**What is Eureka Server**

1. The Eureka server is nothing but an implementation of service discovery pattern.
2. Microservice register themselves so other can discover.
3. Its two thing one **registry** and another **discovery** means microservice which register in Eureka server, so it has all metadata about microservice like name, URL, port number,

Status about micro-service, discovery means any micro-service wants to communicate to other microservice then Eureka server will help to give the information about that microservice,

This server holds information about the client service applications. Each microservice registers into Eureka server and eureka server knows all client applications running on each port and IP address. This server is also known as discovery server.

The **@EnableEurekaServer** annotation is used to make your Spring Boot application acts as a Eureka Server.

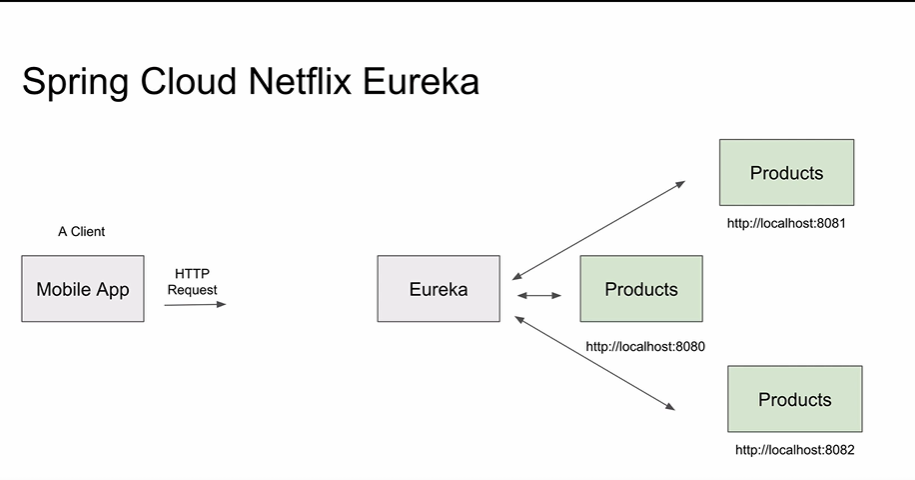
The Eureka Server is a Netflix OSS product, and Spring Cloud offers a declarative way to register and invoke services by Java annotation.

**Why do we need Eureka Server?**

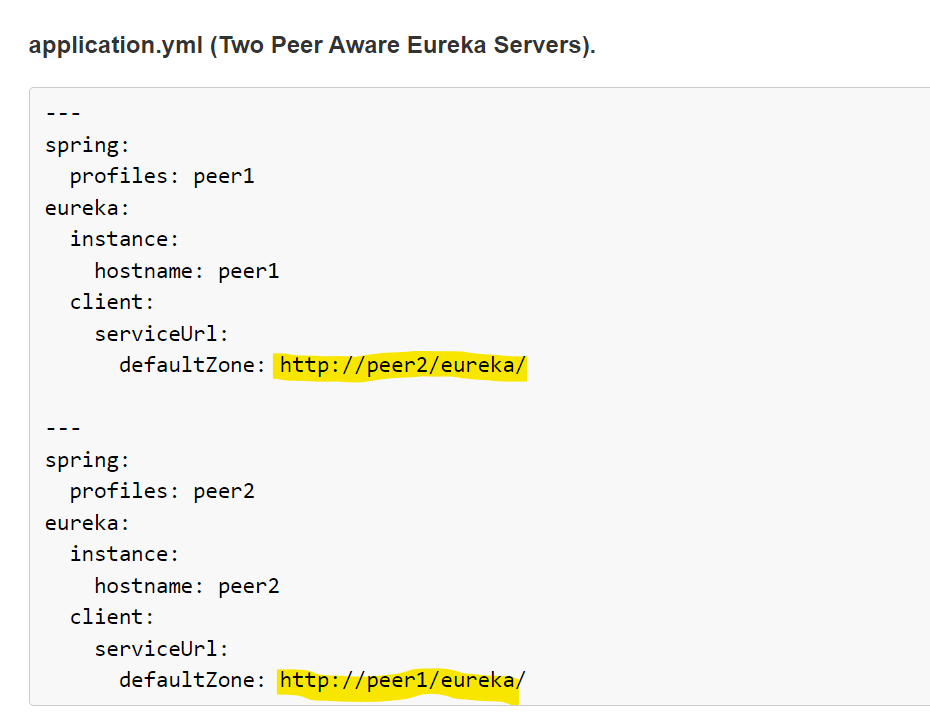
Suppose we have multiple instances of microservice that is running in different IP address/port number now one microservice want to call another one so which instance it will call?

Because its need to have URL of that API that static then it won’t communicate to other instance so this condition Eureka server will help the API to communicate to another service

Dynamically not need worry about other service URL, port, IP address.



**Multiple instance configuration for Eureka Server.**



More details

<https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-eureka-server.html>

**in depth eureka server**

<https://www.aegisinfoways.com/articles/spring-cloud-eureka-service-discovery-implementation.html>

Graphical user interface, text, application

Description automatically generated

Need to create each instance properties files and set run time

--spring.config.location=./ext/eureka-instance-1.properties

<https://medium.com/become-developer/how-to-work-with-multiple-instances-of-eureka-naming-server-to-avoid-a-single-point-of-failure-d953544281d0>

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**Configuring your microservice to work with multiple Eureka Servers**

Add the following line to your **application.properties** of your microservice:eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka, <http://localhost2:8762/eureka>

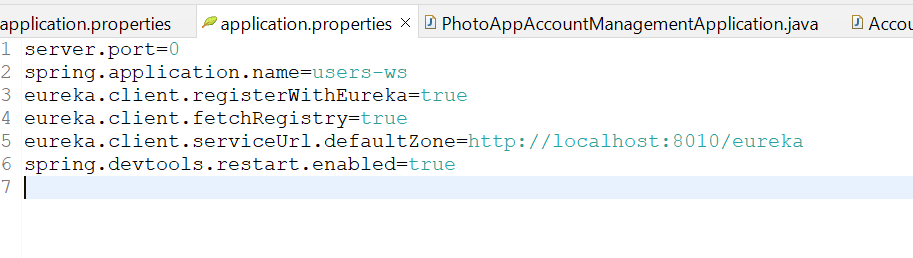
**Eureka Client**

Eureka client application need to register with eureka server then we have to do below setup.

@EnableDiscoverClient

**Application.properties** register with eureka server.

eureka.client.serviceUrl.defaultZone=http://localhost:8010/eureka



**Dependency**

Graphical user interface, text, application

Description automatically generated

**Secure Connection with Eureka Server.**

Graphical user interface, text, application, email

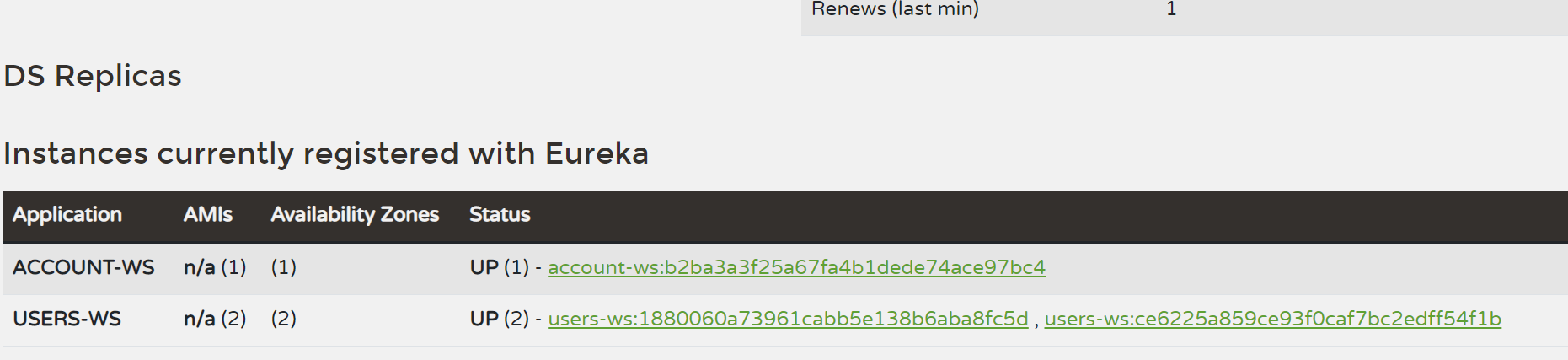
Description automatically generated

**Multiple instances start.**

If we start multiple instances of application, then we need to create unique instance id otherwise it will not register in eureka server.

add below configuration in application. Properties.

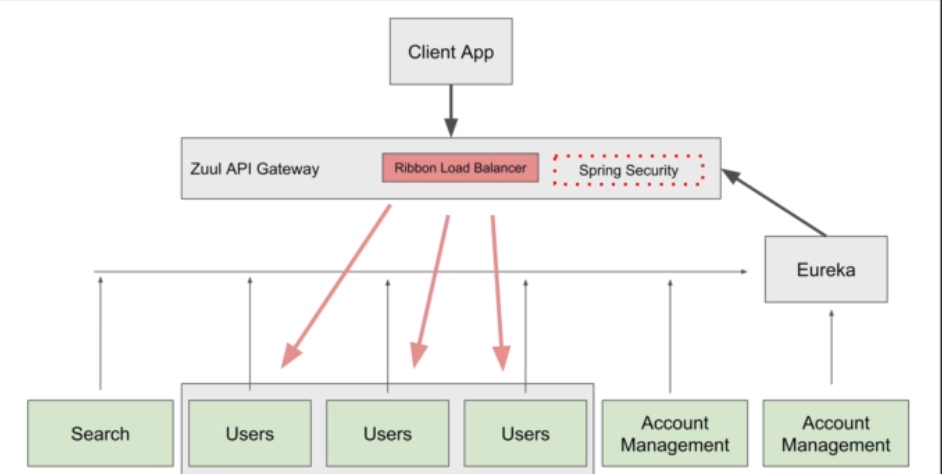
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance-id:${random.value}}



**API GATEWAY**

ZUUL

Its router + server-side load balancer (it will balance request for multiple instances running)



**Different between Zuul API Gateway and Spring Cloud Gateway.**

**Origin:** Zuul has been developed by Netflix OSS as open-source project, very earlier with Servlet 2.5 support as Zuul 1.

. Spring Cloud Gateway has been developed by Spring team and inspired by Zuul based on Spring Web Flux which supports reactive programming from beginning.

**Filter & Predicate:** Both supports rich set of filters and predicates. However, there are some filters like Rate Limiter yet to be delivered by Zuul 2 as open source.

**Performance**: Based on comparison and result, Zuul has respectively higher performance. Refer link.

**Spring Adaptation:** Since Spring Cloud itself providing a gateway, Zuul has been removed from Spring Gateway component list. Now start.spring.io is listing Spring Cloud Gateway as gateway component.

**Communication technology support**: WebSocket & SSE support yet to be provided by Zuul 2. While Spring Cloud Gateway provides full-support for these.

