Making Microservices Resilient

Trong kiến trúc microservices, việc xây dựng các dịch vụ có khả năng chống chịu với lỗi (resilient) là vô cùng quan trọng nhằm đảm bảo hệ thống luôn sẵn sàng và ổn định, ngay cả khi một số thành phần gặp sự cố. Bài học này tập trung vào các mẫu thiết kế resiliency phổ biến và cách triển khai chúng trong hệ thống microservices sử dụng Spring Cloud và các công cụ hỗ trợ.

1. **Nhu cầu về Resiliency trong Microservices**

Microservices thường giao tiếp với nhau thông qua các API. Khi một service chậm phản hồi hoặc không khả dụng, nếu không có các cơ chế kiểm soát, sẽ kéo theo lỗi dây chuyền, ảnh hưởng tới toàn bộ hệ thống. Do đó, việc trang bị cho mỗi service các khả năng tự bảo vệ và xử lý lỗi là điều thiết yếu.

Ví dụ: Client gọi API của Account, API này phụ thuộc vào Loans và Cards. Nếu Cards bị lỗi hoặc chậm → cả hệ thống bị ảnh hưởng.

2. **Các câu hỏi quan trọng đặt ra:**

Làm sao tránh hiệu ứng dây chuyền khi một service gặp sự cố?

- Cần ngắt kết nối sớm, không đợi response quá lâu.
- Hạn chế chiếm dụng tài nguyên (thread, memory).

Xử lý lỗi bằng fallback như thế nào?

- Nếu một service bị lỗi, có thể **gửi thông tin còn lại** từ các service hoạt động bình thường.
- Fallback có thể là: dữ liệu mặc định, cache, database phụ, hoặc service thay thế.

Làm sao để service có khả năng tự phục hồi (self-healing)?

- Cấu hình **timeout và retry** hợp lý.
- Giảm thời gian chờ → giảm áp lực tài nguyên → cho phép service "hồi phục".

3. **Giải pháp: Sử dụng các mẫu (patterns) resilient**

Các pattern hỗ trợ bởi Resilience4j:

Pattern	Công dụng chính
Circuit Breaker	Ngắt kết nối khi phát hiện lỗi liên tiếp
Fallback	Đưa ra giá trị thay thế khi service chính thất bại
Retry	Thử gọi lại một số lần trước khi báo lỗi
Rate Limiter	Giới hạn số lượng request trong một khoảng thời gian
Bulkhead	Giới hạn tài nguyên cho từng service độc lập
Time Limiter	Cài đặt timeout cho các lời gọi
Cache	Trả dữ liệu cache khi hệ thống chính không khả dụng

4. **Circuit Breaker Pattern**

Circuit Breaker giúp bảo vệ hệ thống khỏi lỗi dây chuyền khi một hoặc nhiều microservice gặp lỗi, bằng cách:

- **Giảm thời gian phản hồi:** Tránh chờ đợi response từ microservice đã chết.
- **Cải thiện trải nghiệm người dùng:** Trả về dữ liệu fallback thay vì lỗi hệ thống.
- **Tăng tính sẵn sàng:** Ngăn chặn lỗi lan rộng đến các thành phần khác.
- Gồm 3 trạng thái: **Closed**, **Open**, **Half-Open**.

Closed State (trạng thái mặc định):

- Mọi request được phép đi qua.
- Nếu **tỷ lệ lỗi vượt ngưỡng**, Circuit Breaker sẽ **chuyển sang trạng thái Open**.

Open State:

- Không request nào được gửi đến service bị lỗi (ví dụ: Accounts MS).
- Circuit Breaker **ngay lập tức trả về lỗi 503** (Service Unavailable) → Giúp tiết kiệm tài nguyên cho Gateway.
- Tránh việc "chờ 10s vô ích" như đã thấy trước đó.

Half-Open State:

- Sau một thời gian chờ (ví dụ: 10s), Circuit Breaker **thử gửi lại một vài request kiểm tra**.
- Nếu các request thành công → Trạng thái chuyển về **Closed**.
- Nếu vẫn thất bại → Quay lại **Open**.

Circuit Breaker được triển khai ở cả **API Gateway** và **Feign Client**, giúp kiểm soát cả phía gọi và phía được gọi.

