Feign REST Client for Service Invocation :

FeignClient also known as Spring Cloud OpenFeign is a **Declarative REST Client** in Spring Boot Web Application. But what do you mean by Declarative REST Client? It means we need to specify the client specification as an Interface and Spring Boot will take care of the implementation for us. Writing web services with the help of FeignClient is very easier. FeignClient is mostly used to consume [REST API](https://www.geeksforgeeks.org/rest-api-introduction/) endpoints which are exposed by third-party or microservice.

Without Feign, in Spring Boot application, we use **RestTemplate** to call the User service. To use the Feign, we need to add **spring-cloud-starter-openfeign** dependency in the pom.xml file.

/setting variables to currency exchange service

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

uriVariables.put("from", from);

uriVariables.put("to", to);

//calling the currency-exchange-service

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

CurrencyConversionBean response = responseEntity.getBody();

1. **Dependency:**

**<dependency>**

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

**<artifactId>**spring-cloud-starter-feign**</artifactId>**

**<version>**1.4.4.RELEASE**</version>**

**</dependency>**

1. **Add Annotation:**

**@EnableFeignClients**in the**CurrencyConversionServiceApplication.java**file

@SpringBootApplication

@EnableFeignClients("com.demo.microservices.currencyconversionservice")

**public** **class** CurrencyConversionServiceApplication {

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

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

}

}

1. Create a **Feign proxy** that enables us to talk to external microservices. Let’s create an interface with the name **CurrencyExchangeServiceProxy.**

And add an annotation **@FeignClient.**Pass the attributes **name** and **URL**.

In the **name** attribute, write the name of the service that we are going to consume. In our case, we are going to consume **currency-exchange-service**. In the **URL** attribute, write the port on which the currency-exchange-service is running.

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

**Example:**

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

**public** **interface** CurrencyExchangeServiceClient {

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

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

}

1. we need to define a method that talks to the **currency-exchange-controller**. Open the **Currency-ConverterController.java**file. Copy the **currency-converter** mapping and paste it in the same file.

@GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}") // where {from} and {to} represents the // column//returns a bean back

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

//Call api using feign

CurrencyConversionBean response = client.convertCurrencyFeign(from, to);

//creating a new response bean and getting the response back and taking it into Bean

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

}

1. **Change the mapping name to /currency-converter- feign/from/{from}/to/{to}/quantity/{quantity} and the method name to convertCurrencyFeign.**
2. Make the use of **CurrencyExchangeServiceProxy**and autowired it.

@Autowired

**private** CurrencyExchangeServiceProxy proxy;

1. run the **currency-exchange-service** by invoking the URL <http://localhost:8000/currency-exchange/from/USD/to/INR> after that run the **currency-conversion-service** by using the URL <http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000>.
2. Execute the feign service by using the URL <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>. It returns the same response as currency-converter-service.

Feign is a REST Service client. Feign can call the RESTful web services easily. When we use the RestTemplate to call the RESTful service, it creates **duplication** of code that talks to RESTful services.

When we define Feign, we need only to define a proxy and define a single method into it. Feign helps us to simplify client code to talk to the RESTful web services.

**Senario:**

Consider a scenario in which currency-exchange-service offers fifteen different services. All the details related to these services must be defined in one place that is **CurrencyExchangeServiceProxy**interface.

# Client-Side Load Balancing with Ribbon:

Netflix Ribbon is a Part of **Netflix Open Source Software** (Netflix OSS). It is a cloud library that provides the **client-side load balancing**. It automatically interacts with **Netflix Service Discovery** (Eureka) because it is a member of the Netflix family.

The Ribbon mainly provides client-side load balancing algorithms. It is a client-side load balancer that provides control over the behavior of **HTTP** and **TCP** client. The important point is that when we use **Feign**, the **Ribbon** also applies.

