Relazione del Progetto DSBD 2020/2021

Sviluppo di un sistema "e-commerce" distribuito

Distributed Systems and Big Data

C.d.L.M. Ingegneria Informatica

Docente:

Prof.ssa: **Antonella Di Stefano**

Tutor:

Ing. **Andrea Di Maria**

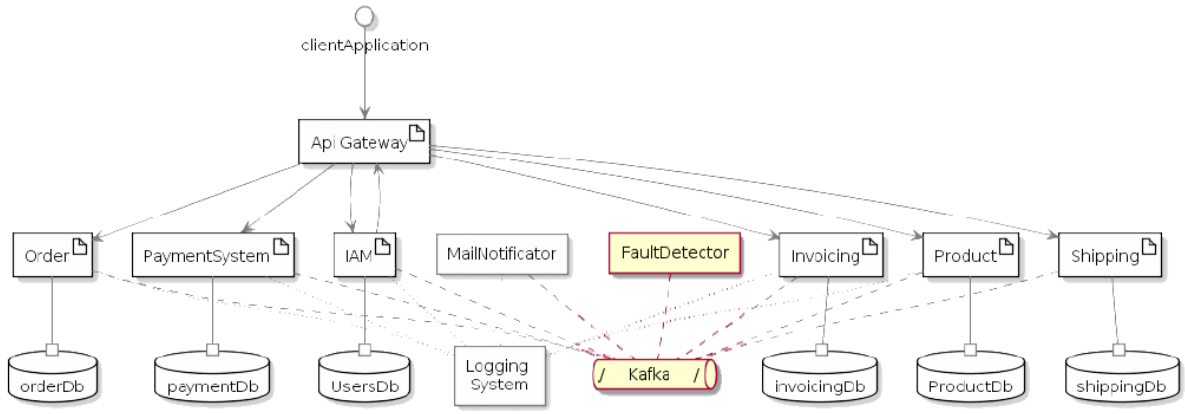
Ing. **Alessandro Di Stefano**

Studente:

Papa Andrea Valentino - Matricola 1000012388

Introduzione

Nel progetto consegnato, è stato implementato il microservizio **Heart-Beat fault detector** (variante 6A – Spring MVC), con sviluppo della strategia di ping denominata nella consegna come **ping-ack mode** (o Ping Endpoint 1).



Il progetto doveva:

- Esporre un endpoint GET /ping che risponde con:

```
{
    "serviceStatus": "up"
}
```

- Esporre un endpoint POST /ping che si occupa di ricevere lo stato dagli altri servizi con richieste di questo tipo:

```
{
    "service": "serviceName",
    "serviceStatus": "up|down",
    "dbStatus": "up|down"
}
```

  Memorizzare lo stato di tutti i servizi in una mappa, e segnalare sul **topic logging** eventuali servizi che riportano uno stato down (servizio o/e database) o servizi che non inviano più richieste sull'endpoint da un periodo fissato (servizi considerati unavailable).

Strategia di ping

Come detto nell'introduzione, per il progetto è stata implementata la strategia **ping-ack mode**.

Semplicemente, come mostrato nella figura seguente, è stato implementato un endpoint, che ricevuta una richiesta GET con path "servizio:8080/ping", risponderà informando dello stato attivo del servizio. Non è stato inserito lo stato del database (dbStatus) perché il microservizio sviluppato in questo progetto, a differenza degli altri, non ha un database ad esso collegato.

```
//GET http://localhost:1234/ping
@GetMapping(path = "/ping")
public @ResponseBody Map<String, String> Hello()
{
    HashMap<String, String> map = new HashMap<>();
    map.put("serviceStatus", "up");
    return map;
}
```

Servizio di heart-beat fault detector

Il cuore del progetto assegnatomi era lo sviluppo del microservizio che si occupava di gestire le richieste degli altri servizi che utilizzavano la hear-beat mode come strategia di ping.

