**MICROSERVICES**

**First learn about disadvantages of monolithic application then know why we are moving to microservices based architecture.**

**Learn what microservices can do and cannot do.**

**Start learning step by step.**

**Step1 : Spring Cloud config Server**

Why do we need this ?

Microservices will have the properties like database credentials and other properties. Some of them will be common and some will be different w.r.t envt, so all these configuration related properties will be kept in a centralized way, so each microservice can pick them up before the start.

Eg : spring cloud config server

**Step 2 : Load Distribution**

Why do we need this ?

If M1 want to distribute load for M2 having 4 instances then we use this

**Eg : spring cloud load balancer, Ribbon (used earlier not now)**

**Step 3 : Dynamic Load balancing**

Why do we need this ?

If one of the instance of M2 is failed then the load should be distributed uniformly in the remaining 3 instances and also if any new addition happens they also load should be distributed uniformly.

**Eg: Eureka Naming Server**

**Step 4 : Tracing**

Why do we need this ?

Whenever any request that is passing through multiple microservices fails then we need to identify in which microservice it failed, for tracing purpose we need this.

**Eg : Zipkin**

**Step 5: Common features**

Why do we need this ?

All microservices needs to having authentication and authorization to provide secure accesss.

**Eg: API gateway, Zuul (used earlier not now)**

**Step 5 : Fault Tolerance**

Why do we need this ?

If one microservice fail then other fails so to avoid this we need circuit breaker to avoid this.

**Eg: Resilience4j, Hystrix(used earlier not now)**

**May 12,2024**

**Step : 1**

Spring Cloud Config Server

1.**config client** dependency should be added for this.

2.For config server running on port 8888 we need to connect from our code , how to include it then ?

Use this in application.properties

**spring.config.import = optional:configserver:http://localhost:8888**

3.Try to hardcode min and max value and print details.

4. Try to fetch the limits (min,max) from application.properties first.

5. Create spring cloud config server for fetching details from it.

Create a new project by adding **config server** dependency

Launch the server at port 8888 by adding server.port=8888 in application.properties file.

6.create git local config repo having a properties containing the min and max values.

**7. Connecting spring cloud config server with git repo.**

Add below in config server running at port 8888

spring.cloud.config.server.git.uri=file://C:\\micorservices\\git-localconfig-repo

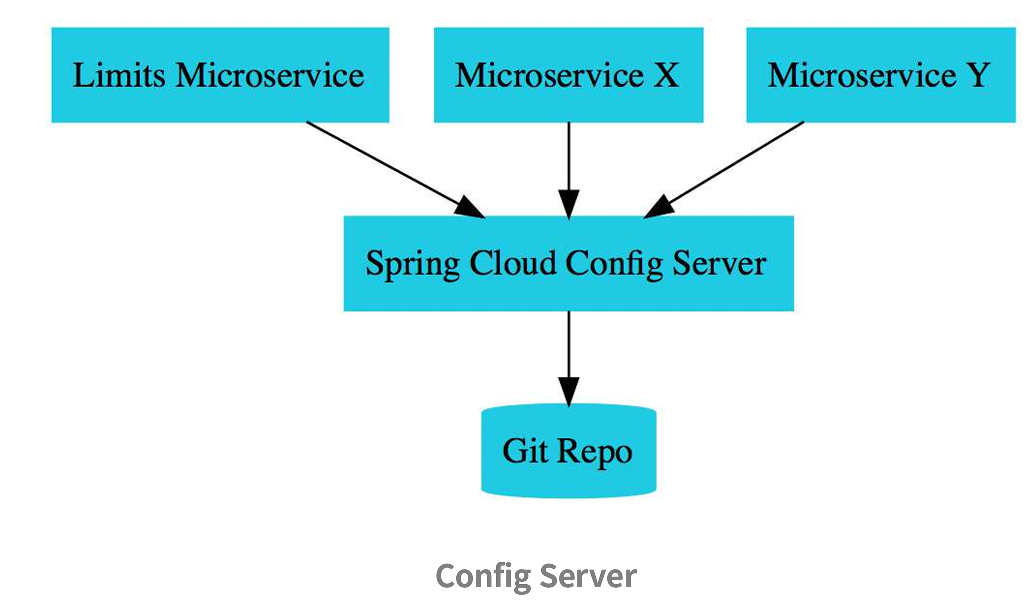
and add @EnableConfigServer on top of @SpringBootApplication.

<http://localhost:8888/limits-service/default>

limits-service is the file name created inside git folder.

