Spring Cloud Gateway

Table of Contents

{spring-cloud-version}

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

Chapter 1. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-gateway`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

If you include the starter, but you do not want the gateway to be enabled, set `spring.cloud.gateway.enabled=false`.

Spring Cloud Gateway is built on Spring Boot 2.x, Spring WebFlux, and Project Reactor. As a consequence, many of the familiar synchronous libraries (Spring Data and Spring Security, for example) and patterns you know may not apply when you use Spring Cloud Gateway. If you are unfamiliar with these projects, we suggest you begin by reading their documentation to familiarize yourself with some of the new concepts before working with Spring Cloud Gateway.

Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work in a traditional Servlet Container or when built as a WAR.

Chapter 2. Glossary

- **Route**: The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
- **Predicate**: This is a Java 8 Function Predicate. The input type is a Spring Framework `ServerWebExchange`. This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filter**: These are instances of {github-code}/spring-cloud-gateway-server/src/main/java/org/springframework/cloud/gateway/filter/GatewayFilter.java[`GatewayFilter`] that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

Chapter 3. How It Works

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

[Spring Cloud Gateway Diagram] | *spring_cloud_gateway_diagram.png*

Clients make requests to Spring Cloud 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.

URIs defined in routes without a port get default port values of 80 and 443 for the HTTP and HTTPS URIs, respectively.

Chapter 4. Configuring Route Predicate Factories and Gateway Filter Factories

There are two ways to configure predicates and filters: shortcuts and fully expanded arguments. Most examples below use the shortcut way.

The name and argument names will be listed as `code` in the first sentance or two of the each section. The arguments are typically listed in the order that would be needed for the shortcut configuration.

4.1. Shortcut Configuration

Shortcut configuration is recognized by the filter name, followed by an equals sign (`=`), followed by argument values separated by commas (`,`).

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - Cookie=mycookie,mycookievalue
```

The previous sample defines the `Cookie` Route Predicate Factory with two arguments, the cookie name, `mycookie` and the value to match `mycookievalue`.

4.2. Fully Expanded Arguments

Fully expanded arguments appear more like standard yaml configuration with name/value pairs. Typically, there will be a `name` key and an `args` key. The `args` key is a map of key value pairs to configure the predicate or filter.

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - name: Cookie
          args:
            name: mycookie
            regexp: mycookievalue
```

This is the full configuration of the shortcut configuration of the `Cookie` predicate shown above.

Chapter 5. Route Predicate Factories

Spring Cloud Gateway matches routes as part of the Spring WebFlux `HandlerMapping` infrastructure. Spring Cloud Gateway includes many built-in route predicate factories. All of these predicates match on different attributes of the HTTP request. You can combine multiple route predicate factories with logical `and` statements.

5.1. 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:

Example 1. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - 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).

5.2. 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:

Example 2. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: before_route
        uri: https://example.org
        predicates:
        - 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).

5.3. The Between Route Predicate Factory

The `Between` route predicate factory takes two parameters, `datetime1` and `datetime2` which are java `ZonedDateTime` objects. 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:

Example 3. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: https://example.org
        predicates:
        - 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.

5.4. 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:

Example 4. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: https://example.org
        predicates:
        - Cookie=chocolate, ch.p
```

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

5.5. The Header Route Predicate Factory

The `Header` route predicate factory takes two parameters, the `header` 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:

Example 5. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: https://example.org
        predicates:
        - 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).

5.6. 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:

Example 6. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: https://example.org
        predicates:
        - 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 `GatewayFilter` factories

5.7. 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:

Example 7. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: https://example.org
        predicates:
        - Method=GET,POST
```

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

5.8. The Path Route Predicate Factory

The `Path` Route Predicate Factory takes two parameters: a list of Spring `PathMatcher patterns` and an optional flag called `matchTrailingSlash` (defaults to `true`). The following example configures a path route predicate:

Example 8. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: path_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment},/blue/{segment}
```

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

If `matchTrailingSlash` is set to `false`, then request path `/red/1/` will not be matched.

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 `GatewayFilter` factories

A utility method (called `get`) is available to make access to these variables easier. The following example shows how to use the `get` method:

```
Map<String, String> uriVariables = ServerWebExchangeUtils.getUriTemplateVariables
(exchange);

String segment = uriVariables.get("segment");
```

5.9. 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:

Example 9. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://example.org
        predicates:
        - Query=green
```

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

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://example.org
        predicates:
        - Query=red, gree.
```

The preceding route matches if the request contained a `red` query parameter whose value matched the `gree.` regexp, so `green` and `greet` would match.

5.10. 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:

Example 10. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: https://example.org
        predicates:
        - RemoteAddr=192.168.1.1/24
```

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

5.10.1. Modifying the Way Remote Addresses Are Resolved

By default, the RemoteAddr route predicate factory uses the remote address from the incoming request. This may not match the actual client IP address if Spring Cloud Gateway sits behind a proxy layer.

You can customize the way that the remote address is resolved by setting a custom `RemoteAddressResolver`. Spring Cloud Gateway comes with one non-default remote address resolver that is based off of the X-Forwarded-For header, `XForwardedRemoteAddressResolver`.

`XForwardedRemoteAddressResolver` has two static constructor methods, which take different approaches to security:

- `XForwardedRemoteAddressResolver::trustAll` returns a `RemoteAddressResolver` that always takes the first IP address found in the `X-Forwarded-For` header. This approach is vulnerable to spoofing, as a malicious client could set an initial value for the `X-Forwarded-For`, which would be accepted by the resolver.
- `XForwardedRemoteAddressResolver::maxTrustedIndex` takes an index that correlates to the number of trusted infrastructure running in front of Spring Cloud Gateway. If Spring Cloud Gateway is, for example only accessible through HAProxy, then a value of 1 should be used. If two hops of trusted infrastructure are required before Spring Cloud Gateway is accessible, then a value of 2 should be used.

Consider the following header value:

```
X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3
```

The following `maxTrustedIndex` values yield the following remote addresses:

`maxTrustedIndex`	**result**
[`Integer.MIN_VALUE`,0]	(invalid, `IllegalArgumentException` during initialization)
1	0.0.0.3
2	0.0.0.2
3	0.0.0.1
[4, `Integer.MAX_VALUE`]	0.0.0.1

The following example shows how to achieve the same configuration with Java:

Example 11. GatewayConfig.java

```
RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...

.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1")
.route("proxied-route",
    r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2")
)
```

5.11. 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:

Example 12. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
        predicates:
```

```
        - Weight=group1, 2
```

This route would forward ~80% of traffic to weighthigh.org and ~20% of traffic to weighlow.org

5.12. The XForwarded Remote Addr Route Predicate Factory

The `XForwarded Remote Addr` 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).

This route predicate allows requests to be filtered based on the `X-Forwarded-For` HTTP header.

This can be used with reverse proxies such as load balancers or web application firewalls where the request should only be allowed if it comes from a trusted list of IP addresses used by those reverse proxies.

The following example configures a XForwardedRemoteAddr route predicate:

Example 13. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: xforwarded_remoteaddr_route
        uri: https://example.org
        predicates:
        - XForwardedRemoteAddr=192.168.1.1/24
```

This route matches if the `X-Forwarded-For` header contains, for example, `192.168.1.10`.

Chapter 6. `GatewayFilter` Factories

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

For more detailed examples of how to use any of the following filters, take a look at the unit tests.

6.1. The `AddRequestHeader GatewayFilter` Factory

The `AddRequestHeader GatewayFilter` factory takes a `name` and `value` parameter. The following example configures an `AddRequestHeader GatewayFilter`:

Example 14. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        filters:
        - AddRequestHeader=X-Request-red, blue
```

This listing adds `X-Request-red:blue` header to the downstream request's headers for all matching requests.

`AddRequestHeader` is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddRequestHeader GatewayFilter` that uses a variable:

Example 15. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment}
        filters:
        - AddRequestHeader=X-Request-Red, Blue-{segment}
```

6.2. The AddRequestParameter GatewayFilter Factory

The AddRequestParameter GatewayFilter Factory takes a name and value parameter. The following example configures an AddRequestParameter GatewayFilter:

Example 16. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: https://example.org
        filters:
        - AddRequestParameter=red, blue
```

This will add red=blue to the downstream request's query string for all matching requests.

AddRequestParameter is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an AddRequestParameter GatewayFilter that uses a variable:

Example 17. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - AddRequestParameter=foo, bar-{segment}
```

6.3. The AddResponseHeader GatewayFilter Factory

The AddResponseHeader GatewayFilter Factory takes a name and value parameter. The following example configures an AddResponseHeader GatewayFilter:

Example 18. application.yml

```
spring:
  cloud:
    gateway:
```

```
      routes:
      - id: add_response_header_route
        uri: https://example.org
        filters:
        - AddResponseHeader=X-Response-Red, Blue
```

This adds `X-Response-Red:Blue` header to the downstream response's headers for all matching requests.

`AddResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddResponseHeader GatewayFilter` that uses a variable:

Example 19. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - AddResponseHeader=foo, bar-{segment}
```

6.4. The `DedupeResponseHeader GatewayFilter` Factory

The DedupeResponseHeader GatewayFilter factory takes a `name` parameter and an optional `strategy` parameter. `name` can contain a space-separated list of header names. The following example configures a `DedupeResponseHeader GatewayFilter`:

Example 20. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: dedupe_response_header_route
        uri: https://example.org
        filters:
        - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-
Allow-Origin
```

This removes duplicate values of `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` response headers in cases when both the gateway CORS logic and the downstream logic add them.

The `DedupeResponseHeader` filter also accepts an optional `strategy` parameter. The accepted values are `RETAIN_FIRST` (default), `RETAIN_LAST`, and `RETAIN_UNIQUE`.

6.5. Spring Cloud CircuitBreaker GatewayFilter Factory

The Spring Cloud CircuitBreaker GatewayFilter factory uses the Spring Cloud CircuitBreaker APIs to wrap Gateway routes in a circuit breaker. Spring Cloud CircuitBreaker supports multiple libraries that can be used with Spring Cloud Gateway. Spring Cloud supports Resilience4J out of the box.

To enable the Spring Cloud CircuitBreaker filter, you need to place `spring-cloud-starter-circuitbreaker-reactor-resilience4j` on the classpath. The following example configures a Spring Cloud CircuitBreaker `GatewayFilter`:

Example 21. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: https://example.org
        filters:
        - CircuitBreaker=myCircuitBreaker
```

To configure the circuit breaker, see the configuration for the underlying circuit breaker implementation you are using.

- Resilience4J Documentation

The Spring Cloud CircuitBreaker filter can also accept an optional `fallbackUri` parameter. Currently, only `forward:` schemed URIs are supported. If the fallback is called, the request is forwarded to the controller matched by the URI. The following example configures such a fallback:

Example 22. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: lb://backing-service:8088
        predicates:
```

```
        - Path=/consumingServiceEndpoint
        filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/inCaseOfFailureUseThis
        - RewritePath=/consumingServiceEndpoint, /backingServiceEndpoint
```

The following listing does the same thing in Java:

Example 23. Application.java

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("circuitbreaker_route", r -> r.path("/consumingServiceEndpoint")
            .filters(f -> f.circuitBreaker(c -> c.name("myCircuitBreaker")
.fallbackUri("forward:/inCaseOfFailureUseThis"))
                .rewritePath("/consumingServiceEndpoint", "
/backingServiceEndpoint")).uri("lb://backing-service:8088")
        .build();
}
```

This example forwards to the `/inCaseofFailureUseThis` URI when the circuit breaker fallback is called. Note that this example also demonstrates the (optional) Spring Cloud LoadBalancer load-balancing (defined by the `lb` prefix on the destination URI).

The primary scenario is to use the `fallbackUri` to define an internal controller or handler within the gateway application. However, you can also reroute the request to a controller or handler in an external application, as follows:

Example 24. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: ingredients
        uri: lb://ingredients
        predicates:
        - Path=//ingredients/**
        filters:
        - name: CircuitBreaker
          args:
            name: fetchIngredients
            fallbackUri: forward:/fallback
      - id: ingredients-fallback
```

```
        uri: http://localhost:9994
        predicates:
        - Path=/fallback
```

In this example, there is no `fallback` endpoint or handler in the gateway application. However, there is one in another application, registered under `localhost:9994`.

In case of the request being forwarded to fallback, the Spring Cloud CircuitBreaker Gateway filter also provides the `Throwable` that has caused it. It is added to the `ServerWebExchange` as the `ServerWebExchangeUtils.CIRCUITBREAKER_EXECUTION_EXCEPTION_ATTR` attribute that can be used when handling the fallback within the gateway application.

For the external controller/handler scenario, headers can be added with exception details. You can find more information on doing so in the FallbackHeaders GatewayFilter Factory section.

6.5.1. Tripping The Circuit Breaker On Status Codes

In some cases you might want to trip a circuit breaker based on the status code returned from the route it wraps. The circuit breaker config object takes a list of status codes that if returned will cause the the circuit breaker to be tripped. When setting the status codes you want to trip the circuit breaker you can either use a integer with the status code value or the String representation of the `HttpStatus` enumeration.

Example 25. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingServiceEndpoint
        filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/inCaseOfFailureUseThis
            statusCodes:
              - 500
              - "NOT_FOUND"
```

Example 26. Application.java

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
```

```
    return builder.routes()
        .route("circuitbreaker_route", r -> r.path("/consumingServiceEndpoint")
            .filters(f -> f.circuitBreaker(c -> c.name("myCircuitBreaker")
.fallbackUri("forward:/inCaseOfFailureUseThis").addStatusCode("INTERNAL_SERVER_ERR
OR"))
                .rewritePath("/consumingServiceEndpoint", "
/backingServiceEndpoint")).uri("lb://backing-service:8088")
        .build();
}
```

6.6. The `FallbackHeaders` `GatewayFilter` Factory

The `FallbackHeaders` factory lets you add Spring Cloud CircuitBreaker execution exception details in the headers of a request forwarded to a `fallbackUri` in an external application, as in the following scenario:

Example 27. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: ingredients
        uri: lb://ingredients
        predicates:
        - Path=//ingredients/**
        filters:
        - name: CircuitBreaker
          args:
            name: fetchIngredients
            fallbackUri: forward:/fallback
      - id: ingredients-fallback
        uri: http://localhost:9994
        predicates:
        - Path=/fallback
        filters:
        - name: FallbackHeaders
          args:
            executionExceptionTypeHeaderName: Test-Header
```

In this example, after an execution exception occurs while running the circuit breaker, the request is forwarded to the `fallback` endpoint or handler in an application running on `localhost:9994`. The headers with the exception type, message and (if available) root cause exception type and message are added to that request by the `FallbackHeaders` filter.

You can overwrite the names of the headers in the configuration by setting the values of the

following arguments (shown with their default values):

- `executionExceptionTypeHeaderName` (`"Execution-Exception-Type"`)
- `executionExceptionMessageHeaderName` (`"Execution-Exception-Message"`)
- `rootCauseExceptionTypeHeaderName` (`"Root-Cause-Exception-Type"`)
- `rootCauseExceptionMessageHeaderName` (`"Root-Cause-Exception-Message"`)

For more information on circuit breakers and the gateway see the Spring Cloud CircuitBreaker Factory section.

6.7. The `MapRequestHeader GatewayFilter` Factory

The `MapRequestHeader GatewayFilter` factory takes `fromHeader` and `toHeader` parameters. It creates a new named header (`toHeader`), and the value is extracted out of an existing named header (`fromHeader`) from the incoming http request. If the input header does not exist, the filter has no impact. If the new named header already exists, its values are augmented with the new values. The following example configures a `MapRequestHeader`:

Example 28. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: map_request_header_route
        uri: https://example.org
        filters:
        - MapRequestHeader=Blue, X-Request-Red
```

This adds `X-Request-Red:<values>` header to the downstream request with updated values from the incoming HTTP request's `Blue` header.

6.8. The `PrefixPath GatewayFilter` Factory

The `PrefixPath GatewayFilter` factory takes a single `prefix` parameter. The following example configures a `PrefixPath GatewayFilter`:

Example 29. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
```

```
        filters:
        - PrefixPath=/mypath
```

This will prefix `/mypath` to the path of all matching requests. So a request to `/hello` would be sent to `/mypath/hello`.

6.9. The `PreserveHostHeader GatewayFilter` Factory

The `PreserveHostHeader GatewayFilter` factory has no parameters. This filter sets a request attribute that the routing filter inspects to determine if the original host header should be sent, rather than the host header determined by the HTTP client. The following example configures a `PreserveHostHeader GatewayFilter`:

Example 30. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: preserve_host_route
        uri: https://example.org
        filters:
        - PreserveHostHeader
```

6.10. The `RequestRateLimiter GatewayFilter` Factory

The `RequestRateLimiter GatewayFilter` factory uses a `RateLimiter` implementation to determine if the current request is allowed to proceed. If it is not, a status of `HTTP 429 - Too Many Requests` (by default) is returned.

This filter takes an optional `keyResolver` parameter and parameters specific to the rate limiter (described later in this section).

`keyResolver` is a bean that implements the `KeyResolver` interface. In configuration, reference the bean by name using SpEL. `#{@myKeyResolver}` is a SpEL expression that references a bean named `myKeyResolver`. The following listing shows the `KeyResolver` interface:

Example 31. KeyResolver.java

```
public interface KeyResolver {
    Mono<String> resolve(ServerWebExchange exchange);
}
```

The `KeyResolver` interface lets pluggable strategies derive the key for limiting requests. In future

milestone releases, there will be some `KeyResolver` implementations.

The default implementation of `KeyResolver` is the `PrincipalNameKeyResolver`, which retrieves the `Principal` from the `ServerWebExchange` and calls `Principal.getName()`.

By default, if the `KeyResolver` does not find a key, requests are denied. You can adjust this behavior by setting the `spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key` (`true` or `false`) and `spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code` properties.

