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.

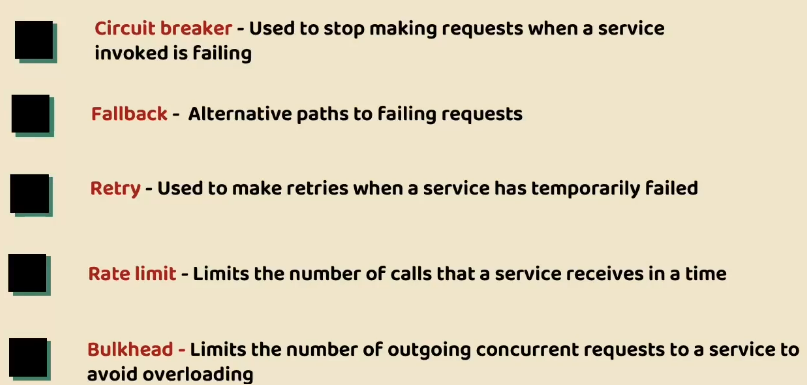
Resiliency4j is a lightweight fault tolerance library designed especially for **functional programming**.

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>

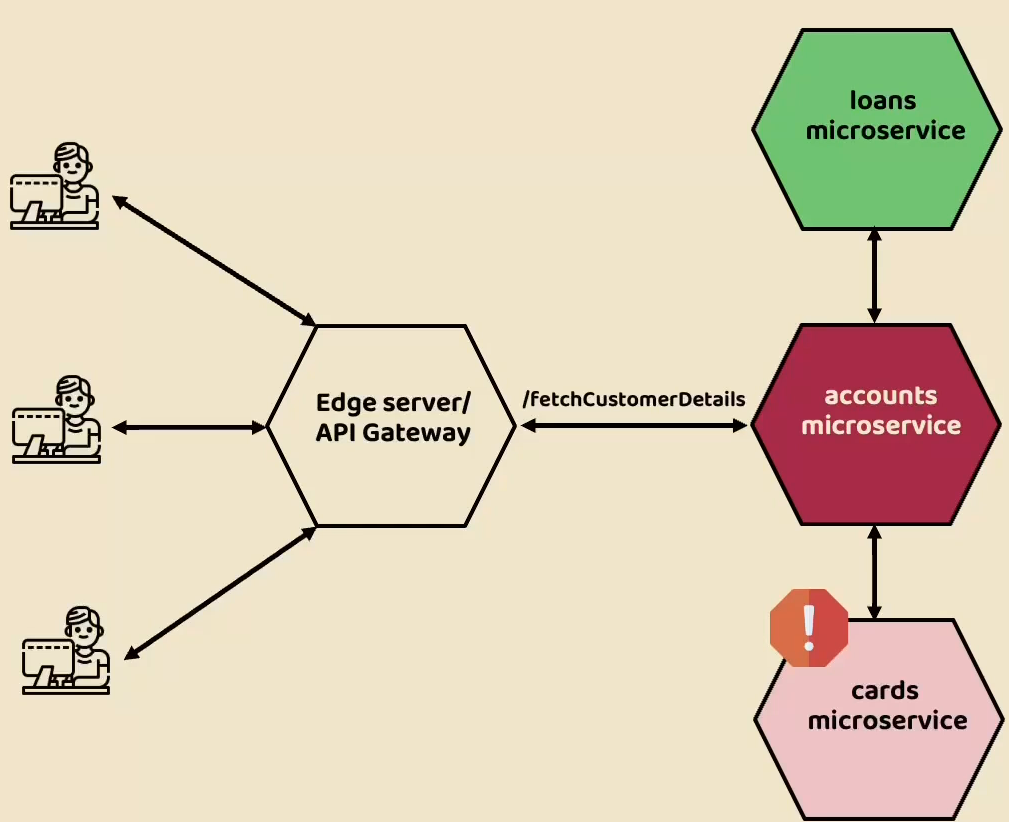
<https://resilience4j.readme.io/docs/feign>

<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

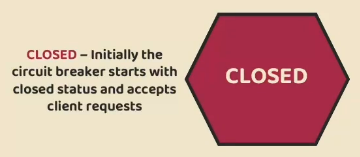
Microservice?

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.

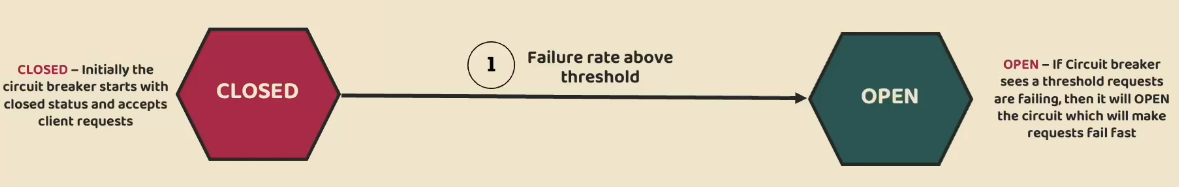
Inside the electric system, whenever the circuit is closed, that means all the current is going to

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.

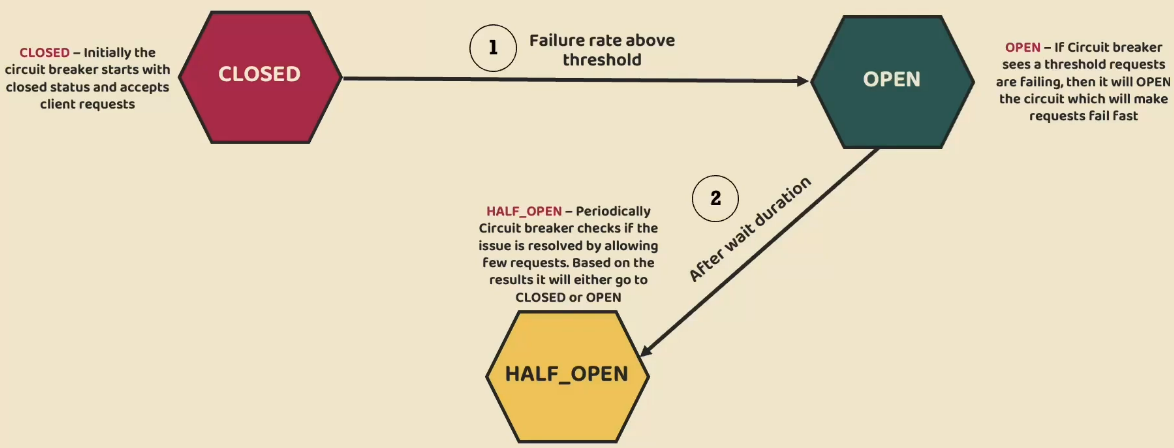
Keeping the circuit breaker open forever means we are never going to allow that traffic towards a particular microservice.

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.

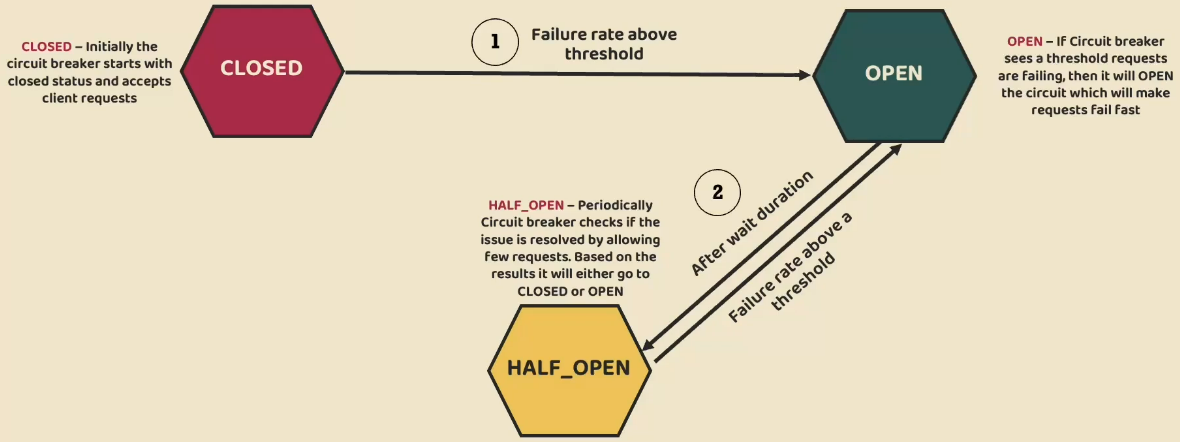


After 90s, my circuit breaker pattern will move into the ***half open status***.

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.



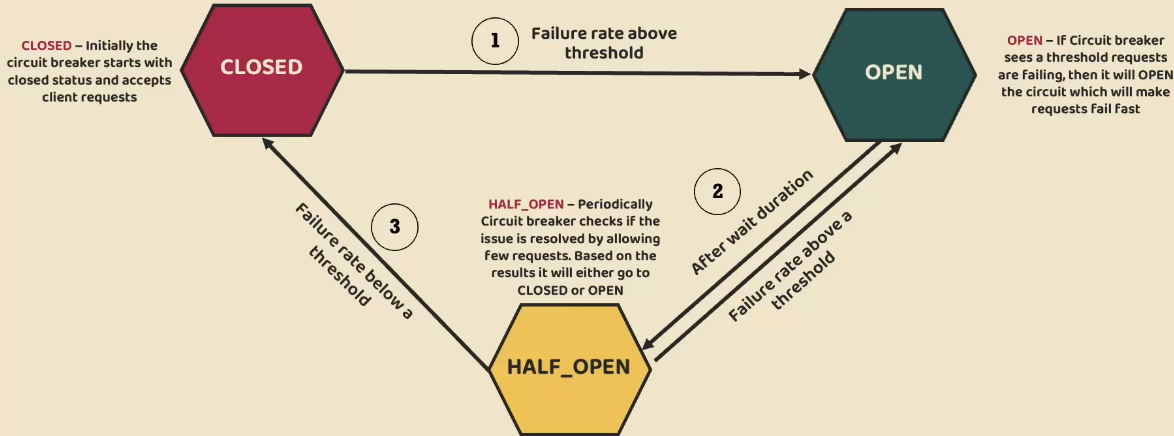
So, this cycle, from open to half open, half open to open will continuously happen periodically.

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

How to implement circuit breaker pattern inside the Gateway server, because as of now, Gateway Server is acting as an edge server inside our microservice network.

There might be some business scenario or there might be some requirements from your clients saying that they want you to implement circuit breaker pattern at the Edge server itself.

So, let us try to understand how to implement the same.

Add the below dependency to gatewayserver project

D:\Experiments\Microservices\sb-bank-application\gatewayserver\pom.xml

<dependency>

  <groupId>org.springframework.cloud</groupId>

  <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>

</dependency>

Since Edge server, our gateway server built based upon spring react to model,

we need to make sure we are mentioning the artifact ID as spring cloud starter circuitbreaker-reactor-resilience4j.

D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\java\com\eazybytes\gatewayserver\GatewayserverApplication.java

We can go to the Spring Boot main class, which is GatewayserverApplication.

Here you can see we have done all routing related configurations.

As of now we are trying to use filters like **rewritePath** and **addResponseHeader**.

Very similarly, I am going to leverage one of the inbuilt provided filters by the spring Cloud gateway,

and this filter is none other than circuit breaker.

So, if you try to invoke the circuit breaker filter, it is going to accept some lambda-based configurations.

D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\resources\application.properties

….

resilience4j.circuitbreaker.configs.default.slidingWindowSize=10

resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=2

resilience4j.circuitbreaker.configs.default.failureRateThreshold=50

resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=10000

….

Using this property, we are communicating to the circuit breaker pattern on how many requests it must initially monitor before it tries to change the status from close to open.

In other words, with this property, I am telling to my circuit breaker pattern, please at least monitor

10 requests coming towards my accounts microservice. After monitoring ten requests, you can take the decision whether to continue with the close status or to move to the open status.

The next property I have mentioned here is, permitted number of calls in half open status. Like we are discussing previously.

Once my circuit breaker pattern moved into open status, it will never be in open status forever.

Periodically it is going to move to the half open state and it is going to allow certain amount of traffic

to the accounts microservice and since circuit breaker pattern cannot decide how many requests it must pass, we need to provide such information using this property.

So here I have mentioned value2 this means I want my secure breaker pattern to allow to request in

the half open status.

Based upon how these two requests are processed, it can decide whether to go back to the open state or move to the closed state.

