WebTestClient

Version 5.2.8.BUILD-SNAPSHOT

`WebTestClient` is a thin shell around WebClient, using it to perform requests and exposing a dedicated, fluent API for verifying responses. `WebTestClient` binds to a WebFlux application by using a mock request and response, or it can test any web server over an HTTP connection.



Kotlin users: See this section related to use of the `WebTestClient`.

Chapter 1. Setup

To create a `WebTestClient` you must choose one of several server setup options. Effectively you're either configuring the WebFlux application to bind to or using a URL to connect to a running server.

1.1. Bind to Controller

The following example shows how to create a server setup to test one `@Controller` at a time:

Java

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

Kotlin

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

The preceding example loads the WebFlux Java configuration and registers the given controller. The resulting WebFlux application is tested without an HTTP server by using mock request and response objects. There are more methods on the builder to customize the default WebFlux Java configuration.

1.2. Bind to Router Function

The following example shows how to set up a server from a RouterFunction:

Java

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

Kotlin

```
val route: RouterFunction<*> = ...
val client = WebTestClient.bindToRouterFunction(route).build()
```

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

1.3. Bind to `ApplicationContext`

The following example shows how to set up a server from the Spring configuration of your application or some subset of it:

Java

```
@SpringJUnitConfig(WebConfig.class) ①
class MyTests {

    WebTestClient client;

    @BeforeEach
    void setUp(ApplicationContext context) {  ②
        client = WebTestClient.bindToApplicationContext(context).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Kotlin

```
@SpringJUnitConfig(WebConfig::class) ①
class MyTests {

    lateinit var client: WebTestClient

    @BeforeEach
    fun setUp(context: ApplicationContext) { ②
        client = WebTestClient.bindToApplicationContext(context).build() ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

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

1.4. Bind to Server

The following server setup option lets you connect to a running server:

Java

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

Kotlin

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

1.5. Client Builder

In addition to the server setup options described earlier, 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 follows:

Java

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

Kotlin

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

Chapter 2. Writing Tests

`WebTestClient` provides an API identical to WebClient up to the point of performing a request by using `exchange()`. What follows after `exchange()` is a chained API workflow to verify responses.

Typically, you start by asserting the response status and headers, as follows:

Java

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

Kotlin

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

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 an empty body.

Then you can use built-in assertions for the body. The following example shows one way to do so:

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.expectBodyList

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

You can also go beyond the built-in assertions and create your own, as the following example shows:

Java

```
   import org.springframework.test.web.reactive.server.expectBody

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

Kotlin

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

You can also exit the workflow and get a result, as follows:

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.expectBody

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



When you need to decode to a target type with generics, look for the overloaded methods that accept `ParameterizedTypeReference` instead of `Class<T>`.

2.1. No Content

If the response has no content (or you do not care if it does) use `Void.class`, which ensures that resources are released. The following example shows how to do so:

Java

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

Kotlin

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound
        .expectBody<Unit>()
```

Alternatively, if you want to assert there is no response content, you can use code similar to the following:

Java

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

Kotlin

```
client.post().uri("/persons")
        .bodyValue(person)
        .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, as follows:

Java

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

Kotlin

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

You can also use JSONPath expressions, as follows:

Java

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

Kotlin

```
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 (for example, `"text/event-stream"` or `"application/stream+json"`), you need to exit the chained API (by using `returnResult`), immediately after the response status and header assertions, as the following example shows:

Java

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

Kotlin

```
import org.springframework.test.web.reactive.server.returnResult

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

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

Java

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

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

Kotlin

```
val eventFlux = 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 API identical to the `WebClient`, and the implementation is mostly a simple pass-through. See the WebClient documentation for

examples on how to prepare a request with a body, including submitting form data, multipart requests, and more.