Introduction to the need of Resiliency inside microservices

How to make our microservices to be resilient?

What is the meaning of resiliency or resilient?

The meaning of resiliency is something which can withstand the tough times and bouncing back.

We should also make our microservices resilient in nature so that they can withstand tough times like network problem or any performance issues.

So, to understand more about these resiliency in microservices, first we need to question ourselves

few questions.

**How do we avoid cascading failures inside our microservices network?**

We all know that inside a microservice network, when my client application or my UI application sending a request to the Microservice network, many microservices they will work together and they will send a combined response to the client application in such scenarios, how we are going to handle a scenario where one of the services is failing or responding very slowly.

So **how do we make sure that it is not having a ripple effect on the other microservices?**

For example, previously we saw a scenario where a client application, they can invoke one of the REST API available inside the accounts microservice, my accounts microservice is going to work together with the loans and cards microservice.

Once the response from loans and cards microservice is received, the total response will

be aggregated at the account microservice and the same will be sent back to the client.

So, in these kinds of scenarios, think like loans or cards, microservice is failing or they are responding

very slowly.

If one of the services is failing or working very slowly, all the dependent services like Accounts,

Gateway Server, they will keep waiting for the response and it will eventually have a ripple effect

on other microservices which will consume all the threads and memory inside these dependent microservices.

So, we need to make sure that the entire chain of microservices does not fail if one of the participating microservice is failing or is responding very slowly.

So, this is the very first problem that we need to think about.

**How do we handle failures gracefully with the fallbacks?**

In the same example where multiple microservices work together to send a response to the client application, how we are going to build a fallback mechanism if one of the microservice is not working.

If my cards microservice is not working properly, at least I should be able to send the accounts and loans information to my client application instead of sending an exception saying that we are not able to send any kind of information.

So, these kinds of scenarios will be very common inside your microservices.

That is why we should always have a fallback mechanism.

Like if one of the participating microservice is not able to respond successfully, we should have some

fallback mechanism that will return a default value or return a value from the cache, or try to invoke

other service or try to fetch the details from another database.

So, the fallback mechanism can be anything, but at least you should make sure there is some fallback

mechanism. So that you are not failing the entire request that is coming from the client applications.

**How we are going to make our services self-healing capable?**

For example, if one of the participating microservice inside a microservice network is responding very slowly due to some performance issues or due to some network issues, how we are going to make our services self-healing capable, maybe due to some network issue or maybe due to a temporary glitch inside my microservice.

It is responding very slowly or it may not be responding at all.

So how we can configure some timeouts and retries and give some time for the failed service to recover itself.

Maybe if I try to retry multiple times, like three times or four times, my service may start working

and I may get a successful response.

Or very similarly, instead of waiting for a large amount of time for my microservice to give a response, if I can timeout within a very short period of time, that will release some memory or threads inside the microservice where we are having a problem.

So, with a quick timeout we can give some time to the failed service to recover itself.

So, there are many strategies on how we can make our services self-healing capable.

So, we are going to discuss them in this section.

So, these are the challenges or the problems or the questions that we may face while we are trying to build our microservices.

Now, let me give you a quick introduction about the solution we need to follow to overcome this challenge.

The solution is inside microservices there are many patterns to building resilient applications.

Long back inside the Java ecosystem, we used to have a library called Hystrix, so this is a library developed by Netflix team itself, and it used to be widely used for implementing resiliency patterns inside any web application or inside microservices.

However, Hystrix entered maintenance mode in 2018, and it is no longer being actively developed.

A new library came into picture with the name **Resiliency4j**.

So, this gained a significant popularity in a short period of time, which helped the developers by filling the gap left by the Hystrix library.

Inside this Resiliency4j library, it provides lot many resilient related patterns which we can

choose based upon our business requirements.

Let me give you a quick introduction about resiliency4j.

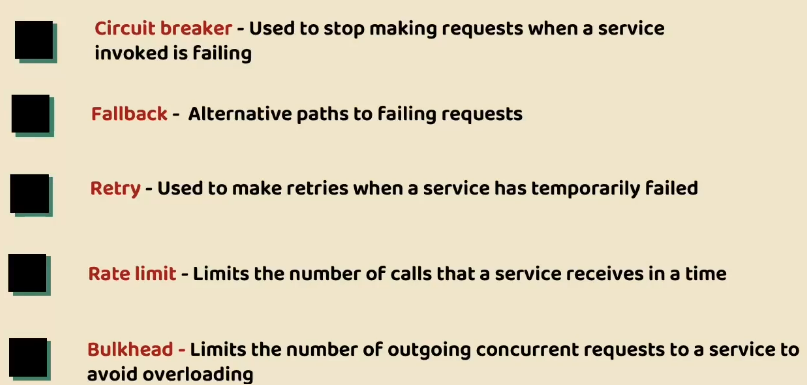
Resiliency4j is a lightweight fault tolerance library designed especially for **functional programming**.

