Microservice **with** Spring Boot

short line

Your Name  
4th September, 20XX

# Microservice Characteristics

Features for MS

1. Loosely coupled
2. Developed independently
3. Deployed independently
4. Scaled independently
5. Should talk to each other

* Synchronously- HTTP call
* Asynchronous- Message broker Kafka/RabbitMQ

Key Component

* API gateway
* Service registry
* Config server
* Distributed tracing

# Configuration Management

## Config Server

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

@SpringBootApplication

@EnableConfigServer

public class ConfigserverApplication {

public static void main(String[] args) {

SpringApplication.*run*(ConfigserverApplication.class, args);

}

}

spring:

application:

name: "configserver"

profiles:

active: native

#active: git

cloud:

config:

server:

native:

search-locations: "classpath:/config"

# search-locations: "file:///Users//eazybytes//Documents//config"

# git:

# uri: "https://github.com/eazybytes/eazybytes-config.git"

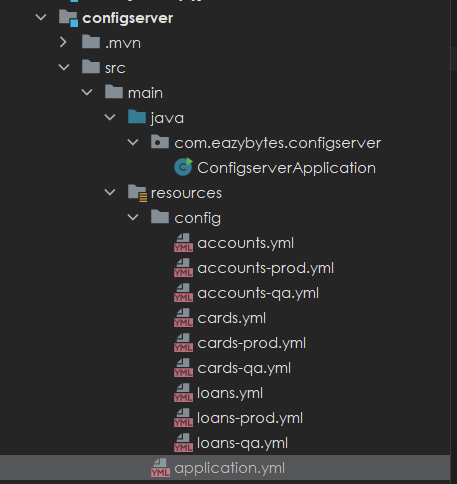
# default-label: main

# timeout: 5

# clone-on-start: true

# force-pull: true

Individual application name with application.name and the profiles



Client

<dependency>

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

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

server:

port: 8080

spring:

application:

name: "accounts"

profiles:

active: "prod" // Default profile can be overwritten by command line argument, spring.profile.active= “qa”

datasource:

url: jdbc:h2:mem:testdb

driverClassName: org.h2.Driver

username: sa

password: ''

h2:

console:

enabled: true

jpa:

database-platform: org.hibernate.dialect.H2Dialect

hibernate:

ddl-auto: update

show-sql: true

config:

import: "optional:configserver:http://localhost:8071/" //Config server location set at optional

rabbitmq:

host: "localhost"

port: 5672

username: "guest"

password: "guest"

## Encryption Config server

Encrypt sensitive element in Github

* Set an encrypted key in config server properties

encrypt:

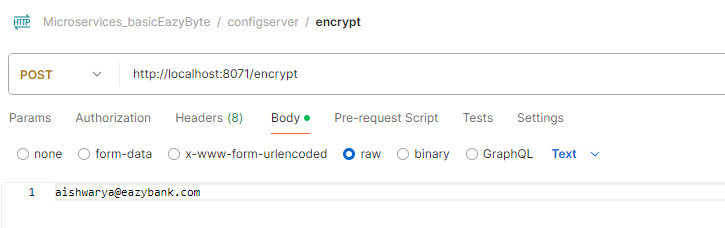
key: "45D81EC1EF61DF9AD8D3E5BB397F9"

* After we set key config server will provide encrypt and decrypt URI

<http://localhost:8071/encrypt>

http://localhost:8071/decrypt

* With encrypt API we send the data we want to encrypt say email



* Once we get the encrypted value we can set it in Github like below. To let config server know this is encrypted we pass keyword cipher

accounts:

message: "Hey, welcome to EazyBank accounts related webhook APIs"

contactDetails:

name: "Reine Aishwarya - Product Owner"

