**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.

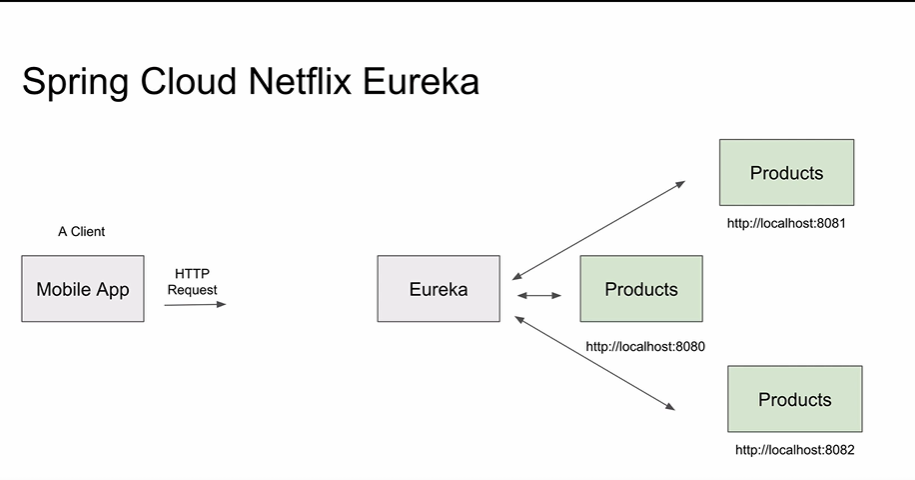
The Eureka Server is a Netflix OSS product, and Spring Cloud offers a declarative way to register and invoke services by Java annotation.

**Why do we need Eureka Server?**

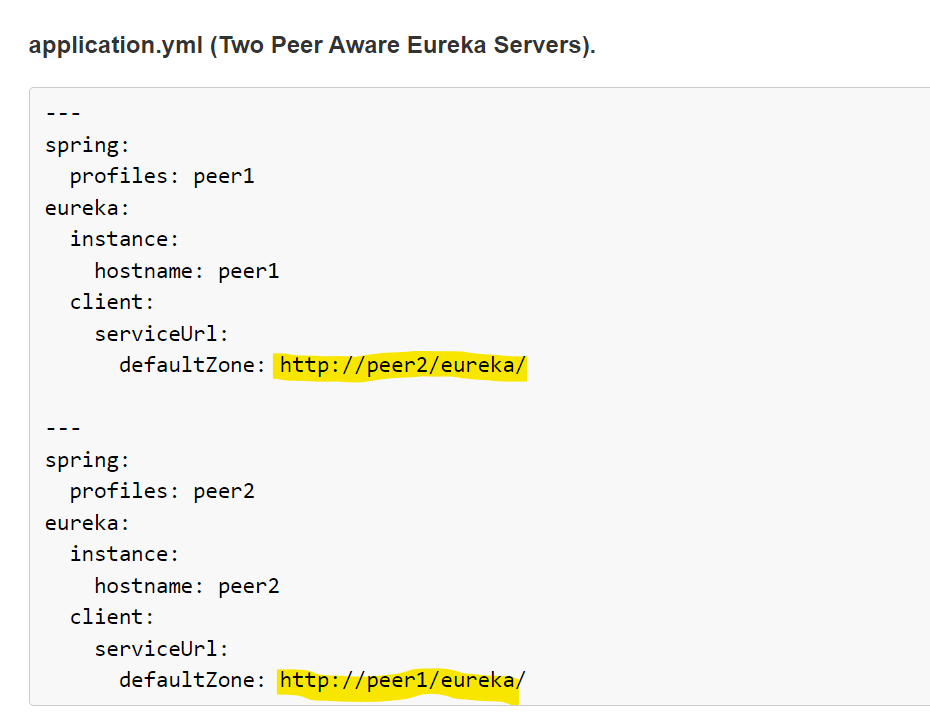
Suppose we have multiple instances of microservice that is running in different IP address/port number now one microservice want to call another one so which instance it will call?