The next property is failure rate threshold. Here I have mentioned 50, 50 means 50%. With this what I am trying to communicate here is, if at least 50% of my requests are failed, then my circuit breaker pattern can move to the open state from the closed state.

And the last property we have here is wait duration in open state.

So here I have mentioned 10,000, which indicates 10,000 milliseconds.

So, with this configuration, my circuit breaker pattern, it is going to wait 10s whenever it tries to move to the half open state and allow the partial traffic.

You can see we have mentioned default.

That means these properties are applicable for all kind of circuit breakers that you are going to create

inside your microservice.

But in case if you want to go with different, different properties for different, different circuit breakers, then you need to use this circuit breaker name, which is **accountCircuitBreaker** and mention the same in the place of default.

package com.eazybytes.gatewayserver;

@SpringBootApplication

public class GatewayserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayserverApplication.class, args);

    }

    @Bean

    public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder){

        return routeLocatorBuilder.routes()

                .route(p -> p.path("/eazybank/accounts/\*\*")

                .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString())

**.circuitBreaker(config -> config.setName("accountsCircuitBreaker")))**

                .uri("lb://ACCOUNTS"))

                .route(p -> p.path("/eazybank/loans/\*\*")

                .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://LOANS"))

                .route(p -> p.path("/eazybank/cards/\*\*")

                .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://CARDS"))

                .build();

    }

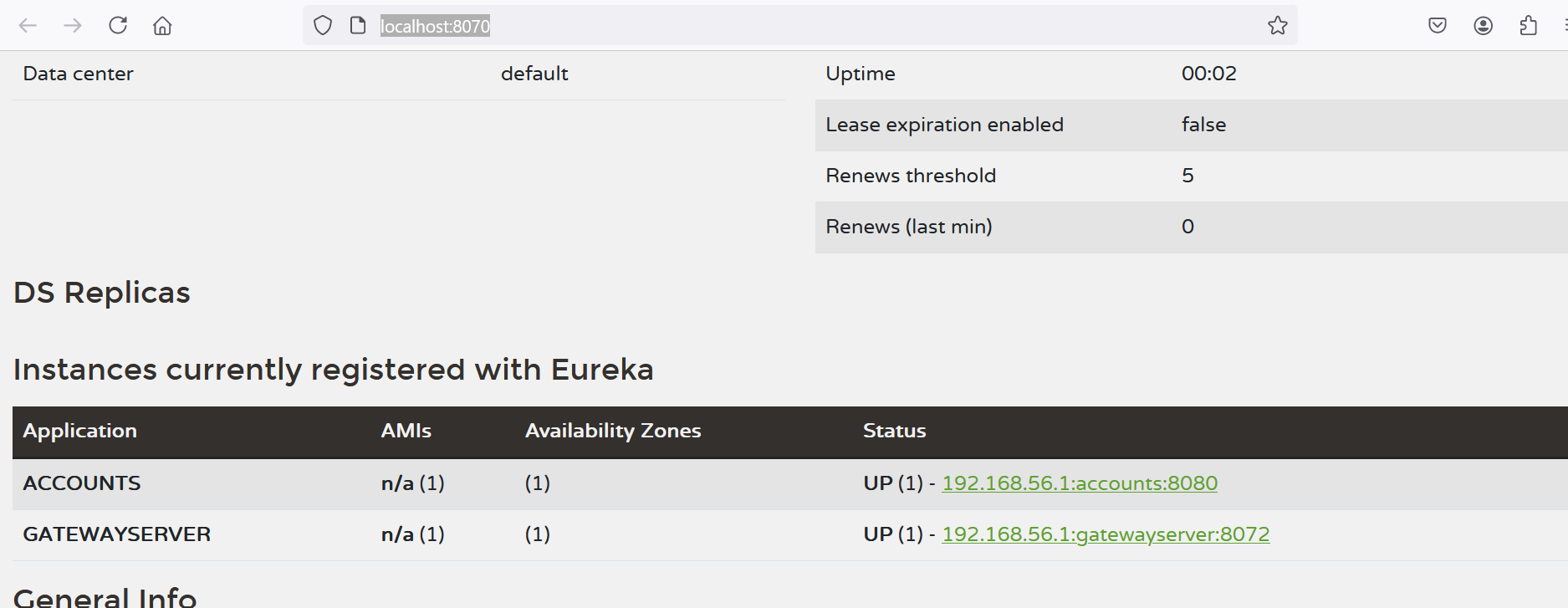
}

Start the microservices in sequence

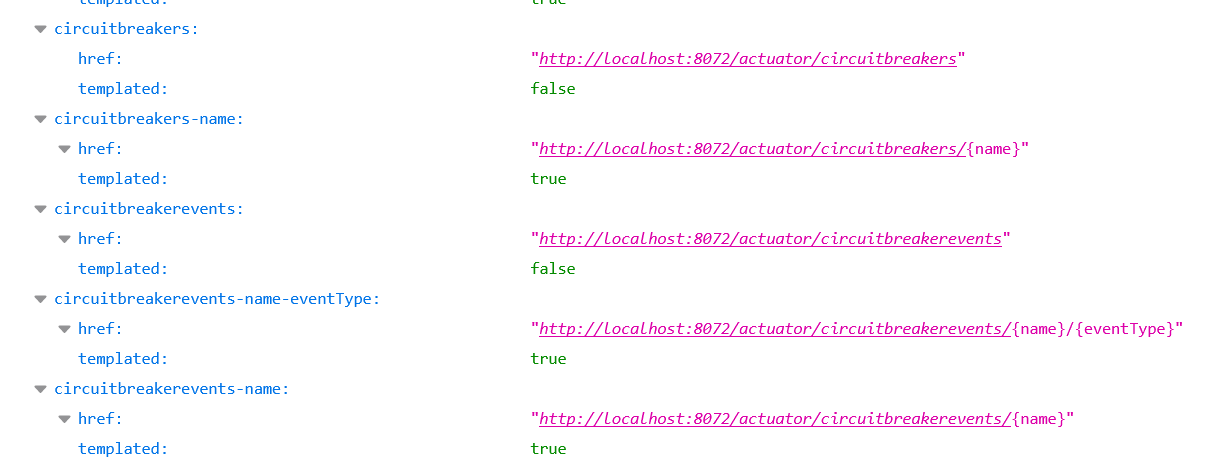
configserver, eureka, accounts, gatewayserver

Do not start loans and cards microservice to see the circuit breaker pattern in action

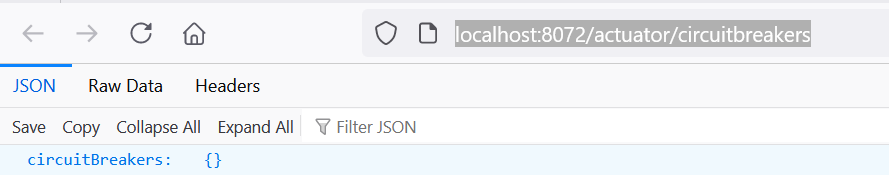
<http://localhost:8070/>



<http://localhost:8072/actuator>

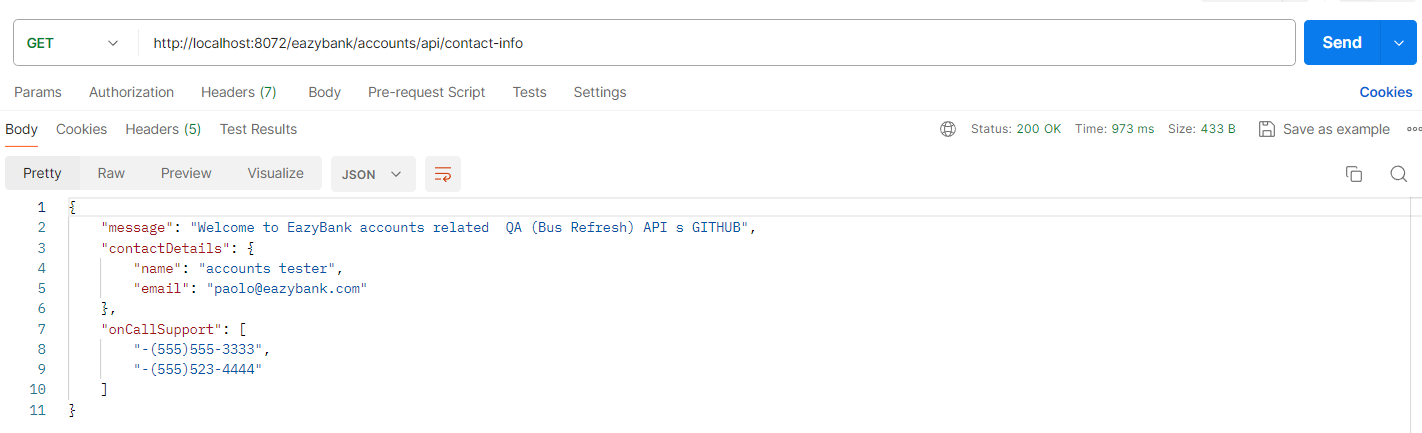


<http://localhost:8072/actuator/circuitbreakers>



Initially the circuit breakers are empty.  
Now when we start testing the accounts microservice with the help of Gateway server, all the circuit

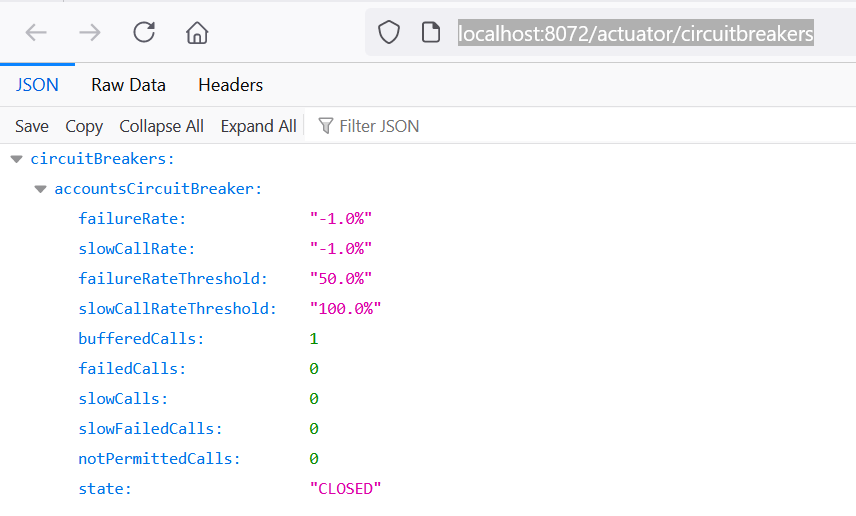
breaker related information is going to populate inside this page.



So, whenever we try to invoke these contact-info available inside the accounts microservice we

know it is going to return the contact details to whom we can reach out when there is an issue with

the accounts microservice. You can see here inside the response we got the message, contact details and on call support details.



If I try to refresh this page, you can see as of now I am able to see the details related to the

**accountsCircuitBreaker**.

This is the same name that we have configured inside the Gateway server.

As of now, the status of the circuit breaker is closed.

That is why the overall state it is showing as closed and the failure rate and slow call rate is -1,

which means there are no requests that are failed or processed with a slow call rate.

That is why you can see failed calls there are zero and slow calls is also zero.

So, there is some information about the circuit breaker.

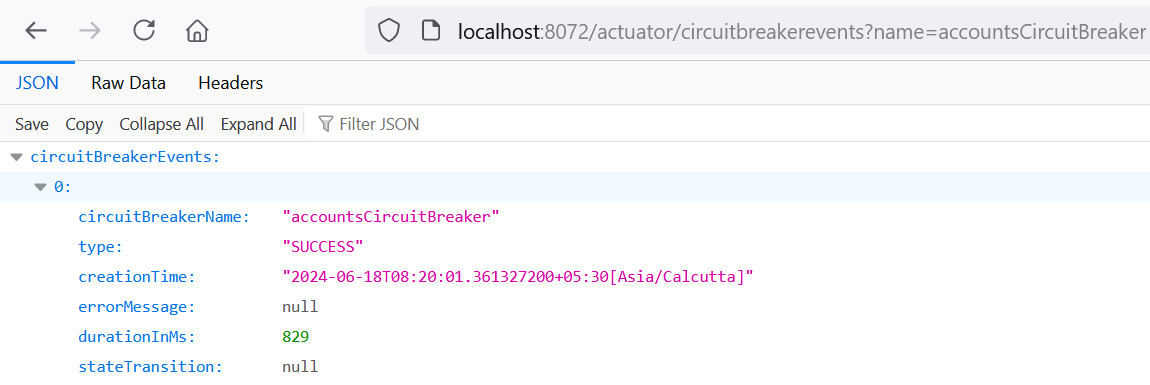
If you have multiple circuit breakers defined inside your application, you can obviously see all

them inside this path.

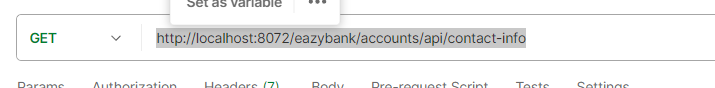
Now, we can also try to understand the events that are happening behind the scenes under the circuit breaker.

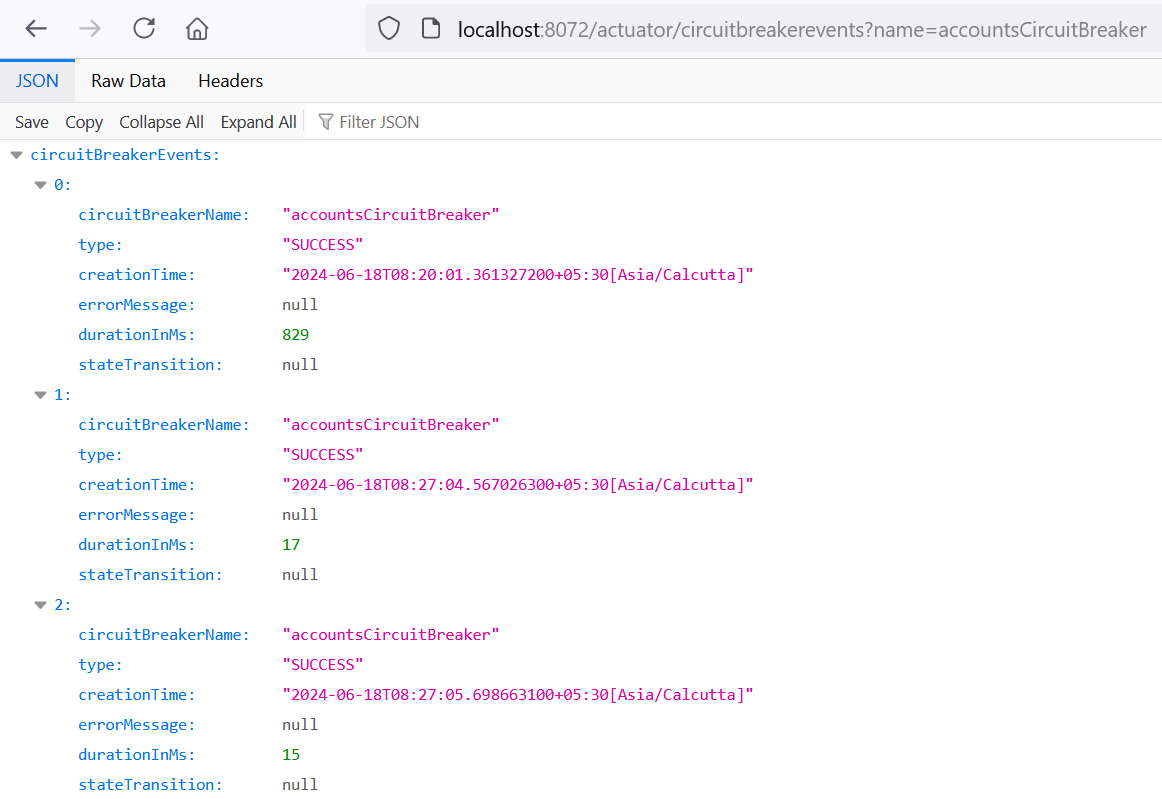
For the same, you can open this path which is **http://localhost:8072/actuator/circuitbreakerevents?name={circuit breaker name}** and what is your circuit breaker name?

http://localhost:8072/actuator/circuitbreakerevents?name=accountsCircuitBreaker

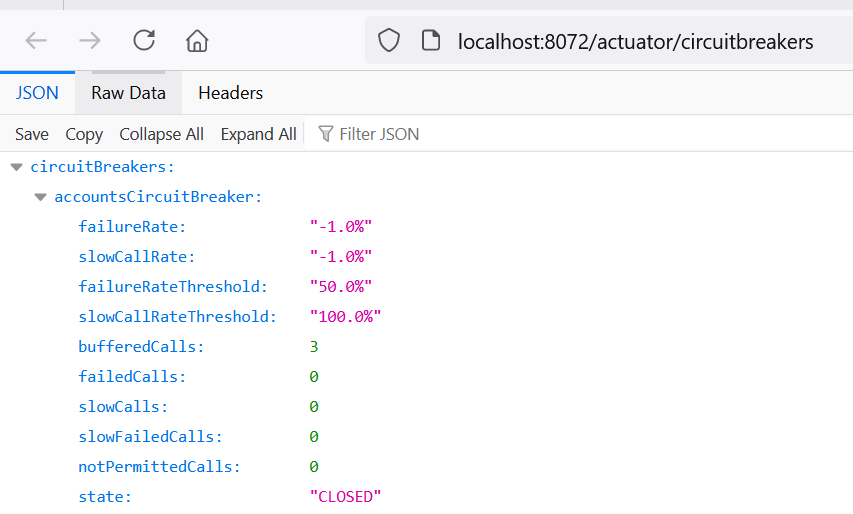


Invoke this 2 more times





So, this means your circuit breaker pattern is continuously monitoring all the invocations happening to your accounts microservice.



As of now you can see the buffered calls became three, and the overall state right now is still closed

because there is no failure rate, there is no slow call rat

Now to see the demo of circuit Breaker Pattern, I will go to the AccountsController.

Inside my AccountsController

I have written the implementation logic related to the contact-info API.

package com.eazybytes.accounts.controller;

@RestController

@RequestMapping(path = "/api", produces = "application/json")

@Validated

public class AccountsController {

....

@GetMapping("/contact-info")

public ResponseEntity<AccountsContactInfoDto> getContactInfo() {

return ResponseEntity

.status(HttpStatus.OK)

.body(accountsContactInfoDto);

}

}

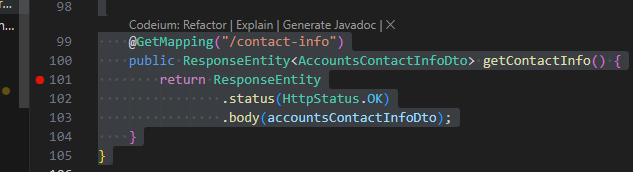
So, what I am going to do is just to mimic the pattern of slow response.

I am going to put a breakpoint here and I will never release this breakpoint.

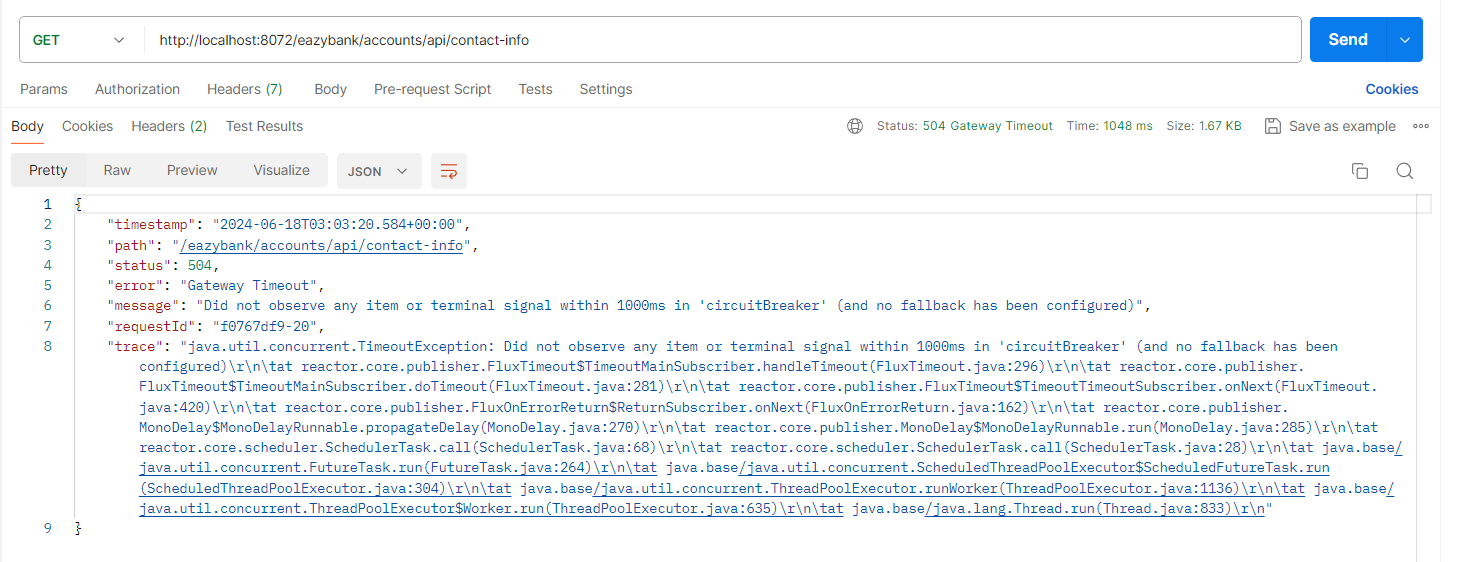
With that, always the request will come to this REST API, but it is not going to respond back to the

gateway server or to the client application.

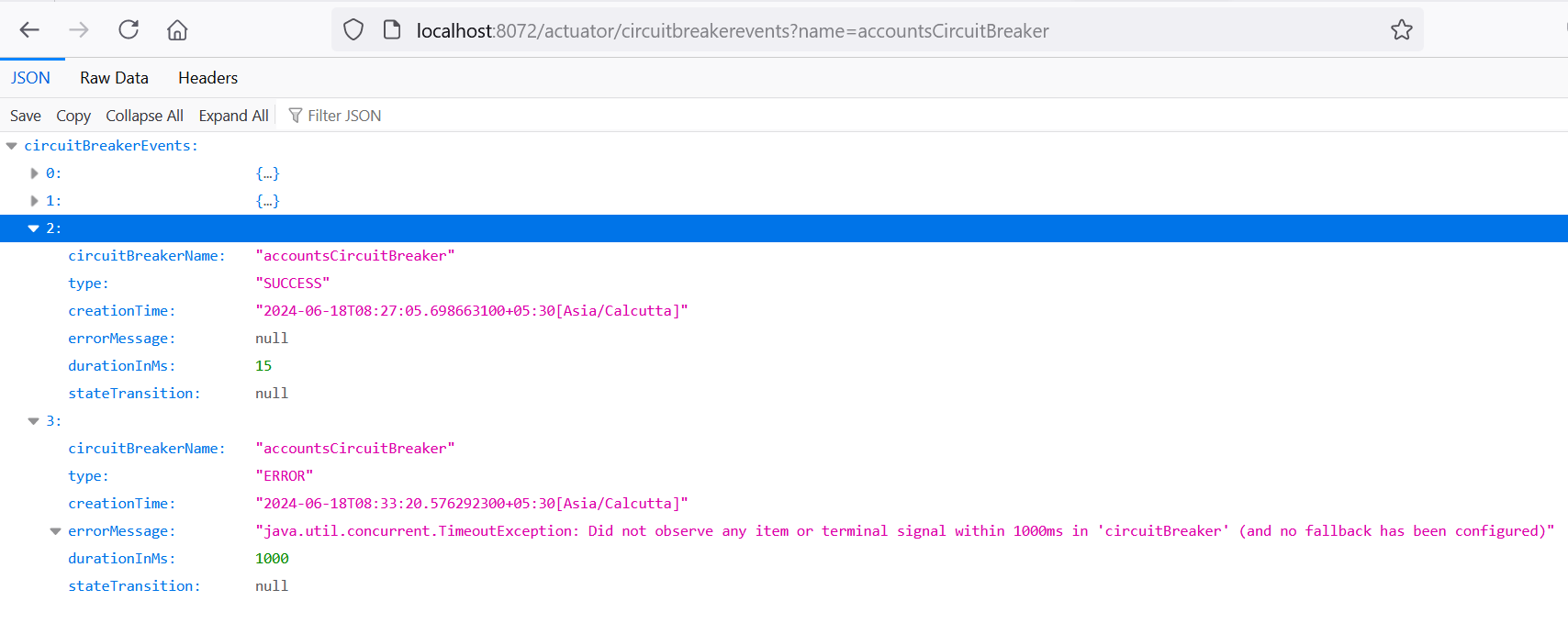
So, with this breakpoint now let us try to understand how the circuit breaker pattern is going to work.





