UNIVERSITÀ
degli STUDI
di CATANIA

Corso di laurea Magistrale in Ingegneria Informatica

DISTRIBUTED SYSTEMS AND BIG DATA

Progetto prova in itinere

WeatherNotif

Aurora Tallarita -Giulio Samperi

Sommario

Introduzione

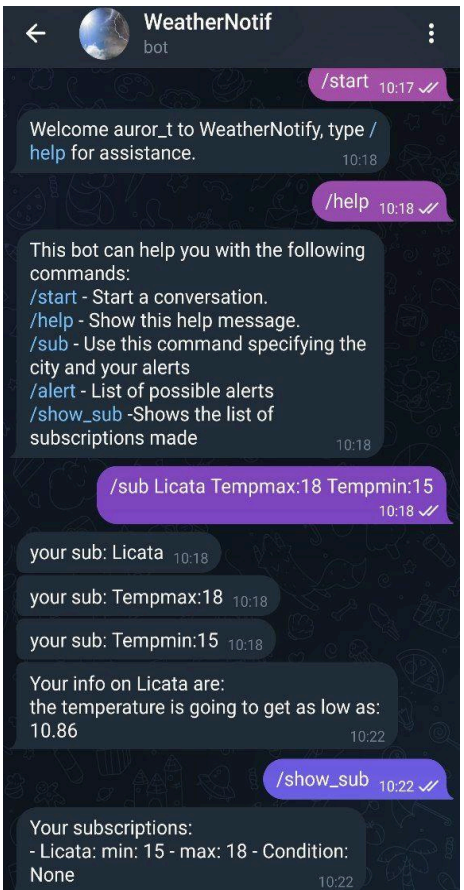
Si è scelto di implementare un applicazione distribuita composta da diversi microservizi, eseguiti su docker containers, seguendo la traccia del tema n.1: weather event notifier.

L'applicazione permette agli utenti, tramite botTelegram WeatherNotif, di ricevere notifiche sul meteo nelle città di interesse. L'utente può sottoscriversi richiedendo le notifiche per una città(ogni sottoscrizione riguarda una sola città), fornendo per ognuno una serie di condizioni da verificare ad esempio: Catania Rain Tempmax:30 Tempmin:20, qualora si verificasse anche solo una delle condizioni indicate nella sottoscrizione, il sistema restituirà una notifica all'utente. Il sistema recupererà le informazioni meteorologiche tramite l'API fornita OpenWeatherMap, filtrando ed elaborando i dati (eseguendo quindi un data scraping) in base alla richiesta dell'utente.

Descrizione dell'applicazione

Per lo sviluppo software come linguaggio è stato usato Python.

Nel microservizio TelegramNotif, si è deciso di utilizzare una classe RedisLink per sfruttare un db redis che crea una correlazione tra username, che verrà salvato assieme ad altri dati dell'utente in un db mysql, e il suo chat id che permette l'invio delle notifiche meteo tramite bot telegram in modo da disaccoppiarlo totalmente da altri microservizi.



Per reallizzare ciò si utilizza la libreria python-telegram-bot, usando il modulo telegram.ext per la gestione dei comandi tramite il bot, nel momento iniziale in cui l'utente avvia il bot ed esegue il comando /start verrà chiamato il metodo link_user dell'istanza di RedisLink che andrà a creare una corrispondenza username/chat-id dell'utente. Inoltre, sono implementati **tutti i comandi che** l'utente avrà a disposizione per poter interagire con il sistema sfruttando il botTelegram.

Comando /sub

Con il comando /sub si è implementata la funzione di gestione sottoscrizione alle informazioni meteorologiche per la città

desiderata dall'utente, esempio /sub Catania Rain Tempmax:30 Tempmin:20
Prima di tutto si è deciso di impostare un controllo sulla città inserita facendo una banale query a Openweather in modo da verificare se l'API abbia informazioni disponibili per quella città, dopodiché si passa elaborare le relative informazioni alla sottoscrizione richiesta e vengono aggiunti i dati dell'utente e le informazioni di sottoscrizione a specifici endpoint attraverso richieste POST.

```
def create_app():
    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+mysqlconnector://wnotif:wnotif@'+os.getenv('MYSQLHOST','localhost:3306')+'/wdb'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    exchange = KExchange('BakedData')
    models.db.init_app(app)
    with app.app_context():
        models.db.create_all()
        try:
            thread = threading.Thread(target=exchange.ConsumeandProduce, args=(app,))
            thread.start()
        except Exception as e:
            logging.error(f"Error producing weather message: {e}")
    @app.route('/add_user', methods=['POST'])
    def adduser():
```

