DIGITALHOUSE

Implementação de Circuit Breaker

Certified Tech Developer
The Ultimate Degree

Implementação de Circuit Breaker

Para começar, precisamos **acrescentar a dependência no POM**:

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

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

Configuração de Resilience4j

Podemos configurar Resilience4j a partir do **application.properties**. Como exemplo, vamos utilizar a seguinte configuração:

slidingWindowType

- Neste parâmetro, vamos indicar se queremos que o Circuit Breaker seja ativado por meio de um contador de **eventos** (por exemplo, três erros consecutivos) ou se queremos que ele seja ativado por meio de um contador de **tempo**. Para o exemplo, usaremos a configuração orientada por eventos, definindo este parâmetro como **COUNT_BASED**.

slidingWindowSize

- O número de chamadas que devem falhar antes que o Circuit Breaker seja ativado e vá para o estado **open**. Definimos este parâmetro como 5.

failureRateThreshold

- A porcentagem de chamadas falhadas que farão com que o Circuit Breaker seja ativado, indo para o estado **open**. Definimos este parâmetro como 50%. Esta configuração, juntamente com o **slidingWindowSize** para 5 (como indicamos no ponto anterior), significa que se três ou mais das últimas 5 chamadas falharem, o circuito será ativado e passaremos para o estado **open**.

automaticTransitionFromOpenToHalfOpenEnabled

- Determina que o Circuit Breaker deve mudar automaticamente para o estado **half-open** uma vez transcorrido o tempo de espera. Definimos como **true**. O Circuit Breaker aguardará a primeira chamada depois de sair para ir para o estado half-open.

waitDurationInOpenState

- Especifica quanto tempo o Circuit Breaker deve esperar no estado **open** antes de mudar para o estado **half-open**. Definimos este parâmetro como **10000 ms**. Esta configuração em conjunto com a transição automática ativada (como indicado no ponto anterior) significa que após 10 segundos no estado open, mudaremos automaticamente para o estado **half-open**.

permittedNumberOfCallsInHalfOpenState

- O número de chamadas que permitiremos no estado **half-open** será usado para analisar se vamos para o estado **closed** ou voltamos para o estado **open**. Se definirmos este parâmetro como 3, isso significa que o padrão aguardará 3 chamadas para determinar para qual estado ele mudará. Lembramos que configuramos que aceitamos 50% das chamadas falhadas. Isto significa que se 2 das 3 chamadas falharem, voltaremos ao estado **open**; caso contrário, ao estado **closed**.

ignoreExceptions

- Usamos este parâmetro para dizer quais exceções queremos que o Circuit Breaker ignore, ou seja, para não considerá-los uma falha. Podemos indicar as exceções que lançamos de acordo com as regras comerciais (por exemplo, NotFoundException no caso de não encontrarmos os dados que estamos procurando), ou uma exceção que lançamos se um campo obrigatório estiver faltando.

registerHealtIndicator

- Este parâmetro permite que o Resilience4j adicione informações no endpoint /**health** do Actuator com informações sobre o estado do Circuit Breaker. Definimos como **true**.

allowHealthIndicatorToFail

- Este parâmetro permite que o Resilience4j altere o estado do endpoint /**health** de “UP” para “DOWN” caso algum serviço tenha um endpoint em estado **open** ou **half-open**. Em nosso caso, definimos este parâmetro como **false**, e programaremos uma função que será executada caso passemos a qualquer um dos estados mencionados.

management.healt.circuitbreakers.enabled

- Este é um parâmetro do Actuator que nos permite habilitar o endpoint /**circuitbreakerevents** para consumi-lo e aprender sobre os eventos.
- Definimos como **true**.

Tentativas

O mecanismo de nova tentativa, ou **retry**, é amplamente utilizado para falhas que normalmente não se repetem, tais como falhas temporárias na rede. O mecanismo de nova tentativa simplesmente reenvia um pedido após receber um erro, permitindo-nos configurar quantas vezes queremos tentar novamente e quanto tempo queremos esperar após cada nova tentativa.

Para configurar esta lógica, fazemos no **application.properties,** configurando os seguintes parâmetros:

- **maxAttempts:** o número de tentativas que queremos fazer antes de contar a solicitação enviada como falha, incluindo a primeira chamada. Definimos como 3, permitindo no máximo 2 tentativas após a primeira chamada.
- **waitDuration:** o tempo que vamos esperar para fazer uma nova tentativa. Definiremos este atributo para **5000 ms**, o que significa que esperamos 5 segundos entre novas tentativas.
- **retryExceptions:** uma lista de exceções que desencadearão uma nova tentativa. Só tentaremos novamente no caso de receber um **InternalServerError**, ou seja, quando recebermos um status code 500.

Acrescentando as anotações necessárias a configuração do Circuit Breaker e novas tentativas

Para o exemplo, assumimos que temos dois microsserviços: **course-service** e **subscription-service**. De **course-service** consumimos a API de **subscription-service** procurando uma assinatura por ID de usuário.

