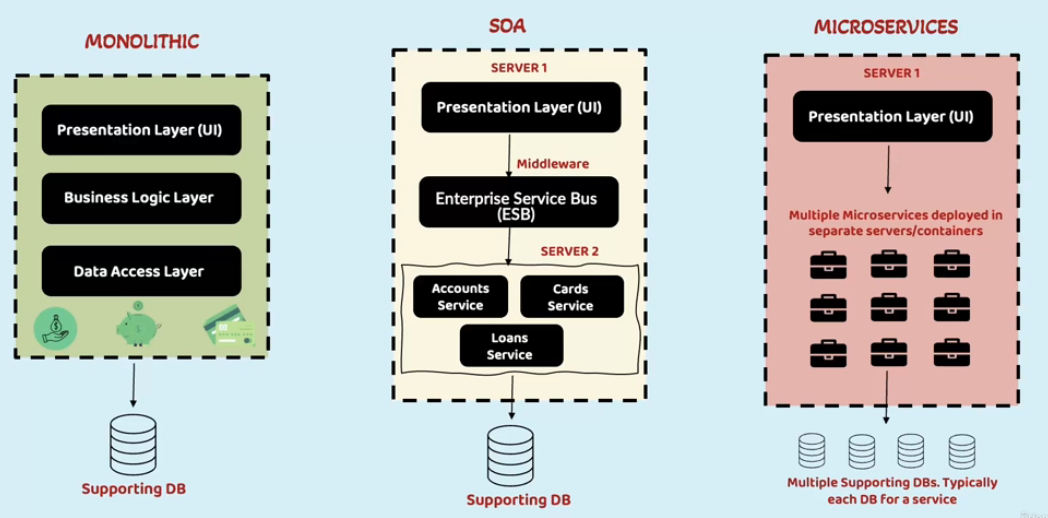
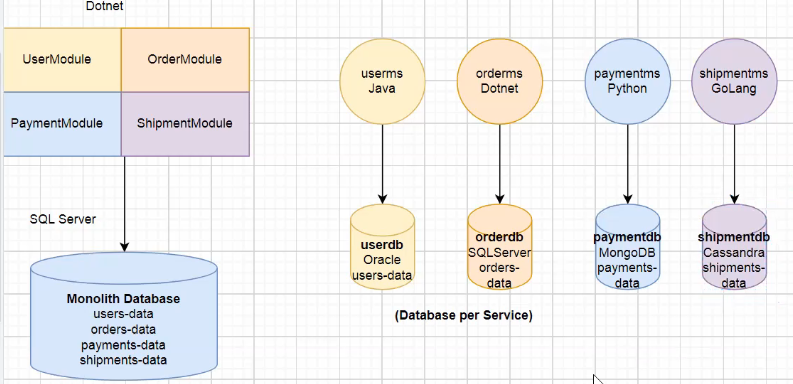
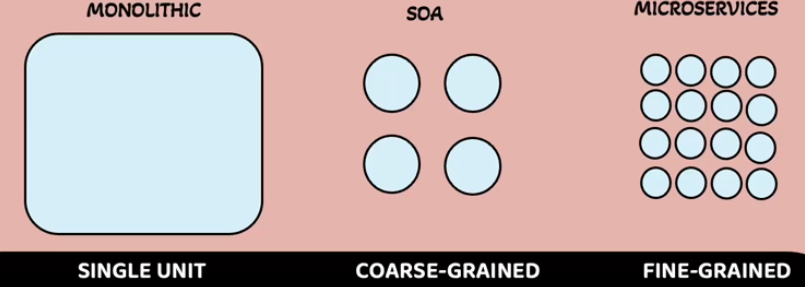
What is problem with SOA service oriented architecture approach?