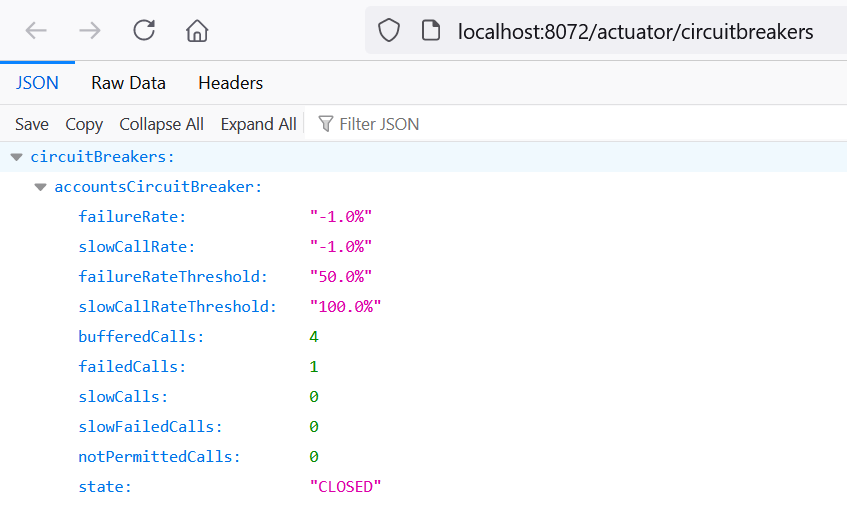


You can see after some time my gateway throws an error, which is 504 gateway timeouts.



Now if I try to refresh this page, you can see both the requests that we sent, it failed with the type error and this is the error message.

But right now, my circuit breaker pattern still will be in the closed status only because 50% of the calls never fail.

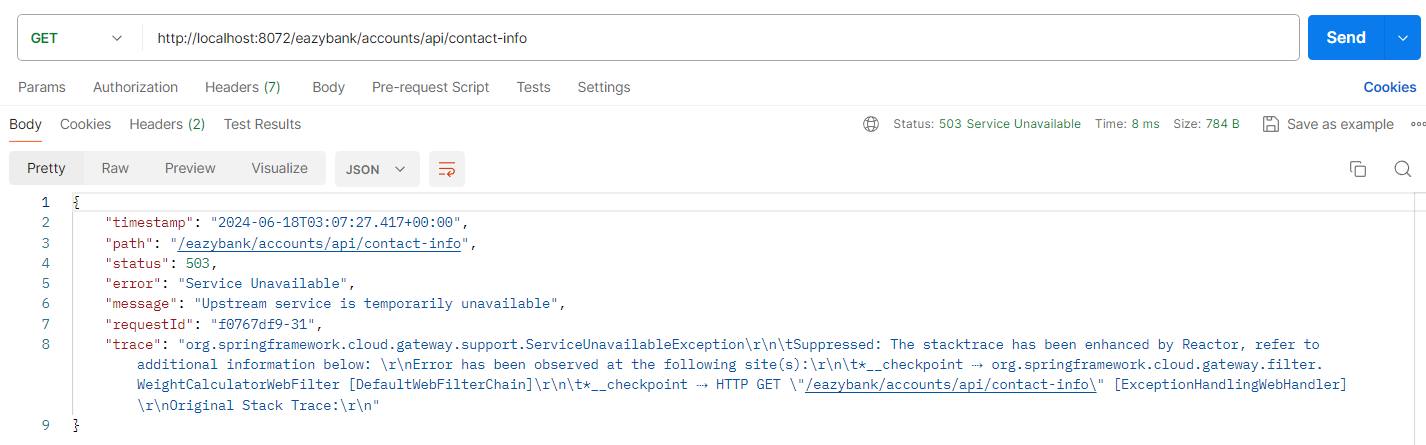


You can see as of now failed calls is only one. To change my circuit breaker pattern from closed state,

I need to send many requests to my contact-info and all of them will fail eventually my circuit breaker pattern will realize many of the requests are failing and I should move from close to open status.

So let me click on the send button multiple times.





So, I am trying to send this multiple times so that we will get this error every time.

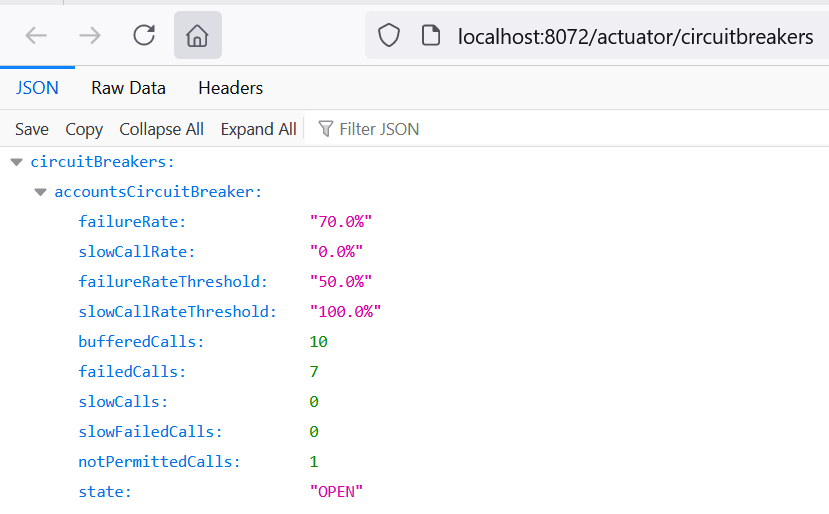
Like Gateway, you can keep observing this error like status as 504 and error is gateway timeout.

At some point of time, you can see the status change, which is 503 and the error right now is service

unavailable.

You can also see the message upstream service is temporarily unavailable.

This error is being thrown by the circuit breaker pattern.



We can also confirm the same by looking at the overall status.

Now the overall status is open.

If you can scroll down, we have many requests with the type as error.

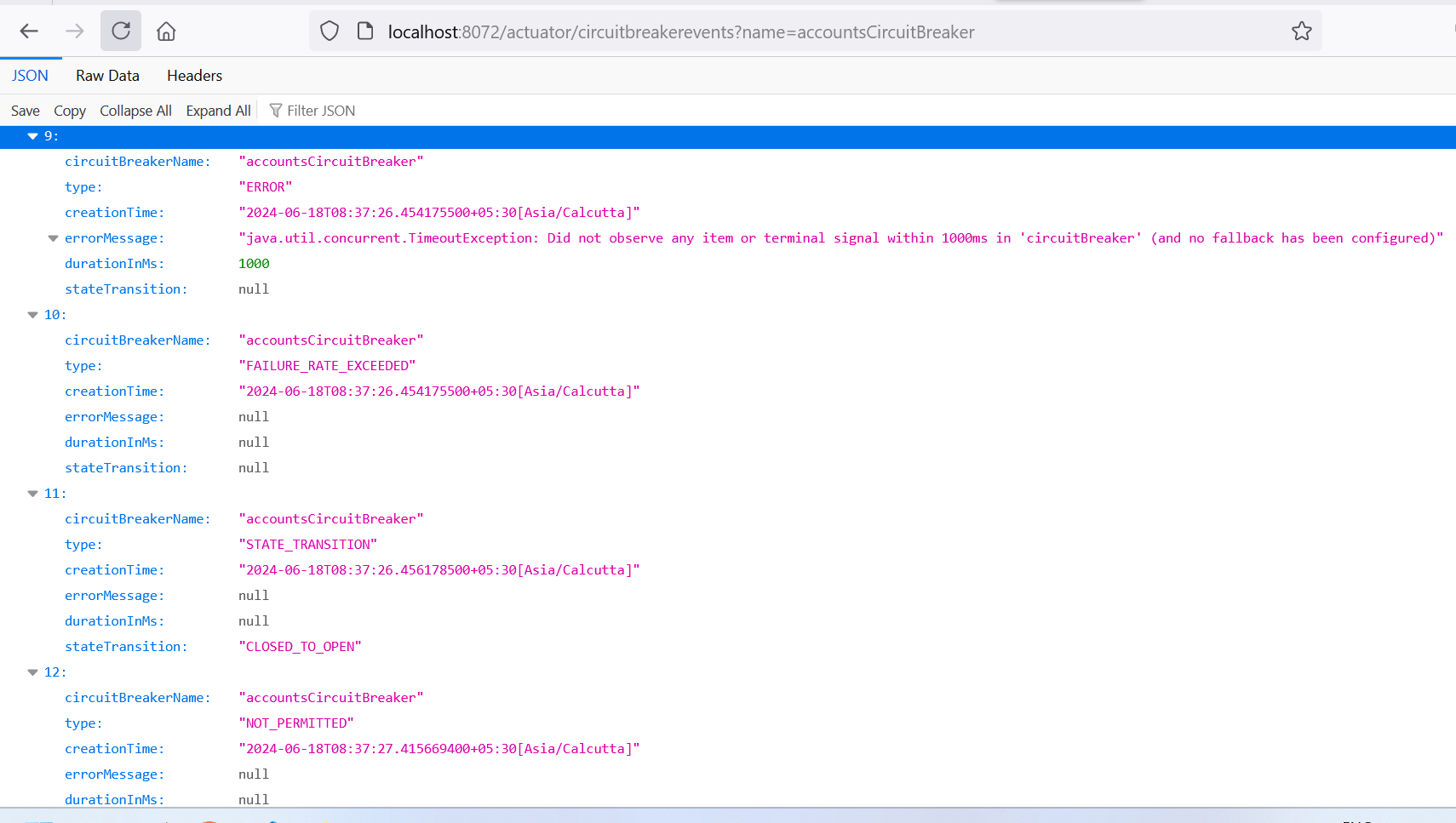
If you keep scrolling, you can see all these types error.

At some point of time my circuit breaker realized the failure rate exceeded.

That is why it is going to state the transition and the state transition is going to be close to open.

That is why my circuit breaker pattern will move to the open status.

Now, all the calls are not going to be permitted to my accounts microservice.



What advantages we are getting with this not permitted. Now my gateway server is not going

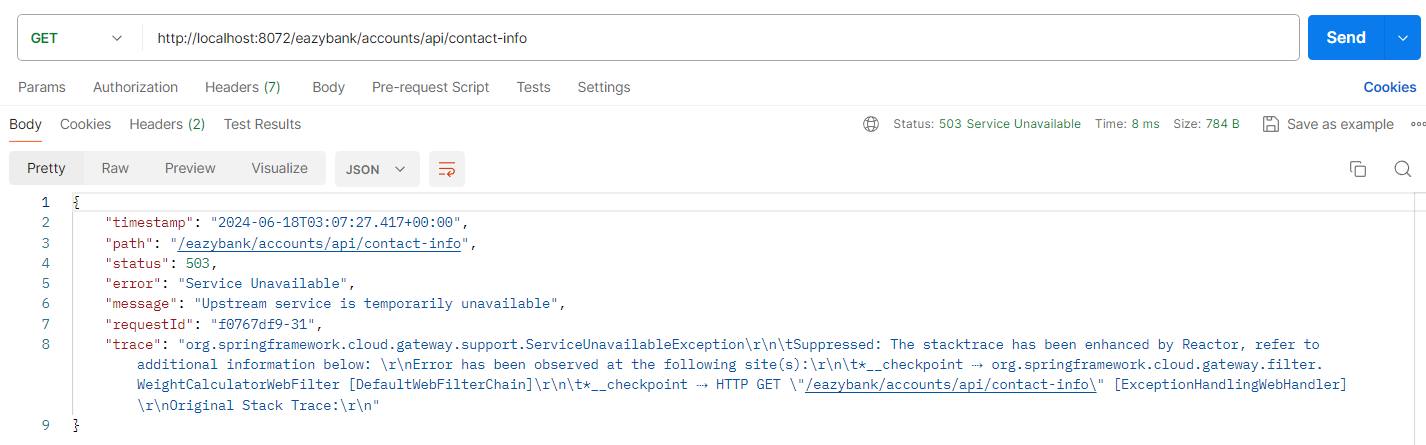
to waste its resources by invoking the accounts microservice my circuit breaker pattern which is sitting in the middle, always throwing an immediate error to the gateway server.