Nel microservizio Wnotif, è stata implementata un'applicazione Flask che offre un servizio di backend per la gestione degli endpoint usati per manipolare i dati riguardo utenti e sottoscrizioni, nello specifico sono stati implementati i seguenti endpoint:
Add_user: riceve i dati dell'utente tramite una richiesta POST, verifica se l'utente esiste già nel database altrimenti aggiunge un nuovo utente
Show_users: genera una risposta contenente le informazioni su tutti gli utenti, non usato nel concreto, usato solo in debug/testing
Find_user: riceve nome ed username dell'utente tramite richiesta POST, esegue la query e ne restituisce le informazioni sull'utente se esiste nel database.
Usato per verificare la corrispondenza username/chat-id
Add_subb: riceve i dati della sottoscrizione tramite una richiesta POST e aggiunge una nuova sottoscrizione al database.
Show_subs: riceve l'username dell'utente tramite una richiesta POST, restituisce le informazioni sulle sottoscrizioni dell'utente in formato json.
All_sub: restituisce tutte le sottoscrizioni nel database in formato json, utilizzata dal microservizio ApiHandler per richiedere i dati da Openweather in base a quali città sono presenti nelle sottoscrizioni.

```python
def process_subs(redis):
    'http://wnotif:5000/all_sub'
    sub_response = requests.get('http://'+os.getenv('APIHOST','localhost')+':5000/all_sub')
    if sub_response.status_code==200:
        record_value = []
        try:
            print(sub_response.json())
            subs_list = sub_response.json()
        except json.decoder.JSONDecodeError as e:
            print(f"Error decoding JSON response: {e}")
            return None
        for key, sub in subs_list.items():
```

Nel microservizio ApiHandler, è stata implementata la funzione per processare le informazioni ottenute dalle sottoscrizioni ottenute dall'endpoint /all_sub, prima di fare una query ad Openweather viene usato un database Redis come cache per evitare di fare query equivalenti per le stesse città presenti in più sottoscrizioni. Per ciascuna sottoscrizione, vengono valutate le diverse condizioni meteorologiche in base alle informazioni ottenute dai dati meteorologici di Openweather, se le condizioni sono soddisfatte, viene costruito un messaggio contenente le informazioni richieste solo delle condizioni che sono state soddisfatte (es. se la temperatura va sotto la temp minima specificata dall'utente ma quella massima viene ancora rispettata, l'utente verrà notificato solo della prima).

Kubernetes

I microservizi sono stati distribuiti su Kubernetes per garantire una gestione scalabile e resiliente dell'infrastruttura. L'utilizzo di Kubernetes ha consentito di orchestrare efficacemente i container e di gestire le risorse in modo dinamico su kind, ottimizzando le prestazioni e facilitando la scalabilità orizzontale.

```
aurora@StrixG531:~/IdeaProjects/WeatherNotif/K8sKind$ sudo kubectl logs usermanager-58cbcc8b49-jt44h
 * Serving Flask app 'WNotif_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.244.1.18:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 208-922-944
10.244.1.20 - - [29/Jan/2024 15:14:42] "GET /all_sub HTTP/1.1" 200 -
10.244.1.20 - - [29/Jan/2024 15:16:22] "GET /all_sub HTTP/1.1" 200 -
10.244.1.20 - - [29/Jan/2024 15:18:02] "GET /all_sub HTTP/1.1" 200 -
10.244.1.20 - - [29/Jan/2024 15:19:42] "GET /all_sub HTTP/1.1" 200 -
10.244.1.19 - - [29/Jan/2024 15:19:52] "POST /add_user HTTP/1.1" 200 -
10.244.1.19 - - [29/Jan/2024 15:19:52] "POST /find_user HTTP/1.1" 200 -
10.244.1.19 - - [29/Jan/2024 15:19:52] "POST /add_subb HTTP/1.1" 200 -
aurora@StrixG531:~/IdeaProjects/WeatherNotif/K8sKind$ sudo kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
kafka-56dfbfccbf-bzlhf            2/2     Running   0          20h
meteoretrieval-5cdd9fd867-wrvbv   1/1     Running   0          7m15s
mysql-0                           1/1     Running   0          20h
redis-0                           1/1     Running   0          20h
tgramnotif-d59d9b997-9bp2c        1/1     Running   0          7m26s
usermanager-58cbcc8b49-jt44h      1/1     Running   0          7m43s
zookeeper-6f8f488996-nktlq        1/1     Running   0          20h
```

