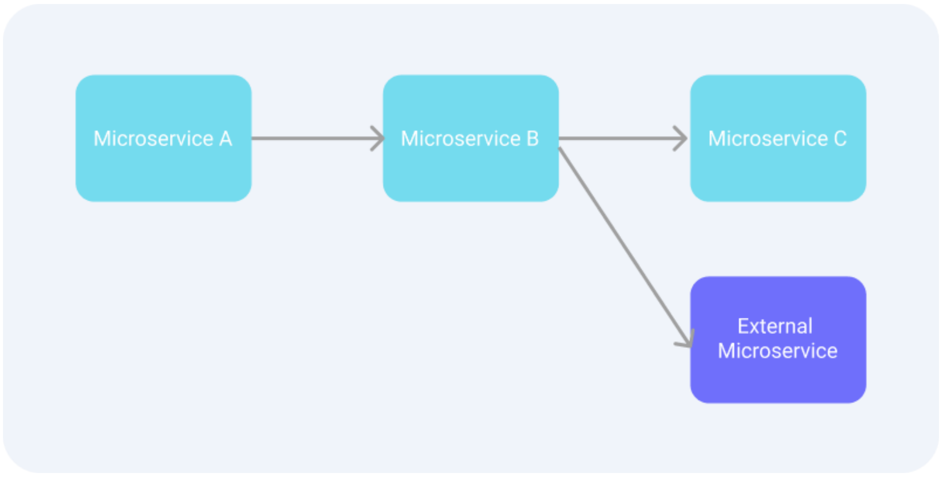
Resilience4J: Circuit Breaker Imp on Spring Boot

Pramuditya Ananta Nur Follow
Jul 2 · 7 min read

> *In a microservice architecture, it's common for a service to call another service. And there is always the possibility that the other service being called is unavailable or unable to respond. So, what can we do when this happens?*

Introduction



Microservices illustration

Let's take a look at example cases below.

- When **microservice B** calls **microservice C** and **microservice C** unavailable or unable to respond, **microservice B** may have encountered an error. And if **microservice A** calls **microservice B** which is encountered an error, this causes **microservice A** is just wasting its resources, *right?*
  Actually, we can tune or optimize the microservice, 'cause it's still within the scope of our workspace.

- When **microservice B** calls the **external microservice** and unfortunately the **external microservice** is down. Actually, **microservice B** is just wasting its resources. This will also definitely affect **microservice B** when called by **microservice A**. In this case, there is nothing to do except wait for a third party to check what's wrong with their microservice.

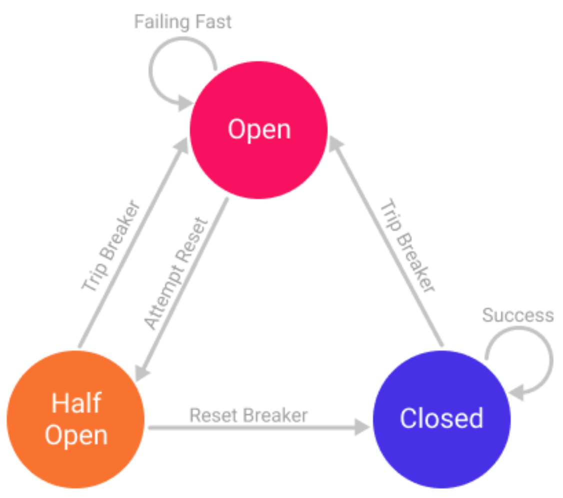
From the 2 cases above, we can conclude that when a microservice encounters an error, it will have an impact on other microservices that call it, and will also have a *domino* effect. Then, what can be done to prevent a *domino* effect like the cases above? Well, the answer is a **circuit breaker** mechanism.

What is Circuit Breaker?

The concept of a circuit breaker is to prevent calls to microservice when it's known the call may fail or time out. This is done so that clients don't wa handling requests that are likely to fail. Using this concept, y spare time to recover.