The upstream service is not available with that, my gateway server resources and threads will not be

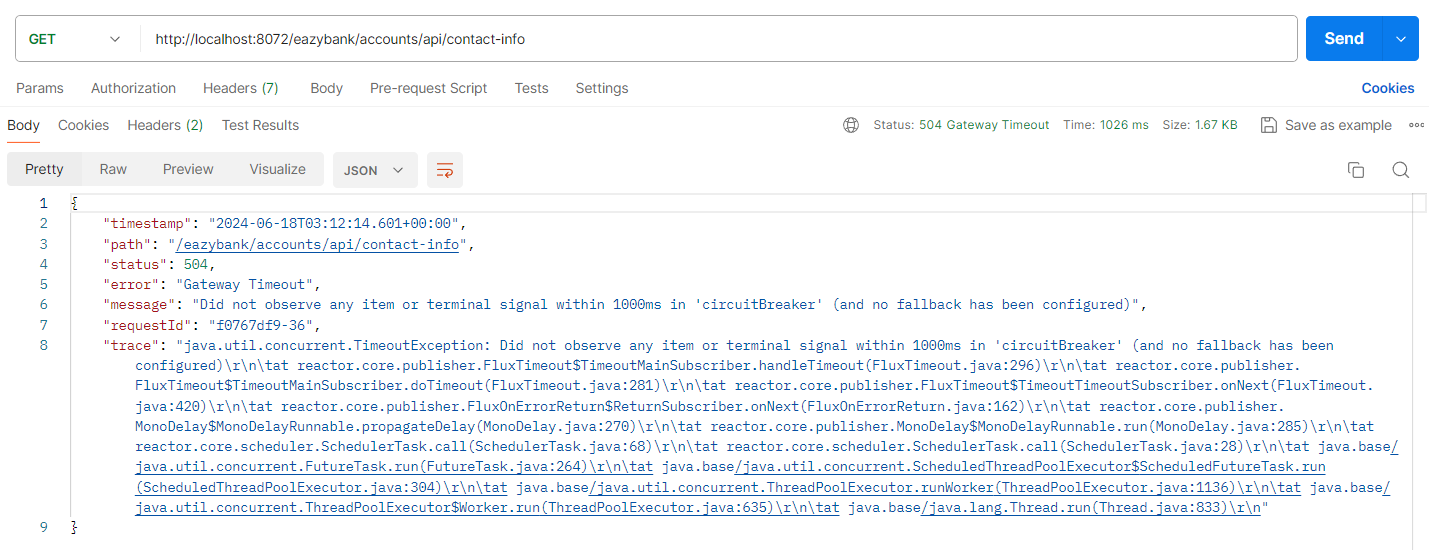
Blocked for a long time

Now we have waited for more than 10s.

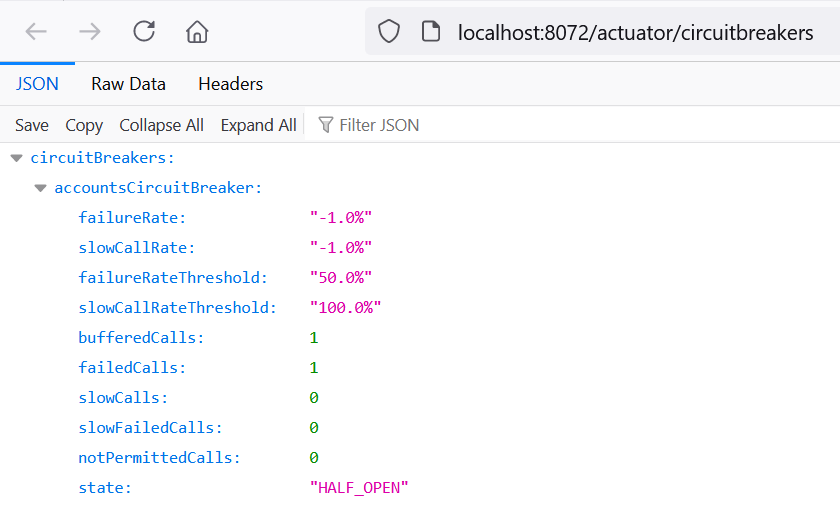
We can go to the postman and try to invoke a new request.

Before   


After 10 seconds

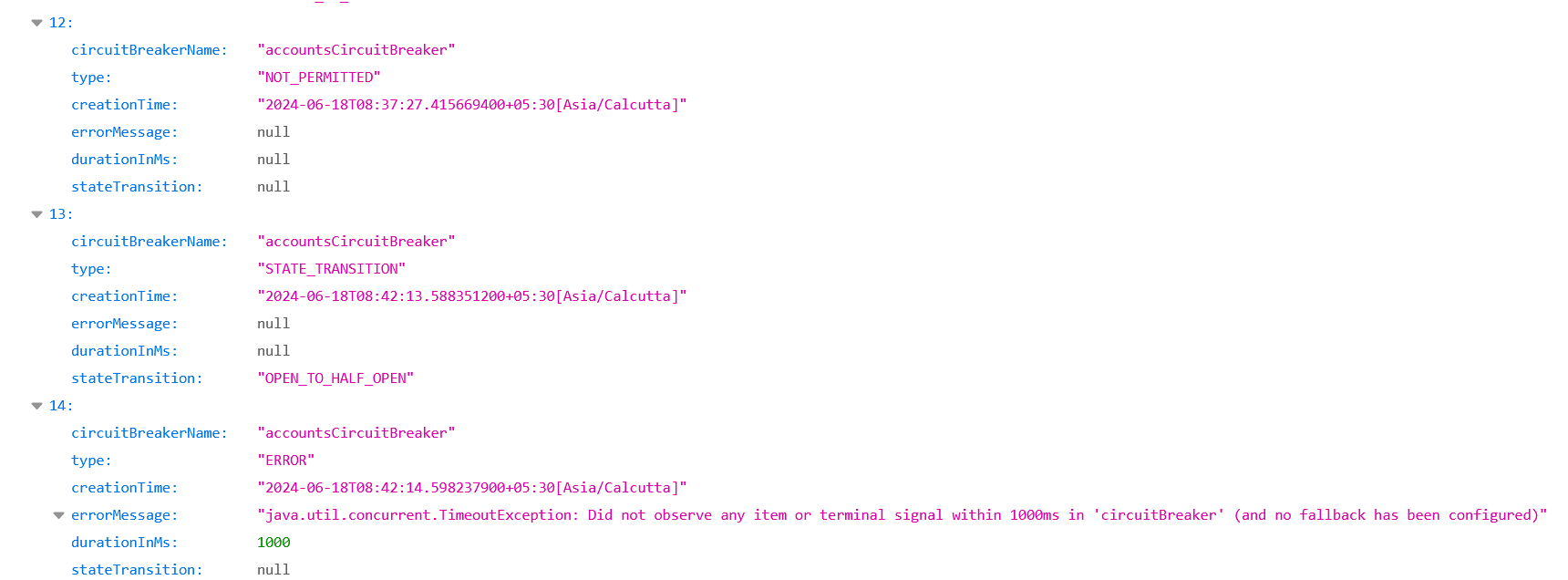


If I try to click on the send button this time, you can see the actual invocation to the accounts microservice happened. That's why we are getting time out.



You can see right now it is in the half open status and if you can refresh the other page, at last

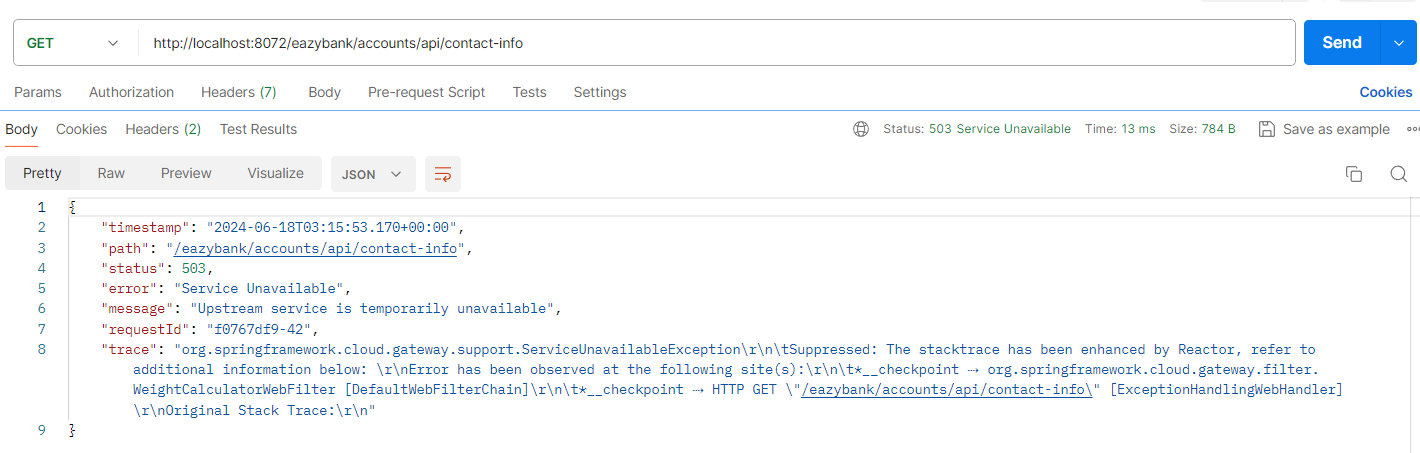
you will be able to see the events where the state transition happened from open to half open and during the half open we sent only one request and it is also failed.

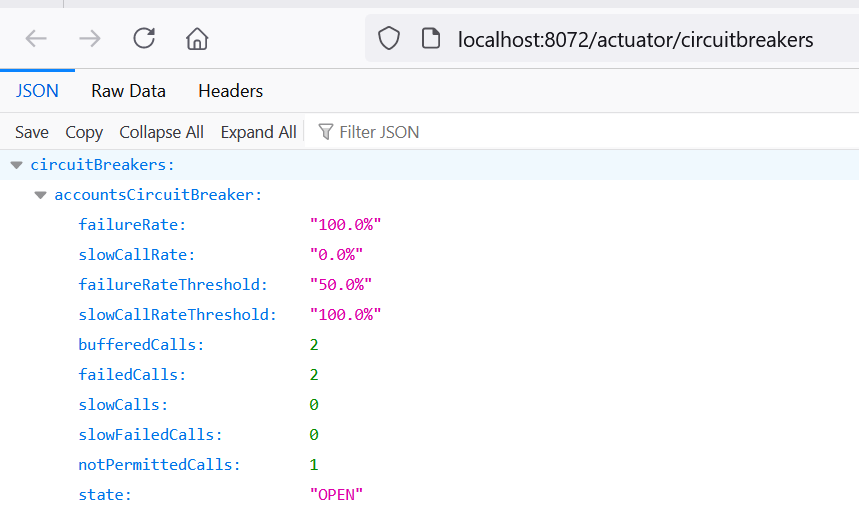


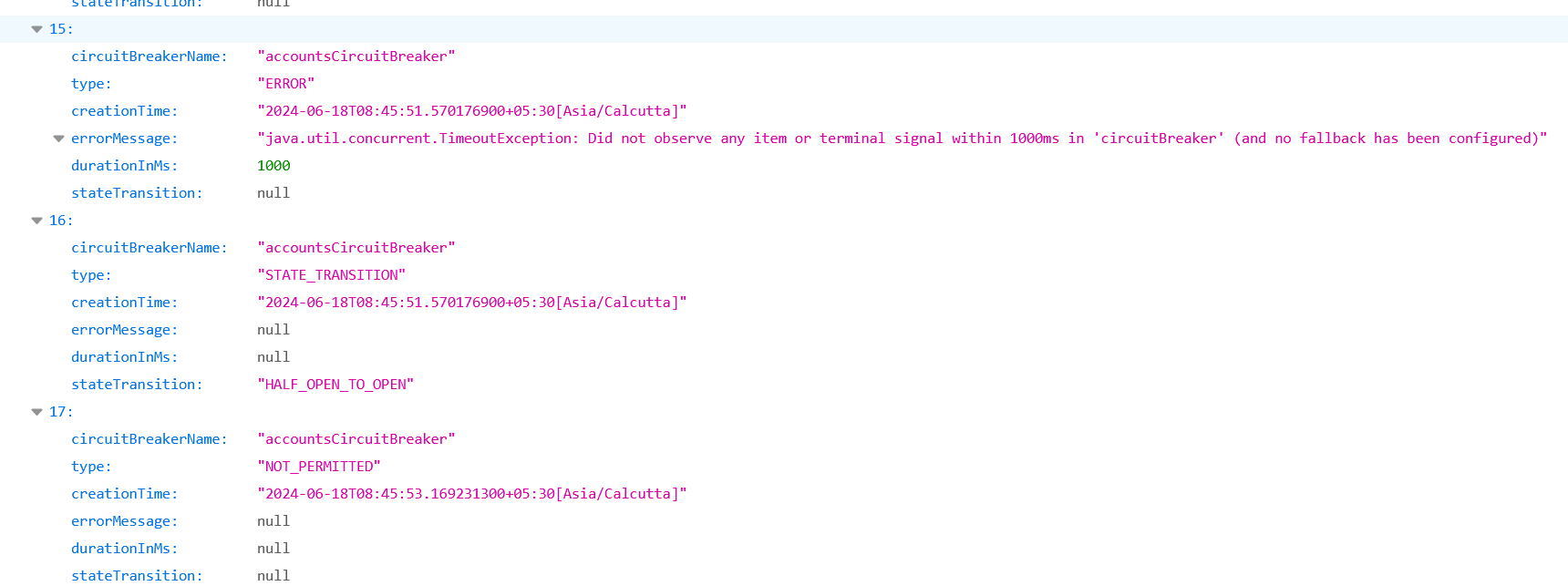
Now I will try to send the request multiple times and eventually my gateway server will go to the open status from the half open status. We can confirm the same by refreshing this page.