That does not mean non-functional programs cannot use this library.

This resiliency4j library offers many patterns to make our applications or microservices to be fault, tolerant and resilient in nature.

If you ask me what are these patterns?

So, these are the patterns with the name like **circuit breaker**, **fallback**, **retry**, **rate limit**, **bulkhead**.



<https://resilience4j.readme.io/>

Resiliency4j is a fault tolerance library for Java based applications.

It is a lightweight fault tolerance library inspired by Netflix Hystrix but designed for functional

programming.

<https://resilience4j.readme.io/docs/circuitbreaker>

<https://resilience4j.readme.io/docs/bulkhead>

<https://resilience4j.readme.io/docs/ratelimiter>

<https://resilience4j.readme.io/docs/timeout>

<https://resilience4j.readme.io/docs/cache>

<https://resilience4j.readme.io/docs/getting-started-3>  
  
<https://resilience4j.readme.io/docs/getting-started-6>

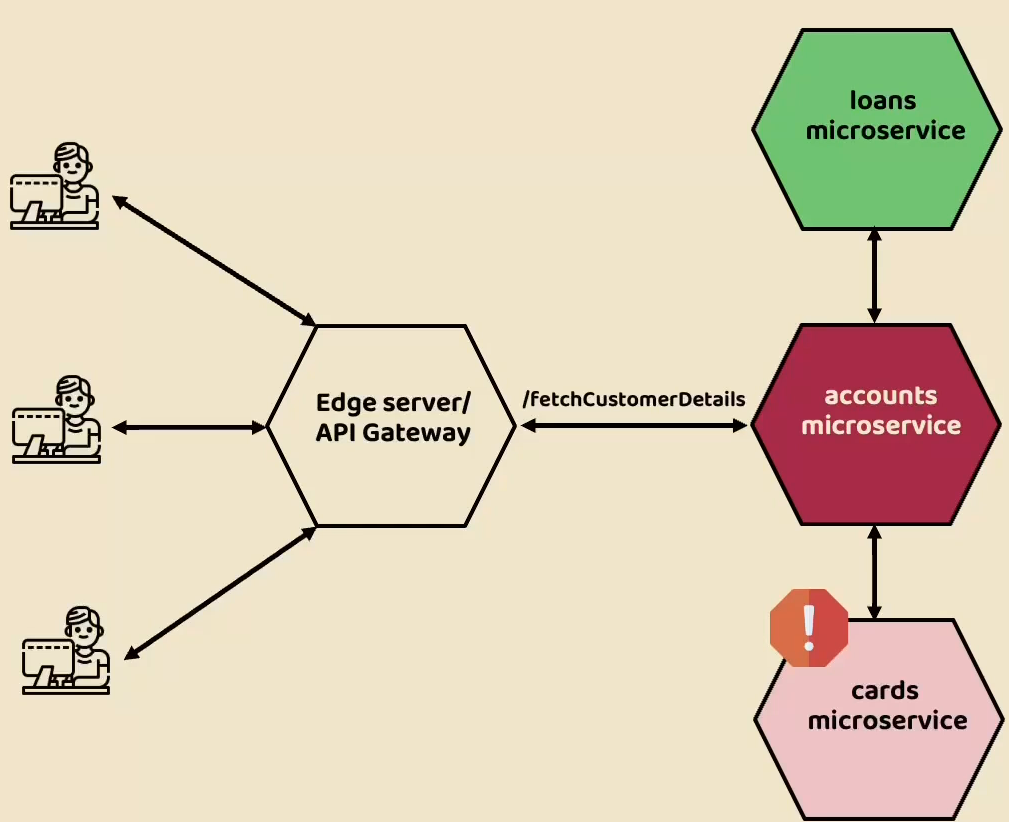
<https://resilience4j.readme.io/docs/feign>

<https://resilience4j.readme.io/docs/micrometer>

<https://resilience4j.readme.io/docs/grafana-1>

Typical use case or scenario for the need or Resiliency

Let me give a typical scenario that we may face inside microservices, so that you will better understand what is fault tolerance and why we should make our microservices resilient to the problems that we may face on day-to-day basis.



Like you can see here, we have Edge Server or API gateway that accepts the external traffic from

the clients.

If you recall, inside our microservices network, we have an REST API available inside the accounts

microservice with the path **fetchCustomerDetails**.

Whenever someone is trying to invoke this API, they are expecting the complete details of the customer, like what are the customer details, what are his loans and cards details.

For the same, initially my gateway server at the edge server.

It is going to forward the request to the accounts microservice.

My accounts microservice will have details about only accounts and customer, but it will not have details around loans and cards microservice.

That is why it is going to depend on other microservices like loans and cards microservice to fetch the

respective information.

Once it fetches the information from other microservice, it is going to aggregate or collate all the

response and send the complete information to the edge server. From Edge Server it will reach to the client applications. So, this is a typical scenario.

Now think like in this scenario, one of the microservice which is participating inside this flow is

not working properly, or maybe it is trying to handle too many requests.

That is why it is trying to respond very slowly. So, the problem can be anything.

So, let us try to understand what will happen due to this issue.

Here I just highlighted or assumed that cards microservice has some issues.

That is why you can see there is a warning symbol on top of the cards microservice.

When my accounts microservice invoke cards microservice to get the cards details for a

customer, it is not going to get the response in an expected time frame.

Maybe it is taking too long or my cards microservice is never responding.

Assume like in a typical successful scenario, your accounts microservice will receive the response

from the cards microservice within less than a second.

But in this scenario, since cards microservice is not working properly or it is responding very slowly,

maybe it took around 10s or even after 10s it is not responding.

So, these 10s inside my accounts microservice, there will be a thread which will wait for my cards microservice response or there might be some resources like memory CPU usage allocated inside my accounts microservice to invoke the cards microservice.

So here my accounts microservice waited for more than 10s which means for a single request,

if you are making your accounts microservice to wait for more than 10s, and if the same thing happens for all the remaining requests coming from the external clients, you can imagine due to cards microservice performance issues, it is going to have a ripple effect on accounts microservice.

Now since accounts microservice is waiting for a long time for cards response, without my accounts, microservice is also going to respond slowly to the edge server.

Now the ripple effect went to Edge server as well, so the edge server will keep waiting and with

that there will be too many threads, there will be too many resources being consumed on the edge server as well.

So, this will also impact the performance of your edge server and with that, all the traffic coming towards your edge server from external client applications not only for **fetchCustomerDetails** for other APIs also is going to get impacted.

So, this is a proper or typical or most common scenario that may happen inside your microservices.

You can see one microservice, which is having a problem, is creating ripple effects throughout your

microservice network.

So, this is what we need to avoid with the help of resiliency patterns.

So, let us try to understand how to overcome this challenge.

So, to handle these kinds of scenarios only we can use one of the resiliency patterns which is Circuit

Breaker Pattern.

Deep dive on Circuit Breaker Pattern

What is the circuit breaker pattern and how it is going to help us in stopping the cascading failures inside our microservice network?

Quick introduction about Circuit Breaker that we use as part of electrical system.

We all use many circuit breakers inside our house or office.

So, what is the purpose of this circuit breaker?

It is a safety device designed to protect the electrical circuit from excessive current or any potential

fire hazards.

Whenever a circuit breaker detects that there is a fault in the flow of the electricity may be due to

the short circuit or overload of the electricity in such scenarios to ensure the safety and the stability

of your house or the office, it is going to automatically open or it is going to automatically trip

itself.

So, there will not be any electricity passing towards the components that you use inside your house or office.

You can see here we also have one sample representation Since the circuit breaker is open now, my electric components like bulb, it is going to get protected because there is a short circuit or there is an overload detected by the circuit breaker.

So, with this, my bulb is not going to get affected.

Think of a scenario if the circuit breaker is not there, the overloaded current at the short circuit

is going to travel to the bulb inside my electric system and eventually my bulb is going to get damaged.

The same concept is considered in building the circuit breaker pattern inside the software development.

Let us try to understand how this is relevant to the circuit breaker pattern inside software development.

Just like inside electrical system, how the electricity is going to pass throughout the network.

Very similarly, in a distributed environment, the calls to the remote resources and services are going

to happen, and these remote calls may fail due to many reasons, maybe due to some transient faults

or due to some slow network or maybe due to some timeout or maybe due to resources being overcommitted or maybe due to temporarily unavailable.

The reason can be anything but.

But most of the times these faults typically correct themselves after a short period of time.

Maybe if one of the microservice is responding very slowly, it is not going to be in such state forever, at some point of time it is going to recover itself.

Or if you take the network problem itself, the network problem is going to be a temporary problem.

It is not going to be a permanent problem.

So, since these kinds of issues are going to get resolved for after a short period of time, we need to

be ready to face these kinds of temporary issues inside our microservices network.

So, let us try to understand how the circuit breaker pattern is going to help in these kinds of scenarios.

The circuit breaker pattern, which is inspired from the electrical circuit breaker itself, it is going

to monitor all the remote calls happening to a particular service.

If a particular service like micro service, if it is taking too long to respond or if it is not responding

or if it is having some network issues, the circuit breaker is going to kill that call.

And at the same time, since it is monitoring all the calls that are going to the cards microservice

and if most of the calls are failing due to the slow response or due to some network issue,

the circuit breaker implementation will pop up and it will make sure all the future requests coming

to the cards microservice are failing immediately, which means it will never allow the traffic to

the cards microservice, instead it is going to tell to the client applications my cards microservice is not working, that is why I am sending the failure response immediately.

With that, my dependent microservices like accounts microservice and gateway server, they don't have to wait for a longer time.

So, by failing fast we are preventing the ripple effect on the gateway server and the accounts microservice.

With this approach, we are also making sure other rest APIs inside our microservices are not affected

just because cards microservice is down.

Maybe there are some microservices or rest APIs where only accounts and loans microservices they have to work together. All such remote calls will be successful because there is no ripple effect happened on my accounts microservice with the help of this circuit breaker pattern.

So here you may have a very good question, which is I said my circuit breaker pattern is going to stop

all the future requests coming towards the cards microservice and it is going to immediate failure response to the client microservices like accounts microservice **will this going to be forever?**

Of course not.

At some point of time your cards microservice is going to get recovered because your circuit breaker

pattern stopped all the incoming traffic and gave enough time for your cards microservice to recover.

That is why periodically the circuit breaker pattern also sends some partial traffic to the cards microservice to check if my cards microservice is recovered or not.

If this partial traffic, which is allowed by the circuit breaker is successful, then my circuit breaker

pattern is going to allow all the incoming requests towards the cards microservice.

Whereas if this partial traffic, which it allowed previously is also failing, then again, it is going

to stop all the incoming requests for few more seconds or few more minutes based upon our configurations.

So, there are very good number of advantages due to this circuit breaker pattern.

The primary advantages are it is going to fail your request very fastly instead of your accounts microservice waiting for more than 10s and realize that my cards microservice is not working.

It is going to fail the incoming request to the cards microservice instantly with that my accounts microservice never have to wait for 10s instead.

Now it is going to get the error response within a one second or even within a second.

And the next advantage that we have with the circuit breaker pattern is we can make our microservices fail gracefully.

Maybe whenever my cards microservice is down or responding very slowly, I can write some fallback mechanism which will fail the request gracefully.

We will explain how to write a fallback mechanism with the help of Circuit Breaker in the coming lectures.

And the last primary advantage that we have is the circuit breaker is going to help microservices to

recover seamlessly because it is going to give them some rest for a short period of time, like 30s

or 90s.

Based upon your configurations within the 30s or 90s, your microservices will try to recover because

there is no incoming traffic.

And if there is some network problem within 30 or 90s, most of the time the network problems will also get resolved.

So, this is a very quick introduction about circuit breaker pattern.

Three states of Circuit Breaker pattern

How circuit breaker pattern is going to control the traffic coming towards a particular

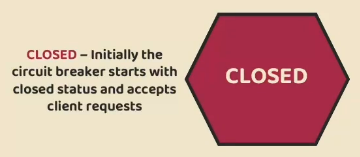
Microservice?

By default, circuit breaker pattern is not going to monitor all your microservices, we need to configure this circuit breaker pattern wherever we need.

Think like you have configured this circuit breaker pattern for one of the microservice.

Whenever we activate the circuit breaker pattern on any microservice, it is going to control the flow

of traffic towards the microservice by using three different states.



The very first state is ***closed state***.

Initially, when the application starts by default, your circuit breaker will be in this closed state.

Inside this closed status, it is going to accept all the requests coming towards your microservice.

You can also try to correlate this with the circuit breaker present inside the electric system.

