Resilience4j

order-service'in inventory-service çağrılarını manuel try/catch hata yönetiminin ötesine taşıması: Resilience4j ile @CircuitBreaker (CLOSED/OPEN/HALF_OPEN durum makinesi) ve @Retry annotation'larını ResilientStockClient'a uygulamak, fallback metotlarının devre açıldığında ne yaptığı, Spring proxy mekanizmasının self-invocation'da neden çalışmadığı, ve rate limiter/bulkhead'in circuit breaker'dan farklı bir riske karşı koruması.

Orta 24 dk

Son güncelleme: 23 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/resilience4j

---

Resilience4j

"Servisler Arası İletişim" dersi, order-service'e inventory-service'i çağırırken bir 404'ü bağlantı hatasından ayırt etmeyi ve ikincisini bir `InventoryServiceUnavailableException`'a çevirmeyi zaten öğretmişti ("Ağ Güvenilmez: inventory-service Ayakta Değilse Ne Olur?" bölümüne bakınız). Bu dürüst bir hata yönetimi -- ama bir boşluğu var: inventory-service birkaç SANİYELİĞİNE bile düşse, stok verisine ihtiyaç duyan HER order-service isteği hemen ve aynı şekilde başarısız olur, ve order-service zaten zorlandığı belli olan bir servisi durmadan yoklamaya devam eder. Bu ders, her iki sorunu da çözen kütüphaneyi tanıtıyor: Resilience4j.

Resilience4j Nedir?

Resilience4j, riskli bir çağrıyı -- en sık başka bir servise yapılan bir çağrıyı -- bir veya daha fazla KORUYUCU davranışla saran hafif bir Java kütüphanesi: bir circuit breaker (açıkça başarısız olan bir servisi çağırmayı DURDURMAK), bir retry (vazgeçmeden önce TEKRAR denemek), bir rate limiter (bir çağrının ne sıklıkla yapılabileceğini SINIRLAMAK), ve bir bulkhead (aynı anda kaç çağrının UÇUŞTA olabileceğini SINIRLAMAK). Her davranış bir metodun üzerine bir annotation ile uygulanır -- metodun kendi kodu, hataya nasıl dayanacağıyla değil, gerçekte ne yaptığıyla ilgilenmeye devam eder.

Neden Var?

`ResourceAccessException`'ı yakalayıp `InventoryServiceUnavailableException` fırlatmak (StockClientWithDiscovery'nin zaten yaptığı gibi) gerekli ama YETERLİ değil. İki gerçek sorun kalıyor: birincisi, inventory-service düşükse, order-service HER isteği denemeye devam

eder, zaten başarısız olacak bağlantıları beklemek için zaman harcar, ve zaten toparlanmaya çalışan bir servise ek yük bindirir. İkincisi, tek bir kısa ağ aksaklığı, bir kez daha denemek muhtemelen başarılı olacaksa isteği doğrudan başarısız kılmamalı. İkisini de elle iyi yönetmek -- hata sayılarını izlemek, ne zaman çağırmayı bırakacağına karar vermek, doğru aralarla tekrar denemek -- tam olarak ustaca yanlış yapılması kolay olan türden altyapı kodu. Resilience4j bunu kod yerine yapılandırma olarak sağlıyor.

Tarihçe

Resilience4j, 2016'da, Netflix Hystrix'in (Netflix'in kendi circuit breaker kütüphanesi, Zuul ve Eureka ile aynı dönemden -- bkz. "Servis Keşfi ve Eureka" ve "API Gateway" derslerinin "Tarihçe" bölümleri) AÇIK bir yerine geçme amacıyla oluşturuldu. Hystrix, 2018'de Netflix tarafından bakım moduna alındı -- Netflix'in kendi açık kaynak altyapı araçlarının birçoğundan geri çekildiği aynı genel dönem. Resilience4j baştan itibaren daha hafif olacak (Hystrix'in sahip olduğu RxJava bağımlılığı olmadan, Java 8+ fonksiyonel tarzı için tasarlandı) ve modüler olacak şekilde tasarlandı -- bir proje, büyük tek bir kütüphane yerine yalnızca circuit breaker modülüne, yalnızca retry'a, ya da herhangi bir kombinasyona bağımlı olabilir.

