazione dei backup. Assicurarsi che ci sia spazio sufficiente per eseguire regolarmente i backup è essenziale per garantire la sicurezza e la ripristinabilità dei dati.
- Controllo dell'Efficienza delle Operazioni di Storage: Analizzare le metriche sull'utilizzo dello storage a livello di nodo può aiutare a valutare l'efficienza delle operazioni di I/O e identificare possibili ottimizzazioni o miglioramenti delle prestazioni.

Metrics_scaper

Infine l'ultima metrica che abbiamo scelto è stata quella relativa al tempo di risposta delle query al database. I perché della scelta sono i seguenti:

- Ottimizzazione delle Prestazioni: Misurare il tempo di esecuzione delle query consente di identificare query lente o inefficienti. Questa informazione è fondamentale per ottimizzare le prestazioni del database, migliorando le query o apportando modifiche al modello dati.
- Identificazione di Colli di Bottiglia: Il tempo di query elevato può indicare la presenza di colli di bottiglia nel sistema. Monitorare queste metriche aiuta a individuare i punti critici in cui le prestazioni possono essere migliorate per evitare rallentamenti o tempi di risposta prolungati.
- Rilevamento di Problemi di Scalabilità: Il monitoraggio del tempo di query è essenziale quando si scala un'applicazione. Aumenti significativi nel tempo di query possono indicare problemi di scalabilità, aiutando gli sviluppatori a identificare quando è necessario ottimizzare il sistema per gestire carichi di lavoro più elevati.
- Miglioramento dell'User Experience: Un tempo di query rapido contribuisce a garantire una risposta veloce alle richieste degli utenti. Monitorare queste metriche è cruciale per offrire un'esperienza utente positiva, evitando ritardi e garantendo tempi di risposta ottimali.
- Identificazione di Query Bloccanti: Il monitoraggio del tempo di query può aiutare a individuare query che bloccano o rallentano altre operazioni nel sistema. Risolvere i problemi di query bloccanti è fondamentale per mantenere la fluidità del sistema e la continuità operativa.
- Analisi delle Tendenze nel Tempo: Analizzando il tempo di query nel tempo, è possibile identificare tendenze o cambiamenti nelle prestazioni del database. Questo è utile per anticipare e risolvere potenziali problemi prima che influiscano negativamente sulle

Sla_manager

Sla_manager

L'ultimo servizio che abbiamo implementato è quello del SLA manager. Questo servizio consiste nell'andare a definire un SLA (Service Level Agreement) dell'applicazione, come composizione di un set di metriche collezionate precedentemente da prometheus. Per realizzare questo servizio abbiamo deciso di andare ad utilizzare flask e di hostare tale servizio sulla porta 3003. Come i microservizi precedenti e lo scraper anche questo servizio viene avviato da dockerfile e gestito con l'ausilio di Docker. Il funzionamento di questo servizio consiste nel esporre un endpoint per andare ad aggiungere una metrica ad SLA. Per ogni metrica abbiamo fornito il valore attuale, il valore desiderato, lo stato della metrica e il numero di violazioni in un tempo predeterminato. Queste informazioni poi saranno salvate all'interno di un database chiamato sla_managerDB. Inoltre tale servizio espone un'interfaccia REST che permettere di visualizzare le informazioni. A tal proposito sono state poi definite le seguenti route:

/sla/configure: questa route implementata con il metodo get serve a restituire semplicemente la pagina dove implementiamo un form per inviare il range di minimo e massimo delle metriche selezionate tramite un checkbox dall'utente, al nostro database.

/sla: questa route utilizza il metodo post e serve a salvare i dati inviati dal form all'interno del database(sla_managerDB) all'interno della tabella sla.

/sla/status: questa route serve a verificare se la metrica è contenuta all'interno del database fa un richiamo per recuperare il valore attuale

Sla_manager

della metrica da prometheus, effettua dei controlli con il range inserito nel form e salva l'eventuale violazione della metrica all'interno del database nella tabella violation.(è possibile richiamare tale ruote all'interno della pagina configura_sla.html tramite un button chiamato aggiorna_sla).

/sla/violations_for_hours: questa route invece serve a verificare il numero di violazioni di una determinata metrica del database nell' intervallo di tempo pari a 1h ,3h ,6h.