In this we should use ESB- enterprise service bus, so to avoid ESB we deprecated this approach

Here when UI wants to call service it should go via ESB, why ESB when we can directly call service, here ESB is waste hence we are not following this approach



When to go for micro services

**Identifying the boundaries:-**

If we have independent functionality then only we can keep it as separate,

, else don’t un-necessarily split the applications and make HTTP rest calls

incase of independent functionality, we have to separate, we should follow DDD – domain driven design

Ex:- DIRS is not independent, it’s purely dependent on DSOS , DIRS will just make RESTful web service call to DSOS , As DIRS is dependent on DSOS this design is bad design

Ex:- consider a bank application which have many independent functionalities like

Loans app, money transfer, Forex, insurances all are independent so if these are in same app, then if we have any change in any apart, entire app must be down

Credit card, loans, if we develop as monolithic, if all functionalities are in same application,

For code change in loans, we have to down entire application and test all,

Ex:2

If that business functionality is independent only then we should separate

If it is a big application with different separate individual functionality, then split

Like ultimatix contains RIGHTS, GESS, leaves, TAX related, project workbench…. Many many,

When the functionality is separate then make it as separate microservices,

Case 1:- uber

Uber has many modules like –users, Bill, payment, notifications, Driver, GPS,

All rest api’s must contact via api gateway

While splitting ms

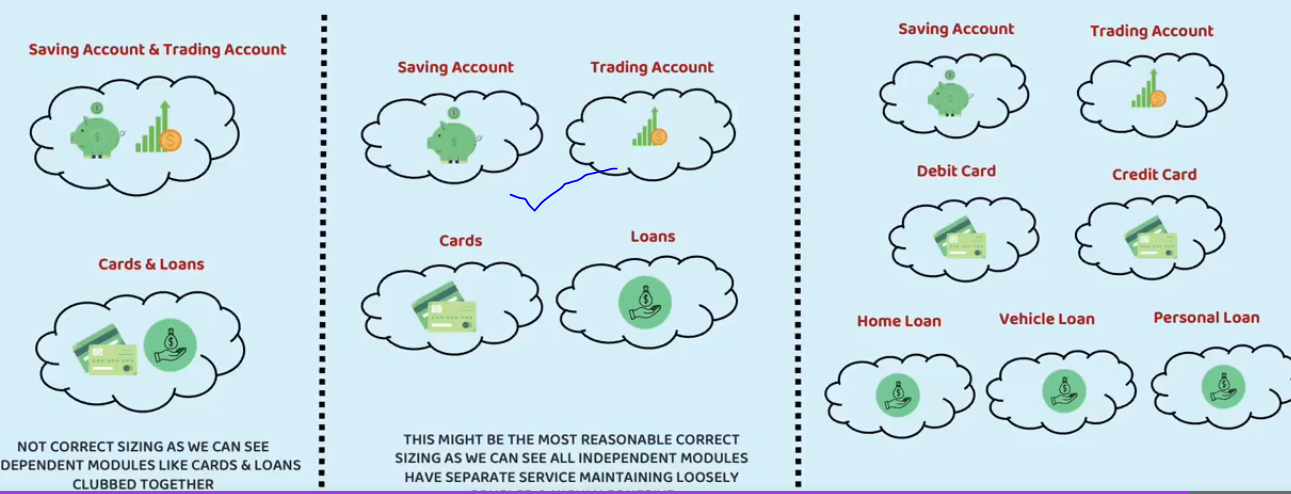
1. Dont unnecessarily split and waste ur time in making rest calls, if they are independent then only split

Ex:- DIRS, DSOS, there were split un necessarily and always making rest calls they are tightly coupled

Note:-

Un-necessary splitting will cause operational or infrastrucutre overhead, if there are too many micro services managing them is very difficult

1. Analyse if there is any tightly coupled dependencies, if tight coupled , if they are totally dependent then don’t split
2. when u design ms, design/ split based on independent business functionality and how often u are going to use it, if u just keep on splitting based on individual functionality without how often u are using its waste, if u just split if ur not going to use it then it will be wasted



For minute business logic difference we shouldnot separate

3)

## What is a micro service?

Micro services is an approach for developing an appln as a suite of small services each communicating with light weight mechanisms (such as REST/ mq ) which are built around

Business capabilities and independently deployable machinery

1. Each service has specific design
2. All svcs should interact with each other using REST, Message based communication

### Advantages of micro services

1. Faster development time

Agile means ability to move quickly, fastly

Agile means reacting to the change, adaptive

Agile == faster, If u follow monolithic architecture, u cant be agile , u cant be faster

1. Scaling Horizontally or vertically a small ms component is easy because it requires less ram

If it is monolithic, to horizontally scale that it needs a separate computer with huge RAM

1. Less downtime- no need to down entire application

Ex:- loans, credit card issuing both are independent functionality so split

In case of independent functionality if we split into 2 apps, because of change in 1 place, we don’t need to shutdown entire application, just shutdown 1 small ms alone