The `RequestRateLimiter` is not configurable with the "shortcut" notation. The following example below is *invalid*:

Example 32. application.properties

```
# INVALID SHORTCUT CONFIGURATION
spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2,
#{@userkeyresolver}
```

6.10.1. The Redis `RateLimiter`

The Redis implementation is based off of work done at Stripe. It requires the use of the `spring-boot-starter-data-redis-reactive` Spring Boot starter.

The algorithm used is the Token Bucket Algorithm.

The `redis-rate-limiter.replenishRate` property is how many requests per second you want a user to be allowed to do, without any dropped requests. This is the rate at which the token bucket is filled.

The `redis-rate-limiter.burstCapacity` property is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero blocks all requests.

The `redis-rate-limiter.requestedTokens` property is how many tokens a request costs. This is the number of tokens taken from the bucket for each request and defaults to `1`.

A steady rate is accomplished by setting the same value in `replenishRate` and `burstCapacity`. Temporary bursts can be allowed by setting `burstCapacity` higher than `replenishRate`. In this case, the rate limiter needs to be allowed some time between bursts (according to `replenishRate`), as two consecutive bursts will result in dropped requests (`HTTP 429 - Too Many Requests`). The following listing configures a `redis-rate-limiter`:

Rate limits bellow `1 request/s` are accomplished by setting `replenishRate` to the wanted number of requests, `requestedTokens` to the timespan in seconds and `burstCapacity` to the product of `replenishRate` and `requestedTokens`, e.g. setting `replenishRate=1`, `requestedTokens=60` and `burstCapacity=60` will result in a limit of `1 request/min`.

Example 33. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
            redis-rate-limiter.requestedTokens: 1
```

The following example configures a KeyResolver in Java:

Example 34. Config.java

```
@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst(
"user"));
}
```

This defines a request rate limit of 10 per user. A burst of 20 is allowed, but, in the next second, only 10 requests are available. The `KeyResolver` is a simple one that gets the `user` request parameter (note that this is not recommended for production).

You can also define a rate limiter as a bean that implements the `RateLimiter` interface. In configuration, you can reference the bean by name using SpEL. `#{@myRateLimiter}` is a SpEL expression that references a bean with named `myRateLimiter`. The following listing defines a rate limiter that uses the `KeyResolver` defined in the previous listing:

Example 35. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            rate-limiter: "#{@myRateLimiter}"
```

```
            key-resolver: "#{@userKeyResolver}"
```

6.11. The RedirectTo GatewayFilter Factory

The RedirectTo GatewayFilter factory takes two parameters, status and url. The status parameter should be a 300 series redirect HTTP code, such as 301. The url parameter should be a valid URL. This is the value of the Location header. For relative redirects, you should use uri: no://op as the uri of your route definition. The following listing configures a RedirectTo GatewayFilter:

Example 36. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - RedirectTo=302, https://acme.org
```

This will send a status 302 with a Location:https://acme.org header to perform a redirect.

6.12. The RemoveRequestHeader GatewayFilter Factory

The RemoveRequestHeader GatewayFilter factory takes a name parameter. It is the name of the header to be removed. The following listing configures a RemoveRequestHeader GatewayFilter:

Example 37. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestheader_route
        uri: https://example.org
        filters:
        - RemoveRequestHeader=X-Request-Foo
```

This removes the X-Request-Foo header before it is sent downstream.

6.13. RemoveResponseHeader GatewayFilter Factory

The RemoveResponseHeader GatewayFilter factory takes a name parameter. It is the name of the header

to be removed. The following listing configures a `RemoveResponseHeader GatewayFilter`:

Example 38. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removeresponseheader_route
        uri: https://example.org
        filters:
        - RemoveResponseHeader=X-Response-Foo
```

This will remove the `X-Response-Foo` header from the response before it is returned to the gateway client.

To remove any kind of sensitive header, you should configure this filter for any routes for which you may want to do so. In addition, you can configure this filter once by using `spring.cloud.gateway.default-filters` and have it applied to all routes.

6.14. The `RemoveRequestParameter GatewayFilter` Factory

The `RemoveRequestParameter GatewayFilter` factory takes a `name` parameter. It is the name of the query parameter to be removed. The following example configures a `RemoveRequestParameter GatewayFilter`:

Example 39. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestparameter_route
        uri: https://example.org
        filters:
        - RemoveRequestParameter=red
```

This will remove the `red` parameter before it is sent downstream.

6.15. `RequestHeaderSize GatewayFilter` Factory

The `RequestHeaderSize GatewayFilter` factory takes `maxSize` and `errorHeaderName` parameters. The `maxSize` parameter is the maximum data size allowed of the request header (incuding key and value). The `errorHeaderName` parameter sets the name of the response header containing an error message, by default it is "errorMessage". The following listing configures a `RequestHeaderSize`

`GatewayFilter`:

Example 40. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestheadersize_route
        uri: https://example.org
        filters:
        - RequestHeaderSize=1000B
```

This will send a status 431 if size of any request header is greater than 1000 Bytes.

6.16. The `RewritePath` `GatewayFilter` Factory

The `RewritePath` `GatewayFilter` factory takes a path `regexp` parameter and a `replacement` parameter. This uses Java regular expressions for a flexible way to rewrite the request path. The following listing configures a `RewritePath` `GatewayFilter`:

Example 41. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritepath_route
        uri: https://example.org
        predicates:
        - Path=/red/**
        filters:
        - RewritePath=/red/?(?&lt;segment&gt;.*), /$\{segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request. Note that the `$` should be replaced with `$\` because of the YAML specification.

6.17. `RewriteLocationResponseHeader` `GatewayFilter` Factory

The `RewriteLocationResponseHeader` `GatewayFilter` factory modifies the value of the `Location` response header, usually to get rid of backend-specific details. It takes `stripVersionMode`, `locationHeaderName`, `hostValue`, and `protocolsRegex` parameters. The following listing configures a `RewriteLocationResponseHeader` `GatewayFilter`:

Example 42. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritelocationresponseheader_route
        uri: http://example.org
        filters:
        - RewriteLocationResponseHeader=AS_IN_REQUEST, Location, ,
```

For example, for a request of `POST api.example.com/some/object/name`, the `Location` response header value of `object-service.prod.example.net/v2/some/object/id` is rewritten as `api.example.com/some/object/id`.

The `stripVersionMode` parameter has the following possible values: `NEVER_STRIP`, `AS_IN_REQUEST` (default), and `ALWAYS_STRIP`.

- `NEVER_STRIP`: The version is not stripped, even if the original request path contains no version.
- `AS_IN_REQUEST` The version is stripped only if the original request path contains no version.
- `ALWAYS_STRIP` The version is always stripped, even if the original request path contains version.

The `hostValue` parameter, if provided, is used to replace the `host:port` portion of the response `Location` header. If it is not provided, the value of the `Host` request header is used.

The `protocolsRegex` parameter must be a valid regex `String`, against which the protocol name is matched. If it is not matched, the filter does nothing. The default is `http|https|ftp|ftps`.

6.18. The `RewriteResponseHeader GatewayFilter` Factory

The `RewriteResponseHeader GatewayFilter` factory takes `name`, `regexp`, and `replacement` parameters. It uses Java regular expressions for a flexible way to rewrite the response header value. The following example configures a `RewriteResponseHeader GatewayFilter`:

Example 43. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewriteresponseheader_route
        uri: https://example.org
        filters:
        - RewriteResponseHeader=X-Response-Red, , password=[^&]+, password=***
```

For a header value of `/42?user=ford&password=omg!what&flag=true`, it is set to `/42?user=ford&password=***&flag=true` after making the downstream request. You must use `$\` to mean `$` because of the YAML specification.

6.19. The `SaveSession GatewayFilter` Factory

The `SaveSession GatewayFilter` factory forces a `WebSession::save` operation *before* forwarding the call downstream. This is of particular use when using something like Spring Session with a lazy data store and you need to ensure the session state has been saved before making the forwarded call. The following example configures a `SaveSession GatewayFilter`:

Example 44. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: save_session
        uri: https://example.org
        predicates:
        - Path=/foo/**
        filters:
        - SaveSession
```

If you integrate Spring Security with Spring Session and want to ensure security details have been forwarded to the remote process, this is critical.

6.20. The `SecureHeaders GatewayFilter` Factory

The `SecureHeaders GatewayFilter` factory adds a number of headers to the response, per the recommendation made in this blog post.

The following headers (shown with their default values) are added:

- `X-Xss-Protection:1 (mode=block)`
- `Strict-Transport-Security (max-age=631138519)`
- `X-Frame-Options (DENY)`
- `X-Content-Type-Options (nosniff)`
- `Referrer-Policy (no-referrer)`
- `Content-Security-Policy (default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline)'`
- `X-Download-Options (noopen)`
- `X-Permitted-Cross-Domain-Policies (none)`

To change the default values, set the appropriate property in the `spring.cloud.gateway.filter.secure-headers` namespace. The following properties are available:

- `xss-protection-header`
- `strict-transport-security`
- `x-frame-options`
- `x-content-type-options`
- `referrer-policy`
- `content-security-policy`
- `x-download-options`
- `x-permitted-cross-domain-policies`

To disable the default values set the `spring.cloud.gateway.filter.secure-headers.disable` property with comma-separated values. The following example shows how to do so:

```
spring.cloud.gateway.filter.secure-headers.disable=x-frame-options,strict-
transport-security
```

The lowercase full name of the secure header needs to be used to disable it..

