



Tolerância a falha

Arquitetura de micro serviço é naturalmente complexa, **o que pode acontecer com o sistema se UM micro serviço estiver lento ou fora do ar?** Você impacta todo o resto do ecossistema.

Numa arquitetura de micro serviços você tem diversas formas de tratar possíveis erros, **“circuit breaker, retry, rate limiter e etc...”**.

Falando em implementação dessas “teorias” no mundo **java/spring** tem o framework **Resilience4J**, ele se define como um framework **fácil de usar e leve**, onde você implementa o que precisa:



API que faz o consumo

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

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

<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

Esse import traz todas os módulos do resilience4j. Mas dá pra importar os módulos separadamente, para utilizar apenas o que precisa



Retry e Fallback Method - Resilience4J

A primeira opção que vem na cabeça quando ocorre um erro de comunicação entre micro serviços é “Vou tentar de novo”, esse processo é o **Retry**:



API que faz o consumo

```
resilience4j.retry.instances.default.maxRetryAttempts=3
resilience4j.retry.instances.default.waitDuration=1s
resilience4j.retry.instances.default.enableExponentialBackoff=true
resilience4j.retry.instances.outraInstancia.maxRetryAttempts=5
```

Uma série de configurações que um Retry pode oferecer

```
@GetMapping("/currency-exchange/sample-api")
@Retry(name = "default")
public String getTeste(){
    logger.info("Enviei o Request");
    return "Testando legal dog";
}
```

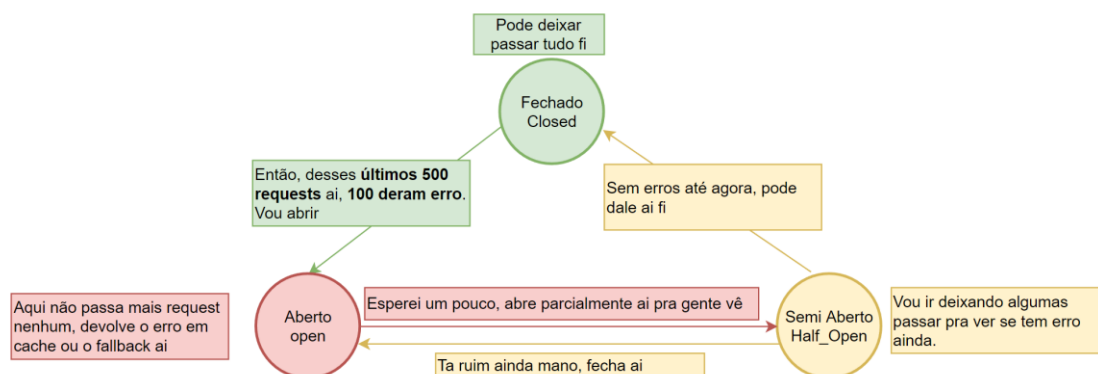
Sem nenhuma onfiguração o padrão de retry é tentar 3 vezes

Além do retry podemos ter outras configurações adicionais, como um **método de fallback**, que basicamente é um método que você vai executar caso de ruim no consumo:



Circuit Breaker – Resilience4J

Circuit Breaker é um **padrão de resiliência**, ele funciona como um “meio” entre uma API que consome e uma API que fornece. O **circuit breaker** tem **3 estados** e ele vai alterando esse estado de acordo com uma métrica de requisição:



A ideia principal é evitar que um **número excessivo de chamadas que podem ter erro chegue na aplicação**, então ele para de permitir chamadas para que isso não se torne uma catástrofe maior (um endpoint com erro sendo chamado milhões de vezes por segundo pode ser problemático).


A implementação mais simples é apenas usar uma anotação, mas é óbvio que existe uma possibilidade imensa de configurações:

```
@GetMapping("/currency-exchange/sample-api")
@CircuitBreaker(name = "default", fallbackMethod = "casoDeRuimFazIsso")
public String getTeste(){
    var resposta : ResponseEntity<String> = new RestTemplate()
        .getForEntity( url: "http:8765/currency-exchange/rato", String.class);
    return "Testando legal dog";
}
```



Rater Limiter – Resilience4J

O Rater Limiter funciona como um limitador de requisições. Então você pode colocar ele na sua requisição e estipular um range de quantas requisições ele deve fazer em X tempo:



API que faz o consumo

```
resilience4j.rateLimiter.instances.default.limitForPeriod=3  
resilience4j.rateLimiter.instances.default.limitRefreshPeriod=10s
```

Aqui eu especifico que o máximo é 3 requests a cada 10 segundos

```
@GetMapping("/currency-exchange/sample-api")  
@RateLimiter(name = "default", fallbackMethod = "casoDeRuimFazIsso")  
public String getTeste(){  
    var resposta : ResponseEntity<String> = new RestTemplate()  
        .getForEntity( url: "http:8765/currency-exchange/rato", String.class);  
    return "Testando legal dog";  
}
```



BulkHead – Resilience4J

Esse é um padrão pra lidar com chamadas concorrentes, não tem detalhes aqui, então é melhor pesquisar depois.