Logging System

PROGETTO: 8

VARIANTE: MCV Mongo

STRATEGIA DI HEALTH-CHECK: Heart-beat

GIUSY AGATA BONGIOVANNI

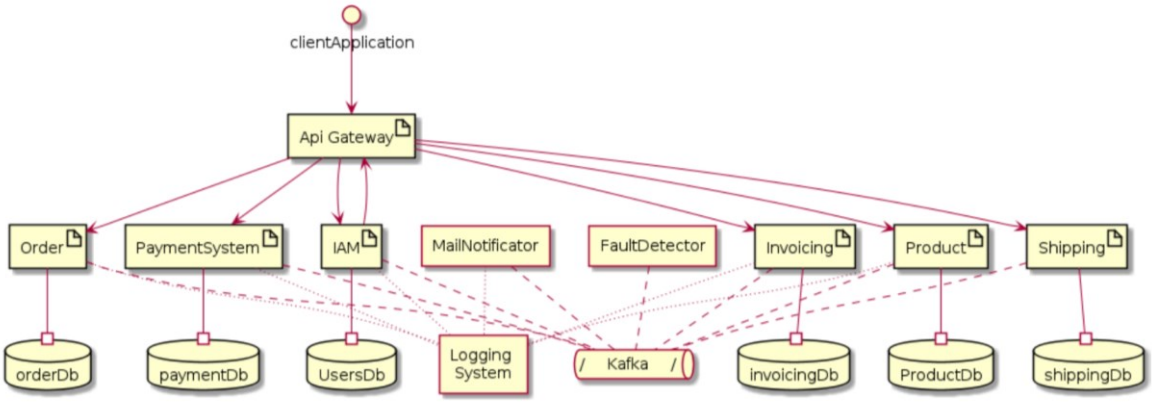
MATRICOLA: 1000008657 | DISTRIBUTED SYSTEMS AND BIG DATA AA 2020/21

Sommario

Introduzione

Il progetto 8 nella variante B prevede la realizzazione di un microservizio, denominato Logging System, che sia sottoscrittore del topic logging di Kafka.

Vediamo di seguito l'architettura generica di riferimento:



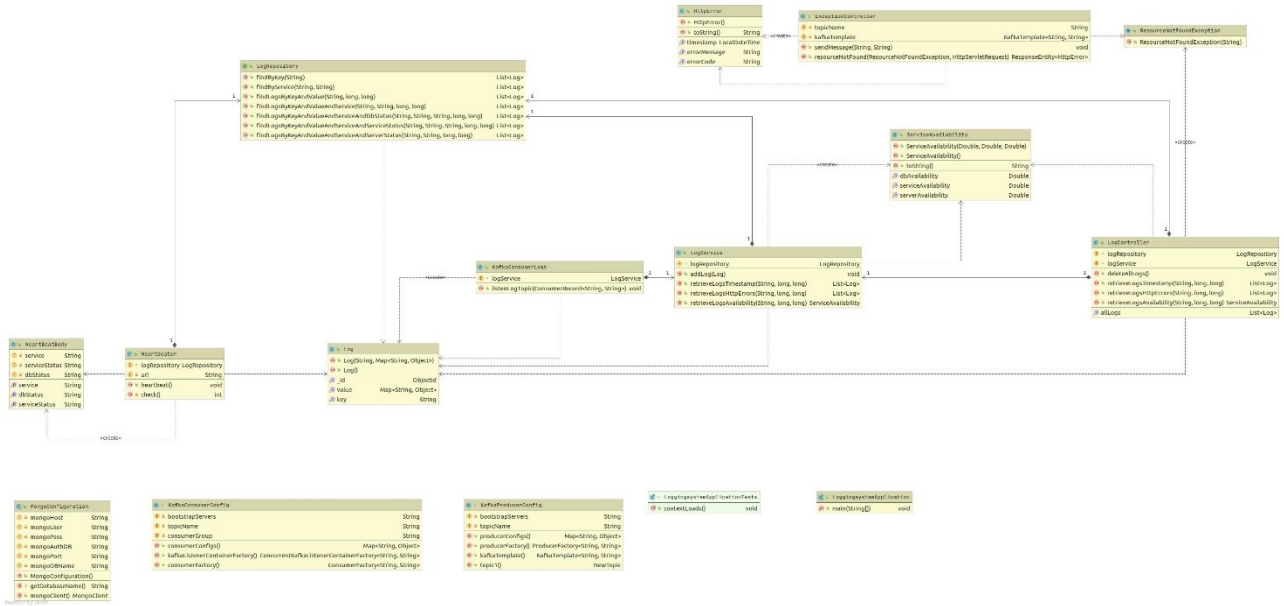
La consegna prevede che, alla ricezione di ognuno dei possibili messaggi, si salvi sul database MongoDB il loro contenuto, aggiungendo un timestamp se non è presente. I log devono essere disponibili via endpoints http. Inoltre, è prevista una gestione degli errori dove, al fallimento delle richieste HTTP, viene inviato un messaggio sul topic logging. La strategia di health-check prevista è quella di heart-beat.

