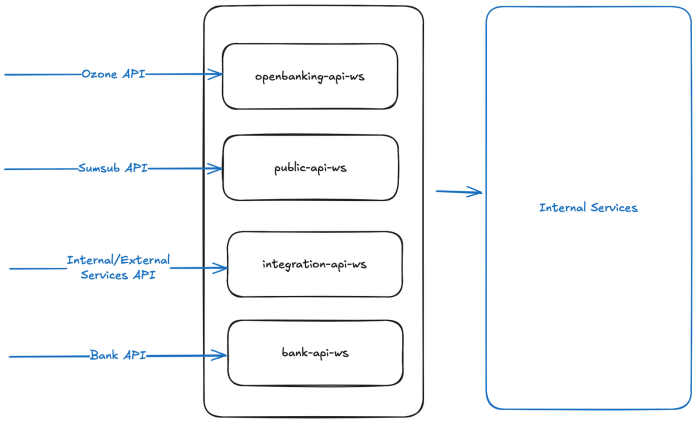
MB_AUTH_SD_Gateway

1. Overview



Generally, to integrate with different parties, regardless internal or external, these services would call to the respective gateway ports, and these gateway will re-route their requests to the dedicated services, with some additional processing. In iGB codebase, it is found that it has 4 modules for dealing with these services, each responsible for different groups of endpoints. However, the reason of having dedicated gateway module (open-banking-ws and public-api-ws) is still confirming with Chee Onn and Kelvin.

iFast Global Bank (iGB)
Repository: https://gitlab.ifastcorp.com/apps/uk/digital-bank

Module	What do they do?
bank-api-ws	Handle the API requests from banks.
integration-api-ws	Handle the API request from different services, both internal and external, e.g. iGV, iGF, Aquila, etc.
openbanking-api-ws	Handle the API request from Dealing with Ozone API for open banking process.
public-api-ws	Handle the API request from Sumsub for EKYC process.

2. Solution Design

2.1. Gateway

api-gateway		
	my-internal-api	For integrating internal services, e.g. FXTS, iGV, iGF, etc.
	my-public-api	For integrating external services, e.g. PayNet, Sumsub, etc.
	common *(suggested changes)*	Defines set of reusable, configuration-driven filters and some convenient utilities.

Generally, these logical groupings (internal-integration-ws, external-integration-ws) is mainly and merely for for grouping purpose only. We can define a one-for-all gateway if we wish, but separating them could preserve more flexibility.

Centralized	Isolated/ Logical group

Regardless of using centralized/ isolated gateways, the design is always extensible and easy to maintain, supporting future enhancement/evolvement as well. just that moving an API endpoint re-routing and processing logic from one gateway to another

gateway might be a breaking change which requires careful treat. So it's advisable to define the modules required clearly before the launch and stick with the designs.

The number of gateways depends on The number of gateways depend on XX factors, where one of them should be performance. The grouping itself is has little meaning.

2.2. Filters/Predicates (Global/Gateway)

2.2.1. Overview

Opinionated Preference:

1. Prefer route-agnostic, configuration-driven, reusable global/ gateway filters as Spring Cloud Gateway's built-in filters do. In other words, a filter shouldn't, or try as best as it could, to keep less route-specific information as possible.
2. Prefer configuration-driven filters first. Followed by @Configuration + @Bean, then only @Component-annotated.

Justification:
Using either ways are fine, but as found in the iGB codebase, some filters, though defined separately, their logic are quite similar, with minor differences lie in the endpoints to be defined. In this case, writing filters for each traffic origin could involves much of the repeated works, and make the logic less expressive, as compared to configuration-driven choice.

Hence, it is personally suggested that the filters should be route-agnostic and configuration drive, meaning that when creating a filter, especially a gateway filter, it should contain minimal route information or none at all. Instead, it should allow for specifying the route information in the YAML file, that's the so-called configuration-driven and route-agnostic. In this way, filters are fully reusable, and the testing could be easier to write. The configuration will also be much easier to read, as all the logic will be clear and straightforward. However, the use of filters that contain route-specific information should be still allowed, but it should be discouraged, unless it's highly specific.

To support the maximum reusability, anyone defines a new endpoints should document what the filter does, what filters doesn't do, and its usage and the related codebase for debugging and enhancement. This solution design aims to serve this purpose.