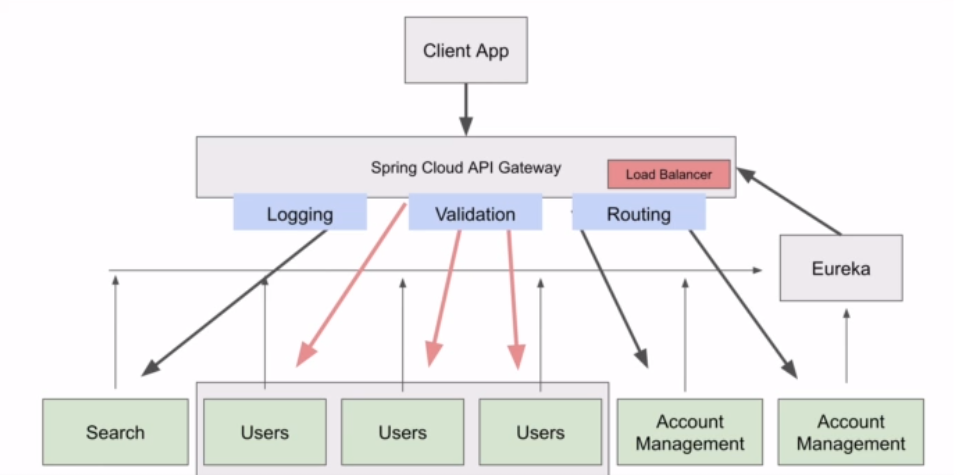
**The Spring Cloud Gateway**

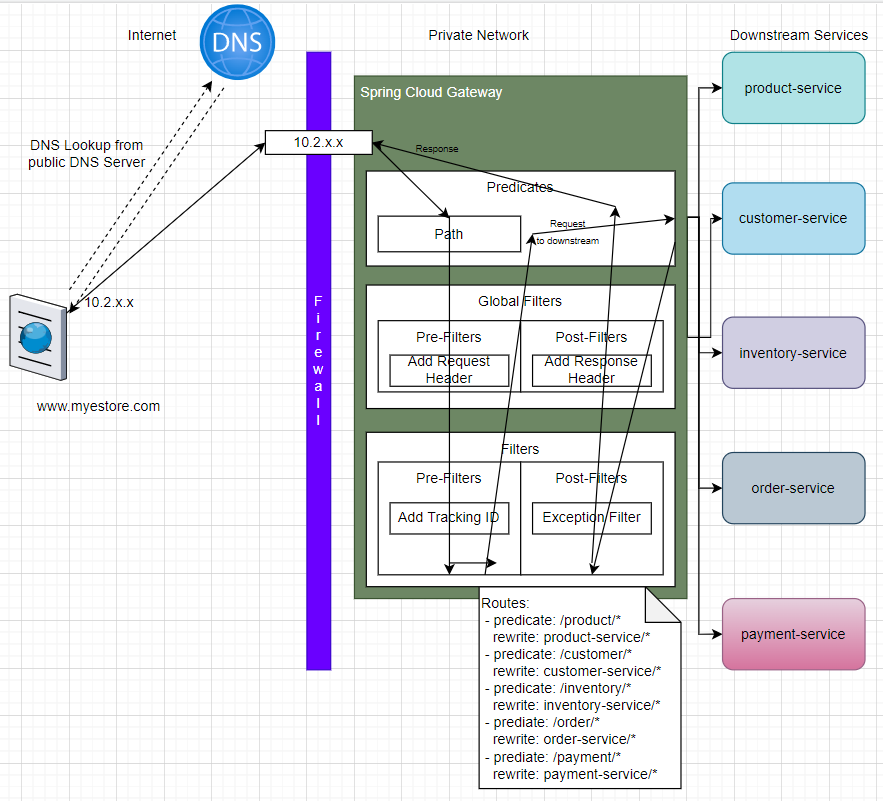
Resilience, security, and Observability

The Spring Cloud Gateway uses **Netty** as the embedded server instead of the [Tomcat server](https://www.java67.com/2019/07/spring-boot-3-ways-to-change-port-of-tomcat.html) because**Tomcat is based on the blocking model** and **Netty is based on the non-blocking model.**

**The functionality of Spring Cloud Gateway**

* Built on [Spring framework 5](https://medium.com/javarevisited/top-10-pluralsight-courses-to-learn-spring-framework-for-java-developers-3d35c4a1dc2), project reactor and [Spring Boot 2.0](https://medium.com/javarevisited/10-free-spring-boot-tutorials-and-courses-for-java-developers-53dfe084587e)
* Able to match routes on any requested attribute
* Predicates and filters are specific to routes
* Circuit Breaker integration
* Spring Cloud Discovery Client integration
* Easy to write Predicates and filters
* Request Rate Limiting
* Path rewriting
* Spring Security Integration
* Entry point
* Handling load balance

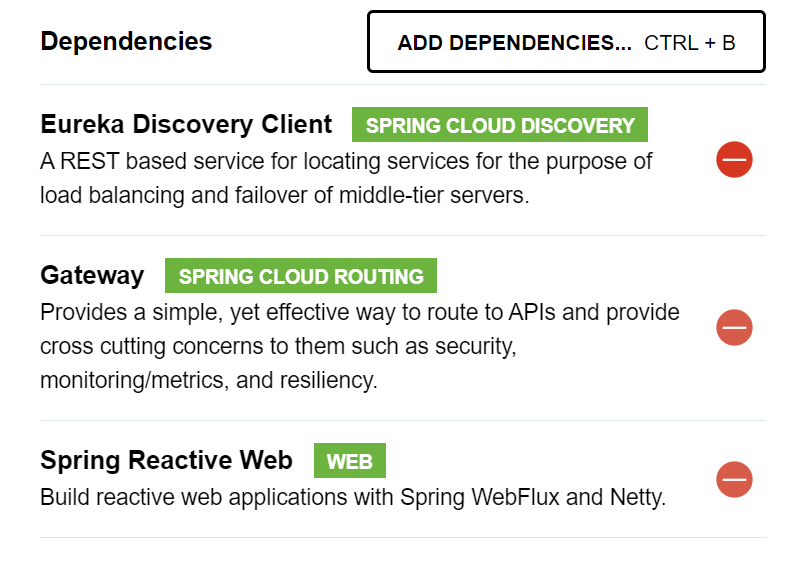




As seen from the above diagram, all the front-end clients talk to a single component and Spring Cloud Gateway takes care of routing the request to the respective microservice and gets the response to the client. In a cloud environment, **Spring Cloud Gateway runs in the Public subnet (public IP) and all other microservices run in the private subnet (private IP).**

Spring Cloud gateway is also a **discovery client** along with other microservices and hence it discovers the [microservice](https://medium.com/javarevisited/10-best-java-microservices-courses-with-spring-boot-and-spring-cloud-6d04556bdfed)from the service discovery registry and routes to those services using its private IP. By doing so, we are reducing the risk of exposing the entire microservices architecture to the public.

Spring Cloud Gateway can also handle **Authentication and Authorization.** Authentication is done by integrating [Spring Security](https://medium.com/javarevisited/top-10-courses-to-learn-spring-security-and-oauth2-with-spring-boot-for-java-developers-8f0222d6066d). Authorization of the JWT tokens can be done here at a centralized place thereby reducing the overhead for every other microservice.



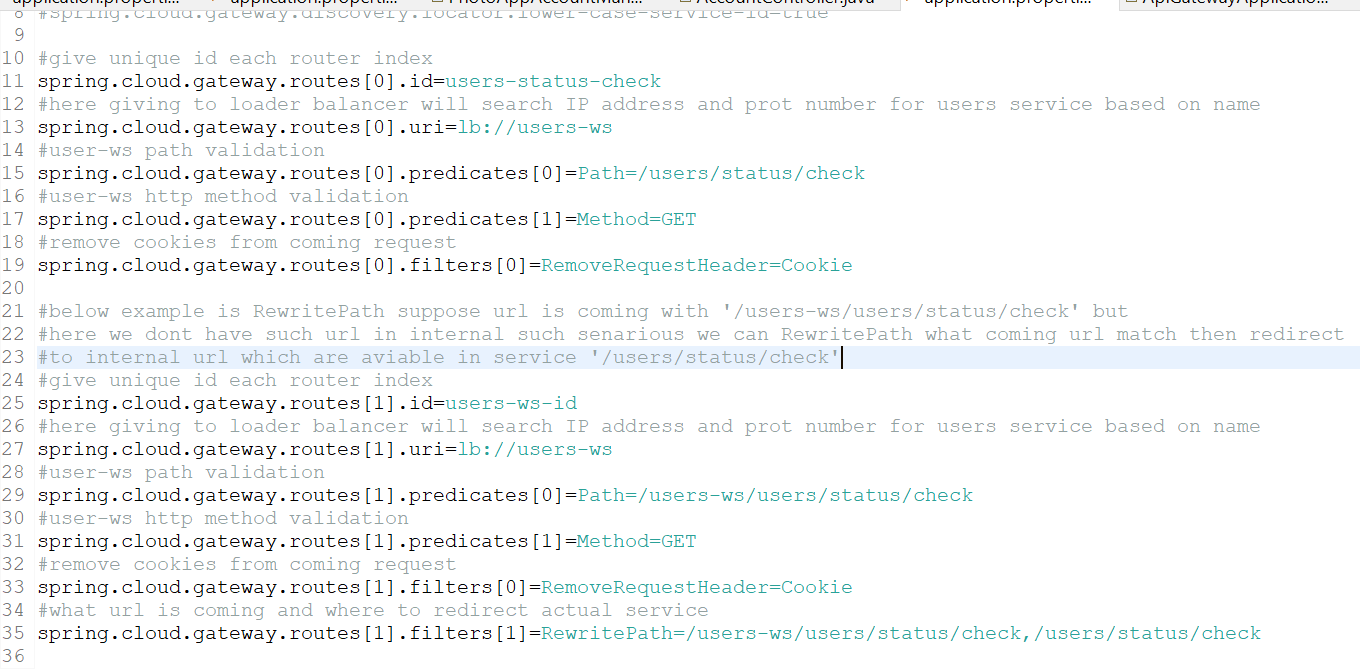
**Some Example about route, predicate, filter**

**We can access the API via API gateway with help of application-name and without application-name.**

<http://localhost:8082/users-ws/users/status/check>

<http://localhost:8082/users/status/check>

above both URL calls users service same method



#### Built-in Route Predicate Factories in Spring Cloud API Gateway

Below is a list of built-in predicates that you can use when creating routes in Spring Cloud API Gateways. These are built-in predicates and you do not need to write any additional Java code for them to work.

Note: The description of the following built-in predicates is taken from the [Spring Cloud API Gateway documentation page](https://cloud.spring.io/spring-cloud-gateway/reference/html/#gateway-request-predicates-factories).

**1. The After Route Predicate Factory**

The After route predicate factory takes one parameter, a datetime (which is a java ZonedDateTime). This predicate matches requests that happen after the specified datetime. The following example configures an after route predicate:

1. spring.cloud.gateway.routes[0].id = after\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **After**=2017-01-20T17:42:47.789-07:00[America/Denver]

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver).

**2. The Before Route Predicate Factory**

The Before route predicate factory takes one parameter, a datetime (which is a java ZonedDateTime). This predicate matches requests that happen before the specified datetime. The following example configures a before route predicate:

1. spring.cloud.gateway.routes[0].id = before\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Before**=2017-01-20T17:42:47.789-07:00[America/Denver]

This route matches any request made before Jan 20, 2017 17:42 Mountain Time (Denver).

