Relazione Homework 3 DSBD

Lorenzo Varsallona (1000084765) - Paola Pappalardo (1000035439)

Indice

1 Introduzione e obiettivi

Il presente documento illustra le modifiche apportate e le decisioni progettuali prese per la terza e ultima parte dell'homework. Quest'ultima parte del lavoro, si è focalizzata sull'introduzione di un sistema di metriche e logs tramite prometheus, e una completa migrazione verso l'utilizzo di kubernetes tramite Kind. Nelle sezioni successive, si approfondiranno nel dettaglio le funzionalità introdotte e le decisioni progettuali.

2 Architettura del sistema e decomposizione

In questa sezione verranno analizzati gli aspetti implementativi dell'applicativo, con particolare attenzione all'evoluzione architetturale introdotta per soddisfare i requisiti della terza parte dell'homework. L'obiettivo principale è stato il passaggio da un'orchestrazione basata su Docker Compose a un deployment completo su **Kubernetes**, integrando al contempo un sistema di monitoraggio.

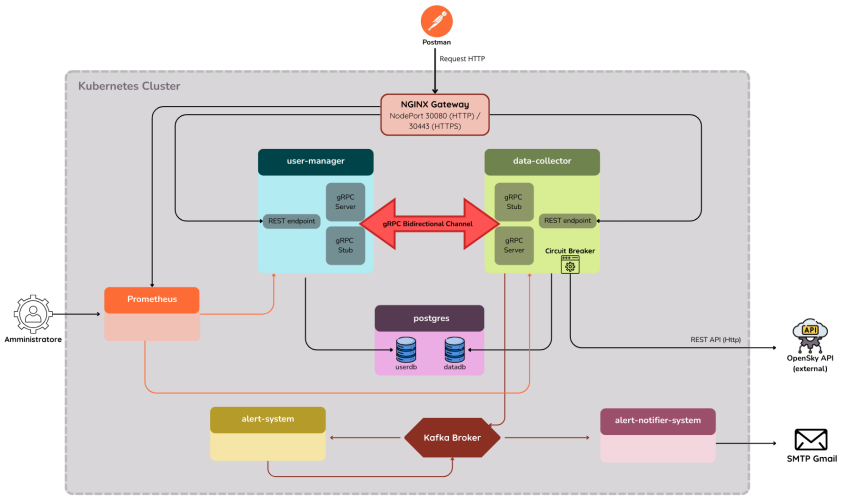

Figura 1: Diagramma architetturale generale del sistema aggiornato.

2.1 Evoluzione dell'Architettura

L'applicativo mantiene la sua struttura a microservizi, consolidata nelle fasi precedenti, ma viene ora eseguito interamente all'interno di un cluster Kubernetes (gestito localmente tramite **Kind**). Come illustrato nella Figura 1, l'architettura è stata aggiornata per riflettere i seguenti cambiamenti strutturali:

- **Cluster Kubernetes**: tutti i componenti del sistema, ad eccezione dei client esterni (Postman, Amministratore) e delle API di terze parti (OpenSky, Gmail), risiedono all'interno del cluster Kubernetes.
- **Ingress Gateway (NGINX)**: il punto di ingresso al cluster è gestito da NGINX, esposto tramite **NodePort** sulle porte **30080 (HTTP)** e **30443 (HTTPS)**. Esso funge da reverse proxy, instradando il traffico verso i servizi `user-manager` e `data-collector`, e ora anche verso **Prometheus**.
- **Monitoraggio con Prometheus**: è stato introdotto un nuovo componente, **Prometheus**, che esegue lo scraping periodico delle metriche esposte dai servizi `user-manager` e `data-collector` tramite le apposite porte configurate.
- **Sistemi di Alerting**: i servizi `alert-system` e `alert-notifier-system` sono configurati come worker puri che consumano messaggi da Kafka, senza esporre interfacce HTTP dirette per il traffico in ingresso.

2.2 Monitoraggio e Observability

Il monitoraggio viene gestito da **Prometheus**, che permette un monitoraggio attivo del cluster. L'approccio adottato è quello del **White Box Monitoring**, che consiste nel monitorare il sistema basandosi sulle metriche esposte direttamente dai suoi componenti interni. A differenza del monitoraggio "Black Box", che osserva il comportamento dall'esterno, il White Box Monitoring permette di analizzare lo stato interno dell'applicazione, come il consumo di risorse, le performance delle query o, come nel nostro caso, contatori specifici della logica di business (es. numero di messaggi puliti o errori di registrazione).

Ogni microservizio, tra quelli principali, è stato quindi strumentato per esporre queste metriche personalizzate, che vengono raccolte periodicamente da Prometheus, offrendo una visione profonda e in tempo reale dello stato di salute dell'intero sistema distribuito.

Infine, l'accesso ai servizi dall'esterno del cluster è gestito tramite un **Ingress Controller NGINX**, configurato per instradare il traffico HTTP e gRPC verso i corretti Service Kubernetes, garantendo un unico punto di ingresso sicuro e gestibile.

2.3 Dettaglio delle Metriche Prometheus

Per monitorare efficacemente lo stato del sistema, sono state definite metriche specifiche per i microservizi principali, suddivise in *Counter* (contatori incrementali) e *Gauge* (valori istantanei).

2.3.1 User-Manager

Il servizio di gestione utenti espone le seguenti metriche:

- **Counter**:
  - `messages_cleaned_total`: conta il numero totale di messaggi scaduti rimossi dal sistema presenti nella cache.
  - `user_registration_errors_total`: traccia il numero totale di errori verificatisi durante la registrazione degli utenti, classificati per tipologia.
- **Gauge**:
  - `cleanup_duration_seconds`: misura il tempo impiegato per l'operazione di pulizia dei messaggi scaduti.