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>

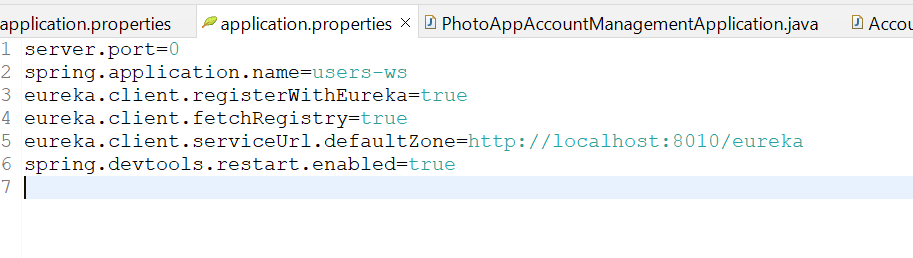
**Eureka Client**

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

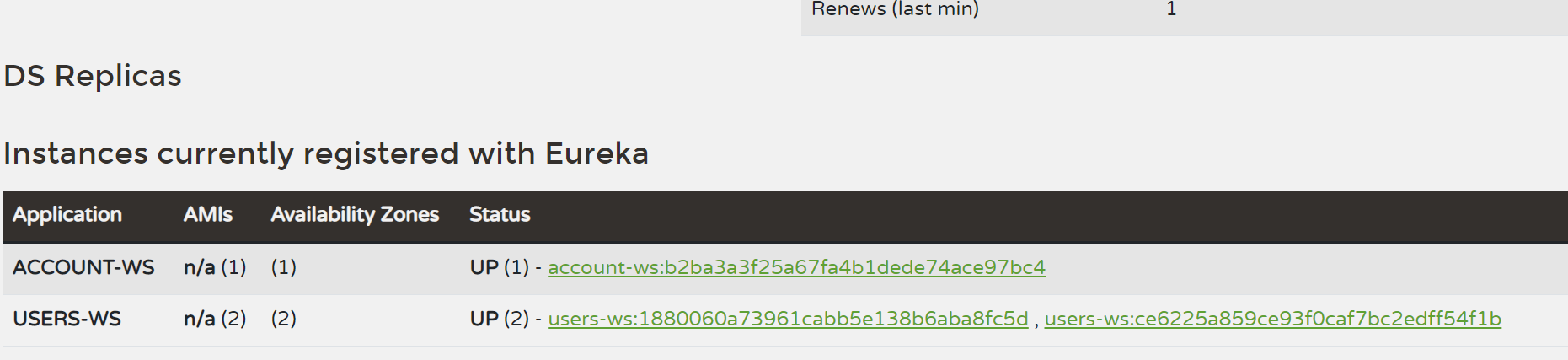
Description automatically generated

**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.

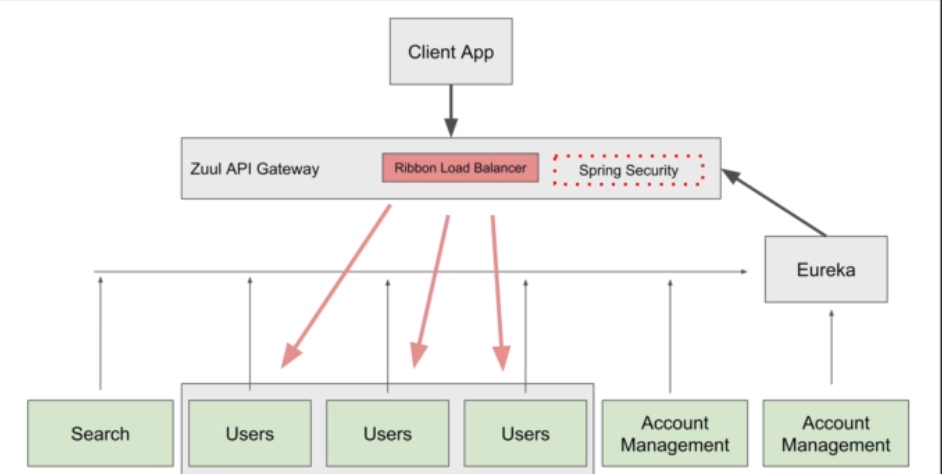
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance-id:${random.value}}