**3. The Between Route Predicate Factory**

The Between route predicate factory takes two parameters, datetime1 and datetime2 which are java ZonedDateTimeobjects. This predicate matches requests that happen after datetime1 and before datetime2. The datetime2 parameter must be after datetime1. The following example configures a between route predicate:

1. spring.cloud.gateway.routes[0].id = between\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Between**=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver) and before Jan 21, 2017 17:42 Mountain Time (Denver). This could be useful for maintenance windows.

**4. The Cookie Route Predicate Factory**

The Cookie route predicate factory takes two parameters, the cookie name and a regexp (which is a Java regular expression). This predicate matches cookies that have the given name and whose values match the regular expression. The following example configures a cookie route predicate factory:

1. spring.cloud.gateway.routes[0].id = cookie\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Cookie**=chocolate, ch.p

This route matches requests that have a cookie named chocolate whose value matches the ch.p regular expression.

**5. The Header Route Predicate Factory**

The Header route predicate factory takes two parameters, the header name and a regexp (which is a Java regular expression). This predicate matches with a header that has the given name whose value matches the regular expression. The following example configures a header route predicate:

1. spring.cloud.gateway.routes[0].id = header\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Header**=X-Request-Id, \d+

This route matches if the request has a header named X-Request-Id whose value matches the \d+ regular expression (that is, it has a value of one or more digits).

**6. The Host Route Predicate Factory**

The Host route predicate factory takes one parameter: a list of host name patterns. The pattern is an Ant-style pattern with . as the separator. This predicates matches the Host header that matches the pattern. The following example configures a host route predicate:

1. spring.cloud.gateway.routes[0].id = host\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Host**=\*\*.somehost.org,\*\*.anotherhost.org

URI template variables (such as {sub}.myhost.org) are supported as well.

This route matches if the request has a Host header with a value of www.somehost.org or beta.somehost.org or www.anotherhost.org.

This predicate extracts the URI template variables (such as sub, defined in the preceding example) as a map of names and values and places it in the ServerWebExchange.getAttributes() with a key defined in ServerWebExchangeUtils.URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE. Those values are then available for use by [GatewayFilterfactories](https://cloud.spring.io/spring-cloud-gateway/reference/html/" \l "gateway-route-filters" \t "_blank)

**7. The Method Route Predicate Factory**

The Method Route Predicate Factory takes a methods argument which is one or more parameters: the HTTP methods to match. The following example configures a method route predicate:

1. spring.cloud.gateway.routes[0].id = method\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Method**=GET,POST

This route matches if the request method was a GET or a POST.

**8. The Path Route Predicate Factory**

The Path Route Predicate Factory takes two parameters: a list of Spring PathMatcher patterns and an optional flag called matchOptionalTrailingSeparator. The following example configures a path route predicate:

1. spring.cloud.gateway.routes[0].id = path\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Path**=/red/{segment},/blue/{segment}

This route matches if the request path was, for example: /red/1 or /red/blue or /blue/green.

This predicate extracts the URI template variables (such as segment, defined in the preceding example) as a map of names and values and places it in the ServerWebExchange.getAttributes() with a key defined in ServerWebExchangeUtils.URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE. Those values are then available for use by [GatewayFilterfactories](https://cloud.spring.io/spring-cloud-gateway/reference/html/" \l "gateway-route-filters" \t "_blank)

**9. The Query Route Predicate Factory**

The Query route predicate factory takes two parameters: a required param and an optional regexp (which is a Java regular expression). The following example configures a query route predicate:

1. spring.cloud.gateway.routes[0].id = query\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **Query**=green

The preceding route matches if the request contained a green query parameter.

**10. The RemoteAddr Route Predicate Factory**

The RemoteAddr route predicate factory takes a list (min size 1) of sources, which are CIDR-notation (IPv4 or IPv6) strings, such as 192.168.0.1/16 (where 192.168.0.1 is an IP address and 16 is a subnet mask). The following example configures a RemoteAddr route predicate:

1. spring.cloud.gateway.routes[0].id = remoteaddress\_route
2. spring.cloud.gateway.routes[0].uri = https://example.org
3. spring.cloud.gateway.routes[0].predicates[0] = **RemoteAddr**=192.168.1.1/24

This route matches if the remote address of the request was, for example, 192.168.1.10.

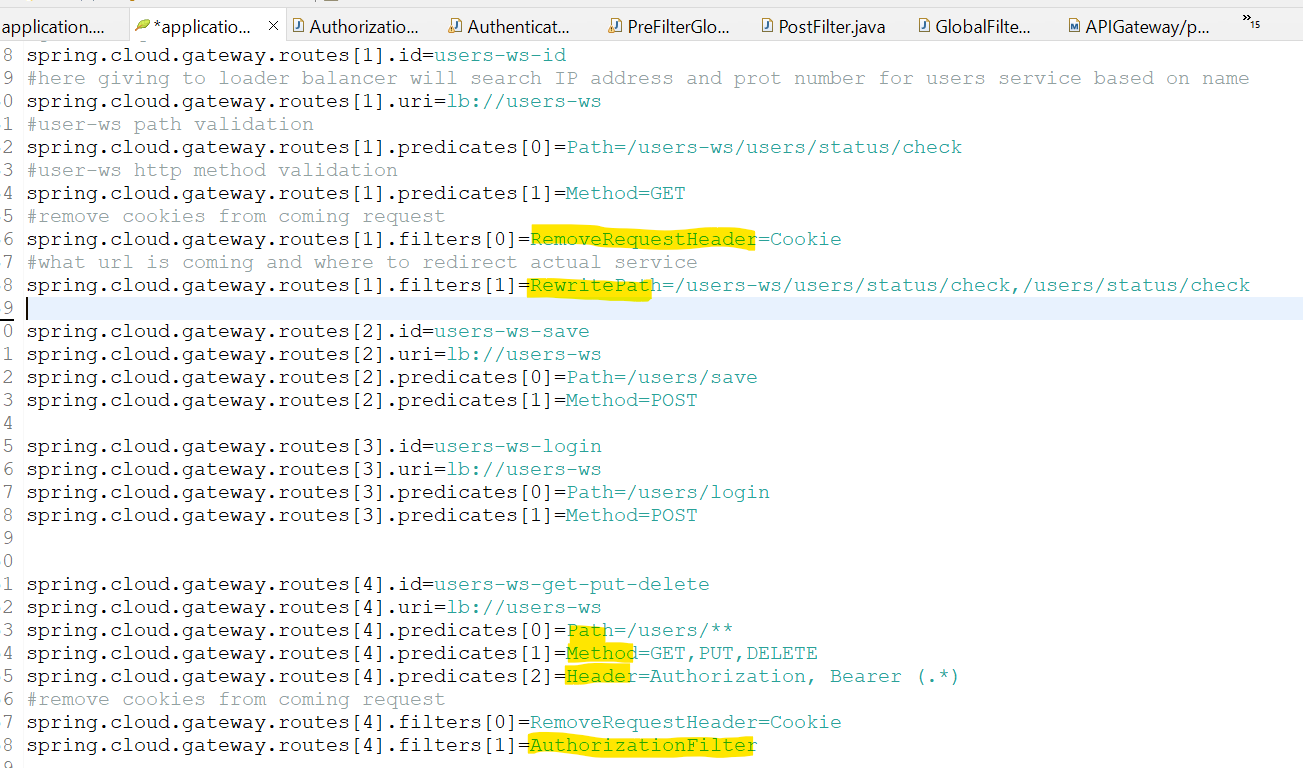
**11. The Weight Route Predicate Factory**

The Weight route predicate factory takes two arguments: group and weight (an int). The weights are calculated per group. The following example configures a weight route predicate:

1. spring.cloud.gateway.routes[0].id = weight\_high
2. spring.cloud.gateway.routes[0].uri = https://wighthigh.org
3. spring.cloud.gateway.routes[0].predicates[0] = Weight=group1, 8
5. spring.cloud.gateway.routes[1].id = weight\_low
6. spring.cloud.gateway.routes[1].uri = https://weighlow.org
7. spring.cloud.gateway.routes[1].predicates[0] = **Weight**=group1, 2

This route would forward ~80% of traffic to [weighthigh.org](https://weighthigh.org/) and ~20% of traffic to [weighlow.org](https://weighlow.org/)

**Some Example**



**AuthorizationFilter** is custom class which execute before redirecting url to destination service

Some Custom class filter logics

Prefilter, postfilter and both prefilter/postfilter

Graphical user interface, text, application, email

Description automatically generated

**Post Filter .then() it will execute before sending response to client.**

Text

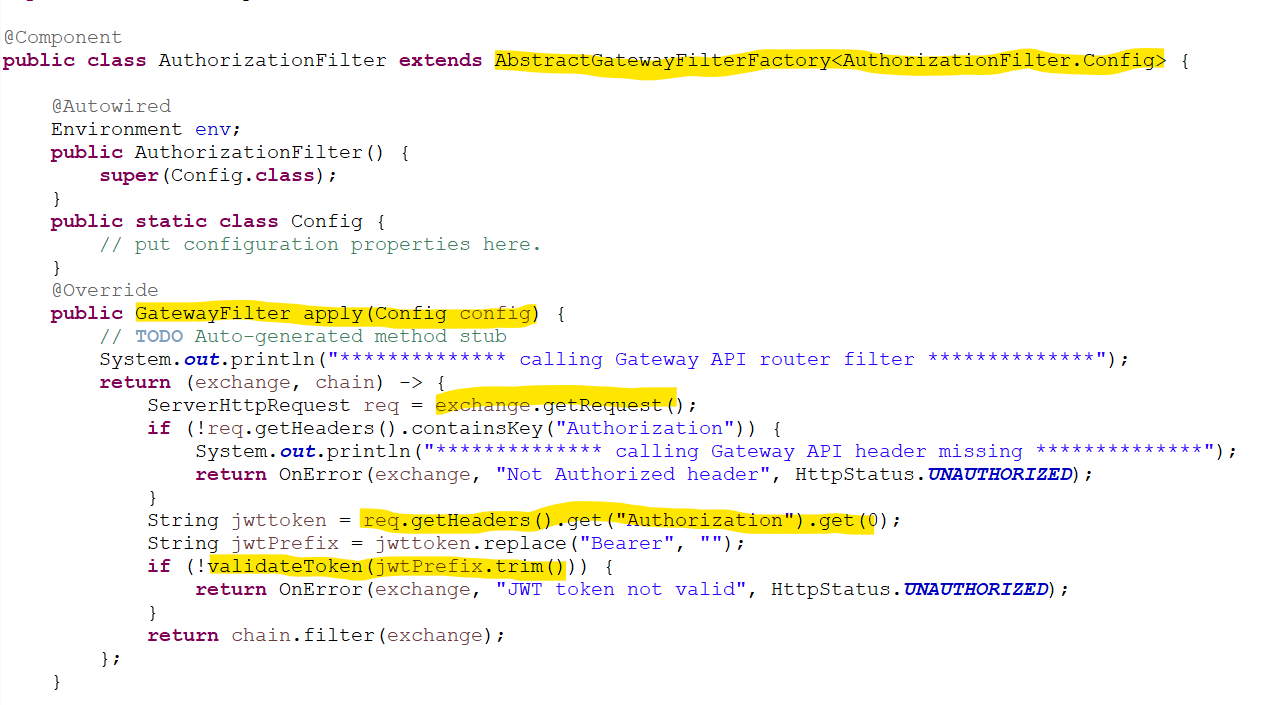
Description automatically generated with medium confidence

**Below Example for both Pre and Post Filter**

lGraphical user interface, text, application

Description automatically generated

below Example real time validate JWT token is valid or not before redirecting to destination service.



