How to Implement and Test Circuit Breaker Pattern in Spring Boot Using Resilience4j

When developing an application, particularly one based on microservices, it’s common to encounter issues during real-time execution. These might include slow response times, network failures, REST call failures, or issues arising from a high volume of requests. To handle such potential faults, it’s essential to implement a fault tolerance mechanism within the application. Resilience4j is a library that can be used to achieve this.

**What is Fault Tolerance ?**

**Fault tolerance in microservices** refers to the system’s ability to continue functioning even when one or more of its components fail. Microservices architectures often consist of many small, independent services that communicate with each other over a network, making them more prone to failures. Ensuring fault tolerance helps maintain availability and reliability despite these failures.

Several patterns help achieve fault tolerance in microservices:

* **Circuit Breaker Pattern**: Protects services from repeated failure by “opening the circuit” when failures exceed a threshold, thus preventing further calls to the failing service for a while.
* **Retry Pattern**: Automatically retries failed requests in case the failure was transient.
* **Fallback Pattern**: Provides alternative methods or static data when a service fails or is unavailable.
* **Bulkhead Pattern**: Isolates different parts of the system into separate pools, preventing one service’s failure from affecting others.
* **Timeouts**: Limits the time a service waits for a response, preventing long delays from unresponsive services.

**Importance of Fault Tolerance:**

* **Minimizes Downtime**: Even when services fail, the system remains partially operational, minimizing the impact on users.
* **Improves User Experience**: Ensures the application continues to serve requests, even if some functionality is degraded.
* **Enhances Scalability**: With proper fault tolerance, services can handle dynamic changes in load and fail gracefully.

**Implanting circuit breaker using resilience4j in springboot**

**1. Use Case Scenario**

In this example, the **Student Service** fetches course details from the **Course Service**. The **Course Service** might become unavailable due to network issues or downtime. Using the **Circuit Breaker Pattern** ensures that if the **Course Service** fails too many times within a short period, the calls to that service will be automatically blocked (open circuit) for a while before allowing them again (half-open, then closed state).

**2. Project Setup**

Ensure you have the following dependencies in your pom.xml:

<dependencies>  
 <dependency>  
 <groupId>io.github.resilience4j</groupId>  
 <artifactId>resilience4j-spring-boot2</artifactId>  
 <version>1.7.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
</dependencies>

**3. Application Configuration**

application.yml

resilience4j.circuitbreaker:  
 instances:  
 courseService:  
 registerHealthIndicator: true  
 slidingWindowSize: 5  
 minimumNumberOfCalls: 3  
 failureRateThreshold: 50  
 waitDurationInOpenState: 10s  
 permittedNumberOfCallsInHalfOpenState: 3

**This configuration states:**

* The circuit breaker will be triggered after 3 failures.
* 50% failure rate will open the circuit.
* The circuit remains open for 10 seconds before going into the half-open state.

**1.**resilience4j.circuitbreaker.instances.courseService :

This part defines the **circuit breaker instance** for a specific service, in this case, courseService. Resilience4j allows you to create different circuit breakers for different services (or parts of your application).

**Property Explanations:**

**2.**registerHealthIndicator: true

* This property indicates whether the circuit breaker will be registered as a **health indicator**.
* When set to true, it allows the circuit breaker to expose its health status, which can be monitored through Spring Boot Actuator. This is useful for detecting whether the circuit is open, half-open, or closed.

**3.**slidingWindowSize: 5

* This defines the size of the **sliding window** for measuring the failure rate of requests. In this case, it tracks the **last 5 requests** to decide whether the failure rate has exceeded the threshold.
* The sliding window allows the circuit breaker to evaluate recent calls within this window size. If the failure rate within these calls exceeds the threshold, the circuit breaker will open.

**4.**minimumNumberOfCalls: 3

* This specifies the **minimum number of requests** that must be made before the circuit breaker starts evaluating the failure rate. In this case, it waits for at least 3 calls before deciding whether to open the circuit.
* Until this minimum number is met, the circuit breaker will not be triggered, even if there are failures.

**5.**failureRateThreshold: 50

* The **failure rate threshold** is set to 50%, meaning that if **50% or more** of the requests within the sliding window (last 5 requests) fail, the circuit breaker will **open**.
* For example, if 2 out of 5 calls fail, the failure rate is 40%, so the circuit breaker remains closed. But if 3 out of 5 fail, the failure rate is 60%, and the circuit will open.