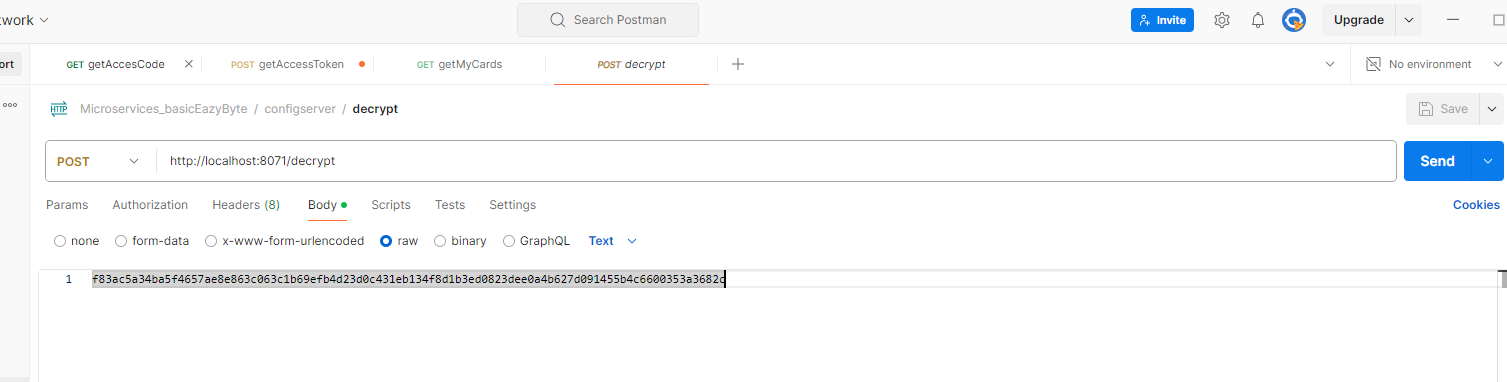
email: "{cipher}f83ac5a34ba5f4657ae8e863c063c1b69efb4d23d0c431eb134f8d1b3ed0823dee0a4b627d091455b4c6600353a3682c"

onCallSupport:

- (453) 392-4829

- (236) 203-0384

* Decryption can be done like this



How the encryption process works.

1. Network team can prevent decrypt URL to be accessed by everyone and only microservice application can access it
2. Infra team will pass the encryption KEY as env parameter or CLI from jenkins.
3. Admin who has the sensitive information encrypts using encrypted endpoints and sets the encrypted value in github.
4. The config server can decrypt the sensitive information using the secret key and pass it to microservices. This endpoint can be accessed only by microservice as the URL is restricted for other users

## Refresh information

Microservice will fetch the config information only during startup and will cache it. If we want to update it during runtime we have to use actuator

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Enable actuator endpoint in config server and all microservice

management:

endpoints:

web:

exposure:

include: "\*"

Now when we fetch configuration using config server it will return the latest value from Git instead of fetching values from cache.

But other microservices have already cached the information during start-up. This we can do using the actuator. We can go to the actuator endpoint of each microservice [http://localhost:actuator](about:blank) . It will have many endpoints. One of them will be that of refresh <http://localhost:8080/actuator/refresh> . If we invoke this endpoint with POST after that the value will change in microservice without restart

## Auto Refresh cache

Actuator is a good option to update configuration. But we may have hundreds of microservice and each may have hundreds of instances. If we want to manually update the information it may take a lot of manual work and we may miss a few pods leading to inconsistency.

So to overcome this challenge we need to use a new project inside the spring cloud, which is the spring cloud bus. Spring cloud bus links all the nodes of a distributed system with a lightweight message broker. And this can be used to broadcast the state changes, for example, configuration changes or any other management instructions. So whenever you are using the spring cloud bus behind the scenes this spring cloud bus is going to interlink all your microservices instances with a lightweight message broker like Rabbitmq or Kafka. With this, the advantage is you need to invoke a bus refresh api path available against your actuator only one time for one of the instances.

If there are 500 instances running inside your production, you don't have to invoke the actuator refresh api for all your 500 instances. Instead, you can simply invoke the bus refresh API for any of the instances inside this total 500 instances. With that, the spring cloud bus will take care of communicating the changes happening on the spring cloud config server to all other nodes or the instances connected to the same message broker like Rabbitmq?

How to implement

* Install rabbit MQ using docker
* Add dependency related to Spring cloud bus and rabbit MQ in All microservice including config server, below will add both

<dependency>

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

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

* Then make sure actuator endpoint is enabled

management:

endpoints:

web:

exposure:

include: "\*"

* Then add the location of RabbitMQ

rabbitmq:

host: "localhost"

port: 5672

username: "guest"

password: "guest"

* Then we need to hit the endpoint <http://localhost:8080/actuator/busrefresh> with POST. If we hit it for one endpoint all instance of all API will get the change

If we want no manual update and update of config on the push of code to github then we have to use web hook. For this on top of previous change we need to add spring cloud config monitor in config server only.