Graphical user interface, text, application

Description automatically generated

**Config Server**

Spring Cloud Config is Spring's client/server approach for storing and serving distributed configurations across

multiple applications and environments.

**Spring Cloud bus:** Config server distributes the dynamic changes properties file in microservices without loading or restart with help of messaging API.

Like Kafka, rabbit MQ etc.

This configuration store is ideally versioned under Git version control and can be modified at application runtime.

1. If properties file with same in config server and other microservice then priority high config server. **(3) global for all microservice means common**
2. In config server follow the properties file naming with application name like application-name.properties. then it will high priority **(2)2nd high priority**
3. If properties file with environment then it will hight priority applicationname-profile.properties like userApp-dev.properties. **(1) high priority**
4. If have multiple properties file in micro-service then follow by application-name-env.properties like dev, staging, prod and set active profile which one you want

Diagram

Description automatically generated

**Spring Cloud bus:** Config server distributes the dynamic changes properties file in microservices without loading or restart with help of messaging API.

Like Kafka, rabbit MQ etc.

Spring Cloud Bus enables broadcasting the state changes among the services over a message broker like Kafka or RabbitMQ.

So, in our scenario, each config client reading from the same config server can communicate over a message broker and refresh themselves in case of configuration changes. That means we do not need to make a refresh request for each config client application instead, we just broadcast a message to the services over Spring Cloud Bus to refresh themselves.

**Actuator:** it contains lot of information about server, bean, health, matrix, refresh, memory, CPU utilization etc. so it will help use to refresh server config server cache to get latest value

so, it will clear the cache of config server and take the latest changes and broadcast to client microservice.

Use POST method to call Server config end points.

**@RefreshScope**: its clear the local cache and re-initiate the bean to get the latest value whenever it used

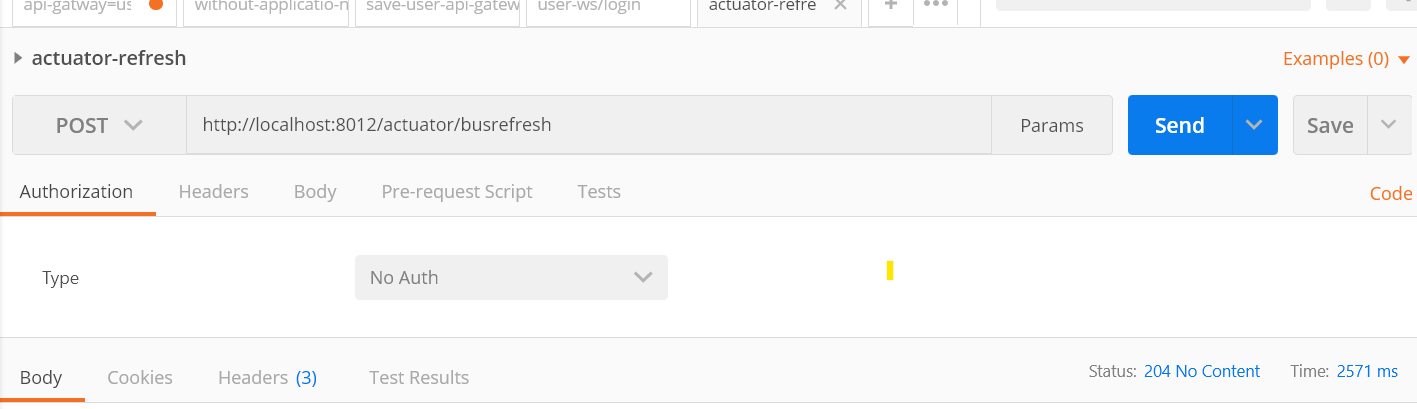
Refresh scope beans are lazy proxies that initialize when they are used (i.e. when a method is called), and the scope acts as a cache of initialized values. To force a bean to re-initialize on the next method call you just need to invalidate its cache entry.

@RefreshScopeare re-instantiated without restarting the DI container by Posting refresh endpoint.

POSTMAN

Post method

<http://localhost:8012/actuator/busrefresh>



Diagram

Description automatically generated

**ServerConfig API setup**

Add dependency. Kafka bus used to distribute the changes in multiple microservice.

Text

Description automatically generated with low confidence

**@EnableConfigServer in main class**

Setup properties file

Setup the git environment

Setup the default zookeeper nodes to broadcast the changes to multiple clients.



Put all properties file inside project only.

**Client Application setup**

Add dependency.

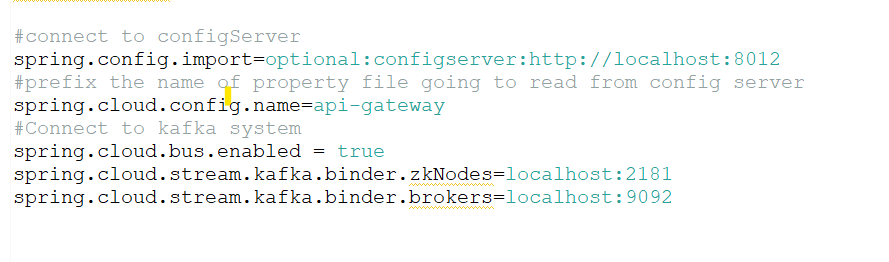
We want to make our services also communicate over a message broker for listening and broadcasting refresh events. So here too,

we need to add the Spring Cloud Bus dependency to the config client’s pom.xml:

Graphical user interface, application

Description automatically generated

**Properties file configuration**

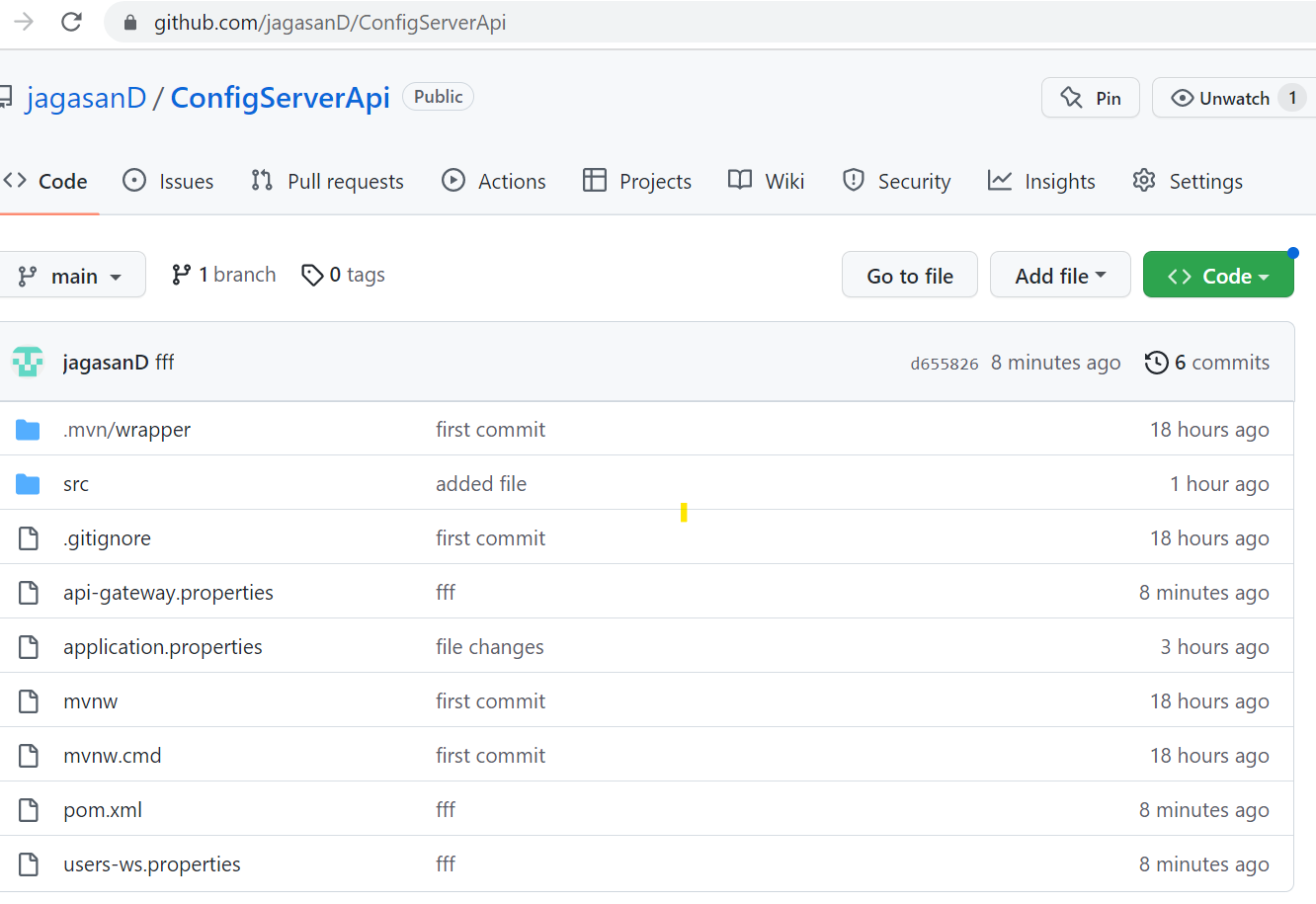


**Set @RefreshCope**

Graphical user interface, text, application, email

Description automatically generated

**GitHub setup spring cloud config server**



**Kafka configuration**

Download Kafka and extract inside c folder.

<https://kafka.apache.org/downloads>

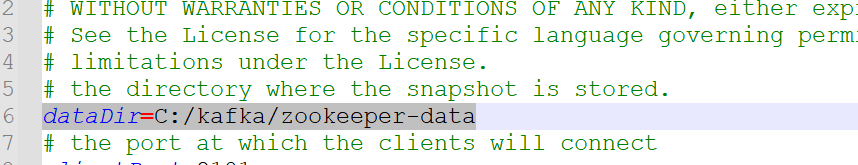
**kafka\_2.13-3.2.1.tgz**

<https://kafka.apache.org/quickstart>

configure logs for zookeeper. properties and server. properties

**zookeeper. properties**

dataDir=C:/kafka/zookeeper-data



**server. Properties**

log.dirs=C:/kafka/kafka-logs

Graphical user interface, application, chat or text message

Description automatically generated with medium confidence

Start zookeeper inside C:/Kafka folder.