**6.**waitDurationInOpenState: 10s

* This defines how long the circuit breaker will stay **open** before it transitions to the **half-open state**. In this case, the circuit remains open for **10 seconds**.
* When the circuit is open, all requests will automatically fail, and no further attempts will be made to call the failing service. After 10 seconds, it moves to the half-open state, where it tests the service again.

**7.**permittedNumberOfCallsInHalfOpenState: 3

* When the circuit is in the **half-open state**, this property defines how many calls are allowed to pass through to test if the service is back to normal. In this case, **3 calls** are allowed.
* If these 3 calls succeed, the circuit will close, and normal traffic will resume. If they fail, the circuit will open again for another 10 seconds.

**4. Service Layer Implementation**

**PlanService.java**

package com.example.resilience4jdemo.service;  
  
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;  
import org.springframework.stereotype.Service;  
import org.springframework.web.client.RestTemplate;  
import org.springframework.web.client.HttpServerErrorException;  
  
@Service  
public class PlanService {  
  
 private final RestTemplate restTemplate = new RestTemplate();  
  
 @CircuitBreaker(name = "planService", fallbackMethod = "fallbackPlanDetails")  
 public String getPlanDetails(String studentId) {  
 String url = "http://localhost:8081/course/" + studentId;  
 return restTemplate.getForObject(url, String.class);  
 }  
  
 public String fallbackPlanDetails(String studentId, Throwable t) {  
 return "Fallback: Planservice is currently unavailable for student " + studentId;  
 }  
}

**1. @CircuitBreaker Annotation**

* This annotation is used to mark a method for which the circuit breaker functionality should be applied.

**2. name Attribute**

* name = "planService"
* This specifies the **name** of the circuit breaker instance. In this case, the circuit breaker is named "planService".
* The name corresponds to the configuration defined in your application.yml or application.properties file. It helps to identify which circuit breaker configuration to apply.

**3. fallbackMethod Attribute**

* fallbackMethod = "fallbackPlanDetails"
* This specifies the name of the method to be called if the circuit breaker is open or if the annotated method fails.
* The fallback method should have the same parameters as the annotated method, with an additional Throwable parameter for capturing exceptions if needed.

**StudentService.java**

package com.example.resilience4jdemo.service;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class StudentService {  
  
 @Autowired  
 private PlanService planService;  
  
 public String getStudentCourseDetails(String studentId) {  
 return planService.getplanDetails(studentId);  
 }  
}

**5. Controller Layer**

**StudentController.java**

package com.example.resilience4jdemo.controller;  
  
import com.example.resilience4jdemo.service.StudentService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class StudentController {  
  
 @Autowired  
 private StudentService studentService;  
  
 @GetMapping("/student/{id}/course")  
 public String getStudentCourse(@PathVariable String id) {  
 return studentService.getStudentCourseDetails(id);  
 }  
}

.

+--------------------+ +--------------------+  
| | | |  
| Student Service |<------>+ Plan Service |  
| | | |  
+--------------------+ +--------------------+  
 |  
 |  
 v  
+---------------------+  
| Resilience4j |  
| Circuit Breaker |  
| (planService) |  
+---------------------+

If you want to create a dummy response for the URL http://localhost:8081/plan/{studentId}, here's how you can implement it in a Spring Boot service:

package com.example.courseservice.controller;  
  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class PlanController {  
  
 @GetMapping("/plan/{studentId}")  
 public String getPlanDetails(@PathVariable String studentId) {  
 // Dummy response  
 return "{ \"planName\": \"Computer Science\", \"studentId\": \"" + studentId + "\", \"duration\": \"4 years\" }";  
 }  
}

You can test the service using **Postman** or **curl** to ensure it returns the response:

* **Request URL:** <http://localhost:8081/plan/123>
* **Method:** GET
* **Expected Response:**

{  
 "planName": "Computer Science",  
 "studentId": "123",  
 "duration": "4 years"  
}

**6. Testing the Circuit Breaker**

To test both **success** and **failure** scenarios using **Postman**, follow the steps below:

**1. Start both services:**

* **Student Service** on port 8080.
* **Plan Service** on port 8081.