For reusability, using @Configuration + @Bean will be more suitable than @Component since @Configuration + @Bean allows for flexible importing. However, if all filters in common/ are required across different platform, using @Component with @ComponentScan might be fine for gateway filters, but for Global Filters, it still required @Configuration + @Bean to avoid accidental applying global filters to all gateways.

2.2.2. Spring Cloud Gateway's Filters

Gateway Filters	Description	Example Usage
Path		
SetPath	Replaces the entire request path with a template (supports variables).	SetPath=/{segment} Incoming: /foo/bar → Outgoing: /bar
PrefixPath	Prepends a static prefix to the existing request path.	PrefixPath=/mypath Incoming: /foo/bar → Outgoing: /mypath/foo/bar
StripPrefix	Removes a fixed number of leading path segments.	StripPrefix=2 Incoming: /api/v1/customers/123 → Outgoing: /customers/123
RewritePath	Uses regex + replacement to flexibly rewrite paths.	RewritePath=/foo/(?.*), /${segment} Incoming: /foo/bar/baz → Outgoing: /bar/baz
Notes: Usually, RewritePath will be considered as the "one-for-all" silver bullet solution for all the scenarios, but it might be error prone and hard to read sometimes. Use SetPath, PrefixPath, or StripPath whenever possible to make it clearer.		
Request (Header)		
AddRequestHeader	Adds a header to the request (appends value if header already exists).	AddRequestHeader=X-Request-Red, Blue Before: *(no header)* → After: X-Request-Red: Blue
AddRequestHeader IfNotPresent		

RemoveRequestHeader	Removes a header from the request before forwarding downstream.	RemoveRequestHeader=X-Request-Foo Before: X-Request-Foo: bar → After: *(header removed)*
SetRequestHeader	Sets (replaces) a header with a specific value (overwrites existing one).	SetRequestHeader=X-Request-Red, Blue Before: X-Request-Red: Green → After: X-Request-Red: Blue
MapRequestHeader	Copies the value of one request header to another.	MapRequestHeader=Blue, X-Request-Red Before: Blue: abc→ After: Blue: abc, X-Request-Red: abc
Request (Parameter)		
AddRequestParameter	Adds a query parameter to the request. If the parameter already exists, it appends another value.	AddRequestParameter=foo, bar Before: /products → After: /products?foo=bar Before (if param exists): /products?foo=abc → After: /products?foo=abc&foo=bar
RemoveRequestParameter	Removes a specific query parameter from the request.	RemoveRequestParameter=red Before: /search?red=123&blue=456 → After: /search?blue=456
Response		
AddResponseHeader	Adds a header to the **response**. (It appends; it does not replace existing values.)	AddResponseHeader=X-Response-Red, Blue ● Before: *(no X-Response-Red)* → After: X-Response-Red: Blue ● Before: X-Response-Red: Green → After: X-Response-Red: Green, Blue
RemoveResponseHeader	Removes the named **response** header before returning to the client	RemoveResponseHeader=X-Response-Foo Before: X-Response-Foo: bar → After: *(header removed)*
SetResponseHeader	Sets (replaces) the **response** header with the given value; if absent, it adds it. All existing values under that name are replaced.	SetResponseHeader=X-Response-Red, Blue ● Before: X-Response-Red: Green → After: X-Response-Red: Blue ● Before: *(no X-Response-Red)* → After: X-Response-Red: Blue
DedupeResponseHeader	Deduplicates values of one or more **response** headers. Optional strategy: RETAIN_FIRST (default), RETAIN_LAST, RETAIN_UNIQUE. Commonly used to clean up duplicate CORS headers.	DedupeResponseHeader=Access-Control-Allow-Origin Access-Control-Allow-Credentials (default strategy) ● Before: Access-Control-Allow-Origin: * , * → After: Access-Control-Allow-Origin: * ● Before: Access-Control-Allow-Credentials: true , true →

