Microservices Currency conversion

1. **limits-service**

First creating a project with name limit-service with basic spring dependencies such as web, actuator, dev-tools and more importantly spring-client.

Storing configuration for the different clients is better to be placed at single place. So as limits-service configuration will also be placed at spring cloud config server. For storing a configuration we generally use git.

In application.properties I used the port 8081

Use url ==> localhost:8081/limits-service/

1. **Spring-cloud config-server**

Creating the project with the name spring-cloud-config-server having basic dependencies and in addition with spring-server.

Don’t forget to add the annotation in spring Application as **@EnableConfigServer**

Creating a git repository at certain path in your desktop.

**Link spring-cloud to git:** Now have to use “link source” the git repository. Add the property file in git repository which we want to access. And in spring-config’s application.properties link them.

Example locally we link it as : spring.cloud.config.server.git.uri = [file:///C:/Users/Admin/Desktop/microservicesProject/git-localconfig-repo](file:///C:\Users\Admin\Desktop\microservicesProject\git-localconfig-repo)

I used 8888 port for spring-cloud-config-server. Use the url : localhost:8888/limits-service/default/ to access limits-service configuration.

**Setting up multiple profiles:** to set up multiple profiles we will create multiple properties file. In this project created limit-services-dev.properties and limit-services-qa.properties in a git repository.

After making changes commit in git repository.

Now to access the details of these configurations from spring-cloud-config-server using the url as: localhost/8888/limits-service/dev/ and localhost/8888/limits-service/qa/ . if any of the property is not defined then that property will be overridden by the properties of default property file.

**Link Spring-cloud-config-server and limits-service(spring client):** to connect them we have to make changes in limits-service. As limits service configuration will be picked from application.properties and we want it to be picked from git repository so we will rename the application.properties file of limits-service to bootstrap.properties.

Now we add the url of spring-cloud in bootstrap.properties to connect each other like:

In bootstrap.properties we add spring.cloud.config.uri =http://localhost:8888/, it will start picking configuration via spring cloud config server.

For instance if we want particular configuration for different environment then we have to add the line in bootstrap.properties as : spring.profiles.active =dev or spring.profiles.active=qa

We can also pass active profiles by vm argument or java application arguments.

Restart the applications when you change the configurations. Other option is refresh url.

1. **Currency-exchange-service**

First creating a project with name currency-exchange-service with basic spring dependencies such as web, actuator, dev-tools and more importantly spring-client.

Configuring server.port = 8000 for this service in application.properties.

Use url ==> localhost:8000/currency-exchange/from/USD/to/INR/ or for any other currency.

To create multiple instance of currency exchange we use run configuration menu option and duplicate the configuration with different port. The port should be written in vm arguments so that it should override the port. Write Dserver.port = 8001 to create instance at port 8001.

**H2 Database**

For setting up in-memory database add h2 and data-jpa dependencies in pom.xml.

Try to open h2-console using : localhost:8000/h2-console/ if any error try to remove and add the line spring.h2.console.enabled= true in application.properties.

In h2-console at jdbc url add : jdbc:h2:mem:testdb and also add spring.datasource.url =jdbc:h2:mem:testdb in application.properties.

1. **Currency-conversion-service**

First creating a project with name currency-conversion-service with basic spring dependencies such as web, actuator, dev-tools and more importantly spring-client.

Configuring server.port = 8100 for this service in application.properties.

Use url ==> localhost:8100/currency-converter/from/USD/to/INR/quantity/1000

Communicating with currency-exchange using class RestTemplate and mapping the response and then calculating the totalAmount.

Code snippet to communicate with currency-exchange service:

@GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversionBean convertCurrency(@PathVariable String from , @PathVariable String to ,@PathVariable BigDecimal quantity ) {

//starts (making request and getting response)

Map<String, String> uriVariables = **new** HashMap<>();

uriVariables.put("from", from); // to map in {from} and {to} inside getForEntity method

uriVariables.put("to", to);

ResponseEntity<CurrencyConversionBean> responseEntity= **new** RestTemplate().getForEntity("http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables);

//ends

CurrencyConversionBean response = responseEntity.getBody();

**return** **new** CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity , quantity.multiply(response.getConversionMultiple()), response.getPort());

}

**Using Feign-Client**

Used to call other rest services. It can be used in place of RestTemplate(). It provides integration with ribbon which provide client side load balancing.

In main application add the annotation @EnableFeignClient(“com.example.microservices.currencyconversionservice;”).

After create a feign proxy to talk with other rest service. Use the annotation @FeignClient at Proxy class.

