UNIVERSITÀ DEGLI STUDI DI CATANIA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROGETTO DI
DISTRIBUTED SYSTEMS AND BIG DATA

Studenti:

Ludovica Beritelli 1000023081
Gianluca Garozzo 1000027062
Samuele Gulino 1000012306

Anno Accademico 2021 - 2022

Sommario

Introduzione

Il progetto Seecity verte sulla realizzazione di un'applicazione **Spring** e **Spring Boot** con **Maven**, utilizzando la piattaforma software **Docker**, la piattaforma open-source **Kubernetes** e **Kafka**, una soluzione basata su broker che opera mantenendo flussi di dati come record all'interno di un cluster di server.
Il progetto consiste in un piccolo sistema distribuito per la gestione delle città, delle recensioni e degli eventi, ciascuna implementata in un micro-servizio che si interfaccia con un database **MySql** o NoSql (**MongoDB**).

Java e Python

Il linguaggio di programmazione che scegliamo di usare per implementare il progetto è principalmente Java. Per le analisi finali scegliamo di usare Python.

Spring e Spring Boot

Abbiamo iniziato a creare il progetto partendo da Spring Initializr, inserendo i dettagli e le opzioni e scaricando un progetto in Maven. Scegliamo di usare Spring Boot per accelerare lo sviluppo delle varie applicazioni dei micro-servizi, concentrandoci più sulle funzionalità e meno sull'infrastruttura.

