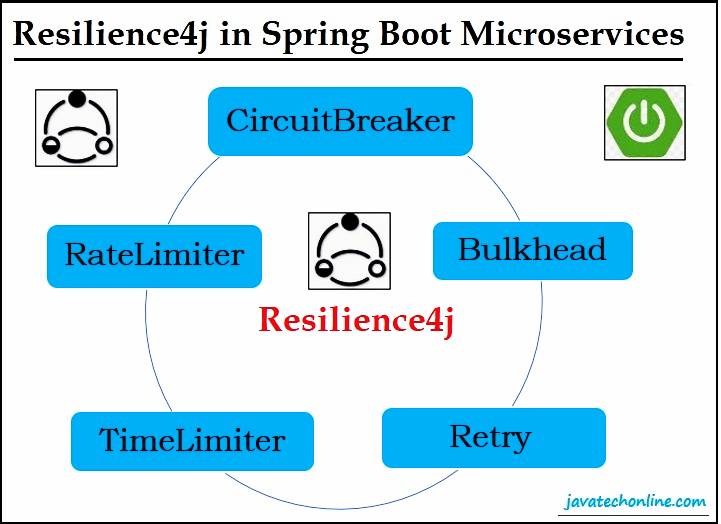
# How to implement Fault Tolerance in Microservices using Resilience4j?



When we develop an application, especially a Microservices-based applications, there are high chances that we experience some deviations while running it in real time. Sometimes, it could be slow response, network failures, REST call failures, failures due to the high number of requests and much more. In order to tolerate these kinds of suspected faults, we need to incorporate Fault Tolerance mechanism in our application. To achieve it, we will make use of Resilience4j library.

Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by  [Netflix Hystrix](https://github.com/Netflix/Hystrix" \o "" \t "), but designed for Java 8 and functional programming. So, our focus in this article will be on ‘How to implement Fault Tolerance in Microservices using Resilience4j?’

After implementing the Fault Tolerance in Microservices using Resilience4j, we make sure that the entire system will not be down if a service (a database, API Server, REST call) fails or goes down. Let’s discuss our topic ‘How to implement Fault Tolerance in Microservices using Resilience4j?’, and its related concepts. In case, you want to learn the implementation of Hystrix, kindly visit our article on ‘[How To Implement Hystrix Circuit Breaker In Microservices Application?](https://javatechonline.com/how-to-implement-hystrix-circuit-breaker-in-microservices-application/)‘.

## What is Fault Tolerance in Microservices?

In a context of Microservices, Fault Tolerance is a technique of tolerating a fault. A Microservice that tolerates the fault is known as Fault Tolerant. Moreover, a Microservice should be a fault tolerant in such a way that the entire application runs smoothly. In order to implement this technique, the Resilience4j offers us a variety of modules based on the type of fault we want to tolerate.

## Core modules of Resilience4j

* resilience4j-circuitbreaker: Circuit breaking
* resilience4j-ratelimiter: Rate limiting
* resilience4j-bulkhead: Bulkheading
* resilience4j-retry: Automatic retrying (sync and async)
* resilience4j-cache: Result caching
* resilience4j-timelimiter: Timeout handling

## Common Setup for How to implement Fault Tolerance in Microservices using Resilience4j?

We will have a common setup for all examples before directly going into ‘How to implement Fault Tolerance in Microservices using Resilience4j?’. It will include creating a Spring Boot Project using STS and adding all dependencies that are required to implement Resilience4j in our project.

### Create a Spring Boot Project including all dependencies using STS

While creating a project in STS, add starter dependencies: ‘Resilience4j’, ‘Spring Boot Actuator’, ‘Spring Web’ and ‘Spring Boot AOP’. You can also add ‘Spring Boot DevTools’ optionally. Since STS doesn’t provide the ‘Spring Boot AOP’ as a starter project, we need to add below dependency in pom.xml. If you are new to Spring Boot, visit our [Internal Link](https://javatechonline.com/saving-data-into-database-using-spring-boot-data-jpa-step-by-step-tutorial/#Step_1_Creating_Starter_Project_using_STS) to create a sample project in spring boot.

**<dependency>**

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

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

**</dependency>**

## What is Rate Limiting?

Rate Limiter limits the number of requests for a given period. Let’s assume that we want to limit the number of requests on a Rest API and fix it for a particular duration. There are various reasons to limit the number of requests that an API can handle, such as protect the resources from spammers, minimize the overhead, meet a service level agreement and many others. Undoubtedly, we can achieve this functionality with the help of annotation **@RateLimiter** provided by Resilience4j without writing a code explicitly.

## How to implement Rate Limiting? : Rate Limiting Example

For example, we want to restrict only 2 requests per 5 seconds duration. In order to achieve this, let’s follow below steps to write code and respective configurations.

### Step#1: Common Setup for All Examples

Make sure you have completed steps mentioned in ‘Common Setup for All Examples’ section of this article.

