Introduce

Assignment

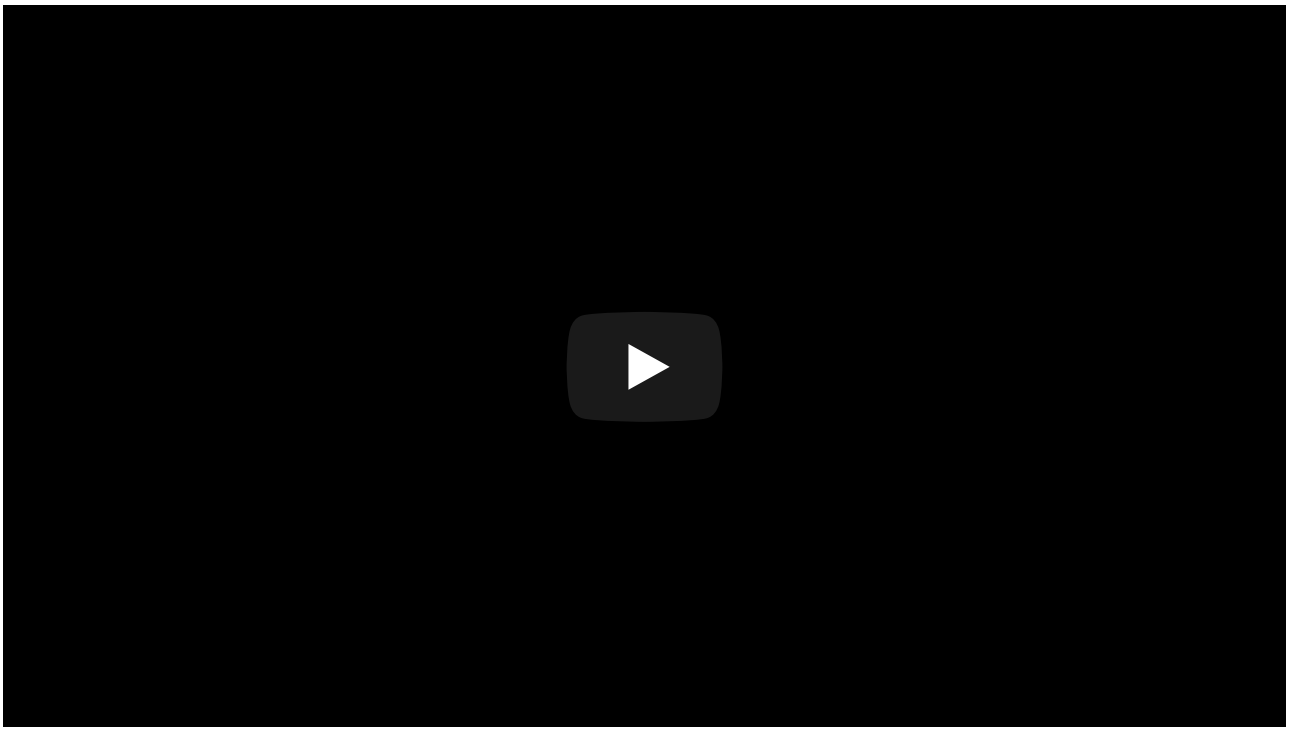
Assignment: simple counter API server based on docker

Write the code that satisfies the following conditions:

1. Produce result as described in detail in steps 1 through 6
2. All the code with the exception of shell scripts must run within Docker containers
3. Write the code in any language of your preference (Python, C++, GOLANG, etc.)
4. Write test code covered all your implementations as possible as you can.
5. Freely use services or libraries other than those suggested in below steps on your own convenience
6. Submit the result with the followings:
   a. Explain how to start your application in detail.
   b. Explain what your idea of implementation is including architectural diagram
   c. An archive of the hole project source directory (including the version control meta-data such as. git directory) or Provide an URL of a private GitHub repository

Assignment video

https://www.youtube.com/embed/Go7MGcnh45I



Quick Start

System Requirements

- **Docker**
- **Minimal 4GB Free Memory**

Using Prebuilt Docker Image

It takes 5 to 10 minutes for the stack to fully run.

```
$ docker run -p 3000:3000 \
```

```
$ docker run -p 3000:3000 \
-v /var/run/docker.sock:/var/run/docker.sock \
--name nexon-assignment \
--privileged --rm \
sppark/nexon-assignment:v1
```

Test Endpoint

After the container has started, check the list of containers on the host machine.

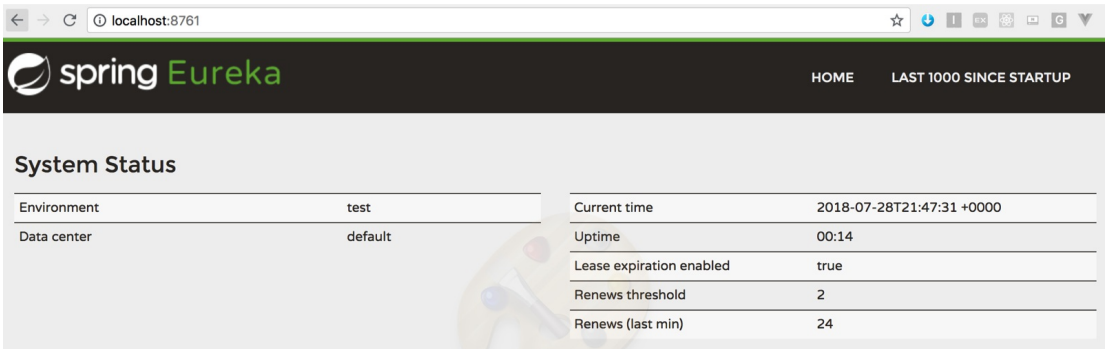
```
$ docker ps

CONTAINER ID        IMAGE                          COMMAND
CREATED             STATUS              PORTS
NAMES
04af0d3a1fbd        project_counter-service        "java -Xmx400M -Dj..."   2
seconds ago       Up 3 seconds        0.0.0.0:32769->9010/tcp
project_counter-service_1
9c3946b45ff0        project_gateway                "java -Xmx400M -Dj..."   3
seconds ago       Up 4 seconds        0.0.0.0:8080->8080/tcp
project_gateway_1
5679a544d577        wurstmeister/kafka             "start-kafka.sh"         4
seconds ago       Up 4 seconds        0.0.0.0:9092->9092/tcp
kafka-reactive-processor
d42c7d1fb562        project_eureka                 "java -Xmx400M -Dj..."   6
seconds ago       Up 5 seconds        0.0.0.0:8761->8761/tcp
project_eureka_1
89067dd16c1f        mongo:latest                   "docker-entrypoint..."   6
seconds ago       Up 5 seconds        0.0.0.0:27017->27017/tcp
mongo
f1da08fe7629        redis:latest                   "docker-entrypoint..."   6
seconds ago       Up 4 seconds        0.0.0.0:6379->6379/tcp
redis
44bd6dbd43e6        wurstmeister/zookeeper         "/bin/sh -c '/usr/..."   6
seconds ago       Up 5 seconds        22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181-
>2181/tcp   project_zookeeper_1
abe7932be8e5        project_nginx                  "nginx -g 'daemon ..."   6
seconds ago       Up 5 seconds        0.0.0.0:80->80/tcp
project_nginx_1
2cdb706b1f57        sppark/nexon-assignment:v1     "/project/entrypoi..."   9
seconds ago       Up 10 seconds       0.0.0.0:3000->3000/tcp
nexon-assignment
```

The <nginx-ip> location in the assignment: the ngnix port floating at 80.

How can I check that all of the stacks are working for test?

open your browser [http://localhost:8761 (http://localhost:8761)](http://localhost:8761)

As shown in the image, if both COUNTER-SERVICE andGATEWAY are in red boxes, the test is possible.



THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
COUNTER-SERVICE	n/a (1)	(1)	UP (1) - 04af0d3a1fbd:counter-service:9010
GATEWAY	n/a (1)	(1)	UP (1) - 9c3946b45ff0:gateway:8080

Build Start (Optional)

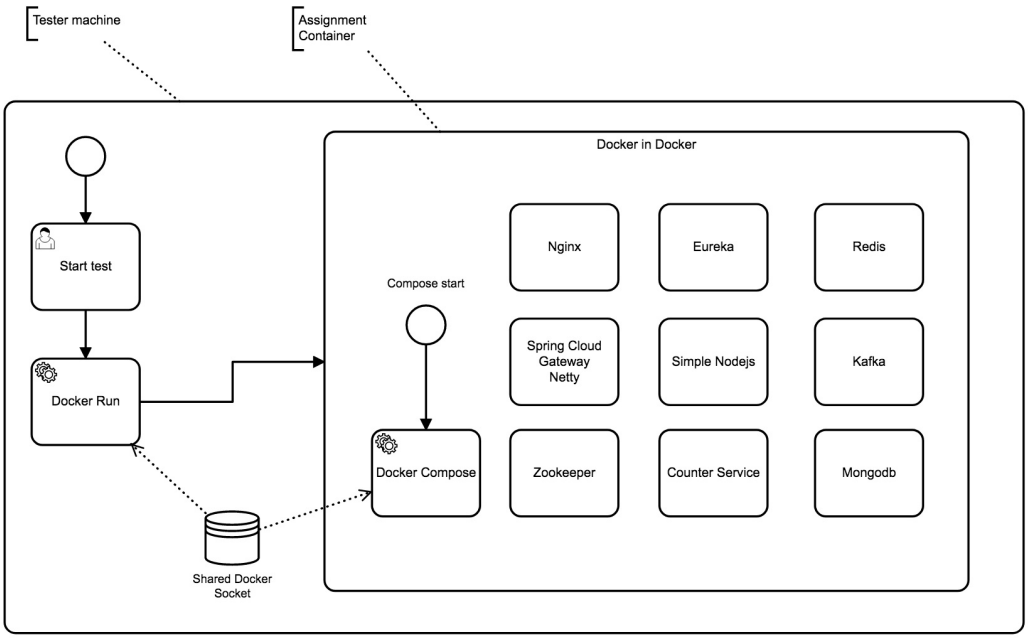
Build Requirements

- **Maven**

Build from source code

```
$ cd <your-project-path>
$ mvn install
$ docker build -t sppark/nexon-assignment:v1 ./
```

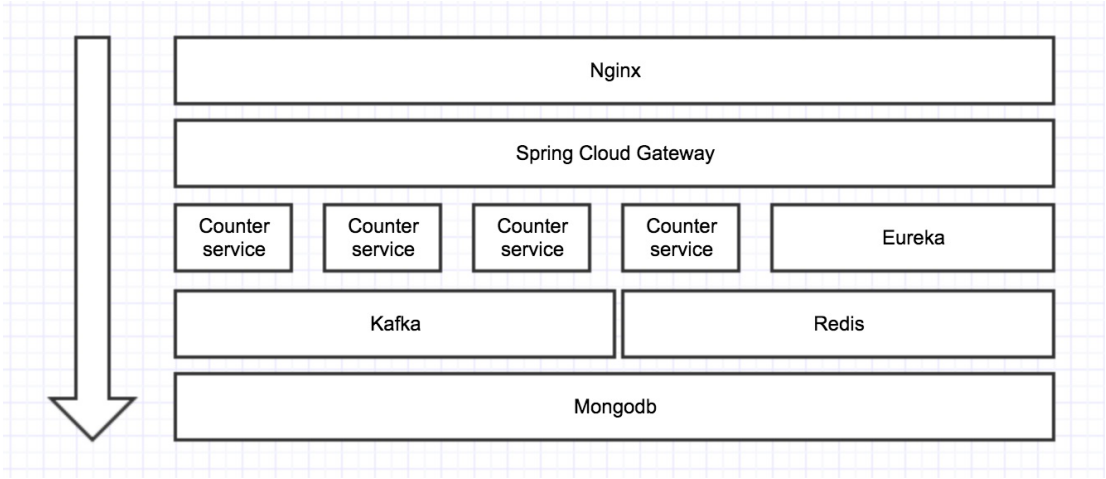
Stack configuration



- Docker compose : Tool for run stack. Run as Docker in Docker, with privileged mode and shared docker socket.
- Ngnix: Just Routing to Spring Cloud gateway
- Eureka: Service Discovery component

- Eureka: Service Discovery component
- Spring Cloud gateway : Netty based api gateway working with eureka.
- Counter service: A simple counter service mirco-service
- Mongodb: Entity repository
- Redis: Http Response cache repository
- Kafka: Message que for CQRS pattern
- Zookeeper: Coordinator for distributed systems. (In this example, only kafka use this)
- Node js: Just docker-compose scale commander

Layer



Ports in stack

Stack	Port
redis	6379
mongodb	27017
kafka	9092
zookeeper	2181
eureka	8761
gateway	8080
nginx	80
counter-service	random

Step1

Assignment

`$ ./setup_api.sh 3`

If script is executed, the above line must produce the following result:

- Spawn a nginx REST-API server docker container that will listen for requests on <nginx-ip> TCP port 80
- Spawn any number of API application back-end containers given as a parameter (3 in the example) that will register themselves to the REST-API server
- When HTTP GET / is requested at the REST-API server, back-end container hostname is returned
- REST-API server is scheduling the requests to its back-end by round-robin algorithm

The following sequence of requests must produce the following result:

```
$ for x in `seq 1 100`; do curl -s http://<nginx-ip>/ ; done | sort | uniq
host1
host2
host3
```

Things to ask when testing

After execute `setup_api.sh` , please wait until all counter services are ready.
You can check this http://localhost:8761 (http://localhost:8761).

If you do not have enough resources in your test environment to increase the number of instances, jvm will temporarily consume a lot of cpu and memory resources at boot time, causing the system to hang.
Thank you for your understanding in grading.

Explain

Container orchestration

Q. Spawn any number of API application back-end containers given as a parameter.

This is a question about understanding the concept of container orchestration.

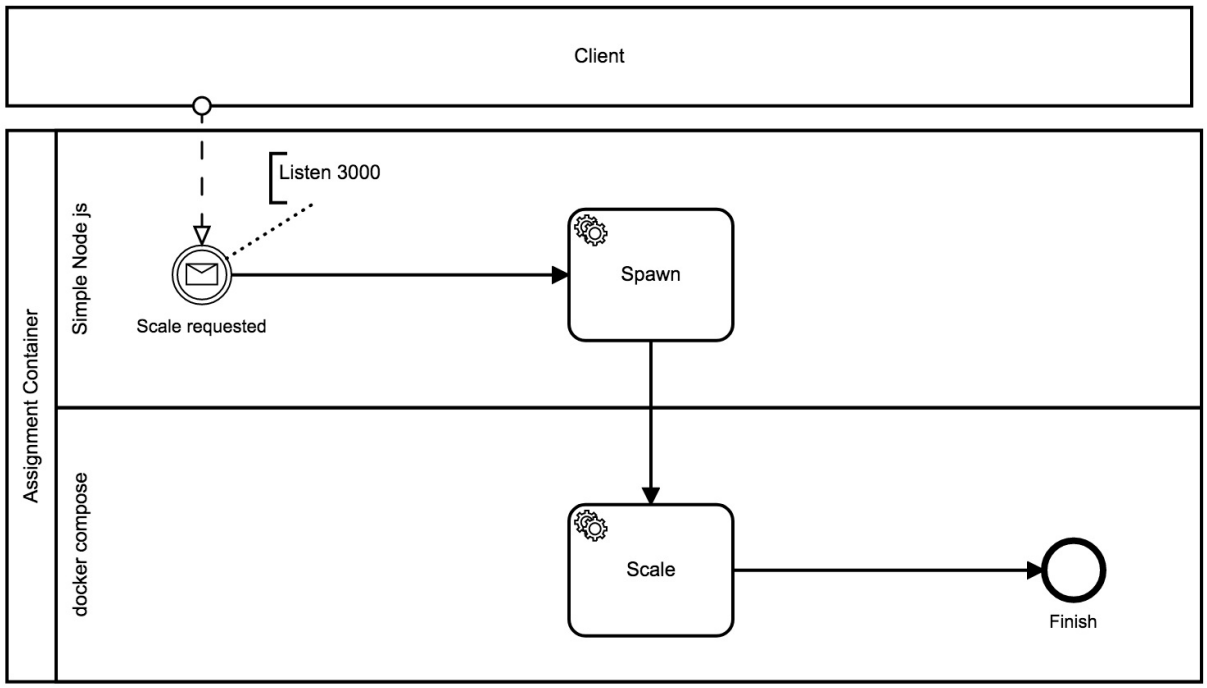
The container orchestrator should be capable of the following functions:

- Service broker / binding automation
- Resilience
- Auto Scaling
- Production debugging

`DC/OS`, `Kubernetes`, and `Docker swarm` are the right choices.