Use url ==> <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>

**Using Ribbon (Client side load balancing)**

Ribbon is used for client side load balancing by distributing calls to different services. Here the call is being made to currency exchange for two different ports 8000 and 8001. Using the annotation as:

@FeignClient(name= "currency-exchange-service")

@RibbonClient(name = "currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

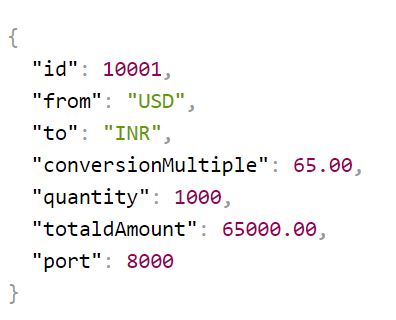
**public** CurrencyConversionBean retrieveExchangeValue(@PathVariable String from , @PathVariable String to);

}

In properties file we have to add the list of servers to which call is being made :

currency-exchange-service.ribbon.listOfServers=http://localhost:8000, <http://localhost:8001>





**Eureka Naming Server**

Eureka is also called naming server. Service registration and service discovery features is used while using eureka. Each service will register itself in eureka naming server and the discovery is done by the ribbon.

Creating a package with spring dependency as eureka Server. And adding annotation as

1. @EnableEurekaServer with having a dependency at pom as 2. spring-cloud-starter-netflix-eureka-server.

Disabling registering and discovery for itself as:

eureka.client.register-with-eureka = false.