Struttura generale e Docker

È stato realizzato un progetto Maven ***project8*** con all'interno un'applicazione Spring Boot ***loggingsystem***.

È stata realizzata una network denominata ***logging-system-network*** sfruttando il driver bridge, il quale ha permesso di realizzare una rete di interconnessione tra i vari docker container in pochi passi. I container creati, hanno al loro interno rispettivamente i services ***loggingsystem***, ***mongodb***, ***zoo*** e ***kafka***. È stato realizzato un ulteriore container denominato ***debug-container***, il quale viene sfruttato per il fake producer ***kafka_producer.py***.

Loggingsystem



Kafka

Il microservizio in questione, è sia produttore che consumatore del topic *logging*. Il deploy di Kafka è stato ottenuto aggiungendo al ***docker-compose.yml*** (il quale si trova all'interno della cartella *deploy* in *project8*) il codice riportato di seguito:

```
x-xxx-common-services-config: &common-services-config
  restart: always

x-kafka-env: &kafka-env
  KAFKA_BROKER_ID: 1
  KAFKA_ADVERTISED_PORT: 9092
  BROKER_ID_COMMAND: "hostname | cut -d'-' -f2"
  KAFKA_ZOOKEEPER_CONNECT: "zoo:2181"
  KAFKA_CREATE_TOPICS: "logging:20:1,pushnotifications:10:1,invoicing:10:1,mailing:10:1,userupdates:10:1,orderupdates:10:1"
  KAFKA_LISTENERS: "PLAINTEXT://:9092"

services:
  zoo:
    <<: *common-services-config
    image: library/zookeeper:3.4.13
    environment:
      ZOO_MY_ID: 1

  kafka:
    <<: *common-services-config
    environment: *kafka-env
    image: wurstmeister/kafka:2.11-2.0.0
```

Il Logging System, in particolare, deve leggere dal topic *logging* i messaggi inviati dagli altri microservizi, dunque, per simulare i producer, è stato utilizzato uno script Python:

```python
from kafka import KafkaProducer
import json

def producer(broker='kafka:9092', topic='my-topic'):
    p=KafkaProducer(bootstrap_servers=[broker], value_serializer=lambda x: json.dumps(x).encode('utf-8'))

    def send_and_flush(key: str, value: dict, *args, **kwargs):
        p.send(topic, key=key.encode('utf-8'), value=value, *args, **kwargs)
        p.flush()

    return send_and_flush
```