However, for minimal test environments, I have just created a docker-compose scale command.

Flow



Client `setup_api.sh`

```
#!/bin/bash
NODEJS_HOST=http://localhost:3000

curl -s ${NODEJS_HOST}?count=$1;
```

Simple Node js

`index.js`

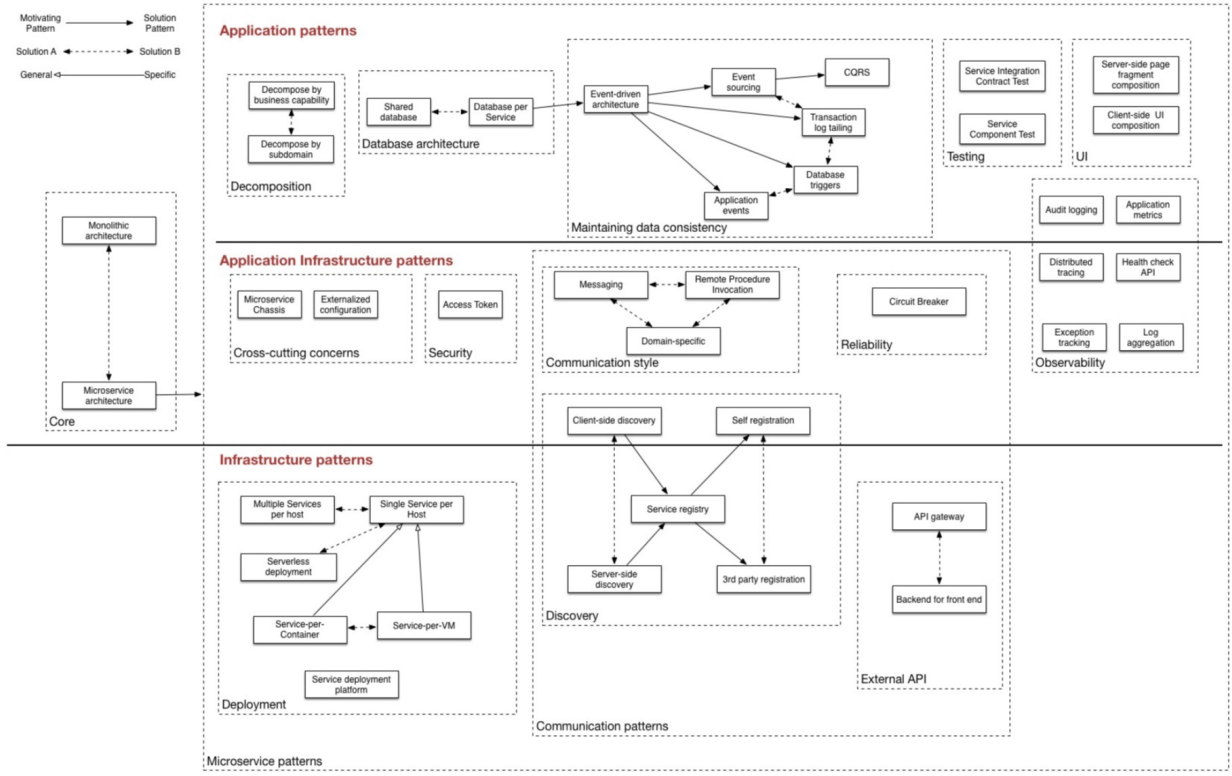
```
const express = require('express')
const app = express()
const { spawn } = require('child_process');

app.get('/', (request, response) => {

    var count = request.param('count');
    console.log(count);

    const command = spawn('docker-compose', ['scale', 'counter-service=' +
count]);
    command.stdout.on('data', (data) => {
        console.log(`stdout: ${data}`);
    });

    command.stderr.on('data', (data) => {
        console.log(`stderr: ${data}`);
    });

    command.on('close', (code) => {
        response.json({
            result: `child process exited with code ${code} , requested server
count ${count}`
        });
    });
});

app.listen(3000)
```

Service Self registration & Api gateway

Q. register themselves to the REST-API server & REST-API server is scheduling the requests to its back-end by round-robin algorithm

This is a question about understanding the concept of service discovery and api-gateway

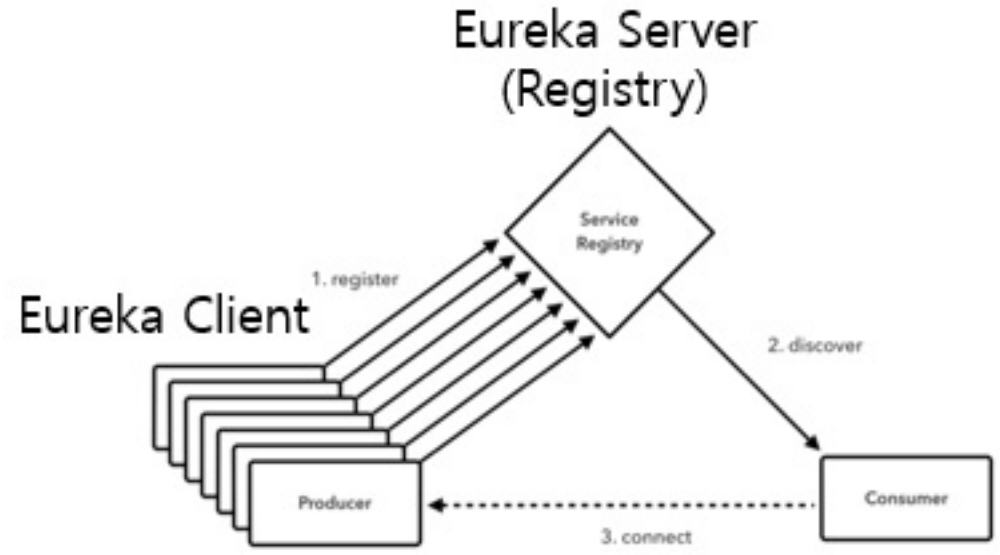
I used nefflix oss microservice pattern for this.



Those minimal components working for this.

- Eureka: Service Discovery component
- Spring Cloud gateway : Api-gateway
- Counter service: A microservice

1. Service discovery



Eureka Server

EurekaApplication.java

```
@SpringBootApplication
@EnableEurekaServer ❶
public class EurekaApplication {

    private static Log logger = LogFactory.getLog(EurekaApplication.class);

    public static void main(String[] args) {
        new
SpringApplicationBuilder(EurekaApplication.class).web(true).run(args);
    }

}
```

❶ Start as a eureka server.

Counter service

Counter service include eureka client, and in boot-up time, it will register to eureka itself.

CounterApplication.java

```
@SpringBootApplication
@EnableDiscoveryClient ❶
public class CounterApplication {

    private final Log logger = LogFactory.getLog(getClass());

    public static void main(String[] args) {
        ConfigurableApplicationContext ctx = new
SpringApplicationBuilder(CounterApplication.class).run(args);
    }
}
```

❶ Start including eureka client.

application.yml

```
eureka:
  client:
    registry-fetch-interval-seconds: 5
    register-with-eureka: true
    serviceUrl:
      defaultZone: http://${EUREKA_HOST}:8761/eureka/ ❶
      healthcheck:
        enabled: true
  instance:
    leaseRenewalIntervalInSeconds: 5
    leaseExpirationDurationInSeconds: 5 ❷
    statusPageUrlPath: ${server.servlet.context-path}info
    healthCheckUrlPath: ${server.servlet.context-path}health
    metadataMap:
        deployment: docker
        profile: docker
```

❶ The eureka server host to registration.

❷ This service will be unregistered if there is no response for 5 seconds.

2. Api gateway

Typically, zuul is used as a micro-service gateway.

