**Resttemplate vs webclient**

**Why RestTemplate Got Deprecated:**

1. **Blocking Nature:** RestTemplate is a blocking, synchronous client. It means that the thread that executes the request blocks until the operation completes, potentially leading to thread pool exhaustion and higher latency under heavy load. This model doesn’t scale well, especially in microservices environments where applications must handle thousands of concurrent requests efficiently.
2. **Limited Scalability:** The synchronous nature of RestTemplate limits scalability. Modern systems, which require high-throughput, low-latency capabilities, find this approach inadequate. The rise of event-driven, reactive programming paradigms is a response to these demands, leading to the adoption of non-blocking APIs like WebClient.
3. **Lack of Reactive Programming Support:** RestTemplate doesn’t support reactive programming, a growing necessity in cloud-based ecosystems. Reactive programming allows systems to be more responsive, resilient, and elastic, but this is unachievable with RestTemplate’s blocking nature.

**The Rise of WebClient:**

WebClient is part of the Spring WebFlux library, introduced with Spring 5. It offers numerous advantages over RestTemplate:

1. **Non-Blocking Operations:** WebClient operates on a non-blocking, reactive paradigm using Project Reactor, allowing it to handle concurrency with fewer threads and less overhead, significantly improving scalability and resource utilization.
2. **Reactive Stack:** WebClient supports the reactive stack, making it suitable for event-loop based runtime environments. It can work efficiently under high concurrency scenarios typical in microservices architectures.
3. **JSON Processing and More:** WebClient provides seamless integration with JSON through the Jackson library, similar to RestTemplate, but with enhanced processing capabilities. It also supports server-sent events (SSE), streaming scenarios, and other advanced use cases.

**Transitioning from RestTemplate to WebClient:**

Let’s explore how to replace RestTemplate with WebClient through examples. We’ll start with a simple GET request.

**Performing a GET Request:**

**RestTemplate:**

RestTemplate restTemplate = new RestTemplate();  
ResponseEntity<String> response = restTemplate.getForEntity("http://example.com", String.class);

**WebClient:**

WebClient webClient = WebClient.create();  
Mono<String> response = webClient.get()  
 .uri("http://example.com")  
 .retrieve()  
 .bodyToMono(String.class);  
response.subscribe(result -> System.out.println(result));

In WebClient, everything is non-blocking. The retrieve() method initiates the request, and bodyToMono converts the response body into a Reactor Mono. The subscribe() method is used to subscribe to the result, which will be processed once available.

**Handling Errors:**

RestTemplate’s error handling occurs through the ErrorHandler interface, which requires a separate block of code. WebClient streamlines this with more fluent handling.

**WebClient:**

WebClient webClient = WebClient.create();  
webClient.get()  
 .uri("http://example.com/some-error-endpoint")  
 .retrieve()  
 .onStatus(HttpStatus::isError, response -> {  
 // Handle error status codes  
 return Mono.error(new CustomException("Custom error occurred."));  
 })  
 .bodyToMono(String.class);

The onStatus() method allows for handling specific HTTP statuses directly within the chain of operations, providing a more readable and maintainable approach.

**POST Request with JSON:**

When making a POST request and submitting JSON, WebClient offers a more straightforward approach with its fluent API.

**RestTemplate:**

RestTemplate restTemplate = new RestTemplate();  
HttpHeaders headers = new HttpHeaders();  
headers.setContentType(MediaType.APPLICATION\_JSON);  
HttpEntity<String> request = new HttpEntity<>("{\"key\":\"value\"}", headers);  
ResponseEntity<String> response = restTemplate.postForEntity("http://example.com", request, String.class);

**WebClient:**

WebClient webClient = WebClient.create();  
Mono<String> response = webClient.post()  
 .uri("http://example.com")  
 .contentType(MediaType.APPLICATION\_JSON)  
 .bodyValue("{\"key\":\"value\"}")  
 .retrieve()  
 .bodyToMono(String.class);

With WebClient, setting headers and the body content is more intuitive and requires less boilerplate code. The contentType() and bodyValue() methods allow for setting the content type and body directly.