A seguire la schermata del bot Telegram in funzione con l'utilizzo di Kubernetes

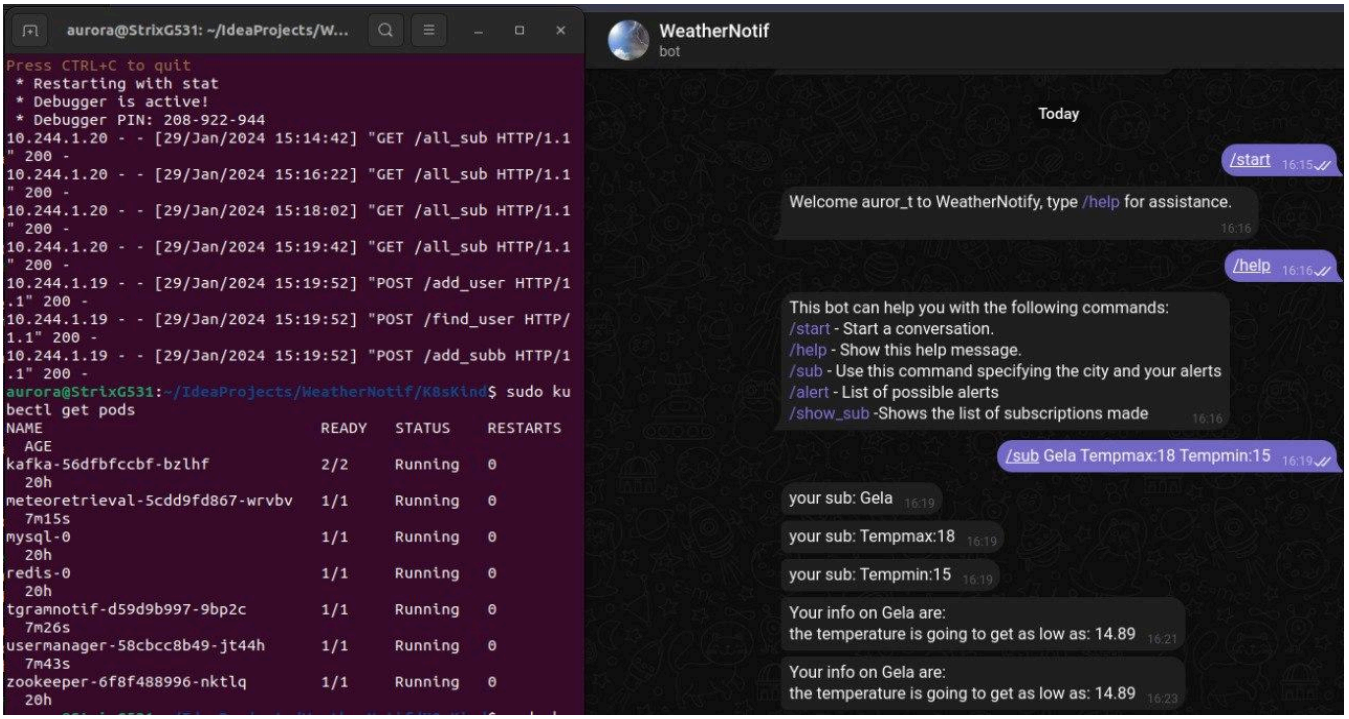