1. Technology diversity-If u split u can code each appln in any kind of language, java ,.net, python
2. Here the services are re-usable , so that other appln can use these services

### Cons of micro services

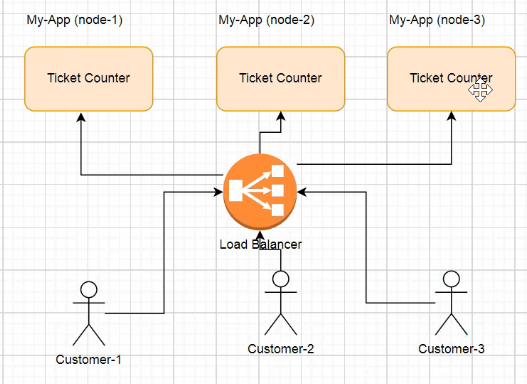
1. Inter service communication-(latency, failures)when app are decomposed, If 1 app wants to talk to another application, we should use http REST calls, there is a chance another app can be down and bec of this latency will be there, in case of monolithic we just need to create object of that class and call that method
2. Transaction management – apply global transaction management on 2-diff dbs is difficult
3. Taking 1 db per ms is not at all recommended –
4. Too many ms leads to operational overhead, maintaining too many ms needs more infrastructure
5. In case of too many ms , security logics needs to be implemented for all services, whereas in monolithic since app is 1 , we will apply security for only for 1 app
6. Too many ms, needs more network latency, bec 1 response may come from multiple ms

Reference url’s

<https://cloud.spring.io/spring-cloud-gateway/reference/html/>

Load balancers

Load balancer

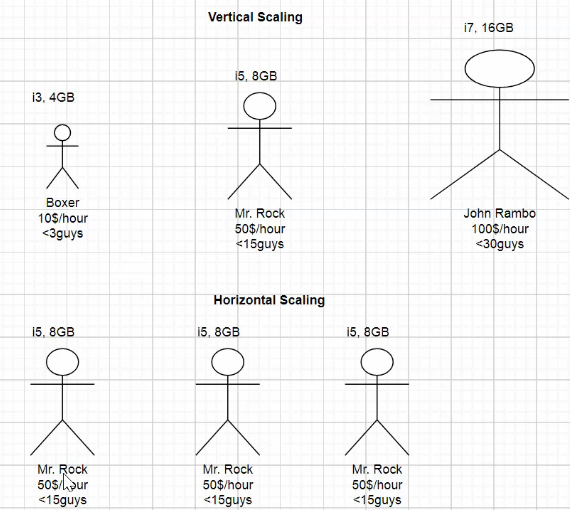


If u go via load balancer, when any request comes it will 1st goto load balancerand it will see whether node-3 is down or not,

If it is down then requests will not be routed to the svc /instance which is down

So always have a load balancer

Horizontal vs. vertical scaling



Horizontal scaling means , using same 3..n machines like hiring many guys for each technology

Vertical scaling means increasing ram on same computer, like 1 guy becoming full stack developer

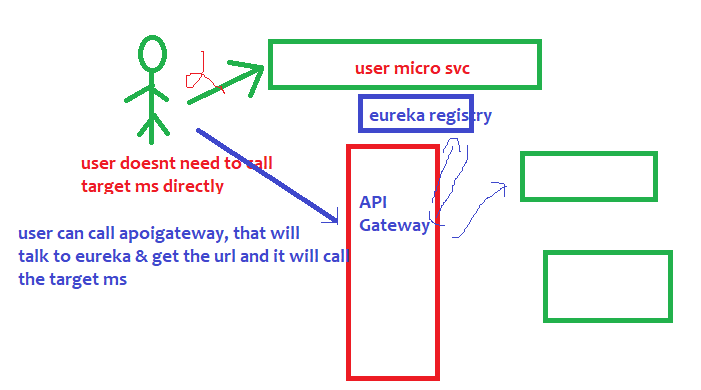
## Design patterns in micro services

1. Configuration
2. Fault tolerance- Resilience4j
3. Service discovery – Eureka

### **Aggregator design pattern**

In microservices the Aggregator Design Pattern is a service that receives a request, then that service makes requests of multiple services, combines the results and responds to the initiating request.

