WebTestClient

Version 5.0.3.RELEASE

`WebTestClient` is a non-blocking, reactive client for testing web servers. It uses the reactive WebClient internally to perform requests and provides a fluent API to verify responses. The `WebTestClient` can connect to any server over an HTTP connection. It can also bind directly to WebFlux applications with mock request and response objects, without the need for an HTTP server.

Kotlin users, please see this section for important information on using the `WebTestClient` in Kotlin.

Chapter 1. Setup

To create a `WebTestClient` you must choose one of several server setup options. Effectively you either configure a WebFlux application to bind to, or use absolute URLs to connect to a running server.

1.1. Bind to controller

Use this server setup to test one `@Controller` at a time:

```
client = WebTestClient.bindToController(new TestController()).build();
```

The above loads the WebFlux Java config and registers the given controller. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects. There are more methods on the builder to customize the default WebFlux Java config.

1.2. Bind to RouterFunction

Use this option to set up a server from a RouterFunction:

```
RouterFunction<?> route = ...
client = WebTestClient.bindToRouterFunction(route).build();
```

Internally the provided configuration is passed to `RouterFunctions.toWebHandler`. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

1.3. Bind to ApplicationContext

Use this option to setup a server from the Spring configuration of your application, or some subset of it:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = WebConfig.class) ①
public class MyTests {

    @Autowired
    private ApplicationContext context; ②

    private WebTestClient client;

    @Before
    public void setUp() {
        client = WebTestClient.bindToApplicationContext(context).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Internally the provided configuration is passed to `WebHttpHandlerBuilder` to set up the request processing chain, see WebHandler API for more details. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

1.4. Bind to server

This server setup option allows you to connect to a running server:

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

1.5. Client builder

In addition to the server setup options above, you can also configure client options including base URL, default headers, client filters, and others. These options are readily available following `bindToServer`. For all others, you need to use `configureClient()` to transition from server to client configuration as shown below:

```
client = WebTestClient.bindToController(new TestController())
        .configureClient()
        .baseUrl("/test")
        .build();
```

Chapter 2. Writing tests

`WebTestClient` is a thin shell around WebClient. It provides an identical API up to the point of performing a request via `exchange()`. What follows after `exchange()` is a chained API workflow to verify responses.

Typically you start by asserting the response status and headers:

```
client.get().uri("/persons/1")
        .accept(MediaType.APPLICATION_JSON_UTF8)
        .exchange()
        .expectStatus().isOk()
        .expectHeader().contentType(MediaType.APPLICATION_JSON_UTF8)
        // ...
```

Then you specify how to decode and consume the response body:

- `expectBody(Class<T>)` — decode to single object.
- `expectBodyList(Class<T>)` — decode and collect objects to `List<T>`.
- `expectBody()` — decode to `byte[]` for JSON content or empty body.

Then you can use built-in assertions for the body. Here is one example:

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBodyList(Person.class).hasSize(3).contains(person);
```

You can go beyond the built-in assertions and create your own:

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .consumeWith(result -> {
            // custom assertions (e.g. AssertJ)...
        });
```

You can also exit the workflow and get a result:

```
EntityExchangeResult<Person> result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();
```

When you need to decode to a target type with generics, look for the overloaded methods that accept {api-spring-framework}/core/ParameterizedTypeReference.html[ParameterizedTypeReference] instead of `Class<T>`.

2.1. No content

If the response has no content, or you don't care if it does, use `Void.class` which ensures that resources are released:

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound()
        .expectBody(Void.class);
```

Or if you want to assert there is no response content, use this:

```
client.post().uri("/persons")
        .body(personMono, Person.class)
        .exchange()
        .expectStatus().isCreated()
        .expectBody().isEmpty();
```

2.2. JSON content

When you use `expectBody()` the response is consumed as a `byte[]`. This is useful for raw content assertions. For example you can use JSONAssert to verify JSON content:

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .json("{\"name\":\"Jane\"}")
```

You can also use JSONPath expressions:

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .jsonPath("$[0].name").isEqualTo("Jane")
        .jsonPath("$[1].name").isEqualTo("Jason");
```

2.3. Streaming responses

To test infinite streams (e.g. `"text/event-stream"`, `"application/stream+json"`), you'll need to exit the chained API, via `returnResult`, immediately after response status and header assertions, as shown below:

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
        .accept(TEXT_EVENT_STREAM)
        .exchange()
        .expectStatus().isOk()
        .returnResult(MyEvent.class);
```

Now you can consume the `Flux<T>`, assert decoded objects as they come, and then cancel at some point when test objects are met. We recommend using the `StepVerifier` from the `reactor-test` module to do that, for example:

```
Flux<Event> eventFux = result.getResponseBody();

StepVerifier.create(eventFlux)
        .expectNext(person)
        .expectNextCount(4)
        .consumeNextWith(p -> ...)
        .thenCancel()
        .verify();
```

2.4. Request body

When it comes to building requests, the `WebTestClient` offers an identical API as the `WebClient` and the implementation is mostly a simple pass-through. Please refer to the WebClient documentation for examples on how to prepare a request with a body including submitting form data, multipart requests, and more.