Introduction to Spring WebClient

What is Spring WebClient?

In simple words, **the Spring WebClient is a component that is used to make HTTP calls to other services. It is part of Spring's web reactive framework, helps building reactive and non-blocking applications**.

To make HTTP requests, you might have used Spring Rest Template, which was simple and always blocking web client. However, Spring has announced it will deprecate the RestTemplate in near future for the new WebClient alternative.

Also, being reactive, the WebClient supports non-blocking web requests using the full features of Spring's Webflux library. Most importantly, we can also use WebClient in blocking fashion, where the code will wait for

the request to finish before progressing further. Having support to non-blocking reactive streams, the web client can decide whether to wait for the request to finish or proceed with other tasks.

WebClient Dependency

The Spring WebClient ships in the Webflux library. In order to use WebClient in a Spring Boot Project we need add dependency on WebFlux library. Like any other Spring Boot dependencies, we have to add a starter dependency (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-webflux) for WebFlux (spring-boot-starter-webflux).

Maven Dependency

For Maven built projects, add the starter dependency for WebClient in pom.xml File.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Gradle Dependency

Or, add the starter dependency in a Gradle built project, through build.gradle file.

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-webflux'
}
```

Build WebClient Instance

Building a WebClient instance is easy and flexible. That is because, the WebClient provides three different ways to build a WebClient. Thus, we can flexibly use the most convenient way for our use cases.

Method #1

At the most basic, we can **create WebClient instance using its *create()* factory method**.

```
WebClient webClient = WebClient.create();
```

This Web Client instance can now make requests, by providing further details of HTTP method and URL etc.

Method #2

Alternatively, a more flexible way is to **create a Web Client instance by specifying the base url of the upstream service**.

```
WebClient webClient = WebClient
    .create("http://localhost:9192");
```

Using this way, we can create a common WebClient for each upstream service.

```
@Bean
public WebClient webClient(){
    return WebClient
        .create("http://localhost:9192");
}
```

Next, we can use such common instance anywhere to execute a specific resources on the base url.

```
WebClient.ResponseSpec responseSpec =
    webClient
        .get()
        .uri("/users/" + userId)
        .retrieve();
```

Method #3

Finally, the most flexible way of creating a WebClient instance is to use its own builder (*WebClient.Builder*). The builder, is a place to do all the common configurations. Once created, we can reuse the builder to instantiate multiple Web Client instances. That helps us avoiding reconfiguring all the client instances.

Create WebClient Builder with common configurations.

```
@Bean
public WebClient.Builder webClientBuilder() {
    return WebClient.builder()
        .baseUrl("http://localhost:9192")
        .defaultHeaders(header ->
            header.setBasicAuth(userName, password)
        )
        .defaultCookie(DEFAULT_COOKIE, COOKIE_VALUE);
}
```

Once this is done, we can reuse the WebClient Builder to create a Web Client.

```
WebClient webClient = webClientBuilder.build();
```

Execute Requests with WebClient

As can be seen above, we can build Spring WebClient instance in multiple ways. Once created we can use the WebClient instance to make HTTP GET, POST, PUT etc. requests.

HTTP Method

For example, next is **configuring the WebClient instance to make a POST request**.

```
webClient.post()
```

Alternatively, we can **use *method()* method and pass the HTTP method we want to use**.

```
webClient.method(HttpMethod.POST)
```

URI

Now it is time to specify the URI of the target endpoint. Note that we can specify the base URL while creating WebClient instance and now can just pass the resource identifier.

First is an **example of passing an instance of java.net.URI class**.

```
webClient
    .post()
    .uri(URI.create("/users/" + userId))
```

Alternatively, **we can also pass the URI as a String**.

```
webClient
    .post()
    .uri("/users/" + userId)
```

Or, **specify URI by using URIBuilder function**.

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
```

HTTP Headers

In order to add HTTP Headers to the request, we can use header() method. Remember, we can add all common headers while creating WebClient instance. However, if there are headers specific to the request, we can add them to each of the requests separately.

For example, **add headers to the request**.

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
```

Alternatively, the API also provides factory methods to build common headers. For example **using header factory methods to add specific headers**.

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .accept(MediaType.APPLICATION_JSON)
```

Request Body

Similarly, we can add request body to the Web Client Request. Also, there are a multiple ways to add request body to the Web Client requests.

Firstly, the simplest way is to **specify request body using bodyValue() method**.

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .bodyValue(userStr)
```

Or, **use body() method to specify a publisher and type of the published element**.

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .body(Mono.just(userObj), User.class);
```

Alternatively, for a more advanced scenarios we can use *BodyInserters*. Because, it provides more flexibility on how we add body to our requests (multi part requests, form data, etc.).

For example, **using BodyInserters to add request body from a publisher.**

```
webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromPublisher(Mono.just(userObj)))
```

Execute Request

Finally, we will execute the request and read server response or errors. In order to do that, we can simply use *retrieve()* method and then convert the response to a Mono or a Flux.

For example, **use *retrieve()* method and covert response to a Mono.**

```
Mono<User> response = webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromPublisher(Mono.just(userObj)))
    .retrieve()
    .bodyToMono(User.class);
```

Also, we can handle any server or client errors by attaching *onStatus()* method. For example, **using *onStatus()* method to throw exception when server response HTTP status indicates failure**.

```
Mono<User> response = webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromPublisher(Mono.just(userObj)))
    .retrieve()
    .onStatus(
        Predicate.not(HttpStatus::is2xxSuccessful), clientResponse ->
            error(new ApplicationException(ERROR_MSG))
    )
    .bodyToMono(User.class);
```

Alternatively, for a more controlled response handling, we can use exchange while executing request. Also, by using that we get option to transform body into Flux (*exchangeToFlux()*) or Mono (*exchangeToMono()*).

Example of **using exchangeToMono to read the response and ensure status if 200**.

```
Mono<ClientResponse> response = webClient
    .post()
    .uri(uriBuilder ->
        uriBuilder.pathSegment("users", "{userId}")
            .build(userId))
    .contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromPublisher(Mono.just(userObj)))
    .exchangeToMono(result -> {
        if (result.statusCode()
                .equals(HttpStatus.OK)) {
            return result.bodyToMono(User.class);
        } else if (result.statusCode()
                .is4xxClientError()) {
            return Mono.error(new ApplicationException(ERROR_MSG))
        }
    });
```