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

A screenshot of a computer

Description automatically generated with medium confidence

**Start Kafka server.**

.\bin\windows\kafka-server-start.bat .\config\server.properties

A screenshot of a computer

Description automatically generated with medium confidence

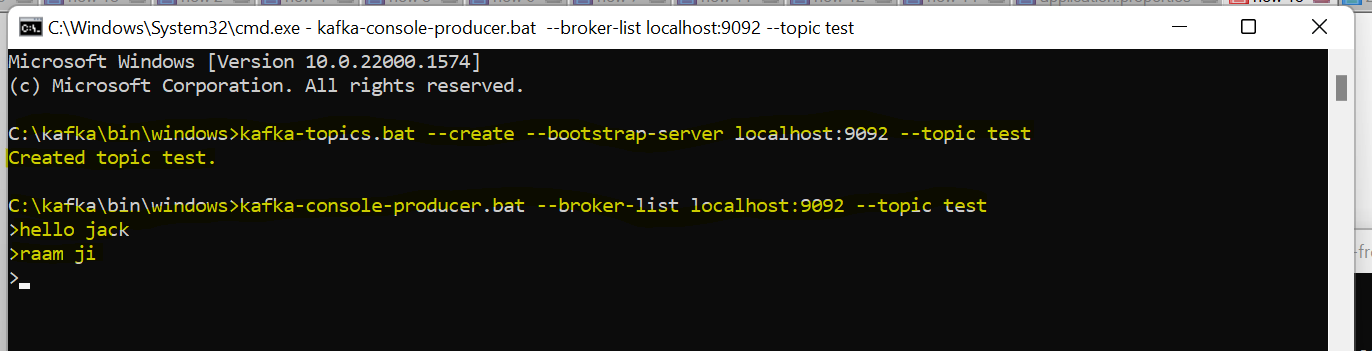
**Create topic and produce message.**

path kafaka>windows> run below command.

kafka-topics.bat --create --bootstrap-server localhost:9092 --topic test

produce message

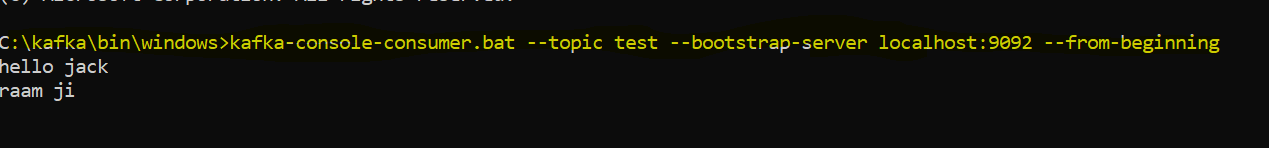
kafka-console-producer.bat --broker-list localhost:9092 --topic test



**Read message from topics.**

path kafaka>windows> run below command.

kafka-console-consumer.bat --topic test --bootstrap-server localhost:9092 --from-beginning



Note

Zookeeper default port number where client can connect see inside zookeeper.properties file (no need change anything )

# the port at which the clients will connect

clientPort=2181

Server default port number find inside server.properties file to connect Kafka server

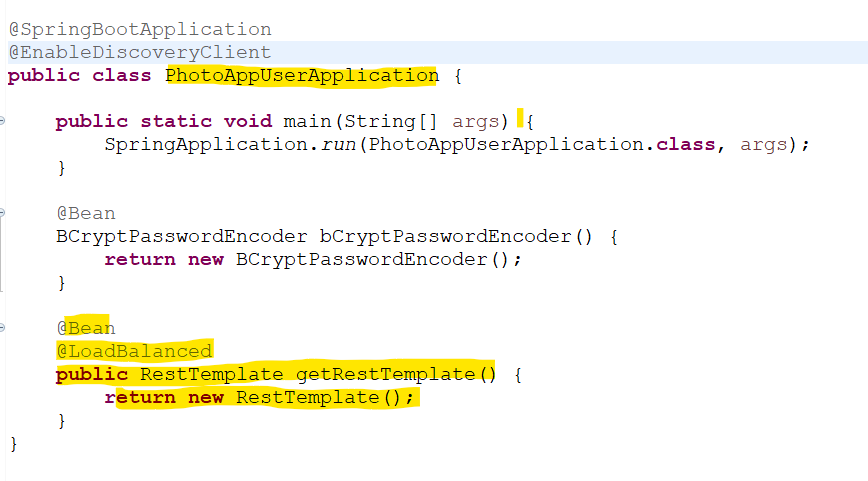
Localhost:9092

Client-Side Load balance call Rest template

**Rest Template**

Diagram

Description automatically generated



**Below no need to hard coded IP address and Port number only destination application name is required.**

Text

Description automatically generated with medium confidence

**Client-side Load balancing using Feign.**

**Graphical user interface, text, application, email

Description automatically generated**

Spring Cloud Open Feign — **a declarative REST client for Spring Boot apps.**

Feign makes writing web service clients easier with pluggable annotation support, which includes Feign annotations and JAX-RS annotations.

Also, Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters as used in Spring Web.

One great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.

It helps in writing web service clients more easily. In order to use Feign, create an interface and apply @FeignClient annotation on it. Moreover,

it internally generates a Proxy class at runtime using Dynamic Proxy Pattern. It actually gets Application name (Service Instance) from Eureka and supports Making HTTP call.

We should use two annotations for this concept, they are:

**@EnableFeignClient** in main class

**@FeignClient(name=”destination-application-name”)**

Here Albums multiple instances running so feign will be distribute the calls based on load.

It takes meta data information like service name and call to eureka discovery to give all information based on

Distribute call for multiple instances, to communicate Albums service we don’t need hard code URL and port number to

Communicate the Albums microservice.



**Calling in Service implantation class.**

Graphical user interface, application, Word

Description automatically generated

**Exception Handing**

Handling exception we need to implements ErrorDecoder , we can write our custom code for exception

**methodKey** 🡪 return clasName#methodName() example : AlbumFeignClient#getAlbums(String Id)

**Response**: it whole details about request and response, http code, reason, header etc

Graphical user interface, text, application

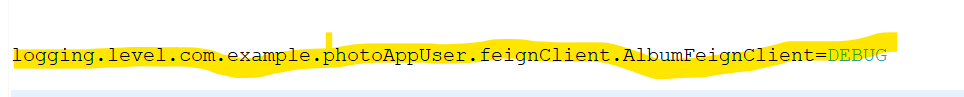
Description automatically generated

**Trace logger of Feign client.**

Properties file

Follow package name with class name;

Logging.level.packagename.classname=



**Main class create bean.**

Text

Description automatically generated

**Web client**

<https://reflectoring.io/spring-webclient/>

<https://howtodoinjava.com/spring-webflux/webclient-get-post-example/>

<https://www.javadevjournal.com/spring/spring-webclient/>

<https://www.baeldung.com/spring-webflux-errors>

**Resilience4J**

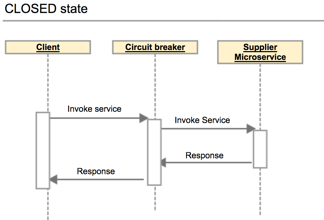
**Resilience4j**is a lightweight fault tolerance library designed for**Java 8** and functional programming. The library uses **[Vavr](http://www.vavr.io/" \t "_blank)**, which does not have any other external library dependencies. **Resilience4j** allows picking what you need.

[**https://resilience4j.readme.io/docs/circuitbreaker**](https://resilience4j.readme.io/docs/circuitbreaker)

Diagram

Description automatically generated

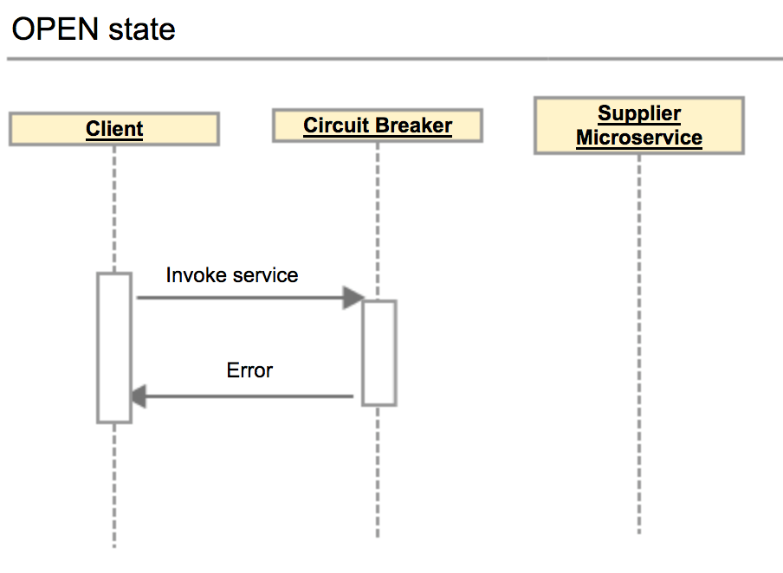
**Closed state:** when health communicate is happening between 2 microservice there is no exception then it will close state.



**Open state:** circuit breaker will open state when something goes wrong destination service and frequently getting error response. Execute fallback method.

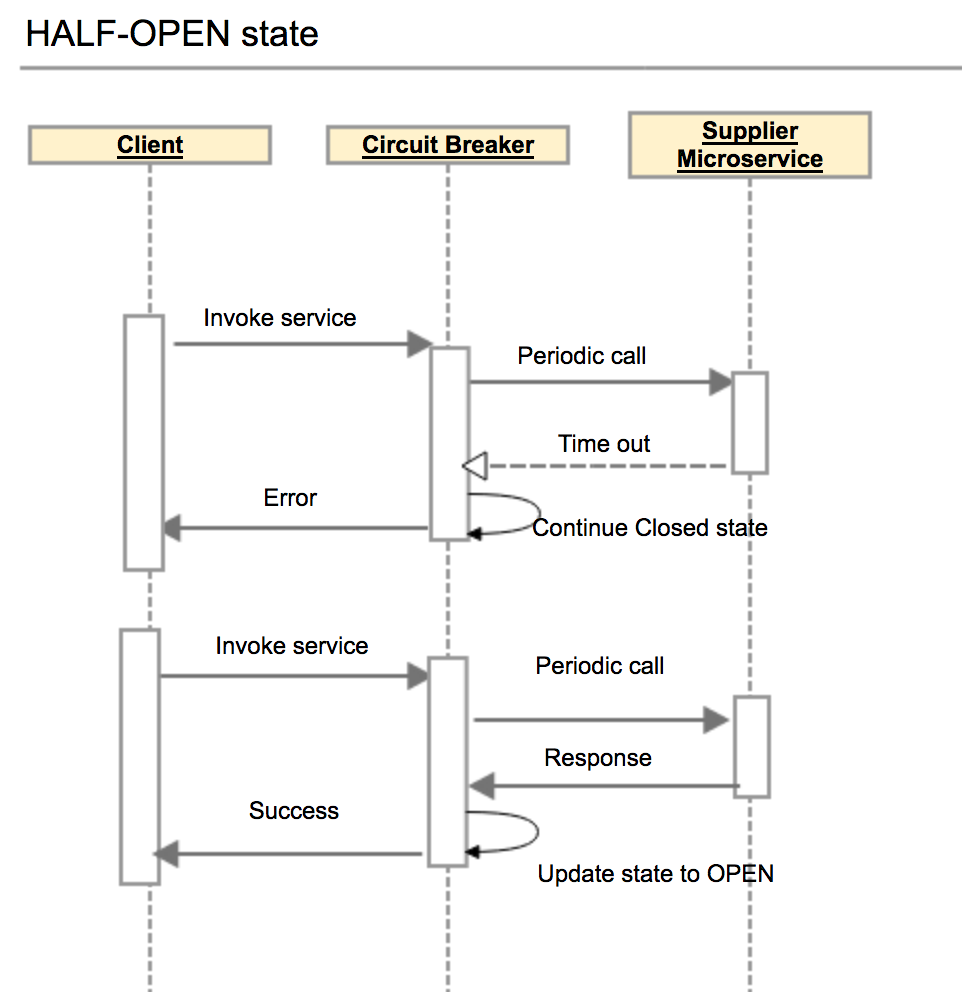
The circuit breaker allows microservices to communicate as usual and monitor the number of failures occurring within the defined time period. If the failure count exceeds the specified threshold value, the circuit breaker will move to the **Open** state.