**2. Success Scenario (Plan Service is UP)**

**2.1 Request**

* **Method**: GET
* **URL**: <http://localhost:8080/student/1/course>
* **Headers**: None

**2.2 Expected Response**

When the **Plan Service** is running and the student ID exists, you’ll get a normal response from the **Plan Service**. The response might look like:

{  
 "planName": "Computer Science",  
 "duration": "4 years"  
}

This verifies that the **Student Service** successfully communicates with the **Plan Service**.

**3. Failure Scenario (Plan Service is DOWN)**

**3.1 Stop the Plan Service**

Stop the **Plan Service** running on port 8081.

**3.2 Request**

Make the same request as before:

* **Method**: GET
* **URL**: <http://localhost:8080/student/1/course>
* **Headers**: None

**3.3 Expected Response**

Once the **Plan Service** is down and the circuit breaker is triggered (after multiple failures based on your slidingWindowSize configuration), the fallback method will respond with:

{  
 "message": "Fallback: Plan service is currently unavailable for student 1"  
}

This demonstrates the **Circuit Breaker Pattern** in action. After a certain number of failures, the circuit breaker will open, and the fallback method will be called, protecting the system from further stress.

**4. Simulating Circuit Breaker States**

**Circuit Breaker states:**

1. **Closed**: Initially, all requests go through to the **Plan Service**.
2. **Open**: After the threshold is reached (e.g., 50% failure rate in the last 3 attempts), the circuit breaker opens, and the fallback is triggered immediately.
3. **Half-open**: After a waiting period (waitDurationInOpenState), the circuit breaker allows some requests to pass through to test if the service is back.

To observe this behavior:

* Stop the **Plan Service**, make several requests until the circuit breaker opens.
* Restart the **Plan Service** after the wait duration (e.g., 10 seconds), and observe the half-open state where some requests pass through to the **Plan Service** before the circuit returns to the closed state.

**Postman History and Results:**

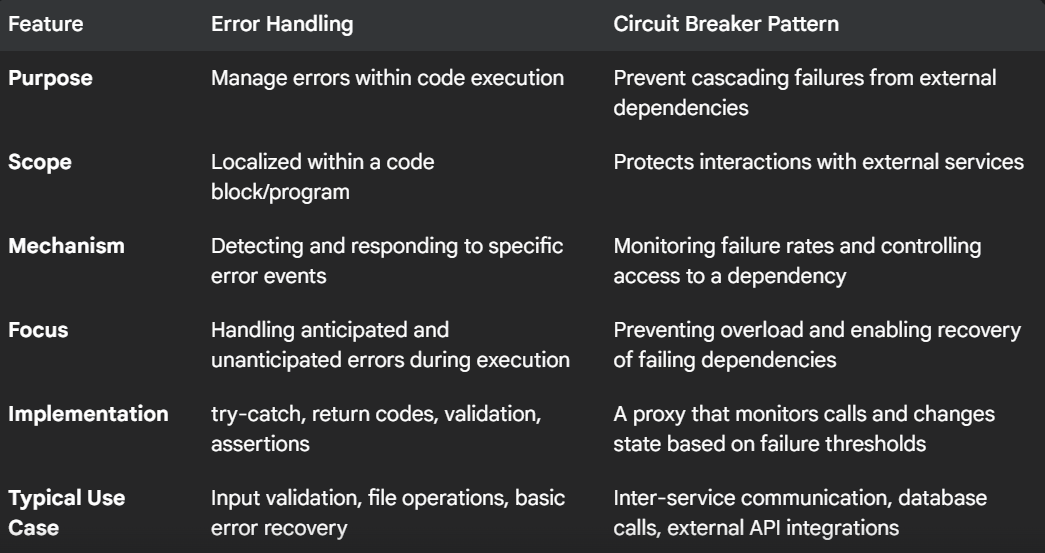
* **Success**: Normal response from the **Plan Service**.
* **Failure**: Fallback response with “Fallback: Course service is currently unavailable”.

This will show both the **success** and **failure** paths handled by the circuit breaker.

**Circuit Breakers vs. Error Handling**

**Circuit Breakers vs. Error Handling: Key Differences**

While both circuit breakers and error handling are mechanisms for dealing with issues in application, they address different concerns and operate at different levels. Error handling is a fundamental programming concept focused on managing expected and unexpected errors within a specific block of code or a program's execution flow. The circuit breaker pattern, on the other hand, is a design pattern primarily used in distributed systems to prevent cascading failures when interacting with potentially unreliable external services.