### Spr cloud API gateway pattern



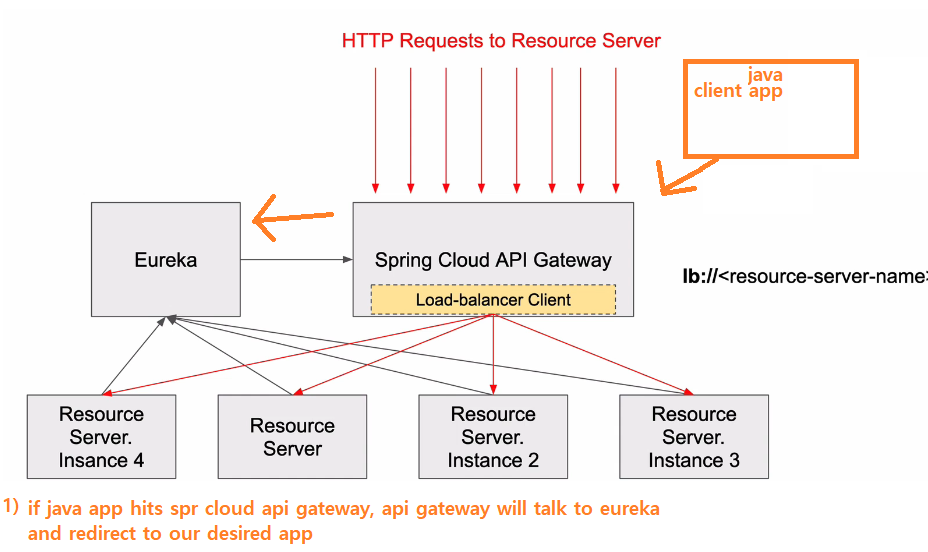
How API gateway know all the microservices URL’s & how it can route?

Because API gateway can interact with eureka registry how? Because API gateway is also a eureka client

#### Adv. of API Gateway

If u want to call 10 API’s u don’t need to hardcode all API URL’s, just call 1 API gateway URL and that will take or redirecting I felt this is 1 advantage

* **Single Entry Point:** An API gateway acts as a single entry point for all API requests, streamlining access and management. This eliminates the need to expose backend services directly, improving security and maintainability.
* **Traffic Routing:** The gateway intelligently routes incoming requests to the appropriate backend services based on factors like URL paths, headers, or other criteria. This simplifies client interactions and reduces complexity for developers.
* **Versioning and Deprecation:** API gateways handle API versioning, allowing you to manage different versions of your API simultaneously and smoothly transition clients to newer versions while deprecating older ones.



The main motto of this api gateway is , we should not call the target micro service directly,

|  |  |
| --- | --- |
| app name that is registered in eureka | All the urls of that application |
| Student-ms | 8081,8082 |
|  |  |

1. we should call only the api gateway url, api gateway will talk to the eureka and get the actual url