order-service'e Resilience4j Eklemek

Resilience4j, Spring Boot'a `resilience4j-spring-boot3` starter'ı ve `application.yml`'de annotation-tabanlı yapılandırmayla entegre olur -- Eureka Server ya da api-gateway'in aksine ayrı bir sunucu ya da altyapı parçasına gerek yok; her koruyucu davranış order-service'in KENDİ İÇİNDE çalışır.

```
# Added on top of order-service's existing application.yml (see the Spring Boot
# Microservice Basics lesson's "Its Own `application.yml`: Port, Application Name,
# and Database" section, and the eureka.client block from the Service Discovery &
# Eureka lesson) -- nothing already there changes, this is purely additive.

resilience4j:
  circuitbreaker:
    instances:
      inventoryService:                          # this name is what @CircuitBreaker
                                                   # in ResilientStockClient refers to
                                                   # -- it does NOT have to match
                                                   # "inventory-service" (the Eureka
                                                   # name), they're independent
        sliding-window-type: COUNT_BASED
        sliding-window-size: 10                   # look at the last 10 calls
        failure-rate-threshold: 50                # if 50% or more of them failed,
                                                    # OPEN the circuit
        wait-duration-in-open-state: 10s          # stay OPEN for 10 seconds before
                                                    # trying a HALF_OPEN test call
        permitted-number-of-calls-in-half-open-state: 3
  retry:
    instances:
      inventoryService:
        max-attempts: 3                           # the ORIGINAL call plus 2 retries
        wait-duration: 500ms                      # pause between attempts
```

> Örnek adı (yukarıdaki `inventoryService`) Resilience4j'nin yapılandırmayı gruplamak için İÇ olarak kullandığı bir isim -- Eureka servis adıyla (`inventory-service`) birebir eşleşmesine hiç gerek yok, ama ilgili bir isim seçmek takip etmeyi kolaylaştırır.

Circuit Breaker: Durumlar ve Yapılandırma

Bir circuit breaker'ın üç durumu vardır. CLOSED normal durumdur -- çağrılar geçer, ve başarısızlıklar sayılır. Başarısızlık oranı yapılandırılmış bir eşiği geçerse, devre OPEN'a düşer -- yapılandırılmış bir bekleme süresi boyunca HER çağrı, gerçek çağrıyı denemeden bile, HEMEN başarısız olur. O bekleme süresinden sonra, devre HALF_OPEN'a geçer, burada az sayıda TEST çağrısına izin verilir -- başarılı olurlarsa devre yeniden kapanır; başarısız olurlarsa yeniden açılır.

StockClient'ı Bir Circuit Breaker ile Sarmak

`@CircuitBreaker` annotation'ı bu durum makinesini TEK bir metoda uygular -- metodun kendi mantığında hiçbir değişiklik gerekmez, yalnızca imzasında ve bir fallback metodunda.

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import io.github.resilience4j.retry.annotation.Retry;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.ResourceAccessException;
```

```
import org.springframework.web.client.RestClient;

// Builds directly on the Service Discovery & Eureka lesson's StockClientWithDiscovery
// (see its own file, and the "Calling a Service by Name with a Load-Balanced RestClient"
// section) -- the constructor and the discovery-aware base URL are UNCHANGED. What's NEW
// is the two annotations on checkStock: @CircuitBreaker and @Retry, both referring to
// the "inventoryService" instance configured in Resilience4jConfig.yml. Neither
// annotation touches the METHOD BODY at all -- Resilience4j wraps the call from the
// OUTSIDE, at the proxy level, exactly like @Transactional does (see the Transaction
// Management lesson).
@Component
class ResilientStockClient {

    private final RestClient restClient;

    ResilientStockClient(RestClient.Builder loadBalancedRestClientBuilder) {
        this.restClient = loadBalancedRestClientBuilder.baseUrl("http://inventory-service").
build();
    }

