Microservices:

Intro:

Small autonomous service that works together.

Challenge:

Bounded context: Define the boundary of MS

Configuration management: for 100s of environment

Dynamic scale up/down:

Visibility: Centralized log and monitoring tool to manage 100s of ms server.

Fault tolerance: If any fundamental service is down then might impact all the other services.

Spring Cloud (Netflix):

Configuration management Spring cloud config server:

Dynamic scale/down: Ribbon Load balancing

Naming server (Eureka(service registry/discovery))

FEIGN

2.Visibilty and monitoring

Zipkin and Netflix zull api gateway (auth,rate limit etc)

3.Fault tolerance

Hystrix

V2(Spring cloud):

Spring cloud load balancer vs ribbon

Spring cloud gateway vs zuul

Resilience4j vs hystrix circuit breaker

https://github.com/in28minutes/spring-microservices-v3/tree/main/03.microservices/01-step-by-step-changes#step-10

**Spring Cloud:**

1. Spring Cloud config Server:

WHY:

As we know, in MS we will be having n no. of microservices, and each service can have different no. of instances in this it is difficult to manage the configuration properties. Usually, we use to define in properties file in keep in resources folder. But in the case of many services, it is difficult to manage.

Solution:

We should have defined that at central places and each MS should be able to get it from that central place in this case there will not be any discrepancies across services and update change will be easier instead updating in each service will be updating at one place.

How: Spring cloud gives feature implement this easily.

We can create config server with below steps:

a).That will dependency

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

b) @EnableConfigServer add this to springbootapplication trigger file

c)Create git repository where will have all the config files. We can define file for each profile also like dev,qa,etc.

d) Add to application. Prop

spring.cloud.config.server.git.uri=file:///D:/Learning/github/MS/git-localconfig-repo

Now we can see that file content using ulr.

Ex: http://localhost:8888/limits-service/default and

for env <http://localhost:8888/limits-service/qa>

Consume client side with below steps:

1. Add below dependency

##<dependency>

# <groupId>org.springframework.cloud</groupId>

# <artifactId>spring-cloud-starter-config</artifactId>

# </dependency>

1. Add to app.prop

spring.application.name=order-service

#above name will be used to pick properties file from below git repo ex: order-service.prop

spring.config.import=optional:configserver:http://localhost:8888(config server url)

spring.cloud.config.profile=qa

1. Create a class:

@Component

@ConfigurationProperties("order-service")//in git repo will have file with this name only

**public** **class** Configuration {//define the prop similar to prop file to map}

Inject and use the above object to get the property value.

1. Naming server:

WHY: As in MS we need to communicate with diff services, for that we need to configure the location of those services. And also for any services or any third party services if they change that address ip/dname in that our services also need to be updated.

Solution: There should be some services who can store the address of all services and when any one need that they can interact and get the details from there.So basically there should be two part one who will handle service registry and another with we can discover the services.

How: Netflix eureka provides service registry and discovery.

We can create one Naming server to register that service as:

1. Add

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

2) @EnableEurekaServer

@SpringBootApplication

**public** **class** NamingServerApplication {}

//Naming server will be up with above

Client side changes

1. ADD

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

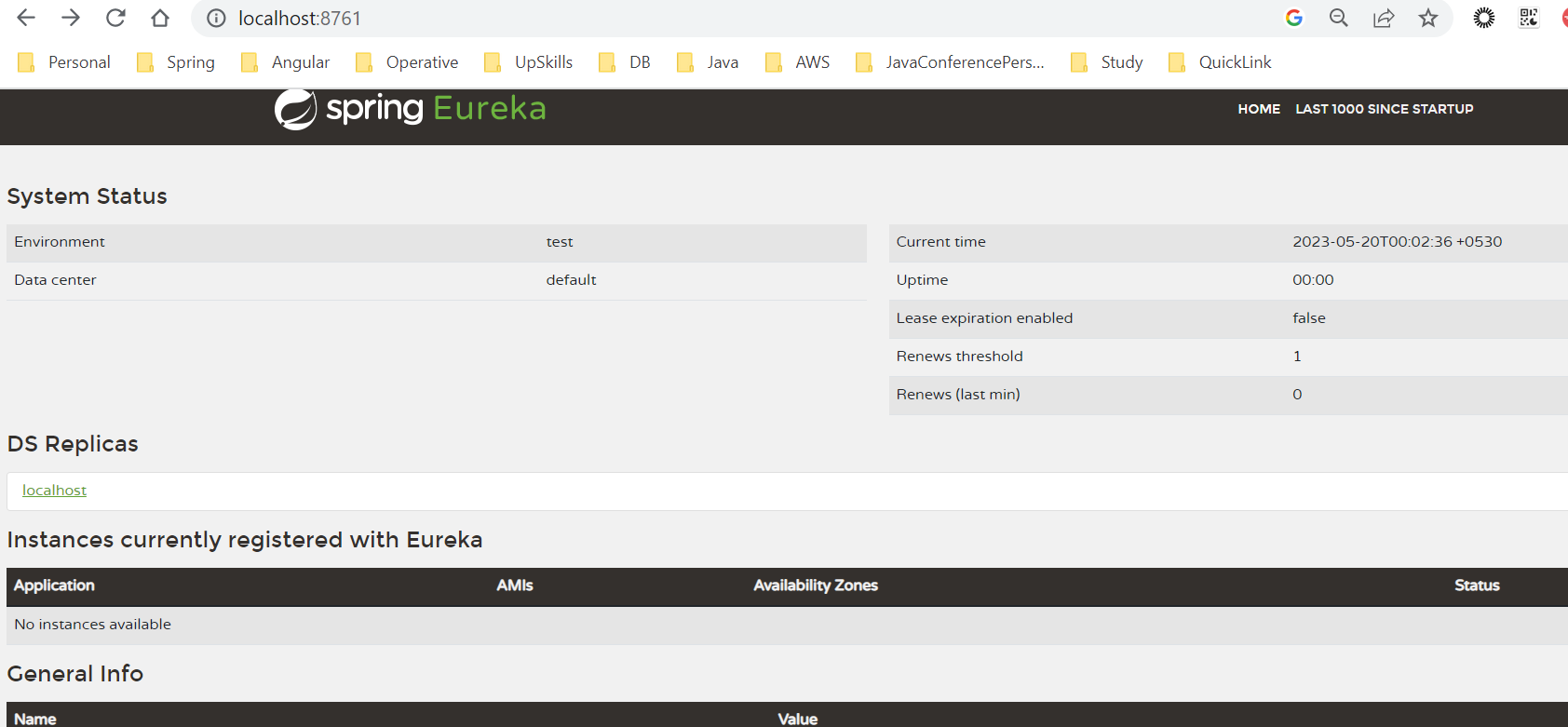
</dependency>

2)Add to app.prop

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

Define address of naming server

3)After adding above when we start app it will register the given name(spring.application.name) and ip with eureka server



4)We can get any service details from eureka by giving name. So no need to change in source code if any other services has change their address. Instead that will get updated to Eureka will get from there.

3.Client-side load balancing:

WHY: AS in MS mostly we need to interact with diff services and there might be dependent service would be running two or more instances in that case it is better to distribute the load on both instances. That we call load balancing here we r talking to handle this at client side it can be server side too.

Solution: When we call api we can switch to both the address.

HOW: FEIGN Client provide this inbuilt capability to switch.

1)ADD

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

1. @EnableFeignClients to SpringBootAPP

3)NOW we can define proxy class with

@FeignClient(name="restaurant", url="localhost:8000")//if one instance it will have final url localhost:8000/restaurant

@FeignClient(name="resturant") //same name registered with Eureka

public interface RestaurantProxy {

@GetMapping("/resturants")

public Restaurant restaurant() {

}

Restaurant restaurant = proxy.restaurant();

Here it will get the Eureka registered value with Naming Server and form URL accordingly. AND if two or more registered with same name the it will distribute the load automatically while calling the api.

It became easily managed else we would have configured and managed the while calling api.