However, since zuul does not support non blocking and socket communication, spring cloud gateway based on netty is getting attention.

https://cloud.spring.io/spring-cloud-gateway/ (https://cloud.spring.io/spring-cloud-gateway/)

gateway

pom.xml

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId> ❶
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
❷
        </dependency>
    </dependencies>
```

❶ Application start as a gateway

❷ Application know service location information from eureka server.

application.yml

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
      routes: ❶
      - id: host
        uri: lb://counter-service ❷
        predicates: ❸
        - Path=/**
        filters: ❹
        - StripPrefix=0
        - AddResponseHeader=Access-Control-Allow-Origin, *
```

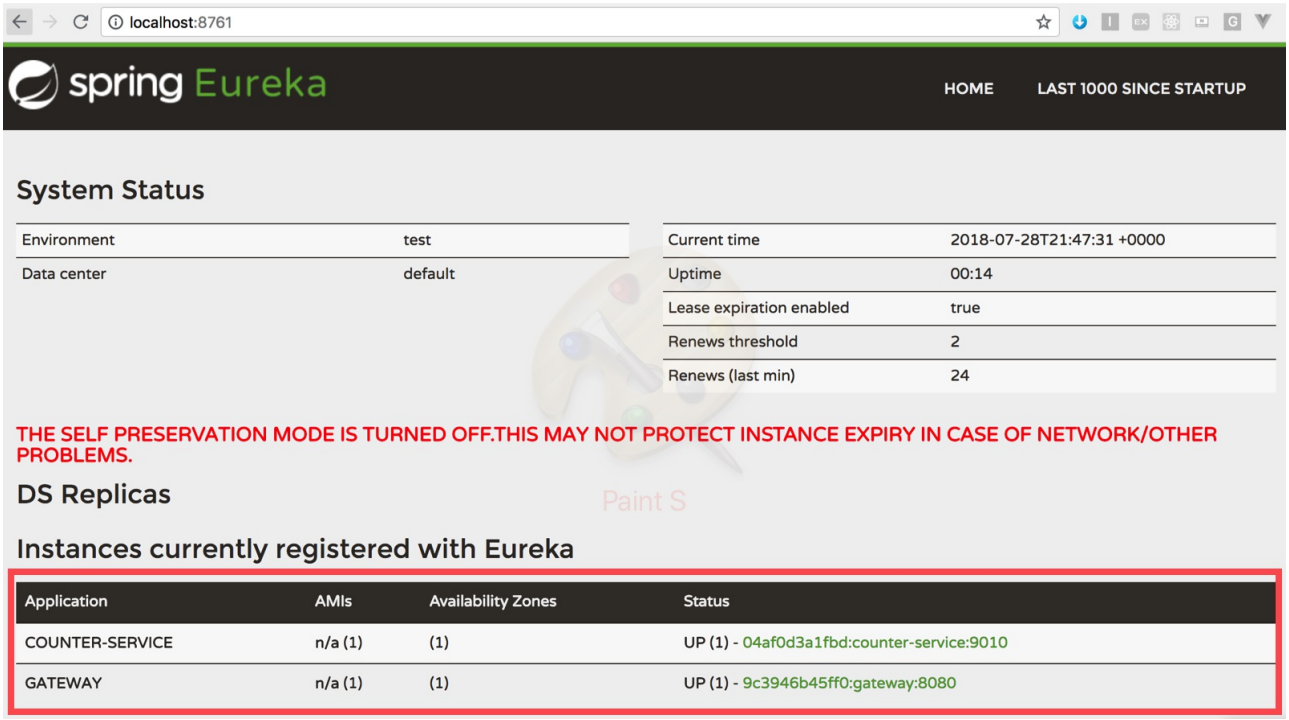
❶ route list

❷ proxy url. `lb://counter-service` means *counter-service* application list in eureka.

❸ input path

❹ simple pre / post filters before and after proxy

After registration, both `COUNTER-SERVICE` and`GATEWAY` are in red boxes.



Step2

Assignment

A) When a POST request to /counter is sent with "to" argument, the following result will be produced:

- A counter data entity will be created whose value will increment its value by 1, each second
- A counter UUID must be returned immediately upon being created
- A counter must disappear when its value reaches 0

B) When GET request to /counter/<UUID> is sent, current value of a counter will be returned in JSON format as displayed below:

```
$ curl -X POST http://<nginx-ip>/counter/?to=1000 A4C2605C-5196-4815-BA26-
463EB03E6C92
$ curl http://<nginx-ip>/counter/A4C2605C-5196-4815-BA26-463EB03E6C92/
{"current", 5, "to": 1000}
$ sleep 1 && curl http://<nginx-ip>/counter/A4C2605C-5196-4815-BA26-
463EB03E6C92/
{"current", 6, "to": 1000}
```

C) Use a create_counter.sh script shown below to generate 1000 counters:

```
$ cat create_counter.sh #!/bin/sh
for x in `seq 1 100`; do
curl -X POST "http://<nginx-ip>/counter/?to=$(((RANDOM%1000)+1000))"; done

$ ./create_couter.sh
```

Explain

Q. Counter entity will be produced:

- A counter data entity will be created whose value will increment its value by 1, each second
- A counter UUID must be returned immediately upon being created
- A counter must disappear when its value reaches 0

This is a question about understanding the concept of "Scheduling Consistency in Distributed Service Environments".

In addition, "CQRS patterns and Non-blocking patterns" can be introduced for better design.

The counter service should be capable of the following functions:

- Basic rest api operation
- Input and output processes are designed as non-blocking
- Schedule time should not be affected by service processing performance.

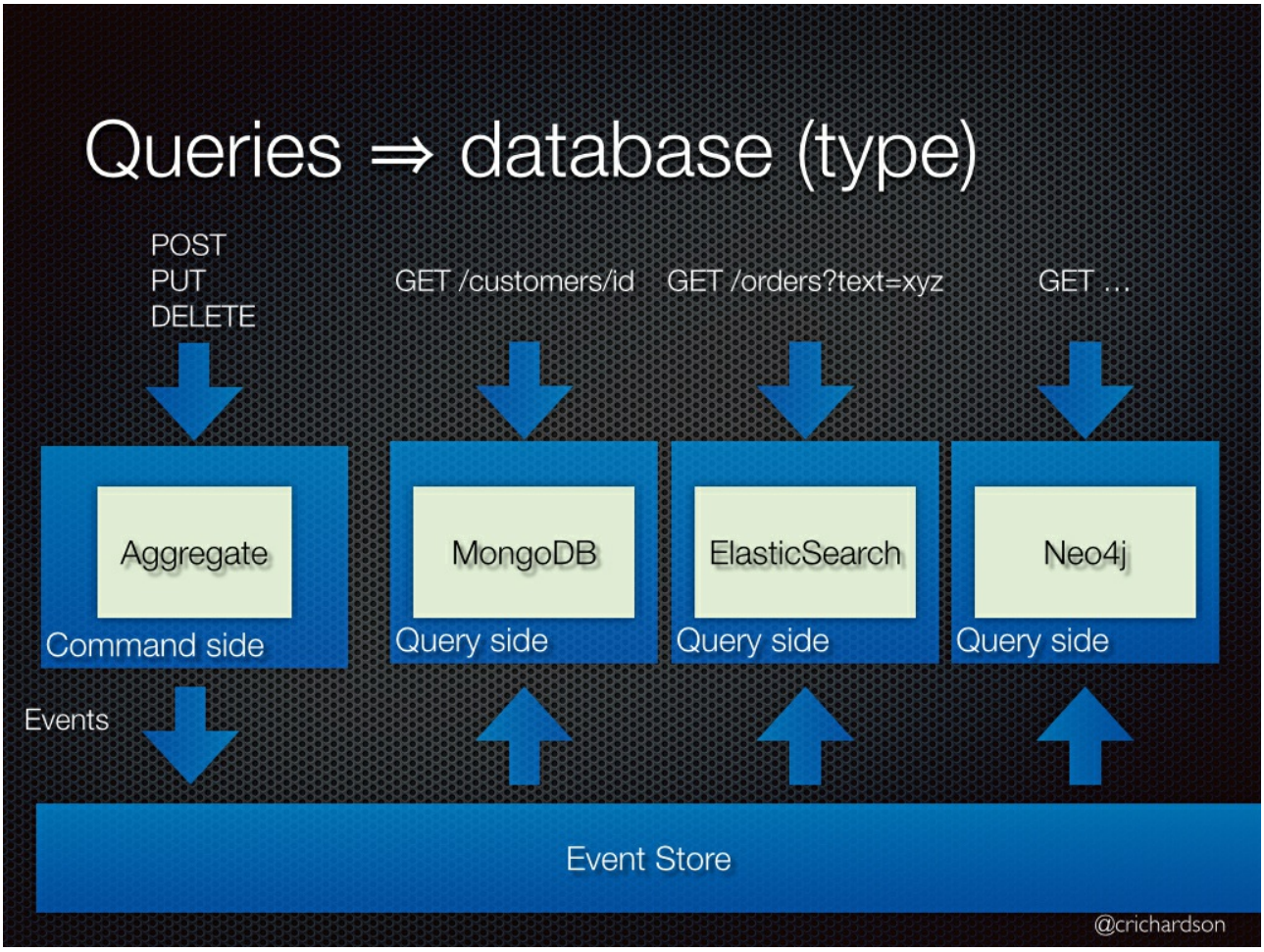
CQRS

Problem

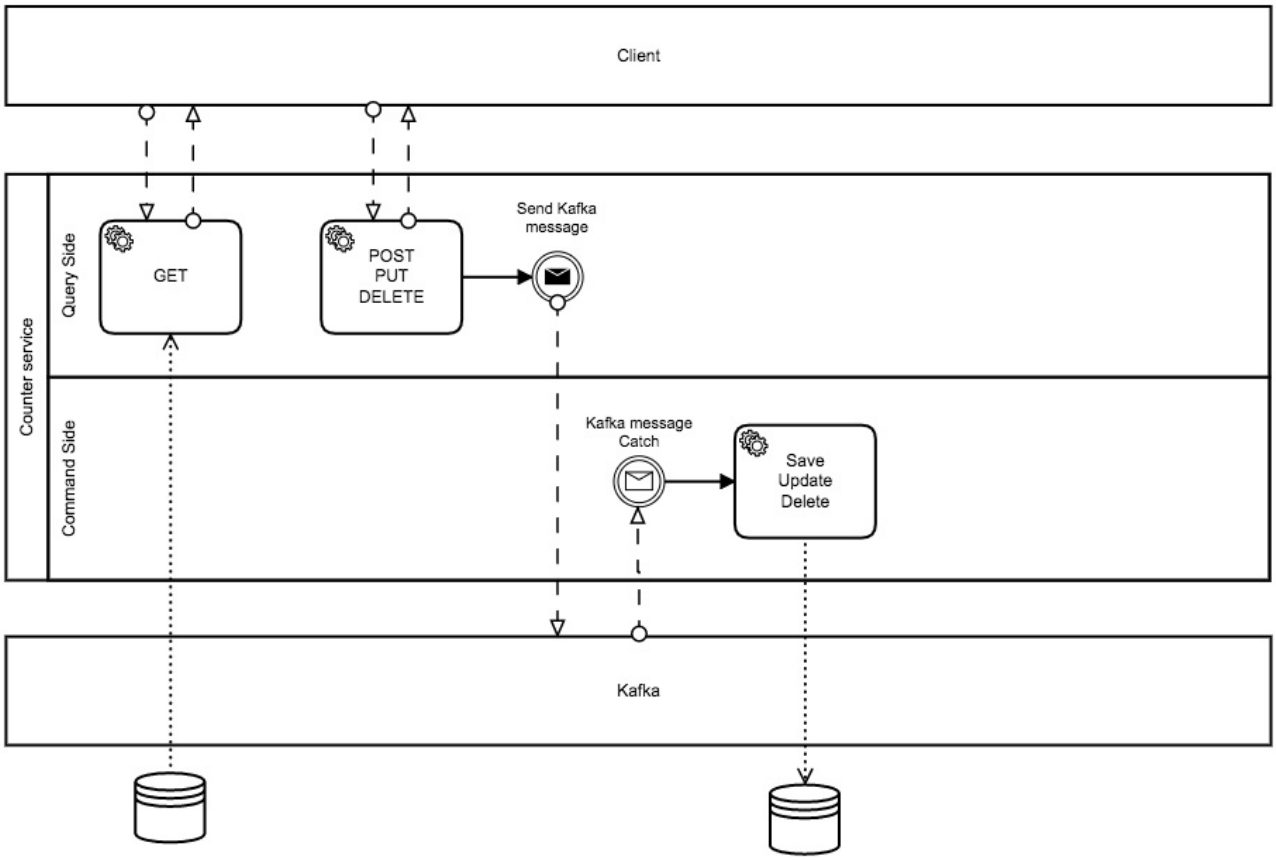
How to implement queries in a microservice architecture?

Solution

Split the application into two parts: the command-side and the query-side. The command-side handles create, update, and delete requests and emits events when data changes. The query-side handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.



Code



Entity

Counter.java

```
@Document(collection = "counter")
@Data
public class Counter {

    @Id
    private String id;

    private String uuid;

    @Range(min = 0, max = 10000)
    private Long current;

    @Range(min = 0, max = 10000)
    private Long to;

    @CreatedDate
    private Date createdDate;
}
```

Repository

CounterRepository.java

```
public interface CounterRepository extends ReactiveCrudRepository<Counter,
String> {

    Mono<Counter> findByUuid(String uuid);
}
```

Kafka chanel

Define the Kafka's input and output channels

CounterStreams.java

```
public interface CounterStreams {
    String INPUT = "counter-target";
    String OUTPUT = "counter-source";

    @Input("counter-target")
    SubscribableChannel counterTarget();

    @Output("counter-source")
    MessageChannel counterSource();
}
```

StreamsConfig.java

```
@EnableBinding(CounterStreams.class)
public class StreamsConfig {

}
```

Kafka message

Sets the format of the message to be input / output via Kafka.

CounterMessage.java

```
public class CounterMessage {
    private Counter counter;

    private String method;

    public Counter getCounter() {
        return counter;
    }

    public void setCounter(Counter counter) {
        this.counter = counter;
    }

    public String getMethod() {
        return method;
    }

    public void setMethod(String method) {
        this.method = method;
    }
}
```

Chanel Processor

Defines input listeners and output channel methods.

CounterProcessor.java

```
@Service
public class CounterProcessor {


    @Autowired
    private CounterRepository counterRepository;

    private final Log logger = LogFactory.getLog(getClass());

    private CounterStreams counterStreams;

    public CounterProcessor(CounterStreams counterStreams) {
        this.counterStreams = counterStreams; ❶
    }

    @Async ❷
    public void sendCounterMessage(final CounterMessage counterMessage) {
        logger.info("Sending counterMessage : " + counterMessage.getMethod());

        MessageChannel messageChannel = counterStreams.counterSource();
        messageChannel.send(MessageBuilder
                .withPayload(counterMessage)
                .setHeader(MessageHeaders.CONTENT_TYPE,
MimeTypeUtils.APPLICATION_JSON)
```

```
                .build()); ❸
    }

    @StreamListener ❹
    public void receiveCounterMessage(@Input(CounterStreams.INPUT) Flux<String>
inbound) {
        inbound ❺
                .log()
                .subscribeOn(Schedulers.elastic())
                .subscribe(value -> { ❻
                    try {
                        CounterMessage counterMessage = new
ObjectMapper().readValue(value, CounterMessage.class);
                        Counter counter = counterMessage.getCounter();
                        String method = counterMessage.getMethod();

                        switch (method) {
                            case "POST": {
                                counterRepository.save(counter)
                                        .block();
                                break;
                            }

                            case "PUT": {
                                counterRepository.save(counter)
                                        .block();
                                break;
                            }

                            case "DELETE": {
                                counterRepository.delete(counter)
                                        .block();
                                break;
                            }
                        }

                    } catch (Exception ex) {

                    }
                }, error -> System.err.println("CAUGHT " + error));

    }
}
```

❶ CounterProcessor EnableBinding with Kafka Chanel Bean "CounterStreams"

❷ It should work Async, cause your system should not block the system during the event queue dispatch time.

❸ Build message and send to kafka

❹ Receive from kafka.

❺ Event comes as "Flux" stream. Flux is a stream listener declaration for non-blocking.

