

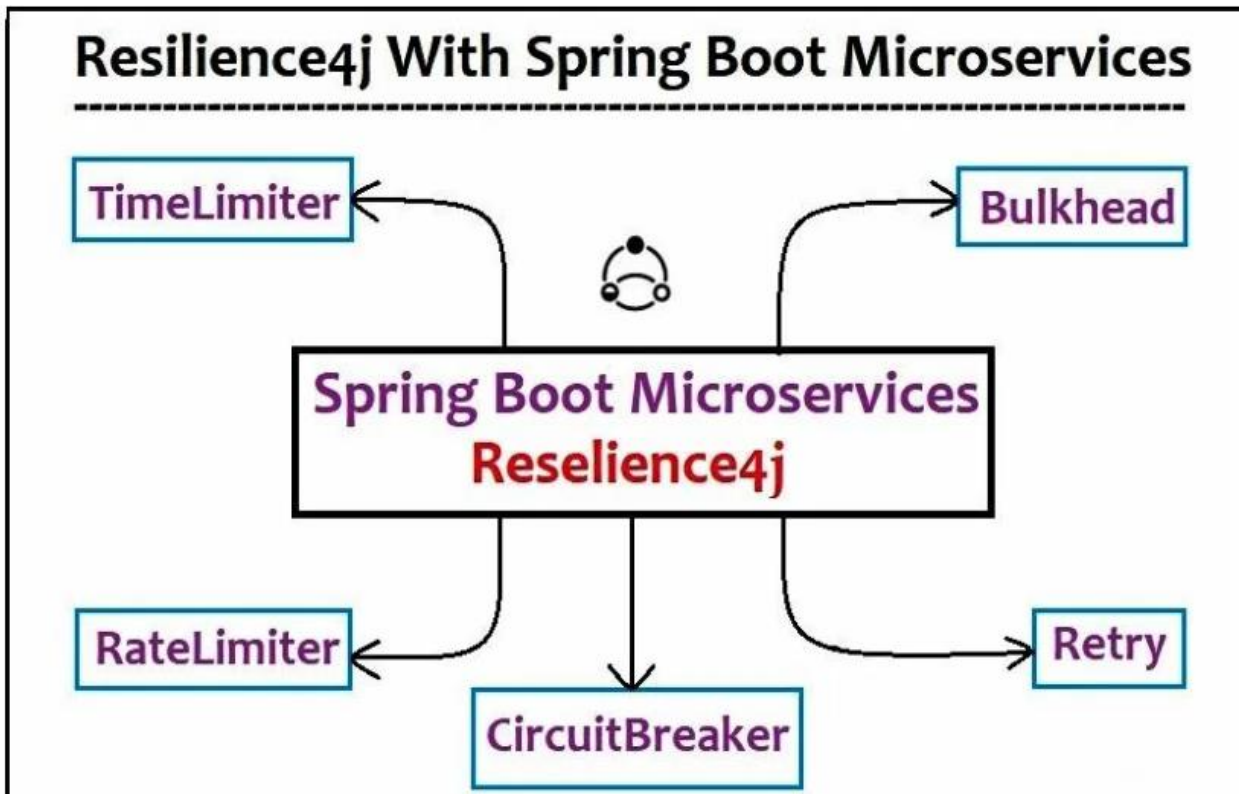
FAULT TOLERANCE WITH RESILIENCE 4J

AGENDA

- ❑ Concept Study
- ❑ Rate Limiting
- ❑ Retry
- ❑ Circuit Breaker
- ❑ Bulkhead
- ❑ Time Limiting or Timeout Handling
- ❑ Conclusion

Concept Study

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 and Resilience4j library is the best candidate for this.



Concept Study

❑ 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**

Concept Study

❑ Common Setup for All Examples

We will have a common setup for all examples before directly going Further. It will include creating a Spring Boot Project using Spring intialzr and adding all dependencies that are required to implement Resilience4j in our project.

❑ Create a Spring Boot Project including all dependencies using Spring intialzr

- **Spring Web**
- **Resilience4J**
- **Spring Boot Actuator**
- **Spring Boot DevTools**
- **Spring Boot AOP (add it manually in our pom.xml)**

Rate Limiting

❑ What is Rate Limiting?

Rate Limiter limits the number of requests for a given period. For example, suppose we want to limit the number of requests on a Rest API and fix it for a particular duration. We can achieve this functionality with the help of annotation `@RateLimiter` provided by Resilience4j without writing a code explicitly.

❑ Rate Limiting 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. We will proceed as follow

- ✓ **Create a RestController class to implement the RateLimiter functionality**
- ✓ **Update application.properties**
- ✓ **Test the implemented RateLimiter using Postman**

Rate Limiting


❑ Rate Limiting Example


Fault Tolerance / Rate Limiter

 Save

▼

...





GET ▼

http://localhost:8080/getMessage

Send ▼

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings


Cookies

Body

Cookies

Headers (5)

Test Results



429 Too Many Requests

9 ms

261 B

Save Response ▼


Pretty


Raw


Preview

Visualize

Text ▼







1

Too many requests : No further request will be accepted. Please try after sometime

Retry

❑ What is Retry?