		After: Access-Control-Allow-Credentials: true
RewriteResponseHeader	Uses **regex** (name, regexp, replacement) to rewrite a **response** header's value	RewriteResponseHeader=X-Response-Red, password=[^&]+, password=*** Before: X-Response-Red: token=abc&password=secret123&role=user → After: X-Response-Red: token=abc&password=***&role=user
Security		
RequestHeaderSize	Rejects requests whose **any single header** exceeds a max size; returns **431 Request Header Fields Too Large**. You can optionally expose an error message header via errorHeaderName.	RequestHeaderSize=1000B Before: header X-Big is **1200B** → After: **431** returned; response includes error message header (default name errorMessage).
RequestSize	Blocks requests whose **body size**exceeds maxSize; returns **413 Payload Too Large** and sets an errorMessageheader.	RequestSize=5MB Before: upload **6 MB** to /upload → After: **413** with errorMessage: Request size is larger than permissible limit….
TokenRelay	Forwards the **OAuth2 access token** from the authenticated user (or specified client registration) to downstream services via Authorization: Bearer …. Supports optional clientRegistrationId.	TokenRelay Before (incoming): Authorization: Bearer abc123→ After (downstream): Authorization: Bearer abc123 forwarded to backend.
RequestRateLimiter	Uses a RateLimiter (Redis or Bucket4j, etc.) to allow/deny requests. **Denied requests return 429 Too Many Requests**.	https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webmvc/filters/ratelimiter.html
Resilience		
CircuitBreaker	Wraps the route call with Spring Cloud CircuitBreaker (Resilience4J by default). Optionally **forwards to a fallback URI** when tripped.	https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webmvc/filters/circuitbreaker-filter.html
FallbackHeaders	When a circuit-breaker fallback route is hit, this	https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-

	filter **adds exception details**(type/message and root cause) as headers to the forwarded **fallback** request. Header names are configurable.	gateway-server-webmvc/filters/fallback-headers.html
Retry	Retries failed downstream calls per policy (e.g., **retries**, **statuses**, **methods**, optional **backoff/jitter/timeout**).	https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webmvc/filters/retry.html
LoadBalancer	Resolves lb://serviceId to a concrete **service instance** (host:port) via Spring Cloud LoadBalancer.	https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webmvc/filters/loadbalancer.html
Others		
PreserveHostHeader	Preserves the **original Host** header from the incoming request instead of letting the HTTP client/runtime change it. No parameters.	PreserveHostHeaderBefore (incoming → outgoing): Host: client.example.com → *(would normally become)* Host: backend.internalAfter (with filter): Host: client.example.com is forwarded to the backend. (Home)
RedirectTo	Issues an HTTP redirect. Parameters: status (3xx) and url (optionally include request params in WebFlux via includeRequestParams).	RedirectTo=302, https://acme.orgBefore: request to /oldAfter: response **302** with Location: https://acme.org. (Home)
SecureHeaders	Adds recommended **security headers**to the response (e.g., HSTS, X-Frame-Options, X-Content-Type-Options, Referrer-Policy). You can disable/tune specific headers.	SecureHeaders=disable=x-frame-options Before: no security headers After: headers like Strict-Transport-Security, X-Content-Type-Options, etc., are present (with X-Frame-Options disabled).
RewriteLocationResponseHeader	Rewrites the **Location** header in responses (typically from backend redirects) to hide backend-specific details; params include mode and host/protocol settings.	*(See docs for full args)* Before: Location: http://backend.internal/app/login After: Location: https://public.example.com/login
SetStatus	Sets the **HTTP status** on the response; accepts an int (e.g., 404) or enum name (e.g., NOT_FOUND). (Home)	SetStatus=404 Before: backend would return 200 OK After: gateway returns 404 Not Found**.**
SetRequestHostHeader	Replaces the **Host** header sent to the downstream service with the provided host value. (Home)	SetRequestHostHeader=api.internal.localBefore (outgoing): Host: client.example.comAfter (outgoing): Host: api.internal.local. (Home)

Reference: https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/gatewayfilter-factories.html
There are still some available filters, but they are used for more advanced use cases, which might be overkill for the current use ca
these filters are not shown in the current solution design. If there's a need in the future, then only adding it to the documentation for reference.