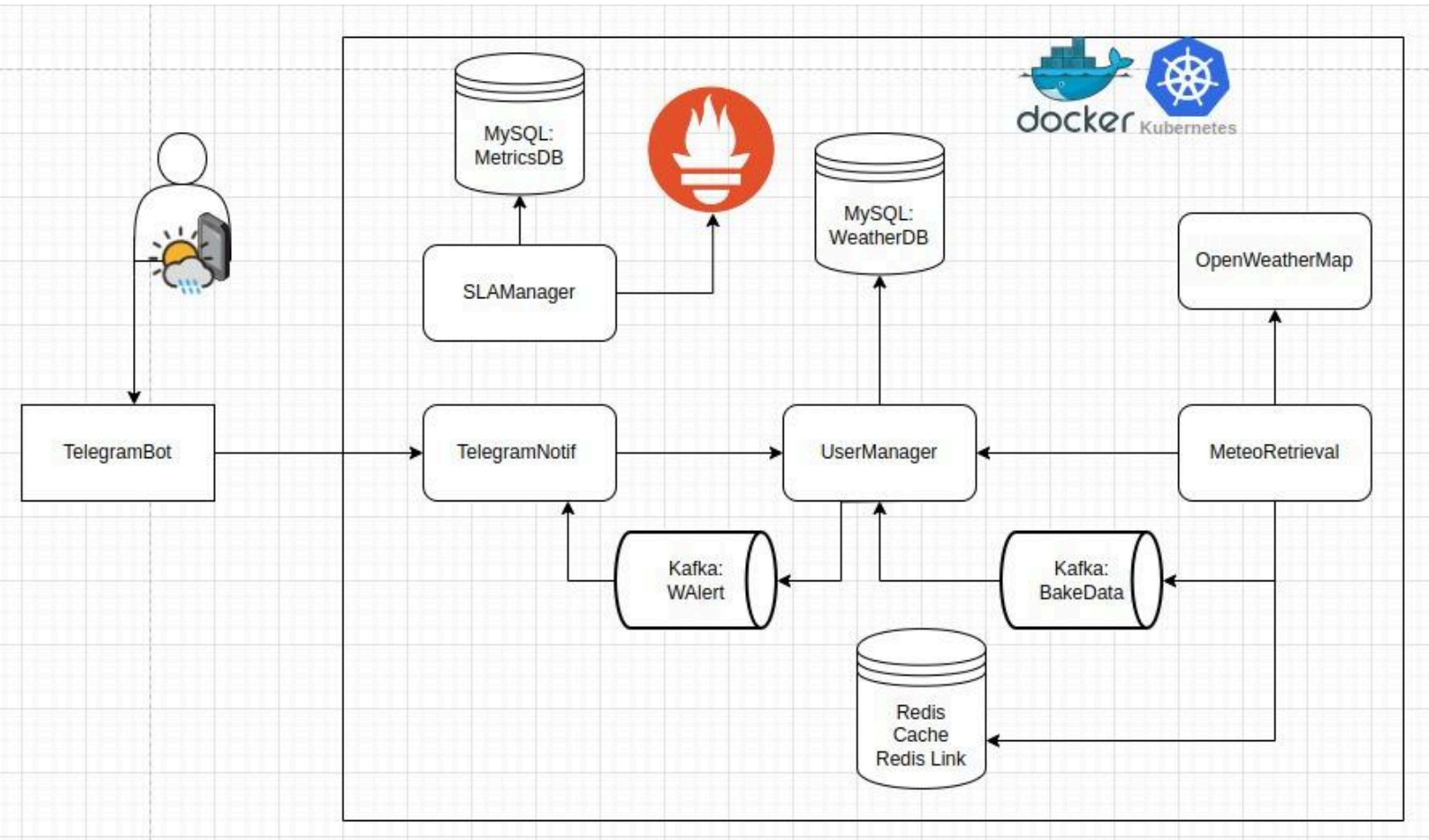
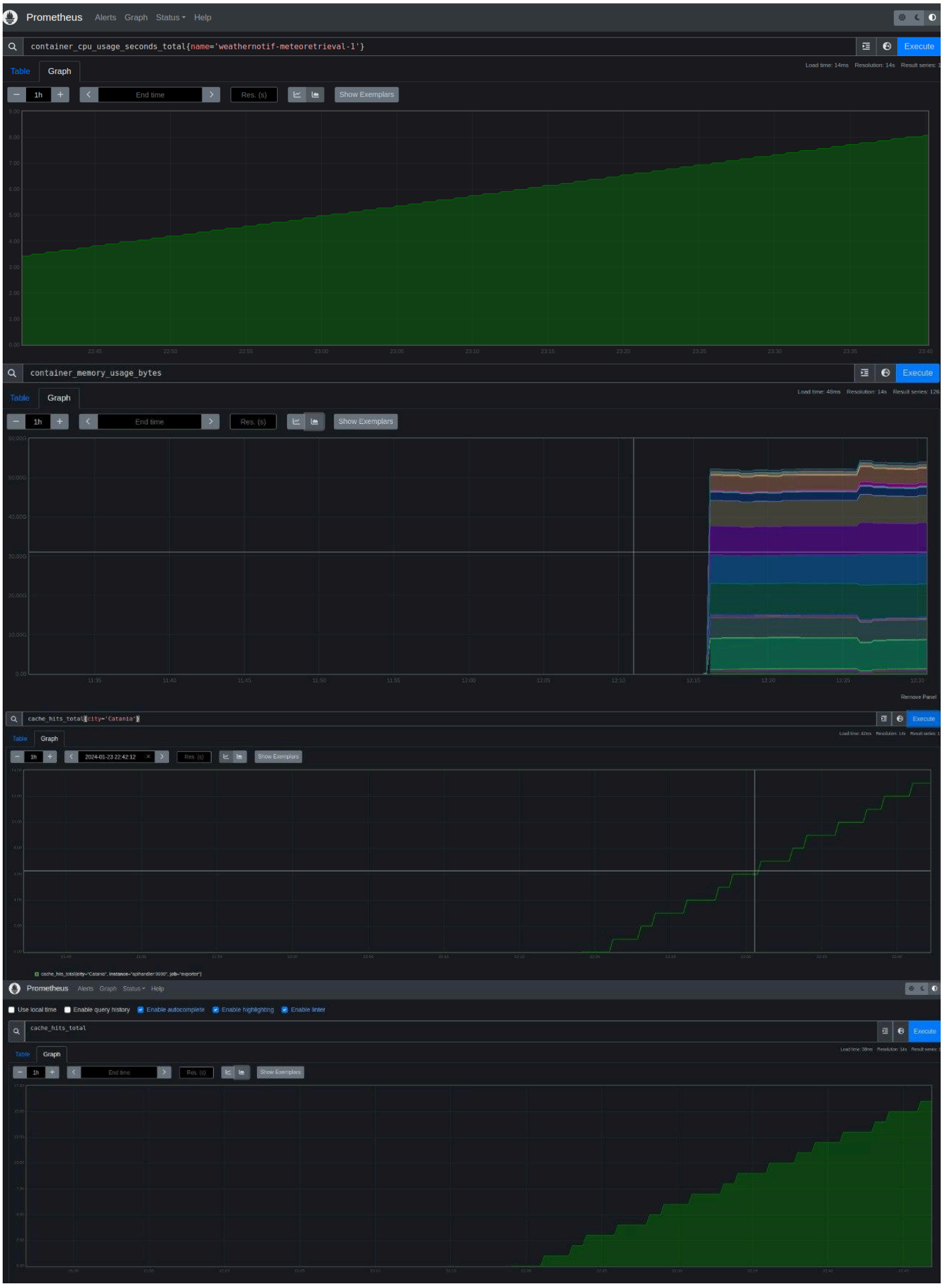