**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.

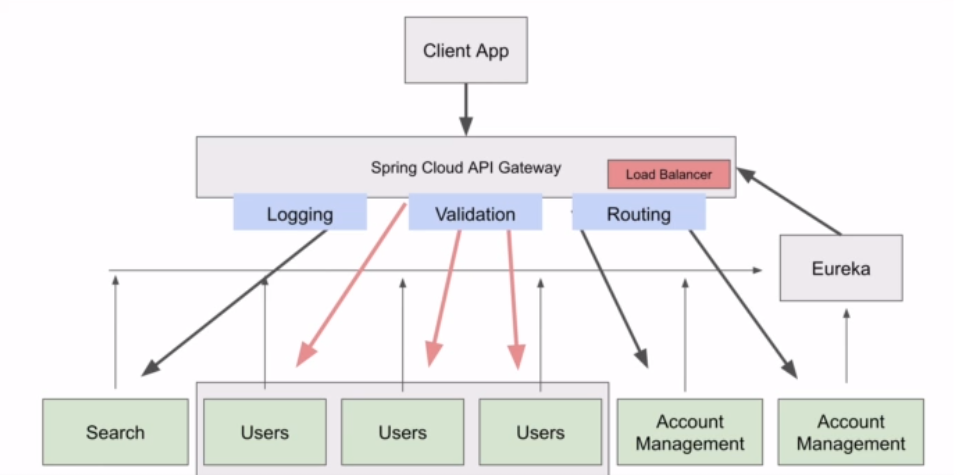
**The Spring Cloud Gateway**

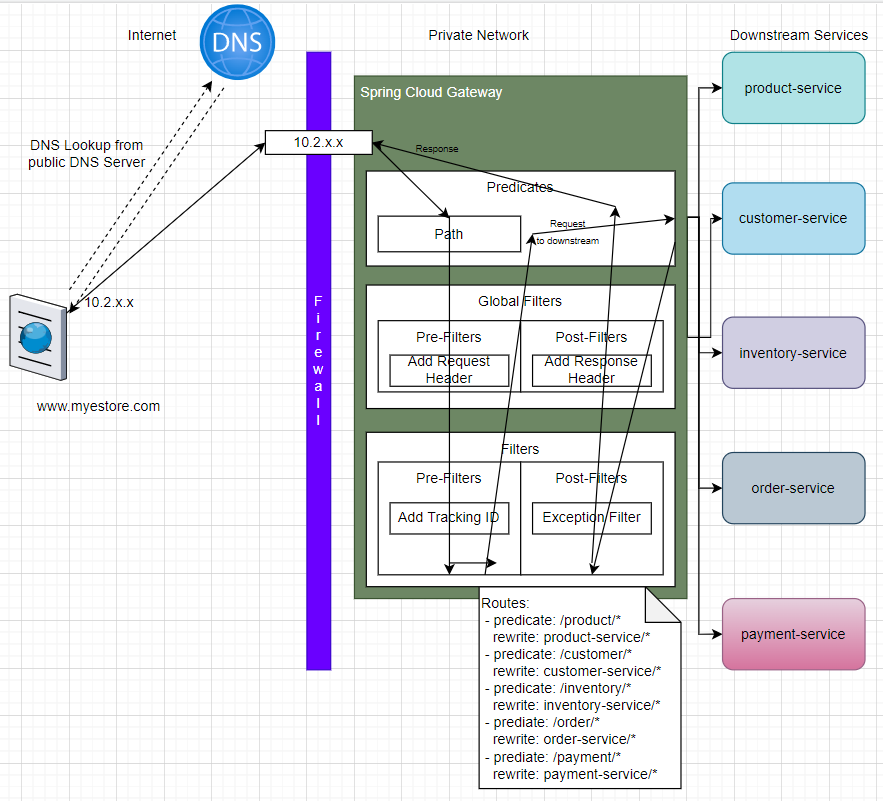
Resilience, security, and Observability

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

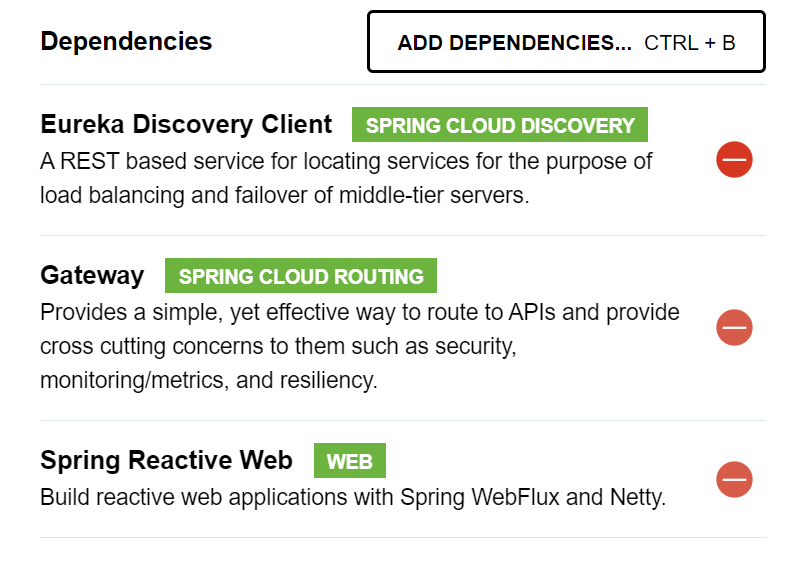




As seen from the above diagram, all the front-end clients talk to a