**Limitations of Relying Only on Error Handling for External Calls:**

* **Cascading Failures:** If a service you depend on becomes unhealthy (slow, erroring, or unavailable), your service, if it just keeps retrying or waiting with standard error handling, can become bogged down. Threads might be consumed waiting for responses, connection pools can be exhausted, and your service's performance and availability can degrade. This can then impact other services that depend on *your* service, leading to a domino effect of failures across the system. Standard error handling doesn't inherently prevent this.
* **Overwhelming the Failing Service:** Continuously sending requests to an already struggling service, even with retries, can exacerbate its problems and delay its recovery. It's like repeatedly calling someone who isn't answering the phone – you're just adding to their burden if they're already overwhelmed.
* **Slow User Experience:** If your service has to wait for a timeout on a call to a failing dependency before its error handling kicks in, the user experiences a significant delay. A circuit breaker can provide an immediate failure response, allowing for a faster fallback experience (e.g., showing cached data or a user-friendly error message).
* **Resource Exhaustion:** As mentioned, repeated failed calls can tie up valuable resources in the calling service, even if you have basic error handling to catch exceptions. This can lead to your service becoming unresponsive to healthy requests.
* **Difficulty in Detecting and Reacting to Persistent Failures:** While error handling can tell you *that* an error occurred, it doesn't easily provide a system-wide view of the *health* of a dependency. You might log individual errors, but determining if a service is experiencing a systemic issue based solely on dispersed error logs across many calling instances is challenging.

**How Circuit Breakers Address These Limitations:**

The circuit breaker pattern provides a dedicated mechanism to monitor the health of external dependencies and proactively stop making calls to services that are exhibiting high failure rates. Here's how it helps where basic error handling falls short:

* **Prevents Cascading Failures:** By "tripping" open when a dependency is failing, the circuit breaker immediately stops traffic to that service. This prevents the calling service from becoming overwhelmed and protects the rest of the system from a spreading failure.
* **Allows Failing Services to Recover:** By halting the influx of requests, the circuit breaker gives the unhealthy service a chance to stabilize and recover without being further bombarded.
* **Enables Fail-Fast Behavior:** When the circuit is open, calls fail instantly. This avoids long waits for timeouts and allows the calling service to quickly execute fallback logic, leading to a more responsive user experience.
* **Conserves Resources:** By not attempting calls to a known-failing service, the circuit breaker prevents the unnecessary consumption of resources like threads and connections in the calling service.
* **Provides a Centralized View of Dependency Health:** The state of the circuit breaker (closed, open, half-open) provides a clear indicator of the health of the downstream service, which can be monitored and used for operational insights.
* **Facilitates Graceful Degradation:** When a circuit is open, the calling service can implement fallback mechanisms (e.g., serving stale data from a cache, returning a default response) to provide a degraded but still functional experience to the user, rather than a complete failure.

**Benefits of using circuit breaker**

1. Prevents Cascading Failures:

* In a microservices architecture, services often depend on each other. If one service fails, it can cause a chain reaction, potentially bringing down other services and the entire system.
* A circuit breaker acts like an electrical circuit breaker, isolating the failing service and preventing the failure from spreading to other services.
* This prevents cascading failures and ensures that the overall system remains stable.

2. Improves Fault Tolerance and Resiliency:

* By isolating failures, circuit breakers make the microservices architecture more fault-tolerant and resilient.
* The system can continue to function even when some services are experiencing issues, improving the overall availability and reliability.

3. Enhances Responsiveness:

* Instead of waiting for a failing service to respond or timeout, the circuit breaker can quickly reject requests to that service.
* This reduces waiting times for clients and improves the overall responsiveness of the system.

4. Saves Resources and Improves Performance:

* By preventing requests to failing services, circuit breakers save resources that would otherwise be wasted on failed calls.
* This can improve the performance and efficiency of the system as a whole.

5. Enables Graceful Degradation:

* Circuit breakers can be configured to provide alternative responses or functionalities when failures occur.
* This allows the system to gracefully degrade to a lower level of functionality when necessary.

6. Facilitates Recovery:

* When the failing service recovers, the circuit breaker can automatically transition back to a normal state, allowing requests to flow again.
* This facilitates a smooth recovery process and minimizes downtime.

In essence, circuit breakers are a crucial design pattern for building resilient and fault-tolerant microservices, protecting them from the impact of failures in their dependencies and ensuring the overall stability and availability of the system.