**Spring Cloud Gateway — Request filtering and redirection**

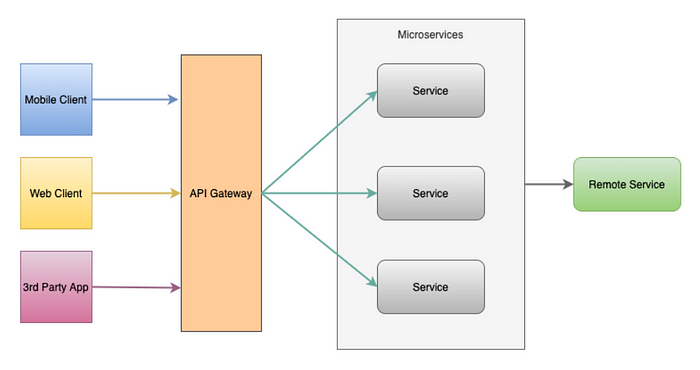
Today I want to talk about an interesting concept of API Gateway, well, to be precise, Spring’s implementation of API Gateway.

**So what is an API Gateway anyway?**

API Gateway is built on top of a **reverse proxy** mechanism and it serves as a single point of entry for multiple client applications that want to communicate with a group of servers behind the gateway. The gateway stands in front of server or a group of servers, hides their identity, and it servers as a single point of communication for all the clients.

Let’s say you have three different applications, a web, a mobile, and a desktop one. Each one of them communicates with maybe 10 different services, and that gets even more complicated if there are different protocols for accessing each one of them. What else could be a problem, and often it is, changes on backend services may mess up your client apps because now you have to update how the communication works, if there is a change in protocol if the response is different, etc.

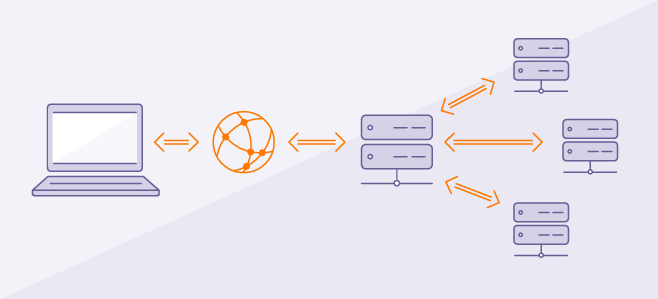
With API Gateway, you have a single entry point for all your applications. Take a look at the picture below:



Here you see that the API Gateway takes care of communicating with services, where some requests are just proxied to the service, and others are filtered, parsed and dispersed to multiple services.

**What does a reverse proxy do?**

Well, a reverse proxy is a type of proxy server that stands in front of servers and acts as a gateway for clients that want to access those servers, where the identity (say IP address) of servers is hidden. Now if you add some monitoring mechanism, security mechanism, and other stuff on top of that, you get an API gateway.



Reverse proxy diagram

**Why?**

Well, the benefits of using an API Gateway are huge, let’s look at some of them:

* Single point of entry
* Unique protocol for all clients
* Clients don’t need to know about multiple server addresses
* Unique response for multiple clients
* Changes on backend services can be hidden from clients
* Multiple platforms
* Services are not exposed
* Load balancing
* Caching
* Monitoring

**Spring Cloud Gateway**

Well, as you probably know, Spring has support for API gateway provided as part of the Spring Cloud ecosystem and relies also on reactive libraries to secure asynchronous support in communication.

Spring Cloud Gateway provides a library for building API gateways on top of Spring and Java.

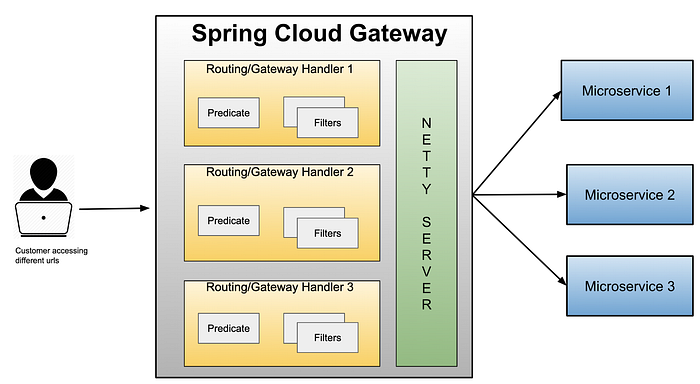
**How does it work?**

It is a Spring Boot application with Spring Cloud stuff that can make it sit between clients and their requests and multiple services, where it offers features such as **predicates** for shaping your gateway routes, the way they can be accessed, **filters** for filtering incoming and outgoing requests, redirecting them, changing their contents, and **route** configurations for your services.