single component and Spring Cloud Gateway takes care of routing the request to the respective microservice and gets the response to the client. In a cloud environment, **Spring Cloud Gateway runs in the Public subnet (public IP) and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



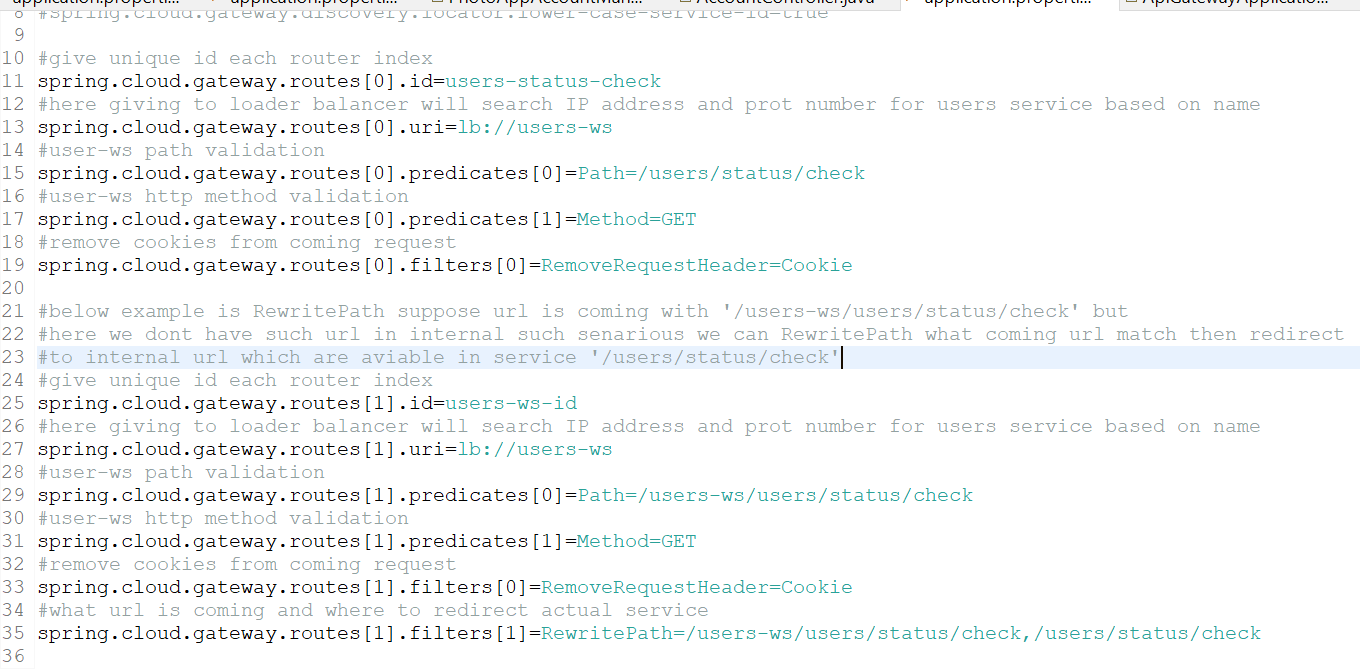
**Some Example about route, predicate, filter**