Các bước cấu hình Circuit Breaker với OpenFeign + Resilience4j

Thêm dependency vào pom.xml

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

Định nghĩa FeignClient với fallback

```
@FeignClient(name = "cards", fallback = CardsFallback.class)
public interface CardsFeignClient {
    @GetMapping("/cards/{mobileNumber}")
    List<CardDto> fetchCardDetails(@PathVariable String mobileNumber);
}
```

Tạo lớp fallback cho FeignClient

```
@Component
public class CardsFallback implements CardsFeignClient {
    @Override
    public List<CardDto> fetchCardDetails(String mobileNumber) {
        // Trả về dữ liệu fallback khi Cards service không phản hồi
        return new ArrayList<>();
    }
}
```

Cấu hình trong application.yml

```
resilience4j:
  circuitbreaker:
    instances:
```

```
    cardsFeignClient:
      registerHealthIndicator: true
      slidingWindowSize: 5
      minimumNumberOfCalls: 5
      failureRateThreshold: 50
      waitDurationInOpenState: 5s
      permittedNumberOfCallsInHalfOpenState: 3
```

5. **Cấu hình Timeout trong Spring Cloud Gateway**

Khi gọi một API từ client (ví dụ như Postman) tới một dịch vụ như LoansController, bình thường phản hồi sẽ đến nhanh chóng. Tuy nhiên, trong thực tế, có thể xảy ra tình trạng dịch vụ phản hồi rất chậm (30 giây, 1 phút hoặc hơn), khiến cho cả gateway lẫn microservice đích phải giữ tài nguyên (thread, memory) chờ phản hồi – điều này dẫn tới hiệu năng hệ thống bị suy giảm.

Giảng viên minh họa bằng cách đặt breakpoint trong LoansController khiến phản hồi bị trì hoãn. Kết quả là Postman tiếp tục chờ trong hơn 2 phút để nhận được kết quả – một tình huống không nên xảy ra trong các hệ thống sản xuất.

Giải pháp: Timeout Configuration

Để giải quyết vấn đề này, ta cần cấu hình timeout trong Gateway để giới hạn thời gian chờ phản hồi từ các microservice. Có hai loại timeout chính:

- connection-timeout: thời gian tối đa để Gateway thiết lập kết nối tới microservice.
- response-timeout: thời gian tối đa để Gateway chờ phản hồi từ microservice.

```
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 2s
```

6. **Retry Pattern**

Retry là kỹ thuật thử lại các yêu cầu khi có lỗi tạm thời như time-out hoặc connection refused:

- Giúp tăng khả năng phản hồi thành công khi service tạm thời gián đoạn.
- Có thể cấu hình số lần thử lại và khoảng thời gian giữa các lần thử.

Retry được áp dụng ở cả Gateway và từng microservice như accounts.

Cấu hình cơ bản

Trong microservice (ví dụ accounts-service), ta có thể áp dụng Retry Pattern bằng cách:

- Sử dụng annotation @Retry(name = "retryName", fallbackMethod = "fallbackMethod") trên phương thức cần retry.
- Tạo một phương thức fallback cùng tên, nhận cùng tham số với phương thức gốc, kèm thêm một tham số kiểu Throwable.

Cấu hình trong `application.yml`

- Trong file `application.yml`, bạn có thể định nghĩa các thuộc tính sau:

```yaml
resilience4j.retry:
  instances:
    retryName:
      max-attempts: 3
      wait-duration: 1s
```

- `max-attempts`: số lần thử lại tối đa.
- `wait-duration`: khoảng thời gian chờ giữa các lần thử lại.

Loại trừ và chỉ định Exception

Retry Pattern có thể cấu hình để **bỏ qua** hoặc **chỉ retry** với một số loại Exception cụ thể:

- **Bỏ qua exception (ignoreExceptions)**: Ví dụ: không retry khi gặp NullPointerException vì lỗi này không thể sửa được bằng cách retry.

```yaml
ignore-exceptions:
  - java.lang.NullPointerException
```

- Chỉ retry một số exception cụ thể (retryExceptions): Khi sử dụng retryExceptions, hệ thống chỉ retry nếu gặp các exception này. Tất cả exception còn lại sẽ bị bỏ qua.

```yaml
retry-exceptions:
  - java.util.concurrent.TimeoutException
```

Lưu ý: Không nên cấu hình cả ignoreExceptions và retryExceptions cùng lúc. Khi đã dùng retryExceptions, các exception khác sẽ tự động bị bỏ qua.