When I say it’s **async**, I mean it, because there is always a thread available to receive your request and put it in a queue for processing, so it’s a **non-blocking** gateway implementation.

So I’ve mentioned three things, routes, filters, and predicates, so here is a brief explanation for them:

* **Route** — contain URL of the service to which request will be forwarded, ad the predicates and filters for the request
* **Predicate** — some set of criteria that needs to match for the incoming request **predicates: — Cookie=chocolate, ch.p**
* **Filter** — components where you can manipulate incoming and outgoing requests and responses

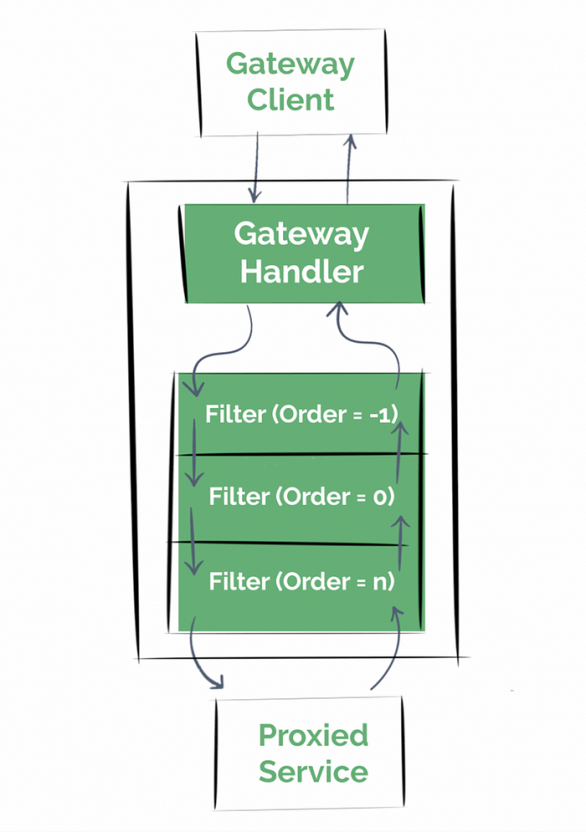


Gateway request flow

* The request comes in to the GatewayHandler
* GatewayHandler decides, based on URI and predicates, it the request matches route
* GatewayHandler moves request to the Gateway Web Handler
* Gateway Web Handler passes the request to the appropriate filter in the filter chain
* The request is proxied to the service

**Gateway filters**

Filters are components where you can modify incoming requests and responses. Filters can be global the for the entire gateway, every request, or tied to a route. Filters can also be “pre” and “post”, since some of them can be activated after the request is proxied to the service. Gateway filters have their order, so for global ones, there is no order since they will be triggered on every request, but for filters that are tied to the route, there is an order in the filter chain.



As you can see here, in the filter chain, each filter is ordered from -1 to n. Filter with the lowest value will be the first to trigger. This is good so that you can organize filters for specific action during the request flow, for example, you can add some headers in one filter, token in other, and redirect request in another one.

There are a number of integrated filters in Spring Cloud Gateway, but what’s cool, it also lets us define our custom filters.

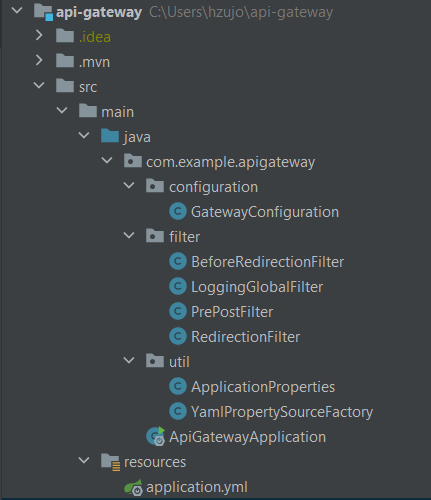
**Demo**

Okay, so I’ve explained some basics on Spring Cloud Gateway, there is more to it [HERE](https://spring.io/projects/spring-cloud-gateway), but now I want to show a demo gateway application, with the following use case:

* An API with two routes
* Client is redirected to either route based on the value received in headers

**The Gateway application**

**Project structure**



Here we have our project structure with following packages:

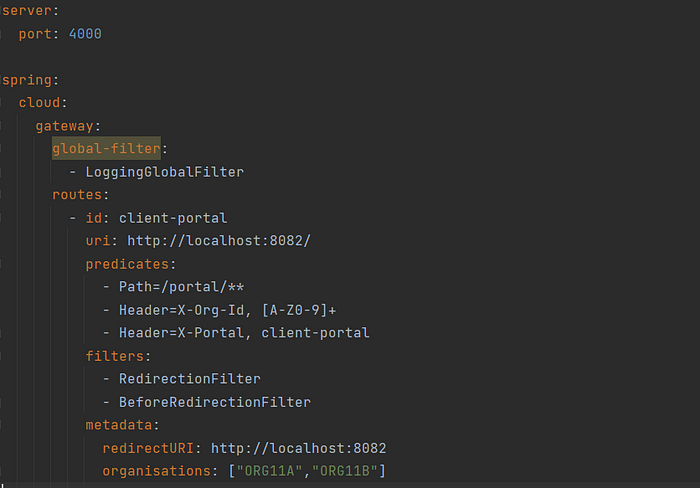
* configuration — I’ll be showing how you can also configure your gateway routes through Java code instead of using YAML files.
* filter — This is where our request and response filters will be, we have 4 filters. “**BeforeRedirectionFilter**” demonstrates ordering and some basic request manipulation that will happen before the “**RedirectionFilter**”. **LoggingGlobalFilter** will log all of our requests (Global filter).  
  The “**PrePost**” filter shows how to act on changes after the request has been sent and there is a response received, and the “**RedirectionFilter**” will do the request manipulation and redirect the request to a different route.
* util — Because we’ll be reading some data from YAML files in our filters, I’ve created some classes to map the values and parse YAML data to POJOs

**Dependencies**

For the gateway application, these are basic dependencies in order to utilize the Spring Cloud Gateway functionalities:

As you can see, there is a spring-cloud-starter-gateway included and some other dependencies such as Lombok processor and configuration processor because we’ll be parsing some YAML configuration files.

**Routes configuration**



As you can see, our gateway will be rolling on port 4000, and will have a global filter for all incoming requests.

There is a routes portion of this configuration where we are describing all routes for our services that will be contacted by our gateway.

What’s really interesting is this predicates part:

* **Path** predicate says that, I’ll match this route with the request only if it starts with /portal/\*\* (wildcard — I don’t care what’s after).
* **Header** — your request headers must contain a header called X-Org-Id with the following regex match which is capital letters and numbers only.  
  There is another header that says that it also must come with a header named X-Portal with the value of client-portal, as I’ll use this to fetch this route by ID when reading from the configuration.
* **Metadata** — I can put anything I want in metadata and read it in our filters later.

You can also configure this through the code, by registering a **RouteLocator** bean:

**Filters**

I’ll take the “BeforeRedirectionFilter” as an example for a route tied filter and the basic filter configuration.

This is a custom filter, so in order to write a custom filter and apply it to a route, you must extend **AbstractGatewayFilterFactory**class where as for the global filters, you must use **GlobalFilter** implementation.

If you’re wondering what’s the Config class doing, well, the superclass requires inner class for configuration of your filter, just in case if you have something specific on the filter level, you can also pass the empty class, it’s important that you declare it.

As you can see, we are returning new **OrderedGatewayFilter** object and passing the value of 0 as a second parameter, this is the catch for the ordering. I’ve set this as my lowest value for filter order, so it means this will be the first route filter to trigger. By retrieving **OrderedGatewayFilter** we are making sure that this filter can be positioned in the filter chain.

There is a lambda expression that has two params **exchange** and **chain**.

Exchange is a **ServerWebExchange** object where we have an access to the request and the response portion. We can always obtain a copy of it and use it to parse the request and modify it.

Chain is a **GatewayFilterChain** object where we call the filter method and pass the request further down the filter chain.