2.3.2 Data-Collector

Il servizio di raccolta dati sui voli espone le seguenti metriche:

- **Counter**:
  - `flights_collected_total`: conta il numero totale di voli raccolti e processati.
  - `collection_errors_total`: traccia il numero totale di errori durante la fase di raccolta dati.
- **Gauge**:
  - `flight_collection_duration_seconds`: misura la durata di ogni ciclo di raccolta dei dati di volo, permettendo di monitorare la latenza delle operazioni esterne.

3 Deployment e Configurazione Kubernetes

Il deployment dell'infrastruttura è stato definito attraverso un insieme di manifest YAML dichiarativi, ciascuno responsabile della configurazione di uno specifico componente del sistema. Di seguito viene analizzata la configurazione adottata per ogni microservizio e risorsa.

3.1 Analisi dei Manifest

3.1.1 Database PostgreSQL (`postgres.yaml`)

Il database relazionale è configurato utilizzando:

- **Deployment**: gestisce l'istanza del database.

- **PersistentVolumeClaim (PVC)**: garantisce la persistenza dei dati anche in caso di riavvio del pod.
- **ConfigMap**: monta lo script `init.sql` per l'inizializzazione automatica dello schema all'avvio.
- **Secret**: le credenziali di accesso sono iniettate in modo sicuro tramite variabili d'ambiente referenziate dal file `secrets.yaml`.
- **Service**: di tipo *ClusterIP*, rende il database raggiungibile dagli altri pod tramite il nome DNS interno `postgres`.

3.1.2 Apache Kafka (`kafka.yaml`)

Il message broker è stato configurato per operare in modalità KRaft (senza Zookeeper):

- **Deployment**: configura il broker Kafka. Un aspetto critico è stato l'impostazione di `enableServiceLinks: false`. Questa direttiva impedisce a Kubernetes di iniettare variabili d'ambiente di servizio (es. `KAFKA_PORT`) che entravano in conflitto con le variabili di configurazione interne dell'immagine Docker, causando errori di avvio.
- **Service**: di tipo *ClusterIP*, espone la porta 9092 per la comunicazione interna tra i microservizi produttori e consumatori.

3.1.3 Microservizi Core (`user-manager.yaml`, `data-collector.yaml`)

I due servizi principali condividono una struttura simile:

- **Deployment**: definisce le repliche e le risorse. Attualmente, il numero di repliche è stato impostato a 1 per semplicità e per limitare il consumo di risorse nell'ambiente di sviluppo locale. Tuttavia, in un ambiente di produzione reale, sarebbe opportuno incrementare questo valore per garantire High Availability e il bilanciamento del carico. Le variabili d'ambiente sensibili (es. stringhe di connessione DB, chiavi API) sono caricate referenziando i **Secret**.
- **Service**: di tipo *ClusterIP*, espongono le porte necessarie per la comunicazione gRPC (50051) e HTTP (5000), oltre alla porta per le metriche Prometheus.

3.1.4 Sistemi di Alerting (`alert-system.yaml`, `alert-notifier-system.yaml`)

Questi componenti agiscono come *worker* puri (consumatori Kafka):

- **Deployment**: gestisce il ciclo di vita dei pod.
- **Assenza di Service**: non dovendo ricevere traffico diretto in ingresso (né HTTP né gRPC), non dispongono di un oggetto Service associato. Comunicano esclusivamente

in uscita verso Kafka e, nel caso del notifier, verso server SMTP esterni (quelli forniti da gmail).

3.1.5 Prometheus (`prometheus.yaml`)

Il sistema di monitoraggio è configurato con:

- **ConfigMap**: contiene il file `prometheus.yml`, che definisce i target di scraping. In questo contesto viene sfruttato il **Service Discovery** di Kubernetes: Prometheus identifica i servizi target (come `user-manager`) tramite i loro nomi DNS interni, rendendo superflua la gestione manuale degli indirizzi IP dinamici dei pod.
- **Deployment**: esegue il server Prometheus.
- **Service**: di tipo *NodePort*, espone la dashboard di Prometheus sulla porta `30090`, permettendo l'accesso diretto dall'host per scopi amministrativi, separando logicamente questo traffico da quello utente gestito da NGINX.

3.1.6 Ingress Gateway (`nginx.yaml`)

NGINX funge da punto di ingresso unico per il cluster:

- **Service NodePort**: a differenza degli altri servizi, NGINX utilizza un Service di tipo **NodePort** esposto sulle porte `30080` (HTTP) e `30443` (HTTPS). Questa scelta è stata preferita al semplice *port-forwarding* per garantire un accesso stabile e persistente all'applicazione dall'host di sviluppo, simulando il comportamento di un LoadBalancer di produzione.
- **ConfigMap**: inietta il file `nginx.conf` che definisce le regole di reverse proxy per instradare il traffico verso i servizi backend e verso la dashboard di Prometheus.
- **Secret**: gestisce i certificati TLS per la terminazione SSL.

3.1.7 Gestione dei secrets: (`secrets.yaml`)

Tutte le informazioni sensibili (password DB, API Key OpenSky, credenziali email,ecc) sono centralizzate in questo manifest. I valori sono codificati in **Base64** e montati come variabili d'ambiente nei container, garantendo che nessuna credenziale appaia in chiaro nelle configurazioni dei Deployment.