So, how do we know if a request is likely to fail? Yeah, this ca the results of several previous requests sent to other microse 5 requests sent failed or timeout, then most likely the next re the same thing.

Circuit Breaker State



Circuit Breaker State

In the circuit breaker, there are 3 states **Closed**, **Open**, and **Half-Open**.

- **Closed**: when everything is normal. Initially, the circuit breaker is in a **Closed state.**
- **Open:** when a failure occurs above predetermined criteria. In this state, requests to other microservices will not be executed and *fail-fast* or *fallback* will be performed if available. When this state has passed a certain time limit, it will automatically or according to certain criteria will be returned to the **Half-Open state.**
- **Half-Open:** several requests will be executed to find out whether the microservices that we are calling are working normally. If successful, the state will be returned to the **Closed state**. However, if it still fails it will be returned to the **Open state**.

Circuit Breaker Type

There are 2 types of circuit breaker patterns, **Count-based** and **Time-based.**

- **Count-based**: the circuit breaker switches from a closed state to an open state when the last N requests have failed or timeout.
- **Time-based**: the circuit breaker switches from a closed state to an open state when the last N time unit has failed or timeout.

In both types of circuit breakers, we can determine what the threshold for failure or timeout is. Suppose we specify that the circuit breaker will trip and go to the **Open state** when 50% of the last 20 requests took more than 2s, or for a time-based, we can specify that 50% of the last 60 seconds of requests took more than 5s.

After we know how the circuit breaker works, then we will try to implement it in the spring boot project.

Prerequisites

One of the libraries that offer a circuit breaker features is **Re** example project, we'll use this library. First, we create a spri required dependencies:

```xml
<dependencies>
  <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>
  <dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-reactor</artifactId>
  </dependency>
</dependencies>
```

pom.xml hosted with ❤ by GitHub view raw

Implementation

We will create a simple REST API to start simulating a circuit breaker. This REST API will provide a response with a time delay according to the parameter of the request we sent. For example, if we send a request with a delay of 5 seconds, then it will return a response after 5 seconds. In the other words, we will make the circuit breaker trips to an **Open State** when the response from the request has passed the time unit threshold that we specify.

```java
private static final String RESILIENCE4J_INSTANCE_NAME = "example";

@CircuitBreaker(name = RESILIENCE4J_INSTANCE_NAME)
public Mono<Response<Boolean>> delay(@PathVariable int delay) {
  return Mono.just(toOkResponse())
      .delayElement(Duration.ofSeconds(delay));
}
```

CircuitBreakerController.java hosted with ❤ by GitHub view raw

Pay attention to line 30. An API with a circuit breaker is simply marked using the **@CircuitBreaker** annotation followed by the name of the circuit breaker.

Next, we will configure what conditions will cause the circuit breaker to trip to the **Open State**.

```properties
# Resiliece4j Configuration
resilience4j.circuitbreaker.configs.shared.register-health-indicator=true
resilience4j.circuitbreaker.configs.shared.sliding-window-type=count_based
resilience4j.circuitbreaker.configs.shared.sliding-window-size=5
```

```
resilience4j.circuitbreaker.configs.shared.failure-rate-threshold=40
resilience4j.circuitbreaker.configs.shared.slow-call-rate-threshold=40
resilience4j.circuitbreaker.configs.shared.permitted-numb
resilience4j.circuitbreaker.configs.shared.max-wait-durat
resilience4j.circuitbreaker.configs.shared.wait-duration-
resilience4j.circuitbreaker.configs.shared.slow-call-dura
resilience4j.circuitbreaker.configs.shared.writable-stack
resilience4j.circuitbreaker.configs.shared.automatic-tran

resilience4j.circuitbreaker.instances.example.base-config
```

application.properties hosted with ❤ by GitHub

The above configuration will create a shared circuit breaker configuration. It can be used for any circuit breaker instance we want to create. Notice that we created an instance named **example**, which we use when we annotate **@CircuitBreaker** on the REST API.