**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.

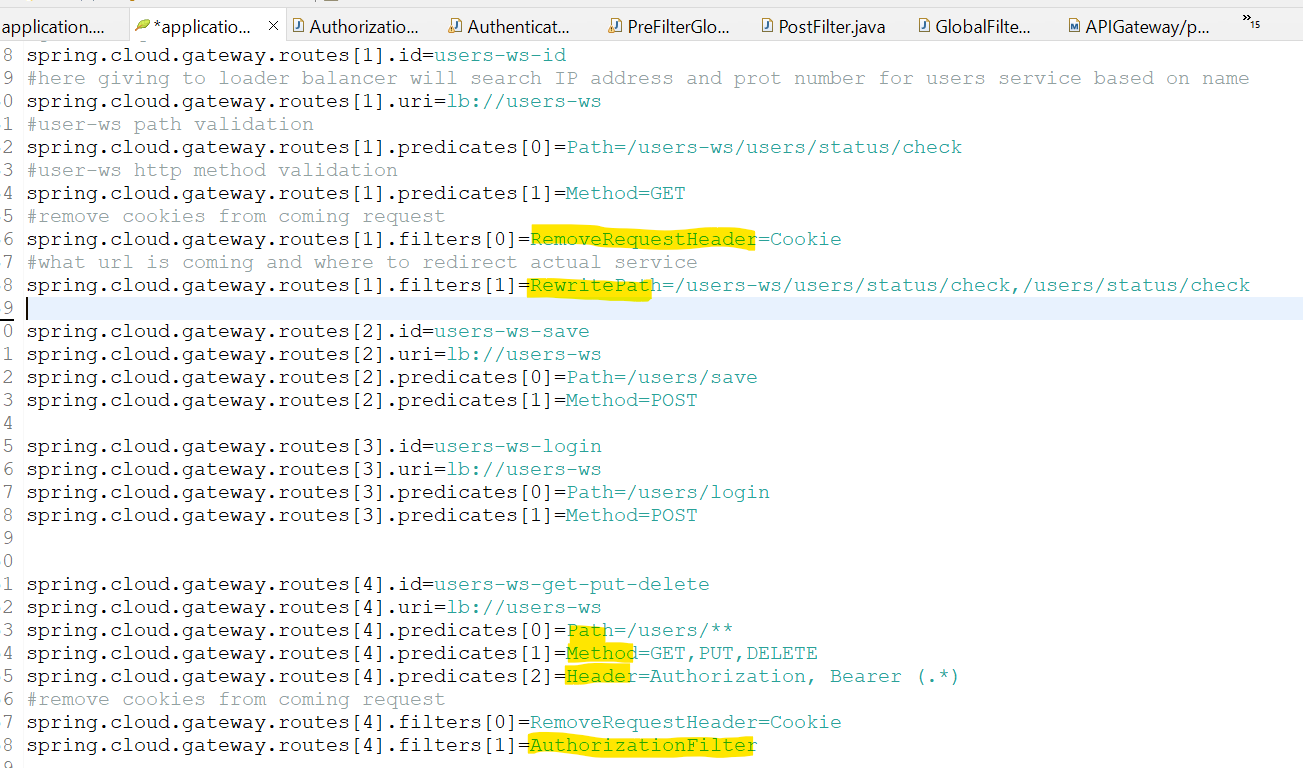
**11. The Weight Route Predicate Factory**

