

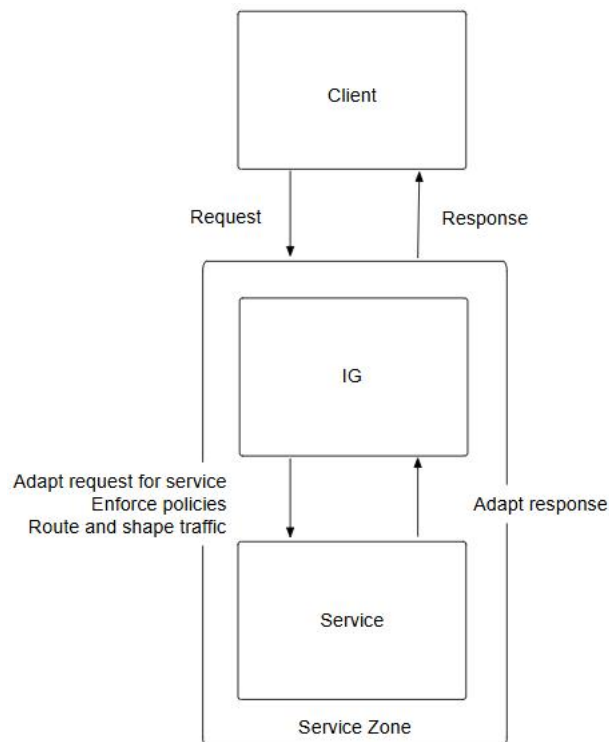
SpringCloud Gateway

IG as a Proxy

Many organizations have existing services that cannot easily be integrated into newer architectures. Similarly, many existing client applications cannot communicate with services. This section describes how IG acts as an intermediary, or proxy, between clients and services.

IG as a reverse proxy

IG as a reverse proxy server is an intermediate connection point between external clients and internal services. IG intercepts client requests and server responses, enforcing policies, and routing and shaping traffic. The following image illustrates IG as a reverse proxy:



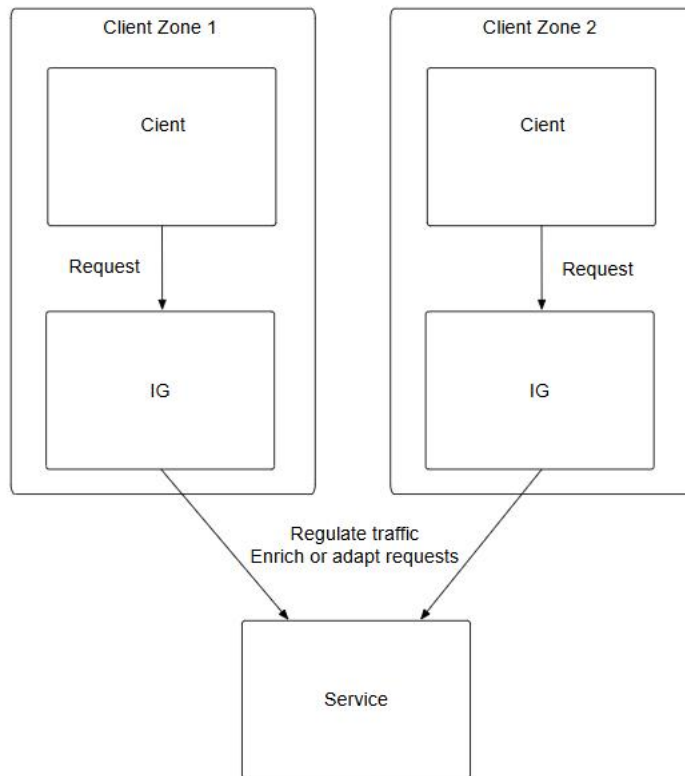
IG provides the following features as a reverse proxy:

- ✓ Access management integration
- ✓ Application and API security
- ✓ Credential replay
- ✓ OAuth 2.0 support
- ✓ OpenID Connect 1.0 support
- ✓ Network traffic control
- ✓ Proxy with request and response capture
- ✓ Request and response rewriting
- ✓ SAML 2.0 federation support
- ✓ Single sign-on (SSO)

SpringCloud Gateway

IG as a forward proxy

In contrast, IG as a forward proxy is an intermediate connection point between an internal client and an external service. IG regulates outbound traffic to the service, and can adapt and enrich requests. The following image illustrates IG as a forward proxy:



IG provides the following features as a forward proxy:

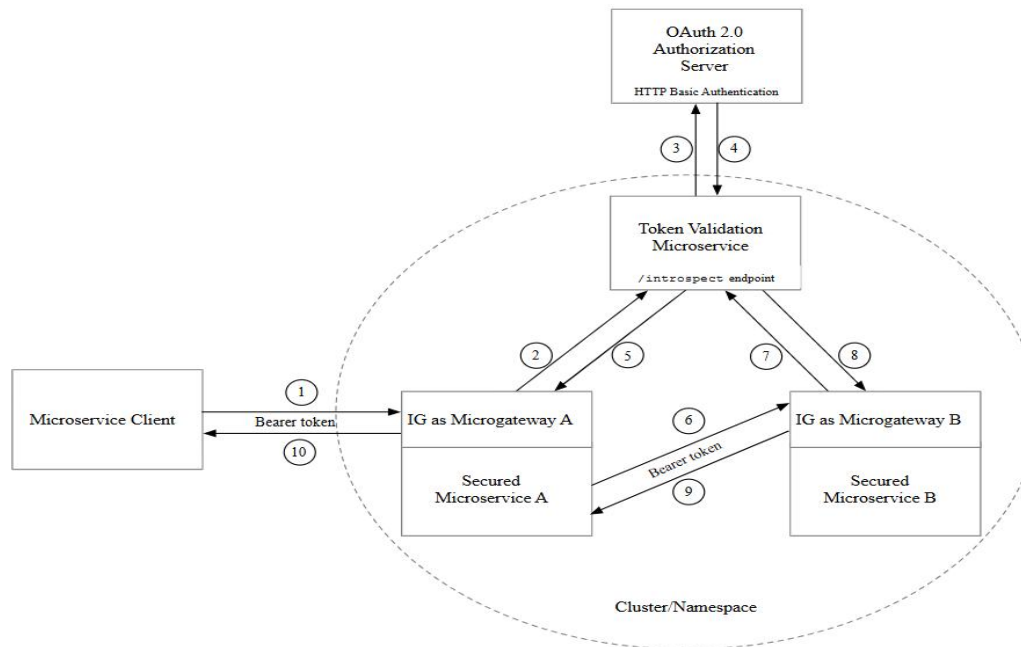
- ✓ Addition of authentication or authorization to the request
- ✓ Addition of tracer IDs to the requests
- ✓ Addition or removal of request headers or scopes

IG as a microgateway

IG is optimized to run as a microgateway in containerized environments. Use IG with business microservices to separate the security concerns of your applications from their business logic. For example, use IG with the Token Validation Microservice to provide access token validation at the edge of your namespace.

For an example, refer to IG as a microgateway. The following image illustrates the request flow in an example deployment:

SpringCloud Gateway



The request is processed in the following sequence:

1. A client requests access to Secured Microservice A, providing a stateful OAuth 2.0 access token as credentials.
2. Microgateway A intercepts the request, and passes the access token for validation to the Token Validation Microservice, using the /introspect endpoint.
3. The Token Validation Microservice requests the Authorization Server to validate the token.
4. The Authorization Server introspects the token, and sends the introspection result to the Token Validation Microservice.
5. The Token Validation Microservice caches the introspection result, and sends it to Microgateway A, which forwards the result to Secured Microservice A.
6. Secured Microservice A uses the introspection result to decide how to process the request. In this case, it continues processing the request. Secured Microservice A asks for additional information from Secured Microservice B, providing the validated token as credentials.
7. Microgateway B intercepts the request, and passes the access token to the Token Validation Microservice for validation, using the /introspect endpoint.

SpringCloud Gateway

8. The Token Validation Microservice retrieves the introspection result from the cache, and sends it back to Microgateway B, which forwards the result to Secured Microservice B.
9. Secured Microservice B uses the introspection result to decide how to process the request. In this case it passes its response to Secured Microservice A, through Microgateway B.
10. Secured Microservice A passes its response to the client, through Microgateway A.