**8. Connecting limits microservice with spring cloud config server.**

Make sure you have spring.application.name=limits-service in application.properties of limits microservice. Properties in git take more priority compared to properties in application.properties.



**How to store properties w.r.t different envt like dev, qa, test and prod ?**

Similar to limits-service.properties create files like limits-service-qa.properties, limits-service-dev.properties based on envt.

<http://localhost:8888/limits-service/qa>

<http://localhost:8888/limits-service/dev>

use the above urls for accessing properties w.r.t envt.

**But how to access these from limits microservice ?**

spring.profiles.active=qa

spring.cloud.config.profile=qa

change the evnt accordingly in the application.properties file.

If the file name in git is different then you can add it in below way

spring.cloud.config.name=

**How you configure it for other microservices ?**

Similar to others create the file names same as application name and follow accordingly.

**Suppose a property changed in the files present in the git folder, how can I make my application to use the updated properties instead of using the old one without restarting the application?**

add **@RefreshScope** on the Controller,  
add **spring-boot-actuator** dependency   
add **management.endpoints.web.exposure.include=\*** in application.properties  
and send a **POST** request to **http://localhost:<port>/actuator/refresh**   
refresh the url once, updated properties will be displayed

@RefreshScope annotation is needed only if we are using @Value. So in the above @RefreshScope is not needed.

The problem with using the /refresh endpoint is that we have to manually fire a POST request to it. It is not automatically done when the configuration changes. Also, we need to fire a request to /refresh on all the services which might get affected by the change in the property. That means we have to keep track of which property is used in which application. If we have 100 microservices using the property, we need to fire /refresh on all those microservices.

The solution is to use **Spring-Cloud-Bus**. This, along with a Queuing service like RabbitMQ, will trigger refresh events on all dependent microservices. However, Spring-Cloud-Bus is beyond the scope of the curriculum.

**POINTS TO PONDER :**

1. What is the use of spring.cloud.config.fail-fast=true property ?  
   if cloud-config server is down, then client will throw error at run time so that config server needs to start first then only microservices are allowed to start.
2. A microservice has both local as well cloud config ?  
   local will override the cloud
3. Which of the following will take priority in cloud config ?  
   application.yml
4. If config server fails in between the microservices which already starting will not fail.
5. If cloud config sever fails/unavailable then it will use properties files in the individual applications as a fallback.
6. Config server properties files can be accessed using rest end point.  
   eg : http://localhost:8888/limits-service.properties

**May 13 , 2024**

**1.Create a new project Currency-Exchange-Service at port 8000.**

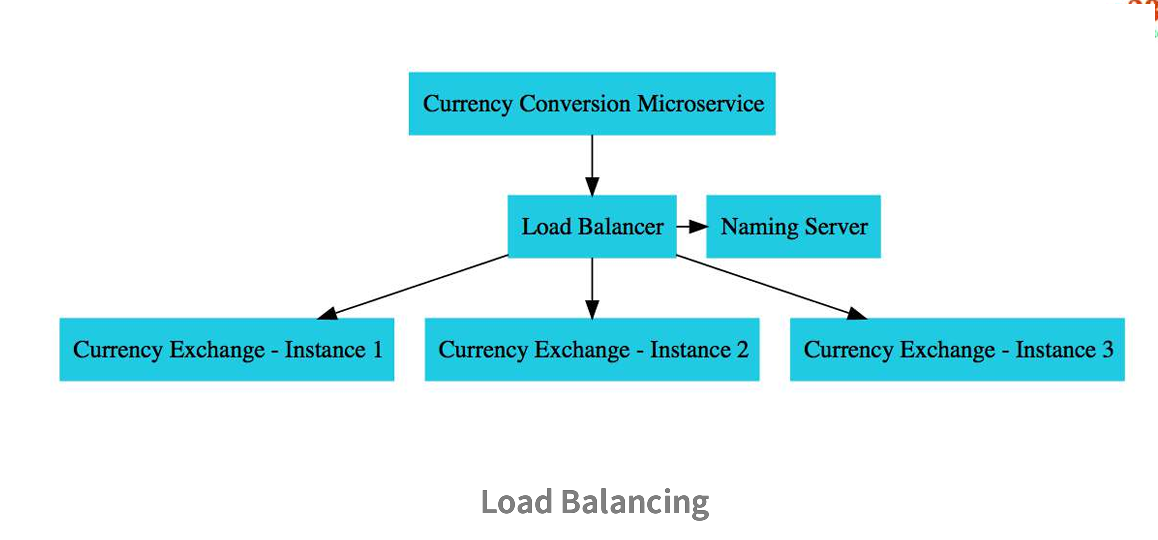
**2.Try to create instance of same application at different ports like 8000,8001**