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

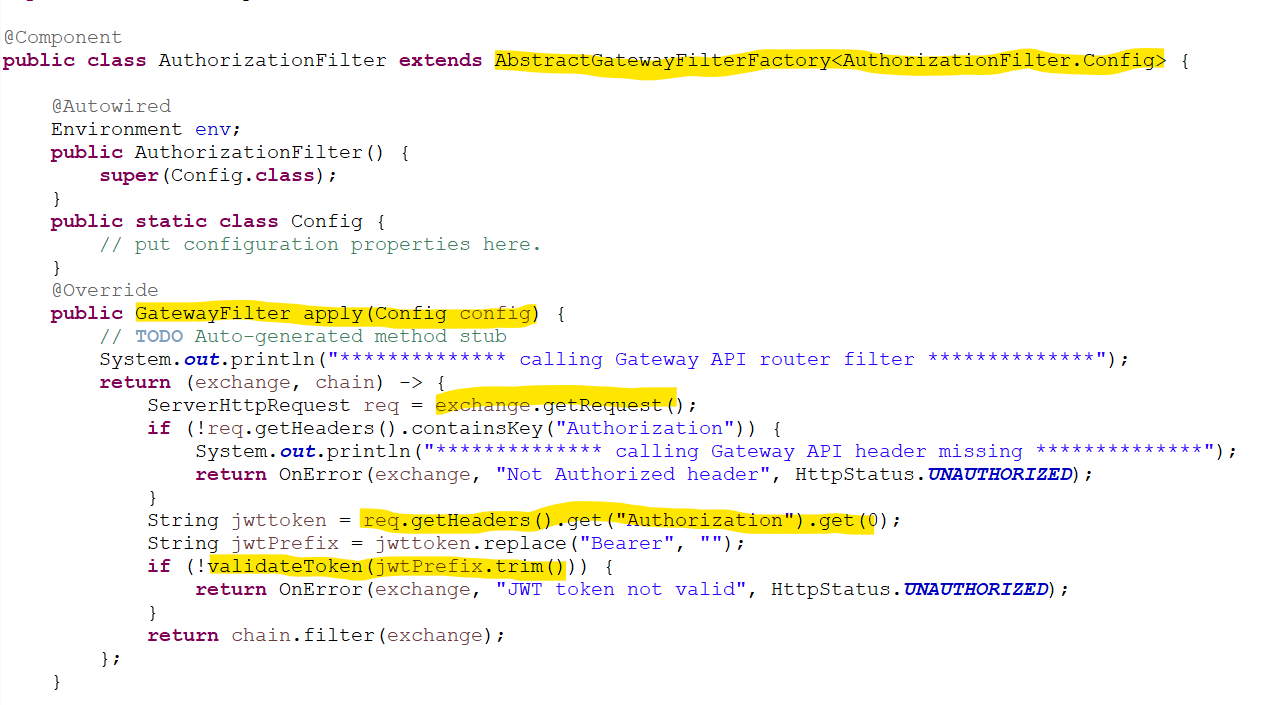
Description automatically generated with medium confidence

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

lGraphical user interface, text, application

Description automatically generated

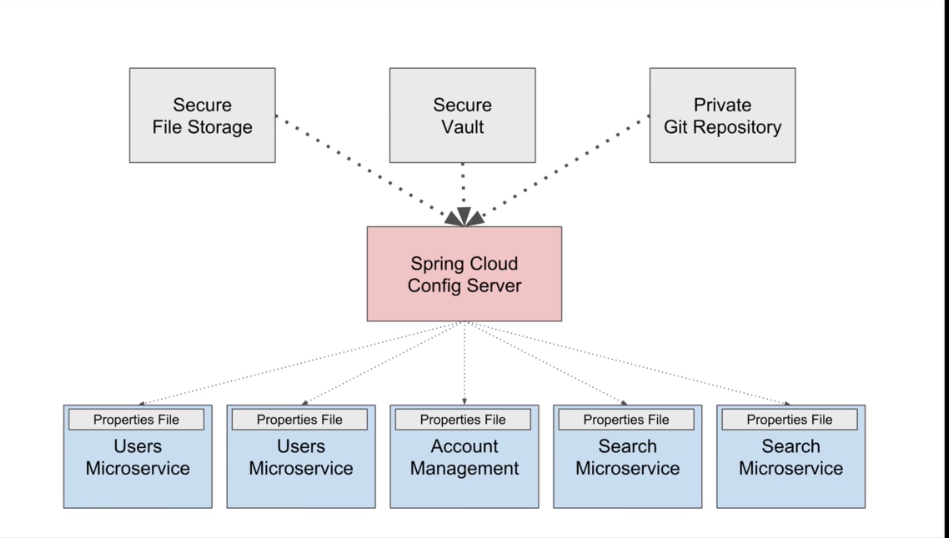
below Example real time validate JWT token is valid or not before redirecting to destination service.



Graphical user interface, text, application

Description automatically generated

Config Server



1. If properties file with same in config server and other microservice then priority high config server.
2. In config server follow the properties file naming with application name like application-name.properties.

Diagram

Description automatically generated

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