Spring Cloud Feign: declarative REST client

Por **Jose Manuel Sánchez Suárez** - 26 septiembre, 2017

0. Índice de contenidos.

- 1. Introducción.
- 2. Entorno.
- 3. Ejemplo de consumo de servicios.
- 4. Propagación del contexto de seguridad.
- 5. Intercepción de mensajes/errores.
- 6. Referencias.
- 7. Conclusiones.

1. Introducción.

Feign es una librería que forma parte del stack de Spring Cloud, desarrollada por Netflix, para generar clientes
de servicios REST de forma declarativa.

Al estilo de los repositorios de Spring Data,
lo único que debemos hacer es anotar una interfaz con las operaciones
de mapeo de los servicios que queremos consumir, parametrizando apropiadamente la entrada y salida de los mismos,
para que se correspondan con los verbos y los datos de las operaciones de los servicios que queremos consumir.

Desde el punto de vista del soporte que tenemos a día de hoy con Spring,
Feign nos facilitaría el trabajo así como lo hace Spring Data, sin necesidad de «bajar» al nivel de RestTemplate,
como Spring Data nos evita trabajar directamente con EntityManager o JdbcTemplate. Y, siguiendo con la comparación,
igualmente la implementación se genera al vuelo en tiempo de arranque del contexto de Spring.

De entre sus características podemos encontrar las siguientes:

- Es altamente configurable, pudiendo usarse diversos encoders y decoders para formatear la información que viaja en cada petición y respuesta.
- Soporta las anotaciones de JAX-RS y Spring MVC para la declaración de los endPoints de los servicios REST.
- Se integra perfectamente con el resto de componentes del stack de Spring Cloud:
  - balanceo de carga con Ribbon,
  - circuit breaker con Hystrix, permitiendo definir fallbacks a nivel de cliente,
  - registro de servicios en Eureka,

En este tutorial veremos un ejemplo de uso de la librería, examinando las posibilidades de customización para
afrontar cuestiones transversales como son la propagación del contexto de seguridad
o la gestión de mensajes/errores, en la invocación entre servicios.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.5 GHz Intel Core i7, 16GB DDR3).
- Sistema Operativo: Mac OS Sierra 10.12.5
- Oracle Java: 1.8.0_25
- Spring Cloud Dalston SR3

3. Ejemplo de consumo de servicios.

Vamos a hacer una prueba muy sencilla consumiendo un servicio fake externo
expuesto en https://jsonplaceholder.typicode.com,
como podría ser el servicio de posts
que devuelve este tipo de información:

```Shell
[{
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
    "body": "quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit
}, {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae sequi sint nihil reprehenderit dolor beatae ea dolores
}
...
]
```

Con este json de respuesta podríamos generar automáticamente un pojo para su mapeo en la web http://www.jsonschema2pojo.org/,
aunque por su sencillez y haciendo uso de lombok,
bastaría con crear una clase declarando las siguientes propiedades:

```Shell
package com.sanchezjm.tuto.feign.dto;

import lombok.Data;

@Data
public class Post {


    private Integer userId;

    private Integer id;

    private String title;

    private String body;

}
```

Una vez hecho esto, tendríamos que crear la interfaz para consumir el servicio, que podría tener un código como el siguiente:

```Shell
package com.sanchezjm.tuto.feign.clients;

import java.util.List;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.sanchezjm.tuto.feign.feign.dto.Post;

@FeignClient(name="posts", url="https://jsonplaceholder.typicode.com")
public interface PostClient {

    @RequestMapping(method = RequestMethod.GET, value = "/posts")
    List<Post> getAll();

}
```

Por supuesto que para externalizar la url del host podríamos hacer uso de propiedades y definirla con Expression Language de Spring.

```Shell
@FeignClient(name="posts", url="${externalServer.url}")
```

Solo quedaría marcar la configuración para habilitar la generación de los clientes de feign (como haríamos con los repositorios de Spring Data)
con la siguiente anotación que escaneará a partir del paquete en el que la ubiquemos de forma recursiva en busca de interfaces de clientes para generar los stubs.

```Shell
@EnableFeignClients
```

Por último, sin que sirva de predecente, un test de integración para comprobar que podemos recuperar información del servicio:

```Shell
package com.sanchezjm.tuto.feign;

import java.util.List;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.sanchezjm.tuto.feign.feign.clients.PostClient;
import com.sanchezjm.tuto.feign.feign.dto.Post;


@RunWith(SpringRunner.class)
@SpringBootTest
public class FeignClientsApplicationTests {

    @Autowired
    private PostClient postClient;

    @Test
    public void shouldLoadAllPosts() {
        final List<Post> posts = postClient.getAll();
        Assert.assertNotNull(posts);
        Assert.assertFalse(posts.isEmpty());
    }

}
```

Y en verde!, aunque lo realmente interesante es la integración de los clientes feign con el resto del ecosistema de Spring Cloud
y la posibilidad de declarar nuestro cliente como consumidor de otro servicio dentro de la nube, para ello, solo tendríamos que
indicar a nivel de cliente el identificador (spring.application.name) en términos de Spring Cloud del microservicio que tiene
el endPoint que queremos consumir. En tiempo de despliegue el cliente de feign preguntará al servicio de registro cómo se ha
registrado el servicio que queremos consumir y como hacer uso del mismo a través del gateway, de modo tal que para nosotros
es totalmente transparente y no tenemos por qué conocer la ubicación física del resto de servicios que queremos consumir en la
nube. Como digo, lo único que tenemos que hacer, en la declaración el cliente, es indicar el nombre del microservicio que queremos consumir:

```Shell
@FeignClient("identity-service")
```

4. Propagación del contexto de seguridad.

Si estamos pensando en consumir un servicio dentro de nuestra propia nube tendremos que habilitar de alguna manera,
la propagación del contexto de seguridad para que el microservicio al que invocamos disponga del mismo contexto de
autenticación y autorización del usuario conectado.

Suponiendo que ya existe un filtro de Spring Security a nivel de servicio que recupera de una cabecera el usuario
autenticado no tendríamos más que crear un interceptor de feign para propagar dicha cabecera.

```Shell
package com.sanchezjm.tuto.feign.interceptor;

import org.springframework.security.core.context.SecurityContextHolder;

import feign.RequestInterceptor;
import feign.RequestTemplate;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class SecurityFeignRequestInterceptor implements RequestInterceptor {

    private static final String AUTHENTICATION_HEADER = "my-security-header";

    @Override
    public void apply(RequestTemplate template) {
        propagateAuthorizationHeader(template);
    }

    private void propagateAuthorizationHeader(RequestTemplate template) {
        if (template.headers().containsKey(AUTHENTICATION_HEADER)) {
            log.trace("the authorization {} token has been already set", AUTHENTICATION_HEAD
        } else {
            log.trace("setting the authorization token {}", AUTHENTICATION_HEADER);
            template.header(AUTHENTICATION_HEADER, SecurityContextHolder.getContext().getAut
```

```
        }
    }

}
```

También estaríamos presuponiendo que el contexto de seguridad se asigna a nivel de servicio, no en una capa superior,
como podría ser el gateway. En tal caso, si trabajásemos con un token enriquecido en una capa superior bastaría con
propagar dicho token.

Haciendo uso de hystrix, para que la propagación sea efectiva, debemos configurarlo para que propague el contexto
de seguridad añadiendo la siguiente propiedad:

```shell
hystrix.shareSecurityContext=true
```

Además de la habilitación de hystrix para feign:

```shell
feign.hystrix.enabled=true
```

Al hacer uso de hystrix se lanza un hilo en segundo plano para controlar el timeout y poder lanzar un fallback, sino
lo especificamos, en ese hilo no se propagará, por defecto, el contexto de seguridad.

Para configurarlo solo tenemos que declarar el bean en una clase anotada con un @Configuration:

```shell
    @Bean
    public RequestInterceptor securityFeignRequestInterceptor() {
        return new SecurityFeignRequestInterceptor();
    }
```

5. Intercepción de mensajes/errores.

Lo normal es que los errores dentro de nuestra nube de servicios tengan una normalización en cuanto a formato
y tipología, aunque si consumimos servicios externos quizás nos tengamos que adaptar a otros formatos de mensaje;
sobrescribendo el comportamiento por defecto del framework que usemos, para por ejemplo, añadir
un identificador único del error o permitir devolver una colección de errores que devuelvan información de
validación de un recurso anotado con el soporte de la JSR-303.

Si damos por hecho que nuestros servicios pueden devolver errores con el siguiente formato, teniendo en cuenta que la tipología del error la delegamos en el estado http:

```shell
```

```
{
    "id": "FINDME_WITH_THIS",
    "items": [{
            "code": "ERROR_NO1",
            "description": "Desc NO1"
        },
        {
            "code": "ERROR_NO2",
            "description": "Desc NO2"
        }
    ]
}
```

Si estamos pensando en disponer de una capa de clientes que consuman servicios que pueden devolver
ese tipo de formato de salida, deberíamos pensar también en preparar un componente que parsee esa información de manera transversal.

Para cubrir este requisito basta con implementar un ErrorDecoder como el siguiente, asumiendo que
el formato de mensajes podemos mapearlo contra el objeto MessageResource:

Shell

```
package com.sanchezjm.tuto.javassist;

import com.fasterxml.jackson.databind.ObjectMapper;
import feign.Response;
import feign.codec.ErrorDecoder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.web.client.RestClientException;

import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

@Slf4j
public class CustomFeignErrorDecoder implements ErrorDecoder {

    private ErrorDecoder delegate = new ErrorDecoder.Default();

    private ObjectMapper mapper = new ObjectMapper();

    @Override
    public Exception decode(String methodKey, Response response) {

        log.trace("An exception has been caught in {}, trying to parse the playload.", metho

        if (response.body() == null) {
            log.error("Failed to parse the playload: Response has no body.");
            return delegate.decode(methodKey, response);
        }

        MessageResource messageResource;
        try {
            messageResource = mapper.readValue(response.body().asInputStream(), MessageResou
        } catch (IOException e) {
            log.trace("Failed to parse the playload. The format of the message does not corr
            return delegate.decode(methodKey, response);
        }

        final HttpStatus status = HttpStatus.valueOf(response.status());

        final String firstMessage =
            messageResource.getMessages().isEmpty() ? status.getReasonPhrase() : messageResc

        log.trace("Throwing proper exception with this message \"{}\" ", firstMessage);

        if (status == HttpStatus.FORBIDDEN || status == HttpStatus.UNAUTHORIZED) {
```

```
48              return new AccessDeniedException(firstMessage);
49          }
50          else if (status.is4xxClientError()) {
51              return new BusinessException(status.getReasonPhrase(), messageResource);
52
53          }
54          else {
55              return new RestClientException(firstMessage);
56          }
57
58      }
59  }
```

Se podría decir que este decoder tiene la lógica inversa al ErrorHandler que ha generado la respuesta de error.

Para que funcione solo tenemos que configurarlo en una clase anotada con un @Configuration:

```Shell
1      @Bean
2      public CustomFeignErrorDecoder customErrorDecoder(){
3          return new CustomFeignErrorDecoder();
4      }
```

Aunque lancemos una BusinessException propia, si tenemos configurado hystrix, las excepciones se encapsularán
dentro de una HystrixRuntimeException que podríamos tratar en un errorHandler. Si no queremos que se encapsule
podemos marcar nuestra excepción para que implemente ExceptionNotWrappedByHystrix. De una manera u otra, con este
decoder podremos tratar la excepción, si para el cliente no se ha configurado un fallback de hystrix.

6. Referencias.

- http://projects.spring.io/spring-cloud/
- http://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-feign
- https://github.com/sanchezjm/tuto-feign-clients

7. Conclusiones.

En el próximo hablaremos de su integración con hystrix, ribbon y, si estáis muy interesados, también con Sleuth.

Un saludo.

Jose

Jose Manuel Sánchez Suárez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.

Somos expertos en Java/Java EE