    // Retry runs FIRST (innermost) -- Resilience4j retries the call up to
    // resilience4j.retry.instances.inventoryService.max-attempts times BEFORE the
    // circuit breaker ever records a single failure from this call. Only once retries
    // are exhausted does the circuit breaker see "this call failed" and count it toward
    // opening the circuit. If the circuit is ALREADY open, neither the method body nor
    // any retry attempt runs at all -- checkStockFallback is called immediately.
    @CircuitBreaker(name = "inventoryService", fallbackMethod = "checkStockFallback")
    @Retry(name = "inventoryService")
    StockCheckResponse checkStock(String productName) {
        try {
            return restClient.get()
                    .uri("/inventory/{productName}", productName)
                    .retrieve()
                    .body(StockCheckResponse.class);
        } catch (HttpClientErrorException.NotFound e) {
            // Same meaning as in every earlier lesson: inventory-service answered, it
            // just doesn't know this product -- this is NOT a failure Resilience4j
            // should retry or count against the circuit breaker.
            return new StockCheckResponse(productName, 0);
        } catch (ResourceAccessException e) {
            throw new InventoryServiceUnavailableException("inventory-service is not
reachable", e);
        }
    }

    // The fallback method's signature MUST match checkStock's (same parameters), plus
    // one extra Throwable parameter at the end -- Resilience4j calls THIS method
    // instead, with the exception that finally triggered it, whenever the circuit is
    // OPEN or every retry attempt has failed. Returning a degraded-but-VALID
    // StockCheckResponse here (instead of letting the exception propagate) is a
    // deliberate choice: order-service can still let the order proceed, treating stock
    // as "unknown, assume none reserved" rather than failing the whole request just
    // because inventory-service is having trouble.
    StockCheckResponse checkStockFallback(String productName, Throwable t) {
        return new StockCheckResponse(productName, 0);
    }
}
```

```
// Unchanged from earlier lessons -- what CAN fail about the call didn't change, only
// what order-service now DOES in response (see checkStockFallback above, instead of
// letting this propagate all the way up).
class InventoryServiceUnavailableException extends RuntimeException {
    InventoryServiceUnavailableException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
// Unchanged from the Service Discovery & Eureka lesson (itself unchanged from Inter-
// Service Communication) -- wrapping the call in a circuit breaker and retry doesn't
// change WHAT inventory-service tells order-service, only what happens when it CAN'T
// be reached at all.
record StockCheckResponse(String productName, int quantityInStock) {
}
```

> `@CircuitBreaker`, tıpkı `@Transactional` gibi (bkz. Transaction Management dersi), yalnızca Spring'in PROXY mekanizması üzerinden çalışır -- `checkStock`'u AYNI sınıftaki BAŞKA bir metottan çağırmak (`this.checkStock(...)`) proxy'yi tamamen atlar, ne circuit breaker ne de retry hiçbir zaman çalışmaz.

Fallback Metotları: Devre Açıldığında Ne Olur?

Bir fallback metodu, devre açık olduğunda ya da her retry denemesi başarısız olduğunda, gerçek metodun YERİNE çalışan metottur -- imzası orijinal metodun parametreleriyle AYNI olmalı, artı sonda bir `Throwable` daha. Yukarıdaki `checkStockFallback`, istisnanın yukarı yayılmasına izin vermek yerine bozuk-ama-geçerli bir `StockCheckResponse` döndürüyor -- bu bilinçli bir tercih, siparişin tamamen başarısız olması yerine "stok rezerve edilmediği varsayılsın" diyerek sipariş işlemenin devam etmesine izin veriyor.

Retry: Vazgeçmeden Önce Tekrar Denemek

Yukarıda `checkStock`'ta da görülen `@Retry`, başarısız bir çağrıyı, aralarında bir bekleme ile, circuit breaker HİÇ bir başarısızlığı kaydetmeden ÖNCE, yapılandırılmış bir sayıda tekrar dener. Bu, gerçekten geçici sorunlar için önemli -- tek bir düşen paket, kısa bir ağ aksaklığı -- burada hemen tekrar denemek muhtemelen başarılı olurdu, ve ilk başarısızlıkta vazgeçmek erken olurdu.

> Retry ve circuit breaker birbiriyle YARIŞAN seçenekler değil -- farklı sorulara cevap veriyorlar. Retry "bu TEK başarısızlık tekrar denemeye değer mi?" diye sorar; circuit breaker "bu servis o kadar TUTARLI başarısız oldu ki artık denemeye bile değmez mi?" diye sorar. İkisini de aynı çağrıya uygulamak (`ResilientStockClient`'ın yaptığı gibi) yaygın, mantıklı bir kombinasyon.

Rate Limiter ve Bulkhead: İki Koruma Daha

Bir circuit breaker ve retry, ikisi de BAŞARISIZLIKLARA tepki verir. Bir rate limiter ve bir bulkhead ise tamamen farklı bir riske karşı korur: order-service'in her şey SAĞLIKLIYKEN bile inventory-service'i (ya da kendisini) aşırı yüklemesi. Bir rate limiter, bir zaman penceresinde kaç çağrıya izin verildiğini sınırlar; bir bulkhead ise aynı anda kaç çağrının UÇUŞTA olabileceğini sınırlar -- ikisi de isimlerini gerçek dünyadaki güvenlik mekanizmalarından ödünç alıyor (elektriksel bir akım sınırlayıcı, bir geminin su almış tek bir bölmenin tüm gemiyi batırmasını önleyen bulkhead bölmeleri).