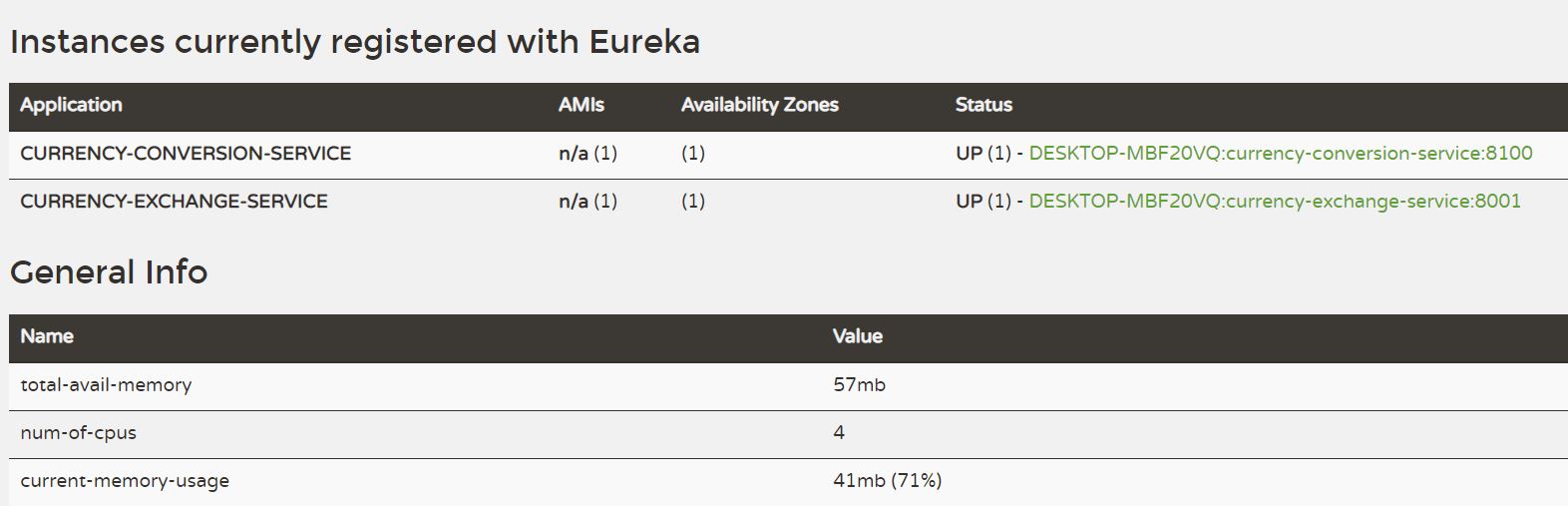
eureka.client.fetch-registry=false

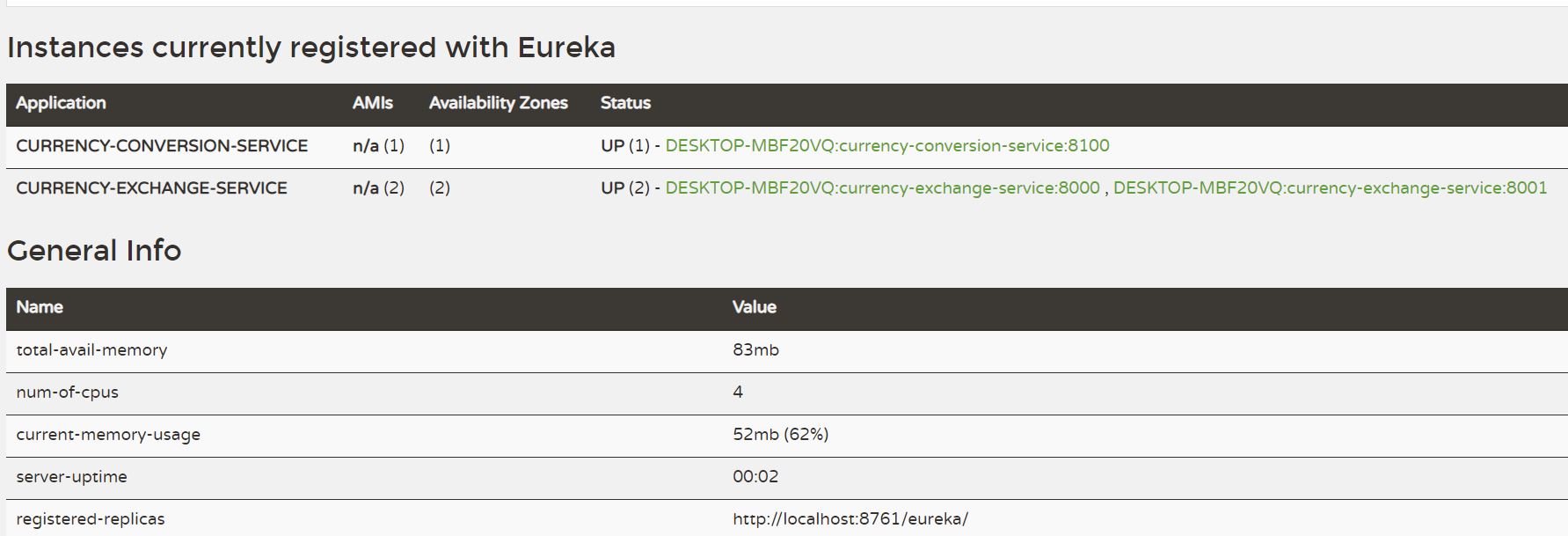
**For Service Registration:**

If we want a particular service to be registered to eureka we have to add dependency on currency-conversion,

1. currency-exchange as spring-cloud-starter-netflix-eureka-client.
2. And enable @EnableDiscoveryClient in currency-conversion
3. Then adding the url in properties file for eureka.

URL=http://localhost:8761 For Eureka Server





**Using Zuul api gateway**

Zuul is used as a gateway for monitoring, rate limits and for various other purpose. We will be configuring zuul at 8765 port.

Extend the class ZuulFilter and implement all the methods.

To pass the request from zull gateway we have to configure urls as:

localhost:8765/{application name}/{application url} e.g localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR/

In Zuul logging filter a class is created with @Component that extends ZuulFilter it is having several methods that will be overridden and will be helpful for api gateway.

**Distributed Tracing:**

It is used to find out the request with unique id. And that request with unique id will be helpful to find the request in each service and will be used for centralized logging.

Spring-cloud-sleuth dependency should be added that will put out the id while making any calls to other services or api.

**RabbitMQ:** for the logging of these requests can be placed at a single place so for that zipkin is used . all the logs will be saved in rabbit MQ and it will then be extracted by zipkin.

For installing Rabbit MQ erlang should be installed. So first install erlang and then Rabbit MQ. There is compatibility with some version of RabbitMQ and Erlang.

**ZIPKIN:**

Zipkin does not get from spring initializer now so installing the jar directly and running it with command java – jar zipin.jar. directly typing zipkin quickstart in google will get us the java download link. After downloading run the jar using run jar command.

localhost:9411/zipkin/.

Add dependency spring-cloud-starter-zipkin and spring-cloud-rabbit.

To tell Zipkin that the RabbitMq server is up an running and zipkin needs to connect to that. So a command to run the zipkin will be modified as:

1.SET RABBIT\_URI=amqp://localhost

2.java -jar zipkin.jar

**Using actuator referesh:**

When commiting changes to git repository for the property file , the latest change will not be picked. For the latest change to be picked we can restart the application or can hit referesh url with a post request as:

https://localhost:8080/actuator/refresh/, <https://localhost:8080/application/refresh/>

for its working enable endpoints for actuator and disable spring security from properties file.

**Spring Cloud Bus:**

As we are using referesh url, we have to use it for every service to pickup the latest change. If there are 50 instances of application services we have to hit the refresh url 50 times to pick new change. That’s why spring cloud bus is being used. We will use amqp RabbitMQ protocol. We need to add dependency on services as well in spring cloud config server as **spring-cloud-starter-bus-amqp** and will be done with **RabbitMQ.** We need to hit only one url and will referesh for all the instances of the services. The post request will be sent tohttp://localhost:8080/actuator/bus-refresh.

**Hystrix:**

Hystrix is used to build fault tolerant microservices. We use the dependency spring-cloud-starter-hystrix and add the annotation on service as @EnableHystrix and on all controller method as **@HystrixCommand(fallbackMethod)** as **@HystrixCommand(fallbackMethod= “fallbackMethoddef”)** and defining fallbackMethoddef. If any exception occurs then fallback method will be called.