**Microservices with Spring Boot**

* Microservices are the services exposed by the REST which has small well chosen Deployable units and as well as they are cloud enabled.
* Here, cloud enabled means we can have multiple instances of a single microservice.
* Suppose, you have 3 microservices. You can have multiple instances of each microservice respectively.
* Spring cloud: It is kind of Umbrella which consists of wide variety of Projects.

**Naming server:**

* All the instances of microservice will register to Eureka Naming Server.

**Ribbon:**

* Client Side load balancing

**Feign:**

* To write simple RestFul Clients.

**For visibility and monitoring:**

* Zipkin Distributed Tracing
* Netflix API Gateway(Zuul Api)

**Fault Tolerance**

* Hysterix

Microservices are an arhitectural style where a single application is built but not as a single application but as a suite of single applications each of which is running as an independent process an each of which communicates with well known protocol.

1. **Limits-service**

**Main class**  
  
@SpringBootApplication  
**public class** LimitsServiceApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(LimitsServiceApplication.**class**, args);  
 }  
  
}

**bootstrap.properties**

spring.application.name=limits-service

spring.cloud.config.uri=http://localhost:8888

#spring.profiles.active=dev

**bean package (Model)**

**public class** Limit {

**private int** maximum;  
 **private int** minimum;  
  
 **protected** Limit () {  
 }

**//Getter Setter**

}

**Configuration**

@Component  
@ConfigurationProperties(**"limits-service"**)  
**public class** Configuration {  
  
 **private int minimum**;  
 **private int maximum**;  
  
**//Getter Setter**

}

**RestController**

@RestController  
**public class** LimitsConfigurationController {  
  
 @Autowired  
 **private** Configuration **configuration**;  
  
 @GetMapping(**"/limits"**)  
 **public** Limit retrieveLimitsFromConfigurations() {  
  
 Limit limit = **new** Limit(**configuration**.getMaximum(), **configuration**.getMinimum());  
  
 **return** limit;  
 }  
}

Here is the working zip version of limits-service



1. **Spring Cloud Config server**

@EnableConfigServer

@SpringBootApplication

**public class** SpringCloudConfigServerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.run(SpringCloudConfigServerApplication.**class**, args);  
 }  
}

**src/main/resources/application.properties**

spring.application.name=spring-cloud-config-server

server.port=8888

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

**git-localconfig-repo**

**limits-service.properties**

limits-service.minimum=81

limits-service.maximum=8888

**limits-service-dev.properties**

limits-service.minimum=1

limits-service.maximum=1111

**limits-service-qa.properties**

limits-service.minimum=2

limits-service.maximum=2222





**Feign client:**

Suppose you have one microservice which is exposed to some particular port say 8000. When you hit the rest end point you get some data (mostly JSON).

Eg: We have **currency-exchange-service** which exposes rest end point:

/currency-exchange/from/{from}/to/{to}

This microservice will give the currency exchange amount like:

So, if we hit <http://localhost:8000/currency-exchange/from/USD/to/INR>, we will get

{

**id**: 10001,

**from**: "USD",

**to**: "INR",

**conversionMultiple**: 65,

**port**: 8000

}

Here, we want to make use of **conversionMultiple** in another microservice

Now that we got exchange amount we want to calculate total amount i.e, if we pass 1000 USD dollar the total we get is Rs. 65000. This is done in another microservice like **currency-converison-service** that is running is different port say 8100**.**

**currency-conversion-service** needs exchange amount (**conversionMultiple**) but that is exposed by **currency-exchange-service**. So, **currency-conversion-service** needs to connect to the **currency-exchange-service**

The way it can connect to other microservice is by using:

1. RestTemplate
2. Feign client

1. **RestTemplate:** lot of boiler plate code is required for RestTemplate



This response contains **conversionMultiple**

1. **Feign Client**

Instead of using RestTemplate we can use FeignClient that can be declared simply by:

@FeignClient(name = "currency-exchange-service", url = "localhost:8000")