Once the circuit breaker moves to the **Open** state, it will completely block the communication between microservices. So, the article service will not receive any requests, and the user service will receive an error from the circuit breaker.



**Half open state:** after certain amount of time, it will again try to call destination service if get success response frequently then state will go to close state otherwise open state.

**Half-Open** state, the circuit breaker will allow a limited number of requests to reach micro-service. If those requests are successful, the circuit breaker will switch the state to **Closed** and allow normal operations. If not, it will again block the requests for the defined timeout period.



# **Advantages of Circuit Breaker Pattern**

* Helps to prevent cascading failures.
* Handles errors gracefully and provides better under experience.
* Reduces the application downtimes.
* Suitable for handling asynchronous communications.
* State changes of the circuit breaker can be used for error monitoring.

**Order of Aspect below** default is first execute function, 2 bulkheads, 3 time limiter, and last circuit breaker,

If you want to change default order then need to do configuration order changes see below example



Example circuit breaker with retry.

**Retry**: it will used for retrying the service number of time when occurring exception with amount of time interval and other parameter.

resilience4j.retry.retry-aspect-order=2

resilience4j.retry.instances.albums-ws.max-attempts=3

resilience4j.retry.instances.albums-ws.wait-duration=2s

# it will increase waiting every failure calls like 2s, 2\*5=10s, 10\*5 =50s

resilience4j.retry.instances.albums-ws.enable-exponential-backoff=true

resilience4j.retry.instances.albums-ws.exponential-backoff-multiplier=5

**Circuit braker property setup**

resilience4j.circuitbreaker.circuit-breaker-aspect-order=1

#configuration %if threshold failed to open circuit breaker

resilience4j.circuitbreaker.instances.albums-ws.failure-rate-threshold=50

#size 6 means if request calls 3 is failed out 6 six(means 50% failed) then circuit breaker will OPEN state

resilience4j.circuitbreaker.instances.albums-ws.sliding-window-size=6

#setting default HALF-OPEN state

resilience4j.circuitbreaker.instances.albums-ws.automatic-transition-from-open-to-half-open-enabled=true

#The time that the Circuit Breaker should wait before transitioning from open to half-open.

resilience4j.circuitbreaker.instances.albums-ws.wait-duration-in-open-state=20s

resilience4j.circuitbreaker.instances.albums-ws.sliding-window-type=count-based

#store recording buffer 10 records only

resilience4j.circuitbreaker.instances.albums-ws.event-consumer-buffer-size=10

#failure rate start calculation after 2 request failed

#resilience4j.circuitbreaker.instances.albums-ws.minimum-number-of-calls=2

**Documents below link**

<https://resilience4j.readme.io/docs/circuitbreaker>

When the failure rate is equal or greater than the threshold the Circuit Breaker transitions to open and starts short-circuiting calls.

failureRateThreshold = 50 default value%

When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.

slowCallRateThreshold = 100 default value%

Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.

slowCallDurationThreshold = 60000 [ms]

Configures the number of permitted calls when the CircuitBreaker is half open.

permittedNumberOfCallsInHalfOpenState =10

Configures a maximum wait duration which controls the longest amount of time a CircuitBreaker could stay in Half Open state, before it switches to open.  
Value 0 means Circuit Breaker would wait infinitely in HalfOpen State until all permitted calls have been completed.

maxWaitDurationInHalfOpenState =0 ms

If the sliding window is COUNT\_BASED, the last slidingWindowSize calls are recorded and aggregated.  
If the sliding window is TIME\_BASED, the calls of the last slidingWindowSize seconds recorded and aggregated.

slidingWindowType= COUNT\_BASED

setup to count when state Close, it counts how may request failed and check with % value is equals to or greater then

if it equals or greater then then stop to call microservice and return the fallback method value.

**Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed**

slidingWindowSize =100

Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate or slow call rate.  
For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated.  
If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.

minimumNumberOfCalls =100

The time that the CircuitBreaker should wait before transitioning from open to half-open.

waitDurationInOpenState =60000 [ms]

If set to true it means that the CircuitBreaker will automatically transition from open to half-open state and no call is needed to trigger the transition. A thread is created to monitor all the instances of CircuitBreakers to transition them to HALF\_OPEN once waitDurationInOpenState passes. Whereas, if set to false the transition to HALF\_OPEN only happens if a call is made, even after waitDurationInOpenState is passed. The advantage here is no thread monitors the state of all CircuitBreakers.

automaticTransitionFromOpenToHalfOpenEnabled =false

A list of exceptions that are recorded as a failure and thus increase the failure rate.  
Any exception matching or inheriting from one of the list counts as a failure, unless explicitly ignored via ignoreExceptions.  
If you specify a list of exceptions, all other exceptions count as a success, unless they are explicitly ignored by ignoreExceptions.

recordExceptions = empty

A list of exceptions that are ignored and neither count as a failure nor success.  
Any exception matching or inheriting from one of the list will not count as a failure nor success, even if the exceptions is part of recordExceptions.

ignoreExceptions = empty

A custom Predicate which evaluates if an exception should be recorded as a failure.  
The Predicate must return true if the exception should count as a failure. The Predicate must return false, if the exception  
should count as a success, unless the exception is explicitly ignored by ignoreExceptions.

recordFailurePredicate = true

A custom Predicate which evaluates if an exception should be ignored and neither count as a failure nor success.  
The Predicate must return true if the exception should be ignored.The Predicate must return false, if the exception should count as a failure.

ignoreExceptionPredicate= false

**bulkhead**

The bulkhead pattern **limits the maximum number of concurrent calls to an external service.**

**The idea behind bulkheads is to set a limit on the number of concurrent calls we make to a remote service. We treat calls to different remote services as different, isolated pools and set a limit on how many calls can be made concurrently.**

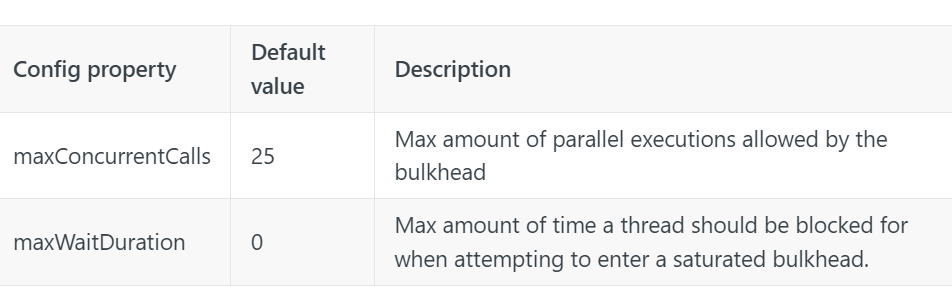
It ensures the failure in one service doesn’t cause the whole system to go down. It means do not burden service with calls more than its capacity and for that **it controls the number of concurrent requests the service can take**, the number of resources waiting for the response from the service can be limited by this way. There are two implementations of bulkhead patterns in Resilience4j.

**Semaphore** – In this approach, we limit the number of concurrent requests to the service. It will reject the incoming requests once the limit is hit.

**FixedThreadPoolBulkhead** – In this approach, we isolate a set of thread pool from system resources, using only that thread pool for the service. We also use a waiting queue apart from the thread pool, if both the thread pool and queue are full then in that case, the request will get rejected with BulkheadFullException

Text, application

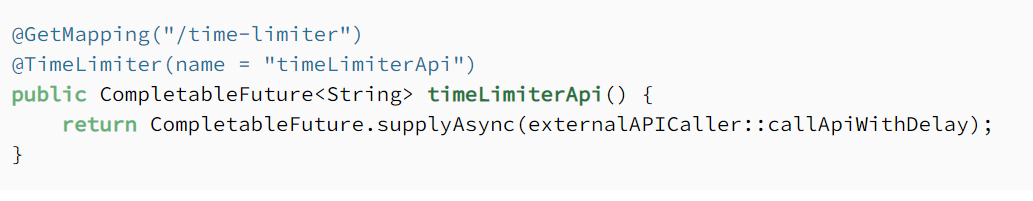
Description automatically generated



**Time Limiter**

We can **use the**[**time limiter pattern**](https://www.baeldung.com/resilience4j#time-limiter)**to set a threshold timeout value for async calls made to external systems**.

**Resilience4j’s [TimeLimiter](https://resilience4j.readme.io/docs/timeout) can be used to set time limits (timeouts) on asynchronous operations implemented with CompleteableFutures**.



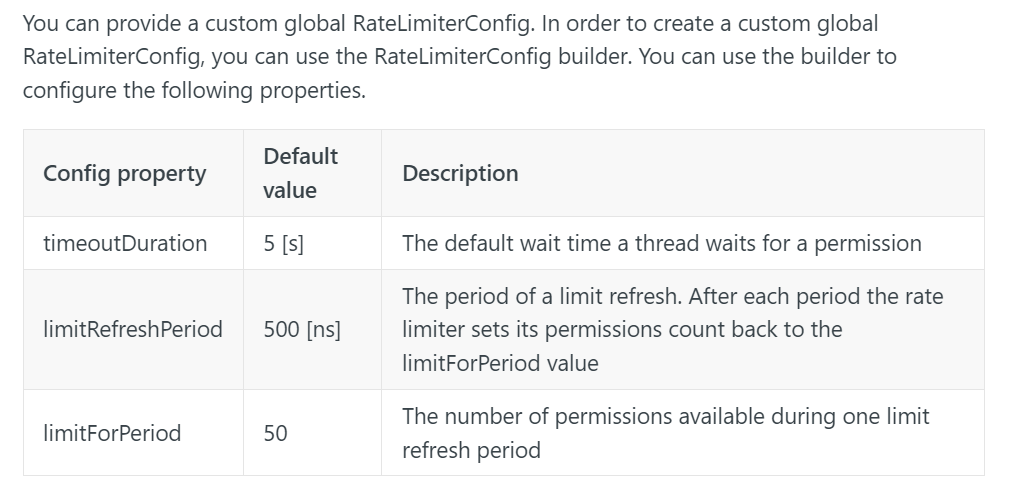
Text

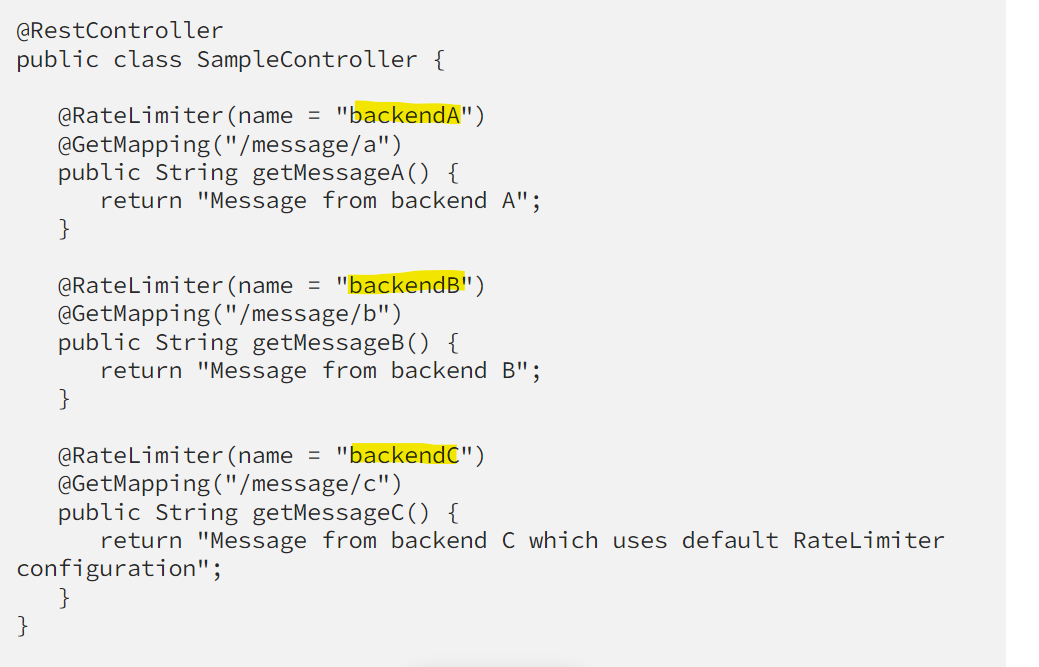
Description automatically generated

**Rate Limiter:**

**The**[**rate limiter pattern**](https://www.baeldung.com/resilience4j#rate-limiter)**limits the rate of requests to a resource.**