Suppose Microservice ‘A’ depends on another Microservice ‘B’. We will assume Microservice ‘B’ is a faulty service, fault may be due to any reason, such as service is unavailable, network failure and so on. If Microservice ‘A’ retries to send request 2 to 3 times, the chances of getting response increases . We can achieve this functionality with the help of annotation @Retry provided by Resilience4j without writing a code explicitly. We will develop a scenario where one Microservice will call another Microservice.

❑ Retry Example

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. In order to achieve this, let’s follow below steps to write code and respective configurations. We will proceed as follow

- ✓ **Create a RestController class to implement the Retry functionality**
- ✓ **Update application.properties**
- ✓ **Test the implemented RateLimiter using Postman**



Retry

❏ Retry Example

Fault Tolerance / Retry

Save

...



GET

⌵

http://localhost:8080/getInvoice

Send


⌵

ParamsAuthorizationHeaders (6)BodyPre-request ScriptTestsSettingsCookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
--	-----	-------	-------------	-----	-----------

BodyCookiesHeaders (5)Test Results



200 OK58 ms210 B

Save Response

⌵

PrettyRawPreviewVisualizeText

⌵



1SERVICE IS DOWN, PLEASE TRY AFTER SOMETIME !!!

Circuit Breaker

❑ What is Circuit Breaker ?

Circuit Breaker is a pattern in developing the Microservices based applications in order to tolerate any fault. Suppose a Microservice 'A' is internally calling another Microservice 'B' and 'B' has some fault. 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'. We can achieve this functionality with the help of annotation `@CircuitBreaker` provided by Resilience4j without writing a code explicitly. We will develop a scenario where one Microservice will call another Microservice.

❑ Circuit Breaker Example

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. In order to achieve this, let's follow below steps to write code and respective configurations. We will proceed as follow

- ✓ **Create a RestController class to implement the Circuit Breaker functionality**
- ✓ **Update application.properties**
- ✓ **Test the implemented RateLimiter using Postman**

Circuit Breaker

❏ Circuit Breaker Example

Fault Tolerance / Circuit Breaker

 Save



GET



http://localhost:8080/getInvoice2

Send



Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

KEY

VALUE

DESCRIPTION



Bulk Edit

Body

Cookies

Headers (5)

Test Results



200 OK

23 ms

210 B

Save Response



Pretty

Raw

Preview

Visualize

Text



1 SERVICE IS DOWN, PLEASE TRY AFTER SOMETIME !!!

Bulkhead

❑ 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. We can achieve this functionality with the help of annotation `@Bulkhead` provided by Resilience4j without writing a code explicitly. We will develop a scenario where one Microservice will call another Microservice.

❑ Bulkhead Example

In order to achieve the Bulkhead functionality, in this 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. We will proceed as follow

- ✓ **Create a RestController class to implement the Bulkhead functionality**
- ✓ **Update application.properties**
- ✓ **Test the implemented RateLimiter using Postman**

Bulkhead

❏ Bulkhead Example

Fault Tolerance / bulkhead

 Save



GET



http://localhost:8080/getMessage2

Send



Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

KEY

VALUE

DESCRIPTION



Bulk Edit

Body

Cookies

Headers (5)

Test Results



200 OK

8 ms

196 B

Save Response



Pretty

Raw

Preview

Visualize

Text



1 Message from getMessage() :Hello

Time Limiting or Timeout Handling

❑ 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 with the help of annotation @Timelimiter provided by Resilience4j without writing a code explicitly. We will develop a scenario where one Microservice will call another Microservice.

❑ TimeLimiter Example

In order to achieve the TimeLimiter functionality, in this 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. We will proceed as follow

- ✓ **Create a RestController class to implement the TimeLimiter functionality**
- ✓ **Update application.properties**
- ✓ **Test the implemented RateLimiter using Postman**

Time Limiting or Timeout Handling

❑ TimeLimiter Example

The screenshot shows a REST client interface with the following components:

- Top Bar:** "Fault Tolerance / TimeLimiter" on the left, "Save" button and a dropdown menu in the center, and edit/comment icons on the right.
- Request Bar:** Method "GET" with a dropdown, and URL "http://localhost:8080/getMessageTL". A blue "Send" button is on the right.
- Request Tabs:** Params, Authorization, Headers (6), Body, Pre-request Script, Tests, Settings, and Cookies.
- Response Tabs:** Body (selected), Cookies, Headers (5), and Test Results.
- Response Summary:** A globe icon, status "200 OK", time "391 ms", size "198 B", and a "Save Response" button with a dropdown.
- Response Body:** A tabbed interface with "Pretty" (selected), "Raw", "Preview", and "Visualize". A "Text" dropdown and a refresh icon are also present.
- Response Content:** A single line of text: "1 Executing Within the time Limit..."

Conclusion

After going through this work, we have improve our skills about distributed system and we have implement it in a Spring boot application using Resilience4j library. The links of our resources and github repository are attached below

<https://javatechonline.com/how-to-implement-fault-tolerance-in-microservices-using-resilience4j/>

<https://github.com/Dilane-Kamga/FaultToleranceImplementation.git>

MERCI
Pour votre attention