6.21. The `SetPath GatewayFilter` Factory

The `SetPath GatewayFilter` factory takes a path `template` parameter. It offers a simple way to manipulate the request path by allowing templated segments of the path. This uses the URI templates from Spring Framework. Multiple matching segments are allowed. The following example configures a `SetPath GatewayFilter`:

Example 45. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setpath_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment}
        filters:
        - SetPath=/{segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request.

6.22. The SetRequestHeader GatewayFilter Factory

The SetRequestHeader GatewayFilter factory takes name and value parameters. The following listing configures a SetRequestHeader GatewayFilter:

Example 46. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setrequestheader_route
        uri: https://example.org
        filters:
        - SetRequestHeader=X-Request-Red, Blue
```

This GatewayFilter replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a X-Request-Red:1234, this would be replaced with X-Request-Red:Blue, which is what the downstream service would receive.

SetRequestHeader is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an SetRequestHeader GatewayFilter that uses a variable:

Example 47. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setrequestheader_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - SetRequestHeader=foo, bar-{segment}
```

6.23. The SetResponseHeader GatewayFilter Factory

The SetResponseHeader GatewayFilter factory takes name and value parameters. The following listing configures a SetResponseHeader GatewayFilter:

Example 48. application.yml

```
spring:
```

```
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: https://example.org
        filters:
        - SetResponseHeader=X-Response-Red, Blue
```

This GatewayFilter replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a `X-Response-Red:1234`, this is replaced with `X-Response-Red:Blue`, which is what the gateway client would receive.

`SetResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and will be expanded at runtime. The following example configures an `SetResponseHeader GatewayFilter` that uses a variable:

Example 49. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - SetResponseHeader=foo, bar-{segment}
```

6.24. The `SetStatus GatewayFilter` Factory

The `SetStatus GatewayFilter` factory takes a single parameter, `status`. It must be a valid Spring `HttpStatus`. It may be the integer value `404` or the string representation of the enumeration: `NOT_FOUND`. The following listing configures a `SetStatus GatewayFilter`:

Example 50. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatusstring_route
        uri: https://example.org
        filters:
        - SetStatus=UNAUTHORIZED
      - id: setstatusint_route
```

```
        uri: https://example.org
        filters:
        - SetStatus=401
```

In either case, the HTTP status of the response is set to 401.

You can configure the `SetStatus GatewayFilter` to return the original HTTP status code from the proxied request in a header in the response. The header is added to the response if configured with the following property:

Example 51. application.yml

```
spring:
  cloud:
    gateway:
      set-status:
        original-status-header-name: original-http-status
```

6.25. The `StripPrefix GatewayFilter` Factory

The `StripPrefix GatewayFilter` factory takes one parameter, `parts`. The `parts` parameter indicates the number of parts in the path to strip from the request before sending it downstream. The following listing configures a `StripPrefix GatewayFilter`:

Example 52. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: nameRoot
        uri: https://nameservice
        predicates:
        - Path=/name/**
        filters:
        - StripPrefix=2
```

When a request is made through the gateway to `/name/blue/red`, the request made to `nameservice` looks like `nameservice/red`.

6.26. The Retry `GatewayFilter` Factory

The `Retry GatewayFilter` factory supports the following parameters:

- `retries`: The number of retries that should be attempted.
- `statuses`: The HTTP status codes that should be retried, represented by using `org.springframework.http.HttpStatus`.
- `methods`: The HTTP methods that should be retried, represented by using `org.springframework.http.HttpMethod`.
- `series`: The series of status codes to be retried, represented by using `org.springframework.http.HttpStatus.Series`.
- `exceptions`: A list of thrown exceptions that should be retried.
- `backoff`: The configured exponential backoff for the retries. Retries are performed after a backoff interval of `firstBackoff * (factor ^ n)`, where `n` is the iteration. If `maxBackoff` is configured, the maximum backoff applied is limited to `maxBackoff`. If `basedOnPreviousValue` is true, the backoff is calculated byusing `prevBackoff * factor`.

The following defaults are configured for `Retry` filter, if enabled:

- `retries`: Three times
- `series`: 5XX series
- `methods`: GET method
- `exceptions`: `IOException` and `TimeoutException`
- `backoff`: disabled

The following listing configures a Retry `GatewayFilter`:

Example 53. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: retry_test
        uri: http://localhost:8080/flakey
        predicates:
        - Host=*.retry.com
        filters:
        - name: Retry
          args:
            retries: 3
            statuses: BAD_GATEWAY
            methods: GET,POST
            backoff:
              firstBackoff: 10ms
              maxBackoff: 50ms
              factor: 2
              basedOnPreviousValue: false
```

When using the retry filter with a `forward:` prefixed URL, the target endpoint should be written carefully so that, in case of an error, it does not do anything that could result in a response being sent to the client and committed. For example, if the target endpoint is an annotated controller, the target controller method should not return `ResponseEntity` with an error status code. Instead, it should throw an `Exception` or signal an error (for example, through a `Mono.error(ex)` return value), which the retry filter can be configured to handle by retrying.

When using the retry filter with any HTTP method with a body, the body will be cached and the gateway will become memory constrained. The body is cached in a request attribute defined by `ServerWebExchangeUtils.CACHED_REQUEST_BODY_ATTR`. The type of the object is a `org.springframework.core.io.buffer.DataBuffer`.

A simplified "shortcut" notation can be added with a single `status` and `method`.

The following two examples are equivalent:

Example 54. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: retry_route
        uri: https://example.org
        filters:
        - name: Retry
          args:
            retries: 3
            statuses: INTERNAL_SERVER_ERROR
            methods: GET
            backoff:
              firstBackoff: 10ms
              maxBackoff: 50ms
              factor: 2
              basedOnPreviousValue: false

      - id: retryshortcut_route
        uri: https://example.org
        filters:
        - Retry=3,INTERNAL_SERVER_ERROR,GET,10ms,50ms,2,false
```

6.27. The `RequestSize GatewayFilter` Factory

When the request size is greater than the permissible limit, the `RequestSize GatewayFilter` factory can restrict a request from reaching the downstream service. The filter takes a `maxSize` parameter. The `maxSize` is a `DataSize` type, so values can be defined as a number followed by an optional

`DataUnit` suffix such as 'KB' or 'MB'. The default is 'B' for bytes. It is the permissible size limit of the request defined in bytes. The following listing configures a `RequestSize GatewayFilter`:

Example 55. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: request_size_route
        uri: http://localhost:8080/upload
        predicates:
        - Path=/upload
        filters:
        - name: RequestSize
          args:
            maxSize: 5000000
```

The `RequestSize GatewayFilter` factory sets the response status as `413 Payload Too Large` with an additional header `errorMessage` when the request is rejected due to size. The following example shows such an `errorMessage`:

```
errorMessage : Request size is larger than permissible limit. Request size is 6.0
MB where permissible limit is 5.0 MB
```

The default request size is set to five MB if not provided as a filter argument in the route definition.

6.28. The `SetRequestHostHeader GatewayFilter` Factory

There are certain situation when the host header may need to be overridden. In this situation, the `SetRequestHostHeader GatewayFilter` factory can replace the existing host header with a specified vaue. The filter takes a `host` parameter. The following listing configures a `SetRequestHostHeader GatewayFilter`:

Example 56. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: set_request_host_header_route
        uri: http://localhost:8080/headers
        predicates:
```

```
        - Path=/headers
        filters:
        - name: SetRequestHostHeader
          args:
            host: example.org
```

The `SetRequestHostHeader` `GatewayFilter` factory replaces the value of the host header with `example.org`.

6.29. Modify a Request Body `GatewayFilter` Factory

You can use the `ModifyRequestBody` filter filter to modify the request body before it is sent downstream by the gateway.

This filter can be configured only by using the Java DSL.

The following listing shows how to modify a request body `GatewayFilter`:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_request_obj", r -> r.host("*.rewriterequestobj.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyRequestBody(String.class, Hello.class, MediaType
.APPLICATION_JSON_VALUE,
                    (exchange, s) -> return Mono.just(new Hello(s.toUpperCase()))
)).uri(uri))
        .build();
}

static class Hello {
    String message;

    public Hello() { }

    public Hello(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
```

```
}
```

if the request has no body, the `RewriteFilter` will be passed `null`. `Mono.empty()` should be returned to assign a missing body in the request.

6.30. Modify a Response Body `GatewayFilter` Factory

You can use the `ModifyResponseBody` filter to modify the response body before it is sent back to the client.

This filter can be configured only by using the Java DSL.

