Relazione progetto Distributed Systems and Big Data

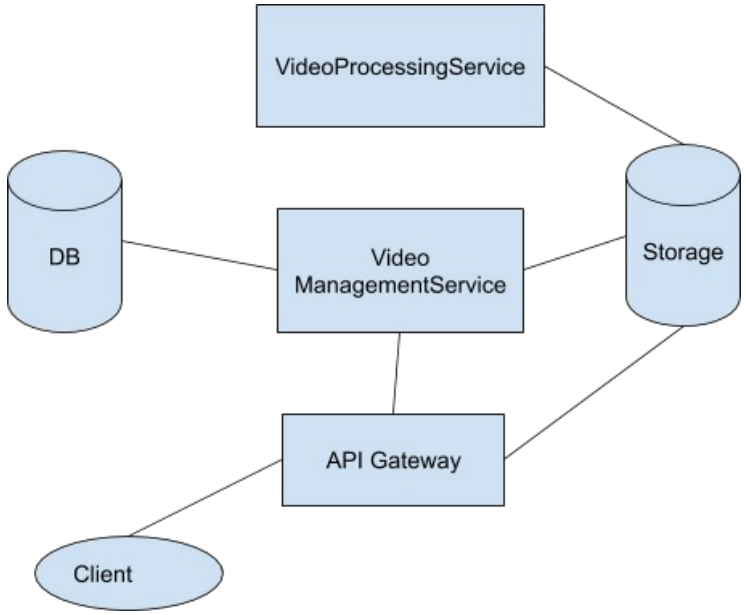
Studenti:

Angelo Marchese (O55000395)

Claudio Curto (O55000428)

Codice Progetto: (a).DB1.GW1.STATS1

Homework 1 – Docker



1.1 Struttura

Nel primo homework vengono realizzati gli ambienti di development e production dell'applicazione effettuando il deployment dei vari componenti su container docker gestiti dalle funzionalità offerte da docker compose. Per ogni componente ci sono due **Dockerfile** (*development* e *production*) e per differenziare i due diversi ambienti vengono utilizzati due differenti file **docker-compose.yml.**

1.2 Api Gateway

L' **api gateway** è costruito a partire dall'immagine *nginx:1-alpine.* Per il development è utilizzato un bind mount per il codice, uno per i log ed un altro per il video storage. In production il codice è salvato direttamente all'interno del container mentre vengono utilizzati due volumi per i log e il video storage. All'interno del file di configurazione (**default.conf**) vengono specificate le diverse route, i campi ed il formato dei log associati alle richieste, i limiti sul tempo di risposta e quelli sulla massima dimensione del payload in input.

1.3 Video Management Service

Il **video management service** è un'applicazione Spring costruita a partire dall'immagine *java:8.* In production si utilizza un multi-stage building aggiungendo pure l'immagine *maven:3-jdk-8* per effettuare la build del file .jar. Per il **video storage** si utilizzano un bind mount e un volume rispettivamente in development e production. In questo caso anche per il development non viene utilizzato un bind mount per il codice. L'implementazione del video management service si basa sul *pattern MVC*. I controller offrono un'interfaccia REST per l'accesso alla business logic realizzata attraverso dei servizi. L'accesso al database (**mariadb**) avviene per mezzo di repository interfaces e i dati vengono modellati dalle diverse classi entity. E' presente inoltre un servizio che permette al video management service di invocare mediante chiamata REST l'operazione di processamento dei video del **video processing service.** Infine la gestione degli errori avviene attraverso l'utilizzo di eccezioni che vengono gestite da un e*xception handler* il quale ritorna degli specifici codici di risposta in base al tipo di errore.

1.4 Video Processing Service

Il **video processing service** è un'applicazione Flask costrutita a partire dall'immagine *python:3* che implementa un metodo controller che riceve richieste di processamento dei video. Tale metodo accetta un *videoId* che utilizza per costruire il path del video mp4 da processare. All'arrivo di una richiesta di processamento viene creato un subprocess che lancia il comando *ffmpeg* per processare il video mp4. Per accedere al video storage questo componente effettua il mount dello stesso volume (bind mount in development) utilizzato dal **video management service**. Come nel caso dell'api gateway in development viene effettuato un bind mount per il codice applicativo.

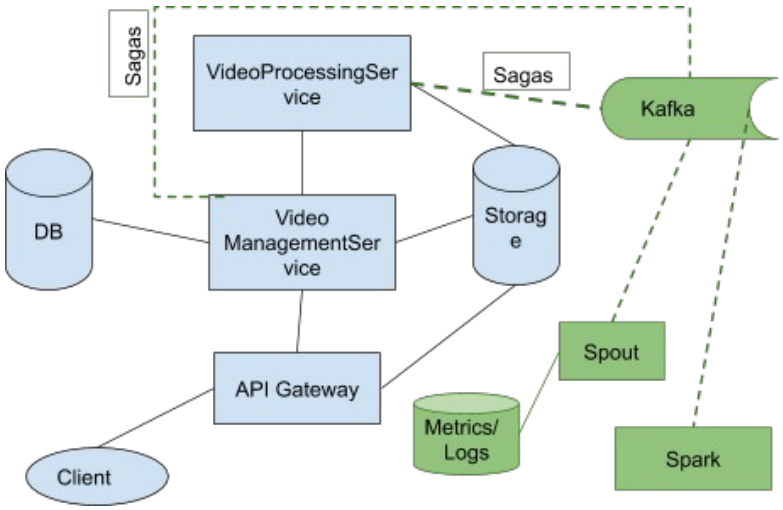
1.5 Metrics Exporter

Il **metrics exporter** è anch'esso un'applicazione Flask il cui scopo è quello di fare un operazione di *tail* sui file di log delle richieste e degli errori. Questo componente accede allo stesso volume su cui l'api gateway salva i log. L'operazione di tail sui log viene effettuata da thread in ascolto sullo stream di input dei file **access.log** e **error.log**. Inoltre è presente un metodo *controller* che gestisce le chiamate all'endpoint /metrics . Questo endpoint viene utilizzato dal metrics exporter per esporre le statistiche raccolte sui log. Queste statistiche vengono esportate secondo uno specifico formato richiesto dal componente **prometheus**.

1.6 Prometheus

Il componente **prometheus** è costruito a partire dall'immagine *ubuntu.* Sia in development sia in production l'eseguibile viene salvato direttamente nel container. Tra i parametri di configurazione all'interno del file **prometheus.yml** viene fornito l'url (in questo caso l'endpoint */metrics* esposto dal metrics exporter) che con una cadenza temporale (sempre specificata come parametro di configurazione) viene contattato per raccogliere le statistiche sui log ed aggregarle secondo una certa logica.

Homework 2 – Kubernetes



2.1 Struttura

Nel secondo homework viene utilizzato kubernetes per realizzare l'ambiente di production dell'applicazione che viene deployata su un cluster all'interno di una macchina virtuale creata da minikube.

2.1 Differenze nei componenti

Per questo secondo homework sia l'**api-gateway** sia il **video management service** presentano una struttura simile a quella mostrata nell'homework precedente. Alcune differenze sostanziali le troviamo invece nel video management service in cui vengono aggiunti un produttore ed un consumatore *kafka* necessari per il messaging asincrono con il video processing service e vengono aggiunti diversi stati della risorsa *Video* necessari per gestire la saga choreography.

Il v**ideo processing service** in questo secondo homework viene realizzato come applicazione *Spring* ma dal punto di vista del processamento del video non ci sono differenze sostanziali. Anche in questo componente vengono aggiunti un produttore ed un consumatore *kafka.*

Al posto dei componenti **metrics exporter e prometheus** vi sono i componenti spout e spark. Il primo, come succedeva nel metrics exporter, sta in ascolto su uno stream di input del file **access.log** per rilevare la presenza di nuovi log salvati dall'api gateway . I log vengono salvati all'interno di una coda da cui vengono successivamente recuperati per essere inviati al componente spark. Nel componente spark i log vengono recuperati attraverso un consumer *kakfa* al quale viene agganciato un data stream necessario per effettuare la racolta e l'elaborazione dei log. Il componente spark è un'applicazione in Scala costruita attraverso un multi-stage building utilizzando le immagini *maven:3-jdk-8* e *openjdk:8-alpine.*

Vengono aggiunti un componente **kafka** in cui è deployato il message broker e un componente *zookeeper* necessario per la coordinazione delle partizioni del broker.

Le coppie **video management service-video processing service** ed **spout-spark** comunicano tra di loro scambiandosi messaggi rispettivamente nei topic *videos* e *logs*.

2.2 Configurazione di Kubernetes

Per rendere accessibile dall'esterno l'applicazione viene utilizzato un componente **Ingress** il cui unico scopo è reinstradare il traffico all'api gateway.

Per l'**api gateway,** il **video management service** e il **database** sono utilizzati sia un Deployment sia un Service di tipo ClusterIP. Per **l'spout** e il v**ideo processing service** viene utilizzato solo un Deployment. Per **kafka** viene utilizzato un Service di tipo ClusterIP e uno StatefulSet mentre per **zookeeper** viene utilizzato un Service di tipo NodePort ed un Deployment.

Il **database** utilizza un volume per salvare i dati e vengono utilizzati due PersistentVolume di tipo *hostpath* per il video storage e i logs. Il mounting del PersistentVolume avviene attraverso dei PersistentVolumeClaims con modalità di accesso *ReadWriteMany*.

Per ogni componente vengono utilizzati i Secret e le ConfigMap in cui si trovano le variabili d'ambiente.

Il componente **spark** viene lanciato per mezzo del comando *spark-submit* in cui viene specificato come url del master l'indirizzo ip della macchina virtuale creata da minikube.

Openshift

Le differenze sostanziali rispetto alla versione del progetto con kubernetes sono le seguenti:

1. per il database viene utilizzato un PersistentVolumeClaim per la persistenza dei dati
2. per lo storage dei video e i log viene utilizzato esclusivamente un PersistentVolumeClaim
3. al post dell'Ingress vi è il componente Route
4. per kafka e zookeeper viene settato il *securityContext*
5. per i vari componenti vengono utilizzati BuildConfig ed ImageStream per costruire e pushare le immagini direttamente a partire dai file presenti nella repository di GitHub