You can see as of now the state is open. If I try to refresh these events page.

Here also at last the state transition happened from half open to open and all the further requests are not permitted for the next 10s.



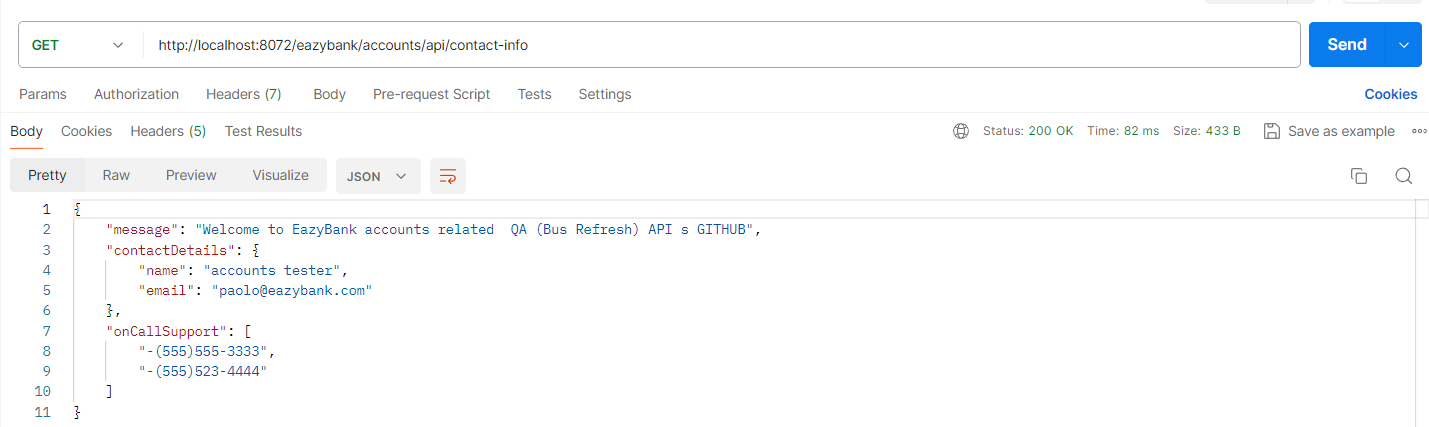




So now we have seen the demo of open and half open to show you the demo that circuit breaker is going to switch to the closed status if most of the request are being processed successfully, I can remove this breakpoint.

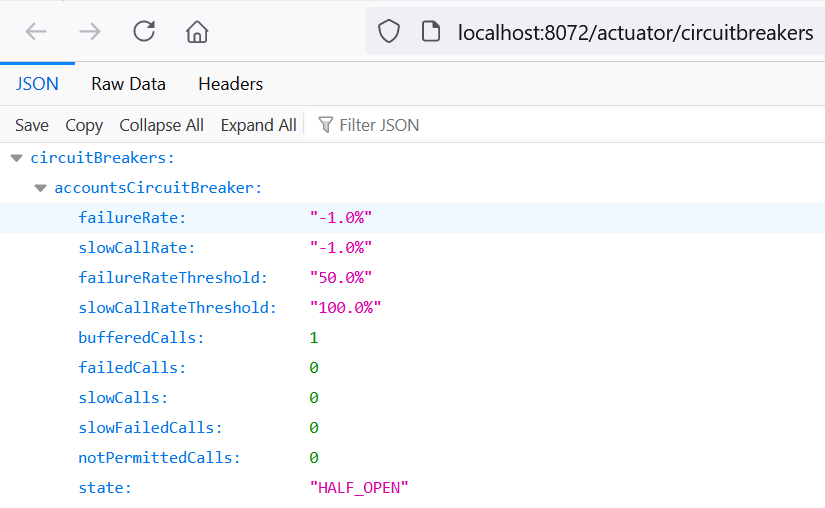
And here inside the postman I am going to click on the send button.

With that you can see we are getting successful response.

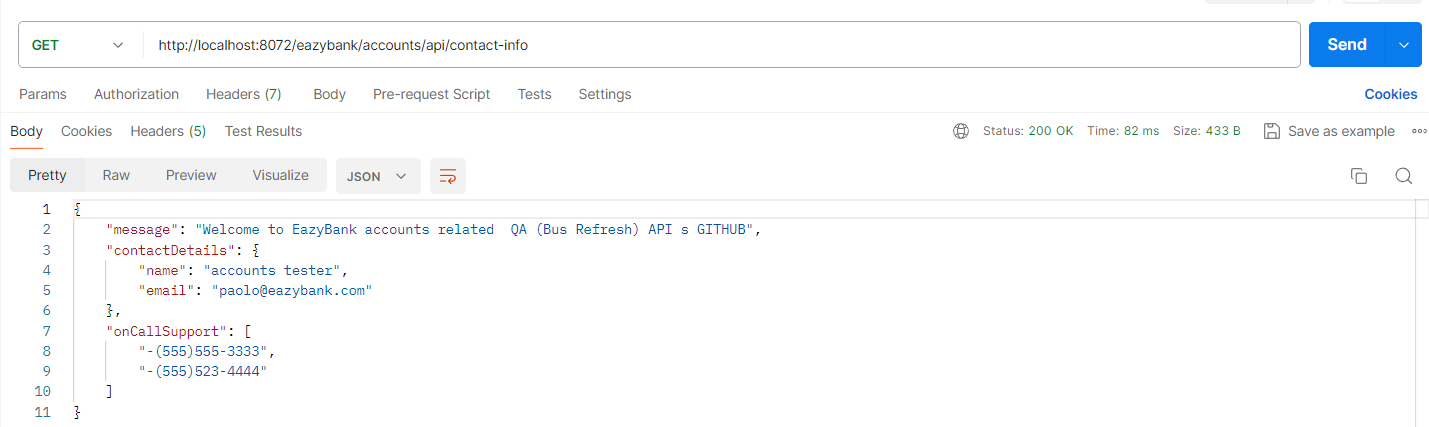


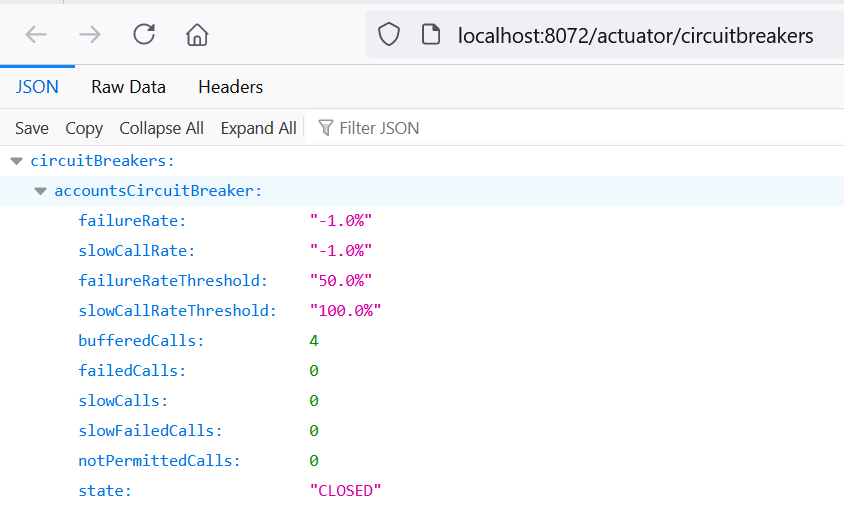
Now I can go to my actuator URLs here if I try to refresh this events page, you can see at some point

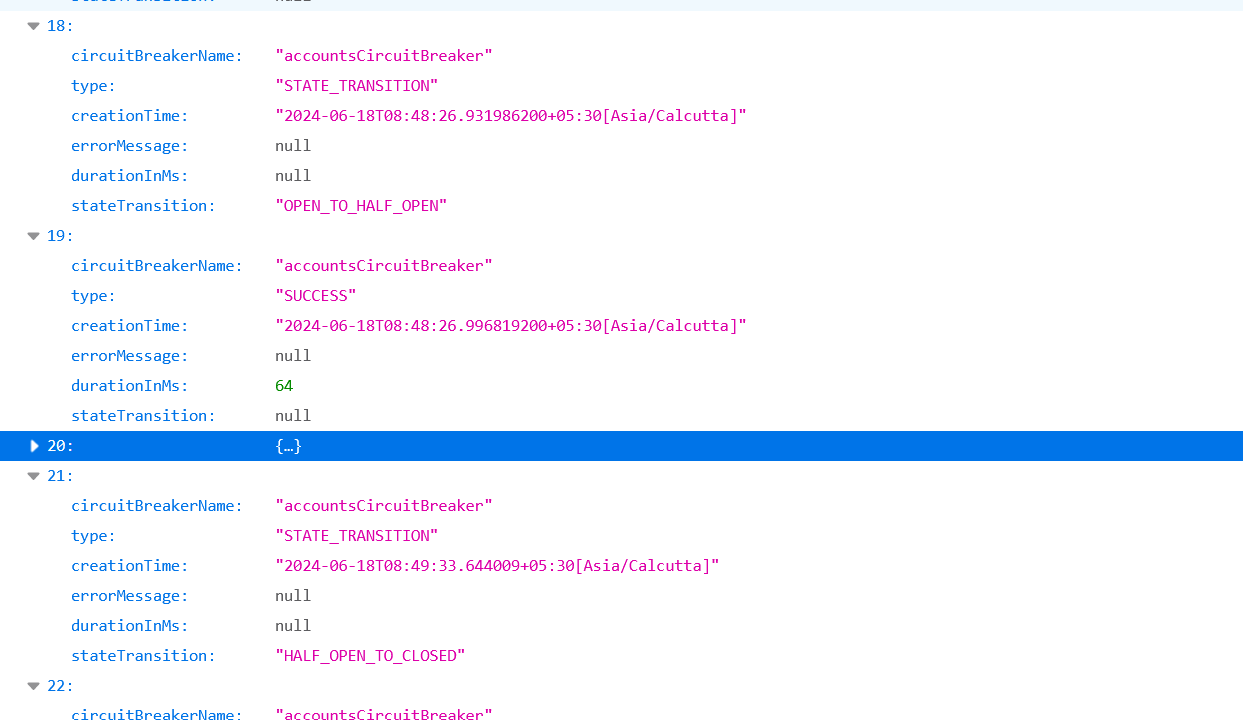
of time the state transition happened from half open to closed because all the requests are being processed successfully.



Send 10 few more requests







Implementing Circuit Breaker pattern in Gateway - Part 2

As of now, we have created a circuit breaker pattern inside the Gateway server, but it does not have

any fallback mechanism.

Since we do not have any fallback mechanism inside the response, we are throwing some runtime exception details like **ServiceUnavailable** or **GatewayTimeout** exceptions. In real business applications throwing **RuntimeExceptions** to the client applications or to the UI applications is not a valid approach.