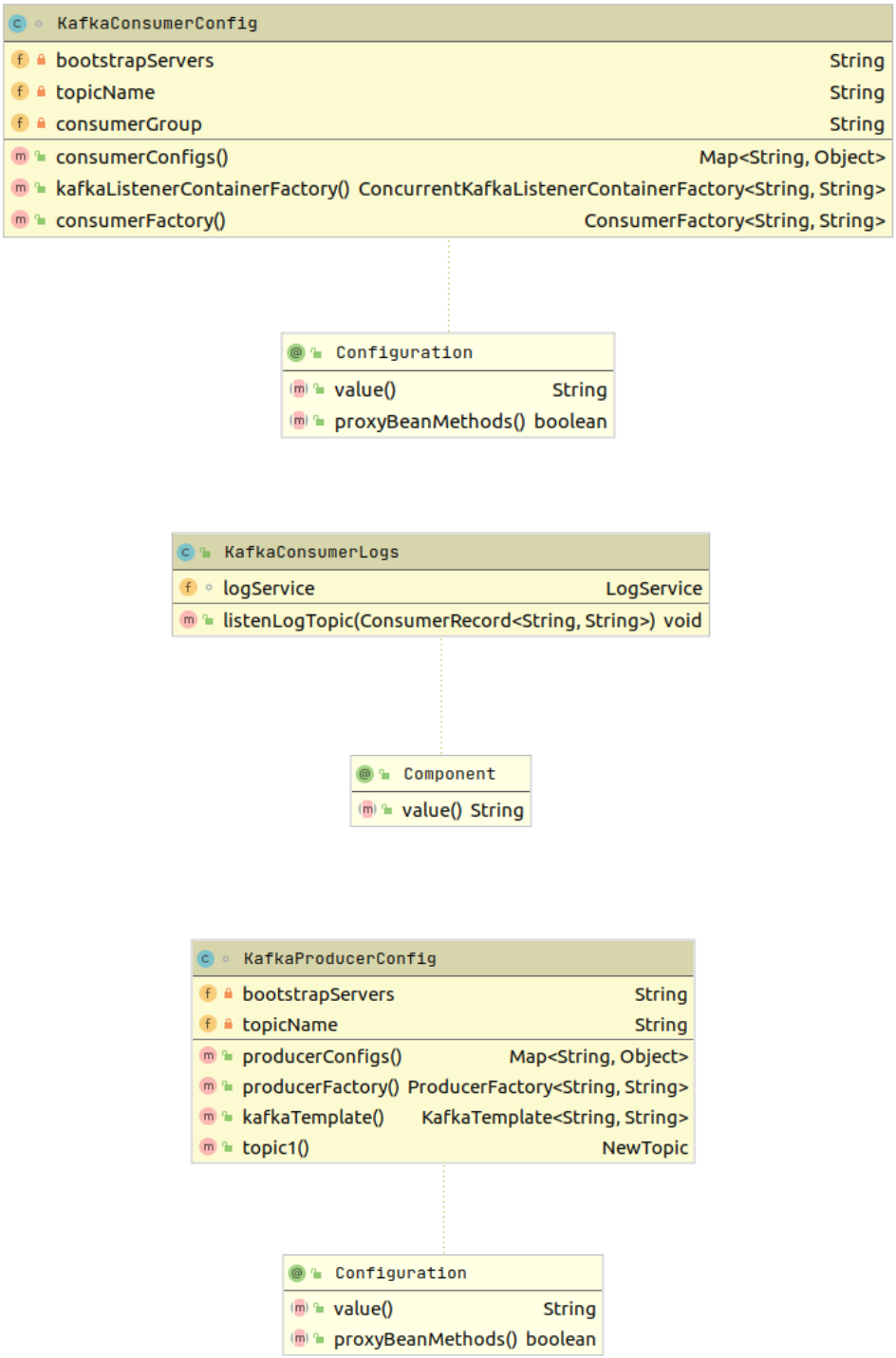
Lo script ***kafka_producer.py*** è stato inserito nella cartella ***fakeproducer***, insieme al suo ***Dockerfile***.

Dunque, all'avvio dell'applicazione attraverso il comando **docker-compose up**, bisognerà aprire un altro terminale in cui digitare **docker-compose run debug-container bpython**. Questo comando è necessario in quanto al docker-compose, nella sezione del debug-container, è stato aggiunto **command: exit 0** in modo tale che il container contenente il fake producer venga creato e mandato in run all'occorrenza. Una volta eseguito questo comando, basterà seguire il seguente esempio per inserire i vari messaggi che verranno visualizzati nel topic logging e che il microservizio salverà nel database:

```
>>> import kafka_producer              as kpi
>>> send = kpi.producer(topic='mailing')
>>> send('myKey', value={'myKeyStr': 'asd'})
```

All'interno dell'applicazione Spring Boot loggingsystem, troviamo un *package* ***kafka***, all'interno del quale troviamo i file .java ***KafkaConsumerConfig*** e ***KafkaProducerConfig***, rispettivamente per la configurazione di consumatore e produttore, e ***KafkaConsumerLogs***, in cui è implementato il Listener. Qui viene eseguito il controllo, dopo aver letto il messaggio log dal topic, del timestamp: se non c'è, viene aggiunto attraverso la funzione ***System.currentTimeMillis()***, la quale restituisce un tempo unix in millisecondi (sarà un *long*).