No **subscription-service** temos um método que tem como parâmetro um atributo chamado **throwError** do tipo boolean. No caso de o throwError ser definido como **true**, um erro é devolvido. Fazemos isso para forçar o erro e testar o Circuit Breaker.

O código fica assim:

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);
}
```

Em **course-service** é onde vamos configurar o Resilience4j, pois é o serviço que faz a chamada para o **subscription-service.** Como exemplo, sempre enviamos o valor true no parâmetro **throwError**. Em nosso caso, fica assim:

02 03

01 @Override
@CircuitBreaker(name="subscription",fallbackMethod ="getSubscriptionFallbackValue")
04 @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

O **Circuit Breaker** será acionado quando o método lançar uma exceção.

02

O parâmetro **name** é usado para configurar a lógica do Circuit Breaker.
no **application.properties**.

03

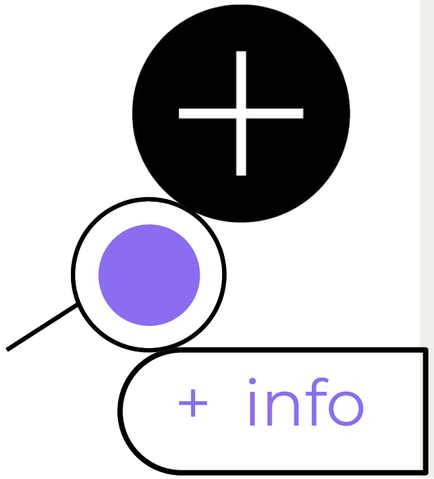
O parâmetro **fallbackMethod** é usado para indicar o nome do método alternativo que será executado caso o Circuit Breaker vá para o estado **open** e nenhuma solicitação seja enviada ao serviço de assinatura.

04

Com a anotação **@Retry**, ativamos as tentativas em caso de falhas. O parâmetro **name** é usado com a mesma finalidade que na anotação **@CircuitBreaker**, para configurar a lógica da **application.properties**.

Nosso método alternativo só retornará uma mensagem de erro e se parece com isto:

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



Esclarecimento

- **CircuitBreakerException** é uma exceção que herda de Exception e é criada por nós.
- O parâmetro **CallNotPermittedException** indica que queremos lidar com exceções do tipo **CallNotPermittedException**, já que esta exceção é a lançada pelo Circuit Breaker quando se encontra em estado open.

Configuração do Circuit Breaker e do mecanismo de nova tentativa

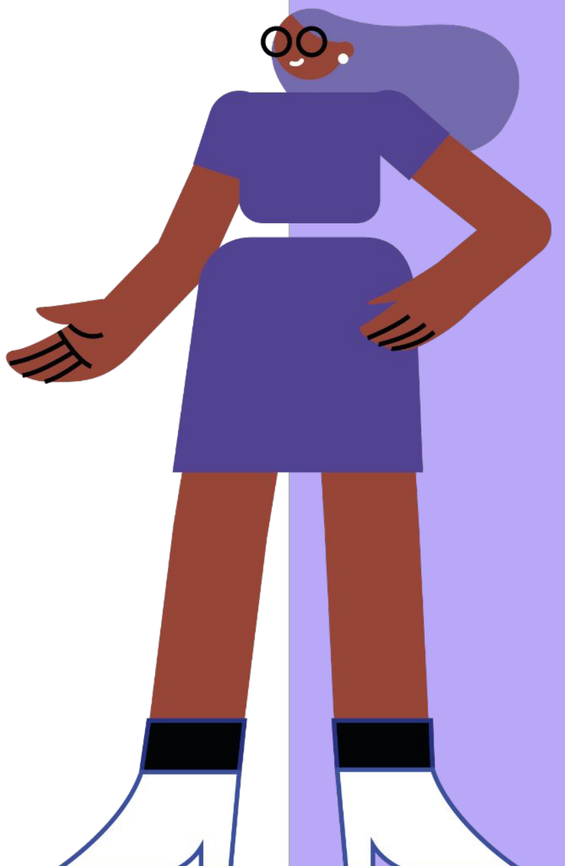
O **application.properties** fica da seguinte maneira:

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


#Configuração do 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

#Configuração do mecanismo de novas tentativas.
resilience4j.retry.instances.subscription.maxAttempts = 3
resilience4j.retry.instances.subscription.waitDuration = 1000
resilience4j.retry.instances.subscription.retryExceptions[0]=feign.FeignException$InternalServerError
```

Em conclusão

Podemos notar que todas as propriedades de Resilience4j começam da mesma maneira. Onde **subscription** é o nome da instância Resilience4j que configuramos no método **findById**.

Desta forma, quando procuramos uma assinatura e o serviço retorna um erro, o Circuit Breaker será ativado. E, usando um método alternativo como exemplo, retornamos uma mensagem de erro. Em outro cenário, poderíamos armazenar em cache a assinatura e devolvê-la, mas a implementação do Circuit Breaker em conjunto com um método alternativo é a mesma.



DIGITALHOUSE

Muito obrigado!