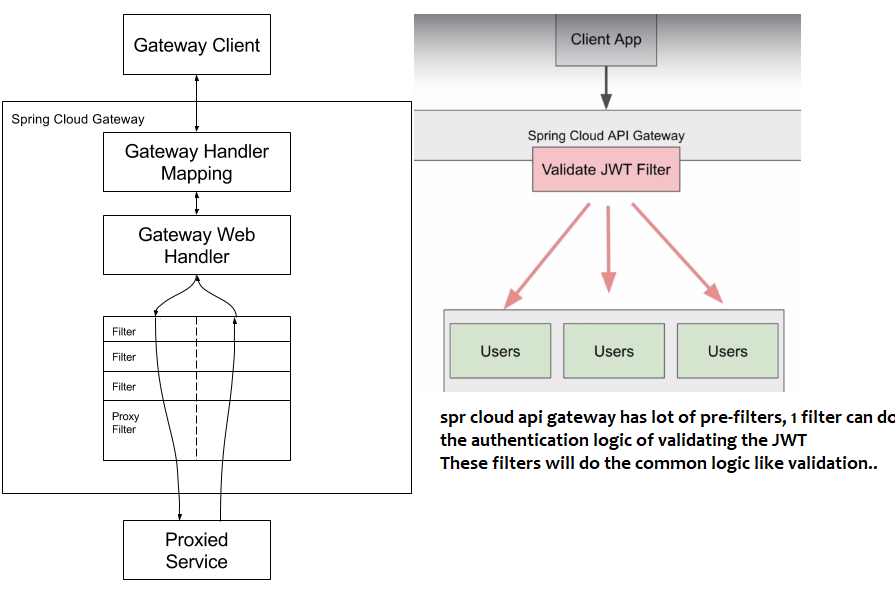
// **8082**- is the port of api gateway

// **student-ms** is the name of student ms that is registered in eureka

**localhost://8082/student-ms/getAllStudents**

1. and api gateway will make actual call to target ms via some prefilters.
2. API gateway have lot of pre,post filters to do the validation – we can check whether header is present or not
3. API gateway will call some pre-filters and then it route the request to actual ms
4. When actual ms was hit, then post filters will be executed then response will be sent back to client
5. In api gateway, we can validate the JWT token
6. Pre filters are those filters which will be execute before routing the request to target ms





The above diagram says, when we call api gateway, it will internally have lot of pre-filters &post filters

The Request before reaching the target ms it will hit all the pre-filters and then it will the target ms

In those pre filters –one pre filter can validate the JWT token



We wrote the above code in API gateway to validate whether the header contains any header named Authorization or not

Predicates are nothing but conditions, if those conditions are satisfied then API gateway will route the request to target ms else it won’t redirect

Here 2 predicates means 2 condition should satisfy

The 3rd predicate says, it will check for a header called Authorization & it should have a value called Bearer

#### Sample APi gateway properties

// give unique id for each flow

spring.cloud.gateway.routes[0].id=users-status-check

//if all predicates/conditions are success then we will redirect to this target uri

spring.cloud.gateway.routes[0].uri = <http://localhost:8081>

or // in below ( lb means load balancer followed by name is the name of app with which it is registered with eureka server) it will ask eureka server and gets the least load bal url if it have multiple instances

spring.cloud.gateway.routes[0].uri = lb://users-ws

// these are the conditions which needs to be validated, this means if below path is same then

spring.cloud.gateway.routes[0].predicates[0]=Path=/users/status/check

spring.cloud.gateway.routes[0].predicates[1]=Method=GET

spring.cloud.gateway.routes[0].predicates[2]=Header=Authorization, Bearer (.\*)

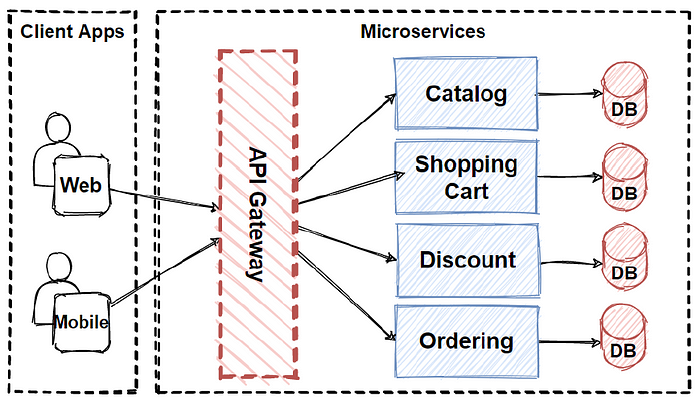
spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie

spring.cloud.gateway.routes[0].filters[1]=RewritePath=/users-ws/(?<segment>.\*), /$\{segment}

spring.cloud.gateway.routes[0].filters[2]=AuthorizationHeaderFilter

My properties which I kept

spring.application.name=ApiGatewayAsEurekaClient  
eureka.client.service-url.default-zone=http://localhost:8761/eureka  
server.port=8083  
  
*#Now if someone calls api gateway it will redirect to original application*spring.cloud.gateway.routes.id=oAuthResourceServer  
*#spring.cloud.gateway.routes[0].uri=http://localhost:80822*spring.cloud.gateway.routes[0].uri=lb://OAUTHRESOURCESERVER  
spring.cloud.gateway.routes[0].predicates[0]=Path=/hello  
spring.cloud.gateway.routes[0].predicates[1]=Method=GET  
spring.cloud.gateway.routes[0].predicates[2]=Header=Authorization, Bearer (.\*)



It is similar to the **facade pattern**of Object-Oriented Design, so it provides a **single entry point**to the APIs with encapsulating the underlying system architecture.