The following listing shows how to modify a response body `GatewayFilter`:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_response_upper", r -> r.host("*.rewriteresponseupper.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyResponseBody(String.class, String.class,
                    (exchange, s) -> Mono.just(s.toUpperCase()))).uri(uri))
        .build();
}
```

if the response has no body, the `RewriteFilter` will be passed `null`. `Mono.empty()` should be returned to assign a missing body in the response.

6.31. Token Relay `GatewayFilter` Factory

A Token Relay is where an OAuth2 consumer acts as a Client and forwards the incoming token to outgoing resource requests. The consumer can be a pure Client (like an SSO application) or a Resource Server.

Spring Cloud Gateway can forward OAuth2 access tokens downstream to the services it is proxying. To add this functionlity to gateway you need to add the `TokenRelayGatewayFilterFactory` like this:

App.java

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
            .route("resource", r -> r.path("/resource")
                    .filters(f -> f.tokenRelay())
                    .uri("http://localhost:9000"))
```

```
            .build();
}
```

or this

application.yaml

```
spring:
  cloud:
    gateway:
      routes:
      - id: resource
        uri: http://localhost:9000
        predicates:
        - Path=/resource
        filters:
        - TokenRelay=
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the services (in this case `/resource`).

To enable this for Spring Cloud Gateway add the following dependencies

- `org.springframework.boot:spring-boot-starter-oauth2-client`

How does it work? The {githubmaster}/src/main/java/org/springframework/cloud/gateway/security/TokenRelayGatewayFilterFactory.java[filter] extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

For a full working sample see this project.

A `TokenRelayGatewayFilterFactory` bean will only be created if the proper `spring.security.oauth2.client.*` properties are set which will trigger creation of a `ReactiveClientRegistrationRepository` bean.

The default implementation of `ReactiveOAuth2AuthorizedClientService` used by `TokenRelayGatewayFilterFactory` uses an in-memory data store. You will need to provide your own implementation `ReactiveOAuth2AuthorizedClientService` if you need a more robust solution.

6.32. The `CacheRequestBody GatewayFilter` Factory

There are certain situation need to read body.Since the request body stream can only be read once, we need to cache the request body. You can use the `CacheRequestBody` filter to cache request body before it send to the downstream and get body from exchagne attribute.

The following listing shows how to cache the request body `GatewayFilter`:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("cache_request_body_route", r -> r.path("/downstream/**")
            .filters(f -> f.prefixPath("/httpbin")
                .cacheRequestBody(String.class).uri(uri))
        .build();
}
```

Example 57. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: cache_request_body_route
        uri: lb://downstream
        predicates:
        - Path=/downstream/**
        filters:
        - name: CacheRequestBody
          args:
            bodyClass: java.lang.String
```

`CacheRequestBody` will extract request body and conver it to body class (such as `java.lang.String`, defined in the preceding example). then places it in the `ServerWebExchange.getAttributes()` with a key defined in `ServerWebExchangeUtils.CACHED_REQUEST_BODY_ATTR`.

This filter only works with http request (including https).

6.33. The `JsonToGrpc GatewayFilter` Factory

The JSONToGRPCFilter GatewayFilter Factory converts a JSON payload to a gRPC request.

The filter takes the following arguments:

- `protoDescriptor` Proto descriptor file.

This file can be generated using `protoc` specifying the `--descriptor_set_out` flag:

```
protoc --proto_path=src/main/resources/proto/ \
--descriptor_set_out=src/main/resources/proto/hello.pb  \
src/main/resources/proto/hello.proto
```

- `protoFile` Proto definition file.
- `service` Short name of the service that will handle the request.
- `method` Method name in the service that will handle the request.

`streaming` is not supported.

application.yml.

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
            .route("json-grpc", r -> r.path("/json/hello").filters(f -> {
                String protoDescriptor = "file:src/main/proto/hello.pb";
                String protoFile = "file:src/main/proto/hello.proto";
                String service = "HelloService";
                String method = "hello";
                return f.jsonToGRPC(protoDescriptor, protoFile, service, method);
            }).uri(uri))
```

```
spring:
  cloud:
    gateway:
      routes:
        - id: json-grpc
          uri: https://localhost:6565/testhello
          predicates:
            - Path=/json/**
          filters:
            - name: JsonToGrpc
              args:
                protoDescriptor: file:proto/hello.pb
                protoFile: file:proto/hello.proto
                service: HelloService
                method: hello
```

When a request is made through the gateway to `/json/hello` the request will be transformed using the definition provided in `hello.proto`, sent to `HelloService/hello`, and transform the response back to JSON.

By default, it will create a `NettyChannel` using the default `TrustManagerFactory`. However, this `TrustManager` can be customized by creating a bean of type `GrpcSslConfigurer`:

```
@Configuration
public class GRPCLocalConfiguration {
    @Bean
    public GRPCSSLContext sslContext() {
        TrustManager trustManager = trustAllCerts();
```

```
            return new GRPCSSLContext(trustManager);
        }
    }
```

6.34. Default Filters

To add a filter and apply it to all routes, you can use `spring.cloud.gateway.default-filters`. This property takes a list of filters. The following listing defines a set of default filters:

Example 58. application.yml

```
spring:
  cloud:
    gateway:
      default-filters:
      - AddResponseHeader=X-Response-Default-Red, Default-Blue
      - PrefixPath=/httpbin
```

Chapter 7. Global Filters

The `GlobalFilter` interface has the same signature as `GatewayFilter`. These are special filters that are conditionally applied to all routes.

This interface and its usage are subject to change in future milestone releases.

7.1. Combined Global Filter and `GatewayFilter` Ordering

When a request matches a route, the filtering web handler adds all instances of `GlobalFilter` and all route-specific instances of `GatewayFilter` to a filter chain. This combined filter chain is sorted by the `org.springframework.core.Ordered` interface, which you can set by implementing the `getOrder()` method.

As Spring Cloud Gateway distinguishes between “pre” and “post” phases for filter logic execution (see How it Works), the filter with the highest precedence is the first in the “pre”-phase and the last in the “post”-phase.

The following listing configures a filter chain:

Example 59. ExampleConfiguration.java

```
@Bean
public GlobalFilter customFilter() {
    return new CustomGlobalFilter();
}

public class CustomGlobalFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        log.info("custom global filter");
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return -1;
    }
}
```

7.2. Forward Routing Filter

The `ForwardRoutingFilter` looks for a URI in the exchange attribute `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a `forward` scheme (such as

`forward:///localendpoint`), it uses the Spring `DispatcherHandler` to handle the request. The path part of the request URL is overridden with the path in the forward URL. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute.

7.3. The `ReactiveLoadBalancerClientFilter`

The `ReactiveLoadBalancerClientFilter` looks for a URI in the exchange attribute named `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a `lb` scheme (such as `lb://myservice`), it uses the Spring Cloud `ReactorLoadBalancer` to resolve the name (`myservice` in this example) to an actual host and port and replaces the URI in the same attribute. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute. The filter also looks in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` attribute to see if it equals `lb`. If so, the same rules apply. The following listing configures a `ReactiveLoadBalancerClientFilter`:

Example 60. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
        predicates:
        - Path=/service/**
```

By default, when a service instance cannot be found by the `ReactorLoadBalancer`, a `503` is returned. You can configure the gateway to return a `404` by setting `spring.cloud.gateway.loadbalancer.use404=true`.

The `isSecure` value of the `ServiceInstance` returned from the `ReactiveLoadBalancerClientFilter` overrides the scheme specified in the request made to the Gateway. For example, if the request comes into the Gateway over `HTTPS` but the `ServiceInstance` indicates it is not secure, the downstream request is made over `HTTP`. The opposite situation can also apply. However, if `GATEWAY_SCHEME_PREFIX_ATTR` is specified for the route in the Gateway configuration, the prefix is stripped and the resulting scheme from the route URL overrides the `ServiceInstance` configuration.

Gateway supports all the LoadBalancer features. You can read more about them in the Spring Cloud Commons documentation.

7.4. The Netty Routing Filter

The Netty routing filter runs if the URL located in the

`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `http` or `https` scheme. It uses the Netty `HttpClient` to make the downstream proxy request. The response is put in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute for use in a later filter. (There is also an experimental `WebClientHttpRoutingFilter` that performs the same function but does not require Netty.)

7.5. The Netty Write Response Filter

The `NettyWriteResponseFilter` runs if there is a Netty `HttpClientResponse` in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute. It runs after all other filters have completed and writes the proxy response back to the gateway client response. (There is also an experimental `WebClientWriteResponseFilter` that performs the same function but does not require Netty.)

7.6. The `RouteToRequestUrl` Filter

If there is a `Route` object in the `ServerWebExchangeUtils.GATEWAY_ROUTE_ATTR` exchange attribute, the `RouteToRequestUrlFilter` runs. It creates a new URI, based off of the request URI but updated with the URI attribute of the `Route` object. The new URI is placed in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute.

If the URI has a scheme prefix, such as `lb:ws://serviceid`, the `lb` scheme is stripped from the URI and placed in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` for use later in the filter chain.

7.7. The Websocket Routing Filter

If the URL located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `ws` or `wss` scheme, the websocket routing filter runs. It uses the Spring WebSocket infrastructure to forward the websocket request downstream.

You can load-balance websockets by prefixing the URI with `lb`, such as `lb:ws://serviceid`.

If you use SockJS as a fallback over normal HTTP, you should configure a normal HTTP route as well as the websocket Route.

The following listing configures a websocket routing filter:

Example 61. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      # SockJS route
      - id: websocket_sockjs_route
        uri: http://localhost:3001
```

```
        predicates:
        - Path=/websocket/info/**
      # Normal Websocket route
      - id: websocket_route
        uri: ws://localhost:3001
        predicates:
        - Path=/websocket/**
```

7.8. The Gateway Metrics Filter

To enable gateway metrics, add spring-boot-starter-actuator as a project dependency. Then, by default, the gateway metrics filter runs as long as the property `spring.cloud.gateway.metrics.enabled` is not set to `false`. This filter adds a timer metric named `spring.cloud.gateway.requests` with the following tags:

- `routeId`: The route ID.
- `routeUri`: The URI to which the API is routed.
- `outcome`: The outcome, as classified by HttpStatus.Series.
- `status`: The HTTP status of the request returned to the client.
- `httpStatusCode`: The HTTP Status of the request returned to the client.
- `httpMethod`: The HTTP method used for the request.

In addition, through the property `spring.cloud.gateway.metrics.tags.path.enabled` (by default, set to false), you can activate an extra metric with the tag:

- `path`: Path of the request.

These metrics are then available to be scraped from `/actuator/metrics/spring.cloud.gateway.requests` and can be easily integrated with Prometheus to create a Grafana dashboard.

To enable the prometheus endpoint, add `micrometer-registry-prometheus` as a project dependency.

7.9. Marking An Exchange As Routed

After the gateway has routed a `ServerWebExchange`, it marks that exchange as “routed” by adding `gatewayAlreadyRouted` to the exchange attributes. Once a request has been marked as routed, other routing filters will not route the request again, essentially skipping the filter. There are convenience methods that you can use to mark an exchange as routed or check if an exchange has already been routed.

- `ServerWebExchangeUtils.isAlreadyRouted` takes a `ServerWebExchange` object and checks if it has been “routed”.
- `ServerWebExchangeUtils.setAlreadyRouted` takes a `ServerWebExchange` object and marks it as

“routed”.

Chapter 8. HttpHeadersFilters

HttpHeadersFilters are applied to requests before sending them downstream, such as in the `NettyRoutingFilter`.

8.1. Forwarded Headers Filter

The `Forwarded` Headers Filter creates a `Forwarded` header to send to the downstream service. It adds the `Host` header, scheme and port of the current request to any existing `Forwarded` header.

8.2. RemoveHopByHop Headers Filter

The `RemoveHopByHop` Headers Filter removes headers from forwarded requests. The default list of headers that is removed comes from the IETF.

The default removed headers are:

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer
- Transfer-Encoding
- Upgrade

To change this, set the `spring.cloud.gateway.filter.remove-hop-by-hop.headers` property to the list of header names to remove.

8.3. XForwarded Headers Filter

The `XForwarded` Headers Filter creates various a `X-Forwarded-*` headers to send to the downstream service. It users the `Host` header, scheme, port and path of the current request to create the various headers.

Creating of individual headers can be controlled by the following boolean properties (defaults to true):

- `spring.cloud.gateway.x-forwarded.for-enabled`
- `spring.cloud.gateway.x-forwarded.host-enabled`
- `spring.cloud.gateway.x-forwarded.port-enabled`
- `spring.cloud.gateway.x-forwarded.proto-enabled`
- `spring.cloud.gateway.x-forwarded.prefix-enabled`

Appending multiple headers can be controlled by the following boolean properties (defaults to true):

- `spring.cloud.gateway.x-forwarded.for-append`
- `spring.cloud.gateway.x-forwarded.host-append`
- `spring.cloud.gateway.x-forwarded.port-append`
- `spring.cloud.gateway.x-forwarded.proto-append`
- `spring.cloud.gateway.x-forwarded.prefix-append`

Chapter 9. TLS and SSL

The gateway can listen for requests on HTTPS by following the usual Spring server configuration. The following example shows how to do so:

Example 62. application.yml

```
server:
  ssl:
    enabled: true
    key-alias: scg
    key-store-password: scg1234
    key-store: classpath:scg-keystore.p12
    key-store-type: PKCS12
```

You can route gateway routes to both HTTP and HTTPS backends. If you are routing to an HTTPS backend, you can configure the gateway to trust all downstream certificates with the following configuration:

Example 63. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          useInsecureTrustManager: true
```

Using an insecure trust manager is not suitable for production. For a production deployment, you can configure the gateway with a set of known certificates that it can trust with the following configuration:

Example 64. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          trustedX509Certificates:
          - cert1.pem
          - cert2.pem
```

If the Spring Cloud Gateway is not provisioned with trusted certificates, the default trust store is used (which you can override by setting the `javax.net.ssl.trustStore` system property).

9.1. TLS Handshake

The gateway maintains a client pool that it uses to route to backends. When communicating over HTTPS, the client initiates a TLS handshake. A number of timeouts are associated with this handshake. You can configure these timeouts can be configured (defaults shown) as follows:

Example 65. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          handshake-timeout-millis: 10000
          close-notify-flush-timeout-millis: 3000
          close-notify-read-timeout-millis: 0
```

Chapter 10. Configuration

Configuration for Spring Cloud Gateway is driven by a collection of `RouteDefinitionLocator` instances. The following listing shows the definition of the `RouteDefinitionLocator` interface:

Example 66. RouteDefinitionLocator.java

```
public interface RouteDefinitionLocator {
    Flux<RouteDefinition> getRouteDefinitions();
}
```

By default, a `PropertiesRouteDefinitionLocator` loads properties by using Spring Boot's `@ConfigurationProperties` mechanism.

The earlier configuration examples all use a shortcut notation that uses positional arguments rather than named ones. The following two examples are equivalent:

Example 67. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatus_route
        uri: https://example.org
        filters:
        - name: SetStatus
          args:
            status: 401
      - id: setstatusshortcut_route
        uri: https://example.org
        filters:
        - SetStatus=401
```

For some usages of the gateway, properties are adequate, but some production use cases benefit from loading configuration from an external source, such as a database. Future milestone versions will have `RouteDefinitionLocator` implementations based off of Spring Data Repositories, such as Redis, MongoDB, and Cassandra.

10.1. RouteDefinition Metrics

To enable `RouteDefinition` metrics, add spring-boot-starter-actuator as a project dependency. Then, by default, the metrics will be available as long as the property `spring.cloud.gateway.metrics.enabled` is set to `true`. A gauge metric named `spring.cloud.gateway.routes.count` will be added, whose value is the number of `RouteDefinitions`.

This metric will be available from `/actuator/metrics/spring.cloud.gateway.routes.count`.

Chapter 11. Route Metadata Configuration

You can configure additional parameters for each route by using metadata, as follows:

Example 68. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: route_with_metadata
        uri: https://example.org
        metadata:
          optionName: "OptionValue"
          compositeObject:
            name: "value"
          iAmNumber: 1
```

You could acquire all metadata properties from an exchange, as follows:

```
Route route = exchange.getAttribute(GATEWAY_ROUTE_ATTR);
// get all metadata properties
route.getMetadata();
// get a single metadata property
route.getMetadata(someKey);
```

Chapter 12. Http timeouts configuration

Http timeouts (response and connect) can be configured for all routes and overridden for each specific route.

12.1. Global timeouts

To configure Global http timeouts:
`connect-timeout` must be specified in milliseconds.
`response-timeout` must be specified as a java.time.Duration

global http timeouts example

```
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 5s
```

12.2. Per-route timeouts

To configure per-route timeouts:
`connect-timeout` must be specified in milliseconds.
`response-timeout` must be specified in milliseconds.

per-route http timeouts configuration via configuration

```
      - id: per_route_timeouts
        uri: https://example.org
        predicates:
          - name: Path
            args:
              pattern: /delay/{timeout}
        metadata:
          response-timeout: 200
          connect-timeout: 200
```

per-route timeouts configuration using Java DSL

```
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.CONNECT_TIMEOUT_ATTR;
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.RESPONSE_TIMEOUT_ATTR;

      @Bean
      public RouteLocator customRouteLocator(RouteLocatorBuilder routeBuilder){
         return routeBuilder.routes()
```

```
            .route("test1", r -> {
               return r.host("*.somehost.org").and().path("/somepath")
                     .filters(f -> f.addRequestHeader("header1", "header-value-1"))
                     .uri("http://someuri")
                     .metadata(RESPONSE_TIMEOUT_ATTR, 200)
                     .metadata(CONNECT_TIMEOUT_ATTR, 200);
            })
            .build();
    }
```

A per-route `response-timeout` with a negative value will disable the global `response-timeout` value.