```
# A SEPARATE Resilience4j config block from Resilience4jConfig.yml -- rate limiter
# and bulkhead protect against two DIFFERENT problems than a circuit breaker does
# (see "Rate Limiter and Bulkhead: Two More Guards"), so they get their own
# instance names here, but the SAME "inventoryService" name could reuse settings
# across all four guard types if a single call needed every one of them at once.

resilience4j:
  ratelimiter:
    instances:
      inventoryService:
        limit-for-period: 20            # at most 20 calls...
        limit-refresh-period: 1s        # ...per 1-second window
        timeout-duration: 0             # don't wait for a free slot -- reject
                                         # immediately if the limit is already hit
  bulkhead:
    instances:
      inventoryService:
        max-concurrent-calls: 5         # at most 5 calls to inventory-service IN
                                         # FLIGHT at once, from THIS service
        max-wait-duration: 0            # reject immediately if all 5 slots are busy,
                                         # don't queue
```

```
import io.github.resilience4j.circuitbreaker.CircuitBreaker;
import io.github.resilience4j.circuitbreaker.CircuitBreakerRegistry;
import jakarta.annotation.PostConstruct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

// The circuit breaker's STATE (see "Circuit Breaker: States and Configuration") isn't
// directly visible anywhere in ResilientStockClient -- @CircuitBreaker manages it behind
// the scenes. This class subscribes to the SAME "inventoryService" instance's state
// transitions, purely to make CLOSED -> OPEN -> HALF_OPEN -> CLOSED visible in the logs
// -- useful while learning, and a real precursor to the metrics/dashboards the
// Observability lesson covers later.
@Component
class CircuitBreakerEventListener {

    private static final Logger log = LoggerFactory.getLogger(CircuitBreakerEventListener.
class);

    private final CircuitBreakerRegistry circuitBreakerRegistry;

    CircuitBreakerEventListener(CircuitBreakerRegistry circuitBreakerRegistry) {
        this.circuitBreakerRegistry = circuitBreakerRegistry;
    }