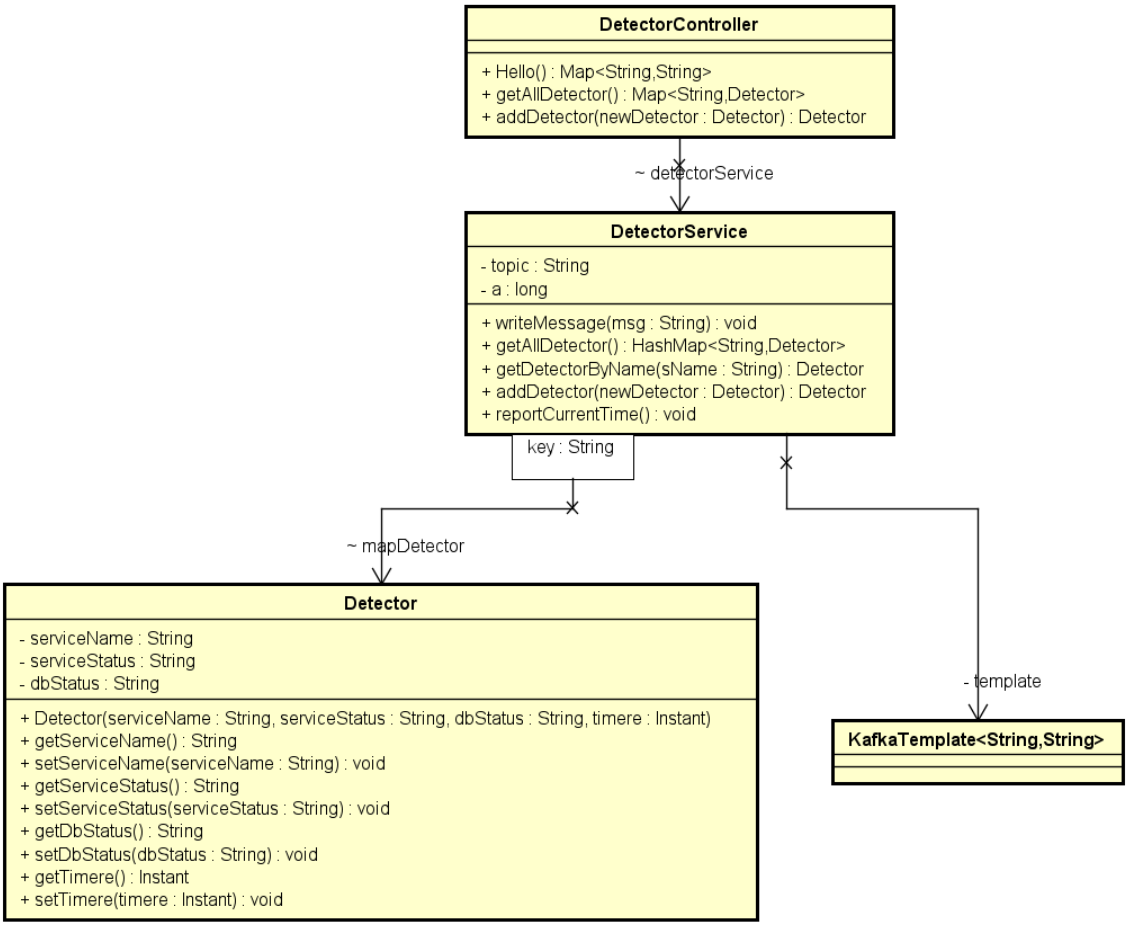
Per far questo, ho deciso di suddividere i "compiti del servizio" in tre:

1. Abbiamo una classe di interfaccia, la stessa utilizzata nel paragrafo precedente, ovvero la classe **DetectorController**, nel progetto si trova all'interno del package Controller.

