Resilience4j - Circuit Breaker

* A **circuit breaker** is a fault-tolerance mechanism designed to prevent repeated failures from overwhelming a system by temporarily blocking access to a failing component.
* **Circuit Breaker** is a design pattern used to prevent repeated failures in a system. It **monitors** service calls, and if failures exceed a threshold (e.g., too many timeouts or errors), it "opens" to block further calls for some time. This helps in **fault tolerance** and **prevents cascading failures** in microservices.
* **Resilience4j** is a Java library that provides fault tolerance mechanisms like **Circuit Breaker.**
* A **Circuit Breaker** in **Resilience4j** is like an automatic switch that prevents your application from making repeated calls to a failing service. Imagine a fuse in your house—if there's too much load, it trips to prevent damage. Similarly, if a service fails too often (like returning errors or taking too long), the circuit breaker "opens" and temporarily blocks requests, allowing the system to recover. Once things stabilize, it "closes" and allows requests again. This helps improve reliability and prevents cascading failures in microservices.
* So during failure cases, instead of repeatedly calling the external system, it will return a dummy response from our end.
* Run both Service A & B, and make successful API call.
* Bring service2 down, and test. Exception triggered. Check flow with breakpoints
* Add Pom dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>

</dependency>

<!-- Spring Boot Actuator (for monitoring) -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<!-- AOP for Resilience4j annotations-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

* This would have version dependency, BOM configuration is needed. Add below config parallel to <parent> in pom.xml
* **Spring Cloud BOM (Bill of Materials)** ensures that all Spring Cloud dependencies in your project use compatible versions, preventing conflicts. It is imported under <dependencyManagement> so you don’t need to specify versions for individual Spring Cloud dependencies. **Spring Cloud** is a framework that provides tools for building distributed systems and microservices in Spring Boot. It simplifies challenges like service discovery (Eureka), API gateways, circuit breakers (Resilience4j), etc
* <https://spring.io/projects/spring-cloud#overview>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>2024.0.0</version> <!-- Correct version for Spring Boot 3.4.x -->

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

## Use @CircuitBreaker & add fallback method

* Note that it tracks the exception being raised from the method. So if you handle it, it won't track. Ensure that when service\_unable cases, your method throws exception.
* THe method invocations should be with **AOP style**, so spring can apply this. Else it won't be triggered.
* When you use @CircuitBreaker (from Resilience4j), it internally applies **AOP** to intercept your method call. Before the method runs, Spring checks the circuit breaker state:
* **If closed (normal state)** → The method executes.
* **If open (too many failures)** → It skips execution and returns a fallback response.
* Spring does this using a **proxy-based mechanism** where it wraps your method with additional logic, which is how **AOP** works.

@CircuitBreaker(name = "payment-processing-service", fallbackMethod = "fallbackProcessPayment")

**public** ResponseEntity<String> makeHttpCall(HttpRequest httpRequest) {

–

}

**public** ResponseEntity<String> fallbackProcessPayment(HttpRequest httpRequest, Throwable t) {

// Handle fallback logic here

***log***.error("Fallback method invoked due to exception:" + t.getMessage());

**throw** **new** ProcessingException(

ErrorCodeEnum.***UNABLE\_TO\_CONNECT\_TO\_STRIPE\_PS***.getErrorCode(),

ErrorCodeEnum.***UNABLE\_TO\_CONNECT\_TO\_STRIPE\_PS***.getErrorMessage(),

HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

## Enable actuators

* # Enable Circuit Breaker metrics in Actuator
* management.endpoints.web.exposure.include=\*
* management.endpoint.health.enabled=true
* management.health.circuitbreakers.enabled=true
* management.endpoint.health.show-details=always

## Add Circuit Breaker Config

* # Resilience4j Circuit Breaker configurations
* resilience4j.circuitbreaker.instances.payment-processing-service.failureRateThreshold=40
* resilience4j.circuitbreaker.instances.payment-processing-service.minimumNumberOfCalls=5
* resilience4j.circuitbreaker.instances.payment-processing-service.waitDurationInOpenState=120s
* resilience4j.circuitbreaker.instances.payment-processing-service.register-health-indicator=true
* resilience4j.circuitbreaker.instances.payment-processing-service.automaticTransitionFromOpenToHalfOpenEnabled=true
* resilience4j.circuitbreaker.instances.payment-processing-service.permittedNumberOfCallsInHalfOpenState=2
* resilience4j.circuitbreaker.instances.payment-processing-service.slidingWindowSize=10
* resilience4j.circuitbreaker.instances.payment-processing-service.slidingWindowType=COUNT\_BASED

## You can monitor here:

* <http://localhost:8082/actuator/health>
* How to add JSON pretty to chrome. Setup.

## Test

* Make 5 consecutive failures.
  + Till 4, only the cout increases. For 5th, it calculates the failure rate, checks with threshold & moves to OPEN state.
* We are keeping in open state for 2 mins (120s)
* In OPEN state, for all request it calls fallback. It never calls the functional system. For entire 2mins, dummy handling at our end, instead of calling 3rd party.
* Observe that after 2mins, it moved to HALF\_OPEN, where it make functional call & calculates failure rate.
* In HALF\_OPEN, make both failed, check 100% failure rate & again OPEN state.
* Wait for 2mins. To get back to HALF\_OPEN. Now 1 success, 1 failure. Since 50% threshold, to again OPEN.
* Wait 2 mins. To get back to HALF\_OPEN. Now 2 success, check moved to CLOSED state. All back to normal.
* In CLOSED state, show past 10 count. Make 6 success, and then start making failure. And show failure-rate increasing. As it touches 40%, it moved to OPEN state. And same process.
* Test with breakpoints.