Spring Cloud Gateway

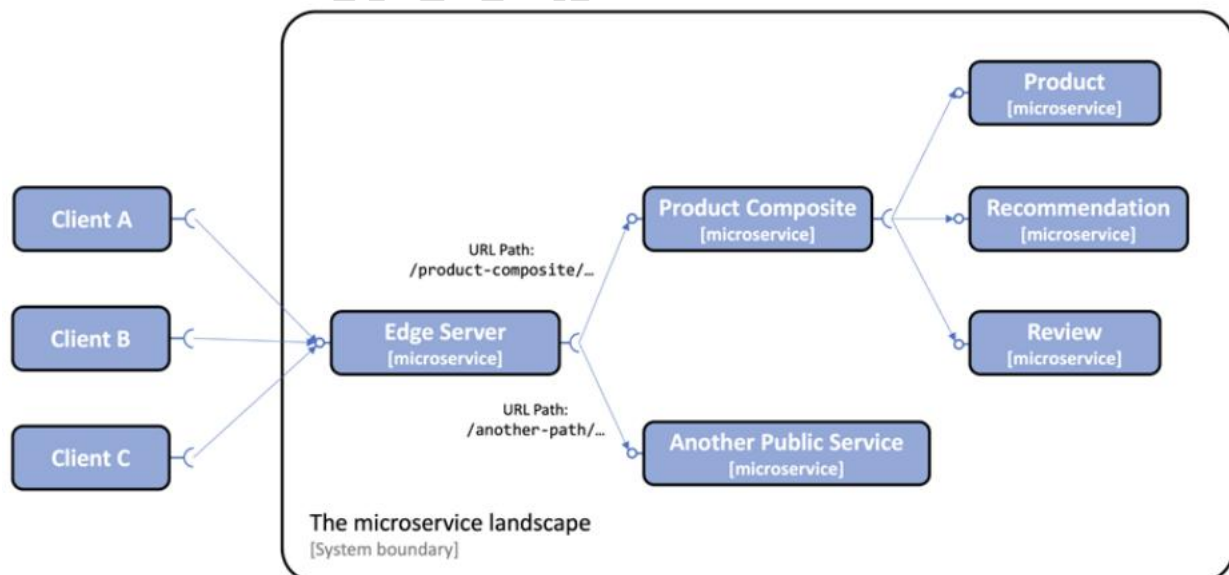
Initially, Spring Cloud used Netflix Zuul v1 as its edge server. Since the Spring Cloud Greenwich release, it's recommended to use **Spring Cloud Gateway** instead.

Spring Cloud Gateway comes with similar support for critical features, such as URL path-based routing and the protection of endpoints via the use of **OAuth 2.0** and **OpenID Connect (OIDC)**.

One important difference between Netflix Zuul v1 and Spring Cloud Gateway is that Spring Cloud Gateway is based on non-blocking APIs that use Spring 6, Project Reactor, and Spring Boot 3, while Netflix Zuul v1 is based on blocking APIs.

This means that Spring Cloud Gateway should be able to handle larger numbers of concurrent requests than Netflix Zuul v1, which is important for an edge server that all external traffic goes through.

The following diagram shows how all requests from external clients go through Spring Cloud Gateway as an edge server. Based on URL paths, it routes requests to the intended microservice:



SpringCloud Gateway

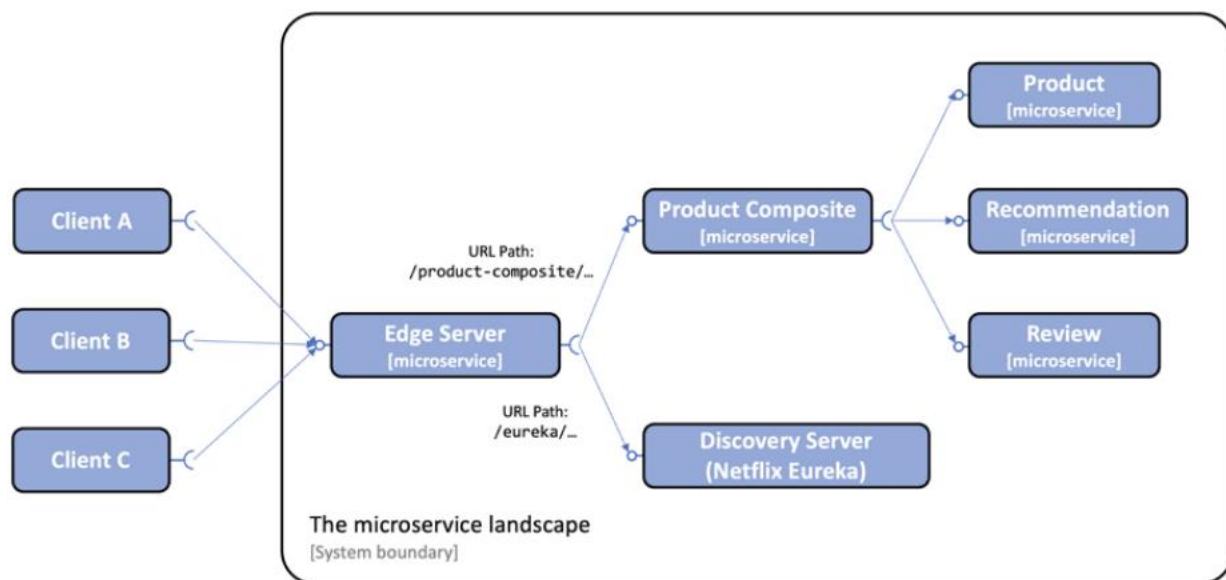
In the preceding diagram, we can see how the edge server will send external requests that have a URL path that starts with `/product-composite/` to the Product Composite microservice. The core services Product, Recommendation, and Review are not reachable from external clients.

With Spring Cloud Gateway introduced, let's see how Spring Cloud can help to manage the configuration of a system landscape of microservices.

Adding an edge server to our system landscape

In this section, we will see how the edge server is added to the system landscape and how it affects the way external clients access the public APIs that the microservices expose.

All incoming requests will now be routed through the edge server, as illustrated by the following diagram:



As we can see from the preceding diagram, external clients send all their requests to the edge server. The edge server can route the incoming requests based on the URL path.

For example, requests with a URL that starts with `/product-composite/` are routed to the product composite microservice, and a request with a URL that starts with `/eureka/` is routed to the discovery server based on Netflix Eureka.

SpringCloud Gateway

Setting up Spring Cloud Gateway:

Step-1:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <!--<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway-mvc</artifactId>
  </dependency>-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
...

```

Step-2: Application.java

```
package com.acma.properties;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class AcmaIdentityGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(AcmaIdentityGatewayApplication.class, args);
    }
}

```

Step-3: application.yml

SpringCloud Gateway

```
server:
  port: ${GATEWAY_PORT:5053}

management:
  endpoints:
    web:
      exposure:
        include: "*"

  endpoint:
    gateway:
      enabled: true

eureka:
  client:
    register-with-eureka: false
    service-url:
      default-zone: http://${ACMA_REGISTRY_HOST:localhost}:${ACMA_REGISTRY_PORT:8761}/eureka/
  instance:
    hostname: ${ACMA_GATEWAY_REGISTRY_HOST:localhost}
    #prefer-ip-address: true
```

Configuring the Routes

1. Route:

- The basic building block of the gateway is the Route. It represents the url to which incoming requests will be forwarded.
- A route is defined by ID, a destination URI, a collection of predicates, and a collection of filters either by the java code(Programatic) or in the configuraton(Declarative).
- A route is matched against an aggregate **Predicate**.

2. Predicate:

- The Predicate is a java8 functional interface, that has the input type as **ServerWebExchange**. The ServerWebExchange is a contract for an HTTP request-response intercation.
- This provides access to the HTTP request and response. It also exposes Server Side Processing for the properties and features such as request attributes. **The Predicate allows matching anything from the HTTP request, such as a request parameters, query or headers by applying the condition.**
- Once the Predicate is matched, the incoming request is forwarded to a perticular route URL.

3. Filter:

- The Filter is an instance of a GatewayFilter which is constructed with a specific factory.
- In the filter, we can modify requests and responses either before sending a request or after receving the response from the downstream.

SpringCloud Gateway

Programatic Approach-RouteLocator

The gateway service uses the GatewayHandler to resolve the route configurations by using the RouteLocator. We can configure the RouteLocator bean in our acma api gateway as shown below:

```
package com.acma.properties.config;

import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import lombok.extern.slf4j.Slf4j;

@Configuration
@Slf4j
public class AcmaRoutesConfig {

    @Bean
    public RouteLocator configureRoute(RouteLocatorBuilder builder) {
        log.info("AcmaRoutesConfig-->builder");
        return builder
            .routes()
            .route("02-authnz-route", (route->route
                .path("/token")
                .uri("lb://ACMA-AUTHN-AUTHZ-SERVICE")))
            .build();
    }
}
```

```
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

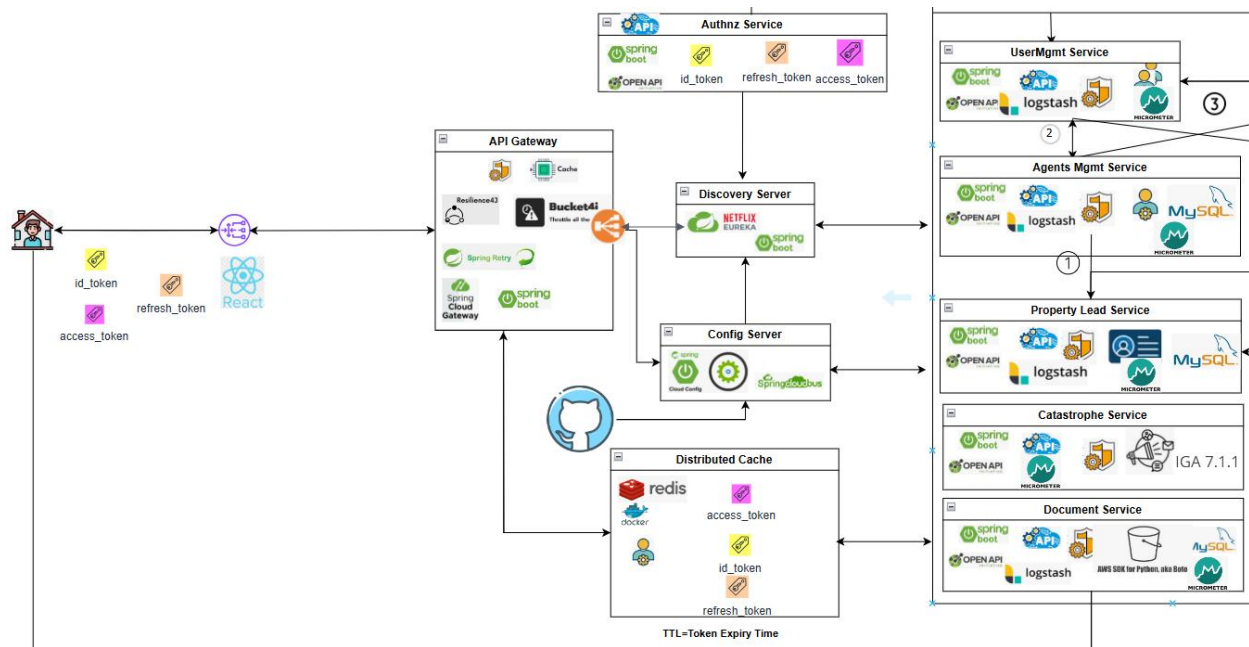
import lombok.extern.slf4j.Slf4j;

@Configuration
@Slf4j
public class AcmaRoutesConfig {

    @Bean
    public RouteLocator configureRoute(RouteLocatorBuilder builder) {
        log.info("AcmaRoutesConfig-->builder");
        return builder
            .routes()
            .route("02-authnz-route", (route->route
                .path("/token")
                .uri("lb://ACMA-AUTHN-AUTHZ-SERVICE")))
            .build();
    }
}
```

The API gateway is positioned between the services and the client as shown below:

SpringCloud Gateway



Once request arrives, the gateway handles it and then utilize the predicate to filter and direct the requests to their appropriate destinations.

Lets try to access the <http://localhost:5053/token>, now we need to keep in mind that Acma-Authnz service has multiple instances running, hence using the static URL will not work. To address this, we must request the load balancer by adding the lb://ACMA-AUTHN-AUTHZ-SERVICE as shown below:

```
route->route
    .path("/token")
    .uri("lb://ACMA-AUTHN-AUTHZ-SERVICE")
```

This dynamic routing allows the API server to route to one of the available instances though the service is running in multiple instances. A RouteLocator can have multiple routes specified as shown below:

SpringCloud Gateway

```
@Bean
public RouteLocator configureRoute(RouteLocatorBuilder builder) {
    log.info("AcmaRoutesConfig--->builder");
    return builder
        .routes()
        .route("02-authnz-route", (route->route
            .path("/token")
            .uri("lb://ACMA-AUTHN-AUTHZ-SERVICE")))

        .route("03a-usermgmt-route", (route->route
            .path("/swagger-ui.html")
            .uri("lb://ACMA-USERMGMGT-SERVICE")))

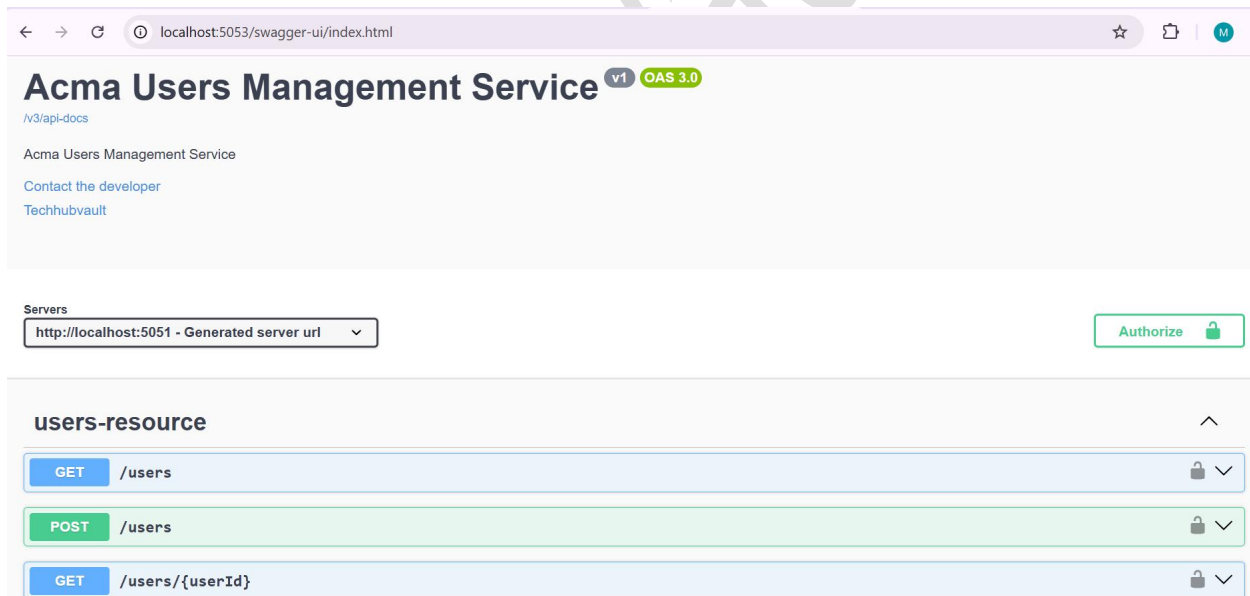
        .route("03b-usermgmt-route", (route->route
            .path("/swagger-ui/**")
            .uri("lb://ACMA-USERMGMGT-SERVICE")))

