Writing Custom Spring Cloud Gateway Filters

Last modified: January 26, 2023

Written by: Ger Roza

Spring Cloud

Spring Cloud Gateway

1. Overview

In this tutorial, we'll learn how to write custom Spring Cloud Gateway filters.

We introduced this framework in our previous post, Exploring the New Spring Cloud Gateway, where we had a look at many built-

On this occasion we'll go deeper, we'll write custom filters to get the most out of our API Gateway.

First, we'll see how we can create global filters that will affect every single request handled by the gateway. Then, we'll write gate

Finally, we'll work on more advanced scenarios, learning how to modify the request or the response, and even how to chain the re

2. Project Setup

We'll start by setting up a basic application that we'll be using as our API Gateway.

2.1. Maven Configuration

When working with Spring Cloud libraries, it's always a good choice to set up a dependency management configuration to handle

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR4</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Now we can add our Spring Cloud libraries without specifying the actual version we're using:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

The latest Spring Cloud Release Train version can be found using the Maven Central search engine. Of course, we should always c

2.2. API Gateway Configuration

We'll assume there is a second application running locally in port *8081*, that exposes a resource (for simplicity's sake, just a simple

With this in mind, we'll configure our gateway to proxy requests to this service. In a nutshell, when we send a request to the gatew

So, when we call */service/resource* in our gateway, we should receive the *String* response.

To achieve this, we'll configure this route using *application properties*:

```
spring:
  cloud:
    gateway:
      routes:
      - id: service_route
        uri: http://localhost:8081
        predicates:
        - Path=/service/**
        filters:
        - RewritePath=/service(?<segment>/?.*), $\{segment}
```

And additionally, to be able to trace the gateway process properly, we'll enable some logs as well:

```
logging:
  level:
    org.springframework.cloud.gateway: DEBUG
    reactor.netty.http.client: DEBUG
```

3. Creating Global Filters

Once the gateway handler determines that a request matches a route, the framework passes the request through a filter chain. Th

In this section, we'll start by writing simple global filters. That means, that it'll affect every single request.

First, we'll see how we can execute the logic before the proxy request is sent (also known as a “pre” filter)

3.1. Writing Global “Pre” Filter Logic

As we said, we'll create simple filters at this point, since the main objective here is only to see that the filter is actually getting exec

All we have to do to create a custom global filter is to implement the Spring Cloud Gateway *GlobalFilter* interface, and add it

```
@Component
public class LoggingGlobalPreFilter implements GlobalFilter {

    final Logger logger =
      LoggerFactory.getLogger(LoggingGlobalPreFilter.class);

    @Override
    public Mono<Void> filter(
      ServerWebExchange exchange,
      GatewayFilterChain chain) {
        logger.info("Global Pre Filter executed");
        return chain.filter(exchange);
    }
}
```

We can easily see what's going on here; once this filter is invoked, we'll log a message, and continue with the execution of the filte

Let's now define a “post” filter, which can be a little bit trickier if we're not familiarized with the Reactive programming model and tl

3.2. Writing Global “Post” Filter Logic

One other thing to notice about the global filter we just defined is that the *GlobalFilter* interface defines only one method. Thus, it

For example, we can define our “post” filter in a configuration class:

```
@Configuration
public class LoggingGlobalFiltersConfigurations {

    final Logger logger =
      LoggerFactory.getLogger(
        LoggingGlobalFiltersConfigurations.class);

    @Bean
    public GlobalFilter postGlobalFilter() {
        return (exchange, chain) -> {
            return chain.filter(exchange)
              .then(Mono.fromRunnable(() -> {
                  logger.info("Global Post Filter executed");
              }));
        };
    }
}
```

Simply put, here we're running a new *Mono* instance after the chain completed its execution.

Let's try it out now by calling the */service/resource* URL in our gateway service, and checking out the log console:

```
DEBUG --- o.s.c.g.h.RoutePredicateHandlerMapping:
  Route matched: service_route
DEBUG --- o.s.c.g.h.RoutePredicateHandlerMapping:
  Mapping [Exchange: GET http://localhost/service/resource]
  to Route{id='service_route', uri=http://localhost:8081, order=0, predicate=Paths: [/service/**],
  match trailing slash: true, gatewayFilters=[[[RewritePath /service(?<segment>/?.*) = '${segment}'], order = 1]]}
INFO  --- c.b.s.c.f.global.LoggingGlobalPreFilter:
  Global Pre Filter executed
DEBUG --- r.netty.http.client.HttpClientConnect:
  [id: 0x58f7e075, L:/127.0.0.1:57215 - R:localhost/127.0.0.1:8081]
  Handler is being applied: {uri=http://localhost:8081/resource, method=GET}
DEBUG --- r.n.http.client.HttpClientOperations:
  [id: 0x58f7e075, L:/127.0.0.1:57215 - R:localhost/127.0.0.1:8081]
  Received response (auto-read:false) : [Content-Type=text/html;charset=UTF-8, Content-Length=16]
INFO  --- c.f.g.LoggingGlobalFiltersConfigurations:
  Global Post Filter executed
DEBUG --- r.n.http.client.HttpClientOperations:
  [id: 0x58f7e075, L:/127.0.0.1:57215 - R:localhost/127.0.0.1:8081] Received last HTTP packet
```

As we can see, the filters are effectively executed before and after the gateway forwards the request to the service.

