(/)

Spring 5 WebClient

Last modified: September 4, 2019

by baeldung (https://www.baeldung.com/author/baeldung/)

HTTP Client-Side (https://www.baeldung.com/category/http/)

Spring Web (https://www.baeldung.com/category/spring/spring-web/)

Spring 5 (https://www.baeldung.com/tag/spring-5/)

WebClient (https://www.baeldung.com/tag/webclient/)

1. Overview

In this article, we're going to show the *WebClient* – a reactive web client that's being introduced in Spring 5.

We're going to have a look at the *WebTestClient* as well – which is a *WebClient* designed to be used in tests.

Further reading:

Spring WebClient Filters (https://www.baeldung.com/spring-webclient-filters)

Learn about WebClient filters in Spring WebFlux

Read more (https://www.baeldung.com/spring-webclient-filters) →

Spring WebClient Requests with Parameters (https://www.baeldung.com/webflux-webclient-parameters)

Learn how to reactively consume REST API endpoints with WebClient from Spring Webflux.

Read more (https://www.baeldung.com/webflux-webclient-parameters) →

2. What Is the *WebClient*?

Simply put, *WebClient* is an interface representing the main entry point for performing web requests.

It has been created as a part of the Spring Web Reactive module and will be replacing the classic *RestTemplate* in these scenarios. The new client is a reactive, non-blocking solution that works over the HTTP/1.1 protocol.

Finally, the interface has a single implementation – the *DefaultWebClient* class – which we'll be working with.

3. Dependencies

Since we are using a Spring Boot application, we need the *spring-boot-starter-webflux* (https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-boot-starter-webflux%22%20AND%20g%3A%22org.springframework.boot%22) dependency, as well as the Reactor project. (https://repo1.maven.org/maven2/org/projectreactor/reactor-spring/)

3.1. Building with Maven

Let's add the following dependencies to the *pom.xml* file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.projectreactor</groupId>
    <artifactId>reactor-spring</artifactId>
    <version>1.0.1.RELEASE</version>
</dependency>
```

3.2. Building with Gradle

With Gradle, we need to add the following entries to the *build.gradle* file:

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-webflux'
    compile 'org.projectreactor:reactor-spring:1.0.1.RELEASE'
}
```

4. Working With the *WebClient*

To work properly with the client, we need to know how to:

- create an instance
- make a request
- handle the response

4.1. Creating a *WebClient* Instance

There are three options to choose from. The first one is creating a *WebClient* object with default settings:

```
WebClient client1 = WebClient.create();
```

The second alternative allows initiating a *WebClient* instance with a given base URI:

```
WebClient client2 = WebClient.create("http://localhost:8080 (http://localhost:8080)");
```

The last way (and the most advanced one) is building a client by using the *DefaultWebClientBuilder* class, which allows full customization:

```
WebClient client3 = WebClient
  .builder()
    .baseUrl("http://localhost:8080 (http://localhost:8080)")
    .defaultCookie("cookieKey", "cookieValue")
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .defaultUriVariables(Collections.singletonMap("url", "http://localhost:8080 (http://localhost:808
  .build();
```

4.2. Preparing a Request

First, we need to specify an HTTP method of a request by invoking the *method(HttpMethod method)* or calling its shortcut methods such as *get*, *post*, *delete*:

```
WebClient.UriSpec<WebClient.RequestBodySpec> request1 = client3.method(HttpMethod.POST);
WebClient.UriSpec<WebClient.RequestBodySpec> request2 = client3.post();
```

The next move is to provide a URL. We can pass it to the *uri* API – as a *String* or a *java.net.URL* instance:

```
WebClient.RequestBodySpec uri1 = client3
  .method(HttpMethod.POST)
  .uri("/resource");

WebClient.RequestBodySpec uri2 = client3
  .post()
  .uri(URI.create("/resource"));
```

Moving on, we can set a request body, content type, length, cookies or headers – if we need to.

For example, if we want to set a request body – there are two available ways – filling it with a *BodyInserter* or delegating this work to a *Publisher*:

```
WebClient.RequestHeadersSpec requestSpec1 = WebClient
  .create()
  .method(HttpMethod.POST)
  .uri("/resource")
  .body(BodyInserters.fromPublisher(Mono.just("data")), String.class);

WebClient.RequestHeadersSpec<?> requestSpec2 = WebClient
  .create("http://localhost:8080 (http://localhost:8080)")
  .post()
  .uri(URI.create("/resource"))
  .body(BodyInserters.fromObject("data"));
```

The *BodyInserter* is an interface responsible for populating a *ReactiveHttpOutputMessage* body with a given output message and a context used during the insertion. A *Publisher* is a reactive component that is in charge of providing a potentially unbounded number of sequenced elements.

The second way is the *body* method, which is a shortcut for the original *body(BodyInserter inserter)* method.