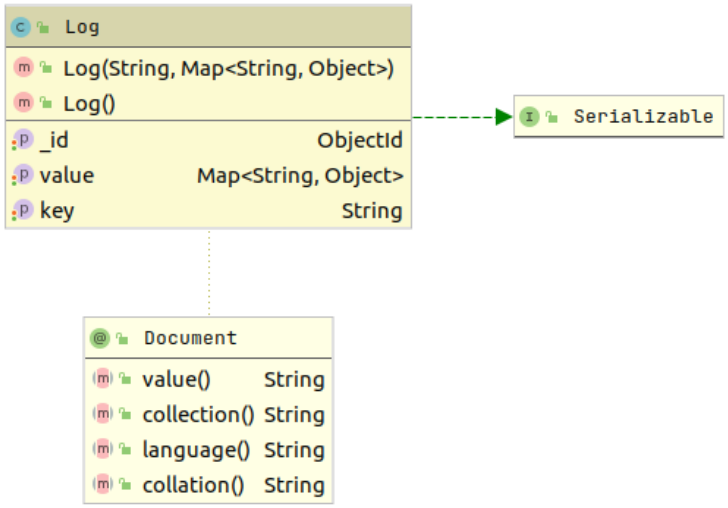
Di seguito sono riportati gli uml delle classi java del package kafka:



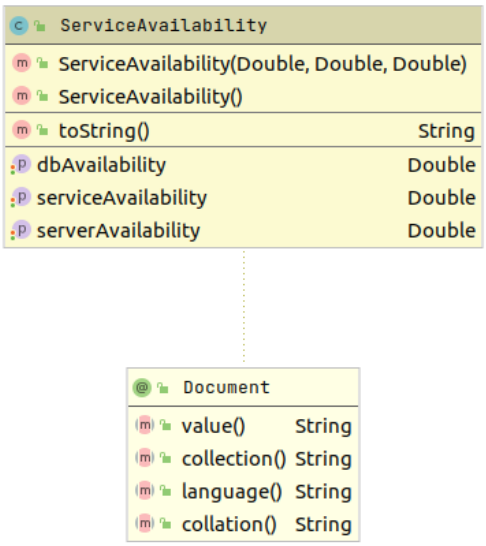
Package model

All'interno del package model, troviamo due classi java: ***Log*** e ***ServiceAvailability***.

La classe *Log* modella la struttura dei messaggi letti dal topic logging; essa possiede una *String key* e un *Map<String,Object> value*, in quanto, dopo un'attenta analisi del formato eterogeneo dei possibili messaggi da leggere, è stata individuata la struttura ricorrente che variava solo nel value: *Object* ci permette di avere qualunque tipo di dato come value della Map.