Naturally, we can combine “pre” and “post” logic in a single filter:

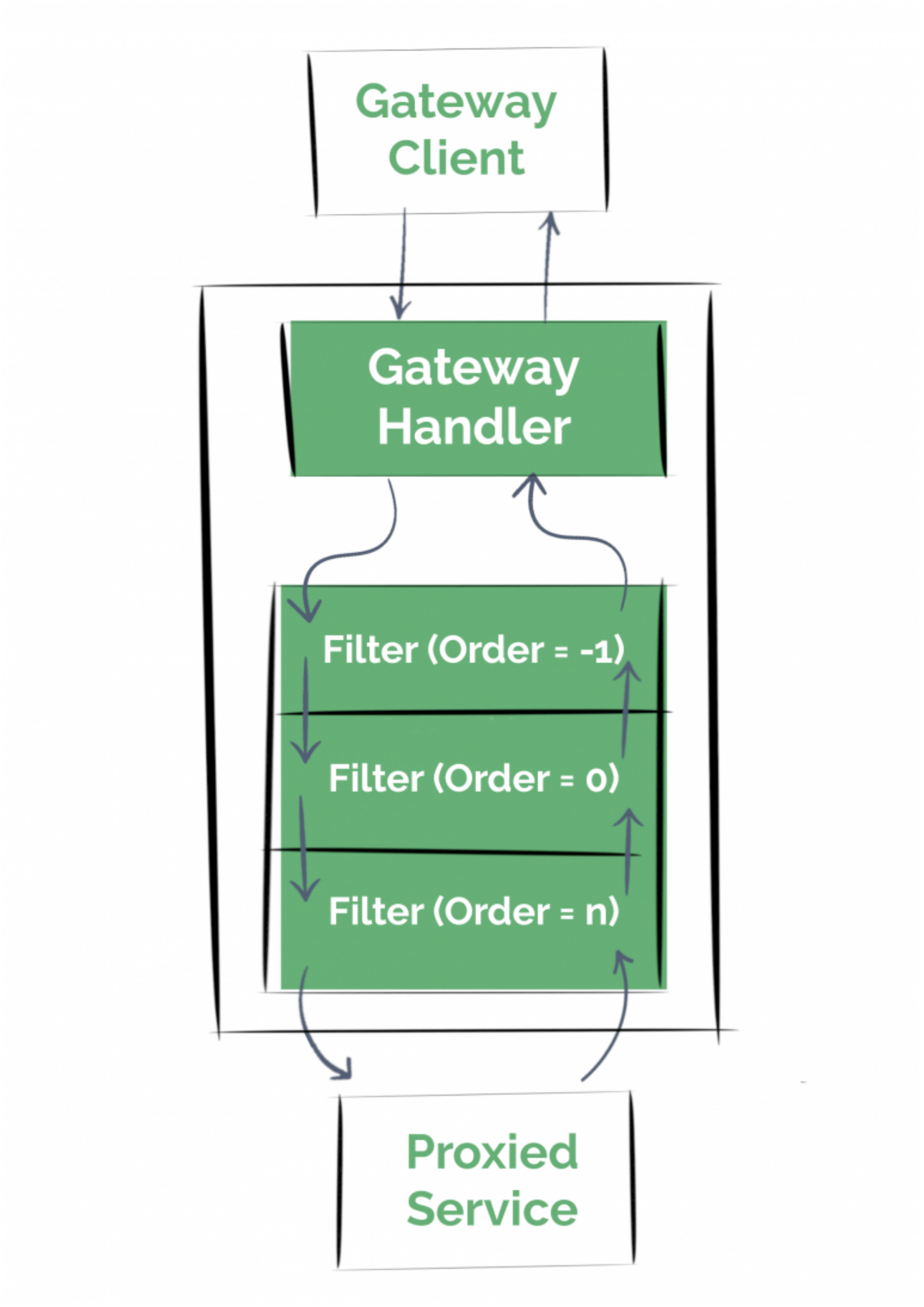
```
@Component
public class FirstPreLastPostGlobalFilter
  implements GlobalFilter, Ordered {

    final Logger logger =
      LoggerFactory.getLogger(FirstPreLastPostGlobalFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
      GatewayFilterChain chain) {
        logger.info("First Pre Global Filter");
        return chain.filter(exchange)
          .then(Mono.fromRunnable(() -> {
              logger.info("Last Post Global Filter");
            }));
    }
    @Override
    public int getOrder() {
        return -1;
    }
}
```

Note we can also implement the *Ordered* interface if we care about the placement of the filter in the chain.