```
    - id: per_route_timeouts
      uri: https://example.org
      predicates:
        - name: Path
          args:
            pattern: /delay/{timeout}
      metadata:
        response-timeout: -1
```

Chapter 13. Fluent Java Routes API

To allow for simple configuration in Java, the `RouteLocatorBuilder` bean includes a fluent API. The following listing shows how it works:

Example 69. GatewaySampleApplication.java

```
// static imports from GatewayFilters and RoutePredicates
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder,
ThrottleGatewayFilterFactory throttle) {
    return builder.routes()
            .route(r -> r.host("**.abc.org").and().path("/image/png")
                .filters(f ->
                        f.addResponseHeader("X-TestHeader", "foobar"))
                .uri("http://httpbin.org:80")
            )
            .route(r -> r.path("/image/webp")
                .filters(f ->
                        f.addResponseHeader("X-AnotherHeader", "baz"))
                .uri("http://httpbin.org:80")
                .metadata("key", "value")
            )
            .route(r -> r.order(-1)
                .host("**.throttle.org").and().path("/get")
                .filters(f -> f.filter(throttle.apply(1,
                        1,
                        10,
                        TimeUnit.SECONDS)))
                .uri("http://httpbin.org:80")
                .metadata("key", "value")
            )
            .build();
}
```

This style also allows for more custom predicate assertions. The predicates defined by `RouteDefinitionLocator` beans are combined using logical `and`. By using the fluent Java API, you can use the `and()`, `or()`, and `negate()` operators on the `Predicate` class.

Chapter 14. The DiscoveryClient Route Definition Locator

You can configure the gateway to create routes based on services registered with a DiscoveryClient compatible service registry.

To enable this, set spring.cloud.gateway.discovery.locator.enabled=true and make sure a DiscoveryClient implementation (such as Netflix Eureka, Consul, or Zookeeper) is on the classpath and enabled.

14.1. Configuring Predicates and Filters For DiscoveryClient Routes

By default, the gateway defines a single predicate and filter for routes created with a DiscoveryClient.

The default predicate is a path predicate defined with the pattern /serviceId/**, where serviceId is the ID of the service from the DiscoveryClient.

The default filter is a rewrite path filter with the regex /serviceId/?(?<remaining>.*) and the replacement /${remaining}. This strips the service ID from the path before the request is sent downstream.

If you want to customize the predicates or filters used by the DiscoveryClient routes, set spring.cloud.gateway.discovery.locator.predicates[x] and spring.cloud.gateway.discovery.locator.filters[y]. When doing so, you need to make sure to include the default predicate and filter shown earlier, if you want to retain that functionality. The following example shows what this looks like:

Example 70. application.properties

```
spring.cloud.gateway.discovery.locator.predicates[0].name: Path
spring.cloud.gateway.discovery.locator.predicates[0].args[pattern]:
"'/'+serviceId+'/**'"
spring.cloud.gateway.discovery.locator.predicates[1].name: Host
spring.cloud.gateway.discovery.locator.predicates[1].args[pattern]: "'**.foo.com'"
spring.cloud.gateway.discovery.locator.filters[0].name: CircuitBreaker
spring.cloud.gateway.discovery.locator.filters[0].args[name]: serviceId
spring.cloud.gateway.discovery.locator.filters[1].name: RewritePath
spring.cloud.gateway.discovery.locator.filters[1].args[regexp]: "'/' + serviceId +
'/?(?<remaining>.*)'"
spring.cloud.gateway.discovery.locator.filters[1].args[replacement]:
"'/${remaining}'"
```

Chapter 15. Reactor Netty Access Logs

To enable Reactor Netty access logs, set `-Dreactor.netty.http.server.accessLogEnabled=true`.

It must be a Java System Property, not a Spring Boot property.

You can configure the logging system to have a separate access log file. The following example creates a Logback configuration:

Example 71. logback.xml

```
    <appender name="accessLog" class="ch.qos.logback.core.FileAppender">
        <file>access_log.log</file>
        <encoder>
            <pattern>%msg%n</pattern>
        </encoder>
    </appender>
    <appender name="async" class="ch.qos.logback.classic.AsyncAppender">
        <appender-ref ref="accessLog" />
    </appender>

    <logger name="reactor.netty.http.server.AccessLog" level="INFO" additivity=
"false">
        <appender-ref ref="async"/>
    </logger>
```

Chapter 16. CORS Configuration

You can configure the gateway to control CORS behavior. The “global” CORS configuration is a map of URL patterns to Spring Framework `CorsConfiguration`. The following example configures CORS:

Example 72. application.yml

```
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '[/**]':
            allowedOrigins: "https://docs.spring.io"
            allowedMethods:
            - GET
```

In the preceding example, CORS requests are allowed from requests that originate from `docs.spring.io` for all GET requested paths.

To provide the same CORS configuration to requests that are not handled by some gateway route predicate, set the `spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping` property to `true`. This is useful when you try to support CORS preflight requests and your route predicate does not evalute to `true` because the HTTP method is `options`.

Chapter 17. Actuator API

The `/gateway` actuator endpoint lets you monitor and interact with a Spring Cloud Gateway application. To be remotely accessible, the endpoint has to be enabled and exposed over HTTP or JMX in the application properties. The following listing shows how to do so:

Example 73. application.properties

```
management.endpoint.gateway.enabled=true # default value
management.endpoints.web.exposure.include=gateway
```

17.1. Verbose Actuator Format

A new, more verbose format has been added to Spring Cloud Gateway. It adds more detail to each route, letting you view the predicates and filters associated with each route along with any configuration that is available. The following example configures `/actuator/gateway/routes`:

```
[
  {
    "predicate": "(Hosts: [**.addrequestheader.org] && Paths: [/headers], match
trailing slash: true)",
    "route_id": "add_request_header_test",
    "filters": [
      "[[AddResponseHeader X-Response-Default-Foo = 'Default-Bar'], order = 1]",
      "[[AddRequestHeader X-Request-Foo = 'Bar'], order = 1]",
      "[[PrefixPath prefix = '/httpbin'], order = 2]"
    ],
    "uri": "lb://testservice",
    "order": 0
  }
]
```

This feature is enabled by default. To disable it, set the following property:

Example 74. application.properties

```
spring.cloud.gateway.actuator.verbose.enabled=false
```

This will default to `true` in a future release.

17.2. Retrieving Route Filters

This section details how to retrieve route filters, including:

- Global Filters
- [gateway-route-filters]

17.2.1. Global Filters

To retrieve the global filters applied to all routes, make a `GET` request to `/actuator/gateway/globalfilters`. The resulting response is similar to the following:

```
{

"org.springframework.cloud.gateway.filter.ReactiveLoadBalancerClientFilter@77856cc
5": 10100,
  "org.springframework.cloud.gateway.filter.RouteToRequestUrlFilter@4f6fd101":
10000,
  "org.springframework.cloud.gateway.filter.NettyWriteResponseFilter@32d22650":
-1,
  "org.springframework.cloud.gateway.filter.ForwardRoutingFilter@106459d9":
2147483647,
  "org.springframework.cloud.gateway.filter.NettyRoutingFilter@1fbd5e0":
2147483647,
  "org.springframework.cloud.gateway.filter.ForwardPathFilter@33a71d23": 0,
  "org.springframework.cloud.gateway.filter.AdaptCachedBodyGlobalFilter@135064ea":
2147483637,
  "org.springframework.cloud.gateway.filter.WebsocketRoutingFilter@23c05889":
2147483646
}
```

The response contains the details of the global filters that are in place. For each global filter, there is a string representation of the filter object (for example, `org.springframework.cloud.gateway.filter.ReactiveLoadBalancerClientFilter@77856cc5`) and the corresponding order in the filter chain.}

17.2.2. Route Filters

To retrieve the `GatewayFilter` factories applied to routes, make a `GET` request to `/actuator/gateway/routefilters`. The resulting response is similar to the following:

```
{
  "[AddRequestHeaderGatewayFilterFactory@570ed9c configClass =
AbstractNameValueGatewayFilterFactory.NameValueConfig]": null,
  "[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]": null,
```

```
  "[SaveSessionGatewayFilterFactory@4449b273 configClass = Object]": null
}
```

The response contains the details of the `GatewayFilter` factories applied to any particular route. For each factory there is a string representation of the corresponding object (for example, `[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]`). Note that the `null` value is due to an incomplete implementation of the endpoint controller, because it tries to set the order of the object in the filter chain, which does not apply to a `GatewayFilter` factory object.

17.3. Refreshing the Route Cache

To clear the routes cache, make a `POST` request to `/actuator/gateway/refresh`. The request returns a 200 without a response body.

17.4. Retrieving the Routes Defined in the Gateway

To retrieve the routes defined in the gateway, make a `GET` request to `/actuator/gateway/routes`. The resulting response is similar to the following:

```
[{
  "route_id": "first_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@1e9d7e7d",
    "filters": [

"OrderedGatewayFilter{delegate=org.springframework.cloud.gateway.filter.factory.Pr
eserveHostHeaderGatewayFilterFactory$$Lambda$436/674480275@6631ef72, order=0}"
    ]
  },
  "order": 0
},
{
  "route_id": "second_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@cd8d298",
    "filters": []
  },
  "order": 0
}]
```

The response contains the details of all the routes defined in the gateway. The following table

describes the structure of each element (each is a route) of the response:

Path	Type	Description
`route_id`	String	The route ID.
`route_object.predicate`	Object	The route predicate.
`route_object.filters`	Array	The `GatewayFilter` factories applied to the route.
`order`	Number	The route order.

17.5. Retrieving Information about a Particular Route

To retrieve information about a single route, make a `GET` request to `/actuator/gateway/routes/{id}` (for example, `/actuator/gateway/routes/first_route`). The resulting response is similar to the following:

```
{
  "id": "first_route",
  "predicates": [{
    "name": "Path",
    "args": {"_genkey_0":"/first"}
  }],
  "filters": [],
  "uri": "https://www.uri-destination.org",
  "order": 0
}
```

The following table describes the structure of the response:

Path	Type	Description
`id`	String	The route ID.
`predicates`	Array	The collection of route predicates. Each item defines the name and the arguments of a given predicate.
`filters`	Array	The collection of filters applied to the route.
`uri`	String	The destination URI of the route.
`order`	Number	The route order.

17.6. Creating and Deleting a Particular Route

To create a route, make a `POST` request to `/gateway/routes/{id_route_to_create}` with a JSON body

that specifies the fields of the route (see Retrieving Information about a Particular Route).

To delete a route, make a `DELETE` request to `/gateway/routes/{id_route_to_delete}`.

17.7. Recap: The List of All endpoints

The folloiwng table below summarizes the Spring Cloud Gateway actuator endpoints (note that each endpoint has `/actuator/gateway` as the base-path):

ID	HTTP Method	Description
`globalfilters`	GET	Displays the list of global filters applied to the routes.
`routefilters`	GET	Displays the list of `GatewayFilter` factories applied to a particular route.
`refresh`	POST	Clears the routes cache.
`routes`	GET	Displays the list of routes defined in the gateway.
`routes/{id}`	GET	Displays information about a particular route.
`routes/{id}`	POST	Adds a new route to the gateway.
`routes/{id}`	DELETE	Removes an existing route from the gateway.

17.8. Sharing Routes between multiple Gateway instances

Spring Cloud Gateway offers two `RouteDefinitionRepository` implementations. The first one is the `InMemoryRouteDefinitionRepository` which only lives within the memory of one Gateway instance. This type of Repository is not suited to populate Routes across multiple Gateway instances.

In order to share Routes across a cluster of Spring Cloud Gateway instances, `RedisRouteDefinitionRepository` can be used. To enable this kind of repository, the following property has to set to true: `spring.cloud.gateway.redis-route-definition-repository.enabled` Likewise to the RedisRateLimiter Filter Factory it requires the use of the spring-boot-starter-data-redis-reactive Spring Boot starter.

Chapter 18. Troubleshooting

This section covers common problems that may arise when you use Spring Cloud Gateway.

18.1. Log Levels

The following loggers may contain valuable troubleshooting information at the `DEBUG` and `TRACE` levels:

- `org.springframework.cloud.gateway`
- `org.springframework.http.server.reactive`
- `org.springframework.web.reactive`
- `org.springframework.boot.autoconfigure.web`
- `reactor.netty`
- `redisratelimiter`

18.2. Wiretap

The Reactor Netty `HttpClient` and `HttpServer` can have wiretap enabled. When combined with setting the `reactor.netty` log level to `DEBUG` or `TRACE`, it enables the logging of information, such as headers and bodies sent and received across the wire. To enable wiretap, set `spring.cloud.gateway.httpserver.wiretap=true` or `spring.cloud.gateway.httpclient.wiretap=true` for the `HttpServer` and `HttpClient`, respectively.

Chapter 19. Developer Guide

These are basic guides to writing some custom components of the gateway.

19.1. Writing Custom Route Predicate Factories

In order to write a Route Predicate you will need to implement `RoutePredicateFactory` as a bean. There is an abstract class called `AbstractRoutePredicateFactory` which you can extend.

MyRoutePredicateFactory.java

```
@Component
public class MyRoutePredicateFactory extends AbstractRoutePredicateFactory
<MyRoutePredicateFactory.Config> {

    public MyRoutePredicateFactory() {
        super(Config.class);
    }

    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        // grab configuration from Config object
        return exchange -> {
            //grab the request
            ServerHttpRequest request = exchange.getRequest();
            //take information from the request to see if it
            //matches configuration.
            return matches(config, request);
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

19.2. Writing Custom GatewayFilter Factories

To write a `GatewayFilter`, you must implement `GatewayFilterFactory` as a bean. You can extend an abstract class called `AbstractGatewayFilterFactory`. The following examples show how to do so:

Example 75. PreGatewayFilterFactory.java

```
@Component
public class PreGatewayFilterFactory extends AbstractGatewayFilterFactory
<PreGatewayFilterFactory.Config> {
```

```
    public PreGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            //If you want to build a "pre" filter you need to manipulate the
            //request before calling chain.filter
            ServerHttpRequest.Builder builder = exchange.getRequest().mutate();
            //use builder to manipulate the request
            return chain.filter(exchange.mutate().request(builder.build()).build(
));
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

PostGatewayFilterFactory.java

```
@Component
public class PostGatewayFilterFactory extends AbstractGatewayFilterFactory
<PostGatewayFilterFactory.Config> {

    public PostGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            return chain.filter(exchange).then(Mono.fromRunnable(() -> {
                ServerHttpResponse response = exchange.getResponse();
                //Manipulate the response in some way
            }));
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

19.2.1. Naming Custom Filters And References In Configuration

Custom filters class names should end in `GatewayFilterFactory`.

For example, to reference a filter named `Something` in configuration files, the filter must be in a class named `SomethingGatewayFilterFactory`.

It is possible to create a gateway filter named without the `GatewayFilterFactory` suffix, such as `class AnotherThing`. This filter could be referenced as `AnotherThing` in configuration files. This is **not** a supported naming convention and this syntax may be removed in future releases. Please update the filter name to be compliant.

19.3. Writing Custom Global Filters

To write a custom global filter, you must implement `GlobalFilter` interface as a bean. This applies the filter to all requests.

The following examples show how to set up global pre and post filters, respectively:

```
@Bean
public GlobalFilter customGlobalFilter() {
    return (exchange, chain) -> exchange.getPrincipal()
        .map(Principal::getName)
        .defaultIfEmpty("Default User")
        .map(userName -> {
          //adds header to proxied request
          exchange.getRequest().mutate().header("CUSTOM-REQUEST-HEADER", userName
).build();
          return exchange;
        })
        .flatMap(chain::filter);
}

@Bean
public GlobalFilter customGlobalPostFilter() {
    return (exchange, chain) -> chain.filter(exchange)
        .then(Mono.just(exchange))
        .map(serverWebExchange -> {
          //adds header to response
          serverWebExchange.getResponse().getHeaders().set("CUSTOM-RESPONSE-
HEADER",
              HttpStatus.OK.equals(serverWebExchange.getResponse().getStatusCode(
)) ? "It worked": "It did not work");
          return serverWebExchange;
        })
        .then();
}
```

Chapter 20. Building a Simple Gateway by Using Spring MVC or Webflux

The following describes an alternative style gateway. None of the prior documentation applies to what follows.

Spring Cloud Gateway provides a utility object called `ProxyExchange`. You can use it inside a regular Spring web handler as a method parameter. It supports basic downstream HTTP exchanges through methods that mirror the HTTP verbs. With MVC, it also supports forwarding to a local handler through the `forward()` method. To use the `ProxyExchange`, include the right module in your classpath (either `spring-cloud-gateway-mvc` or `spring-cloud-gateway-webflux`).

The following MVC example proxies a request to `/test` downstream to a remote server:

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public ResponseEntity<?> proxy(ProxyExchange<byte[]> proxy) throws Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }

}
```

The following example does the same thing with Webflux:

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public Mono<ResponseEntity<?>> proxy(ProxyExchange<byte[]> proxy) throws
Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }

}
```

Convenience methods on the `ProxyExchange` enable the handler method to discover and enhance the URI path of the incoming request. For example, you might want to extract the trailing elements of a path to pass them downstream:

```
@GetMapping("/proxy/path/**")
public ResponseEntity<?> proxyPath(ProxyExchange<byte[]> proxy) throws Exception {
  String path = proxy.path("/proxy/path/");
  return proxy.uri(home.toString() + "/foos/" + path).get();
}
```

All the features of Spring MVC and Webflux are available to gateway handler methods. As a result, you can inject request headers and query parameters, for instance, and you can constrain the incoming requests with declarations in the mapping annotation. See the documentation for `@RequestMapping` in Spring MVC for more details of those features.

You can add headers to the downstream response by using the `header()` methods on `ProxyExchange`.

You can also manipulate response headers (and anything else you like in the response) by adding a mapper to the `get()` method (and other methods). The mapper is a `Function` that takes the incoming `ResponseEntity` and converts it to an outgoing one.

First-class support is provided for “sensitive” headers (by default, `cookie` and `authorization`), which are not passed downstream, and for “proxy” (`x-forwarded-*`) headers.

Chapter 21. Configuration properties

To see the list of all Spring Cloud Gateway related configuration properties, see the appendix.