        .route("03c-usermgmt-route", (route->route
            .path("/v3/api-docs/**")
            .uri("lb://ACMA-USERMGMGT-SERVICE")))

        .build();
}
```

Lets restart the acma identity gateway and access the endpoint <http://localhost:5053/swagger-ui.html>

We will get the response as expected as shown below:



← → ↻ ⓘ localhost:5053/swagger-ui/index.html ☆ ⓘ M

Acma Users Management Service v1 OAS 3.0

[/v3/api-docs](#)

Acma Users Management Service

[Contact the developer](#)

[Techhubvault](#)

Servers

Authorize

users-resource

GET	/users	🔒	▼
POST	/users	🔒	▼
GET	/users/{userId}	🔒	▼

Declarative Approach-application.yml

We can use a configuration file as well rather than the bean configuration for routing. To achieve this, we can update the application.yml file as shown below:

SpringCloud Gateway

```
1@spring:
2@ application:
3    name: acma-identity-gateway
4
5    cloud:
6        gateway:
7            discovery:
8                locator:
9                    enabled: true
10
11        routes:
12
13            #Acma Authnz Service Routes
14            - id: 02-authnz-route
15              uri: lb://ACMA-AUTHN-AUTHZ-SERVICE
16              predicates:
17                  - Path=/token
18
19            - id: 03a-usermgmt-route
20              uri: lb://ACMA-USERMGMGT-SERVICE
21              predicates:
22                  - Path=/swagger-ui.html
23
24            - id: 03b-usermgmt-route
25              uri: lb://ACMA-USERMGMGT-SERVICE
26              predicates:
27                  - Path=/swagger-ui/**
28
29            - id: 03c-usermgmt-route
30              uri: lb://ACMA-USERMGMGT-SERVICE
31              predicates:
32                  - Path=/v3/api-docs/**
```

Note:

We must comment the @Bean Configuration within the AcmaRoutesConfig.java.

Now when restart the api gateway and when we can access the endpoint

<http://localhost:5053/swagger-ui.html> we can see the same response as shown below:

SpringCloud Gateway

localhost:5053/swagger-ui/index.html

Swagger
Supported by SMARTBEAR

/v3/api-docs

Explore

Acma Users Management Service v1 OAS 3.0

/v3/api-docs

Acma Users Management Service

[Contact the developer](#)

[Techhubvault](#)

Servers

http://localhost:5051 - Generated server url

Authorize

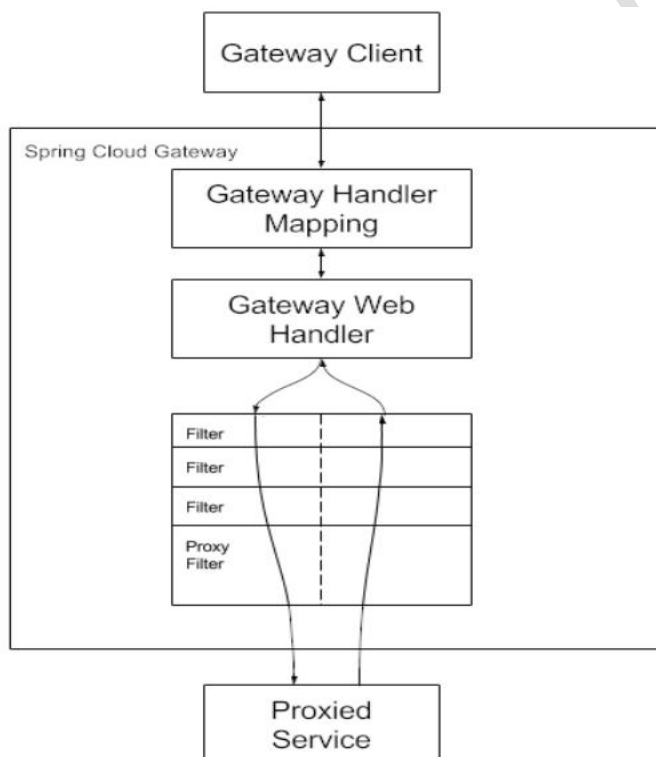
users-resource

GET /users

POST /users

Internal working of Gateway Routing

The following diagram provides a high-level overview of how Spring Cloud Gateway works:



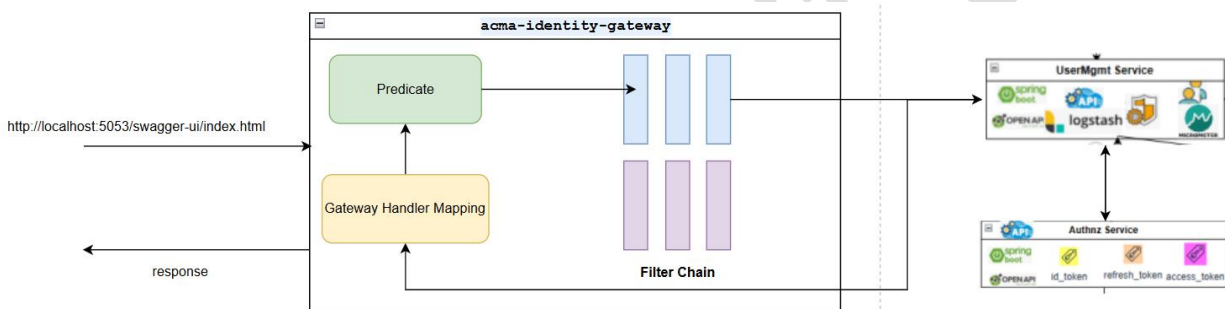
- Clients make requests to Spring Cloud Gateway.

SpringCloud Gateway

- If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler.
- This handler runs the request through a filter chain that is specific to the request.
- The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent.
- All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.

Spring Cloud Gateway employs the **Spring WebFlux** handler mapping to compare routes, which consists of a range of pre-existing route predicate factories.

These Predicates Match the different attributes of the incoming HTTP request. On ePredicate can combine multiple route predicate factories using logical and statements.



Flow:

- When the client requests spring cloud gateway, the first component in the act is Gateway Handler Mapping. It determines whether the incoming request matches a route using the route configured with the Predicate. If it matches, then it is directed to the **Gateway Web Handler**.
- The handler propagates the requests through a chain of filters. This chain is specific to the request and handler decides which all filters will be applied.
- Filters can either executed before sending the request to the downstream microservice or applied after obtaining the response from the downstream microservices.
- Depending up on whether the filter applied before or after they are categorized as pre-filters or post-filters. One can decide whether to execute only pre-filter , only post-filter or even both.
- Once all the pre-filters are executed then the proxy request is made to the service and a response is produced.
- This response now may again pass through the post-filter chain and the final response will be sent to the client.

SpringCloud Gateway

Commonly used Predicates:

1. PrefixPath GatewayFilter Factory:

PrefixPath GatewayFilter Factory prefixes some value to the path for all the matching requests. It takes prefix as parameter. The following configuration adds a PrefixPath to all matching requests.

