Setup a Circuit Breaker with Hystrix, Feign Client and Spring Boot

In a microservices architecture, several things can go wrong. A middleware, the network or the service you want to contact can be down. In this world of uncertainty, you have to anticipate problems in order not to break the entire chain and throw an error to the end user when you could offer a partially degraded service instead.

The goal of this article is to show how to implement the circuit breaker pattern using Hystrix, Feign Client and Spring Boot.

Feign Client Crash Course

[Feign](https://github.com/OpenFeign/feign) is an HTTP client created by Netflix to make HTTP communications easier. It is integrated to Spring Boot with the spring-cloud-starter-feign starter.

To create a client to consume an HTTP service, an interface annotated with @FeignClient must be created. Endpoints can be declared in this interface using an API that is very close to the Spring MVC API. The @EnableFeignClients annotation must also be added to a Spring Configuration class.

|  |
| --- |
| @Configuration  @EnableFeignClients  public class FeignConfiguration {  } |

|  |
| --- |
| @FeignClient(name = "videos", url = "http://localhost:9090/videos")  public interface VideoClient {  @PostMapping(value = "/api/videos/suggest")  List<Suggestion> suggest(@RequestBody ViewingHistory history);  } |

An instance of VideoClient is automagically injected into the Spring application context and can be autowired and used throughout the application. Moreover, if the videos microservice is registred to the same discovery service as the current microservice, there is no need for an URL as it will be retrieved for you based on the name.

If the videos service, a middleware or the network happens to be down or overloaded, the suggest method will throw a FeignException that will be propagated throughout the stack if not caught.

## Create a Fallback Implementation

Fortunately, Spring Cloud comes with a solution to this problem: a circuit breaker. In this article, we will use [Hystrix](https://github.com/Netflix/Hystrix). It is also created by Netflix and also integrated to Spring Boot using the spring-cloud-starter-hystrix starter.

The idea is to create an implementation of the VideoClient and mark it as the default behaviour if videos is unreachable or overloaded. Like a lot of other Spring features, it is enabled using an annotation: @EnableCircuitBreaker.

|  |
| --- |
| @Configuration  @EnableFeignClients  @EnableCircuitBreaker  public class FeignConfiguration {  } |

|  |
| --- |
| @FeignClient(name = "videos", url = "http://localhost:9090/videos", fallback = VideoClientFallback.class)  public interface VideoClient {  @PostMapping(value = "/api/videos/suggest")  List<Suggestion> suggest(@RequestBody ViewingHistory history);  } |

|  |
| --- |
| @Component  public class VideoClientFallback implements VideoClient {  @Override  public List<Suggestion> suggest(ViewingHistory history) {  // Degraded service: no suggestion to offer  return new ArrayList<>();  }  } |

A configuration property has to be added to the application.yml file of the Spring Boot application to tell Feign to enable Hystrix.

|  |
| --- |
| feign:  hystrix:  enabled: true |

Voila! Every time the remote service will be unavailable, the suggest method of the VideoClientFallback will be called and the end user will not get an error violently thrown at her.

Keep Track of the Source Error

With this setup, the fallback will be called regardless of the initial error that will be swallowed. If you want to retrieve this error and do something with it, you can use a FallbackFactory.

|  |
| --- |
| @FeignClient(name = "videos", url = "http://localhost:9090/videos", fallbackFactory = VideoClientFallbackFactory.class)  public interface VideoClient {  @PostMapping(value = "/api/videos/suggest")  List<Suggestion> suggest(@RequestBody ViewingHistory history);  } |

|  |
| --- |
| @Component  public class VideoClientFallbackFactory implements FallbackFactory<VideoClient> {  @Override  public VideoClient create(Throwable throwable) {  return new VideoClientFallback(throwable);  }  } |

|  |
| --- |
| public class VideoClientFallback implements VideoClient {  private final Throwable cause;  public VideoClientFallback(Throwable cause) {  this.cause = cause;  }  @Override  public List<Suggestion> suggest(ViewingHistory history) {  if (cause instanceof FeignException && ((FeignException) cause).status() == 404) {  // Treat the HTTP 404 status  }  return new ArrayList<>();  }  } |

## Conclusion

Microservices foster low coupling between components and resiliency. Hence, it would be sad to throw an error every time a service or a middleware is down. The circuit breaker pattern explained in this article allows you to ensure the continuity of service, even if it has to be offered in a degraded manner. As always, Spring Boot is a great help to setup this mechanism very easily.