<dependency>

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

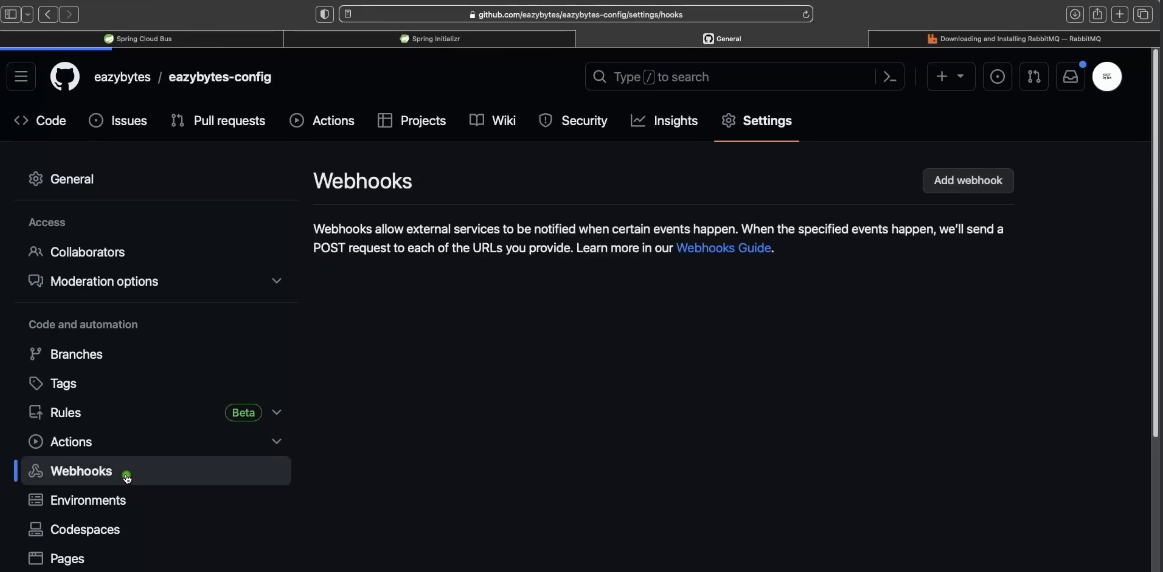
<artifactId>spring-cloud-config-monitor</artifactId>

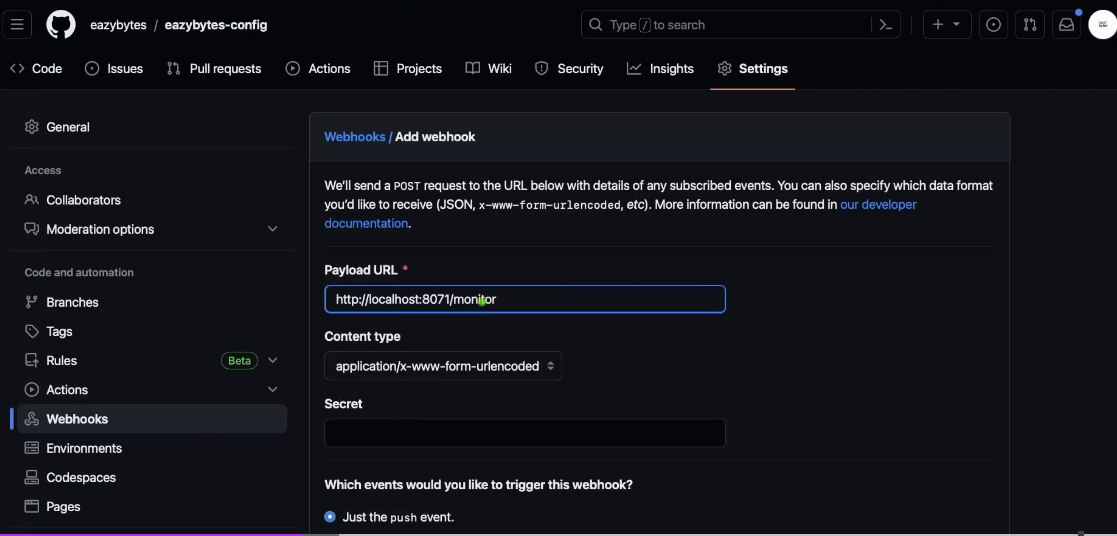
</dependency>

<dependency>

This will add a new API path /monitor. This is not part of the actuator. Using this monitor API path available inside the Spring Cloud Config server, we can create a webhook inside the GitHub repo saying that whenever a change happens inside GitHub repo, like a new property is added, please invoke the monitor API path. As soon as this monitor API path receives a webhook request from the GitHub repo, behind the scenes it is going to invoke the refresh event with the help of Spring Cloud Bus and RabbitMQ.

Changes in git hub





This will not work for local host so we can try hookdeck.com

## Dockerize config server

In docker local host will not work so we need to profive the config server path, the profile and the application name for each microservice

environment:

SPRING\_APPLICATION\_NAME: "loans"

SPRING\_PROFILES\_ACTIVE: qa

SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/

When we are going to deploy our microservices, our applications as Docker containers and these containers, we don't have to manually monitor them and we don't have to manually handle the scaling or elasticity requirements. Instead, container orchestration products like Kubernetes, they are going to handle that. So to handle these containers effectively, platforms like Kubernetes or even Docker, they need to understand whether my running container is working without any issues and the health of my running container is fine. They will try to understand whether my container is running properly. If not, they will try to make some corrective steps like maybe they will try to restart the container even after restarting the container. If the health probes are not working properly, they will try to create a new container by deleting the existing one and the same.

