**Handling WebTestClient Blocking Issue with Infinite Flux**

**Overview**

When testing a **Spring WebFlux** endpoint that returns an **infinite Flux** (e.g., server-sent events), developers may encounter an issue where **WebTestClient gets stuck on .exchange()**. This happens because WebTestClient expects a completion signal, which an infinite stream does not provide.

**Problem Statement**

**Issue**

* When asserting a **controller method that returns an infinite Flux**, WebTestClient gets stuck at .exchange().
* Assertions like .expectStatus() are never executed, and the test never completes.

**Expected Behavior**

* StepVerifier should assert a few elements, cancel the subscription, and allow the test to finish successfully.

**Actual Behavior**

* WebTestClient.exchange() **never completes**.
* .expectStatus() is **never executed**.
* The test remains **stuck indefinitely**.

**Root Cause Analysis**

**Why Does This Happen?**

1. **Infinite Flux Never Completes**
   * WebTestClient.exchange() expects a **completion signal** (end of response stream).
   * Since an **infinite Flux does not complete**, exchange() waits indefinitely.
2. **Blocking Nature of exchange()**
   * exchange() **blocks until the response completes**.
   * Since an **infinite Flux has no termination signal**, exchange() hangs forever.
3. **No Automatic Cancellation in WebTestClient**
   * When using StepVerifier, we can **manually cancel** the Flux.
   * But WebTestClient.exchange() **lacks explicit cancellation controls**.

**Solutions & Workarounds**

**Solution 1: Use .responseTimeout() to Prevent Infinite Blocking**

webTestClient

.get()

.uri("/stream-endpoint")

.exchange()

.expectStatus().isOk()

.expectBodyList(String.class)

.hasSize(3)

.returnResult()

.getResponseBody();

✅ **Pros:** Prevents blocking indefinitely.  
❌ **Cons:** Still requires consuming the response manually.

**Solution 2: Limit the Flux in the Controller (.take(n))**

Modify the controller method to **limit the number of elements**:

@GetMapping(value = "/stream", produces = MediaType.TEXT\_EVENT\_STREAM\_VALUE)

public Flux<String> stream() {

return Flux.interval(Duration.ofSeconds(1))

.map(i -> "Event " + i)

.take(3); // Limits the Flux to 3 elements

}

✅ **Pros:** Ensures test completion, avoids blocking. ❌ **Cons:** Not useful if real-world behavior requires infinite streaming.

**Solution 3: Use StepVerifier Instead of exchange()**

If you need to assert only a subset of elements, use StepVerifier:

FluxExchangeResult<String> result = webTestClient.get()

.uri("/stream-endpoint")

.exchange()

.returnResult(String.class);

StepVerifier.create(result.getResponseBody())

.expectNext("Event 0", "Event 1", "Event 2")

.thenCancel()

.verify();

✅ **Pros:** Allows manual cancellation, avoids blocking. ❌ **Cons:** Requires fetching the response as a Flux.

**Ensure You Have reactor-test Dependency**

If StepVerifier is not recognized, add this dependency: **Maven:**

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-test</artifactId>

<scope>test</scope>

</dependency>

**Gradle:**

testImplementation 'io.projectreactor:reactor-test'

**Solution 4: Use .consumeWith() for Response Handling**

webTestClient.get()

.uri("/stream-endpoint")

.exchange()

.expectStatus().isOk()

.expectBodyList(String.class)

.consumeWith(response -> {

List<String> body = response.getResponseBody();

assert body != null;

assertEquals(3, body.size());

});

✅ **Pros:** Provides manual control over response consumption. ❌ **Cons:** Only works if Flux completion is enforced.

**Alternative Approach: Using RestTemplate**

If WebTestClient is not suitable, you can use **RestTemplate**:

**1. Configure RestTemplate in Spring Boot**

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

**2. Fetch Data Using RestTemplate**

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response = restTemplate.getForEntity("http://localhost:8080/stream", String.class);

System.out.println("Response: " + response.getBody());

✅ **Pros:** Simpler than WebTestClient for debugging. ❌ **Cons:** Not reactive, cannot handle infinite streams well.

**3. Using ClientHttpRequestFactory for Streaming Responses**

HttpComponentsClientHttpRequestFactory requestFactory = new HttpComponentsClientHttpRequestFactory();

requestFactory.setConnectionRequestTimeout(5000);

requestFactory.setConnectTimeout(5000);

requestFactory.setReadTimeout(5000);

RestTemplate restTemplate = new RestTemplate(requestFactory);

ResponseEntity<String> response = restTemplate.getForEntity("http://localhost:8080/stream", String.class);

System.out.println(response.getBody());

✅ **Pros:** Supports streaming responses better. ❌ **Cons:** Still not fully reactive.

**Final Recommendations**

|  |  |
| --- | --- |
| **Scenario** | **Recommended Approach** |
| Testing a finite Flux | WebTestClient with .take(n) in controller |
| Testing an infinite Flux | WebTestClient with StepVerifier and .thenCancel() |
| Avoiding blocking | .responseTimeout() or .consumeWith() |
| Need an alternative to WebTestClient | Use RestTemplate for basic debugging |

**Key Takeaways**

1. **WebTestClient blocks on infinite Flux** because .exchange() expects a termination signal.
2. **Best approach:** Use StepVerifier or limit the Flux using .take(n).
3. **Alternative:** Use RestTemplate if WebTestClient is unsuitable.
4. **For infinite streaming tests:** Avoid .exchange(), prefer StepVerifier or .consumeWith().