To alleviate this process of filling a *BodyInserter*, there is a *BodyInserters* class with a number of useful utility methods:

```
BodyInserter<Publisher<String>, ReactiveHttpOutputMessage> inserter1 = BodyInserters
  .fromPublisher(Subscriber::onComplete, String.class);
```

It is also possible with a *MultiValueMap*:

```
LinkedMultiValueMap map = new LinkedMultiValueMap();

map.add("key1", "value1");
map.add("key2", "value2");

BodyInserter<MultiValueMap, ClientHttpRequest> inserter2
 = BodyInserters.fromMultipartData(map);
```

Or by using a single object:

```
BodyInserter<Object, ReactiveHttpOutputMessage> inserter3
 = BodyInserters.fromObject(new Object());
```

After we set the body, we can set headers, cookies, acceptable media types. **Values will be added to those have been set when instantiating the client.**

Also, there is additional support for the most commonly used headers like *"If-None-Match", "If-Modified-Since", "Accept", "Accept-Charset".*

Here's an example how these values can be used:

```
WebClient.ResponseSpec response1 = uri1
  .body(inserter3)
    .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .accept(MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML)
    .acceptCharset(Charset.forName("UTF-8"))
    .ifNoneMatch("*")
    .ifModifiedSince(ZonedDateTime.now())
  .retrieve();
```

4.3. Getting a Response

The final stage is sending the request and receiving a response. This can be done with either the *exchange* or the *retrieve* methods.

They differ in return types; the *exchange* method provides a *ClientResponse* along with its status, headers while the *retrieve* method is the shortest path to fetching a body directly:

```
String response2 = request1.exchange()
  .block()
  .bodyToMono(String.class)
  .block();
String response3 = request2
  .retrieve()
  .bodyToMono(String.class)
  .block();
```

Pay attention to the *bodyToMono* method, which will throw a *WebClientException* if the status code is *4xx* (client error) or *5xx* (Server error). We used the *block* method on *Mono*s to subscribe and retrieve an actual data which was sent with the response.

5. Working With the *WebTestClient*

The *WebTestClient* is the main entry point for testing WebFlux server endpoints. It has very similar API to the *WebClient*, and it delegates most of the work to an internal *WebClient* instance focusing mainly on providing a test context. The *DefaultWebTestClient* class is a single interface implementation.

The client for testing can be bound to a real server or work with specific controllers or functions. To complete end-to-end integration tests with actual requests to a running server, we can use the *bindToServer* method:

```
WebTestClient testClient = WebTestClient
  .bindToServer()
  .baseUrl("http://localhost:8080 (http://localhost:8080)")
  .build();
```

We can test a particular *RouterFunction* by passing it to the *bindToRouterFunction* method:

```
RouterFunction function = RouterFunctions.route(
  RequestPredicates.GET("/resource"),
  request -> ServerResponse.ok().build()
);

WebTestClient
  .bindToRouterFunction(function)
  .build().get().uri("/resource")
  .exchange()
  .expectStatus().isOk()
  .expectBody().isEmpty();
```

The same behavior can be achieved with the *bindToWebHandler* method which takes a *WebHandler* instance:

```
WebHandler handler = exchange -> Mono.empty();
WebTestClient.bindToWebHandler(handler).build();
```

A more interesting situation occurs when we're using the *bindToApplicationContext* method. It takes an *ApplicationContext*, analyses the context for controller beans and *@EnableWebFlux* configurations.

If we inject an instance of the *ApplicationContext*, a simple code snippet may look like this:

```
@Autowired
private ApplicationContext context;

WebTestClient testClient = WebTestClient.bindToApplicationContext(context)
  .build();
```

A shorter approach would be providing an array of controllers we want to test by the *bindToController* method. Assuming we've got a *Controller* class and we injected it into a needed class, we can write:

```
@Autowired
private Controller controller;

WebTestClient testClient = WebTestClient.bindToController(controller).build();
```

After building a *WebTestClient* object, all following operations in the chain are going to be similar to the *WebClient* up to the *exchange* method (one way to get a response), which provides the *WebTestClient.ResponseSpec* interface to work with useful methods like the *expectStatus*, *expectBody*, *expectHeader*.

```
WebTestClient
  .bindToServer()
    .baseUrl("http://localhost:8080 (http://localhost:8080)")
    .build()
    .post()
    .uri("/resource")
  .exchange()
    .expectStatus().isCreated()
    .expectHeader().valueEquals("Content-Type", "application/json")
    .expectBody().isEmpty();
```

6. Conclusion

In this tutorial, we've considered a new enhanced Spring mechanism for making requests on the client side – the *WebClient* class.

Also, we have looked at the benefits it provides by going all the way through request processing.

All of the code snippets, mentioned in the article, can be found in our GitHub repository (https://github.com/eugenp/tutorials/tree/master/spring-5-reactive).

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT