DIGITALHOUSE

Implementación de Circuit Breaker

Certified Tech Developer
The Ultimate Degree

Implementación de Circuit Breaker

Para comenzar debemos **agregar la dependencia en el POM**:

```
<dependency>
            <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

Configuración de Resilience4j

Podemos configurar Resilience4j desde el **application.properties**. A modo de ejemplo, utilizaremos la siguiente configuración:

slidingWindowType

- En este parámetro vamos a indicar si queremos que el Circuit Breaker se active mediante un contador de **eventos** (por ejemplo, tres errores consecutivos) o si queremos que se active mediante un contador de **tiempo**. Para el ejemplo utilizaremos la configuración basada en eventos, seteando este parámetro como **COUNT_BASED**.

slidingWindowSize

- El número de llamadas que tienen que fallar para que se active el Circuit Breaker, pasando al estado **open**. Seteamos este parámetro en 5.

failureRateThreshold

- El porcentaje de llamadas fallidas que causará que se active el Circuit Breaker, pasando al estado **open**. Seteamos este parámetro en 50%. Esta configuración, en conjunto con **slidingWindowSize** en 5 (como indicamos en el punto anterior), significa que si tres o más de las últimas 5 llamadas fallan, el circuito se activa y pasamos al estado **open**.

automaticTransitionFromOpenToHalfOpenEnabled

- Determina que el Circuit Breaker pasará automáticamente al estado **half-open** una vez que el tiempo de espera se haya cumplido. Lo seteamos en **true**. El Circuit Breaker esperará a la primera llamada luego de cumplir el tiempo de espera para pasar al estado half-open.

waitDurationInOpenState

- Especificamos cuánto tiempo el Circuit Breaker deberá esperar en el estado **open** para pasar al estado **half-open**. Seteamos este parámetro en **10000 ms**. Esta configuración en conjunto con la transición automática activada (como indicamos en el punto anterior) significa que luego de 10 segundos en el estado open, automáticamente pasaremos al estado **half-open**.

permittedNumberOfCallsInHalfOpenState

- El número de llamadas que permitiremos en el estado **half-open** serán utilizadas para analizar si pasamos al estado **closed** o volvemos a **open**. Seteamos este parámetro en 3, significa que el patrón esperará a recibir 3 llamadas para determinar a qué estado pasará. Recordemos que configuramos que aceptamos un 50% de llamadas fallidas. Esto quiere decir que si 2 de las 3 llamadas fallan, volveremos al estado **open**; caso contrario, a **closed**.

ignoreExceptions

- Utilizamos este parámetro para indicarle qué excepciones queremos que el Circuit Breaker ignore, es decir, no las cuente como un fallo. Podemos indicarle las excepciones que lanzamos según reglas de negocio (por ejemplo, NotFoundException en el caso que no encontramos un dato buscado), o alguna excepción que lanzamos si falta algún campo requerido.

registerHealtIndicator

- Este parámetro habilita a Resilience4j a agregar información en el endpoint /**health** de Actuator con información sobre el estado del Circuit Breaker. Lo seteamos en **true**.

allowHealtIndicatorToFail

- Este parámetro habilita a Resilience4j a cambiar el estado del endpoint /**health** de “UP” a “DOWN” en caso de que algún servicio tenga algún endpoint en estado **open** o **half-open**. En nuestro caso, seteamos este parámetro en **false**, y programaremos una función que se ejecutará en caso de que pasemos a alguno de los estados mencionados.

management.healt.circuitbreakers.enabled

- Este es un parámetro de Actuator que nos permite habilitar el endpoint **/circuitbreakerevents** para poder consumirlo y conocer los eventos. Lo seteamos en **true**.

Reintentos

El mecanismo de reintentos, o **retry**, es muy utilizado para fallos que no suelen ser reiterados, como fallos temporales en la red. El mecanismo de reintentos simplemente vuelve a enviar una solicitud luego de recibir un error, permitiéndonos configurar cuántas veces queremos reintentar y cuánto tiempo queremos esperar luego de cada reintento.

Para configurar esta lógica lo hacemos en el **application.properties** configurando los siguientes parámetros:

- **maxAttempts:** el número de intentos que queremos realizar antes de contar la solicitud enviada como fallida, incluyendo el primer llamado. Nosotros seteamos en 3, permitiendo un máximo de 2 reintentos después del primer llamado.
- **waitDuration:** el tiempo que esperaremos para realizar un nuevo reintento. Setearemos este atributo en **5000 ms**, esto quiere decir que esperamos 5 segundos entre reintentos.
- **retryExceptions:** una lista de excepciones que desencadenarán un reintento. Nosotros solo realizaremos un reintento en caso de recibir un **InternalServerError**, es decir, al recibir un status code 500.

Agregando las anotaciones necesarias para configurar el Circuit Breaker y los reintentos

Para el ejemplo, partimos de la base que contamos con dos microservicios: **course-service** y **subscription-service**. Desde **course-service** consumimos la API de **subscription-service** buscando una suscripción por ID de usuario.

En **subscription-service** tenemos un método que tiene como parámetro un atributo llamado **throwError** de tipo boolean. En caso de que throwError sea igual a **true**, nos retorna un error. Hacemos esto para forzar el error y probar el Circuit Breaker.

El código nos queda de la siguiente manera:

Subscription controller

```
@GetMapping("/find")
public Subscription findSubscriptionByUser(@RequestParam Integer userId, @RequestParam(defaultValue =
"false") Boolean throwError, HttpServletResponse response){
   return subscriptionService.findSubscriptionByUserId(userId,throwError);
}
```

Subscription service

```
@Override
public Subscription findSubscriptionByUserId(Integer userId, Boolean throwError ) throws RuntimeException{
   if(throwError)
       throw new RuntimeException();

   return subscriptionRepository.findByUserId(userId);
}
```

En **course-service** es donde vamos a configurar el Resilience4j, ya que es el servicio que realiza la llamada a **subscription-service**. A modo de ejemplo, enviamos siempre el valor true en el parámetro **throwError**. En nuestro caso, nos queda así:

02 03

01 04

```
@Override
@CircuitBreaker(name="subscription",fallbackMethod ="getSubscriptionFallbackValue")
@Retry(name = "subscription")
public Course findById(Integer courseId,Integer userId) throws BusinessError {
   Course course = null;
                              ResponseEntity<SubscriptionDTO>           response           =
feignSubscriptionRepository.findByUserId(userId,true);

   checkSubscription(response);

   course = courseRepository.findById(courseId).orElse(null);
   return course;
}
```

01

El **Circuit Breaker** se activará cuando el método arroja una excepción.

02

El parámetro **name** es utilizado para configurar la lógica del Circuit Breaker en el **aplication.properties**.

03

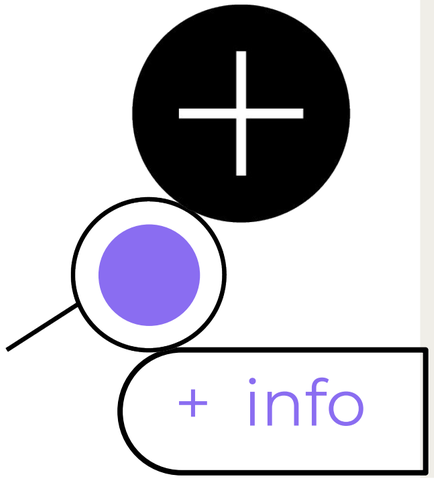
El parámetro **fallbackMethod** es utilizado para indicar el nombre del método alternativo que se ejecutará en caso de que el Circuit Breaker pase a estado **open** y no se envíen solicitudes al servicio de suscripciones.

04

Con la anotación **@Retry** activamos los reintentos en caso de fallas. El parámetro **name** es utilizado con el mismo propósito que en la anotación **@CircuitBreaker**, para configurar la lógica desde el **application.properties**.

Nuestro método alternativo lo único que hace es retornar un mensaje de error y nos queda de la siguiente manera:

```
private void getSubscriptionFallbackValue(CallNotPermittedException ex) throws
CircuitBreakerException {
   throw new CircuitBreakerException("Circuit breaker was activated");
}
```



Aclaración

- **CircuitBreakerException** es una excepción que hereda de Exception y la creamos nosotros.
- El parámetro **CallNotPermittedException** indica que queremos manejar las excepciones de tipo **CallNotPermittedException**, ya que esta excepción es la que arroja el Circuit Breaker cuando se encuentra en estado open.

Configurando el Circuit Breaker y el mecanismo de reintentos

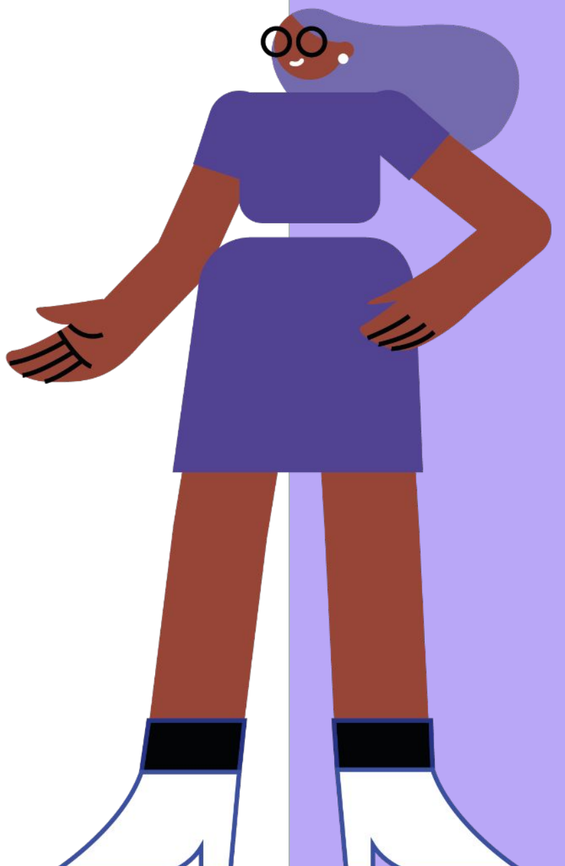
El **application.properties** nos queda de la siguiente manera:

```
#Configuracion de  actuator
management.endpoints.web.exposure.include=circuitbreakers,circuitbreakerevents,health,info
management.health.circuitbreakers.enabled= true
management.endpoint.health.show-details=always


#Configuracion del circuit breaker
resilience4j.circuitbreaker.instances.subscription.allowHealthIndicatorToFail = false
resilience4j.circuitbreaker.instances.subscription.registerHealthIndicator= true
resilience4j.circuitbreaker.instances.subscription.slidingWindowType=COUNT_BASED
resilience4j.circuitbreaker.instances.subscription.slidingWindowSize = 5
resilience4j.circuitbreaker.instances.subscription.failureRateThreshold= 50
resilience4j.circuitbreaker.instances.subscription.waitDurationInOpenState = 15000
resilience4j.circuitbreaker.instances.subscription.permittedNumberOfCallsInHalfOpenState = 3
resilience4j.circuitbreaker.instances.subscription.automaticTransitionFromOpenToHalfOpenEnabled = true

#Configuracion del mecanismo de reintentos.
resilience4j.retry.instances.subscription.maxAttempts = 3
resilience4j.retry.instances.subscription.waitDuration = 1000
resilience4j.retry.instances.subscription.retryExceptions[0]=feign.FeignException$InternalServerError
```

A modo de cierre

Podemos notar que todas las propiedades de Resilience4j comienzan de la misma manera. En donde **subscription** es el nombre de la instancia de Resilience4j que configuramos en el método **findById**.

De esta manera, cuando busquemos una suscripción y el servicio nos retorna un error, se activará el Circuit Breaker. Y, mediante el uso de un método alternativo a modo de ejemplo, retornamos un mensaje de error. En otro escenario, podríamos buscar en caché la suscripción y retornarla, pero la implementación del Circuit Breaker en conjunto con un método alternativo es la misma.



DIGITALHOUSE

¡Muchas gracias!