❻ Define what to do per each message

Controller

CounterContoller.java

```
@RestController
@RequestMapping("/counter")
public class CounterController {

    private final Log logger = LogFactory.getLog(getClass());

    @Autowired
    CounterRepository counterRepository;

    @Autowired
    CounterProcessor counterProcessor;

    @GetMapping("")
    public Flux<String> listCounterUUIds() {
        Flux<Counter> all = counterRepository.findAll();
        return all
                .log()
                .map(counter -> counter.getUuid() + "\n");
    }

    @GetMapping("/counts-all")
    public Mono<Long> counts() {
        return counterRepository.count();
    }

    @PostMapping("")
    public String sendCounterCreateMessage(@RequestParam(required = true, value
= "to") Long to) {
        Counter counter = new Counter();
        counter.setTo(to);
        counter.setUuid(UUID.randomUUID().toString());

        CounterMessage message = new CounterMessage();
        message.setMethod("POST");
        message.setCounter(counter);

        counterProcessor.sendCounterMessage(message);
        return counter.getUuid() + "\n";
    }

    @GetMapping("/{uuid}")
    public Mono<ResponseEntity<Map>> getCounterSimpleFormat(@PathVariable(value
= "uuid") String uuid) {
        Map map = new HashMap();
        return counterRepository.findByUuid(uuid)
```

```
                .log()
                .map(counter -> {
                    map.put("current", counter.getCurrent());
                    map.put("to", counter.getTo());
                    return new ResponseEntity<Map>(map, HttpStatus.OK);
                })
                .defaultIfEmpty(new ResponseEntity<>(new HashMap(),
HttpStatus.NOT_FOUND));
    }

    @PostMapping("/{uuid}/stop")
    public ResponseEntity<String> deleteCustomer(@PathVariable("uuid") String
uuid) {
        try {
            counterRepository.findByUuid(uuid)
                    .map(counter -> {
                        return counterRepository.delete(counter)
                                .subscribeOn(Schedulers.elastic())
                                .subscribe();
                    })
                    .subscribeOn(Schedulers.elastic())
                    .subscribe();
        } catch (Exception e) {
            return new ResponseEntity<>("", HttpStatus.OK);
        }
        return new ResponseEntity<>("", HttpStatus.OK);
    }
}
```

Leaders election in a Distributed System Environment

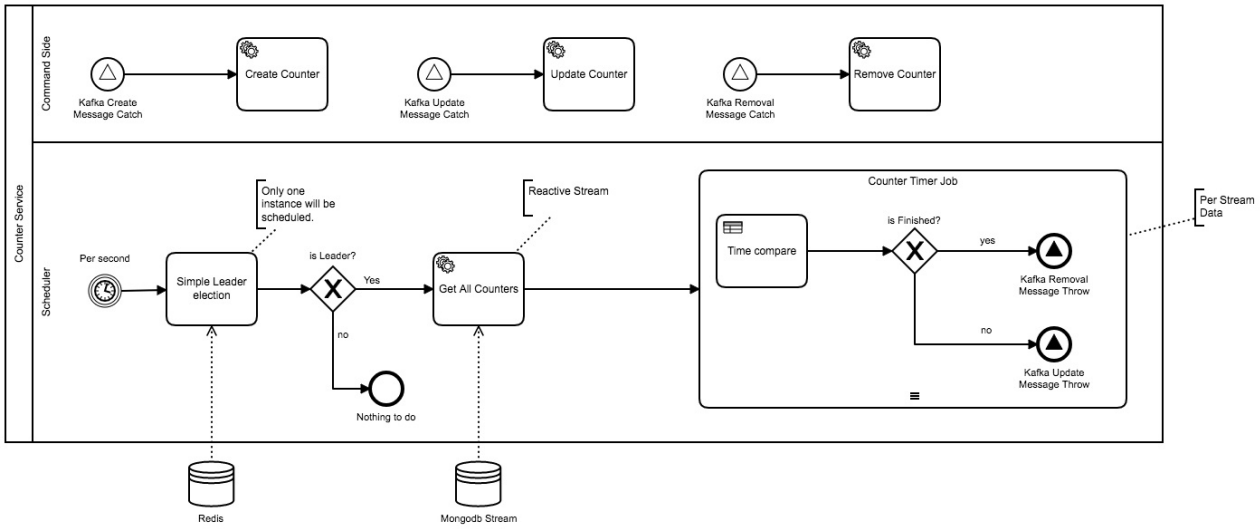
Problem

Some microsystems need to control their own scheduling of their own data.

Solution

Let microservice choose a leader to do its own scheduling. And let's apply the cloud stream concept so that the actual operation code for the schedule can be performed by all the micro services.

Flow



1. Use redis lock for leader election.
2. Per second, read from mongodb stream all counters.
3. Sends a message immediately to the Kafka for each stream data.
4. All microservices receive kafka message and actual operation code is executed.

Code

Leader Election

LeaderWrapper.java

```
@Service
public class LeaderWrapper {

    private static final int LOCK_TIMEOUT = 5000;
    private static final String LEADER_LOCK = "LEADER_LOCK";
    private boolean isLeader;

    @Autowired
    private String myApplicationId; ❶

    @Autowired
    private RedisTemplate redisTemplate; ❷

    private ValueOperations valueOperations;

    @PostConstruct
    private void init() {
        valueOperations = redisTemplate.opsForValue();
    }

    @Scheduled(fixedDelay = 2000) ❸
    public void tryToAcquireLock() {

        try {
            Object existLock = valueOperations.get(LEADER_LOCK); ❹
```

```
            //if null, set me new leader
            if (existLock == null) { ❺
                valueOperations.set(LEADER_LOCK, myApplicationId, LOCK_TIMEOUT,
TimeUnit.MILLISECONDS);
                isLeader = true;
                return;
            }


            //if existLock equals myApplicationId, reset value with timeout.
            isLeader = myApplicationId.equals(existLock); ❻
            if (isLeader) {
                valueOperations.set(LEADER_LOCK, myApplicationId, LOCK_TIMEOUT,
TimeUnit.MILLISECONDS);
            }

        } catch (Exception ex) {
            isLeader = false;
        }

        if (isLeader) {
            System.out.println("[" + myApplicationId + "] Now I am leader
node");
        } else {
            System.out.println("[" + myApplicationId + "] It's sad being a non-
leader node :-( ");
        }
    }

    public boolean amILeader() { ❼
        return isLeader;
    }

}
```

❶ Unique application id is myApplicationId

❷ Redis connection

❸ Execute Leader election code per 2 seconds

❹ Get current redis lock

❺ If lock is null, set me new leader

❻ If existLock equals myApplicationId, reset value with timeout.

❼ Services know am i leader by query this method.

Scheduler

CounterScheduler.java

```
@Component
```

```
public class CounterScheduler {

    @Autowired
    private CounterRepository counterRepository;

    @Autowired
    private CounterProcessor counterProcessor;

    private static final Logger LOGGER =
LoggerFactory.getLogger(CounterScheduler.class);

    @Autowired
    private LeaderWrapper leaderWrapper;

    @Scheduled(initialDelay = 1000, fixedDelay = 1000) ❶
    public void leaderScheduler() {
        //if not leader, skip.
        if (!leaderWrapper.amILeader()) {
            return;
        }
        try {
            this.processTimerJob();
        } catch (Exception ex) {

        }
    }

    @Async ❷
    public void processTimerJob() {
        try {
            LOGGER.info("leader processTimerJob start");

            long currentTime = new Date().getTime();
            counterRepository.findAll() ❸
                    .flatMap(counter -> {
                        CounterMessage counterMessage = new CounterMessage();
                        long diff = (currentTime -
counter.getCreatedDate().getTime()) / 1000;
                        counter.setCurrent(diff); ❹

                        if (counter.getTo() <= diff) {
                            LOGGER.info("Kafka Removal Message Throw, {}",
counter.getUuid());
                            counterMessage.setMethod("DELETE");
                            counterMessage.setCounter(counter);

counterProcessor.sendCounterMessage(counterMessage); ❺
                        } else {
                            counterMessage.setMethod("PUT");
                            counterMessage.setCounter(counter);

counterProcessor.sendCounterMessage(counterMessage); ❺
```

```
                        }
                        return Mono.just(counter);
                    })
                    .subscribeOn(Schedulers.elastic())
                    .subscribe(); ❻
        } catch (Exception ex) {
            LOGGER.error("leader processTimerJob failed.");
        }
    }
}
```