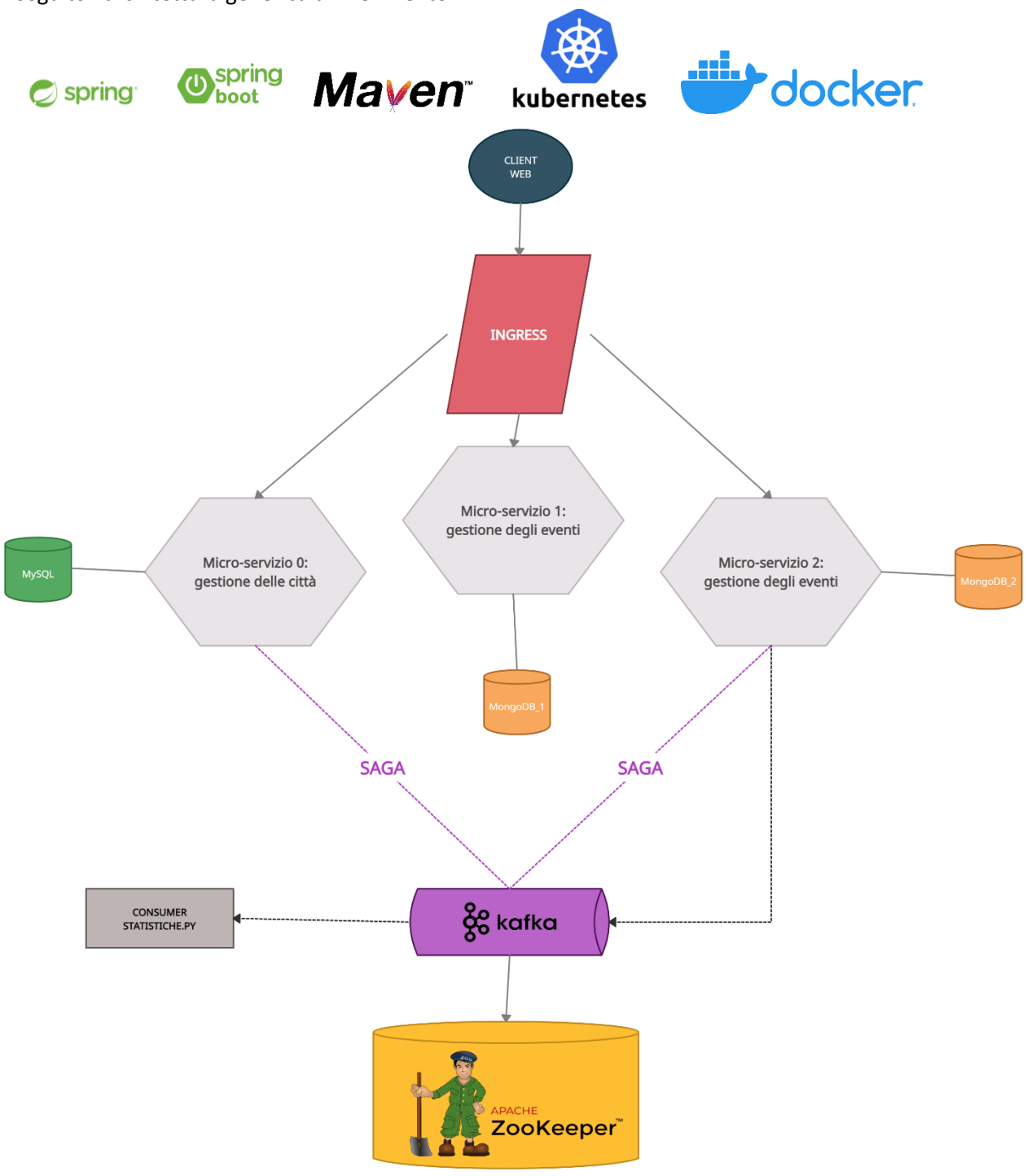
Maven

Maven è lo strumento di compilazione e di gestione dei progetti che scegliamo di usare nel nostro framework creato in Java. Utilizzando Maven, evitiamo di inserire manualmente tutte le dipendenze, perché garantisce che i file JAR (seecity-0.0.1-SNAPSHOT.jar, seecity-MS1-0.0.1-SNAPSHOT.jar, seecity-MS2-0.0.1-SNAPSHOT.jar) e le librerie del progetto vengano scaricati automaticamente. Maven è controllato dal file pom.xml (Project Object Model), che è presente in ogni micro-servizio del nostro sistema distribuito e contiene le dipendenze presenti nel progetto, la directory del file sorgente, le informazioni sul plug-in, l'artifactId che consiste in un file jar distribuito nel repository Maven, il groupId che serve per riconoscersi in modo univoco da tutti i progetti, la versione che specifica quella del jar del progetto e il repository. Questi sono i dati necessari a Maven per realizzare completamente il progetto. Maven legge il file pom per ottenere tutte queste informazioni, compila e impacchetta il codice sorgente in file JAR, si connette al repository Maven Central e li carica in locale.
L'IDE che abbiamo usato per lo sviluppo di questi progetti con Maven è **IntelliJ**.
Nella seguente figura è riportata una parte del file pom.xml del micro-servizio MS0.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.3</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.seecity</groupId>
    <artifactId>seecity</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>seecity</name>
    <description>Progetto DSDB - City Service</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-r2dbc</artifactId>
        </dependency>
```

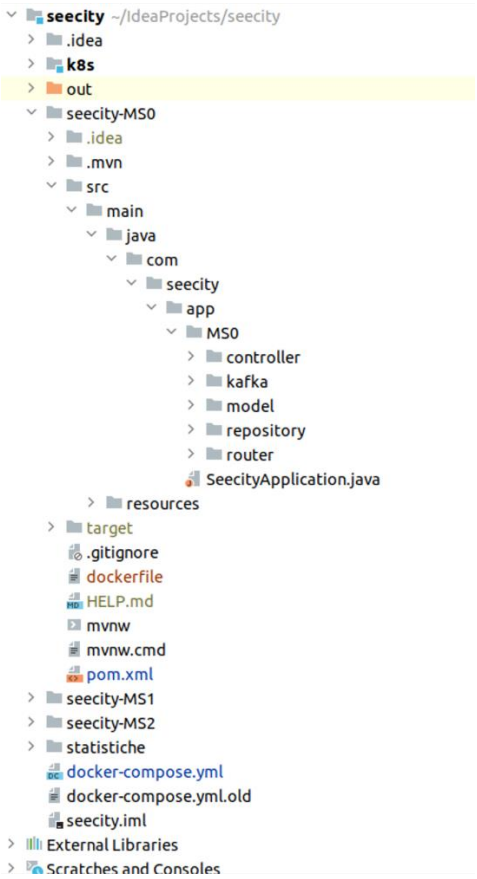
Architettura

Di seguito l'architettura generica di riferimento.



Struttura

Di seguito presentiamo come è strutturato il nostro progetto caricato su GitHub al seguente link: https://github.com/garozzogianluca/ProgettoDSBD_BeritelliGarozzoGulino



Micro-servizi e Rest API

La logica applicativa del sistema è suddivisa in micro-servizi indipendenti di piccole dimensioni, che comunicano fra loro tramite API (operazioni CRUD) basate sul protocollo http.

1. Micro servizio per la gestione delle città (MS0) che comunica attraverso la porta 8080
   API:
   - GET /: risponde con la rappresentazione json delle città
   - GET /{nome}: risponde con la rappresentazione json della città {nome}
   - GET /{nome}/exists: risponde con true se la città esiste
   - POST /: aggiunge una città al DB
2. Micro servizio per la gestione delle recensioni delle città (MS1) che comunica attraverso la porta 8081
   API:
   - POST /feedback/add: aggiunge una recensione al DB
   - GET /: risponde con la rappresentazione json dei feedback
   - GET /feedback/{citta}: risponde con la rappresentazione json dei feedback di citta {citta}
   - DELETE /{id}: elimina la recensione della città con id {id}
3. Micro servizio per la gestione degli eventi locali (MS2) che comunica attraverso la porta 8082
   API:
   - POST /: aggiunge un evento al DB

- PUT /{id}: modifica un evento tramite id {id}
- DELETE /{id}: elimina l'evento con id {id}
- GET /{id}/exists: risponde con true se l'evento esiste
- GET /{id}: risponde con la rappresentazione json dell'evento con l'id {id}
- GET /events/{citta}: risponde con la rappresentazione json dell'evento nella città {city}
- GET /: risponde con la rappresentazione json degli eventi

Docker e Docker Compose

Per soddisfare i nostri requisiti realizziamo dei containers docker al cui interno vengono messe in esecuzione delle applicazioni Spring Boot.
La piattaforma software Docker ha permesso di automatizzare la distribuzione del nostro sistema in modo non troppo complesso: attraverso il dockerfile abbiamo creato semplicemente le immagini Docker che contengono i dati dell'applicazione, utilizzate da Docker per creare i containers. In questo modo i containers contengono tutte le dipendenze dell'applicazione e non richiedono particolari configurazioni: sono autosufficienti e portabili, in quanto sono distribuiti in un formato standard (le immagini) che possono essere letti ed eseguiti da qualunque server Docker. Ciascun micro-servizio e ciascun database sono stati inseriti all'interno di un container.
Con i seguenti comandi vediamo le immagini e i container delle nostre applicazioni.