## Liveness Readiness

Liveness is a concept using which we can send a signal from the container or the application indicating whether my container is running properly or it has some health issues. If the output is that the container is alive and it is working properly, then there is no action required because the current state or current health is already good. Whereas if the container is dead, then an attempt should be made by the products like Kubernetes or any other products to heal the application by restarting or by creating a new container.

A readiness probe used to know whether the container or app that we are trying to probe is ready to start receiving the network traffic from its clients. Sometimes, especially during the startup, your container might be alive. The output from the liveness probe can be positive, but it may not be ready to accept any new traffic. It might be doing some background work or it might be warming up to accept the request behind the scenes, it might be doing some database initialization. So to get ready and accept the traffic, your container will take more time, especially during the startup time. So that's why to avoid any scenarios where my Kubernetes or any other platform trying to send the request before it is completely ready, we can make this readiness probe to send an output saying that I am not yet ready. Please give me some more time. So platforms like Kubernetes, they will make sure both liveness and readiness, they are giving a positive response in order to send the request that it is being received from the client applications.

We need to know if the config server is live and ready as other servers depend on it . This is done with the help of 2 actuator endpoints /actuator/health/liveness and /actuator/health/readiness. This needs to be enabled in config server properties like below

management:

endpoints:

web:

exposure:

include: "\*"

health:

readiness-state:

enabled: true

liveness-state:

enabled: true

endpoint:

health:

probes:

enabled: true

In order to expose the readiness and livelines we need to make below changes in the configserver section of Docker Compose. It hits the health readiness URL of the config server and if it return UP then success otherwise failure

configserver:

image: "eazybytes/configserver:s6"

container\_name: configserver-ms

ports:

- "8071:8071"

depends\_on:

rabbit:

condition: service\_healthy

healthcheck:

test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"

interval: 10s

timeout: 5s

retries: 10

start\_period: 10s

In individual service we we add depend on

accounts:

image: "eazybytes/accounts:s6"

container\_name: accounts-ms

ports:

- "8080:8080"

depends\_on: // To check if config server is UP

configserver:

condition: service\_healthy

environment:

SPRING\_APPLICATION\_NAME: "accounts"

Same we can do for RabbitMQ

services:

rabbit:

image: rabbitmq:3.13-management

hostname: rabbitmq

ports:

- "5672:5672"

- "15672:15672"

healthcheck:

test: rabbitmq-diagnostics check\_port\_connectivity

interval: 10s

timeout: 5s

retries: 10

start\_period: 5s

extends:

file: common-config.yml

service: network-deploy-service

## Common Config

There are a lot of common items in Docker compose which are repetitive. We can move them to a common config

services: // Will be used by all service including config and rabbit MQ

network-deploy-service:

networks:

- eazybank

microservice-base-config: // used by our build service and config

extends: // import the above service

service: network-deploy-service

deploy:

resources:

limits:

memory: 700m

environment:

SPRING\_RABBITMQ\_HOST: "rabbit"

microservice-configserver-config: // Not needed by config and rabbit MQ only by custom service

extends: // import the above service

service: microservice-base-config

environment:

SPRING\_PROFILES\_ACTIVE: qa

SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/

An individual service and add them this will remove the repeatedness

accounts:

image: "eazybytes/accounts:s6"

container\_name: accounts-ms

ports:

- "8080:8080"

depends\_on:

configserver:

condition: service\_healthy

environment:

SPRING\_APPLICATION\_NAME: "accounts"

extends: // import the service

file: common-config.yml

service: microservice-configserver-config

# Service Discovery

Discovery Server

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

@SpringBootApplication

@EnableEurekaServer

public class EurekaserverApplication {

public static void main(String[] args) {

SpringApplication.*run*(EurekaserverApplication.class, args);

}

}

spring:

application:

name: "eurekaserver"

config:

import: "optional:configserver:http://localhost:8071/"

management:

endpoints:

web:

exposure:

include: "\*"

health:

readiness-state:

enabled: true

liveness-state:

enabled: true

endpoint:

health:

probes:

enabled: true

Properties from GIT properties file

server:

port: 8071

server:

port: 8070

eureka:

instance:

hostname: localhost

client:

fetchRegistry: false

registerWithEureka: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

In client side

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

management:

endpoints:

web:

exposure:

include: "\*"

endpoint:

shutdown:

enabled: true // Will allow the actuator to shut down

info:

env:

enabled: true // Enable env info related properties

endpoints:

shutdown:

enabled: true // Will allow the actuator to shut down

eureka:

instance:

preferIpAddress: true // Will register with IP

client:

fetchRegistry: true

registerWithEureka: true

serviceUrl:

defaultZone: http://localhost:8070/eureka/

info:

app:

name: "accounts"

description: "Eazy Bank Accounts Application"

version: "1.0.0"

Important URL

<http://localhost:8070/eureka/apps>

If we send header Accept application/json will get as JSON

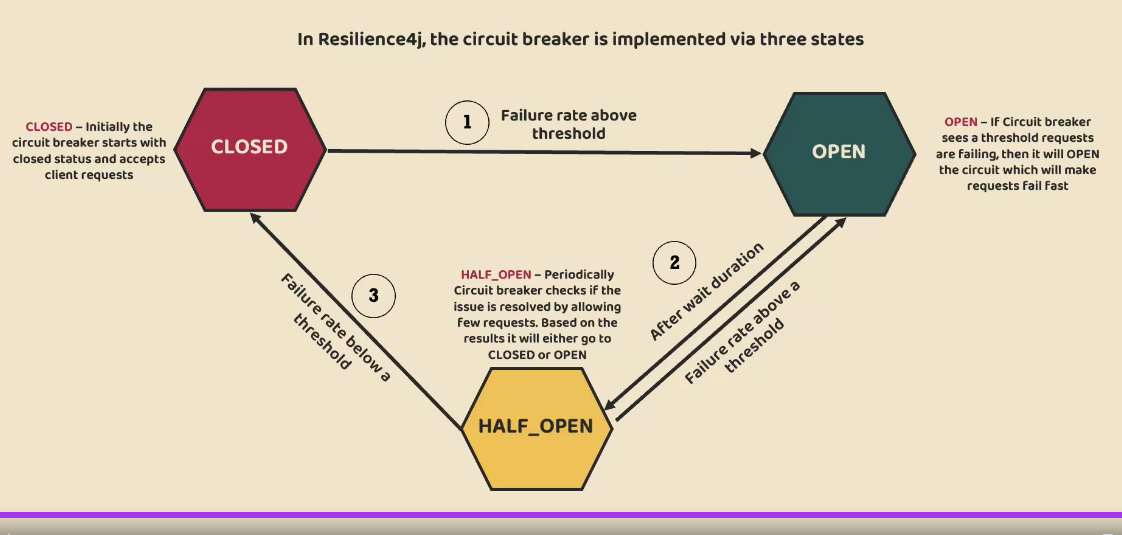
As soon as my Eureka Server realizes that the majority of the microservices instances are not sending the heartbeat, then it is not going to react and remove all the service instance details from the registry. Instead, it is going to enter into a self-preservation mode and once the Eureka Server enters into this mode, even if it is not receiving the heartbeat signals from the service instances, it is not going to remove the service details from the service registry. So this prevents the Eureka server from evicting all the instances due to some temporary network delays or temporary network glitches. Many times the network issues can be on the internet provider or it can be on the cloud provider side, or it can be within the microservice network regardless of where the issue is, after a few minutes or few seconds, the network related issue will get automatically resolved. So for these kinds of scenarios only, we have these Eureka self-preservation modes. Inside this self-preservation mode. The Eureka Server continues to serve the registered instances to client applications. Even it suspects that some instances are no longer available.

On a high level the summary is, Eureka Server is not going to panic whenever it is not receiving heartbeats from the majority of the instances. Instead, it will be calm and enter into the self-preservation mode and it will do the meditation. So during the self-preservation mode or during this meditation process, it is not going to evict all the instances from the service registry. So this feature is a savior where the network glitches are common and help us to handle the false positive alarms.

# Circuit Breaker

3 states of circuit breaker

1. Closed → Will continue to send
2. Open – > Will not send request and fail
3. Half Open → In open state it will not stay forever, periodically it will send certain amount of traffic



<dependency>

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

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

</dependency>

Setting circuit breaker in Gateway

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") //Name of circuit breaker

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

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

resilience4j.circuitbreaker:

configs:

default:

slidingWindowSize: 10 // monitor 10 request before moving to close from open

permittedNumberOfCallsInHalfOpenState: 2 // send 2 request in half open and decide which state to go open or closed

failureRateThreshold: 50 // if 50% request are failing then go to open state

waitDurationInOpenState: 10000 //Will wait 10 seconds before going to half open

This is default circuit breaker we can give name to circuit breakers and put individual configurations

Circuit breaker using feign client

If you haven’t used spring reactor use below dependency

</dependency>

<dependency>

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

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

</dependency>

cloud:

openfeign:

circuitbreaker:

enabled: true

resilience4j.circuitbreaker:

configs:

default:

slidingWindowSize: 10

permittedNumberOfCallsInHalfOpenState: 2

failureRateThreshold: 50

waitDurationInOpenState: 10000

We fall back in the Feign client

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

public interface CardsFeignClient {

@GetMapping(value = "/api/fetch",consumes = "application/json")

public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id")

String correlationId, @RequestParam String mobileNumber);

}

In case of error it switches to fallback

@Component

public class CardsFallback implements CardsFeignClient{

@Override

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

return null;

}

}

## Timeout:

Time out setup

spring:

application:

name: "gatewayserver"

config:

import: "optional:configserver:http://localhost:8071/"

cloud:

gateway:

discovery:

locator:

enabled: false

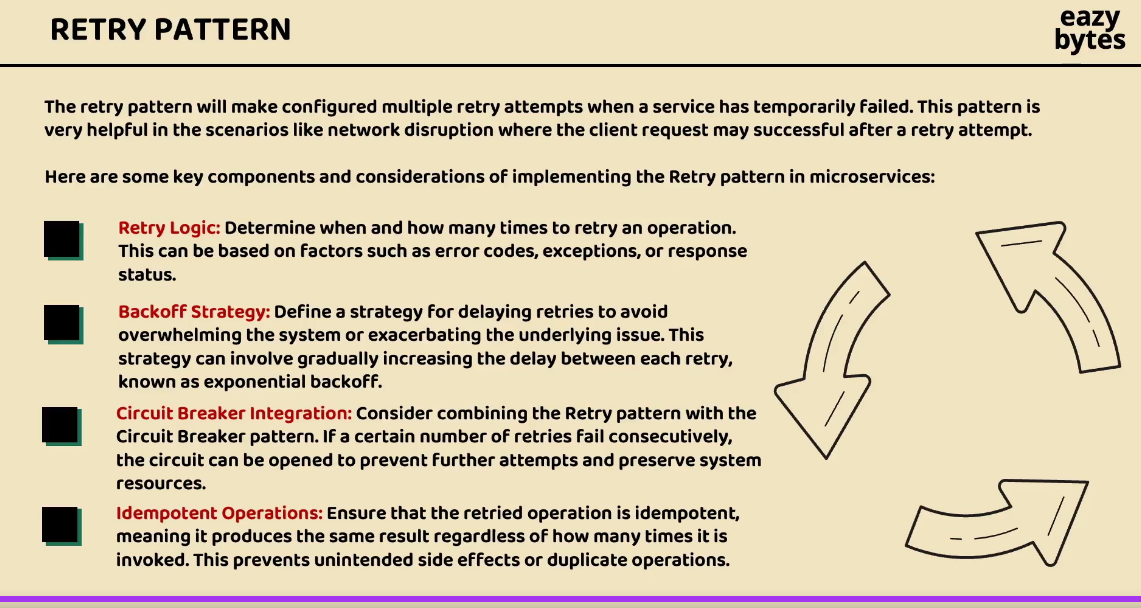
lowerCaseServiceId: true

httpclient:

connect-timeout: 1000 //if not getting connected in 1 second we are not going to wait and we are going to kill the request.

response-timeout: 2s //maximum time for which gateway server will wait to receive the response

## Retry Pattern



.route(p -> p

.path("/eazybank/loans/\*\*")

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

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

.retry(retryConfig -> retryConfig.setRetries(3) // Retry 3 times

.setMethods(HttpMethod.***GET***) // Which operations should we retry

.setBackoff(Duration.*ofMillis*(100),Duration.*ofMillis*(1000),2,true))) //first backoff will be in 100 ms, then next backoff //will be in a factor 100 \*2 upto a max of 1000ms

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

Retry in Controller

@Retry(name = "getBuildInfo",fallbackMethod = "getBuildInfoFallback")

@GetMapping("/build-info")

public ResponseEntity<String> getBuildInfo() {

***logger***.debug("getBuildInfo() method Invoked");

return ResponseEntity

.*status*(HttpStatus.***OK***)

.body(buildVersion);

}

public ResponseEntity<String> getBuildInfoFallback(Throwable throwable) {

***logger***.debug("getBuildInfoFallback() method Invoked");

return ResponseEntity

.*status*(HttpStatus.***OK***)

.body("0.9");

}

resilience4j.retry:

configs:

default:

maxRetryAttempts: 3

waitDuration: 500

enableExponentialBackoff: true

exponentialBackoffMultiplier: 2

ignoreExceptions:

- java.lang.NullPointerException

retryExceptions:

- java.util.concurrent.TimeoutException

## Rate Limiter

So based upon your requirements, you need to provide the details with the help of KeyResolver interface. There is also a default implementation of KeyResolver, which is PrincipalNameKeyResolver. If you are using spring security to secure your microservices, then with the help of these PrincipalNameKeyResolver, it is going to fetch the current logged in user name and accordingly it is going to enforce RateLimiter and by default you can see if the KeyResolver does not find a key, the requests are going to be denied if needed.