❶ Scheduler call "processTimerJob" every 1 seconds, if i am leader

❷ Async call processTimerJob

❸ Stream Lead Reservation from mongodb

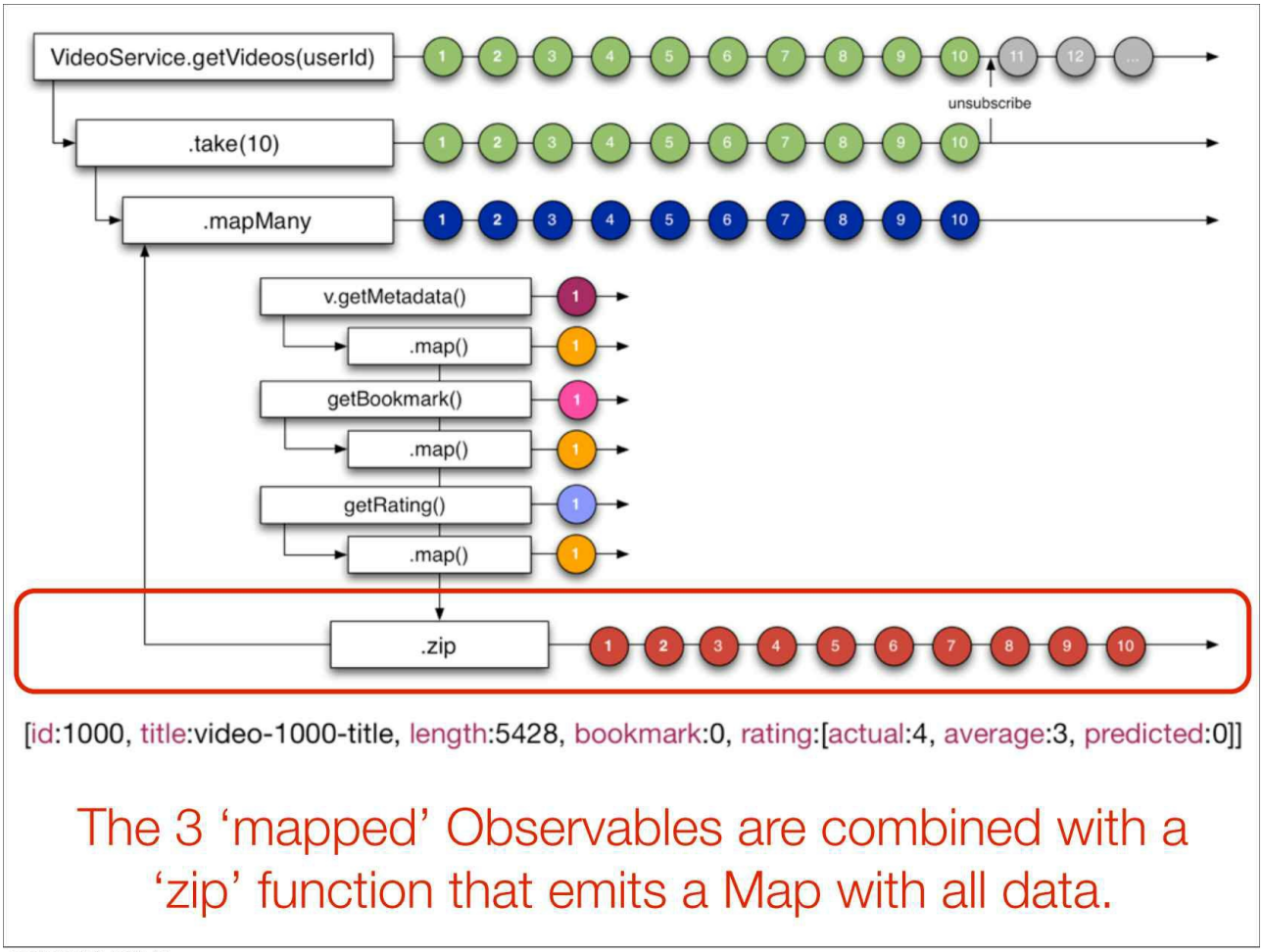
❹ Per Stream data, time compare

❺ Send to kafka with entity operation.

❻ Subscribe the stream, work as non blocking reactor stream.

What is reactor stream?

Reactor is a fourth-generation Reactive library for building non-blocking applications on the JVM based on the Reactive Streams Specification



Step3

Assignment

Query all created counters with the server API and show their current values, as shown in the following example:

```
$ cat list_counter.sh

#!/bin/bash
for x in $(curl -s http://<nginx-ip>/counter/); do
    curl -s http://<nginx-ip>/counter/${x}/
done

$ ./list_counter.sh
{"current": 5, "to": 1000} ....

$ ./list_counter.sh | wc -l
100
```

Explain

There is nothing special, and it is the same as the explanation of Step2 (Step2.md).

Step4

Assignment

Dynamically change the number of API application servers:

- Reduce the number of API application servers to 0
- Then increase it to 5
- Execute step 3 again

The counters generated in step 2) must to be preserved even when no API application servers exist

```
$ ./setup_api.sh 0

$ curl http://<nginx-ip>/
503 Service Unavailable

$ ./setup_api.sh 5
$ for x in `seq 1 100`; do curl -s http://<nginx-ip>/ ; done | sort | uniq
host1

host2
host3
host4
host5

$ ./list_counter.sh | wc -l
100
```

Things to ask when testing

After execute `setup_api.sh` , please wait until all counter services are ready.
You can check this http://localhost:8761 (http://localhost:8761).

If you do not have enough resources in your test environment to increase the number of instances, jvm will temporarily consume a lot of cpu and memory resources at boot time, causing the system to hang.
Thank you for your understanding in grading.

Explain

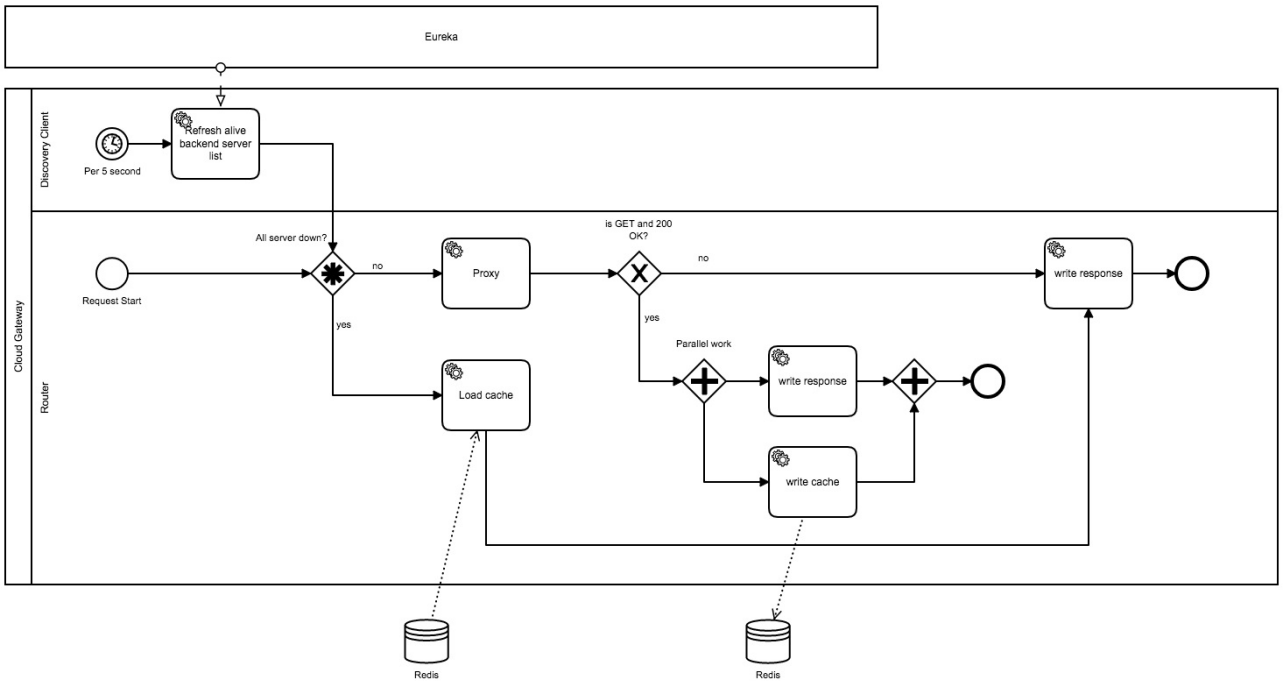
Q. Reduce the number of API application servers to 0, Then increase it to 5.