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

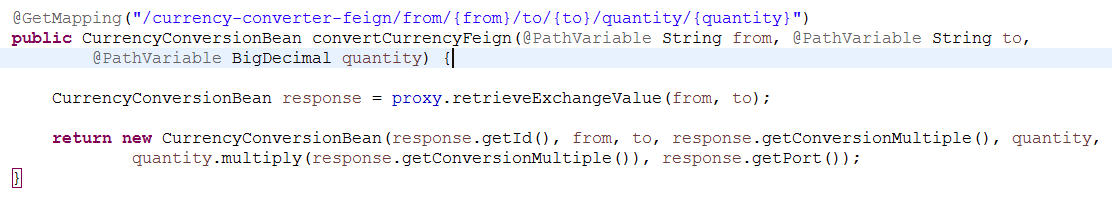
@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrenyConversionBean retrieveExchangeValue(@PathVariable String from,

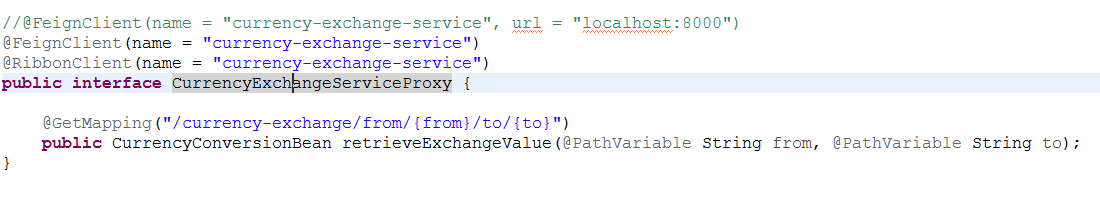
@PathVariable String to);

}

* Annotation @FeignClient is used to define an interface as a Feign Client which can contains RequestMapping which contains the rest endpoint of any other microservice(in our case: currency-exchange-service)
* name = "currency-exchange-service": name of the service for which you want to define a Feign Client.
* url = "localhost:8000" : this is the PORT number of the rest endpoint.



**Ribbon**



It is used for client side Load distribution

@RibbonClient Annotation is used to define an interface as Ribbon Client

Now, you can define the port numbers of a different instances of a microservices like:

**application.properties:**

spring.application.name=currency-conversion-service

server.port=8100

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

Now, as it is Ribbon enabled on ports 8000, 8001 of microservice currency-exchange-service, when you will hit

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

* first time it may use an instance of **currency-exchange-service** microservice on port 8000
* Next time is may use an instance of **currency-exchange-service** microservice on port 8001

Hence, Ribbon is smart enough to distribute the load between instances of same microservice.

**Drawbacks of using Ribbon without using Eureka**

1. Every time a new instance of microservice is created, we have to change every time in **application.properties** and add the port number(URL) of instance of microservice.
2. So to overcome this problem i.e, dynamically adding and removing instance of microservice Eureka naming server comes into the picture.

**Eureka Naming Server**

Every instance of microservice registers in eureka naming server.

It runs on:

http://localhost:8761

This is called service discovery. Whenever a microservice wants to know about the instances of other microservices, it talks to the Eureka naming server.

If **currency-conversion-service** wants to know how many instances of **currency-exchange-service** are available, it must talk to the Eureka Naming server.

So, at the time of starting every instance of a microservice registers to the Eureka naming server.

Now, instead of adding all the ports in application.properties for Ribbon, Ribbon talks to the Eureka naming server to know all the instances of the microservice.

spring.application.name=netflix-eureka-naming-server

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**#** application.properties of **netflix-eureka-naming-server** microservice

**Steps:**

1. We have to create our Eureka Naming server i.e, another microservice on some port say 8761. This microservice will behave as a Server of Eureka.
2. Other microservices must connect to this Eureka server as clients by using:

* eureka.client.service-url.default-zone= [http://localhost:8761/eureka in application.properties](http://localhost:8761/eureka%20in%20application.properties) of other microservices.

@EnableEurekaServer Annotation is used to Define your microservice as Eureka Naming Server.

Now, rather than:

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

we can use in other microservices to connect with Eureka Server

spring.application.name=currency-conversion-service

server.port=8100

eureka.client.service-url.default-zone=http://localhost:8761/eureka

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

Here, currency-conversion-service want to connect with the Eureka naming server.