   ```
   //POST http://localhost:1234/ping
   @PostMapping(path = "/ping")
   public @ResponseBody Detector addDetector(@RequestBody Detector newDetector)
   {
       return detectorService.addDetector(newDetector);
   }
   ```

2. La classe **Detector**, in cui è implementato l'oggetto, con i suoi attributi (serviceName, serviceStatus, dbStatus e timere), offre l'entità che poi verrà memorizzata all'interno della mappa. L'attributo timere viene utilizzato per conservare l'istante di tempo dell'ultimo aggiornamento di stato ricevuto da un servizio.
   La classe si trova all'interno del package Data del mio progetto.

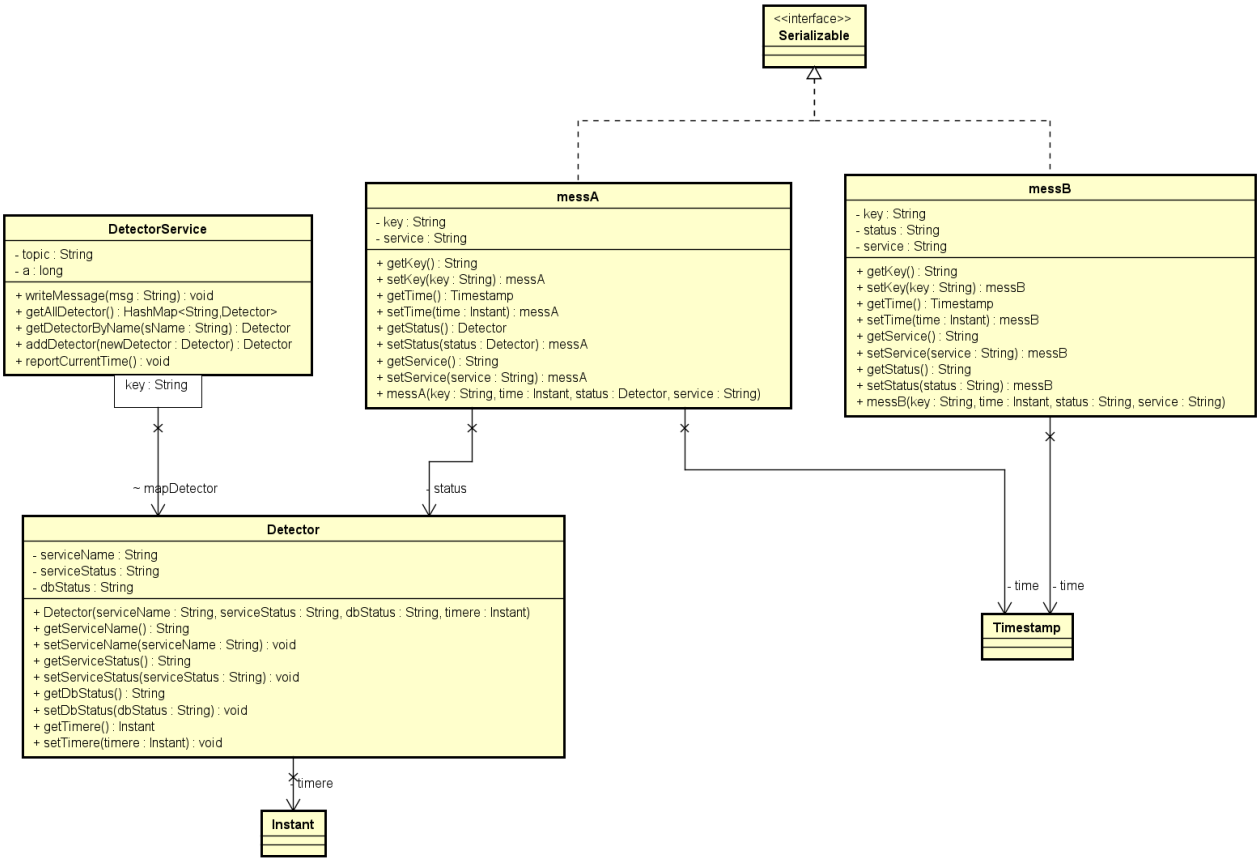
3. La classe **DetectorService**, che è la classe che si occupa di tutti i compiti svolti dal microservizio (memorizzazione della mappa, iterazione della mappa e segnalazioni con messaggi asincroni su kafka).