2.2.3. Spring Cloud Global's Filters

References: https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/global-filters.html

2.2.3.1. Use `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:

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

2.2.3.1. Use `GlobalFilter` interface

```
@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;
    }
}
```

2.2.3. Spring Cloud Gateway's Predicates

Notes: Among all these predicates, `Path` is the most common and one of the most powerful predicates. In most cases in existing iGB codebase, `Path` is used the most, and the others are rarely been used. To enforce stricter rules, may consider to use `Header` and `Method`.

Predicate	Description	Example
After	Reference: 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.	`After=2017-01-20T17:42:47.789-07:00[America/Denver]`
Before	Reference: 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`.	`Before=2017-01-20T17:42:47.789-07:00[America/Denver]`
Between	Reference: The `Between` route predicate factory takes two	`Between=2017-01-20T17:42:47.789-`

	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`.	`07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]`
Cookie	Reference: 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.	`Cookie=chocolate, ch.p`
Header	Reference: 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.	`Header=X-Request-Id, \d+`
Host	Reference: 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.	`Host=**.somehost.org,**.anotherhost.org`
Method	Reference: The `Method` Route Predicate Factory takes a `methods` argument which is one or more parameters: the HTTP methods to match.	`Method=GET,POST`
Path	Reference: The `Path` Route Predicate Factory takes two parameters: a list of Spring `PathMatcher` `patterns` and an optional flag called `matchTrailingSlash` (defaults to `true`).	`Path=/red/{segment},/blue/{segment}`
Query	Reference: The `Query` route predicate factory takes two parameters: a	`Query=green`

	required `param` and an optional `regexp` (which is a Java regular expression).	
RemoteAddr	Reference: 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).	`RemoteAddr=192.168.1.1/24`

2.2.4. iGB's Filters (Suggested reusable filters)

2.2.5. Writing Custom Predicates and Filters

2.2.6. Others

Configuring Route Predicate Factories and Gateway Filter Factories
Http timeouts configuration
CORS Configuration
Route Metadata Configuration

2.3. Constants

From the iGB's codebase, it is found that there's a lot of constants have been defined, while these constants have been already provided by Spring's Web with more options available and is ensured that it is always compliant with RFC, CORS and other widely-accepted standards. Following these standards makes it easier for us to integrate with other parties in the future, and it's proved to be future-proof as it's widely accepted.

org.springframework.http.	Description	Examples
HttpStatus		HttpStatus.Series.
HttpHeaders		
MediaType		
HttpMethod		

Miscellaneous: UriComponentsBuilders, HttpServletRequest, HttpEntity, RequestEntity, ResponseEntity, RestTemplate, WebClient, DefaultErrorAttributes

Remarks

Biometric Authentication to verify if the users has go through biometric authentication

WebClient vs FeignClient

We should use WebClient.

Currently, my-ifast-pay and other api-gateway (iGB, iGV) are using reactive, NIO (Non Blocking I/O) gateway. However, FeignClient is BIO (Blocking I/O), which might weaken the power of the reactive gateway. To keep the high concurrency and the NIO nature for reactive gateway, we should use WebClient rather than FeignClient.

Technical Documentation regarding the use of WebClient can be found here:
https://sgnas.ifastfinancial.com/drive/oo/r/15HsBaKQcizqfvYYHfBMUejq74HoobZ9

Dimension	**WebClient**	**Feign (Spring Cloud OpenFeign)**
Programming model	Reactive (Mono/Flux). Non-blocking, event-loop	Imperative. Blocking by default
Best fit	WebFlux apps, Spring Cloud Gateway filters, high-concurrency I/O, streaming/SSE	Spring MVC services making simple REST ca
Performance/concurrency	Excellent C10k with small thread pools; backpressure	Thread-per-request; scale by threads/cores
Timeouts/retries/circuit breaker	Very granular (connect/read/response), easy with Resilienc + Reactor retryWhen	Supported via Resilience4j + Feign interceptors; coarser control
Serialization	Codec flexibility (Jackson, byte[], DataBuffer, NDJSON)	Jackson by default; conventional JSON best
Error handling	onStatus, exchangeToMono, map status → domain errors	Error decoders/fallback factories
In Spring Cloud	Natural (reactive). Recommended	Generally discouraged (blocking + bean

Gateway		graph/cycles)
Learning curve	Higher (reactive, operators)	Lower (declarative, MVC-like)

Web Security

Lets understand how the /oauth2/token works. The mechanism is simple.