```
docker images
```

```
seecity_city                latest    1c333c820769   About an hour ago   559MB
seecity_event               latest    2a176329082b   19 hours ago        785MB
seecity_feedback            latest    b0293c0cde97   24 hours ago        651MB
image0                      latest    4f2891cabad6   25 hours ago        558MB
image2                      latest    5e16fe3a248d   41 hours ago        785MB
image1                      latest    1eb3c2844a0e   2 days ago          651MB
seecity-feedback            v1        c302c0f3e584   2 days ago          651MB
provectuslabs/kafka-ui      latest    f1e30cd9d423   5 days ago          283MB
openjdk                     17        3d4c93dc003c   10 days ago         471MB
confluentinc/cp-kafka       latest    5069d65bcc55   2 weeks ago         780MB
confluentinc/cp-zookeeper   latest    3a7ea656f1af   2 weeks ago         780MB
mongo                       latest    ee13a1eacac9   3 weeks ago         696MB
ubuntu                      latest    d13c942271d6   3 weeks ago         72.8MB
openjdk                     latest    5f94f53bbced   5 weeks ago         471MB
gcr.io/k8s-minikube/kicbase v0.0.29   64d09634c60d   5 weeks ago         1.14GB
zookeeper                   latest    36c607e7b14d   5 weeks ago         278MB
maven                       3-jdk-8   b69971c8f574   5 weeks ago         537MB
mysql                       latest    ecac195d15af   3 months ago        516MB
alpine                      latest    14119a10abf4   5 months ago        5.6MB
```

