Ritesh Panigrahi

Follow

Making microservices Fault Tolerant and Resilient

Ritesh Panigrahi

Aug 19, 2021 • 5 min read

In this article, we will understand

1. Why do we need to make our microservices fault-tolerant and resilient?
2. What do fault tolerance and resilience mean?
3. Various reasons for failures for the microservices
4. Design Patterns to achieve Resiliency

Why do we need to make our microservices fault-tolerant and resilient?

In a monolithic application, a single error will break the whole architecture, so to avoid this microservice architecture is adopted because it contains multiple independently deployable units which won't affect the entire system.
So does that mean the microservice architecture will never fail? No, suppose

there are 3 microservices A, B, and C:- A calls B and B calls C, what will happen when service C is down?
The whole architecture will be at a halt as B will not work because it can't call C, which will affect A. Even though only one service is down, the whole architecture got disturbed due to dependencies.
So, from this what we can understand is just creating microservices is not enough we even need to think about What to do if any of the services fail? What will the fallback plan be?
We can do several things to ensure that the entire chain of microservices does not fail with the failure of a single component.

What does fault tolerance and resilience mean?

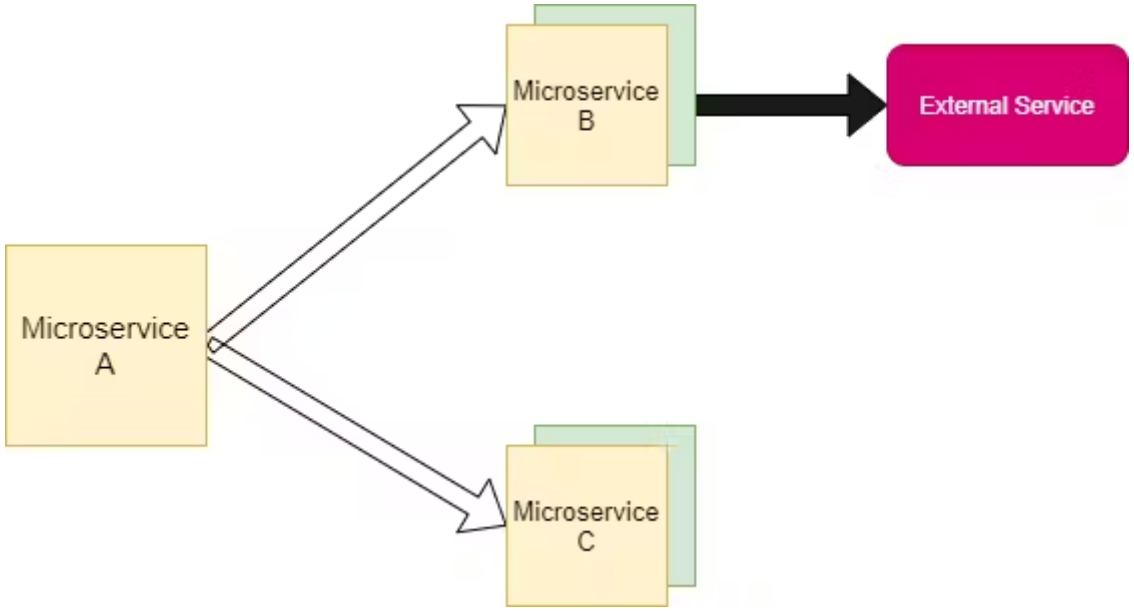
Generally, both these terms are used interchangeably but there is a slight difference in both.

Fault tolerance means given a microservice application how much tolerant it is or what is the impact when there is a fault, will the whole application goes down or part of functionality goes down or there are some arrangements to handle failures.
Tolerance the system has for a fault - Fault Tolerance

Resilience means how many faults the system can tolerate.

Various reasons for failures for the microservices

Consider this below architecture, here service A calls service B and service C. Service B calls an external service.



Now we will understand different cases

Case 1:- *If any of service B or C is down*
Solution:- We can run duplicate instances of the same services so that even if any one of the instances goes down then we can use another instance available. If you are using spring cloud then thanks to service discovery- Eureka and client-side load balancing-Ribbon which helps us handle this easily.

Case 2:- *If any microservice is slow*

Suppose the external service to which service B calls is slow, this will result in service B is slow.
This might seem not a big issue but due to this call to service C becomes slow even though services B and C are independent.
How?
It is because of threads, if there are multiple requests to service B which is slow the thread pool is filled with those threads which are for service B so now if there comes a request for service C because the pool is filled the request has to wait which makes service C slow.

Solution: Add timeouts to the request, so the request will no longer be in the pool if there is no response within the given timeout.

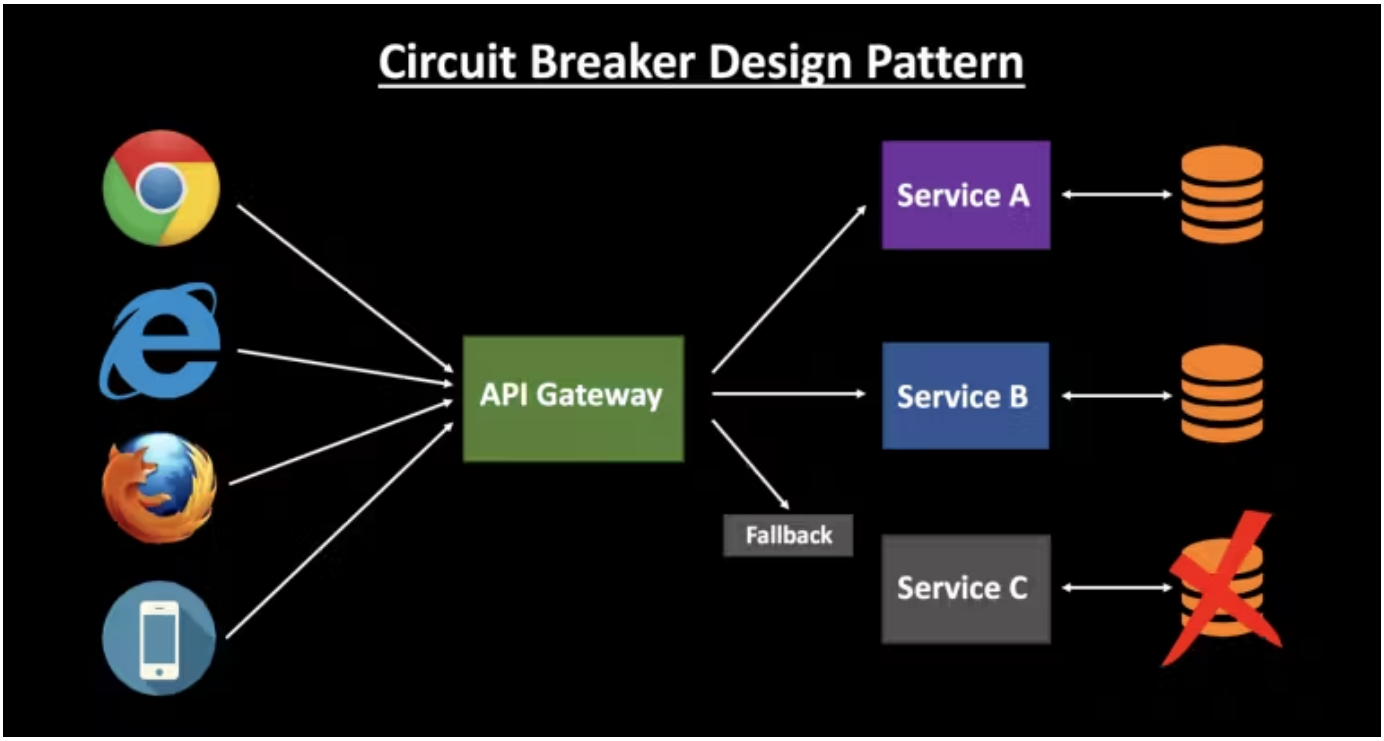
But still, this will partly solve the issue because let's assume we added timeouts for 5 secs and the request is coming per sec this will also lead to a thread pool issue.

There can be multiple other reasons like Database is not accessible, Server is down, etc which can cause faults.

Design Patterns to achieve Resiliency

Circuit Breaker Pattern

As in the above example, we have seen that service B is slow due to which service C also becomes slow.
So to avoid this, we can first detect where the problem is and then stop calling that component for some amount of time. Basically, we have to deactivate that component which gives a problem for some time.
This pattern is called Circuit Breaker pattern as we are breaking the call to the service which gives a problem for a certain amount of time and then again resuming.



We can use some popular third-party libraries to implement circuit breaking in your application, such as Polly and Hystrix.

Retry Design Pattern

In this pattern, we can retry the connection which has failed earlier. This is useful for temporary issues. A lot of times a simple retry might fix the issue.

Timeout Design Pattern

This pattern states that we should not wait for an indefinite amount of time to get the response. If some service is not responding within a certain time, throw an exception and stop the request.

Bulkhead Pattern

The Bulkhead pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function.
It's named after the sectioned partitions (bulkheads) of a ship's hull. If a ship's hull is compromised, only the damaged section fills with water, which prevents the ship from sinking.

This is it for this article. If you found this article helpful do like it.
As this is my first article, if you have any suggestions then please put them in the comments.
Thanks for reading !!

Subscribe to my newsletter

Read articles from **Ritesh Panigrahi** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

Enter your email address SUBSCRIBE

Springboot Microservices Java Spring

WRITTEN BY

Ritesh Panigrahi

Follow

🙌 Hi, I'm Ritesh. I am a Software Engineer and I write Technical Blogs on Java, SpringBoot,(add++).

MORE ARTICLES

Ritesh Panigrahi

Top OOPS Interview Questions & Answers

1. What is Method Signature and its use? Let's consider the below method and try to get its method s...

Ritesh Panigrahi

Spring Security: Architecture and Internal Workflow

In the previous article, we learned about CSRF. If you have not followed previous articles if this s...