Netflix OSS

Netflix Open Source Software (OSS) is a collection of tools and libraries that Netflix developed and open-sourced to address the challenges of building and running a large-scale, distributed microservices architecture. These tools were instrumental in Netflix's transition from a monolithic application to a microservices-based system. Key components of the Netflix OSS ecosystem include:

- **Eureka**: A service registry for service discovery.
- **Ribbon** (now replaced by Spring Cloud Load Balancer): A client-side load balancer.
- **Hystrix** (now in maintenance mode, replaced by Resilience4j): A circuit breaker for handling failures in a distributed system.
- **Zuul** (now replaced by Spring Cloud Gateway): An API Gateway for routing and filtering requests.

Spring Cloud

Spring Cloud is a project that provides tools for developers to quickly build common patterns in distributed systems, such as configuration management, service discovery, and load balancing. It integrates many of the Netflix OSS components (or their modern equivalents) with the Spring framework, making it easy to create and manage microservices. Spring Cloud projects include:

- **Spring Cloud Netflix**: Provides integration with Netflix OSS components like Eureka.
- **Spring Cloud Gateway**: A modern, reactive API gateway.
- **Spring Cloud Load Balancer**: A client-side load balancer.
- **OpenFeign**: A declarative REST client.

Service Discovery with Spring Cloud Netflix Eureka

In a microservices architecture, services need to find and communicate with each other. This is called service discovery. With Spring Cloud Netflix Eureka, this is handled by a central server (the Eureka Server) and clients (the microservices themselves).

- **Eureka Server**: A service registry that acts as a central hub where all other microservices register themselves.
- **Eureka Client**: Each microservice is a client that registers its location (hostname, port) with the Eureka Server upon startup. It also sends periodic "heartbeats" to the server to let it know it's still alive. Other microservices can query the Eureka Server to find the location of a service they need to call.

Client-Side Load Balancing with Spring Cloud Load Balancer

Client-side load balancing is a strategy where the client service is responsible for distributing requests across multiple available instances of a target service.

- **How it works**: The client, using information from the service discovery registry (like Eureka), gets a list of all available instances for a particular service. It then uses a load-balancing algorithm (e.g., Round Robin) to choose one instance from the list for each request. This is different from a traditional server-side load balancer, which sits in front of the services. Spring Cloud Load Balancer provides this functionality out of the box.

OpenFeign – Declarative REST Client

OpenFeign is a declarative REST client. It simplifies the process of making HTTP requests to other microservices.

- **How it works**: Instead of manually building `RestTemplate` or `WebClient` calls, you define an interface with annotations that map to the REST endpoints of the target service. Spring Cloud automatically generates an implementation of this interface and handles the details of the HTTP communication, including service discovery and client-side load balancing. This makes the code cleaner and more readable.

Circuit Breakers and Cascading Failures

In a microservices architecture, a **cascading failure** occurs when the failure of one service leads to failures in other dependent services, eventually bringing down a large part of the system. Imagine a chain of services: `Service A` calls `Service B`, which calls `Service C`. If `Service C` starts to fail (e.g., due to a database connection issue or high load), `Service B`'s requests to it will time out. The threads in `Service B` will become tied up waiting for a response that never comes. This leads to a thread pool exhaustion in `Service B`, causing it to fail. Now, when `Service A` calls the now-failing `Service B`, the same problem propagates, and `Service A` eventually fails, taking down the whole chain. 😬

A **software circuit breaker** is a design pattern that prevents this from happening. It acts like an electrical circuit breaker, monitoring calls to a service. If the number of failures or timeouts exceeds a certain threshold, the circuit "trips" and enters an **open** state. In this state, the circuit breaker immediately rejects all subsequent calls to the failing service without even attempting to make the call. This is known as "failing fast."

This provides several key benefits:

- **It frees up resources** in the calling service, preventing its thread pool from becoming exhausted.
- **It provides time for the failing service to recover**, as it's no longer being bombarded with requests.
- **It allows the calling service to handle the failure gracefully** by providing a **fallback** response, such as cached data or a default message, instead of throwing an error.

The circuit breaker has three states:

1. **Closed**: This is the normal state. Requests are allowed to pass through to the service. The circuit breaker monitors for failures.

2. **Open**: When the failure threshold is reached, the circuit opens. All calls are immediately rejected, and a fallback is executed.

3. **Half-Open**: After a configurable waiting period (the `waitDurationInOpenState`), the circuit transitions to this state. It allows a limited number of "test" requests to pass through. If these requests are successful, it assumes the service has recovered and returns to the `Closed` state. If they fail, it immediately goes back to the `Open` state.