**Asynchronous Processing:**

One of the most significant advantages of using WebClient is its support for asynchronous processing. This is particularly useful when your application needs to make multiple independent API calls; these can be executed concurrently, drastically reducing the total time needed for these operations.

WebClient webClient = WebClient.create();  
Mono<String> responseOne = webClient.get()  
 .uri("http://example.com/endpointOne")  
 .retrieve()  
 .bodyToMono(String.class);  
  
Mono<String> responseTwo = webClient.get()  
 .uri("http://example.com/endpointTwo")  
 .retrieve()  
 .bodyToMono(String.class);  
  
// Use Mono.zip to execute requests concurrently  
Mono.zip(responseOne, responseTwo).subscribe(results -> {  
 System.out.println("Result 1: " + results.getT1());  
 System.out.println("Result 2: " + results.getT2());  
});

In this example, Mono.zip is used to combine the results of multiple requests. These requests are executed concurrently, and the results are processed once all requests are completed. This approach is far more efficient than the sequential execution inherent in RestTemplate's synchronous operations.

**Streaming Data:**

WebClient also supports retrieving response bodies as a stream of data, which is particularly useful when dealing with large amounts of data that you don’t want to hold in memory all at once.

WebClient webClient = WebClient.create();  
webClient.get()  
 .uri("http://example.com/stream")  
 .accept(MediaType.TEXT\_EVENT\_STREAM) // for Server-Sent Events (SSE)  
 .retrieve()  
 .bodyToFlux(String.class) // convert the response body to a Flux  
 .subscribe(data -> System.out.println("Received: " + data));

In this scenario, bodyToFlux is used to convert the response body into a Flux, which represents a stream of data. The subscribe method is then used to process each piece of data as it arrives. This is a stark contrast to RestTemplate, which would require the entire response body to be loaded into memory before processing, regardless of size.

**Retry Mechanism:**

WebClient provides a more sophisticated approach to retrying failed requests, leveraging the reactive programming model.

WebClient webClient = WebClient.builder().baseUrl("http://example.com").build();  
Mono<String> response = webClient.get()  
 .uri("/retry-endpoint")  
 .retrieve()  
 .bodyToMono(String.class)  
 .retryWhen(Retry.backoff(3, Duration.ofSeconds(1)) // number of retries and backoff configuration  
 .maxBackoff(Duration.ofSeconds(10))) // maximum backoff time  
 .onErrorResume(e -> Mono.just("Fallback response")); // fallback if retries all fail  
  
response.subscribe(result -> System.out.println(result));

In this example, the retryWhen method is used to define a retry strategy, specifying the number of retries and a backoff configuration. If all retries fail, onErrorResume provides a fallback mechanism.

**Custom WebClient Configuration:**

WebClient is highly configurable, and you might find yourself in situations where the default settings don’t fit your needs. For instance, you might want to tweak the connection timeout, or you might need to add default headers that should be sent with every request.

// Build a custom WebClient with specified timeout and default headers  
WebClient customWebClient = WebClient.builder()  
 .baseUrl("http://example.com")  
 .clientConnector(new ReactorClientHttpConnector(  
 HttpClient.create()  
 .option(ChannelOption.CONNECT\_TIMEOUT\_MILLIS, 2000) // 2 seconds timeout  
 .responseTimeout(Duration.ofSeconds(2)) // Set response timeout  
 .doOnConnected(conn ->   
 conn.addHandlerLast(new ReadTimeoutHandler(2)) // 2 seconds read timeout  
 .addHandlerLast(new WriteTimeoutHandler(2))))) // 2 seconds write timeout  
 .defaultHeader(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE) // Default header  
 .defaultHeader("Another-Header", "Another-Value") // Another default header  
 .build();

In this example, we’re customizing WebClient to have a specific timeout configuration and default headers. This instance of WebClient will apply these settings to all requests it executes, ensuring consistent behavior across your application.

**WebClient Filters:**

WebClient supports the use of filters for cross-cutting concerns. These filters can be used to manipulate the request or response, or even to handle concerns like logging, metrics, or authorization.

// Custom WebClient with a filter  
WebClient filteredWebClient = WebClient.builder()  
 .baseUrl("http://example.com")  
 .filter((request, next) -> {  
 // Log request data  
 System.out.println("Request: " + request.method() + " " + request.url());  
 return next.exchange(request).doOnSuccessOrError((response, error) -> {  
 if (response != null) {  
 // Log response data  
 System.out.println("Response Status: " + response.statusCode());  
 }  
 if (error != null) {  
 // Log error  
 System.out.println("Error: " + error.getMessage());  
 }  
 });  
 })  
 .build();

This filter logs the HTTP method and URL for every request made through this WebClient instance, and the status code for every response received. It also logs any error that might occur during the exchange.

**Mutual TLS Authentication:**

In scenarios where you need heightened security, like internal microservices communication, you might require mutual TLS (mTLS) authentication. WebClient can be configured for such scenarios.

// Preparing the SSL context with both trust store and key store  
SslContext sslContext = SslContextBuilder  
 .forClient()  
 .trustManager(InsecureTrustManagerFactory.INSTANCE) // for demo purposes only!  
 .keyManager(new File("path/to/client.crt"), new File("path/to/client.key")) // client certificate and private key  
 .build();  
  
// Configuring the WebClient with the SSL context  
WebClient secureWebClient = WebClient.builder()  
 .clientConnector(new ReactorClientHttpConnector(HttpClient.create().secure(sslContextSpec -> sslContextSpec.sslContext(sslContext))))  
 .build();

In this example, we’re setting up mTLS by configuring the WebClient with an SSL context that includes both the client’s certificate and private key. This setup ensures both the client and server authenticate each other during the SSL handshake.

**WebClient Best Practices**

Here are some guidelines to ensure you’re using WebClient efficiently and effectively:

**Singleton Pattern:**

Unlike RestTemplate, which was often instantiated per request or service, WebClient is designed to be used as a singleton. This means you should create a single instance of WebClient and reuse it across your application. This approach ensures efficient resource utilization and avoids the overhead of repeatedly creating and destroying WebClient instances.

@Bean  
public WebClient.Builder webClientBuilder() {  
 return WebClient.builder();  
}

By defining a WebClient.Builder bean in your configuration, you can autowire it wherever you need and customize it for specific use cases without creating a new WebClient instance each time.

**Error Handling:**

Reactive streams propagate errors downstream until they’re handled or they reach the end of the stream. Always handle errors in your reactive chains to avoid unexpected behaviors. The onErrorResume, onErrorReturn, and doOnError operators can be particularly useful.

webClient.get()  
 .uri("/endpoint")  
 .retrieve()  
 .bodyToMono(String.class)  
 .doOnError(e -> log.error("Error occurred", e))  
 .onErrorResume(e -> Mono.just("Fallback value"));

**Timeout Configuration:**

Always configure a timeout. Without a timeout, a WebClient request might hang indefinitely if the server doesn’t respond. Use the timeout operator to set a specific duration after which the request will be terminated.

webClient.get()  
 .uri("/endpoint")  
 .retrieve()  
 .bodyToMono(String.class)  
 .timeout(Duration.ofSeconds(10));

**Backpressure:**

One of the core principles of reactive programming is backpressure, which allows consumers to signal to producers how much data they can handle. When dealing with Flux (a stream of 0 to N items), be aware of backpressure and ensure you're not overwhelming consumers. Use operators like limitRate to control the rate of data flow.

**Logging:**

Logging is crucial for debugging and monitoring. WebClient provides built-in logging capabilities. By setting the logger reactor.netty.http.client.HttpClient to DEBUG, you can view detailed logs of your requests and responses.

**Thread Context:**

In reactive programming, operations might switch threads multiple times. If you rely on thread-local variables (like those used in logging or security contexts), be aware that these might not be automatically propagated across threads. Libraries like reactor-context can help propagate context across threads in a reactive flow.

**Avoid Blocking Calls:**

One of the main benefits of WebClient and reactive programming is the non-blocking nature of operations. However, this benefit is negated if you introduce blocking calls within your reactive chains. Always avoid blocking operations within your reactive streams. If you must use a blocking call, consider offloading it to a separate thread pool using subscribeOn.

Mono.fromCallable(() -> blockingMethod())  
 .subscribeOn(Schedulers.boundedElastic());