That is why we need to have some fallback mechanism and inside this fallback mechanism we can write some logic where we can send some message to the client applications, which is going to make sense for them.

As of now, you can see we have this filters package gatewayserver project.

Very similarly, I am going to create a new package inside my gateway server with the name controller

and inside this controller I am going to create a new controller class and the class name is going to be FallbackController.

package com.eazybytes.gatewayserver.controller;

import reactor.core.publisher.Mono;

@RestController

public class FallbackController {

    @RequestMapping("/contactSupport")

    public Mono<String> contactSupport() {

        return Mono.just("An error occurred. Please try after some time or contact support team!!!");

    }

}

So, this is a simple business logic that I have written as a fallback mechanism.

In your real projects you may have some complex fallback requirements like triggering an email to the support team or sending some default response, so it is up to your client requirements and accordingly you can implement your own logic.

As of now, I am trying to send a simple business error from my fallback REST API since we are trying

to build a REST API inside the Gateway server which is implemented on top of spring reactive, we need to make sure we are wrapping the return String with the help of mono.

And very similarly here also we need to use Mono.just() and to this just() we need to pass what

is a String that we want to send.

As a next step, we need to integrate this REST API into our circuit breaker pattern.

For the same, let's go to the Gateway server application just after the set name.

We can also invoke setFallbackUri().

And to this fallbackUri I'm going to send what is a fallback REST API URL details.

So here I'm going to mention forward:/contactsupport.

package com.eazybytes.gatewayserver;

@SpringBootApplication

public class GatewayserverApplication {

…

    @Bean

    public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder){

        return routeLocatorBuilder.routes()

                .route(p -> p.path("/eazybank/accounts/\*\*")

                .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString())

                               .circuitBreaker(config -> config.setName("accountsCircuitBreaker")

**.setFallbackUri("forward:/contactSupport")))**

                .uri("lb://ACCOUNTS"))

                .route(p -> p.path("/eazybank/loans/\*\*")

                .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://LOANS"))

                .route(p -> p.path("/eazybank/cards/\*\*")

                .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://CARDS"))

                .build();

    }

}

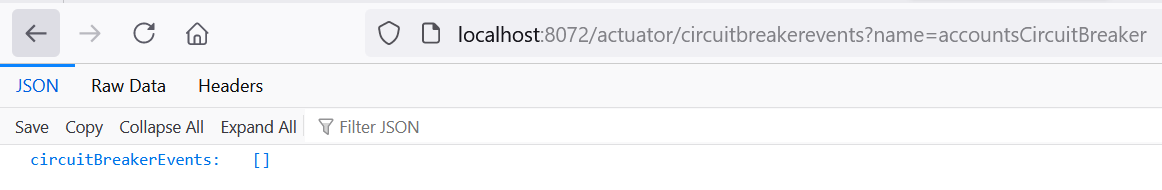
With this, I am telling to my circuit breaker pattern. Whenever there is an exception happens, please invoke this fallback by forwarding the request to the contact support.

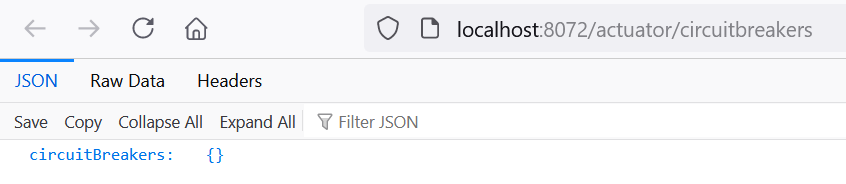
With this we have some fallback mechanism.

Start the microservices in sequence

configserver, eureka, accounts, gatewayserver

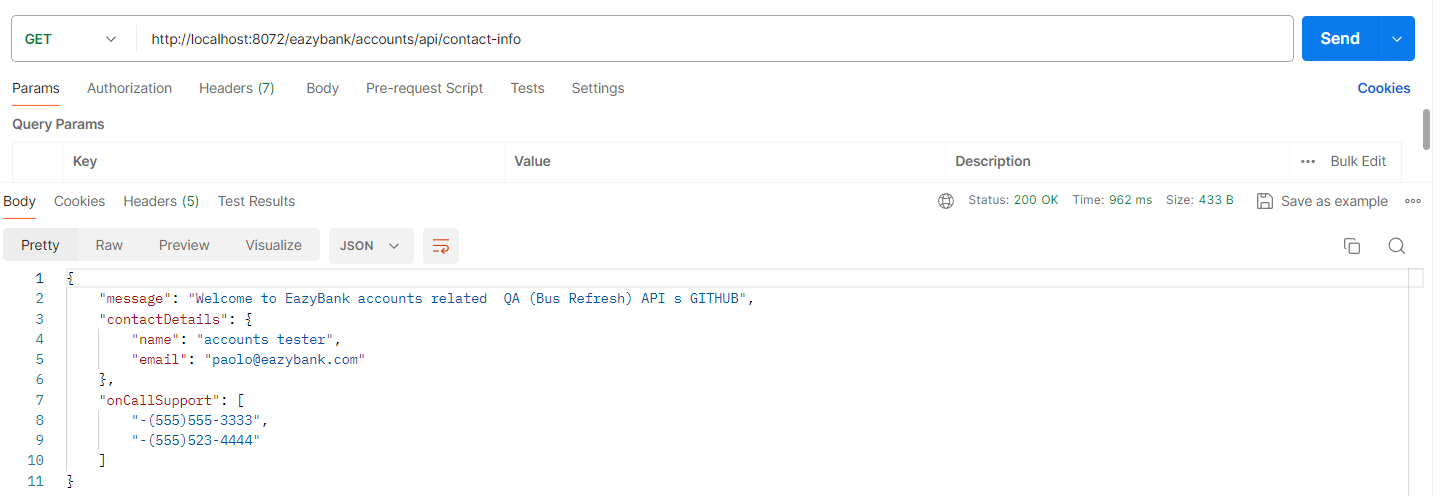
Do not start loans and cards microservice to see the circuit breaker pattern in action

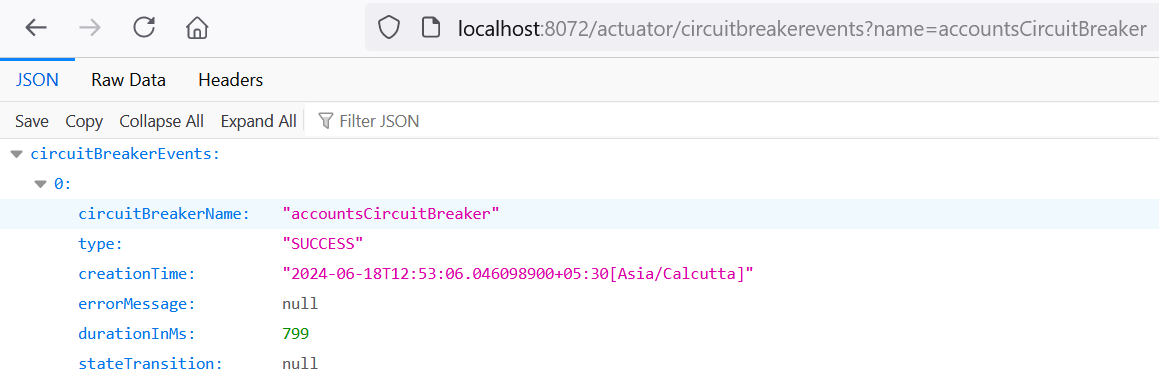


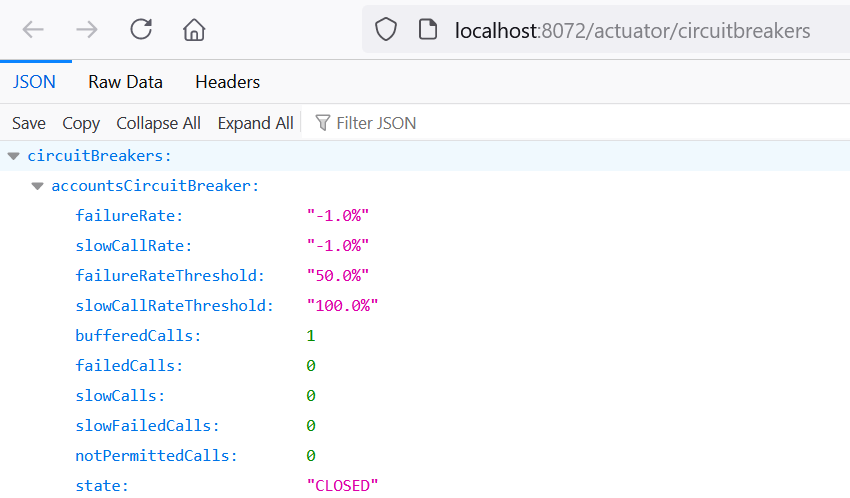


First, I'll try to click on the send button.

You'll get a happy response so everyone is happy. Inside the actuator you can see as of now the overall status is closed and if I try to refresh this page, there is one request received after my gateway server is restarted and the request is also processed successfully.







You will get a happy response so everyone is happy. Inside the actuator you can see as of now the overall status is closed and if I try to refresh this page, there is one request received after my gateway server is restarted and the request is also processed successfully.

Now I will put a breakpoint inside my accounts controller and try to mimic the scenario of slow response from the accounts microservice.

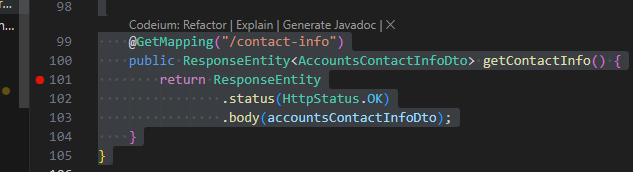
So, what I am going to do is just to mimic the pattern of slow response.

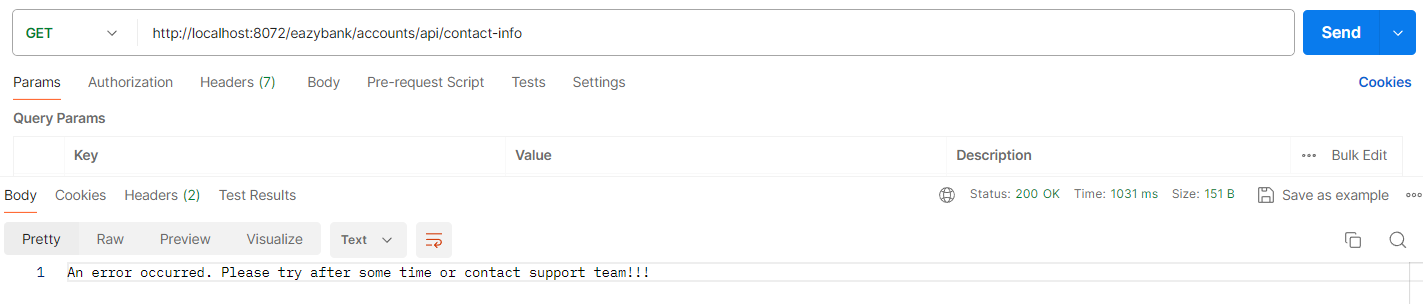
I am going to put a breakpoint here and I will never release this breakpoint.

With that, always the request will come to this REST API, but it is not going to respond back to the

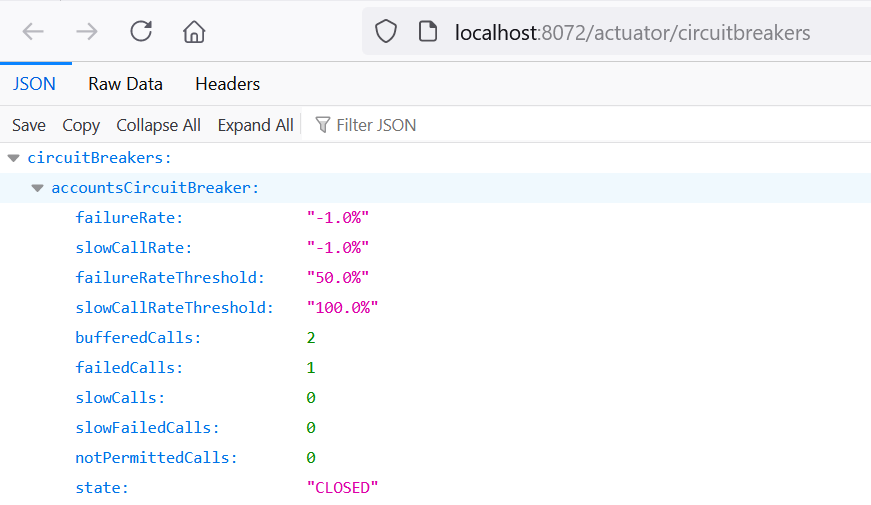
gateway server or to the client application.