Spring Cloud Circuit Breaker with Resilience4J

Resilience4J is a lightweight, easy-to-use fault tolerance library that's well-suited for functional programming. **Spring Cloud Circuit Breaker** provides an abstraction layer that allows you to easily integrate Resilience4J (or other libraries) into your Spring Boot application.

1. Setup

First, you need to set up a Spring Boot project with the necessary dependencies.

- Create a new Spring Boot project.
- Add the following dependencies to your `pom.xml`:
    - `spring-boot-starter-web`: For creating REST endpoints.
    - `spring-cloud-starter-circuitbreaker-resilience4j`: The core dependency for the circuit breaker.
    - `spring-boot-starter-actuator`: For monitoring circuit breaker health and metrics.
    - `io.micrometer:micrometer-registry-prometheus`: To expose metrics in a format Prometheus can scrape.

XML

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-
resilience4j</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

2. Configuration

Next, configure the circuit breaker in your `application.yml` or `application.properties` file. This is where you define the behavior of your circuit breaker instance.

YAML

```
spring:
  application:
    name: circuit-breaker-demo
management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    health:
      show-details: always
    metrics:
      enabled: true
    prometheus:
      enabled: true
resilience4j:
  circuitbreaker:
    instances:
      externalService:
        registerHealthIndicator: true
        slidingWindowType: COUNT_BASED
        slidingWindowSize: 10
        failureRateThreshold: 50
        minimumNumberOfCalls: 5
        permittedNumberOfCallsInHalfOpenState: 3
        waitDurationInOpenState: 5s
```

- `externalService`: This is the unique name for your circuit breaker instance.
- `slidingWindowSize`: The number of recent calls to consider for calculating the failure rate.
- `failureRateThreshold`: If more than 50% of the calls in the sliding window fail, the circuit will open.
- `minimumNumberOfCalls`: The circuit breaker won't start calculating the failure rate until this many calls have been made.

- `waitDurationInOpenState`: The duration the circuit stays `Open` before moving to `Half-Open`.

3. Implementation

You can apply the circuit breaker to a method using the `@CircuitBreaker` annotation. It's often used in a service layer class that calls an external dependency.

Java

```
package com.example.demo.service;

import
io.github.resilience4j.circuitbreaker.annotation.CircuitBreak
er;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class ExternalService {

    private final RestTemplate restTemplate = new
RestTemplate();

    @CircuitBreaker(name = "externalService", fallbackMethod
= "fallbackMethod")
    public String callExternalApi() {
        // Simulating a call to a potentially failing
external service
        // Let's say this URL fails sometimes
        String response = restTemplate.getForObject("http://
localhost:9090/flaky-endpoint", String.class);
        return "Success: " + response;
    }

    // The fallback method must have the same signature as
the original method,
    // plus an extra Throwable parameter
    public String fallbackMethod(Throwable t) {
        return "Fallback: Service is currently unavailable. "
+ t.getMessage();
    }
}
```

- The `@CircuitBreaker(name = "externalService")` annotation links this method to the configuration you defined in `application.yml`.
- The `fallbackMethod` attribute specifies which method to call when the circuit is `Open` or a failure occurs.

4. Controller

Finally, create a controller to expose the endpoint that calls your service.

Java

```
package com.example.demo.controller;

import com.example.demo.service.ExternalService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {

    @Autowired
    private ExternalService externalService;

    @GetMapping("/call")
    public String makeCall() {
        return externalService.callExternalApi();
    }
}
```

Monitoring with Prometheus

Spring Boot Actuator and Micrometer make it easy to expose circuit breaker metrics that can be scraped by a Prometheus server and visualized in Grafana.

1. **Run your Spring Boot application.**

2. **Access the metrics endpoint**: Navigate to `http://localhost:8080/actuator/prometheus` (assuming your app is on port 8080). You will see a list of metrics exposed by Resilience4J, such as `resilience4j_circuitbreaker_state`, `resilience4j_circuitbreaker_calls_total`, and `resilience4j_circuitbreaker_failure_rate`.

3. **Set up Prometheus**:

    - Create a `prometheus.yml` file with a `scrape_config` to target your application's metrics endpoint.

4. YAML

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'spring-app'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['localhost:8080']
```

    - Start the Prometheus server with this configuration file.

12. **Set up Grafana**:

    - Launch Grafana and add Prometheus as a data source.

    - Import a pre-built Resilience4j dashboard (there are many available on the Grafana website) or create your own panels. You can plot metrics like `resilience4j_circuitbreaker_state` to see the state changes (e.g., `OPEN`, `HALF_OPEN`, `CLOSED`) over time, or `resilience4j_circuitbreaker_calls_total` to track the number of successful and failed calls.

This setup provides a powerful dashboard to proactively monitor the health of your services and see when your circuit breakers are actively protecting your system.