    @PostConstruct
    void subscribeToStateTransitions() {
        CircuitBreaker inventoryServiceBreaker = circuitBreakerRegistry.circuitBreaker
("inventoryService");
        inventoryServiceBreaker.getEventPublisher()
                .onStateTransition(event ->
                        log.warn("inventoryService circuit breaker: {} -> {}",
                                event.getStateTransition().getFromState(),
                                event.getStateTransition().getToState()));
    }
}
```

Best Practices

- **Gerçekten geçici başarısız olabilecek çağrılarda `@Retry` ve `@CircuitBreaker`'ı BİRLİKTE uygula** -- retry kısa aksaklığı yönetir, circuit breaker sürekli başarısızlığı yönetir, ve her biri diğerinin cevaplamadığı bir soruyu cevaplar.
- **Her zaman ÇAĞIRAN için anlamlı bir fallback sağla**, yalnızca genel bir hata değil -- `checkStockFallback`'in "stok rezerve edilmediği varsayılsın"ı, order-service'in daha büyük akışının tamamen başarısız olmak yerine devam etmesine izin veriyor.
- **Circuit breaker/retry örneklerine ne koruduklarıyla temiz eşleşen isimler ver** -- burada `inventoryService`, ilgisiz çağrılar arasında paylaşılan genel bir isim değil.
- **Geliştirme sırasında durum geçişlerini logla (ya da metrik olarak sun)**, tıpkı `CircuitBreakerEventListener` gibi -- sessizce başarısız olan açık bir devreyi teşhis etmek zordur.

Yaygın Hatalar

- **`@CircuitBreaker` / `@Retry` ile işaretli bir metodu AYNI sınıftaki başka bir metottan çağırmak.** Bu, Spring'in proxy'sini tamamen atlar -- iki annotation'ın da hiçbir etkisi olmaz (yukarıdaki uyarıya bakınız).
- **İmzası eşleşmeyen bir fallback metodu yazmak.** Orijinal metodun parametrelerini artı sonda bir `Throwable`'ı kabul etmeli, aksi halde Resilience4j onu bağlayamaz.
- **`wait-duration-in-open-state`'i çok kısa ayarlamak.** Devre, muhtemelen henüz toparlanmamış bir servise test çağrısını yeniden açar, servise nefes alma alanı verme amacını boşa çıkarır.
- **Bir bulkhead ya da rate limiter'ı bir circuit breaker'ın YERİNE kullanmak.** Bunlar FARKLI risklere karşı korur (aşırı yük vs. sürekli başarısızlık) -- gerçekten düşmüş bir servis hâlâ bir circuit breaker'a ihtiyaç duyar, hiçbir eşzamanlılık sınırlaması bunu düzeltmez.

Özet, Cheat Sheet ve Terimler Sözlüğü

Resilience4j, riskli bir çağrıyı annotation'larla uygulanan koruyucu davranışlarla sarar: `@CircuitBreaker` sürekli başarısız olan bir servisi çağırmayı durdurur (CLOSED -> OPEN -> HALF_OPEN -> CLOSED), `@Retry` geçici başarısız olan bir çağrıyı vazgeçmeden önce tekrar dener, ve rate limiter/bulkhead'ler servis sağlıklıyken bile aşırı yüke karşı korur. Bir fallback metodu, devre açık olduğunda ya da retry'lar tükendiğinde gerçek metodun yerine çalışır -- imzası artı sonda bir `Throwable` eşleşmeli. Bunların hepsi yalnızca Spring'in proxy mekanizması üzerinden çalışır, bu yüzden aynı sınıf içindeki self-invocation bunu tamamen atlar.

Hızlı referans:

```
@CircuitBreaker(name = "inventoryService", fallbackMethod = "checkStockFallback")
@Retry(name = "inventoryService")
StockCheckResponse checkStock(String productName) { ... }

StockCheckResponse checkStockFallback(String productName, Throwable t) {
    return new StockCheckResponse(productName, 0);   // bozuk ama geçerli yanıt
}

// application.yml
// resilience4j.circuitbreaker.instances.inventoryService.failure-rate-threshold: 50
// resilience4j.retry.instances.inventoryService.max-attempts: 3
```

Terimler Sözlüğü

Circuit Breaker — Sürekli başarısız olan bir servisi çağırmayı durduran, CLOSED, OPEN ve HALF_OPEN durumları arasında dönen bir koruma.

Retry — Başarısız bir çağrıyı, vazgeçmeden önce yapılandırılmış bir sayıda otomatik olarak tekrar deneyen bir koruma.

Rate Limiter — Başarı/başarısızlıktan bağımsız olarak bir zaman penceresi içinde kaç çağrıya izin verildiğini sınırlayan bir koruma.

Bulkhead — Bir bağımlılığa aynı anda kaç çağrının uçuşta olabileceğini sınırlayan bir koruma.

Fallback Metodu — Bir devre açık olduğunda ya da retry'lar tükendiğinde Resilience4j'nin gerçek metot yerine çağırdığı metot.