So, with this breakpoint now let us try to understand how the circuit breaker pattern is going to work.









I will go to the postman inside my postman you can see I am getting the response from my fallback mechanism, which is an error occurred please try after some time or contact support team. With this my client applications will never know what is happening behind the scenes.

They will never receive gateway timeout or they will never receive upstream.

Service is not available.

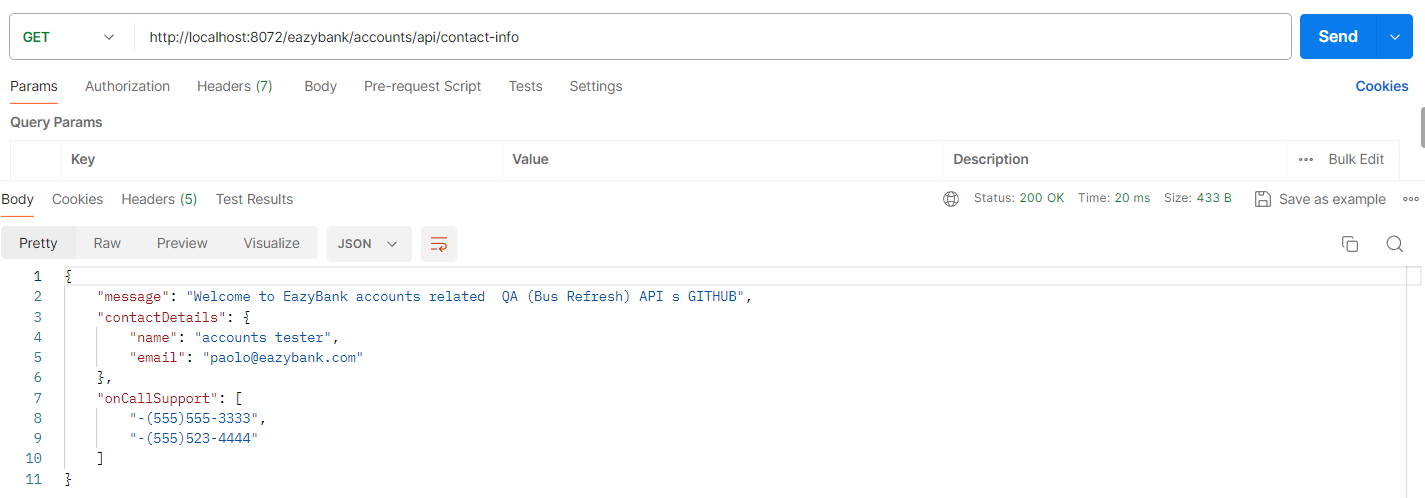
So, it is up to you what kind of fallback mechanism you want to write inside your fallback REST API.

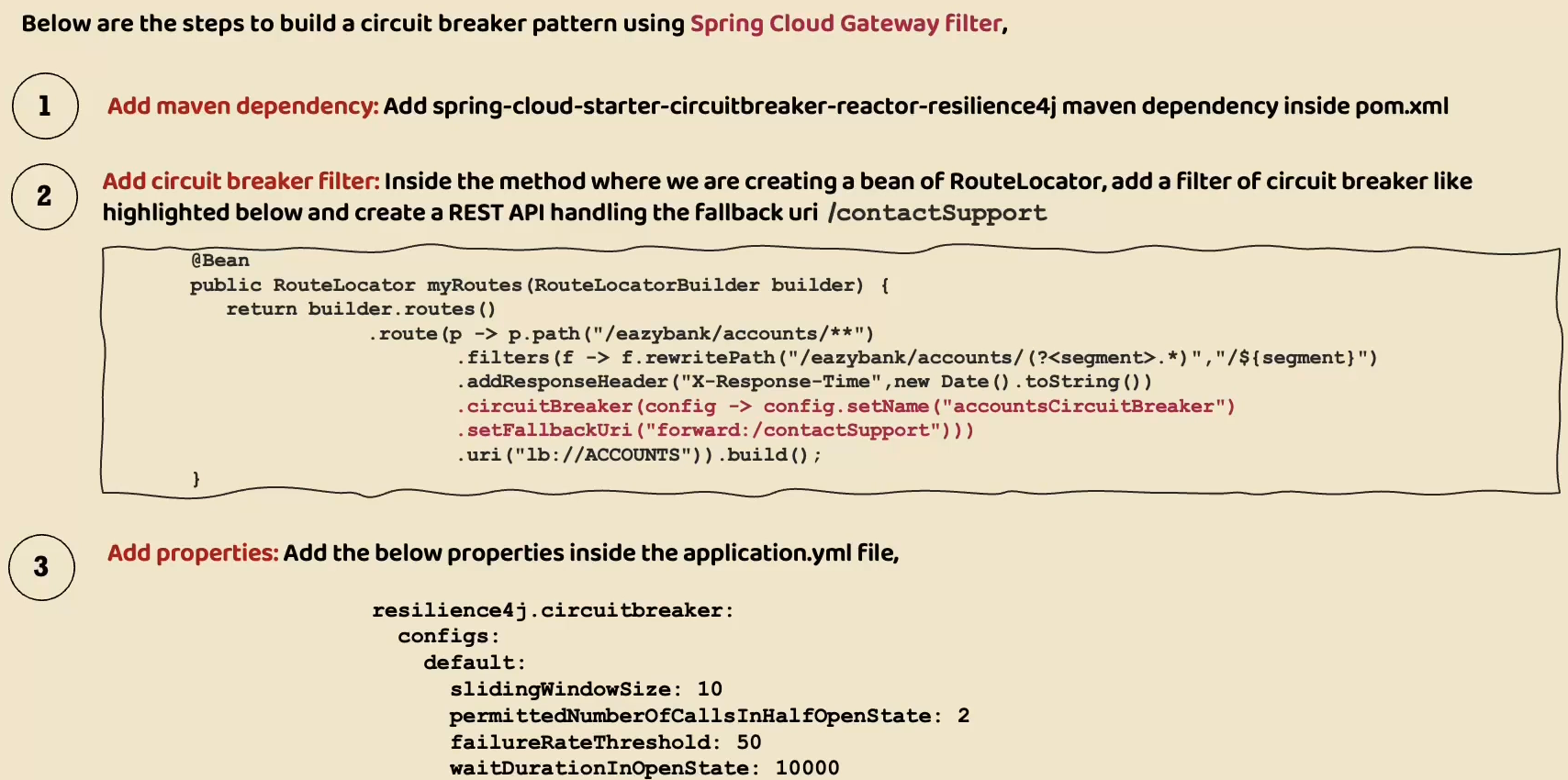
And this fallback will never get invoked if the request is being processed successfully.

So let me release this breakpoint and release the previous requests that I have.

Now I am going to click on the send button.

You can see in the success scenario; the fallback is never coming into picture.





Implementing Circuit Breaker pattern with Feign Client - Part 1

Let us try to implement circuit breaker pattern inside accounts microservice.

Inside our accounts microservice, if you can recall, there is a REST API with the name **fetchCustomerDetails**. As part of this REST API.

My accounts microservice is going to invoke cards, microservice and loans microservice.

But if one of the dependent services like loans and cards microservice is responding very slowly or if

they are completely down or there are some network issues. In such scenarios like we discussed previously, it is going to have a ripple effect on my accounts microservice and from my account microservice to Gateway server.

Our accounts microservice is leveraging feign client to invoke loans and cards microservice, that is why we need to see if there is any integration between feign client and circuit breaker is available,

which we can leverage with less efforts or less configurations inside accounts microservice. For the same inside the official website of spring.io, we can click on this Spring cloud project.

Under this spring cloud project, we can look for Spring Cloud, open feign.

<https://docs.spring.io/spring-cloud-openfeign/docs/3.1.9/reference/html/#spring-cloud-feign-circuitbreaker>

If Spring Cloud circuit breaker is on the class path and if we mention

spring.cloud.openfeign.circuitbreaker.enabled=true

inside our application.properties Feign will wrap all the methods with a circuit breaker.

Add dependency inside D:\Experiments\Microservices\sb-bank-application\accounts\pom.xml

<dependency>

  <groupId>org.springframework.cloud</groupId>

  <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>

</dependency>

Since we have not built accounts microservice based upon spring reactor we should not use circuitbreaker-reactor

D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties

…  
spring.cloud.openfeign.circuitbreaker.enabled=true

…  
  
Apart from these changes, we should also mention other properties inside the application.properties , just like how we mentioned inside the gateway server.  
D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties

resilience4j.circuitbreaker.configs.default.slidingWindowSize=10

resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=2

resilience4j.circuitbreaker.configs.default.failureRateThreshold=50

resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=10000

package com.eazybytes.accounts.service.client;

@Component

public class CardsFallback implements CardsFeignClient{

    @Override

    public ResponseEntity<CardsDto> fetchCardDetails(String correlationId, String mobileNumber) {

        return null;

    }

}

package com.eazybytes.accounts.service.client;

@Component

public class LoansFallback implements LoansFeignClient{

    @Override

    public ResponseEntity<LoansDto> fetchLoanDetails(String correlationId, String mobileNumber) {

        return null;

    }

}

Logic is simply to return the null value from this fallback method.

Instead of throwing a RuntimeException to the client applications or to the gateway server, I am going to send the null information related to loans.

And with this at least my client applications will receive accounts and cards microservice related

information.

And if you don't have this fallback mechanism, even if two microservices are working but loans is not

working, the overall response is going to be a RuntimeException.

So, we are trying to avoid that with the help of this fallback mechanism and circuit breaker pattern.

Here also, I am going to return null whenever my cards microservice is having some issues, but in your real projects you can write your own business logic or maybe you want to send some values from the cache or maybe you want to read the details from a different database.

So, it is up to you.

package com.eazybytes.accounts.service.client;

**@FeignClient(name = "loans", fallback = LoansFallback.class)**

public interface LoansFeignClient {

….

}

package com.eazybytes.accounts.service.client;

**@FeignClient(name = "cards", fallback = CardsFallback.class)**

public interface CardsFeignClient {

…

}

I am going to mention **name** parameter just before loans and post that I am going to invoke a new parameter which is **fallback** and to this fallback we need to mention the class which is LoansFallback.class and CardsFallback.class respectively.

So, whenever there is an exception, we are executing a fallback mechanism which will return a null value. So, let us go to the service layer and check if we need to perform any null checks

package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class CustomerServiceImpl implements ICustomerService {

…

    @Override

    public CustomerDetailsDto fetchCustomerDetails(String mobileNumber, String correlationId) {

…

        ResponseEntity<LoansDto> loansDtoResponseEntity = loansFeignClient.fetchLoanDetails(correlationId, mobileNumber);

        customerDetailsDto.setLoansDto(loansDtoResponseEntity.getBody());

        ResponseEntity<CardsDto> cardsDtoResponseEntity = cardsFeignClient.fetchCardDetails(correlationId, mobileNumber);

        customerDetailsDto.setCardsDto(cardsDtoResponseEntity.getBody());

        return customerDetailsDto;

    }

}

Here, whenever we are trying to invoke the loansFeignClient or/and cardsFeignClient, we are getting this entity from this entity. We are trying to invoke the getBody(). But with the new code that we have written, there are some scenarios where this can come as null, especially with the fallback code.

So that is why we need to make sure we are putting some null check on top of this code.

If it is not null only then please execute this line of code.

package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class CustomerServiceImpl implements ICustomerService {

….

        ResponseEntity<LoansDto> loansDtoResponseEntity = loansFeignClient.fetchLoanDetails(correlationId, mobileNumber);

**if(null != loansDtoResponseEntity)**

        customerDetailsDto.setLoansDto(loansDtoResponseEntity.getBody());

        ResponseEntity<CardsDto> cardsDtoResponseEntity = cardsFeignClient.fetchCardDetails(correlationId, mobileNumber);

**if(null != cardsDtoResponseEntity)**

        customerDetailsDto.setCardsDto(cardsDtoResponseEntity.getBody());

…

    }

}

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