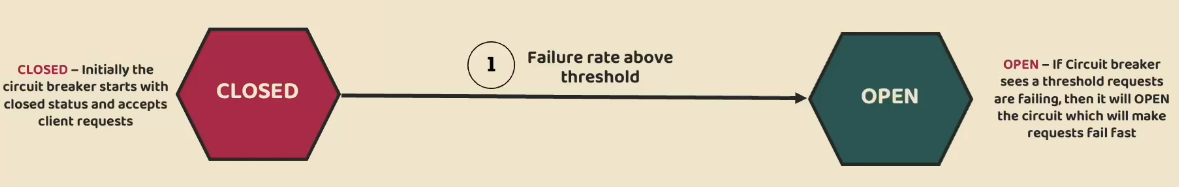
Inside the electric system, whenever the circuit is closed, that means all the current is going to

be passed to the next components.

Very similarly inside microservices, when the circuit breaker pattern is in the closed status, it

means it is going to allow all the traffic towards a microservice.

With this closed status, your circuit breaker pattern is going to monitor all the requests coming towards your microservices and it will try to understand whether the microservice is responding properly or if there are any network issue or is there any slow response from the microservice.



So based upon these monitoring and analysis, if my circuit breaker identifies that most of the requests coming to the microservices are being failed, then it is going to immediately jump to the ***open status***.

How it is going to transition from close to open status is based upon the **failure rate threshold** that

you have defined.

Maybe you might have configured saying that if 50% of my traffic that is coming towards my microservice is failing, then please move to the open status.

So based upon these configurations, if the circuit breaker identifies that 50% of the traffic is failed,

then immediately it is going to jump from the close to the open status.

Whenever the circuit breaker is in open status, that means it is not going to send any request to the

actual microservice.

Instead, it is going to fail immediately and send the error response to the invoking microservices

at the client applications.

This way it will make sure there is no ripple effect on the invoking microservices due to the issues

in a particular microservice.

**And do you think this open status will be forever?**

Off course not.

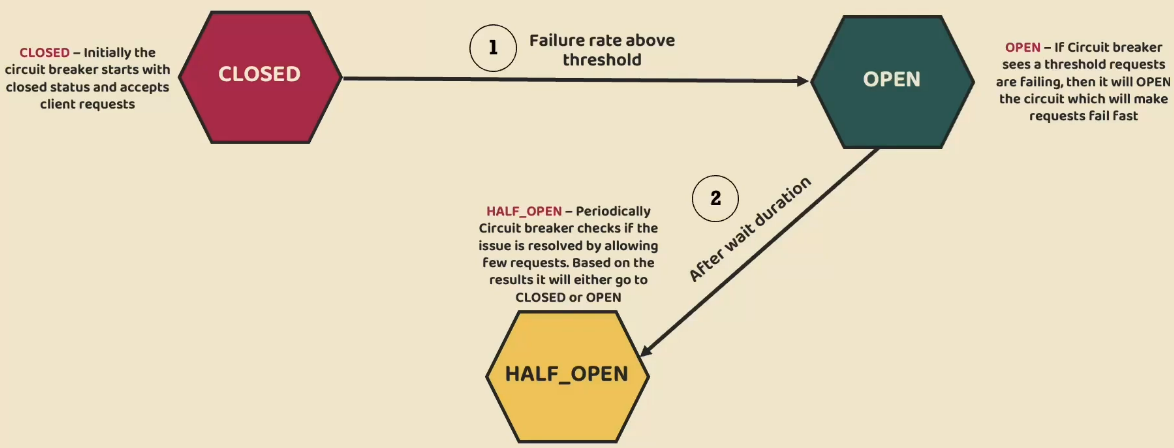
Keeping the circuit breaker open forever means we are never going to allow that traffic towards a particular microservice.

So, what behind the scenes will happen is my circuit breaker pattern will wait in the open status based upon your configurations.

Maybe if you have configured for 90s up to 90s, the circuit breaker pattern will be in open status

and it will give 90s time for my actual microservice to recover from the failures or from the network

issues.