**Redirection Filter**

I’ll present another interesting filter here, where the redirection of the request is being done. Prior to this, we mentioned that we can take a copy of our exchange object and modify the request, well, that is exactly what we are doing here.

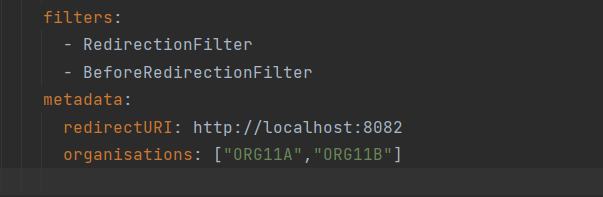
On line 8 you can see that we are mutating the request by accessing the **mutate()** method which returns “Builder” for the request. The important part is on line number 19, where we are accessing the original URL of the exchange. So in order for our redirection to work, especially when the redirection is going to take place on a different server, we must modify this property.

**Testing the app**

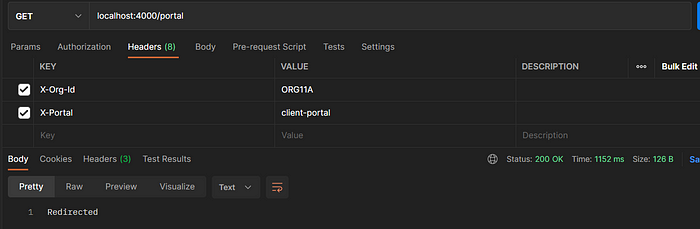


Here I have a simple web application running on port 8082 where our gateway will proxy the requests.

**Original service address:**[**http://localhost:8082,**](http://localhost:8082,/)**but since we are accessing it through our gateway, we’ll access it on port 4000.**

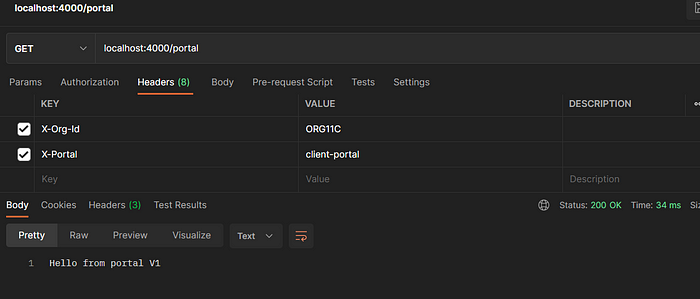


The redirection will happen if the request contains header X-Org-Id with one of the following values “ORG11A” or “ORG11B”, then it will go to the /redirected route, if not, it will be forwarded to the default /portal route only.



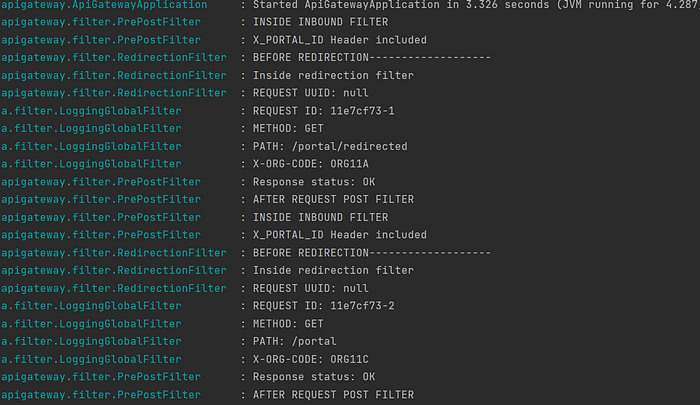
Here is the result, as you can see, it found the value in the route metadata, and matched it with the header value of “ORG11A”, meaning that it can redirect the request, this check was done in our **RedirectionFilter**.

If the value did not match, it will forward it to the designated route.



We should not forget about monitoring, and previously I’ve mentioned the

LoggingGlobalFilter, so here is the output of it:



As you can notice, we have all the request and response details in our filters, and you can notice that there is some “post” logic as well in that PrePostFilter, where we reacted to the response passed from the service.

In this part, we will ignore the processing until the filter chain has ended and the response has been sent back from the server.

As with everything, the API gateway has its own drawbacks such as:

* Performance issues since it is a central point of handling all the traffic
* Security issues
* Single point of failure
* Complexity
* Additional network hop