```
routes:

#Acma Authnz Service Routes
- id: 02-authnz-route
  uri: lb://ACMA-AUTHN-AUTHZ-SERVICE
  predicates:
    - Path=/token
  filters:
    - PrefixPath=/acma
```

2. RewritePath GatewayFilter Proxy:

In some cases, for security reasons, it may be necessary to secure the actual url by hiding it with some proxy url. The RewritePath GatewayFilter factory facilitates us to write such a path. It takes an actual path and replacement path as a parameter in the format of the regex expression. This enables a flexible way to rewrite the request path.

```
routes:

#Acma Authnz Service Routes
- id: 02-authnz-route
  uri: lb://ACMA-AUTHN-AUTHZ-SERVICE
  predicates:
    - Path=/login
  filters:
    #- PrefixPath=/acma
    - RewritePath=/login/(?<segment>.*),/token/${segment}
```

3. AddRequestHeader GatewayFilter Factory

Global Filters:

The global filters are the special filters as they are applied to all the routes depending up on some conditions. The GlobalFilter interface has the same signature as that of the GatewayFilter.

Once the Gateway handler determines that the request matches a route using the predicate, then the framework passes that request through a chain of filters.

Few of these filters will execute the logic before the request is propagated to the downstream service and others may be applied after the response received from the service.

SpringCloud Gateway

This behaviour is generally dependent on the conditions that have been configured for the route which are applied before or after the request has been processed. This is referred to as **pre-processing** and **post-processing** of the request.

Lets write a custom global filter `AcmaLoggingFilter.java` to add logging to perform pre-processing of the request. This class should implement the **GlobalFilter** interface and override the **filter()** method.

The filter() method looks as :

```
@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

    //pre processing logic

    return chain.filter(exchange);