### Step#2: Create a RestController class to implement the RateLimiter functionality

Here, in this example, we will create a RestController with a simple method that will demonstrate our functionality. Additionally, we will create a fallback method to tolerate the fault.

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.http.HttpStatus;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RequestParam;**

**import org.springframework.web.bind.annotation.RestController;**

**import io.github.resilience4j.ratelimiter.RequestNotPermitted;**

**import io.github.resilience4j.ratelimiter.annotation.RateLimiter;**

**@RestController**

**public class RateLimitController {**

**Logger logger = LoggerFactory.getLogger(RateLimitController.class);**

**@GetMapping("/getMessage")**

**@RateLimiter(name = "getMessageRateLimit", fallbackMethod = "getMessageFallBack")**

**public ResponseEntity<String> getMessage(@RequestParam(value="name", defaultValue = "Hello") String name){**

**return ResponseEntity.ok().body("Message from getMessage() :" +name);**

**}**

**public ResponseEntity<String> getMessageFallBack(RequestNotPermitted exception) {**

**logger.info("Rate limit has applied, So no further calls are getting accepted");**

**return ResponseEntity.status(HttpStatus.TOO\_MANY\_REQUESTS)**

**.body("Too many requests : No further request will be accepted. Please try after sometime");**

**}**

**}**

### Step#3: Update application.properties

**resilience4j.ratelimiter.instances.getMessageRateLimit.limit-for-period=2**

**resilience4j.ratelimiter.instances.getMessageRateLimit.limit-refresh-period=5s**

**resilience4j.ratelimiter.instances.getMessageRateLimit.timeout-duration=0**

The above properties represent that only 2 requests are allowed in 5 seconds duration. Also, there is no timeout duration which means after completion of 5 seconds, the user can send request again.

### Step#4: How to test the implemented RateLimiter?

1) Open the Browser and hit the URL : http://localhost:8080/getMessage

2) You should see the result “Message from getMessage() :Hello” on the browser.

3) Now let’s refresh the browser more than 2 times within 5 seconds period.

4) Once you refresh third time within 5 seconds, you should see the message “Too many requests : No further request will be accepted. Please try after sometime”

5) In console also you should see the logger message as ‘Rate limit has applied, So no further calls are getting accepted’

6) Now update limit-for-period=10 and limit-refresh-period=1s in application.xml. Then, After refreshing the browser multiple times you should see only success message as “Message from getMessage() :Hello” in the browser.

## What is Retry?

Suppose Microservice ‘A’  depends on another Microservice ‘B’. Let’s assume Microservice ‘B’ is a faulty service and its success rate is only upto 50-60%. However, fault may be due to any reason, such as service is unavailable, buggy service that sometimes responds and sometimes not, or an intermittent network failure etc. However, in this case, if Microservice ‘A’ retries to send request 2 to 3 times, the chances of getting response increases. Obviously, we can achieve this functionality with the help of annotation **@Retry** provided by Resilience4j without writing a code explicitly.

Here, we have to implement a Retry mechanism in Microservice ‘A’. We will call Microservice ‘A’ as Fault Tolerant as it is participating in tolerating the fault. However, Retry will take place only on a failure not on a success. By default retry happens 3 times. Moreover, we can configure how many times to retry as per our requirement.

## How to implement Retry? :Retry Example

We will develop a scenario where one Microservice will call another Microservice.

### Step#1: Common Setup for All Examples

Make sure you have completed steps mentioned in ‘Common Setup for All Examples’ section of this article.

### Step#2: Create a RestController class to implement the Retry functionality

In order to achieve the Retry functionality, in this example, we will create a RestController with a method that will call another Microservice which is down temporarily. Additionally, we will create a fallback method to tolerate the fault.

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import org.springframework.web.client.RestTemplate;**

**import io.github.resilience4j.retry.annotation.Retry;**

**@RestController**

**public class RetryController {**

**Logger logger = LoggerFactory.getLogger(RetryController.class);**

**RestTemplate restTemplate=** **new RestTemplate();**

**@GetMapping("/getInvoice")**

**@Retry(name = "getInvoiceRetry", fallbackMethod = "getInvoiceFallback")**

**public String getInvoice() {**

**logger.info("getInvoice() call starts here");**

**ResponseEntity<String> entity= restTemplate.getForEntity("http://localhost:8080/invoice/rest/find/2", String.class);**

**logger.info("Response :" + entity.getStatusCode());**

**return entity.getBody();**

**}**

**public String getInvoiceFallback(Exception e) {**

**logger.info("---RESPONSE FROM FALLBACK METHOD---");**

**return "SERVICE IS DOWN, PLEASE TRY AFTER SOMETIME !!!";**

**}**

**}**

### Step#3: Update application.properties

**resilience4j.retry.instances.getInvoiceRetry.max-attempts=5**

**resilience4j.retry.instances.getInvoiceRetry.wait-duration=2s**

**resilience4j.retry.instances.getInvoiceRetry.retry-exceptions=org.springframework.web.client.ResourceAccessException**

As aforementioned, By default the retry mechanism makes 3 attempts if the service fails for the first time. But here we have configured for 5 attempts, each after 2 seconds interval. Additionally, if business requires it to retry only if a specific exception occurs, that can also be configured as above. If we want Resilience4j to retry when any type of exception occurs, we don’t need to mention the property ‘retry-exceptions’.

### Step#4: How to test the implemented Retry?

1) Make the called Microservice down.

2) Open the browser and hit the URL : http://localhost:8080/getInvoice

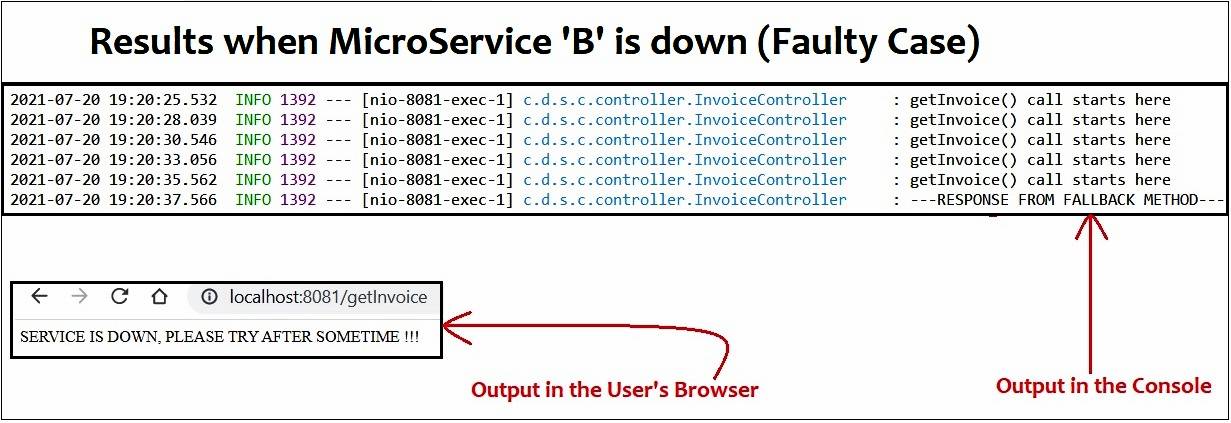
3) You should see “getInvoice() call starts here” message 5 times in the console. It means it has tried 5 attempts.

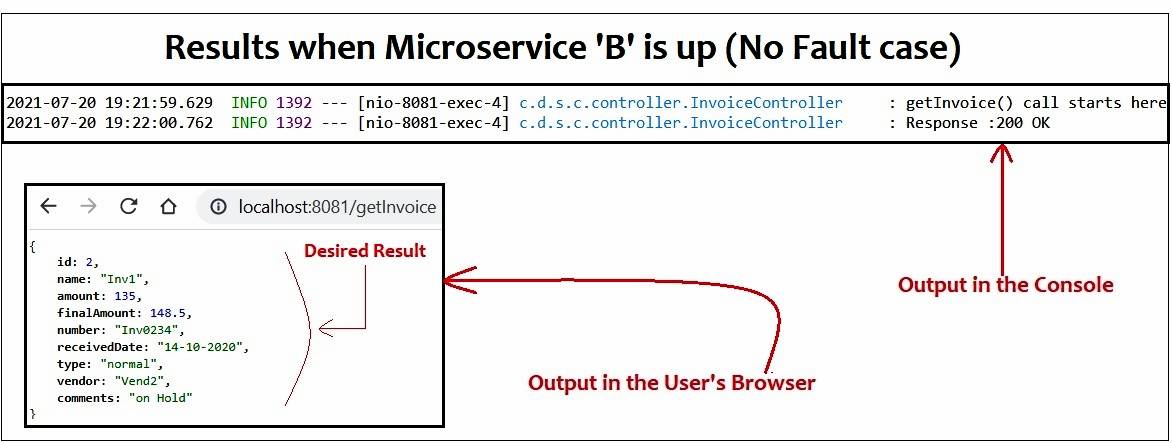
4) Once 5 attempts completes, you should see the message “—RESPONSE FROM FALLBACK METHOD—” in the console. It indicates that the fallback method called.

5) Subsequently, You will see the “SERVICE IS DOWN, PLEASE TRY AFTER SOMETIME !!!” message in the browser. It indicates that a common message is getting shown to the user.

6) Now let’s make the called Microservice up. Hit the URl again to see the desired results.

7) If you are getting the desired results successfully, neither Microservice should attempt any retry nor fallback method should be called.





## What is Circuit Breaker ?

Circuit Breaker is a pattern in developing the Microservices based applications in order to tolerate any fault. As the name suggests, ‘Breaking the Circuit’. Suppose a Microservice ‘A’ is internally calling another Microservice ‘B’ and ‘B’ has some fault. Needless to say, in Microservice Architecture ‘A’ might be dependent on other Microservices and the same is true for Microservice ‘B’.

In order to escape the multiple microservices from becoming erroneous as a result of cascading effect, we stop calling the faulty Microservice ‘B’. Instead, we call a dummy method that is called a ‘Fallback Method’. Therefore, calling a fallback method instead of an actual service due to a fault is called breaking the circuit. That’s why, we call this as a ‘Circuit Breaker’ Pattern. Moreover, there are generally three states of a Circuit Breaker Pattern : Closed, Open, Half Open.

### Closed

When a Microservice calls the dependent Microservice continuously, then we call the Circuit is in Closed State.

### Open

When a MicroService doesn’t call the dependent Microservice, Instead, it calls the fallback method that is implemented to tolerate the fault. We call this state as Open State. When a certain percentage of requests get failed, let’s say 90%, then we change the state from Closed to Open.

### Half-open

When a Microservice sends a percentage of requests to dependent Microservice and the rest of them to Fallback method. We call this state as Half-open. During the open state, we can configure the wait duration. Once wait duration is over, the Circuit Breaker will come in Half-open state. In this state Circuit Breaker checks if the dependent service is up. In order to achieve this, it sends a certain percentage of requests to dependent service that we can configure. If it gets a positive response from dependent service, it would switch to the closed state, otherwise it would again go back to the Open State.

## When to use Circuit Breaker?

For example, if a Microservice ‘A’ depends upon Microservice ‘B’. For some reason, Microservice ‘B’ is experiencing an error. Instead of repeatedly calling Microservice ‘B’, the Microservice ‘A’ should take a break (not calling) until Microservice ‘B’ is completely or partially recovered. Using **Circuit Breaker** we can eliminate the flow of failures to downstream/upstream. We can achieve this functionality easily with the help of annotation **@CircuitBreaker** without writing a specific code.

## How to implement Circuit Breaker ? : Circuit Breaker Example

We will develop a scenario where one Microservice will call another Microservice.

### Step#1: Common Setup for All Examples

Make sure you have completed steps mentioned in ‘Common Setup for All Examples’ section of this article.

### Step#2: Create a RestController class to implement the Circuit Breaker functionality

In order to achieve the Circuit Breaker functionality, in this example, we will create a RestController with a method that will call another Microservice which is down temporarily. Additionally, we will create a fallback method to tolerate the fault.

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import org.springframework.web.client.RestTemplate;**

**import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;**

**@RestController**

**public class CircuitBreakerController {**

**Logger logger = LoggerFactory.getLogger(CircuitBreakerController.class);**

**RestTemplate restTemplate = new RestTemplate();**

**@GetMapping("/getInvoice")**

**@CircuitBreaker(name = "getInvoiceCB", fallbackMethod = "getInvoiceFallback")**

**public String getInvoice() {**

**logger.info("getInvoice() call starts here");**

**ResponseEntity<String> entity= restTemplate.getForEntity("http://localhost:8080/invoice/rest/find/2", String.class);**

**logger.info("Response :" + entity.getStatusCode());**

**return entity.getBody();**

**}**

**public String getInvoiceFallback(Exception e) {**

**logger.info("---RESPONSE FROM FALLBACK METHOD---");**

**return "SERVICE IS DOWN, PLEASE TRY AFTER SOMETIME !!!";**

**}**

**}**

### Step#3: Update application.properties

**resilience4j.circuitbreaker.instances.getInvoiceCB.failure-rate-threshold=80**

**resilience4j.circuitbreaker.instances.getInvoiceCB.sliding-window-size=10**

**resilience4j.circuitbreaker.instances.getInvoiceCB.sliding-window-type=COUNT\_BASED**

**resilience4j.circuitbreaker.instances.getInvoiceCB.minimum-number-of-calls=5**

**resilience4j.circuitbreaker.instances.getInvoiceCB.automatic-transition-from-open-to-half-open-enabled=true**

**resilience4j.circuitbreaker.instances.getInvoiceCB.permitted-number-of-calls-in-half-open-state=4**

**resilience4j.circuitbreaker.instances.getInvoiceCB.wait-duration-in-open-state=1s**

1) ‘**failure-rate-threshold=80**‘ indicates that if 80% of requests are getting failed, open the circuit ie. Make the Circuit Breaker state as Open.

2) ‘**sliding-window-size=10**‘ indicates that if 80% of requests out of 10 (it means 8) are failing, open the circuit.

3) ‘**sliding-window-type=COUNT\_BASED**‘ indicates that we are using COUNT\_BASED sliding window. Another type is TIME\_BASED.

4) ‘**minimum-number-of-calls=5**‘ indicates that we need at least 5 calls to calculate the failure rate threshold.

5) ‘**automatic-transition-from-open-to-half-open-enabled=true**‘ indicates that don’t switch directly from the open state to the closed state, consider the half-open state also.

6) ‘**permitted-number-of-calls-in-half-open-state=4**‘ indicates that when on half-open state, consider sending 4 requests. If 80% of them are failing, switch circuit breaker to open state.

7) **‘wait-duration-in-open-state=1s’** indicates the waiting time interval while switching from the open state to the closed state.

These attributes are the important part of an implementation of a Circuit Breaker. We can configure the values as per our requirement and test the implemented functionality accordingly.

## What is Bulkhead?

In the context of the Fault Tolerance mechanism, if we want to limit the number of concurrent requests, we can use Bulkhead as an aspect. Using Bulkhead, we can limit the number of concurrent requests within a particular period. Please note the difference between Bulkhead and Rate Limiting. Rate Limiter never talks about concurrent requests, but Bulkhead does. Rate Limiter talks about limiting number of requests within a particular period. Hence, using **Bulkhead** we can limit the number of concurrent requests. We can achieve this functionality easily with the help of annotation **@Bulkhead** without writing a specific code.

## How to implement Bulkhead ? : Bulkhead Example

For example, we want to limit only 5 concurrent requests. In order to achieve this, let’s follow below steps to write code and respective configurations.

### Step#1: Common Setup for All Examples

Make sure you have completed steps mentioned in ‘Common Setup for All Examples’ section of this article.

### Step#2: Create a RestController class to implement the Bulkhead functionality

Here, in this example, we will create a RestController with a simple method that will demonstrate our functionality. Additionally, we will create a fallback method to tolerate the fault.

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.http.HttpStatus;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RequestParam;**

**import org.springframework.web.bind.annotation.RestController;**

**import io.github.resilience4j.ratelimiter.RequestNotPermitted;**

**import io.github.resilience4j.bulkhead.annotation.Bulkhead;**

**@RestController**

**public class BulkheadController {**

**Logger logger = LoggerFactory.getLogger(BulkheadController.class);**

**@GetMapping("/getMessage")**

**@Bulkhead(name = "getMessageBH", fallbackMethod = "getMessageFallBack")**

**public ResponseEntity<String> getMessage(@RequestParam(value="name", defaultValue = "Hello") String name){**

**return ResponseEntity.ok().body("Message from getMessage() :" +name);**

**}**

**public ResponseEntity<String> getMessageFallBack(RequestNotPermitted exception) {**

**logger.info("Bulkhead has applied, So no further calls are getting accepted");**

**return ResponseEntity.status(HttpStatus.TOO\_MANY\_REQUESTS)**

**.body("Too many requests : No further request will be accepted. Plese try after sometime");**

**}**

**}**

### Step#3: Update application.properties

**resilience4j.bulkhead.instances.getMessageBH.max-concurrent-calls=5**

**resilience4j.bulkhead.instances.getMessageBH.max-wait-duration=0**

‘max-concurrent-calls=5’ indicates that if the number of concurrent calls exceed 5, activate the fallback method.

‘max-wait-duration=0’ indicates that don’t wait for anything, show response immediately based on the configuration.

## What is Time Limiting or Timeout Handling?

Time Limiting is the process of setting a time limit for a Microservice to respond. Suppose Microservice ‘A’ sends a request to Microservice ‘B’, it sets a time limit for the Microservice ‘B’ to respond. If  Microservice ‘B’ doesn’t respond within that time limit, then it will be considered that it has some fault. We can achieve this functionality easily with the help of annotation **@Timelimiter** without writing a specific code.

## How to implement TimeLimiter ? :TimeLimiter Example

For example, we want to limit the duration of getting the response of a request. In order to achieve this, let’s follow below steps to write code and respective configurations.

### Step#1: Common Setup for All Examples

Make sure you have completed steps mentioned in ‘Common Setup for All Examples’ section of this article.

### Step#2: Create a RestController class to implement the TimeLimiter functionality

One point is to be noted here that the method which is annotated with @TimeLimiter should return a type CompletableFuture mandatorily.

**import java.util.concurrent.CompletableFuture;**

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import io.github.resilience4j.timelimiter.annotation.TimeLimiter;**

**@RestController**

**public class TimeLimiterController {**

**Logger logger = LoggerFactory.getLogger(TimeLimiterController.class);**

**@GetMapping("/getMessageTL")**

**@TimeLimiter(name = "getMessageTL")**

**public CompletableFuture<String> getMessage() {**

**return CompletableFuture.supplyAsync(this::getResponse);**

**}**

**private String getResponse() {**

**if (Math.random() < 0.4) { //Expected to fail 40% of the time**

**return "Executing Within the time Limit...";**

**} else {**

**try {**

**logger.info("Getting Delayed Execution");**

**Thread.sleep(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**return "Exception due to Request Timeout.";**

**}**

**}**

### Step#3: Update application.properties

**resilience4j.timelimiter.instances.getMessageTL.timeout-duration=1ms**

**resilience4j.timelimiter.instances.getMessageTL.cancel-running-future=false**

‘timeout-duration=1ms’ indicates that the maximum amount of time a request can take to respond is 1 millisecond

‘cancel-running-future=false’ indicates that do not cancel the Running Completable Futures After TimeOut.

In order to test the functionality, Run the application as it is. You will get TimeOutException on the Browser. When you change the value of timeout-duration=1s, you will receive “Executing Within the time Limit…” message in the browser.

## Complete YAML file for all examples

Here is the combined application.yml file, including all examples in this article.

**resilience4j:**

**bulkhead:**

**instances:**

**getMessageBH:**

**max-concurrent-calls: 5**

**max-wait-duration: 0**

**circuitbreaker:**

**instances:**

**GetInvoiceCB:**

**automatic-transition-from-open-to-half-open-enabled: true**

**failure-rate-threshold: 80**

**minimum-number-of-calls: 5**

**permitted-number-of-calls-in-half-open-state: 4**

**sliding-window-size: 10**

**sliding-window-type: COUNT\_BASED**

**wait-duration-in-open-state: 10s**

**ratelimiter:**

**instances:**

**getMessageRateLimit:**

**limit-for-period: 2**

**limit-refresh-period: 10s**

**timeout-duration: 0**

**retry:**

**instances:**

**getInvoiceRetry:**

**enable-exponential-backoff: true**

**max-attempts: 5**

**retry-exceptions: org.springframework.web.client.ResourceAccessException**

**wait-duration: 2s**

**timelimiter:**

**instances:**

**getMessageTL:**

**cancel-running-future: false**

**timeout-duration: 1s**

## How to implement multiple Aspects/patterns in a single method?

If we are learning ‘How to implement Fault Tolerance in Microservices using Resilience4j?’, it becomes crucial to know how to apply multiple aspects/patterns in a single service. Yes, undoubtedly we can apply multiple aspects in a single method using separate annotations for each. The important point here is the order of their execution. Generally, we follow the order as given below, which is the default order specified by Resilience4J:

1) Bulkhead  
2) Time Limiter.  
3) Rate Limiter.  
4) Circuit Breaker  
5) Retry

Moreover, the application.properties file will look like below for the ordering part:

**resilience4j.bulkhead.bulkheadAspectOrder =1**

**resilience4j.timelimiter.timeLimiterAspectOrder =2**

**resilience4j.ratelimiter.rateLimiterAspectOrder =3**

**resilience4j.circuitbreaker.circuitBreakerAspectOrder =4**

**resilience4j.retry.retryAspectOrder =5**

## Conclusion

After going through all the theoretical & example part of ‘How to implement Fault Tolerance in Microservices using Resilience4j?’, finally, we should be able to integrate Resilience4j with the Microservices. Similarly, we expect from you to further extend these examples and implement them in your project accordingly. I hope you might be convinced by the article ‘How to implement Fault Tolerance in Microservices using Resilience4j?’. In addition, If there is any update in the future, we will also update the article accordingly. Moreover, Feel free to provide your comments in the comments section below.

Ah! you want a **direct comparison between Hystrix and Spring Cloud Resilience4j**. Let’s break it down clearly so you can understand why most modern Spring projects use Resilience4j instead of Hystrix.

**1️⃣ Project Status**

| **Feature** | **Hystrix** | **Resilience4j (Spring Cloud)** |
| --- | --- | --- |
| Maintenance | Deprecated / Maintenance | Actively maintained |
| Updates | No major updates | Yes, active |
| Spring Boot / Cloud Support | Legacy (Spring Cloud Netflix) | Native Spring Cloud Circuit Breaker support |
| Popularity | Old projects | Modern microservices |

**2️⃣ Architecture / Design**

| **Feature** | **Hystrix** | **Resilience4j** |
| --- | --- | --- |
| Circuit Breaker | Yes | Yes |
| Timeout / Isolation | Thread or semaphore based | Thread/semaphore optional |
| Retry / Bulkhead | Limited (thread pool isolation) | Built-in modules: retry, bulkhead, rate limiter |
| Fallback | Yes (fallbackMethod annotation) | Yes (fallbackMethod or lambda) |
| Functional style | Imperative / annotations | Functional & annotation support |
| Metrics / Monitoring | Hystrix Dashboard / Turbine | Micrometer / Prometheus |

**3️⃣ Configuration and Usage**

**Hystrix Example**

@HystrixCommand(fallbackMethod = "fallback")

public BigDecimal convertCurrency(...) {

// call currency exchange service

}

public BigDecimal fallback(...) {

return BigDecimal.ZERO;

}

* Requires **Netflix OSS dependencies** (Hystrix, Ribbon, Feign, Turbine).
* Uses **thread isolation** by default.

**Resilience4j Example**

@CircuitBreaker(name = "currencyService", fallbackMethod = "fallback")

public BigDecimal convertCurrency(...) {

// call currency exchange service

}

public BigDecimal fallback(Exception ex) {

return BigDecimal.ZERO;

}

* Lightweight, modular, purely annotation-driven.
* Supports **retry, bulkhead, rate limiter, time limiter** out of the box.
* Integrates with **Spring Cloud LoadBalancer** instead of Ribbon.
* Metrics are exposed via **Micrometer / Prometheus**.

**4️⃣ Advantages of Resilience4j over Hystrix**

1. **Actively maintained** → Hystrix is no longer updated.
2. **Modular** → pick only what you need (circuit breaker, retry, bulkhead…).
3. **Functional programming style** → more modern, works well with Java 8+ lambdas.
4. **Better metrics integration** → Prometheus, Grafana, Micrometer.
5. **No legacy Netflix OSS dependencies** → lighter microservice footprint.

**5️⃣ Recommendation**

* **New projects** → Use **Resilience4j + Spring Cloud Circuit Breaker**.
* **Legacy projects using Hystrix** → Only maintain if migration is difficult, otherwise migrate to Resilience4j.

If you want, I can give a **ready-to-use example of your currency-conversion-service using Resilience4j** to replace old Hystrix + Ribbon setup, including:

* Circuit breaker
* Fallback
* Load-balanced service call

**1️⃣ Hystrix Example**

@HystrixCommand(

fallbackMethod = "doDummyPay",

commandProperties = {

@HystrixProperty(name= "circuitBreaker.requestVolumeThreshold", value="6"),

@HystrixProperty(name= "circuitBreaker.sleepWindowInMilliseconds", value="10000"),

@HystrixProperty(name= "circuitBreaker.enabled", value = "false")

}

)

public String doPayment() {

// call external payment service

}

**2️⃣ Resilience4j Equivalent**

Resilience4j uses **@CircuitBreaker annotation** and **application.yml properties** for configuration.

**Java code**

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

import org.springframework.stereotype.Service;

@Service

public class PaymentService {

@CircuitBreaker(name = "paymentService", fallbackMethod = "doDummyPay")

public String doPayment() {

// call external payment service

return "Payment Done";

}

// fallback method must have same params + Throwable at the end

public String doDummyPay(Throwable ex) {

return "Dummy Payment (Fallback)";

}

}

**3️⃣ Circuit Breaker Configuration in application.yml**

resilience4j:

circuitbreaker:

instances:

paymentService:

registerHealthIndicator: true

slidingWindowType: COUNT\_BASED

slidingWindowSize: 6 # same as requestVolumeThreshold

minimumNumberOfCalls: 6 # min calls before evaluating failure rate

failureRateThreshold: 50 # % of failures to open CB

waitDurationInOpenState: 10s # same as sleepWindowInMilliseconds

permittedNumberOfCallsInHalfOpenState: 3

automaticTransitionFromOpenToHalfOpenEnabled: true

enable: false # equivalent to circuitBreaker.enabled=false

**1️⃣ registerHealthIndicator: true**

* **Description:** Registers a Spring Boot **health indicator** for this circuit breaker.
* **Effect:** You can see the CB status in /actuator/health.
* **Type:** Boolean (true/false)
* **Example:** true → /actuator/health will show "circuitBreakers": {"currencyExchangeService": "CLOSED"}

**2️⃣ slidingWindowType: COUNT\_BASED**

* **Description:** Determines **how the circuit breaker evaluates failures**.
* **Options:**
  + COUNT\_BASED → failure rate is calculated over a **fixed number of calls**.
  + TIME\_BASED → failure rate is calculated over a **fixed time window**.
* **Effect:** COUNT\_BASED is useful if traffic volume is predictable; TIME\_BASED is better for uneven traffic.

**3️⃣ slidingWindowSize: 6**

* **Description:** The **size of the sliding window** (number of calls or time intervals, depending on slidingWindowType).
* **COUNT\_BASED:** Number of recent calls considered to calculate failure rate.
* **TIME\_BASED:** Number of time buckets in the window.
* **Example:** 6 → CB evaluates failure rate over the last 6 calls.

**4️⃣ minimumNumberOfCalls: 6**

* **Description:** **Minimum number of calls required** before the CB can calculate failure rate and decide to open.
* **Effect:** CB ignores failures until this number of calls is reached.
* **Example:** If minimumNumberOfCalls = 6, and 2 of the first 5 calls fail, CB **won’t open** yet.

**5️⃣ failureRateThreshold: 50**

* **Description:** **Threshold (%) of failures** to open the circuit breaker.
* **Effect:** If failure rate ≥ 50%, CB moves from **CLOSED → OPEN**.
* **Example:** If last 6 calls (sliding window) have 3 or more failures, CB opens.

