(/)

Burlándose de un cliente web en primavera

Última modificación: 11 de octubre de 2019

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

Primavera (https://www.baeldung.com/category/spring/) +
Pruebas (https://www.baeldung.com/category/testing/)

1. Información general

En estos días, esperamos llamar a las API REST en la mayoría de nuestros servicios. Spring ofrece algunas opciones para crear un cliente REST, y **se recomienda *WebClient*** .

En este tutorial rápido, veremos cómo **unir los servicios de prueba que usan *WebClient* para llamar a las API** .

2. Burlándose

Tenemos dos opciones principales para burlarse en nuestras pruebas:

- Use **Mockito (https://www.baeldung.com/mockito-series)** para imitar el comportamiento de *WebClient*
- Use *WebClient* de verdad, pero simule el servicio al que llama usando ***MockWebServer* (okhttp) (https://github.com/square/okhttp/tree/master/mockwebserver)**

3. Usando Mockito

Mockito (https://www.baeldung.com/mockito-series) es la biblioteca de burlas más común para Java. Es bueno para proporcionar respuestas predefinidas a las llamadas a métodos, pero las cosas se ponen difíciles cuando se burlan de API fluidas. Esto se debe a que en una API fluida, muchos objetos pasan entre el código de llamada y el simulacro.

Por ejemplo, tengamos una clase *EmployeeService* con un método *getEmployeeById* para obtener datos a través de HTTP usando *WebClient* :

```
public class EmployeeService {

    public Mono<Employee> getEmployeeById(Integer employeeId) {
        return webClient
                .get()
                .uri("http://localhost:8080/employee/ (http://localhost:8080/employee/){id}", emplo
                .retrieve()
                .bodyToMono(Employee.class);
    }
}
```

Podemos usar Mockito para burlarse de esto:

```
@ExtendWith(MockitoExtension.class)
public class EmployeeServiceTest {

    @Test
    void givenEmployeeId_whenGetEmployeeById_thenReturnEmployee() {

        Integer employeeId = 100;
        Employee mockEmployee = new Employee(100, "Adam", "Sandler",
          32, Role.LEAD_ENGINEER);
        when(webClientMock.get())
          .thenReturn(requestHeadersUriSpecMock);
        when(requestHeadersUriMock.uri("/employee/{id}", employeeId))
          .thenReturn(requestHeadersSpecMock);
        when(requestHeadersMock.retrieve())
          .thenReturn(responseSpecMock);
        when(responseMock.bodyToMono(Employee.class))
          .thenReturn(Mono.just(mockEmployee));

        Mono<Employee> employeeMono = employeeService.getEmployeeById(employeeId);

        StepVerifier.create(employeeMono)
          .expectNextMatches(employee -> employee.getRole()
            .equals(Role.LEAD_ENGINEER))
          .verifyComplete();
    }

}
```

Como podemos ver, necesitamos proporcionar un objeto simulado diferente para cada llamada en la cadena, con cuatro diferentes *cuando / luego* se requieren llamadas de *devolución* . **Esto es detallado y engorroso** . También requiere que conozcamos los detalles de implementación de cómo exactamente nuestro servicio utiliza *WebClient,* lo que lo convierte en una forma frágil de prueba.

¿Cómo podemos escribir mejores pruebas para *WebClient?*

4. Usando *MockWebServer*

MockWebServer (https://github.com/square/okhttp/tree/master/mockwebserver) , creado por el equipo de Square, es un pequeño servidor web que puede recibir y responder solicitudes HTTP.

La interacción con *MockWebServer* desde nuestros casos de prueba permite que nuestro código use llamadas HTTP reales a un punto final local . Obtenemos el beneficio de probar las interacciones HTTP previstas y ninguno de los desafíos de burlarse de un cliente fluido complejo.

Usando *MockWebServer* se recomienda por el Equipo de primavera (https://github.com/spring-projects/spring-framework/issues/19852#issuecomment-453452354) para escribir pruebas de integración .