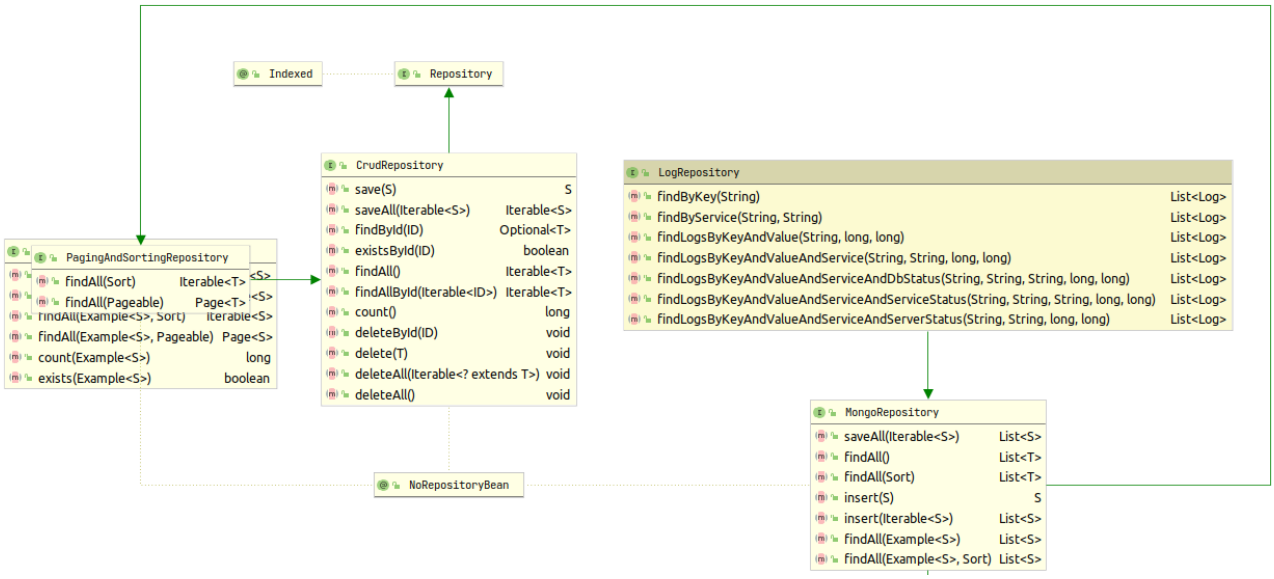


La classe *ServiceAvailability* invece modella il formato del messaggio di ritorno della GET che prevedeva il calcolo della availability di un dato service.



Package repository

Nel package repository troviamo la classe java ***LogRepository*** che estende *MongoRepository<Log,ObjectId>*. Questo permette di avere le operazioni CRUD di Mongo; in più, sono state implementate ulteriori query non presenti tra le CRUD attraverso l'annotazione **@Query**.



Package service

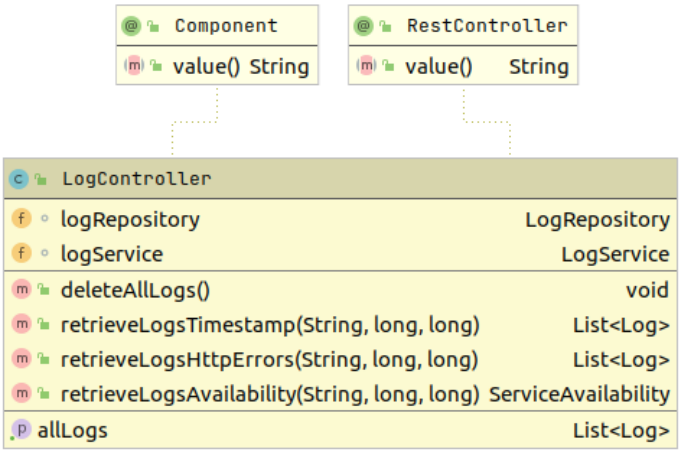
Nel package service troviamo due classi java: ***LogService*** e ***MongoConfiguration***.

LogService implementa la business logic e contiene tutti i metodi che vengono poi invocati dal controller.

MongoConfiguration invece serve alla configurazione di Mongo.

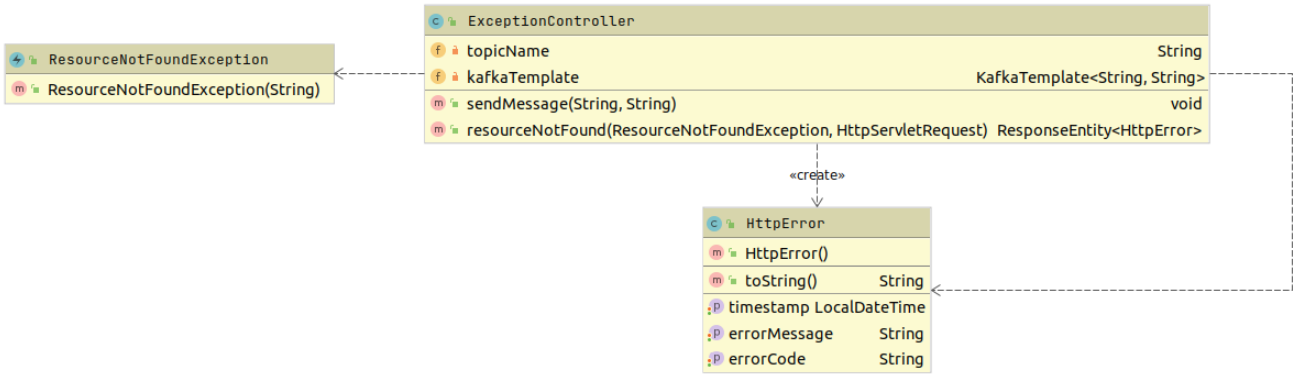
Package controller

Nella classe ***LogController*** troviamo le varie *@RequestMapping* per gli endpoints HTTP. Per comodità, sono state lasciate due *@GetMapping* rispettivamente per visualizzare tutti i log nel database e per eliminarli tutti. Sempre in questa classe vengono lanciate le eccezioni sui vari controlli delle GET.



Package errorhandling

In questo package troviamo tre classi: ***HttpError***, ***ExceptionController*** e ***ResourceNotFound***.



La classe *ResourceNotFoundException* modella il tipo di eccezione che viene lanciata nei seguenti casi:

- Se in **GET /keys/{key}?from=unixTimestampStart&end=unixTimestampEnd** viene utilizzata una key non presente nei log nel database;
- Se in **GET /keys/{key}?from=unixTimestampStart&end=unixTimestampEnd** vengono utilizzati range temporali all'interno dei quali non esistono log salvati associati a quella key;
- Se in **GET /http_errors/services/{service}?from=unixTimestampStart&end=unixTimestampEnd** viene inserito un service non presente nei log con key http_errors;

- Se in **GET /http_errors/services/{service}?from=unixTimestampStart&end=unixTimestampEnd** vengono utilizzati range temporali all'interno dei quali non esistono log salvati associati a quella key e a quel service;
- Se in **GET /uptime/services/{service}?from=unixTimeStampStart&end=unixTimestampEnd** unixTimestampStart e unixTimestampEnd non differiscono di 86400;
- Se in **GET /uptime/services/{service}?from=unixTimeStampStart&end=unixTimestampEnd** non vengono trovati log con quella key appartenenti a quel service.

La classe *HttpError* viene utilizzata come modello di messaggio che verrà visualizzato quando si effettuerà la richiesta GET.

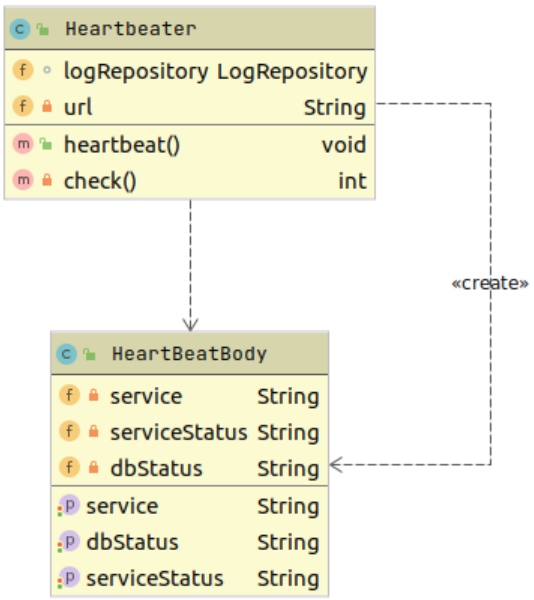
La classe *ExceptionController* è stata annotata con *@ControllerAdvice*, la quale è una specializzazione dell'annotazione *@Component*, e consente di gestire le eccezioni nell'intera applicazione in un unico componente di gestione globale. Può essere visto come un intercettore di eccezioni lanciate da metodi annotati con *@RequestMapping* e simili.

L'annotazione *@ExceptionHandler* è stata utilizzata per la gestione dell'eccezione *ResourceNotFound*; durante questa gestione, viene costruito il messaggio da inviare nel topic logging con il seguente formato:

```
key = http_errors
value = {
    timestamp: UnixTimestamp,
    sourceIp: sourceIp
    service: products,
    request: path + method
    error: error
}
```

Package health

All'interno di questo package troviamo le due classi utilizzate per l'health-checking: ***HeartBeatBody*** e ***Heartbeater***.



HeartBeatBody modella il formato di messaggio di heart-beat che viene inviato periodicamente.

La classe *Heartbeater* viene utilizzata per il check del database status e per la richiesta post. Lo stato del database viene testato eseguendo una query: se essa ha esito negativo, lo status viene considerato down.

Non avendo gli altri processi a cui mandare l'heart-beat message, ho messo un *url* a caso da cambiare in base al suo utilizzo; affinché l'esecuzione non risentisse dell'errore legato all'url non valido, ho utilizzato un *try{}catch(){}* che cattura l'eccezione *ResourceAccessException* e stampa un messaggio *"Connection error to /ping"*.