(/)

Spring WebClient vs. RestTemplate

Última modificación: 9 de agosto de 2019

por Drazen Nikolic (https://www.baeldung.com/author/drazen-nikolic/)
(https://www.baeldung.com/author/drazen-nikolic/)

HTTP del lado del cliente (https://www.baeldung.com/category/http/)
DESCANSO (https://www.baeldung.com/category/rest/)
Primavera (https://www.baeldung.com/category/spring/) +
RestTemplate (https://www.baeldung.com/tag/resttemplate/)
Cliente web (https://www.baeldung.com/tag/webclient/)

1. Introducción

En este tutorial, vamos a comparar dos de las implementaciones de clientes web de Spring: *RestTemplate (https://www.baeldung.com/rest-template)* y el nuevo *WebClient (https://www.baeldung.com/spring-5-webclient)* reactivo alternativo de Spring 5 .

2. Cliente bloqueante versus no bloqueante

Es un requisito común en las aplicaciones web hacer llamadas HTTP a otros servicios. Por lo tanto, necesitamos una herramienta de cliente web.

2.1. *Cliente de* bloqueo *RestTemplate*

Durante mucho tiempo, Spring ha estado ofreciendo *RestTemplate* como una abstracción de cliente web. Bajo el capó, ***RestTemplate* utiliza la API de Java Servlet, que se basa en el modelo de subprocesos por solicitud** .

Esto significa que el hilo se bloqueará hasta que el cliente web reciba la respuesta. El problema con el código de bloqueo se debe a que cada subproceso consume cierta cantidad de memoria y ciclos de CPU.

Consideremos tener muchas solicitudes entrantes, que esperan un servicio lento necesario para producir el resultado.

Tarde o temprano, las solicitudes que esperan los resultados se acumularán. **En consecuencia, la aplicación creará muchos hilos, que agotarán el grupo de hilos u ocuparán toda la memoria disponible** . También podemos experimentar una degradación del rendimiento debido al frecuente cambio de contexto (subproceso) de la CPU.

2.2. Cliente sin bloqueo de *WebClient*

Por otro lado, ***WebClient* utiliza una solución asincrónica y sin bloqueo proporcionada por el marco Spring Reactive** .

Mientras *RestTemplate* crea un nuevo *hilo* para cada evento (llamada HTTP), *WebClient* creará algo así como una "tarea" para cada evento. Detrás de escena, el marco Reactivo pondrá en cola esas "tareas" y las ejecutará solo cuando esté disponible la respuesta adecuada.

El marco reactivo utiliza una arquitectura basada en eventos. Proporciona medios para componer una lógica asincrónica a través de la API Reactive Streams (https://www.baeldung.com/java-9-reactive-streams) . Como resultado, el enfoque reactivo puede procesar más lógica mientras usa menos hilos y recursos del sistema, en comparación con el método síncrono / de bloqueo.

WebClient es parte de la biblioteca Spring WebFlux (https://www.baeldung.com/spring-webflux) . Por **lo** tanto, también **podemos escribir código de cliente utilizando una API funcional y fluida con tipos reactivos (*Mono* y *Flux*) como composición declarativa** .

3. Ejemplo de comparación

Para demostrar las diferencias entre estos dos enfoques, necesitaríamos ejecutar pruebas de rendimiento con muchas solicitudes concurrentes de clientes. Veríamos una degradación significativa del rendimiento con el método de bloqueo después de un cierto número de solicitudes de clientes paralelos.

Por otro lado, el método reactivo / sin bloqueo debería proporcionar rendimientos constantes, independientemente del número de solicitudes.

A los efectos de este artículo, **implementemos dos puntos finales REST, uno con *RestTemplate* y el otro con *WebClient*** . Su tarea es llamar a otro servicio web REST lento, que devuelve una lista de tweets.

Para empezar, necesitaremos la dependencia de arranque Spring Boot WebFlux (https://search.maven.org/search?q=a:spring-boot-starter-webflux) :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Además, aquí está nuestro punto final REST de servicio lento:

```
@GetMapping("/slow-service-tweets")
private List<Tweet> getAllTweets() {
    Thread.sleep(2000L); // delay
    return Arrays.asList(
      new Tweet("RestTemplate rules", "@user1"),
      new Tweet("WebClient is better", "@user2"),
      new Tweet("OK, both are useful", "@user1"));
}
```

3.1. Usando *RestTemplate* para llamar a un servicio lento

Implementemos ahora otro punto final REST que llamará a nuestro servicio lento a través del cliente web.

En primer lugar, usaremos *RestTemplate* :

```
@GetMapping("/tweets-blocking")
public List<Tweet> getTweetsBlocking() {
    log.info("Starting BLOCKING Controller!");
    final String uri = getSlowServiceUri();

    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<Tweet>> response = restTemplate.exchange(
      uri, HttpMethod.GET, null,
      new ParameterizedTypeReference<List<Tweet>>(){});

    List<Tweet> result = response.getBody();
    result.forEach(tweet -> log.info(tweet.toString()));
    log.info("Exiting BLOCKING Controller!");
    return result;
}
```

Cuando llamamos a este punto final, debido a la naturaleza síncrona de *RestTemplate* , el código bloqueará la espera de la respuesta de nuestro servicio lento. Solo cuando se haya recibido la respuesta, se ejecutará el resto del código de este método. En los registros, veremos:


```
Starting BLOCKING Controller!
Tweet(text=RestTemplate rules, username=@user1)
Tweet(text=WebClient is better, username=@user2)
Tweet(text=OK, both are useful, username=@user1)
Exiting BLOCKING Controller!
```

3.2. Usar *WebClient* para llamar a un servicio lento

En segundo lugar, usemos *WebClient* para llamar al servicio lento:

```
@GetMapping(value = "/tweets-non-blocking",
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Tweet> getTweetsNonBlocking() {
    log.info("Starting NON-BLOCKING Controller!");
    Flux<Tweet> tweetFlux = WebClient.create()
      .get()
      .uri(getSlowServiceUri())
      .retrieve()
      .bodyToFlux(Tweet.class);

    tweetFlux.subscribe(tweet -> log.info(tweet.toString()));
    log.info("Exiting NON-BLOCKING Controller!");
    return tweetFlux;
}
```

En este caso, *WebClient* devuelve un editor *Flux* y se completa la ejecución del método. Una vez que el resultado esté disponible, el editor comenzará a emitir tweets a sus suscriptores. Tenga en cuenta que un cliente (en este caso, un navegador web) que llame a este punto final */ tweets-non-* block también se suscribirá al objeto *Flux* devuelto .

Observemos el registro esta vez:

```
Starting NON-BLOCKING Controller!
Exiting NON-BLOCKING Controller!
Tweet(text=RestTemplate rules, username=@user1)
Tweet(text=WebClient is better, username=@user2)
Tweet(text=OK, both are useful, username=@user1)
```

Tenga en cuenta que este método de punto final se completó antes de recibir la respuesta.

4. Conclusión

En este artículo, exploramos dos formas diferentes de usar clientes web en Spring.

RestTemplate utiliza la API de Java Servlet y, por lo tanto, es síncrona y bloqueante. Por el contrario, *WebClient* es asíncrono y no bloqueará el hilo de ejecución mientras espera que vuelva la respuesta. **Solo cuando la respuesta esté lista se producirá la notificación.**

RestTemplate se seguirá utilizando. En algunos casos, el enfoque sin bloqueo utiliza muchos menos recursos del sistema en comparación con el bloqueo. Por lo tanto, en esos casos, *WebClient* es una opción preferible.

Todos los fragmentos de código, mencionados en el artículo, se pueden encontrar en GitHub (https://github.com/eugenp/tutorials/tree/master/spring-5-reactive-2) .

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)