Due to the nature of the filter chain, a filter with lower precedence (a lower order in the chain) will execute its "pre" logic in ar



4. Creating *GatewayFilters*

Global filters are quite useful, but we often need to execute fine-grained custom Gateway filter operations that apply to only some

4.1. Defining the *GatewayFilterFactory*

In order to implement a *GatewayFilter*, we'll have to implement the *GatewayFilterFactory* interface. Spring Cloud Gateway al

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

    final Logger logger =
      LoggerFactory.getLogger(LoggingGatewayFilterFactory.class);

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

    @Override
    public GatewayFilter apply(Config config) {
        // ...
    }

    public static class Config {
        // ...
    }
}
```

Here we've defined the basic structure of our *GatewayFilterFactory*. **We'll use a *Config* class to customize our filter when we init**

In this case, for example, we can define three basic fields in our configuration:

```
public static class Config {
    private String baseMessage;
    private boolean preLogger;
    private boolean postLogger;

    // contructors, getters and setters...
}
```

Simply put, these fields are:

1. a custom message that will be included in the log entry
2. a flag indicating if the filter should log before forwarding the request
3. a flag indicating if the filter should log after receiving the response from the proxied service

And now we can use these configurations to retrieve a *GatewayFilter* instance, which again, can be represented with a lambda fur

```
@Override
public GatewayFilter apply(Config config) {
    return (exchange, chain) -> {
        // Pre-processing
        if (config.isPreLogger()) {
            logger.info("Pre GatewayFilter logging: "
              + config.getBaseMessage());
        }
        return chain.filter(exchange)
          .then(Mono.fromRunnable(() -> {
              // Post-processing
              if (config.isPostLogger()) {
                  logger.info("Post GatewayFilter logging: "
                    + config.getBaseMessage());
              }
          }));
    };
}
```

4.2. Registering the *GatewayFilter* with Properties

We can now easily register our filter to the route we defined previously in the application properties:

```
...
filters:
- RewritePath=/service(?<segment>/?.*), $\{segment}
- name: Logging
  args:
    baseMessage: My Custom Message
    preLogger: true
    postLogger: true
```

We simply have to indicate the configuration arguments. **An important point here is that we need a no-argument constructor ar**

If we want to configure the filter using the compact notation instead, then we can do:

```
filters:
- RewritePath=/service(?<segment>/?.*), $\{segment}
- Logging=My Custom Message, true, true
```

We'll need to tweak our factory a little bit more. In short, we have to override the *shortcutFieldOrder* method, to indicate the order

```
@Override
public List<String> shortcutFieldOrder() {
    return Arrays.asList("baseMessage",
      "preLogger",
      "postLogger");
}
```

4.3. Ordering the *GatewayFilter*

If we want to configure the position of the filter in the filter chain, we can retrieve an *OrderedGatewayFilter* instance from the

```
@Override
public GatewayFilter apply(Config config) {
    return new OrderedGatewayFilter((exchange, chain) -> {
        // ...
    }, 1);
}
```

4.4. Registering the *GatewayFilter* Programmatically

Furthermore, we can register our filter programmatically, too. Let's redefine the route we've been using, this time by setting up a *R*

```
@Bean
public RouteLocator routes(
  RouteLocatorBuilder builder,
  LoggingGatewayFilterFactory loggingFactory) {
    return builder.routes()
      .route("service_route_java_config", r -> r.path("/service/**")
        .filters(f ->
            f.rewritePath("/service(?<segment>/?.*)", "$\\{segment}")
              .filter(loggingFactory.apply(
              new Config("My Custom Message", true, true))))
            .uri("http://localhost:8081"))
      .build();
}
```

5. Advanced Scenarios

So far, all we've been doing is logging a message at different stages of the gateway process.

Usually, we need our filters to provide more advanced functionality. For instance, we may need to check or manipulate the reques

Next, we'll see examples of these different scenarios.

5.1. Checking and Modifying the Request

Let's imagine a hypothetical scenario. Our service used to serve its content based on a *locale* query parameter. Then, we changec

Thus, we want to configure the gateway to normalize following this logic:

1. if we receive the *Accept-Language* header, we want to keep that
2. otherwise, use the *locale* query parameter value
3. if that's not present either, use a default locale
4. finally, we want to remove the *locale* query param

Note: To keep things simple here, we'll focus only on the filter logic; to have a look at the whole implementation we'll find a link to

Let's configure our gateway filter as a "pre" filter then:

```
(exchange, chain) -> {
    if (exchange.getRequest()
      .getHeaders()
      .getAcceptLanguage()
      .isEmpty()) {
        // populate the Accept-Language header...
    }

    // remove the query param...
    return chain.filter(exchange);
};
```

Here we're taking care of the first aspect of the logic. We can see that inspecting the *ServerHttpRequest* object is really simple. At

```
String queryParamLocale = exchange.getRequest()
  .getQueryParams()
  .getFirst("locale");

Locale requestLocale = Optional.ofNullable(queryParamLocale)
  .map(l -> Locale.forLanguageTag(l))
  .orElse(config.getDefaultLocale());
```

Now we've covered the next two points of the behavior. But we haven't modified the request, yet. For this, **we'll have to make use**

With this, the framework will be creating a *Decorator* of the entity, maintaining the original object unchanged.

Modifying the headers is simple because we can obtain a reference to the *HttpHeaders* map object:

```
exchange.getRequest()
  .mutate()
  .headers(h -> h.setAcceptLanguageAsLocales(
    Collections.singletonList(requestLocale)))
```

But, on the other hand, modifying the URI is not a trivial task.

We'll have to obtain a new *ServerWebExchange* instance from the original *exchange* object, modifying the original *ServerHttpReq*

```
ServerWebExchange modifiedExchange = exchange.mutate()
  // Here we'll modify the original request:
  .request(originalRequest -> originalRequest)
  .build();

return chain.filter(modifiedExchange);
```

Now it's time to update the original request URI by removing the query params:

```
originalRequest -> originalRequest.uri(
  UriComponentsBuilder.fromUri(exchange.getRequest()
    .getURI())
  .replaceQueryParams(new LinkedMultiValueMap<String, String>())
  .build()
  .toUri())
```

There we go, we can try it out now. In the codebase, we added log entries before calling the next chain filter to see exactly what is

5.2. Modifying the Response

Proceeding with the same case scenario, we'll define a "post" filter now. Our imaginary service used to retrieve a custom header tc

Hence, we want our new filter to add this response header, but only if the request contains the *locale* header we introduced in the

```
(exchange, chain) -> {
    return chain.filter(exchange)
      .then(Mono.fromRunnable(() -> {
          ServerHttpResponse response = exchange.getResponse();

          Optional.ofNullable(exchange.getRequest()
            .getQueryParams()
            .getFirst("locale"))
            .ifPresent(qp -> {
                String responseContentLanguage = response.getHeaders()
                  .getContentLanguage()
                  .getLanguage();

                response.getHeaders()
                  .add("Bael-Custom-Language-Header", responseContentLanguage);
                });
        }));
}
```

We can obtain a reference to the response object easily, and we don't need to create a copy of it to modify it, as with the request.

This is a good example of the importance of the order of the filters in the chain; if we configure the execution of this filter after the

It doesn't even matter that this is effectively triggered after the execution of all the "pre" filters because we still have a reference tc

5.3. Chaining Requests to Other Services

The next step in our hypothetical scenario is relying on a third service to indicate which *Accept-Language* header we should use.

Thus, we'll create a new filter which makes a call to this service, and uses its response body as the request header for the proxied

In a reactive environment, this means chaining requests to avoid blocking the async execution.

In our filter, we'll start by making the request to the language service:

```
(exchange, chain) -> {
    return WebClient.create().get()
      .uri(config.getLanguageEndpoint())
      .exchange()
      // ...
}
```

Notice we're returning this fluent operation, because, as we said, we'll chain the output of the call with our proxied request.

The next step will be to extract the language – either from the response body or from the configuration if the response was not su

```
// ...
.flatMap(response -> {
    return (response.statusCode()
      .is2xxSuccessful()) ? response.bodyToMono(String.class) : Mono.just(config.getDefaultLanguage());
}).map(LanguageRange::parse)
// ...
```

Finally, we'll set the *LanguageRange* value as the request header as we did before, and continue the filter chain:

```
.map(range -> {
    exchange.getRequest()
      .mutate()
      .headers(h -> h.setAcceptLanguage(range))
      .build();

    return exchange;
}).flatMap(chain::filter);
```

That's it, now the interaction will be carried out in a non-blocking manner.

6. Conclusion

Now that we've learned how to write custom Spring Cloud Gateway filters and seen how to manipulate the request and response

As always, all the complete examples can be found in over on GitHub. Please remember that in order to test it, we need to run inte

Comments are closed on this article!