Let's look at the following configurations:

- **Sliding-window-type**: we will use a **count-based** circuit breaker type. In this type, the circuit will trip or move to an open state based on each incoming request.
- **Sliding-window-size**: we will use this parameter to record the last N requests to make the circuit breaker trip or open. Here, we will record the last 5 requests.
- **Failure-rate-threshold**: it shows the percentage of the total **sliding-window-size** that fails and will cause the circuit breaker trips to open state. This means, with a configuration of 40%, 2 out of 5 failed requests will cause the circuit breaker trips to open state.
- **Slow-call-rate-threshold**: it shows the percentage of the total **sliding-window-size** that fails which will cause the circuit breaker trips to open state. From the configuration above, it can be seen that 2 out of 5 failed requests will cause the circuit breaker trips to open state.
- **Slow-call-duration-threshold**: is the time taken to indicate the received response exceeds this configuration time will be recorded as an error count.

For other configurations, please refer to the Resilience4J documentation.

Testing

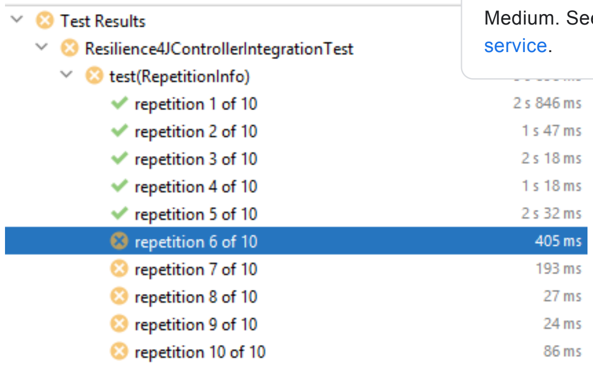
To simulate the circuit breaker above, I will use the Integration Test on the REST API that has been created.

```java
@SpringBootTest
@AutoConfigureWebTestClient
@ExtendWith(SpringExtension.class)
public class Resilience4JControllerIntegrationTest {
  @Autowired
  private WebTestClient webTestClient;

  @RepeatedTest(10)
  public void test(RepetitionInfo repetitionInfo) {
    int delay = 1 + (repetitionInfo.getCurrentRepetition() % 2);
    webTestClient.get()
        .uri("/api/delay/{delay}", delay)
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
        .exchange()
        .expectStatus()
        .isOk();
  }
}
```

CircuitBreakerControllerTest.java hosted with ❤ by GitHub view raw

Pay attention to the code. I am using **@RepeatedTest** annotation from Junit5. With this annotation, we can test with as many iterations as we want.
iterations to call the API that we created earlier.
Each iteration will be delayed for N seconds. When the itera
response will be delayed for 2s which will increase the failur
breaker. Whereas when the iteration is even then the respon
When the above test is run, it will produce the following out



```
Test Results
  Resilience4JControllerIntegrationTest
    test(RepetitionInfo)
      repetition 1 of 10                2 s 846 ms
      repetition 2 of 10                1 s 47 ms
      repetition 3 of 10                2 s 18 ms
      repetition 4 of 10                1 s 18 ms
      repetition 5 of 10                2 s 32 ms
      repetition 6 of 10                405 ms
      repetition 7 of 10                193 ms
      repetition 8 of 10                27 ms
      repetition 9 of 10                24 ms
      repetition 10 of 10               86 ms
```

Let's look at iterations 6, 7, through 10. *Failed right?* Why are that happened? Yaps, because the counter for circuit breaker trips to open state has been fulfilled (≥ 40% of the last 5 requests). This causes the next request to be considered a failure. If we look in more detail at the 6th iteration log we will find the following log:

```
2021-06-06 10:51:16.714 ERROR 29336 --- [     parallel-3] c.n.c.listener.Resilience4jLis
2021-06-06 10:51:16.942 ERROR 29336 --- [     parallel-3] a.w.r.e.AbstractErrorWebExcept

io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'example
        at io.github.resilience4j.circuitbreaker.CallNotPermittedException.createCallNot

2021-06-06 10:51:17.103 ERROR 29336 --- [           main] o.s.t.w.reactive.server.Exchar

> GET /api/delay/1
> WebTestClient-Request-Id: [6]
> Content-Type: [application/json]

No content

< 500 INTERNAL_SERVER_ERROR Internal Server Error
< Content-Type: [application/json]
< Content-Length: [135]

{"timestamp":"2021-06-06T03:51:16.864+00:00","path":"/api/delay/1","status":500,"error":
```

