WebClient Technical Documentation

Introduction

It’s good to document the convention to allow others to understand why a design decision has been made, and what are those conventions, to help developers focus on developing business rules rather than being trapped by the boring, cumbersome technical implementation details and those implicit convention. This document aims to serve as a guide for any codebase that aims to use Non-Blocking I/O (NIO) WebClient.

Convention

No	Convention
1	Each `@Configuration` class should define **exactly one WebClient instance** along with its dedicated dependencies. This ensures clear separation of concerns and easier maintenance.
2	● Filters that are **shared across multiple WebClients** should be placed under the filter/ package to promote reusabili ● Filters that are **specific to a single WebClient** should be defined within that WebClient's dedicated configuration cl maintain separation of concerns.
3	Follow the *Convention over Configuration* principle by relying on *method names as bean names* instead of explicitly nam beans. This reduces boilerplate and improves readability.
4	Avoid using *constructor injection* or *field injection* within configuration classes. Instead, define dependencies ***explicitly a method parameters*** in `@Bean` methods for better visibility and clearer dependency management.

For services that call remote endpoints, the interface should be named ***ApiService***, and its implementation should be named ***ApiServiceImpl***.

Adapter Pattern

There are two main approaches for calling remote services using **WebClient**:

Use WebClient directly	**Use HttpExchange**
Pros This approach provides full flexibility for customizing error handlir on a per-method basis. However, it can be harder to use and ma introduce more boilerplate code.	Pros This approach offers a **FeignClient-like declarative style**, makir more developer-friendly. However, it allows only coarse-grained customization of the WebClient, which means fine-tuning behavic the method level is generally not feasible.
Cons	Cons

However, to deal with complex or edge-case scenarios and evolving business need, it's always good to consider a hybrid approach, use each of them whenever necessary. To address this, it is proposed to introduce an abstraction layer called ApiServiceImpl that delegates all its operations to either HttpExchange, or customized using WebClient to hide underlying complexities while maintaining flexibility and long-term maintainability. Although this approach slightly increases development effort, the overhead is minimal and almost negligable because **ApiServiceImpl** mainly acts as a delegator.

In most cases, where **HttpExchange** is used, **ApiServiceImpl** may seem unnecessary. However, it serves as a valuable **extension point** for future enhancements, ensuring the system remains adaptable to evolving business needs.

Also, with delegation pattern, we can perform resilience fine-tuning on method level, such as handle the different types of errors at the service

impl manually, or with the webClient.onErrorResume more easily and explicitly, suiting most developers' development experience.

Recommended Usage

- **Default approach:** Use **ExchangeApiService** for most scenarios, as it simplifies development through its declarative style.
- **Exception cases:** When fine-grained, method-level tuning is truly required, create an **ApiServiceImpl** and use **WebClient** directly within it.

This hybrid approach achieves an optimal balance between **development productivity** and **flexibility**.
In theory, there should be minimal need for method-level fine-tuning, but the design remains open to accommodate it if necessary.

Defining a WebClient

Method	Purpose	Typical Use Case
`embeddedValueResolver()`	Resolves ${} placeholders and SpEL	Inject base URLs or tokens from config
`customArgumentResolver()`	Handles custom parameter binding	Support complex DTOs as method arguments
`conversionService()`	Handles type conversions	Custom enum/date/string mappings
`exchangeAdapter()`	Defines underlying HTTP engine	Choose WebClient (reactive) or RestClient (blocking)

If you are using SPEL in HttpExchange, you should defined the embeddedValueResolver. The common use case is to resolve the path. However, there are two approaches to handle it, though the first approach is generally more recommended because it keeps the Httpexchange interface simpler and ensure the reusability if WebClient is used independently without HttpExchange interface.