There is nothing special, and it is the same as the explanation of Step1 (Step1.md).

Q. The counters generated in step 2) must to be preserved even when no API application servers exist.

This is a question about understanding the concept of api-gateway response cache.

Flow



1. Per 5 seconds, gateway refresh alive counter-services list from eureka.
2. When request started, check if all server is down.
3. If yes, gateway load cache from redis.
4. If no, gateway proxy to counter-service
5. After proxy, if 200 ok
6. Sends a message immediately to the Kafka for each stream data.
7. All microservices receive kafka message and actual operation code is executed.

Code

CacheEntity.java

```
@Data
public class CacheEntity {

    private String content;

    private Map<String,String> headers;
}
```

CacheService.java

```
@Service
public class CacheService {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    private ValueOperations valueOperations;

    @PostConstruct
    private void init() {
        valueOperations = redisTemplate.opsForValue();
    }

    public void save(String uri, byte[] content, Map headers) { ❶
        CacheEntity entity = new CacheEntity();
        entity.setContent(new String(content, Charset.forName("UTF-8")));
        entity.setHeaders(headers);

        try {
            String s = new ObjectMapper().writeValueAsString(entity);
            valueOperations.set(uri, s);
        } catch (Exception ex) {

        }

    }

    public CacheEntity load(String uri) { ❷
        try {
            String s = (String) valueOperations.get(uri);
            if (s != null) {
                return new ObjectMapper().readValue(s, CacheEntity.class);
            } else {
                return null;
            }
        } catch (Exception ex) {
            return null;
        }
    }
}
```

❶ Save cache with url key, body, headers

❷ Load cache via url key.

CacheGlobalFilter.java

```
@Component
public class CacheGlobalFilter implements GlobalFilter, Ordered {

    @Autowired
    DiscoveryClient discoveryClient; ❶
```

```
    @Autowired
    CacheService cacheService; ❷

    @Override
    public int getOrder() {
        return -2; // -1 is response write filter, must be called before that
    }

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //cache key
        String requestUri = exchange.getRequest().getURI().toString();
        String path = exchange.getRequest().getPath().toString();

        //All Get Method using cache logic.
        if (exchange.getRequest().getMethod().equals(HttpMethod.GET)) {
            ServerHttpResponse originalResponse = exchange.getResponse();
            DataBufferFactory bufferFactory = originalResponse.bufferFactory();

            //For certain cases, be sure registered backend servers is 0 in
Eureka.
            List<ServiceInstance> instances =
discoveryClient.getInstances("counter-service"); ❶
            if (instances.isEmpty()) {

                //load cache
                CacheEntity entity = cacheService.load(requestUri); ❸
                if (entity != null) {
                    ServerHttpResponseDecorator decoratedResponse = new
ServerHttpResponseDecorator(originalResponse);

                    //fill body
                    DataBuffer buffer =
exchange.getResponse().bufferFactory().wrap(entity.getContent().getBytes(Charse
t.forName("UTF-8")));

                    //fill headers
                    for (Map.Entry<String, String> entry :
entity.getHeaders().entrySet()) {
                        decoratedResponse.getHeaders().add(entry.getKey(),
entry.getValue());
                    }
                    exchange.getResponse().setStatusCode(HttpStatus.OK);

                    //return stream
                    return decoratedResponse.writeWith(Flux.just(buffer)); ❹
                } else {
                    return chain.filter(exchange);
                }
            } else {
```

```
                ServerHttpResponseDecorator decoratedResponse = new
ServerHttpResponseDecorator(originalResponse) {

                    @Override
                    public Mono<Void> writeWith(Publisher<? extends DataBuffer>
body) {

                        //if 200 status, save cache
                        if (this.getStatusCode().equals(HttpStatus.OK) &&
path.length() > 1) { ❺
                            if (body instanceof Flux) {

                                Flux<? extends DataBuffer> fluxBody = (Flux<?
extends DataBuffer>) body;

                                return super.writeWith(fluxBody.map(dataBuffer
-> { ❻
                                    // probably should reuse buffers
                                    byte[] content = new
byte[dataBuffer.readableByteCount()];
                                    dataBuffer.read(content);

                                    saveCache(requestUri, content,
this.getHeaders().toSingleValueMap());
                                    //cacheService.save(requestUri, content,
this.getHeaders().toSingleValueMap());

                                    //return non blocking stream
                                    return bufferFactory.wrap(content);
                                }));
                            }
                            return super.writeWith(body); // if body is not a
flux. never got there.
                        } else {
                            return super.writeWith(body);
                        }
                    }
                };
                //return chain.filter(exchange);
                return
chain.filter(exchange.mutate().response(decoratedResponse).build()); // replace
response with decorator
            }
        } else {

            //Normal response
            return chain.filter(exchange);
        }
    }

    @Async
    public void saveCache(String requestUri, byte[] content, Map headers) {
```

```
            cacheService.save(requestUri, content, headers);
        }
    }
```

❶ EurekaClient. Check is instances all down.

❷ Redis cache service

❸ If all down, load cache and

❹ write cache as stream

❺ If proxy response is 200 OK and GET method,

❻ The proxy response and the read cache size will operate in parallel using the same byte buffer.

Step5

Assignment

Implement an API call to delete the counter, and delete all existing counters as shown below:

```
$ ./create_couter.sh

$ for x in $(curl -s http://<nginx-ip>/counter/); do curl -X POST
http://<nginx-ip>/counter/${x}/stop/ ; done

$ ./list_counter.sh | wc -l 0
```

Explain

There is nothing special, and it is the same as the explanation of Step2 (Step2.md).

Step6

Assignment

Create counters again with create_counter.sh as in step 2C) and implement a wait_counter.sh script that will wait for all counters to expire.
(wait_counter.sh logic does not need to be implemented in shell, it can call any language of choice)

```
$ ./create_counter.sh

$ ./wait_counter.sh

$ ./list_counter.sh | wc -l 0
```

Explain

There is nothing special, and it is the same as the explanation of Step2 (Step2.md).

Simple shell script for await rest api

wait_counter.sh

```
#!/bin/bash

NGINX_HOST=http://localhost:80

#------------------------------------------------------------------------
# 카운터 종료 대기

echo "Waiting counter complete....."

# 최대 1 시간 동안 기다리기. 1200 * 3s = 1200s = 60min
MAX_COUNT=1200
INTERVAL=3
CURRENT_COUNT=0
ERR_COUNT=0

while true
do
  COUNTS="$(curl --request GET \
              -H 'content-type: text/plain' \
              ${NGINX_HOST}/counter/counts-all)"

  HTTP_STATUS="$(curl --request GET \
                -s -o /dev/null -w "%{http_code}" \
                -H 'content-type: text/plain' \
                ${NGINX_HOST}/counter/counts-all)"
```

```
  if [ $HTTP_STATUS -eq 200 ];then

     echo "que $CURRENT_COUNT : $COUNTS counters are running"

     #------------------------------------------------------------------------
     # 카운터가 모두 종료되었을 경우

     if [ $COUNTS -eq 0 ];then
        echo "All counter  completed!!"
        break
     fi

     #------------------------------------------------------------------------
     # 호출 카운트 증가

     CURRENT_COUNT=$((CURRENT_COUNT + 1))
     ERR_COUNT=0

     #------------------------------------------------------------------------
     # 타임아웃이 걸렸을 경우

     if [ "$CURRENT_COUNT" -gt "$MAX_COUNT" ];then
        echo "Time out. Restart script to continue counter monitoring.";
        exit 1
     fi
     sleep 3

  else
     echo "Failed to get counters."

     #------------------------------------------------------------------------
     # 에러 카운트 증가

     ERR_COUNT=$((ERR_COUNT + 1))

     #------------------------------------------------------------------------
     # 에러가 30초 이상 지속시 종료

     if [ $ERR_COUNT -gt 10 ];then
        echo "Failed to connecting ngnix. pleas check out your ngnix network
environment."
        exit 1
     fi

     sleep 3

  fi
done
```