**How to run two instances of a application in different ports ?**

**Click on Run as -> Run Configuration**

**Then create duplicate of current running application and provide -Dserver.port=8001 in VM arguments and click on Apply and then run.**

**Load Balancer**



**Things imp to know while using h2 datdabase.**

**1.for showing query’s in the console use below command in application.properties**

spring.jpa.show-sql=true

**2.For seeing h2 in console mode**

spring.h2.console.enabled=true

**3.Spring data source url for h2**

spring.datasource.url=jdbc:h2:mem:testdb

**4.From spring >2.5.0 the query’s are getting executed before the tables are created. TO avoid that use the below command**

spring.jpa.defer-datasource-initialization=true

**16 May,2024**

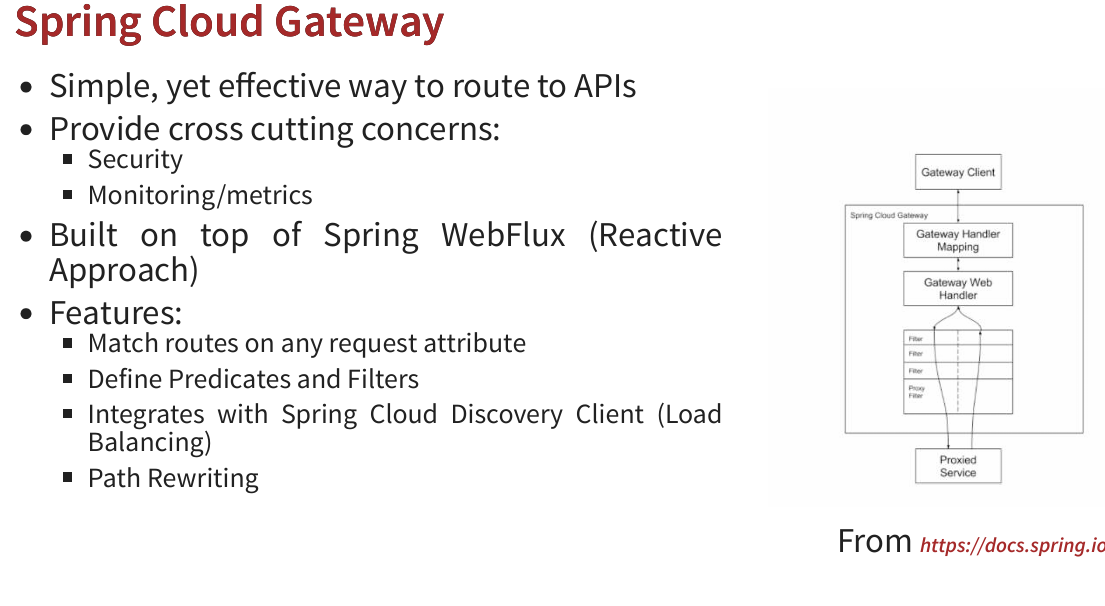
**Feign makes it easy to call other microservices. It will reduces a lot of code for calling other microservices.**

**Naming server used here is – Eureka.**

**Included api-gateway , which is the common point for the microservices they enter for authorization and authentication.**

**May 18,2024**

**Custom routes with API gateway**



**Circuit breaker with resilience4j**

**Dependency’s needed :**

**Sping-boot-starter-actuator**

**Spring-boot-starter-aop**

**Spring-boot-resilience4j**

**May 19,2024**

Default retry of circuit breaker is **3 times.**

If you want **it more than 3 times** then use below

resilience4j.retry.instances.sample-api.maxAttempts=5 along with @Retry(name = "sample-api")

resilience4j.retry.instances.sample-api.wait-duration=1s

resilience4j.retry.instances.sample-api.enable-exponential-backoff=true

**circuit breaker pattern**

@CircuitBreaker(name = "default", fallbackMethod="goToFallback")

**rate limiting – we are specifying how many api calls we need to make in a specific time period.**

@RateLimiter(name="default")  
resilience4j.ratelimiter.instances.default.limitForPeriod=2

resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s

**Bulkhead – how many concurrent calls are allowed**

@Bulkhead(name="default")  
resilience4j.bulkhead.instances.default.max-concurrent-calls=10