**6️⃣ waitDurationInOpenState: 10s**

* **Description:** How long the CB stays in the **OPEN state** before trying a request in **HALF\_OPEN state**.
* **Effect:** Prevents calling a failing service continuously.
* **Example:** 10s → after 10 seconds, CB allows a few test requests in HALF\_OPEN state.

**7️⃣ permittedNumberOfCallsInHalfOpenState: 3**

* **Description:** Number of **trial calls allowed** when CB is in **HALF\_OPEN state**.
* **Effect:** If these trial calls succeed, CB **closes**; if they fail, CB goes **OPEN** again.
* **Example:** 3 → CB will allow 3 requests to test if the service is healthy.

**8️⃣ automaticTransitionFromOpenToHalfOpenEnabled: true**

* **Description:** Determines whether CB automatically moves from **OPEN → HALF\_OPEN** after waitDurationInOpenState.
* **Effect:**
  + true → automatic transition (default, usually what you want)
  + false → you must manually transition the state.

**9️⃣ enable: false**

* **Description:** Enables or disables the circuit breaker.
* **Effect:**
  + true → CB is active and enforces open/closed/fallback behavior
  + false → CB is **disabled**; calls go directly to the service without any protection
* **Example:** Useful in dev or local environments if you don’t want CB logic to interfere.

**🔹 Summary of Flow**

1. **CLOSED:** All requests pass. Failure rate is measured.
2. **OPEN:** Requests fail fast (fallback). Wait for waitDurationInOpenState.
3. **HALF\_OPEN:** A few permittedNumberOfCallsInHalfOpenState trial requests go through.
   * Success → CLOSE
   * Failure → OPEN

**4️⃣ Key differences**

| **Feature** | **Hystrix** | **Resilience4j** |
| --- | --- | --- |
| Fallback annotation | @HystrixCommand(fallbackMethod=...) | @CircuitBreaker(fallbackMethod=...) |
| Request volume threshold | circuitBreaker.requestVolumeThreshold | slidingWindowSize / minimumNumberOfCalls |
| Sleep window | circuitBreaker.sleepWindowInMilliseconds | waitDurationInOpenState |
| Enable/disable CB | circuitBreaker.enabled | enable in YAML |
| Sliding window type | COUNT or TIME based | slidingWindowType |
| Metrics | Hystrix Dashboard | Micrometer / Prometheus |

**✅ Notes**

1. **Fallback method** signature must include Throwable as the last parameter in Resilience4j.
2. Circuit breaker configs are **not inline annotations**; they go in application.yml or application.properties.
3. @RefreshScope is optional if you want dynamic updates from Spring Cloud Config.

If you want, I can rewrite your **currency-conversion-service** to **fully replace Hystrix annotations with Resilience4j**, including:

* Fallback
* Circuit breaker
* Sliding window config
* Load-balanced service calls

Do you want me to do that?

**1️⃣ Hystrix Annotations**

@SpringBootApplication

@EnableHystrix

@EnableHystrixDashboard

public class MyApplication { }

* @EnableHystrix → enables Hystrix circuit breaker support.
* @EnableHystrixDashboard → enables Hystrix Dashboard for monitoring metrics.

**2️⃣ Resilience4j Equivalent**

Resilience4j **does not require @EnableHystrix** because it is **annotation-driven** (@CircuitBreaker, @Retry, etc.).

* Simply use:

@SpringBootApplication

public class MyApplication { }

* And annotate your methods with:

@CircuitBreaker(name = "paymentService", fallbackMethod = "fallback")

public String doPayment() { ... }

**3️⃣ Monitoring / Dashboard**

Hystrix Dashboard is **specific to Hystrix**. Resilience4j uses **Micrometer + Spring Boot Actuator**:

management:

endpoints:

web:

exposure:

include: health,info,prometheus

* Expose metrics via **Prometheus**.
* You can then visualize them in **Grafana**, no separate dashboard annotation is required.
* Optional: add **resilience4j-spring-boot2** dependency to expose **circuit breaker metrics**:

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

</dependency>

* Metrics endpoint /actuator/metrics/circuitbreaker will show circuit breaker stats.

**✅ Summary**

| **Feature** | **Hystrix** | **Resilience4j** |
| --- | --- | --- |
| Enable CB | @EnableHystrix | Not needed (annotation-driven) |
| Dashboard | @EnableHystrixDashboard | Use Actuator + Prometheus/Grafana |
| Circuit breaker config | Inline annotations / properties | application.yml + @CircuitBreaker |
| Metrics | Hystrix Dashboard | Micrometer / Prometheus |

If you want, I can give a **fully working example of currency-conversion-service** using **Resilience4j**, with **circuit breaker, fallback, load balancing, and monitoring** — basically replacing all Hystrix + Ribbon functionality.