Here the api gateway is in between client and internal microservices, It routes the requests ,

It provides authentication, SSL termination, cache..

Chained pattern provides a single output, which is a combination of multiple chained outputs.

Event sourcing pattern – helps in tracking changes

Branch pattern – this pattern can simultaneously process the requests from 2 or more services

**Build-In Predicate Factories**

#### **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/#gateway-route-filters)

**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/#gateway-route-filters)

**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://example.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://example.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/)

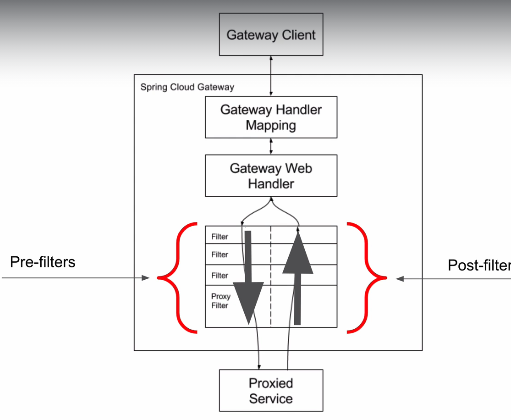
**Gateway Filters**

A list of API Gateway filters is very long to be included in a single lecture. Below are direct links to gateway filters published in Spring Cloud API Gateway documentation.

* <https://cloud.spring.io/spring-cloud-gateway/reference/html/#gatewayfilter-factories>
* <https://cloud.spring.io/spring-cloud-gateway/reference/html/#global-filters>
* <https://cloud.spring.io/spring-cloud-gateway/reference/html/#httpheadersfilters>

You can bookmark these pages in your browser for quick access.

Global pre-filters & Global post filters



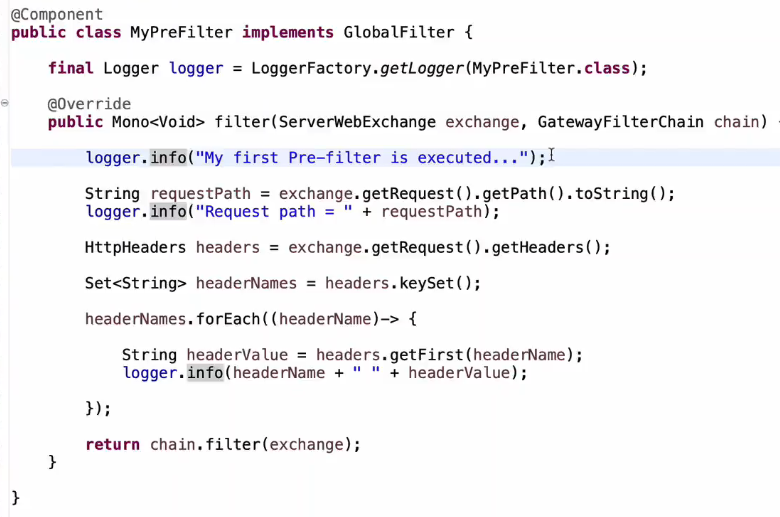
Pre-filters are those filters which will be executed before invoking actual api & post filters are those

Filters which will be invoked after hitting actual api

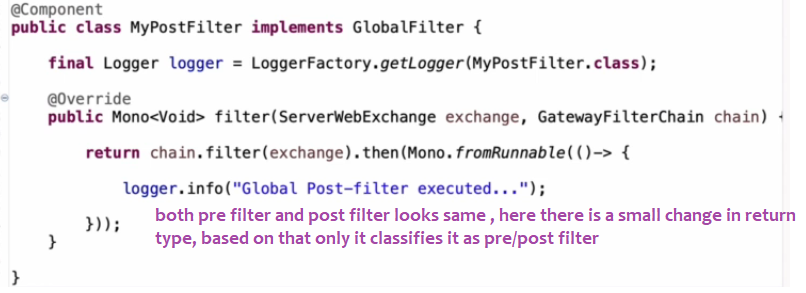
1. Global pre-filter: this pre-filter will be executed before actual ms invocation

Both pre-filter and post filter looks same by implementing GlobalFilter

Based on implementation, spring will decide which one to consider as pre-filter and which one to consider as post filter



1. Global post filter



1. Both pre,post filters at once



