Relazione Progetto: SmartParking - Microservizi in Kubernetes

Descrizione dell'Applicazione

L'applicazione **SmartParking** è un sistema di gestione per un parcheggio intelligente, basato su un'architettura a microservizi, progettato per essere facilmente scalabile, modulare e resiliente. L'architettura è pensata per gestire e monitorare il flusso di veicoli e prenotazioni di posti auto in un sistema integrato, che sfrutta tecnologie moderne come Docker, Kubernetes e Kafka per garantire la comunicazione tra i diversi microservizi. Il sistema è composto da tre microservizi principali:

1. **Parking Service**: Gestisce lo stato dei posti auto nel parcheggio, tenendo traccia dei posti disponibili, occupati e liberi.
2. **Booking Service**: Gestisce le prenotazioni dei posti auto. Permette agli utenti di prenotare, modificare e cancellare prenotazioni per i parcheggi.
3. **Notification Service**: Gestisce la notifica degli eventi, ad esempio l'invio di messaggi agli utenti quando la prenotazione è stata effettuata o modificata.

Schema Architetturale

L'architettura dell'applicazione è suddivisa in tre microservizi principali, ognuno dei quali ha le proprie responsabilità e funzionalità. La comunicazione tra i microservizi avviene tramite code Kafka, il che consente una comunicazione asincrona, migliorando l'affidabilità e la scalabilità.

Componenti principali:

- **Kafka**: utilizzato per la comunicazione asincrona tra i microservizi, garantendo che le informazioni vengano processate anche se un servizio è temporaneamente non disponibile.
- **MongoDB**: utilizzato come database per archiviare le informazioni relative alle prenotazioni e ai posti auto.
- **Prometheus e Grafana**: utilizzati per il monitoraggio e la visualizzazione delle metriche del sistema.
- **Ingress**: utilizzato per gestire l'accesso esterno ai microservizi in Kubernetes.

Architettura a Microservizi su Kubernetes

Ogni microservizio è eseguito in un container Docker e orchestrato tramite Kubernetes, il che permette di gestire facilmente la scalabilità, l'alta disponibilità e l'affidabilità del sistema. L'API Gateway è implementato tramite un oggetto **Ingress** in Kubernetes, che instrada le richieste HTTP ai microservizi appropriati.

Flusso dell'architettura:

- I client inviano richieste all'API Gateway (Ingress).
- L'Ingress instrada le richieste ai microservizi **Parking Service, Booking Service**, e **Notification Service**.
- I microservizi comunicano tra di loro tramite Kafka per gestire la logica asincrona.

Lista delle API Implementate

Ogni microservizio offre almeno due API basate su HTTP/REST, come segue:

1. **Booking Service API**:
    - POST /bookings: Crea una nuova prenotazione per un posto auto.

    - GET /bookings/{id}: Recupera le informazioni di una prenotazione esistente.
2. **Parking Service API**:
    - GET /parkings: Ottiene la lista dei posti auto disponibili nel parcheggio.
    - PUT /parkings/{id}: Aggiorna lo stato di un posto auto (disponibile, occupato, etc.).
3. **Notification Service API**:
    - POST /notifications: Invia una notifica, per esempio un messaggio, agli utenti in seguito a un evento (come una prenotazione).
    - GET /notifications/{id}: Recupera lo stato di una notifica inviata.

Tecnologie Utilizzate

- **Docker**: Ogni microservizio è containerizzato utilizzando Docker, garantendo l'isolamento, la portabilità e la coerenza dell'ambiente di esecuzione.
- **Kubernetes**: Utilizzato per l'orchestrazione dei container, la gestione del ciclo di vita dei microservizi e per garantirne la scalabilità e l'affidabilità.
- **Kafka**: Adoperato per la comunicazione asincrona tra i microservizi. Kafka è utilizzato per inviare messaggi e notifiche tra i servizi senza bloccare i processi.
- **MongoDB**: Un database NoSQL utilizzato per memorizzare i dati relativi alle prenotazioni.
- **Prometheus e Grafana**: Per il monitoraggio delle performance e delle metriche di sistema, permettendo di tracciare l'uso delle risorse, la latenza e altre metriche cruciali.

Processo di Build & Deploy

1. **Build dei Contenitori Docker**:
    - I microservizi sono stati containerizzati utilizzando **Dockerfile**. Ogni microservizio ha il proprio Dockerfile che definisce l'ambiente di esecuzione, le dipendenze e la configurazione.
    - Dopo aver creato i Dockerfile, è stato eseguito il comando docker build per costruire le immagini Docker per ogni microservizio.
2. **Docker Compose per Esecuzione Locale**:
    - È stato utilizzato un file **docker-compose.yml** per eseguire tutti i microservizi localmente durante la fase di sviluppo.
    - Il file docker-compose.yml include anche i servizi come **MongoDB**, **Redis** e **Kafka**, necessari per il funzionamento del sistema.
3. **Distribuzione su Kubernetes**:
    - Una volta che il sistema è stato testato con Docker Compose, i microservizi sono stati distribuiti su **Kubernetes**.
    - Per ogni microservizio è stato creato un file **YAML** di **Deployment** che descrive il numero di repliche, le immagini Docker da usare, e le configurazioni necessarie.
    - Inoltre, per ogni microservizio è stato creato un file **Service** per esporre le API internamente al cluster Kubernetes.

    - Infine, è stato configurato un oggetto **Ingress** per gestire il traffico esterno e fare da punto di ingresso per le richieste HTTP.
4. Riguardo l'uso di Prometheus e Grafana sono riuscito a configurarli e avviarli tramite installazione Helm con il port forwarding e raggiungerli (localhost:3000 Grafana e localhost:9090 Prometheus) ma ho avuto problemi soprattutto con Prometheus e non sono riuscito a configurarlo in tempo per poter effettuare il monitoring whitebox e black-box, quindi il progetto risulta mancante di questa ultima parte.

Considerazioni Conclusive

L'architettura basata su microservizi è stata progettata per essere scalabile, resiliente e facilmente mantenibile. Grazie all'uso di Kubernetes, il sistema è facilmente gestibile e può scalare in base alle necessità. Kafka è stato scelto per la comunicazione asincrona tra i microservizi, garantendo che i processi possano essere eseguiti in modo indipendente e senza blocchi. L'uso di Prometheus e Grafana per il monitoraggio del sistema fornirebbe una visibilità dettagliata sulle performance e sullo stato dei microservizi, permettendo di individuare e risolvere rapidamente eventuali problemi.

La rimozione dell'API Gateway su Kubernetes in favore dell'Ingress è stata una scelta per semplificare la gestione del traffico e per sfruttare le funzionalità native di Kubernetes per l'instradamento delle richieste. Questo approccio garantisce una gestione più pulita e scalabile del traffico in ingresso.

L'applicazione è pronta per essere distribuita sia in ambienti locali (con Docker Compose) che in ambienti di produzione (su Kubernetes), con monitoraggio e gestione delle risorse integrati.