Diagramma Architettura



SLA Manager

Per il monitoraggio delle metriche si è fatto uso di Prometheus, abbiamo implementato un exporter personalizzato ed usato per raccogliere le informazioni dal microservizio MeteoRetrieval, raccogliendo i dati del numero di richieste che avvengono tramite query al servizio Api OpenWeatherMap e il numero di richieste che invece viene trovato in cache.
Per monitorare le risorse e le prestazioni dei container in esecuzione su Kubernetes/Docker utilizziamo cAdvisor (Container Advisor), un componente integrato di Prometheus.
Abbiamo sviluppato la struttura iniziale, che permette di inserire, eliminare e recuperare le sla metrics e verificare, con un thread,se sono state violate e quante volte (non time sensitive).
Non è implementato alcun meccanismo di predizione delle violazioni, lo SLA Manager è stato testato solo in ambiente docker (no Kubernetes)
API Esposte:
POST /sla/metric: creare una nuova sla metric
GET /sla/metric/<int:id>: restituire una sla metric specifica
PUT /sla/metric/<int:id>: aggiornare i valori di una sla metric
DELETE /sla/metric/<int:id>: rimuove una sla metric specifica
GET /sla/metrics: restituisce tutte le sla metric
GET sla/metrics/test: ai fini di testare le query Prometheus


Prometheus
Alerts
Graph
Status
Help
container_cpu_usage_seconds_total{name='weathernotif-meteoretrieval-1'}
Execute
Table
Graph
1h
End time
Res. (s)
Show Exemplars
container_memory_usage_bytes
Execute
Table
Graph
1h
End time
Res. (s)
Show Exemplars
Remove Panel
cache_hits_total{city='Catania'}
Execute
Table
Graph
1h
2024-01-23 22:42:12
Res. (s)
Show Exemplars
Prometheus
Alerts
Graph
Status
Help
Use local time
Enable query history
Enable autocomplete
Enable highlighting
Enable linter
cache_hits_total
Execute
Table
Graph
1h
End time
Res. (s)
Show Exemplars