    //post processing logic
}
```

Any operation that we perform through the code before invoking the **chain.filter()** method is considered as pre-processing of the request. In this process, we can execute a different tasks like modifying the request body, adding some request headers to the down stream service, and so on.

Also, after receiving the response from the downstream, we can add the headers, modifying the response body, and so on before it sending to the client. This logic will be written after the invocation of filter() method.

To perform the pre-processing of the incoming request, we can customize the behaviour of the filter() method by overriding it. In the below example, lets read the value of **JSESSIONID** header from the request and then send this value to the another header called **AcmaJgCookie** to the downstream.

SpringCloud Gateway

```
@Component
@Slf4j
public class AcmaLoggingFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        log.info("Global Pre Filter Executed");
        ServerHttpRequest request = exchange.getRequest();
        log.info("Path: "+request.getPath());
        log.info("address "+request.getRemoteAddress());
        log.info("method" +request.getMethod());
        log.info("URI "+request.getURI());
        log.info("Headers "+request.getHeaders());

        ServerHttpRequest newRequest = exchange.getRequest().mutate()
            .headers(httpHeaders->httpHeaders.add("AcmaIgCookie", request.getHeaders().get("JSESSIONID")
            .get(0))).build();

        //pre processing logic

        return chain.filter(exchange);

        //post processing logic
    }
}
```

It is important to remember that to annotate the class with the `@Component` annotation to register it as a bean in the spring core container. We can also register it by adding the method returning `GlobalFilter` annotated by `@Bean` annotation to register the custom filters as shown below:

```
@Bean
public GlobalFilter acmaLoggingFilter() {
    return new AcmaLoggingFilter();
}
```

Now let's write the code for post-processing. We will try to add post-header in the response as shown below:

```
@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

    log.info("Global Pre Filter Executed");
    ServerHttpRequest request = exchange.getRequest();
    log.info("Path: "+request.getPath());
    log.info("address "+request.getRemoteAddress());
    log.info("method" +request.getMethod());
    log.info("URI "+request.getURI());
    log.info("Headers "+request.getHeaders());

    ServerHttpRequest newRequest = exchange.getRequest().mutate()
        .headers(httpHeaders->httpHeaders.add("AcmaIgCookie", request.getHeaders().get("JSESSIONID")
        .get(0))).build();

    //pre processing logic

    return chain.filter(exchange).then(Mono.fromRunnable(()->{
        ServerHttpResponse httpResponse = exchange.getResponse();
        HttpHeaders headers = httpResponse.getHeaders();
        headers.add("Location", "http://localhost:5053/login");
        log.info("Post Global Filter");
    }));

    //post processing logic
}
```

Overriding the GatewayFilter and GlobalFilter

SpringCloud Gateway

When any incoming request matches a route, then the filtering web handler adds all the instances of the **GlobalFilter** and all route-specific instances of the **GatewayFilter** to the chain. This chain is recognized as the filter chain.

Now `org.springframework.core.Ordered` interface will sort this combined after chain. To Order the filters we need to implement the `getOrder()` method. Any filter with the highest precedence will be the first in pre phase. In the post phase, the filter chain is executed in reverse order. So the first filter will be the last filter in the post phase.

```
@Component
@Slf4j
public class AcmaLoggingFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        log.info("Global Pre Filter Executed");
        ServerHttpRequest request = exchange.getRequest();
        log.info("Path: " + request.getPath());
        log.info("address " + request.getRemoteAddress());
        log.info("method" + request.getMethod());
        log.info("URI " + request.getURI());
        log.info("Headers " + request.getHeaders());

        // ServerHttpRequest newRequest = exchange.getRequest().mutate()
        //                                     .headers(httpHeaders->httpHeaders.add("AcmaIgCookie", request.getHeaders().get("JSESSIONID"))
        //                                     .get(0)).build();

        // pre processing logic

        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
            ServerHttpResponse httpResponse = exchange.getResponse();
            HttpHeaders headers = httpResponse.getHeaders();
            headers.add("Location", "http://localhost:5053/login");
            log.info("Post Global Filter");
        }));

        // post processing logic
    }

    @Override
    public int getOrder() {
        // TODO Auto-generated method stub
        return -1;
    }
}
```

This filter being the lowest in the number will be executed first in the pre-phase and last in the post-phase.

Ref: <https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/gatewayfilter-factories>

Gateway as Resource Server

Step-1:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

SpringCloud Gateway

Step-2:

application.yml × acma-identity-gateway/pom.xml Audio VI

```
1 spring:
2   application:
3     name: acma-identity-gateway
4   security:
5     oauth2:
6       resourceserver:
7         jwt:
8           issuer-uri: http://localhost:8081/realms/acma
```

Step-3:

```
logging:
  file:
    name: gateway.logs
    path: /var/log
  level:
    root: debug
```