Using this Redis, we can implement these RateLimiter in gate way

<dependency>

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

<artifactId>spring-boot-starter-data-redis-reactive</artifactId>

</dependency>

@Bean

public RedisRateLimiter redisRateLimiter() {

return new RedisRateLimiter(1, 1, 1);

}

@Bean

KeyResolver userKeyResolver() {

return exchange -> Mono.*justOrEmpty*(exchange.getRequest().getHeaders().getFirst("user"))

.defaultIfEmpty("anonymous");

}

Rate limiter in API

resilience4j.ratelimiter:

configs:

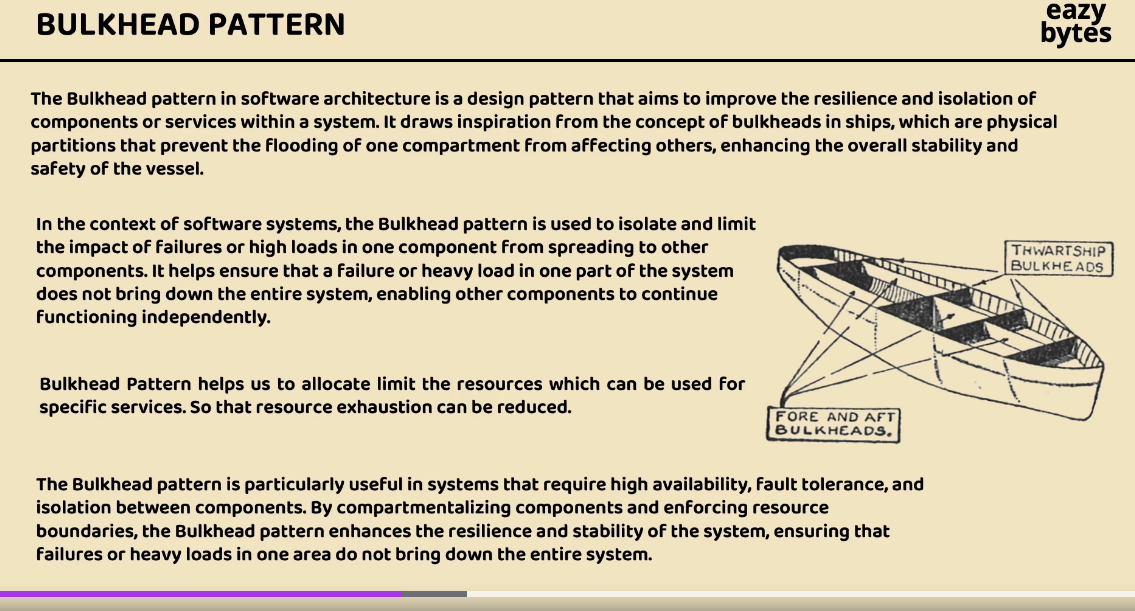
default:

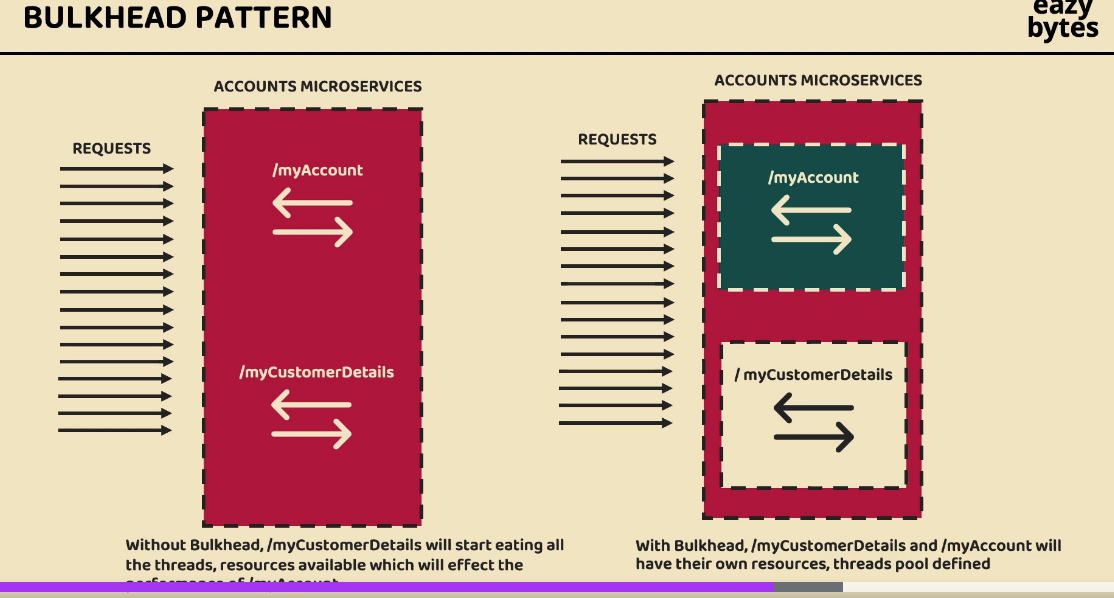
timeoutDuration: 1000

limitRefreshPeriod: 5000

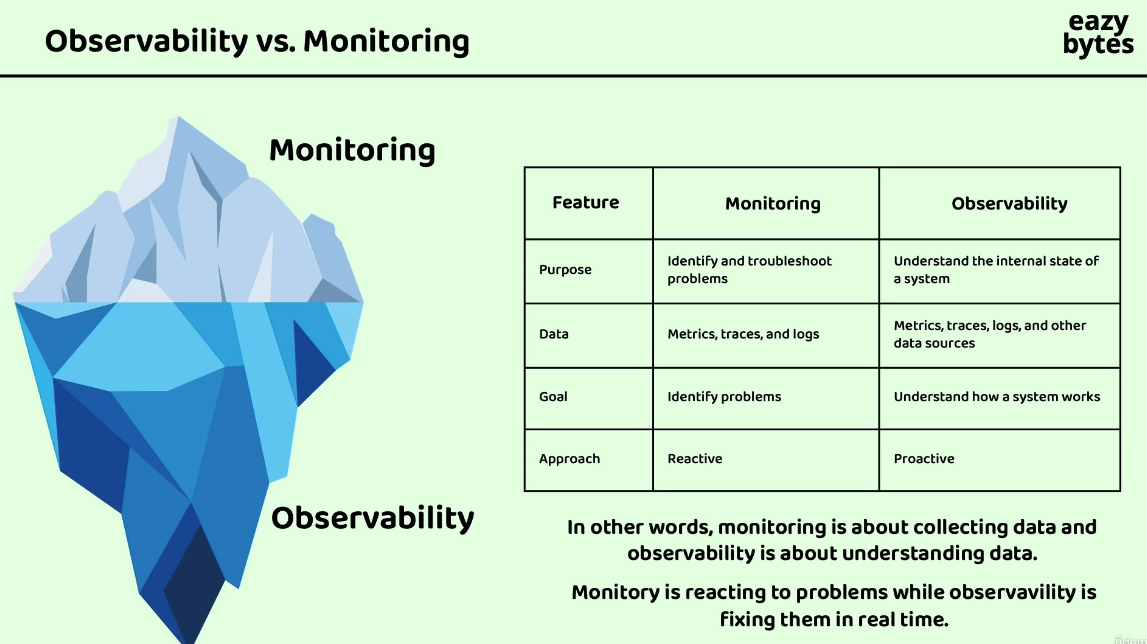
limitForPeriod: 1

## BulkHead Pattern

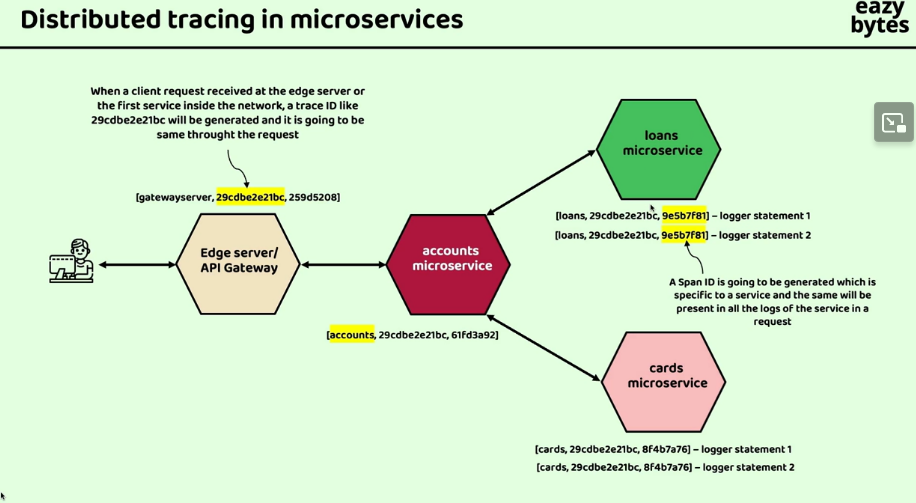




# Monitoring and Observability



## Distributed tracing



Spring cloud sleuth helps to generate metadata(apiName), traceId and spanId automatically in logs and it can be integrated with zipkin, however sleuth is going out of support

Same thing is done by micrometer

Also open telemetry