1	Use Environment.getProperty() to resolve it in baseUrl without the need to register a embeddedValueResolver()
	`@Bean`
	`public WebClient authWebClient(WebClient.Builder defaultWebClientBuilder,`
	`Environment environment,`
	`ExchangeFilterFunction authWebClientHeaderFilter) {`
	`return defaultWebClientBuilder`
	`.baseUrl(Objects.requireNonNull(environment.getProperty("mm-gateway.url")))`
	`.filter(authWebClientHeaderFilter)`
	`.build();`
	`}`

2	Register Environment.resolvePlaceaHolders() as the embeddedValueaResolver.
	`@Bean`
	`public AuthApiService authApiService(WebClient authWebClient, Environment environment) {`

```
        HttpServiceProxyFactory factory = HttpServiceProxyFactory.builder()
            .embeddedValueResolver(environment::resolvePlaceholders)
            .exchangeAdapter(WebClientAdapter.create(authWebClient))
            .build();
        return factory.createClient(AuthExchangeApiService.class);
    }
```

HttpExchange

To promote decoupling, the proposed solution it to have a tech-agnostic ApiService and the tech-aware implementation, i.e., ExchangeApiService. However, the ExchangeApiService doesn't aware of which WebClient to use, so we have to let them know explicitly.

```
@Configuration
public class AuthWebClientConfig {

    @Bean
    public WebClient authWebClient(WebClient.Builder defaultWebClientBuilder,
                                   Environment environment,
                                   ExchangeFilterFunction authWebClientHeaderFilter) {
        return defaultWebClientBuilder
            .baseUrl(Objects.requireNonNull(environment.getProperty("mm-gateway.url")))
            .filter(authWebClientHeaderFilter)
            .build();
    }

    @Bean
    public AuthApiService authApiService(WebClient authWebClient, Environment environment) {
        HttpServiceProxyFactory factory = HttpServiceProxyFactory.builder()
            .exchangeAdapter(WebClientAdapter.create(authWebClient))
            .build();
        return factory.createClient(AuthExchangeApiService.class);
    }

    @Bean
    public ExchangeFilterFunction authWebClientHeaderFilter() {
        return ExchangeFilterFunction.ofRequestProcessor(Mono::just);
    }
}
```

Fallback

We should always consider the failing case, hence the design should consider the fallback and error handling as well. With the previously mentioned Adapter Pattern, the fallback can be done using WebClient only. Generally, we use Mono<T> with WebClient. Mono<T> provides the following error handling operations:

1. Recover from the error by switching to another Mono or value
2. Transform the error into a different error
3. Suppress error under certain conditions or convert it into an empty completion
4. Execute side-effects on error or termination

Operator	Behaviour / Purpose
`onErrorResume(Function<Throwable, Mono<? extend T>> fallback)`	Catch any error (or a subset) and switch to a fallback Mono.
`onErrorResume(Predicate<Throwable>, Function<Throwable, Mono<? extends T>> fallback`	Only when the predicate matches, fallback; else propagate error.
`onErrorReturn(T fallbackValue)`	On any error, instead of failing, emit a constant fallback value.
`onErrorReturn(Class<E>, T fallbackValue)`	Only if the error is of type E (or subclass), return fallback; else erro
`onErrorComplete()`	On error, quietly complete (i.e. treat the error as a silent terminatior
`onErrorComplete(Predicate<Throwable>)`	Conditionally complete (suppress error) for errors matching the predicate.
`onErrorMap(Function<Throwable, Throwable> mappe`	Transform the error into another exception type.

`timeout(Duration timeout)`	If no element or error is emitted before the timeout, emit a TimeoutException error.
`timeout(Duration timeout, Mono<? extends T> fallback)`	Similar, but if timeout happens, switch to fallback Mono instead of e

Common Use Case

```
webClient
    .get()
    .uri("/user/{id}", id)
    .retrieve()
    .bodyToMono(User.class)
    .onErrorResume(error -> Mono.just(new User("fallback", "0")));

webClient
    .get()
    .uri("/user/{id}", id)
    .retrieve()
    .bodyToMono(User.class)
    .onErrorResume(WebClientResponseException.class, ex -> {
         if (ex.getStatusCode().is5xxServerError()) {
             return fallbackMono;
         } else {
             return Mono.error(ex);  // rethrow for 4xx etc
         }
     });
```

```
return webClient
    .get()
    .uri("/user/{id}", id)
    .retrieve()
    .bodyToMono(User.class)
    .onErrorReturn(error -> Mono.just(new User("fallback", "0")));
```

Notes:

- Reference: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html
- Ask chatGPT/ Gemini for more examples on the error handling.