Here’s an explanation of each property:

* # Resilience4j Circuit Breaker configurations
* resilience4j.circuitbreaker.instances.payment-processing-service.failureRateThreshold=40
* resilience4j.circuitbreaker.instances.payment-processing-service.minimumNumberOfCalls=5
* resilience4j.circuitbreaker.instances.payment-processing-service.waitDurationInOpenState=60s
* resilience4j.circuitbreaker.instances.payment-processing-service.register-health-indicator=true
* resilience4j.circuitbreaker.instances.payment-processing-service.automaticTransitionFromOpenToHalfOpenEnabled=true
* resilience4j.circuitbreaker.instances.payment-processing-service.permittedNumberOfCallsInHalfOpenState=2
* resilience4j.circuitbreaker.instances.payment-processing-service.slidingWindowSize=10
* resilience4j.circuitbreaker.instances.payment-processing-service.slidingWindowType=COUNT\_BASED

### **How it Works Together**

1. Calls to the service are monitored.
2. When at least 5 calls have been made (minimumNumberOfCalls), the failure rate is calculated over the last 10 calls (slidingWindowSize).
3. If the failure rate exceeds 40% (failureRateThreshold), the circuit breaker opens for 60 seconds (waitDurationInOpenState).
4. After 60 seconds, the circuit breaker transitions to half-open (automaticTransitionFromOpenToHalfOpenEnabled) and permits 2 test calls (permittedNumberOfCallsInHalfOpenState).
5. If these test calls succeed, the circuit breaker closes, resuming normal operation. If they fail, the circuit breaker reopens.

### **1. failureRateThreshold=40**

* **Purpose**: Sets the failure rate threshold (in percentage) to trigger the circuit breaker into the **OPEN** state.
* **How it Works**:
  + When 40% of the requests in the defined sliding window fail, the circuit breaker will open.
  + Failures are typically defined by exceptions or responses that don't meet expectations.
* **Example**:
  + If the sliding window contains 10 calls, and 4 of them fail, the failure rate reaches 40%, and the circuit breaker opens.

### **2. minimumNumberOfCalls=5**

* **Purpose**: Ensures the circuit breaker only evaluates the failure rate after at least this many calls have been made.
* **How it Works**:
  + If fewer than 5 calls have been made in the sliding window, the failure rate is not calculated.
* **Example**:
  + In a window of 10 calls, if only 3 have been made, the circuit breaker does not evaluate or transition states.

### **3. waitDurationInOpenState=60s**

* **Purpose**: Defines the duration the circuit breaker remains in the **OPEN** state before transitioning to the **HALF-OPEN** state.
* **How it Works**:
  + During this period, no requests are forwarded to the service; instead, failures are returned immediately.
* **Example**:
  + If the circuit breaker opens at T=0, it will remain open until T=60s. Afterward, it will transition to half-open and allow a limited number of test calls.

### **4. register-health-indicator=true**

* **Purpose**: Registers the circuit breaker as a health indicator, making its state visible in tools like Actuator in Spring Boot.
* **How it Works**:
  + The state (CLOSED, OPEN, or HALF-OPEN) and metrics (failure rate, call count, etc.) are exposed to monitoring systems.

### **5. automaticTransitionFromOpenToHalfOpenEnabled=true**

* **Purpose**: Automatically transitions the circuit breaker from the **OPEN** state to the **HALF-OPEN** state after the waitDurationInOpenState elapses.
* **How it Works**:
  + This removes the need for manual intervention to transition the circuit breaker.
* **Example**:
  + After 60 seconds in the open state, the circuit breaker will allow test calls in the half-open state.

### **6. permittedNumberOfCallsInHalfOpenState=2**

* **Purpose**: Defines how many test calls are allowed during the **HALF-OPEN** state to evaluate if the service has recovered.
* **How it Works**:
  + During the half-open state, 2 test calls are permitted.
  + If these calls succeed, the circuit breaker transitions to **CLOSED**.
  + If either fails, it transitions back to **OPEN**.

### **7. slidingWindowSize=10**

* **Purpose**: Sets the size of the sliding window used to calculate the failure rate.
* **How it Works**:
  + The window keeps track of the last 10 calls to the service.
  + The failure rate is calculated based on the success and failure of these 10 calls.

### **8. slidingWindowType=COUNT\_BASED**

* **Purpose**: Determines the type of sliding window used.
* **How it Works**:
  + **COUNT\_BASED**: Tracks a fixed number of calls (e.g., the last 10 calls).
  + **TIME\_BASED**: Tracks all calls within a fixed time duration (e.g., last 1 minute).
  + Here, it uses a count-based window, focusing on the most recent 10 calls.

### **Purpose of Sliding Window**

The **sliding window** is used to evaluate the performance of the service over a recent set of calls or time duration. Its purpose is to provide a mechanism for:

* **Failure rate monitoring**: It ensures that decisions (e.g., opening the circuit breaker) are based on recent data rather than historical trends.
* **Dynamic adjustment**: If a service recovers or fails frequently, the window reflects these changes quickly.
* **Flexibility**: By allowing COUNT or TIME-based windows, it adapts to different use cases (e.g., low traffic vs. high traffic).