log_failed.log hosted with ❤ by GitHub view raw

Resilience4J will *fail-fast* by throwing a **CallNotPermittedException**, until the state changes to closed or according to the configuration we made. So how do we handle it when it's **Open State** but we don't want to throw an exception, but instead make it return a certain response? It's easy enough to add a fallback to the **@CircuitBreaker** annotation and create a function with the same name. Let's add the following line of code on the CircuitBreakerController file.

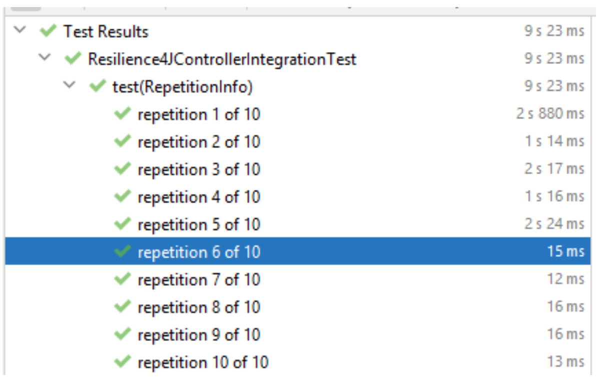
```
private static final String RESILIENCE4J_INSTANCE_NAME = "example";
private static final String FALLBACK_METHOD = "fallback";

@CircuitBreaker(name = RESILIENCE4J_INSTANCE_NAME, fallbackMethod = FALLBACK_METHOD)
public Mono<Response<Boolean>> delay(@PathVariable int delay) {
  return Mono.just(toOkResponse())
      .delayElement(Duration.ofSeconds(delay));
}
```

```
public Mono<Response<Boolean>> fallback(Exception ex) {
    return Mono.just(toResponse(HttpStatus.INTERNAL_SERVE
        .doOnNext(result -> log.warn("fallback executed")
}
```

fallback.java hosted with ❤ by GitHub

We will create a function with the name fallback, and registe annotation. So, when the circuit breaker trips to **Open state** **CallNotPermittedException** but instead will return the resp **INTERNAL_SERVER_ERROR**. We try to prove it by re-running the integration test that was previously made, and will get the following results:



```
✓ Test Results                                   9 s 23 ms
  ✓ Resilience4JControllerIntegrationTest        9 s 23 ms
    ✓ test(RepetitionInfo)                       9 s 23 ms
      ✓ repetition 1 of 10                       2 s 880 ms
      ✓ repetition 2 of 10                       1 s 14 ms
      ✓ repetition 3 of 10                       2 s 17 ms
      ✓ repetition 4 of 10                       1 s 16 ms
      ✓ repetition 5 of 10                       2 s 24 ms
      ✓ repetition 6 of 10                       15 ms
      ✓ repetition 7 of 10                       12 ms
      ✓ repetition 8 of 10                       16 ms
      ✓ repetition 9 of 10                       16 ms
      ✓ repetition 10 of 10                      13 ms
```

As we can see, all integration tests were executed successfully. That way the client from our application can handle when an **Open State** occurs, and will not waste their resources for requests that might be failed.

Actually, the **Resilience4J** library doesn't only have features for circuit breakers, but there are other features that are very useful when we create microservices, if you want to take a look please visit the Resilience4J Documentation.

The full source code for this article is available in my Github.

References:

- **Resilience4J**
- **JUnit5: Repeated Test**
- **WebTestClient**

Spring Boot Resilience4j Microservices Circuit Breaker Bliblidotcom