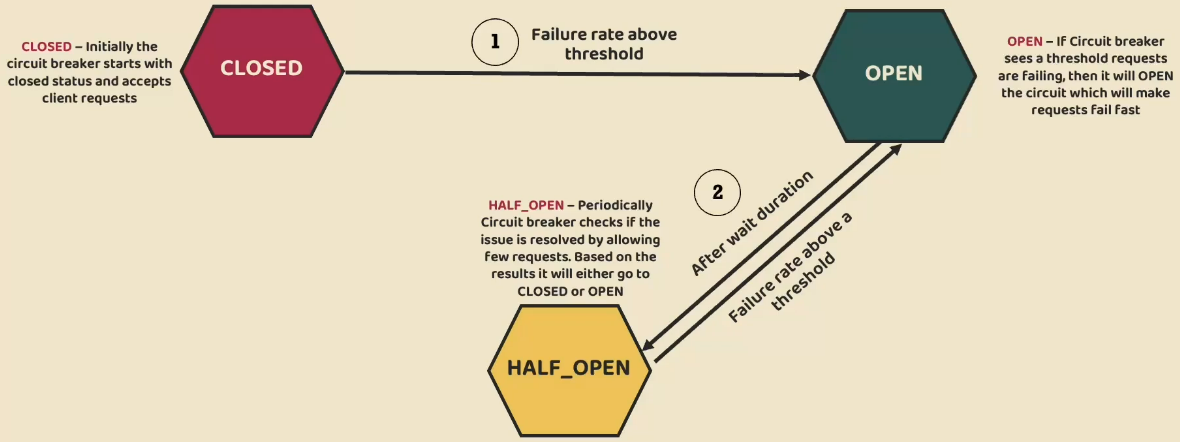


After 90s, my circuit breaker pattern will move into the ***half open status***.

Inside this half open status the circuit breaker will allow only few requests based upon your configurations.

Maybe it will allow 10 requests or 20 requests, and inside these 10 or 20 requests, again, if at least 50% of the requests are failing, then it will again jump back to the open status

and it will wait again for 90s.



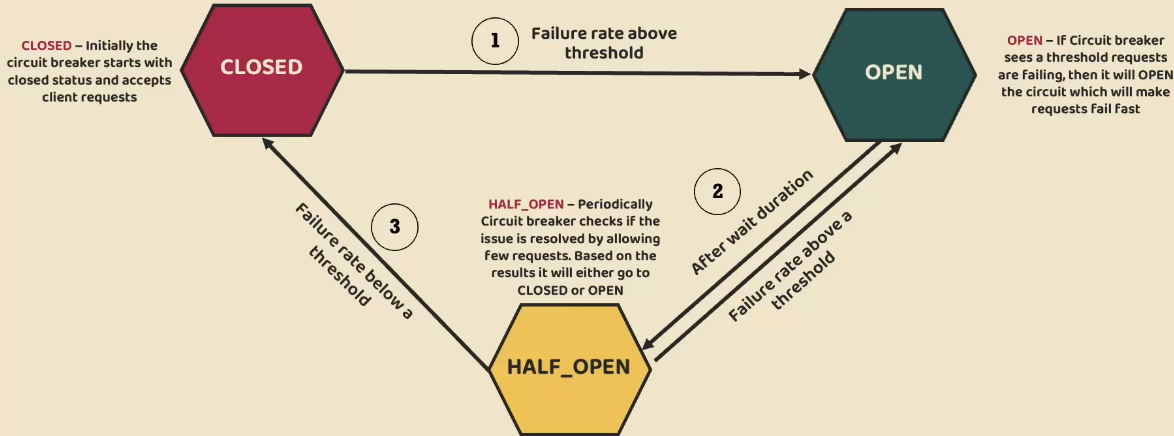
So, this cycle, from open to half open, half open to open will continuously happen periodically.

At some point of time when my circuit breaker realizes that majority of the traffic or the requests

are being processed successfully, then it will move from half open status to ***closed status***.

And with that my microservice, where I have configured, circuit breaker pattern will continue to work

normally.



This sounds very complicated to achieve, but believe me, this is going to be super, super easy to

configure circuit breaker pattern inside microservices because we have friends like Resiliency4j

and Spring boot Framework.

With the help of these libraries and frameworks, we can easily configure circuit breaker pattern.

Implementing Circuit Breaker pattern in Gateway - Part 1

Implementing Circuit Breaker pattern in Gateway - Part 2

Implementing Circuit Breaker pattern with Feign Client - Part 1

Implementing Circuit Breaker pattern with Feign Client - Part 2

Http timeout configurations

Introduction to Retry pattern

Implementing Retry pattern in Gateway

Implementing Retry pattern in accounts – Part 1

Implementing Retry pattern in accounts – Part 2

Introduction to Rate Limiter pattern

Introduction to Redis Rate Limiter in Gateway Server

Implementing Redis Rate Limiter in Gateway Server

Implementing Redis Rate Limiter in accounts

Introduction to Bulkhead pattern

Aspect order of Resiliency patterns

Demo of Resiliency patterns using Docker containers & Docker compose