4.1. Dependencias de *MockWebServer*

Para usar *MockWebServer* , necesitamos agregar dependencias de Maven para okhttp (https://search.maven.org/search?q=g:com.squareup.okhttp3%20AND%20a:okhttp&core=gav) y mockwebserver (https://search.maven.org/search?q=g:com.squareup.okhttp3%20AND%20a:mockwebserver&core=gav) a nuestro pom.xml:

```
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
    <version>4.0.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>mockwebserver</artifactId>
    <version>4.0.1</version>
    <scope>test</scope>
</dependency>
```

4.2. Agregar *MockWebServer* a nuestra prueba

Probemos nuestro *EmployeeService* con *MockWebServer* :

```
public class EmployeeServiceMockWebServerTest {

    public static MockWebServer mockBackEnd;

    @BeforeAll
    static void setUp() throws IOException {
        mockBackEnd = new MockWebServer();
        mockBackEnd.start();
    }

    @AfterAll
    static void tearDown() throws IOException {
        mockBackEnd.shutdown();
    }
}
```

En la clase de prueba JUnit anterior, el método *setUp* y *tearDown* se encarga de crear y cerrar *MockWebServer*.

El siguiente paso es **asignar el puerto de la llamada de servicio de un descanso efectivo a la *de MockWebServer* puerto**

```
@BeforeEach
void initialize() {
    String baseUrl = String.format("http://localhost:%s (http://localhost:%s)",
      mockBackEnd.getPort());
    employeeService = new EmployeeService(baseUrl);
}
```

Ahora es el momento de crear un código auxiliar para que *MockWebServer* pueda responder a una *HttpRequest* .

4.3. Tropezando una respuesta

Vamos a usar *de MockWebServer* útil *enqueue* método de hacer cola una respuesta de prueba en el servidor web:

```
@Test
void getEmployeeById() throws Exception {
    Employee mockEmployee = new Employee(100, "Adam", "Sandler",
      32, Role.LEAD_ENGINEER);
    mockBackEnd.enqueue(new MockResponse()
      .setBody(objectMapper.writeValueAsString(mockEmployee))
      .addHeader("Content-Type", "application/json"));

    Mono<Employee> employeeMono = employeeService.getEmployeeById(100);

    StepVerifier.create(employeeMono)
      .expectNextMatches(employee -> employee.getRole()
        .equals(Role.LEAD_ENGINEER))
      .verifyComplete();
}
```

Cuando la llamada API real se realiza desde el *método getEmployeeById (Integer employeeId)* en nuestra clase *EmployeeService* , ***MockWebServer* responderá con el código auxiliar en cola** .

4.4. Comprobación de una solicitud

También es posible que deseemos asegurarnos de que *MockWebServer* recibió la *HttpRequest* correcta .

MockWebServer tiene un método útil llamado *takeRequest* que devuelve una instancia de *RecordedRequest* :

```
RecordedRequest recordedRequest = mockBackEnd.takeRequest();

assertEquals("GET", recordedRequest.getMethod());
assertEquals("/employee/100", recordedRequest.getPath());
```

Con *RecordedRequest* , podemos verificar la *HttpRequest* que se recibió para asegurarnos de que nuestro *WebClient lo* envió correctamente .

5. Conclusión

En este tutorial, hemos tratado de las dos principales opciones disponibles para **simulacro *de WebClient* código de cliente basada en REST** .

Si bien Mockito funcionó y quizás sea una buena opción para ejemplos simples, el enfoque recomendado es usar *MockWebServer* .

Como siempre, el código fuente de este artículo está disponible en GitHub. (https://github.com/eugenp/tutorials/tree/master/spring-5-reactive-client)

Acabo de anunciar el nuevo curso *Learn Spring* , centrado en los fundamentos de Spring 5 y Spring Boot 2:

>> VER EL CURSO (/ls-course-end)

¿Estás aprendiendo a construir tu API **con Spring** ?

Enter your email address

>> Obtenga el libro electrónico

Deja una respuesta

Start the discussion...

Suscribir

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)