4.API Gateway:

WHY: As in MS will have more services deployed independently. In that there will be some tasks that is common across all the services or might be need to track all the services.

Single point of entry, Unique protocol for all clients, Clients don’t need to know about multiple server addresses, Unique response for multiple clients, Changes on backend services can be hidden from clients, Multiple platforms, Services are not exposed, Load balancing, Caching,

Monitoring

EX: Authentication, routing, rate limiting, billing, monitoring, analytics, policies, alerts, and security, etc.

So instead of defining each service we can define that in one place and route the services to actual one.

Solution: Need a system that will intercept all the request or all request so go thru this and do the needful after rerouting that.

Implementation: Spring cloud provides implementation for this.

1. Add

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

2)And here as we need the services details also to re-route , we need to add eureka client dependency also.

3) We need to define Configuration file as per our need

Ex:

@Configuration

public class ApiGatewayConfiguration {

@Bean

public RouteLocator gatewayRouter(RouteLocatorBuilder builder) {

return builder.routes()

.route(p -> p

.path("/get")

.filters(f ->

f.addRequestHeader("MyHeader","MyURI")

.addRequestParameter("Param", "MyValue")) .uri("http://httpbin.org:80"))

}

As per above config all req with /get, will be added header and reroute to given.

Similarly, we can define Rate Limiter and so on.

Cloud also provides these services: AWS with easily configurable refer AWS notes.

\*\*\*\*\*Enable Discovery Locator\*\*\*\*

In real time scenario we will be having gateway URL as client interaction.

Through this only will be accessing all the other services. Ideally also for any application we will be having a single URL else user won’t go and try diff MS URL.

Solution: User should hit APIGateway URL by providing the MS name and api gateway should hit that url internally/redirect to that.

EX:

Say we have MS with URL: "http://localhost:8000/orderIds

And this MS is registered with Eureka with name: orders

And API Gateway: url is: "http://localhost:8765

Then to access order service we can hit as : <http://localhost:8765/orders/orderIds>

HOW?

Spring cloud api gateway provides discovery locator feature if we enable this then it will auto get service details from eureka using given name from URL

Option 1:

#spring.cloud.gateway.discovery.locator.enabled=true

#spring.cloud.gateway.discovery.locator.lower-case-service-id=true

Option: 2:

We can have configuration class also to write the custom logic based on url

https://haris-zujo.medium.com/spring-cloud-gateway-request-filtering-and-redirection-9e4b6d559d1a

5.Circuit Breaker:

WHY: As we have discussed already in MS arch. we have service that is running independently but we need to always interact with each other. In that case when any other dependent services are down then that api call will fail and throw exception. Ideally it should not happen, failure of one component doesn't mean the other parts will stop working. We should be handling it in such a way system should not throw and running smoothly with fallback data.

Solution: We can have such pattern who track the api call and manage it.

Ex: It can retry to such defined number and still fail then it should return the fallback mesgs given by dev.

And also, it should be able to decide when to avoid calling if already known that services is down and keep try checking after some time to see if that is available with less cost (in terms of retry api call).

Can be designed with below concepts:

* We initialize the Circuit Breaker object with certain parameters: timeout, failureThreshold and retryTimePeriod which help determine how resilient the API is.
* Initially, we are in the closed state and no’s remote calls to the API have occurred.
* Every time the call succeeds, we reset the state to as it was in the beginning.
* If the number of failures cross a certain threshold, we move to the open state, which acts just like an open circuit and prevents remote service calls from being made, thus saving resources. (Here, we return the response called stale response from API)
* Once we exceed the retry timeout period, we move to the half-open state and make another call to the remote service again to check if the service is working so that we can serve fresh content. A failure sets it back to open state and another attempt is made after retry timeout period, while a success sets it to closed state so that everything starts working normally again.

https://java-design-patterns.com/patterns/circuit-breaker/#programmatic-example

HOW: Spring Cloud provides this capability using Resilience4j.

We can use below annotation as per need, where we call api that need to be handle with fallback.

//@Retry(name = "sample-api", fallbackMethod = "hardcodedResponse")

@CircuitBreaker(name = "default", fallbackMethod = "hardcodedResponse")

//@RateLimiter(name="default") within certain time period how many call allowed

@Retry(name="default") //default 3 times retry

@Bulkhead(name="sample-api") //no. of concurrent call

#circuit breaker https://resilience4j.readme.io/docs/circuitbreaker hit multi time watch curl http://localhost:8000/sample-api

resilience4j.retry.instances.sample-api.maxAttempts=5

#resilience4j.retry.instances.sample-api.maxRetryAttempts=5 #OLD

resilience4j.retry.instances.sample-api.waitDuration=1s

resilience4j.retry.instances.sample-api.enableExponentialBackoff=true

#resilience4j.circuitbreaker.instances.default.failureRateThreshold=90

#ratelimit 2 req per 10 sec

resilience4j.ratelimiter.instances.default.limitForPeriod=2

resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s

#bulkhead no. of concurrent call

resilience4j.bulkhead.instances.default.maxConcurrentCalls=10

Resilience:

A screenshot of a computer program

Description automatically generated with low confidence

Yaml sample

A screen shot of a computer

Description automatically generated with medium confidence

A picture containing text, font, screenshot

Description automatically generated

Aspect-order will define which is has more priority, by default circuit breaker has more than retry

**6.Distributed Tracing:**

**WHY:**

With microservice one more problem is to be tracing a request. Since one request might go through diff MS in that case it is difficult to trace the request across Services/multi-threaded env.

In monolith we use to get the thread name and search that, meaning all log with this thread represent the same request for that time span.

But in MS it will go through different thread in different services to tracing with thread name will not be helpful.

Solution:

Let’s consider if we can generate one ID and pass it across services for each request. Meaning we will generate one traceId that will flow with the request across service till the end of that request. In that case we can use that traceId to find the log for request.

Now another issue will be as it will go thru diff MS and all will have the same ID then how to differentiate the log from MS specific, I mean how will know which log is from what MS belongs to.

Solution:

We can have one more span-id that will be created every time when req flows to new MS and it will be always associated with parent trace Id.

So finally, we will have [traceId (scope request), spanId (for MS)]

And, within request in same MS we need to segregate some part of process to track the log for that part, we can create new span for that process. In that case we can easily identify the part of request process.

How: Spring provides library to generate the traceId and spanId related feature called **sleuth.**

Spring–sleuth has Tracer to manage the traceId and append this with log level(internal).

Ex: [Service Name, TraceId, SpanId]

2023-05-28 12:48:45.943 INFO **[sleuth-zipkin,b8014fd9cadc055c,b8014fd9cadc055c]** 38356 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'

\*\* To enable this, we have to add dependency as:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

This will enable the sleuth feature. It will add traceID and spanId to log.

\*\*\* With upgraded version of sleuth, it has inbuilt **brave** feature too.

“**brave**”: Opensource lib to achieve the same managing Tracer like Sleuth. It has some additional features too. So now both merged and provided together.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-brave</artifactId>

</dependency>

Q. How this is handled with async and multi-threaded cases.

In case of multi-threaded we need to customize impl to pass the tracing info.

Ex: @Bean

**public** Executor executor() {

ThreadPoolTaskExecutor threadPoolTaskExecutor

= **new** ThreadPoolTaskExecutor();

threadPoolTaskExecutor.setCorePoolSize(1);

threadPoolTaskExecutor.setMaxPoolSize(1);

threadPoolTaskExecutor.initialize();

**return** **new** LazyTraceExecutor(beanFactory, threadPoolTaskExecutor);

}

the use of LazyTraceExecutor. This class comes from the Sleuth library and is a special kind of executor that will propagate our traceID to new threads and create new spanIds in the process.

For more details to Handle Async/schedule, etc. see below post:

<https://www.baeldung.com/spring-cloud-sleuth-single-application>

Q. How sleuth propagates the info for HTTP call to other services?

This uses header to propagate the context across services. We can customize this also.

Q. How sleuth add details to logger?

This overrides the logging property.

For details refer below

<https://stackoverflow.com/questions/65846859/how-did-spring-cloud-sleuth-add-tracing-information-to-logback-log-lines>

Terminology:

[application name, traceId, spanId, export]

Application name – This is the name we set in the properties file and can be used to aggregate logs from multiple instances of the same application.

TraceId – This is an id that is assigned to a single request, job, or action. Something like each unique user initiated web request will have its own traceId.

SpanId – Tracks a unit of work. Think of a request that consists of multiple steps. Each step could have its own spanId and be tracked individually. By default, any application flow will start with same TraceId and SpanId.

Export – This property is a Boolean that indicates whether or not this log was exported to an aggregator like Zipkin. Zipkin is beyond the scope of this article but plays an important role in analyzing logs created by Sleuth.

**\*\*\*\*\* Notes\*\*\*\*\*:**

spring-cloud-starter-zipkin is deprecated, you should not use it anymore. You can use spring-cloud-starter-sleuth and spring-cloud-sleuth-zipkin (3.x).

If you check the dependencies of spring-cloud-starter-zipkin you will see that it depends on spring-cloud-starter-sleuth and spring-cloud-sleuth-zipkin so it is pulling in Sleuth and Sleuth's Zipkin support (which pulls in Brave).

From the high-level point of view Sleuth is doing three things:

It provides an API abstraction for Tracing libraries (Brave is the default library under the hood, OTel is incubating, and you can implement your own tracing library integration)

Instruments other Spring Projects

Integrates with other components (e.g.: log correlation, Tomcat access log support, etc.)

Please see the docs: https://docs.spring.io/spring-cloud-sleuth/docs/current/reference/htmlsingle/

<https://docs.spring.io/spring-cloud-sleuth/docs/current/reference/htmlsingle/spring-cloud-sleuth.html>

spring.sleuth.async.enable = false

**7.Zipkin**

WHY: We discussed why we need sleuth. It is next step of that.

After sleuth we have log where each request has traceId and spanId that will represent part of that req processing.

Now to trace req we have to search that traceId to find out all related logs. It will be difficult find and club manually.

Solution: We can have a tool that we club all logs with same traceID and spanid show together.

How: Zipkin provides that capability it will get the log from sleuth and show it in an easily understandable UI.

Install Step:

Go to: https://zipkin.io/pages/quickstart

A screenshot of a computer

Description automatically generated with low confidence

OR get the docker image and run

OR:

Download jar

from <https://repo1.maven.org/maven2/io/zipkin/java/zipkin-server/2.12.9/>

go to jar dir.:

run java -jar zipkin-server-2.12.9-exec.jar

After that go to port 9411

Here will see nice UI that has diff filter ex: serviceName, spanId, etc.

A screenshot of a computer

Description automatically generated with medium confidence

Zipkin: SB2.x sleuth->brave ->Zipkin

SB3.x micrometer->open telemetry->Zipkin

Spring boot 3 has replaced sleuth with micrometer.

<https://github.com/in28minutes/spring-microservices-v3/blob/main/v3-upgrade.md>

spring.sleuth.enabled=true

spring.sleuth.sampler.probability=1

spring.zipkin.enabled= true

spring.zipkin.base-url= http://localhost:9411

spring.sleuth.async.enabled= false

spring.sleuth.async.configurer.enabled= false

Distributed tracing:

**Zipkin server setup:**

It helps gather timing data needed to troubleshoot latency problems in microservice architectures. Has a management console that helps to visualize time statistics generated by various microservices. It supports various back-ends where the span details can be saved. Currently it stores the span details in-memory by default but can be configured to store in Elastic search.