@EnableDiscoveryClient Annotation must be used over the microservices which wants to register themselves to the Eureka Server.

**Zuul API**

Typically there are 100’s of microservices which can have a common features that we want to implement in every microservices.

@EnableZuulProxy is used

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

**public** **class** NetflixZuulApiGatewayServerApplication {

**public** **static** **void** main(String[] args) {

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

}

}

We have to build a microservice for Zuul Api.

As this is a microservice and we want to register it to Eureka Naming Server, we have to use Annotation @EnableDiscoveryClient which will make it enable to get register to the Naming server

**application.properties**

spring.application.name=netflix-zuul-api-gateway-server

server.port=8765

eureka.client.service-url.default-zone=http://localhost:8761/eureka

Now that we have configured Zuul API, it will intercept all the microservices if we either used it in @FeignClient(name = "netflix-zuul-api-gateway-server") or explicity wrap the rest endpoint like this

http://localhost:8100/currency-exchange/from/USD/to/INR

**TO**

http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR

here, in localhost:8765/currency-exchange-service, localhost:8765 is Zuul API microservice PORT and currency-exchange-service is name of the microservice

As we have created FeignClient for currency-exchange-service, we can pass

@FeignClient(name = "netflix-zuul-api-gateway-server")

Instead of

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

@FeignClient(name = "netflix-zuul-api-gateway-server")

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

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

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

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

}

**ZuulLoggingFilter.java**

@Component

**public** **class** ZuulLoggingFilter **extends** ZuulFilter{

**private** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());

@Override

**public** **boolean** shouldFilter() {

**return** **true**;

}

@Override

**public** Object run() {

HttpServletRequest request =

RequestContext.*getCurrentContext*().getRequest();

logger.info("request -> {} request uri -> {}",

request, request.getRequestURI());

**return** **null**;

}

@Override

**public** String filterType() {

**return** "pre";

}

@Override

**public** **int** filterOrder() {

**return** 1;

}}

**ZipKin Distributive System**

There are so many microservices and hence there is a need of centralized logging system. For this the concept of distributive systems come.

We use Spring Cloud Sleuth for assing a unique id to all the requests.

All the logs from the servies are sent to RabbitMQ and from RabbitMQ to Zipkin server where it is consolidated and you can look into it.

You have to download zipkin jar seperately via link:

https://zipkin.io/quickstart.sh

In windows command promt you have to run

C:\NotBackedUp\downloads> SET RABBIT\_URI=amqp://localhost

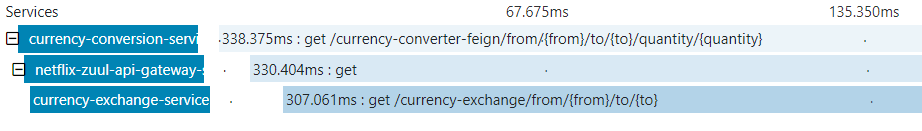
C:\NotBackedUp\downloads> java –jar zipkin-server-2.12.0.exec.jar

**It runs on:**

http://localhost:9411/zipkin

* Suppose, If one microservice makes use of another microservice i.e, one microservice makes use of the rest endpoint exposed by the another microservice. In small scenerio it can be possible to look at the log in the component level of microservices but in real time there are 100’sof microservices and onr microservice can make use of 10 other microservice internally.
* To remember about which microservice is making use of which other microservices is a very difficult task. If somehow in between some microservice is giving an error and we want to know exactly which microservice is a culprit, there is no way to know directly but by digging all the logs as well as lots of time will be wasted.
* For tracing the microservices flow we make use of Zipkin distributive system which is nothing but central logging system and also a UI dashboard for tracing the microservices flow as well as all the logs.
* In the zipkin UI dashboard you can know the flow of microservices. For eg if currency-conversion-service calls currency-exchange-service via zuul-gateway we can see the flow as:

1. currency-conversion-service
2. zuul-gateway
3. currency-exchange-service



If any error comes, we can also see that here only. So, it becomes very easy to get a track of microservices.