Rate-limiting is an important concept to be considered while building several types of applications such as websites, microservices, client applications, batch processing applications etc. For example, you might consider limiting the incoming requests to a RESTful service hosted on a server with limited amount of resources (CPU, memory etc.). Or it may be necessary to limit the number of requests sent from the client application you are developing to an external service. Not only it is important to limit the load, it also helps to eliminate the security concerns against malicious attacks (e.g. [DoS or DDoS](https://en.wikipedia.org/wiki/Denial-of-service_attack)) to ensure application availability and build more robust applications.





Text

Description automatically generated

**Distributed tracing Web Flex (Using Zipkin and Micrometer)**

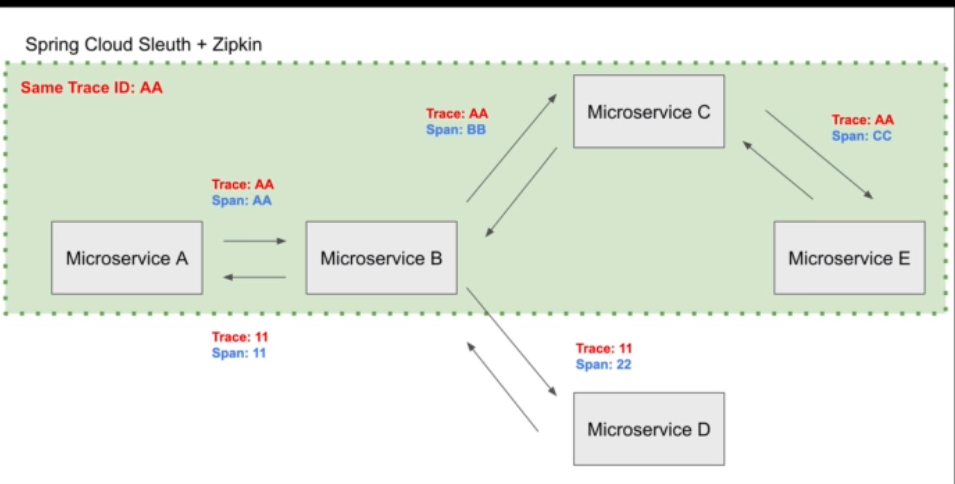
**Trace**: tracing id is common for all micro-service call for single request

**Span**: span Id is different each microservice for single request

**Zipkin**: it used for visualization, dashboard for trace your Trace and Span ID and more details about tracing.

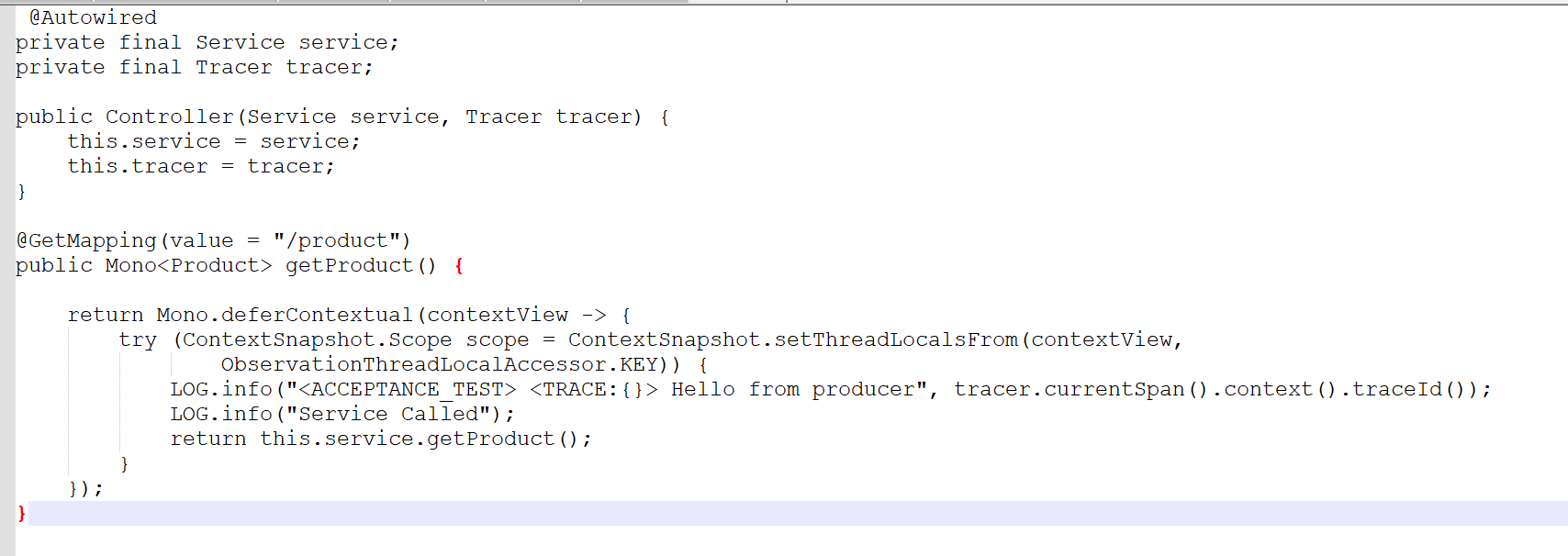
**Micrometer:**

the Spring Cloud Sleuth project will be [removed](https://github.com/spring-cloud/spring-cloud-release/wiki/Spring-Cloud-2022.0-Release-Notes) from the release train. However, the core of the Sleuth project is being moved into the Micrometer Tracing project. Because of this, the autoconfiguration from Sleuth has been moved into Spring Boot 3 with the addition of support for **Micrometre’s new Observation API**.



Only dependency and properties file setup enough to trace logs if you want some extra functionality then you can do through java code.

**Programmatic approach to implement and get traceId**



**Properties file setup each microservice**



**Dependency. Each microservice.**

Text

Description automatically generated

**Zipkin**

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

If you have a trace ID in a log file, you can jump directly to it. Otherwise, you can query based on attributes such as service, operation name, tags and duration. Some interesting data will be summarized for you, such as the percentage of time spent in a service, and whether or not operations failed.

More details

<https://betterprogramming.pub/tracing-in-spring-boot-3-webflux-d432d0c78d3e#f12f>

<https://openvalue.blog/posts/2022/12/16/tracing-in-spring-boot-2-and-3/>

**Zipkin download jar and Run.**

<https://zipkin.io/pages/quickstart>

If you have Java 8 or higher installed, the quickest way to get started is to fetch the [latest release](https://search.maven.org/remote_content?g=io.zipkin&a=zipkin-server&v=LATEST&c=exec) as a self-contained executable jar:

java -jar zipkin.jar

**access zipkin.**

<http://localhost:9411>



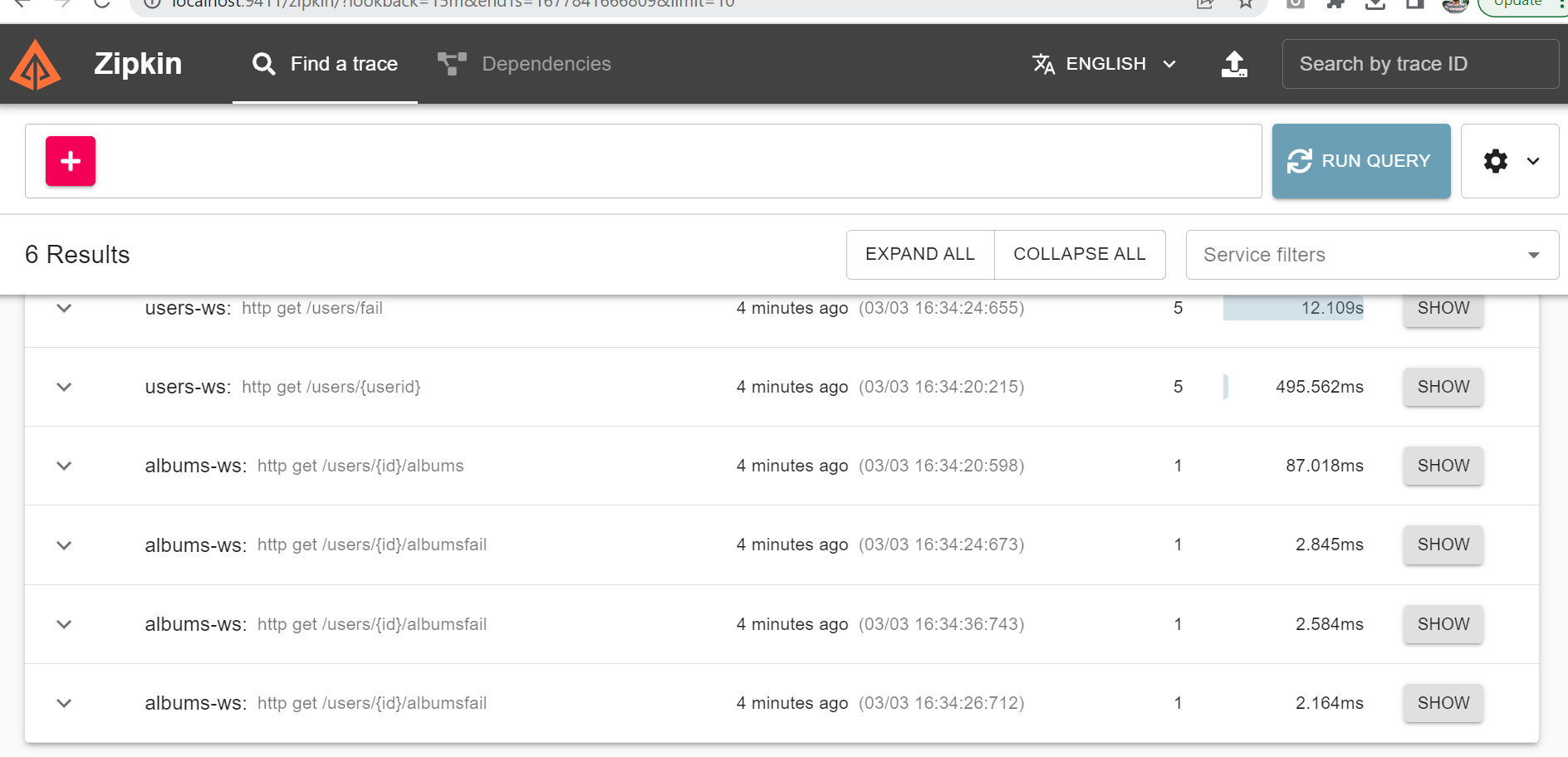
Text

Description automatically generated

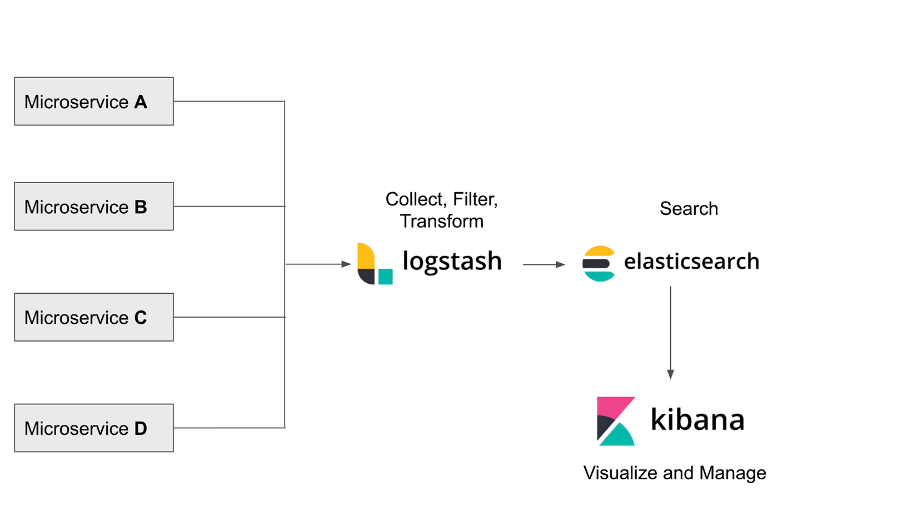
Zipkin dashboard

Graphical user interface, text, application, email

Description automatically generated



**Aggregating Logs with ELK (Elasticsearch, Logstash and Kibana).**



**Logstash**

Logstash is used to aggregate and process data and send it to Elasticsearch. It is an open-source, server-side data processing pipeline that collect data from a multitude of sources simultaneously, transforms it, and then sends it to collect. It also transforms and prepares data regardless of format by identifying named fields to build structure and transform them to converge on a common format. For example, since data is often scattered across different systems in various formats, Logstash allows you to tie different systems together like web servers, databases, Amazon services, etc. and publish data to wherever it needs to go in a continuous streaming fashion.

**Elasticsearch (**analytics engine**,** search based on index**)**

Elasticsearch is a distributed, free, and open **search** and **analytics engine** for all types of **data, including textual, numerical, store, geospatial, structured, and unstructured.**

Elasticsearch allows you to store, search, and analyze huge volumes of data quickly and in near real-time and give back answers in milliseconds. It’s able to achieve fast search responses because instead of searching the text directly, **it searches an index.** It uses a structure based on documents instead of tables and schemas and comes with extensive REST APIs for storing and searching the data. At its core, you can think of Elasticsearch as a server that can process JSON requests and give you back JSON data.

**Kibana** (log analysis)

[Kibana](https://www.knowi.com/blog/grafana-vs-kibana/) is a data visualization and management tool for Elasticsearch that provides real-time histograms, line graphs, pie charts, and maps. It lets you visualize your Elasticsearch data and navigate the Elastic Stack. You can select the way you give shape to your data by starting with one question to find out where the interactive visualization will lead you.

**Create home directory inside c drive ELT-tools**

**Logstash configuration inside Logstash home for read all logs file and send to elastic search.**

**Input section**

1. **Create new config file and put inside logstash home folder.**
2. **setup read the file in different location in input section.**

**output section**

1. **Check the type of log condition if match.**
2. **Create elastic search setup hostname, user, password, cacert.**
3. **You will get username and password form console, when first time run the elastic search.**
4. **Keep username, password, token some file that required configure.**
5. **Elastic token is required in kibana setup.**
6. **Token will expire in 30minutes by default.**

**Below config file setup**

input {

file {

type=>"users-ws-log"

path=>"C:/Users/003JNH744/Desktop/micro-service-2023/photoAppUser/users-ws.log"

}

file {

type=>"albums-ws-log"

path=>"C:/Users/003JNH744/Desktop/micro-service-2023/PhotoAppApiAlbums-master/albums-ws.log"

}

file {

type=>"api-gateway-log"

path=>"C:/Users/003JNH744/Desktop/micro-service-2023/APIGateway/api-gateway.log"

}

}

output {

if [type] == "users-ws-log" {

elasticsearch {

hosts => ["https://localhost:9200"]

ssl => true

ssl\_certificate\_verification => true

index => "users-ws-%{+YYYY.MM.dd}"

user => "elastic"

password => "vvo3kJ5MxFSAONAVaT\*l"

cacert => "C:/ELK-tools/elasticsearch/config/certs/http\_ca.crt"

ilm\_enabled => false

manage\_template => false

}

}

if [type] == "albums-ws-log" {

elasticsearch {

hosts => ["https://localhost:9200"]

ssl => true

ssl\_certificate\_verification => true

index => "albums-ws-%{+YYYY.MM.dd}"

user => "elastic"

password => "vvo3kJ5MxFSAONAVaT\*l"

cacert => "C:/ELK-tools/elasticsearch/config/certs/http\_ca.crt"

ilm\_enabled => false

manage\_template => false

}

}

if [type] == "api-gateway-log" {

elasticsearch {

hosts => ["https://localhost:9200"]

ssl => true

ssl\_certificate\_verification => true

index => "api-gateway-%{+YYYY.MM.dd}"

user => "elastic"

password => "vvo3kJ5MxFSAONAVaT\*l"

cacert => "C:/ELK-tools/elasticsearch/config/certs/http\_ca.crt"

ilm\_enabled => false

manage\_template => false

}

}

stdout { codec => rubydebug }

}

**Run Logstash**

Inside Home folder of Logstash

bin\logstash -f simple-config.conf

elastic search first run then it will do setup internally security, http SSL etc.

**execute blow command inside elastic search home folder.**

bin\elasticsearch.bat

**create token command.**

bin\elasticsearch-create-enrollment-token -s kibana

**reset password command.**

bin\elasticsearch-reset-password -u elastic

**Kibana**

Run kibana and setup the token which are generated by Elasticsearch.

bin\kibana.bat

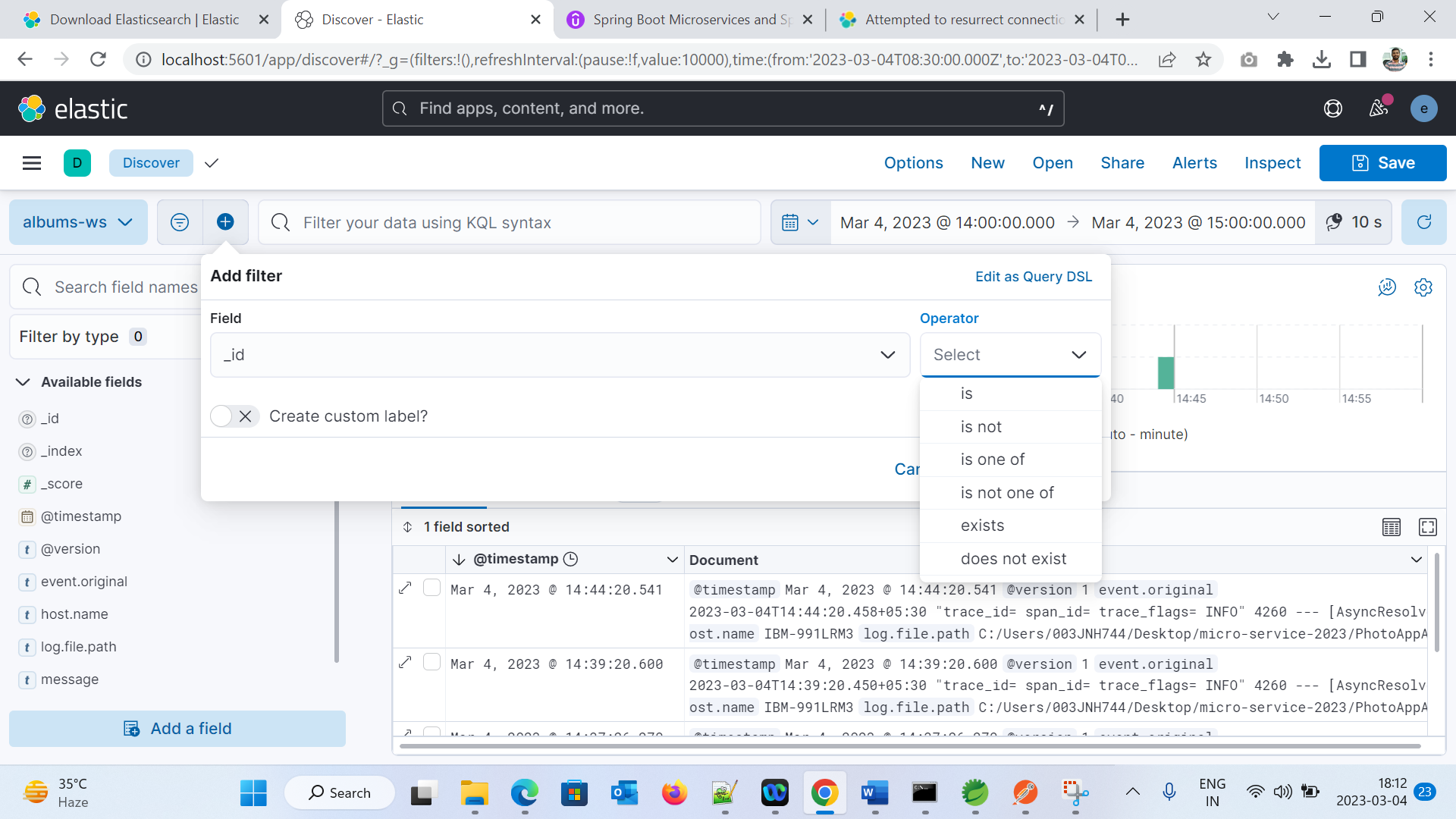
kibna.yml

**enable in kibana.yml**

elasticsearch.hosts: ['https://192.168.1.2:9200']

it will do automatic setup in kibana.yml

token, certification authorities,





Graphical user interface

Description automatically generated with medium confidence

Grafana

<https://piotrminkowski.com/2022/11/03/spring-boot-3-observability-with-grafana/>

Security for microservice URL

<https://medium.com/javarevisited/spring-boot-authorization-creating-an-authorization-server-for-your-microservices-50a3aefd6ce8>