WHY:

Distributed tracing systems provide detailed information about subsystem latency. It uses Zipkin. It supports retrieving Span information from various storage systems. **Currently it supports In-memory and Elastic-search.**

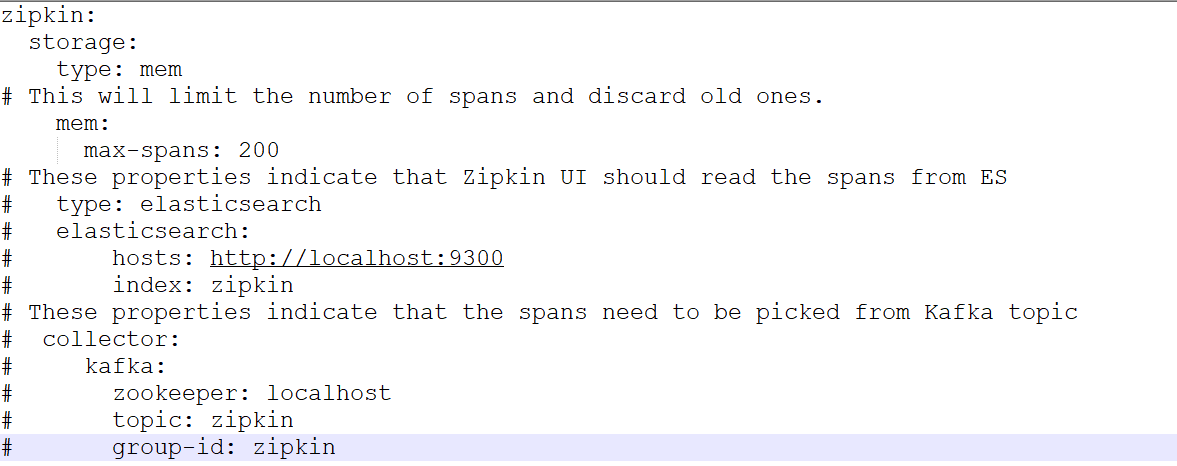
As we understood sleuth will be managing trace and spanid. That needs to be store so that zipkin UI-can fetch the details from there and show.

To store this, we have option of in-memory and ES.

Receiving Spans via Http is not efficient as that would make Zipkin single point of failure. So scalable option would be to read spans from Kafka and persist those in ES.

Provision is provided to read Span details from Kafka and push to ES based on the configuration. Also, support has been added to receive spans via Http but store them in Elastic search without Kafka.

To enable Kafka some useful properties:



<https://dev.to/dj_kev/how-to-set-up-distributed-tracing-in-microservices-with-spring-boot-zipkin-or-the-elk-stack-444a>

**Spring Boot Admin:**

Spring boot admin is used to manage and monitor Spring boot applications. It internally uses Spring Actuator. Actuator helps to monitor and manage the application.

**WHY:**

**In MS we will be having many microservices to track/monitor each of them separately will be tedious. We need a single dashboard where we should be able to monitor each service.**

**Solution:**

**We can build a service that will collect info from each service and show at single dashboard.**

**To achieve that we can get the services details from either Eureka or client itself will register with admin server. And to get the application info we can get the help of actuator.**

**Each client application needs to have Spring Actuator jars in it. The endpoints provided by the Actuator jar is polled by the Spring Boot Admin server to get the metrics of that particular application.**

**HOW:**

**Spring Boot admin is a community project use to manage and monitor your Spring Boot applications. The client application gets register themselves with the admin server (via Http) or is discovered using Spring Cloud discover server like Eureka, Consul.**  
To provide a unified Dashboard View of all Microservices statistics. Auditing, health and metrics gathering can be automatically applied to the application. All microservices that are registered with Eureka, Service Discovery would become clients of Spring boot admin automatically.

There are 2 ways:

1. Add dependency ‘spring-boot-admin-starter-client ‘to client and specify the admin-server url

#spring boot admin

spring.boot.admin.client.url=http://localhost:8080

management.endpoints.web.exposure.include=\*

management.endpoint.health.show-details=always

2.Using naming server

If you have a Spring Cloud Discovery (like Eureka) for your application, you don’t need to have Spring Boot Admin Client jar in each of your client application

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.client.register-with-eureka=false

Spring Boor Admin Server will get all client application details from Eureka and poll them for metrics.

Reference: See admin-server project and their readme file

https://www.javadevjournal.com/spring-boot/spring-boot-admin/

https://www.baeldung.com/spring-boot-admin

Use of actuator?

Actuators endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own

https://www.javadevjournal.com/spring-boot/spring-boot-actuator/

How to get namigserver details from eureka client