```
docker container ls
```

```
9cc9112468b2   seecity_feedback:latest           "/bin/sh -c 'java -j…"   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp                    seecity_feedback_1
ae0c50abf06f   confluentinc/cp-kafka:latest      "/etc/confluent/dock…"   0.0.0.0:29092->9092/tcp, :::29092->9092/tcp                  seecity_kafka_1
8d0915f3029a   seecity_city:latest               "java -jar city.jar"     0.0.0.0:8080->8080/tcp, :::8080->8080/tcp                    seecity_city_1
528e24d79914   seecity_event:latest              "/bin/sh -c 'java -j…"   0.0.0.0:8082->8082/tcp, :::8082->8082/tcp                    seecity_event_1
6d306be5937e   mongo:latest                      "docker-entrypoint.s…"   0.0.0.0:27017->27017/tcp, :::27017->27017/tcp                seecity_feedback_db_1
3353618831d4   confluentinc/cp-zookeeper:latest  "/etc/confluent/dock…"   2888/tcp, 0.0.0.0:2181->2181/tcp, :::2181->2181/tcp, 3888/tcp seecity_zookeeper_1
d2337020844c   provectuslabs/kafka-ui            "/bin/sh -c 'java $J…"   0.0.0.0:8083->8080/tcp, :::8083->8080/tcp                    kafka-ui
fc91b03d5431   mysql:latest                      "docker-entrypoint.s…"   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 33060/tcp         seecity_city_db_1
374d7103d429   mongo:latest                      "docker-entrypoint.s…"   27017/tcp, 0.0.0.0:27018->27018/tcp, :::27018->27018/tcp     seecity_event_db_1
```

Successivamente, lo strumento Docker Compose ci ha permesso di definire e condividere applicazioni multi-container, creando inizialmente un file yaml per ogni micro-servizio dell'applicazione, per testare eventuali errori del micro-servizio e, successivamente, un unico file yaml "docker-compose.yml" per tutto il progetto, in modo che avviamo l'applicazione digitando da terminale i seguenti comandi:

```
docker-compose build
docker-compose up
```

Kubernetes

Dopo aver sviluppato il nostro progetto con Docker, abbiamo usato Kubernetes, un software di orchestrazione open source che fornisce un'API che consente di controllare come e dove eseguire i containers. Infatti, ci ha permesso di eseguire i containers Docker, raggruppati in pod, cioè delle unità operative di base per Kubernetes.

I vantaggi di k8s sono tanti: gestisce automaticamente l'individuazione di servizi, incorpora il bilanciamento del carico, verifica l'allocazione delle risorse ed esegue il ridimensionamento in base all'utilizzo delle risorse di calcolo. Controlla anche l'integrità delle singole risorse e consente la riparazione automatica delle app mediante il riavvio o la replica automatica dei containers.

Per prima cosa abbiamo definito l'ip di minikube nel file etc/hosts del sistema locale (Ubuntu versione 20.04 LTS), aggiungendo la riga "seecity.dev.loc", con l'IP di k8s 192.168.49.2, per permettere di effettuare la risoluzione dei nomi tramite DNS.

Nel nostro progetto è presente la directory "k8s" che contiene vari file Yaml.

Di seguito i dettagli della nostra applicazione in k8s.

```
NAME                                READY   STATUS    RESTARTS        AGE
pod/citta-5f7467557c-bh4dj          1/1     Running   1 (5m ago)      6m7s
pod/citta-5f7467557c-f2skb          1/1     Running   1 (5m ago)      6m7s
pod/citta-5f7467557c-hdtqp          1/1     Running   1 (4m52s ago)   6m7s
pod/event-c9b597499-hzrb5           1/1     Running   1 (3m47s ago)   6m6s
pod/event-c9b597499-jxxvv           1/1     Running   1 (3m47s ago)   6m6s
pod/event-c9b597499-ml82f           1/1     Running   0               6m6s
pod/feedback-7497bbf9c-k47sb        1/1     Running   0               6m7s
pod/feedback-7497bbf9c-k7xk4        1/1     Running   0               6m6s
pod/feedback-7497bbf9c-t8rpp        1/1     Running   0               6m6s
pod/mongo-event-64d4c4ccfd-4z2fh    1/1     Running   0               6m7s
pod/mongo-feedback-57fcf6ff79-v5kmz 1/1     Running   0               6m7s
pod/mysql-67d9f9c546-w6nkm          1/1     Running   0               6m7s
pod/zookeeper-84457678f7-2bfql      1/1     Running   0               6m6s

NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/citta               ClusterIP   10.104.204.191   <none>        8080/TCP         6m6s
service/event               ClusterIP   10.104.144.41    <none>        8082/TCP         6m6s
service/feedback            ClusterIP   10.96.58.53      <none>        8081/TCP         6m6s
service/kafka               ClusterIP   10.108.234.42    <none>        9092/TCP         6m6s
service/kubernetes          ClusterIP   10.96.0.1        <none>        443/TCP          9m
service/mongo-1             ClusterIP   10.101.78.161    <none>        27017/TCP        6m7s
service/mongo-2             ClusterIP   10.105.241.35    <none>        27018/TCP        6m7s
service/mysql               ClusterIP   10.100.179.60    <none>        3306/TCP         6m7s
service/zookeeper-service   NodePort    10.106.136.2     <none>        2181:31332/TCP   6m6s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/citta               3/3     3            3           6m7s
deployment.apps/event               3/3     3            3           6m7s
deployment.apps/feedback            3/3     3            3           6m7s
deployment.apps/mongo-event         1/1     1            1           6m7s
deployment.apps/mongo-feedback      1/1     1            1           6m7s
deployment.apps/mysql               1/1     1            1           6m7s
deployment.apps/zookeeper           1/1     1            1           6m6s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/citta-5f7467557c               3         3         3       6m7s
replicaset.apps/event-c9b597499                3         3         3       6m6s
replicaset.apps/feedback-7497bbf9c             3         3         3       6m7s
replicaset.apps/mongo-event-64d4c4ccfd         1         1         1       6m7s
replicaset.apps/mongo-feedback-57fcf6ff79      1         1         1       6m7s
replicaset.apps/mysql-67d9f9c546               1         1         1       6m7s
replicaset.apps/zookeeper-84457678f7           1         1         1       6m6s
```

Ogni pod è controllato da 3 replicaSet che a loro volta sono gestite dal deployment.

Tutti i service sono dei ClusterIp, tranne Zookeeper che invece è un nodePort.

Nella nostra applicazione è l'ingress che rappresenta il punto di accesso del cluster ed è stato impostato nel file deployments.yml, in cui sono state settate varie "regole": host "seecity.dev.loc", path e servizi con i quali servire le richieste.

Oltre all'ingress, il file deployments.yml contiene i vari deployment dei micro-servizi che, comunicando con le replicheSet dei pod, hanno il compito di aggiornare i pod. Sono state definite anche alcune regole per il

rollback: è possibile tornare ad una versione precedente del pod in quanto il parametro "RevisionHistoryLimit" è settato a 3, e contiene una vera e propria strategia per la gestione degli aggiornamenti: nel nostro caso RollingUpdate, con maxSurge = 1.
Di seguito riportiamo i comandi da terminale per avviare minikube e far partire la nostra applicazione con Kubernetes:

```
minikube start --vm-driver=docker
eval $(minikube docker-env)
minikube addons enable ingress
docker build -t seecity-citta:v1 ./seecity-MS0
docker build -t seecity-feedback:v1 ./seecity-MS1
docker build -t seecity-event:v1 ./seecity-MS2
kubectl apply - f ./k8s
```

Pattern

DATABASE PER SERVICE

Abbiamo applicato il pattern "database per service", per consentire il basso accoppiamento tra i servizi, caratteristica tipica dei sistemi distribuiti.
Abbiamo deciso di utilizzare un database relazionale per MS0 e 2 database NoSQL per MS1 e MS2. Questo modello consente ai servizi di gestire i dati del dominio in modo indipendente su un archivio dati che meglio si adatta ai tipi di dati e allo schema. Inoltre, consente al servizio di ridimensionare i suoi datastore su richiesta e lo isola dai guasti di altri servizi.

- Il micro-servizio MS0 ha il suo database MySql, che contiene la tabella delle città
- MS1 ha il suo database MongoDB, che contiene la collezione delle recensioni delle città
- MS2 ha il suo database MongoDB, che contiene la collezione degli eventi delle città.

Così facendo, eventuali modifiche a un database non influiscono sugli altri micro-servizi. Non è possibile accedere al database del servizio direttamente da altri micro-servizi.

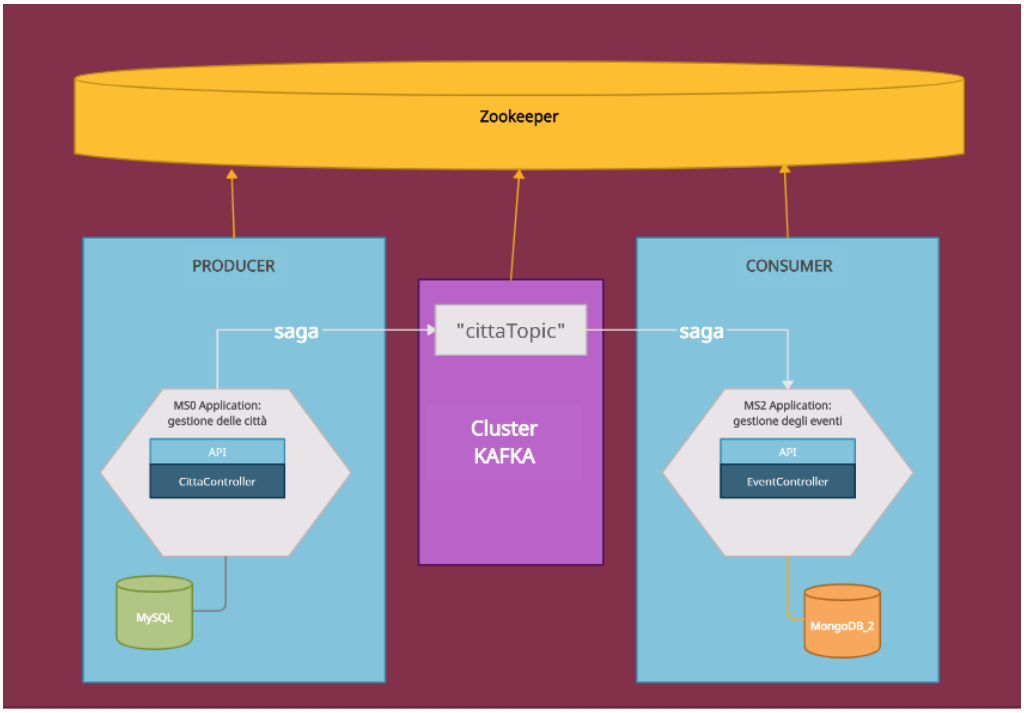
REACTIVE PROGRAMMING

Nei micro-servizi MS0 e MS2 abbiamo utilizzato il pattern reactive programming che ci ha consentito di rendere reattive le applicazioni. È basato sul modello publish/subscribe di stream di dati, con due principali protagonisti: il Publisher per la pubblicazione di dati e il Subscriber per la loro elaborazione in modo asincrono. Quindi, abbiamo usato la libreria Reactor, dichiarata come dipendenza sul file pom.xml, che offre due diversi tipi di dato, Mono e Flux, e mette a disposizione alcuni metodi come just, usato nel progetto.

SAGA

Il modello dell'architettura Saga fornisce la gestione delle transazioni utilizzando una sequenza di transazioni locali, cioè l'unità di lavoro svolta da MS0 e MS2, i partecipanti alla saga. Ogni operazione che fa parte della Saga può essere annullata da una transazione compensativa e il modello Saga garantisce che tutte le operazioni siano state completate correttamente o che le corrispondenti transazioni di compensazione vengano eseguite per annullare il lavoro precedentemente completato. Dunque, abbiamo implementato il pattern SAGA per eseguire il rollback e compensare se l'operazione di inserimento dell'evento (POST) di MS2 non riesce, nel caso in cui la città per la quale si vuole caricare l'evento non esiste nel database mysql di MS0. Abbiamo fatto ciò gestendo lo stato dell'evento, inizialmente settandolo a PENDING al momento della creazione, e poi a CONFIRMED oppure DELETED dopo il controllo della città nel database.
Il controllo avviene tramite Kafka e Zookeeper:

1. MS2 al momento della POST scrive un messaggio con i vari dati dell'evento sul topic "cittaTopic" nel cluster Kafka
2. MS0:
   - legge il messaggio scritto sul topic "cittaTopic"
   - controlla nel database mysql se è presente la stessa città scritta sul topic
   - risponde al messaggio inviato sul topic "cittaTopic" con l'esito della ricerca.
3. Lo stato dell'evento inserito sul database Mongo viene modificato.



CIRCUIT BREAKER

Abbiamo usato il pattern circuit breaker disponibile su K8S a livello di pod lifecycle (readiness e liveness probe) e service, per consentire ai 3 micro-servizi (MS0, MS1, MS2) di continuare a funzionare in caso di errore di un servizio correlato, impedendo che l'errore si verifichi a cascata e concedendo al servizio in errore il tempo di ripristino.
Abbiamo scelto di usare il service ClusterIp.
La readinessProbe indica che qualunque richiesta inviata con dei pod non ready, non riceve traffico; la livenessProbe è un check periodico della salute del pod che inizia dopo il readinessProbe e consiste nel riavvio del pod nel caso in cui fallisce.
Di seguito riportiamo, la parte di codice del service e del deployment di MS1, in cui abbiamo impostato gli attributi livenessProbe e readinessProbe.

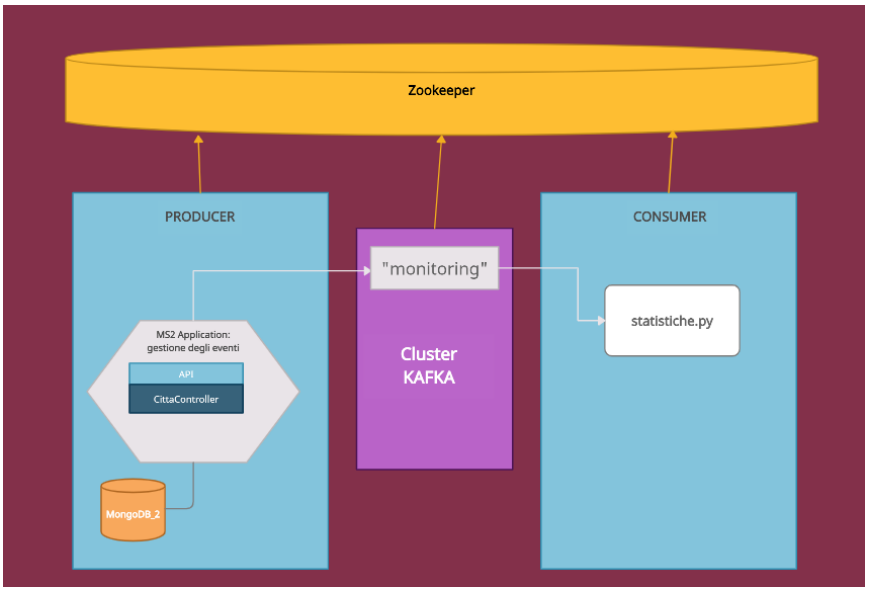
```yaml
apiVersion: v1
kind: Service
metadata:
  name: feedback
  labels:
    svc: feedback
spec:
  ports:
    - port: 8081
      protocol: TCP
      name: http
  selector:
    app: seecity-feedback
```

```yaml
livenessProbe:
  httpGet:
    port: 8081
    path: /ping
  initialDelaySeconds: 120
  periodSeconds: 5
  failureThreshold: 3
readinessProbe:
  httpGet:
    port: 8081
    path: /ping
  initialDelaySeconds: 60
  periodSeconds: 5
  failureThreshold: 5
```

Monitoring e kafka

Scegliamo di usare la whitebox monitoring con Kafka, inviando nel topic "monitoring" per ogni richiesta di MS2:

- il tipo di richiesta
- Ip remoto
- Ip locale
- il tempo di risposta
- lo status code
- la data e l'ora



Analisi

Abbiamo raccolto i dati inviati sul topic "monitoring" ad ogni richiesta di MS2, attraverso uno script in Python3, e abbiamo applicato le seguenti analisi:

1) Il numero di richieste in base al tipo di richiesta
2) Il numero di richieste in base alla data
3) Il tempo di elaborazione in base ad ogni richiesta.