7. **Retry trong Spring Cloud Gateway**

Gateway cũng hỗ trợ Retry Pattern, nhưng chỉ cung cấp tùy chọn cấu hình **các exception để retry** (qua Java DSL như setExceptions(...)). Gateway không hỗ trợ cấu hình ignoreExceptions. Ngoài ra, cũng có thể cấu hình retry dựa trên **HTTP status code**.

```
.retry(config -> config
    .setRetries(3)
    .setStatuses(HttpStatus.INTERNAL_SERVER_ERROR)
    .setExceptions(IOException.class)
)
```

8. **Rate Limiter Pattern**

Rate Limiting kiểm soát số lượng request gửi đến một service trong khoảng thời gian cụ thể:

- Tránh tình trạng bị "flood" hoặc quá tải.
- Trong khóa học, pattern này được triển khai bằng **Redis RateLimiter** tại Gateway và các service con.

Triển khai RateLimiter bằng Resilience4j trong Microservice

Bên cạnh cách triển khai tại Gateway, chúng ta có thể sử dụng **Resilience4j RateLimiter** trực tiếp bên trong một microservice (ví dụ như AccountsService). Việc này đặc biệt hữu ích trong các tình huống:

- Cần giới hạn lưu lượng cho một API cụ thể, ví dụ API có độ ưu tiên thấp.
- Cơ sở hạ tầng chỉ cho phép xử lý tối đa một số lượng request nhất định mỗi giây.

Cách cấu hình và sử dụng

a. Thêm annotation @RateLimiter

Annotation @RateLimiter(name = "getJavaVersion", fallbackMethod = "...") được thêm vào phương thức cần giới hạn, ví dụ getJavaVersion(). Tên RateLimiter nên trùng với tên cấu hình bên trong application.yml.

b. Cấu hình trong application.yml

```
resilience4j.ratelimiter:
  instances:
    getJavaVersion:
      limit-refresh-period: 5s
      limit-for-period: 1
      timeout-duration: 1s
```

- `limit-refresh-period`: Thời gian làm mới hạn mức (ở đây là 5 giây).
- `limit-for-period`: Số request tối đa trong mỗi chu kỳ (ở đây là 1).
- `timeout-duration`: Thời gian tối đa mà thread sẽ chờ để lấy quota mới trước khi bị từ chối.

9. Fallback Method

Khi số lượng request vượt quá hạn mức, fallback method sẽ được gọi thay vì trả lỗi 500. Ví dụ, getJavaVersionFallback(Throwable t) có thể trả về dữ liệu mặc định như "Java 17" hoặc thông báo thân thiện: "Please try again later".

Lợi ích và ứng dụng thực tế

- Tránh quá tải hệ thống hoặc bảo vệ API có chi phí xử lý cao.
- Phân loại API theo ưu tiên cao/thấp, từ đó cấp hạn mức phù hợp.
- Dễ dàng mở rộng và tích hợp với các pattern khác như Retry, Circuit Breaker, hoặc Fallback.

10. Bulkhead Pattern

Bulkhead giúp cô lập các phần khác nhau của hệ thống để tránh ảnh hưởng lẫn nhau:

- Nếu một phần của service bị lỗi hoặc quá tải, phần còn lại vẫn hoạt động bình thường.
- Giúp tối ưu tài nguyên, tăng độ ổn định tổng thể của hệ thống.

Lợi ích chính của Bulkhead Pattern

- **Tăng tính ổn định của hệ thống** bằng cách phân chia tài nguyên (thread, pool...) giữa các thành phần.
- **Cô lập lỗi**, đảm bảo rằng một REST API hoạt động chậm hay lỗi không ảnh hưởng đến các API khác.
- **Giảm nguy cơ quá tải**, đặc biệt trong trường hợp các API có độ phức tạp xử lý khác nhau.

Ví dụ minh họa

Giả sử một microservice AccountsService có hai REST API:

- myAccount: đơn giản, không phụ thuộc microservice khác.
- myCustomerDetails: phức tạp, gọi thêm LoansService và CardsService.

Nếu không dùng Bulkhead: Khi myCustomerDetails nhận nhiều request, nó có thể chiếm toàn bộ tài nguyên (threads), khiến myAccount không có tài nguyên để phản hồi → ảnh hưởng cả hệ thống.

Với Bulkhead: Ta cấu hình giới hạn riêng cho mỗi API. Ví dụ, mỗi API chỉ được sử dụng một số thread nhất định. Nhờ đó, myAccount vẫn phản hồi tốt ngay cả khi myCustomerDetails quá tải.

11. Thứ tự áp dụng các Resiliency Patterns

Một điểm quan trọng là phải áp dụng các mẫu thiết kế này theo đúng thứ tự (Aspect Order), thường là: **RateLimiter → Bulkhead → CircuitBreaker → Retry**

Ứng dụng trong Microservices

- Trong hệ thống phân tán (distributed system), các microservice thường gọi lẫn nhau qua mạng.
- Các lỗi như: chậm phản hồi, lỗi mạng, hết tài nguyên... là thường gặp và thường chỉ là **tạm thời**.

Lợi ích chính

- **Fail Fast:** Giảm thời gian chờ của dịch vụ gọi (VD: từ 10s xuống còn 1s).
- **Fail Gracefully:** Có thể tích hợp fallback logic để trả về phản hồi mặc định khi lỗi xảy ra.
- **Giúp phục hồi nhanh:** Ngắt lưu lượng đến dịch vụ bị lỗi → giảm tải, giúp nó hồi phục sớm hơn.

12. Tình huống mô phỏng lỗi trong microservices:

1. **Kiến trúc tổng thể:**
    - Có một **API Gateway (Edge Server)** nhận các request từ client.
    - Request được chuyển tiếp đến **Accounts Microservice** để xử lý.
    - Accounts MS có thông tin về **tài khoản và khách hàng**, nhưng cần gọi thêm **Loans** và **Cards Microservice** để lấy thông tin đầy đủ.
2. **Vấn đề xảy ra:**
    - **Cards Microservice bị lỗi** hoặc phản hồi chậm.

- Accounts MS phải **chờ response** từ Cards MS → chiếm tài nguyên (thread, CPU, RAM...).
- Nếu có nhiều request cùng lúc → Accounts MS **bị nghẽn**.
- Kết quả là Edge Server cũng **bị nghẽn**, ảnh hưởng đến toàn bộ hệ thống, kể cả các API khác.

3. **Hậu quả:**
   - Một lỗi nhỏ ở một service (Cards MS) **lan rộng** và ảnh hưởng đến toàn bộ hệ thống – gọi là **ripple effect**.

13. Tình huống thực tế về sự cố trong hệ thống Microservices

1. Mô hình hoạt động thông thường:

- Client gửi request đến **Edge Server (API Gateway)**.
- Gateway chuyển tiếp request đến **Accounts Microservice** để lấy thông tin khách hàng (fetchCustomerDetails).
- Accounts Microservice sẽ:
  - Trả thông tin về account & customer từ chính nó.
  - Gọi **Loans Microservice** và **Cards Microservice** để lấy thêm thông tin về khoản vay và thẻ.
  - Tổng hợp dữ liệu và gửi lại về Gateway → Client.

2. Tình huống lỗi (Cards Microservice gặp sự cố):

- **Cards Microservice phản hồi chậm hoặc không phản hồi**.
- Accounts Microservice sẽ **chờ** phản hồi từ Cards, ví dụ chờ **10s hoặc hơn**.
- Tài nguyên như **thread, CPU, memory** bị giữ lại trong lúc chờ.

3. Hiệu ứng lan truyền (Ripple Effect):

- Accounts Microservice phản hồi chậm do chờ Cards.
- **Edge Server** cũng bị chậm theo vì đang đợi Accounts.

- **Tài nguyên của Edge Server** cũng bị chiếm giữ → các API khác cũng bị ảnh hưởng.
- **Toàn bộ hệ thống trở nên chậm chạp**, chỉ vì 1 microservice hoạt động không hiệu quả.

14. Demo of Resiliency Patterns using Docker & Docker Compose

Các Resiliency Patterns được demo:

- Retry: Tự động thử lại khi gặp lỗi tạm thời (ví dụ timeout hoặc lỗi 500).
- Timeout: Đặt thời gian giới hạn để tránh request treo lâu ảnh hưởng toàn hệ thống.
- Circuit Breaker: Ngắt dòng xử lý tạm thời nếu một service liên tục lỗi, giúp tránh "domino effect".
- Rate Limiting: Giới hạn số lượng request từ client nhằm bảo vệ service backend.

Công nghệ sử dụng:

- Spring Cloud Resilience4j: Thư viện cung cấp các resiliency patterns.
- Docker Compose: Dùng để dựng toàn bộ hệ thống microservices, có cấu hình lỗi mô phỏng (ví dụ delay, crash service).

Demo thực tế:

- Mỗi service được chạy trong container riêng biệt.
- Một số services cố tình cấu hình delay hoặc lỗi để kiểm tra cơ chế fallback.
- Logs, metrics và tracing được tích hợp với Grafana stack để quan sát hiệu quả của các patterns khi hệ thống gặp sự cố.