Il controller, ricevuta una richiesta valida, come mostrato nell'immagine, richiama la funzione **addDetector**, della classe DetectorService, la quale si occuperà di memorizzare la mappa, aggiungendo l'orario di inserimento o di aggiornamento dell'orario, che viene successivamente utilizzato per l'identificazione dei servizi unavailable.

Producer kafka

Come da consegna del progetto, il microservizio sviluppato, sarà un producer di messaggi sul topic logging nel caso in cui un altro microservizio si presenta con uno stato “down”, o nel caso un microservizio entra nello stato di unavailable.



Per l’invio dei messaggi, sono state utilizzate altre due classi (**messA** e **messB**), classi che differiscono tra loro soltanto dal tipo del parametro **status**. Le due classi sono contenute nel package Data, e vengono utilizzate dalla classe DetectorService, che per la comunicazione asincrona, si occupa di trasformare il contenuto dell’oggetto da Json a String, e poi invia il messaggio facendo uso della classe **KafkaTemplate**.

La configurazione della comunicazione viene implementata in una classe chiamata **KafkaConfiguration**, contenuta all’interno del package config.

Di seguito sono riportate le immagini dell’implementazione delle funzioni addDetector e reportCurrentTime. Quest’ultima si occupa della segnalazione di eventuali servizi unavailable.

In fase di test, è stato utilizzato il fake consumer da console.

```java
public Detector addDetector(Detector newDetector)
{
    Instant instant = Instant.now();
    newDetector.setTimere(instant);
    mapDetector.put(newDetector.getServiceName(), newDetector);
    if(newDetector.getServiceStatus().equals("down") || newDetector.getDbStatus().equals("down
    {
        messA messres = new messA( key: "service_down",instant,newDetector,newDetector.getServic
        writeMessage(new Gson().toJson(messres));
    }
    return newDetector;
}
```

```java
@Value("30000")
private long a;

@Scheduled(fixedRateString = "${period}")
public void reportCurrentTime()
{
    for (Map.Entry<String, Detector> entry : mapDetector.entrySet())
    {
        Instant now = Instant.now();
        if(now.getEpochSecond() - entry.getValue().getTimere().getEpochSecond() > (a/1000))
        {
            messB messres = new messB( key: "service_down",now, status: "{ serverUnavailable: No
            writeMessage(new Gson().toJson(messres));
        }
        //System.out.println("Key : " + entry.getKey() + " Value : " + entry.getValue().getT
    }
}
```

Variabili d'ambiente

All'interno del progetto si fa uso di variabili d'ambiente fisse, che è possibile prefissare prima del lancio dell'applicazione. Tra queste sono state inserite:

- **period,** che indica il periodo in millisecondi ogni qualvolta deve essere iterata la mappa dei servizi in memoria dal fault detector;
- **server** e **topic** utilizzati per la configurazione kafka.

Le variabili sopracitate sono settate e possono essere modificate all'interno del file **application.properties**, dove troviamo anche il nome del microservizio e la configurazione della porta locale (**1234**).

Docker

Il microservizio è stato creato come un modulo spring initializr all'interno di un progetto maven. All'interno del modulo **fault_detector** troviamo il **Dockerfile** di compilazione del microservizio.

Troviamo infine, all'interno del progetto, il **docker-compose.yml**, nel quale vengono dockerizzati, insieme al container con il nostro microservizio, le immagini di kafka e zookeeper.

```yaml
version: '3'

services:

  fault_detector:
    build:
      context: .
      dockerfile: fault_detector/Dockerfile
    ports:
      - "8080:1234"
    container_name: fault_detector
    restart: always

  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
    container_name: zookeeper

  kafka:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    container_name: kafka
```