## Features of Ribbon

* Load balancing
* Fault tolerance
* Multiple protocol support in Asynchronous model
* Caching and batching

## Modules

* **ribbon:** It is an API that integrates **load balancing, fault-tolerance, caching**
* **ribbon-loadbalancer:** It is a Load balancer API that can be used independently or with other modules.
* **ribbon eureka:** It uses **Eureka** client that provides a dynamic server list for the Spring Cloud.
* **ribbon-transport:** It is a transport client that supports **HTTP, TCP,** and **UDP** These protocols use **RxNetty** with load balancing capability.
* **ribbon-httpclient:** It is a REST client built on top of Apache HttpClient integrated with load balancers.
* **ribbon-core:** It is a Client Configuration API.

## Types of Load Balancing:

There are two types of load balancing

* **Server Side Load Balancing:** Server side load balancing is a **monolithic** It applies between the client and the server. It accepts incoming network, application traffic, and distributes the traffic across the multiple backend servers by using various methods. The middle component is responsible for distributing the client requests to the server.
* **Client-Side Load Balancing:**The client holds the list of server’s IPs so that it can deliver the requests. The client selects an IP from the list, randomly, and forwards the request to the server.

1. **Dependency:**

Add the dependency in **currency-conversion-service.**

<dependency>

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

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

1. Open the **CurrencyExchangeServiceProxy.java**file. Enable **Ribbon** by adding an annotation **@RibbonClient**and specify the name of the service which we want to talk to. Ribbon client provide the declarative configuration for a client.

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

//enabling feign

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

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

//enabling ribbon

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

**public** **interface** CurrencyExchangeServiceClient {

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

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

}

**Note:**

In the annotation **@FeignClient,** remove the attribute **URL**. Because we do not need to talk with one particular service. We will configure that URL in the **application.properties** file.

1. Open the **application.properties** file of the project **currency-conversion- service**and configure the servers. The property that we have to configure is:

name-of-the-application.ribbon.listOfServers=URLs

**Ex:**

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

## Running Client Side Load Balancing with Ribbon

1. Run the CurrencyExchangeServiceApplication on port **8000** and then run the CurrencyExchangeServiceApplication on port **8001**.
2. After running the CurrencyExchangeServiceApplication on both the ports, run the **CurrencyConversionServiceApplication.java**

request <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/10000>

Now, refresh the page. We get the same response except for the port number and quantity because we have changed the quantity in the request.

Then you can see service running on port 8001 represents that the currency-exchange-service is running on port 8001 and handling the current request.

# Eureka Naming Server

# **Eureka naming server** is a REST-based server that is used in the **AWS Cloud** services for load balancing and failover of middle-tier services.

# Eureka naming server is an application that holds information about all client service applications. Each microservice registers itself with the Eureka naming server. The naming server registers the client services with their ****port numbers**** and ****IP addresses****. It is also known as ****Discovery Server.****  Eureka naming server comes with the bundle of Spring Cloud. It runs on the default port ****8761.****

## The need of naming server:

We have another load balancer application, which is known as **AWS Cloud**. Because of its inherent nature, server ups and down. There is no middle-tier load balancer. Eureka naming server fills the gap between the **client** and the **middle tier load balancer.**

**Ex:** Suppose that we want to start another instance of currency-exchange-service that is CurrencyExchangeService3 and launch it on port 8002. Here a question arises, will ribbon be able to distribute the load to it?

If the Ribbon wants to distribute the load to the new server, we need to add it to the configuration. Based on the load, we can increase or decrease the number of instances of the services.

If we keep on changing in the CurrencyCalulationService, based on how many CurrenyExchangeService are active right, then, it becomes very difficult to maintain.

The Eureka naming server comes into existence when we want to make maintenance easier. All the instances of all microservices will be register with the Eureka naming server. Whenever a new instance of a microservice comes up, it would register itself with the Eureka naming server. The registration of microservice with the naming server is called Service Registration.

Whenever a service wants to talk with another service, suppose CurrencyCalculationService wants to talk to the CurrencyExchangeService. The CurrencyCalculationService first talk with the Eureka naming server. The naming server provides the instances of CurrencyExchangeService that are currently running. The process of providing instances to other services is called **Service Discovery.**

**Service registration** and **service discovery**are the two important features of the naming server. In the next step, we will set up a Eureka naming server.

## Setting up Eureka naming server :

* 1. Enable Eureka naming server by using an annotation **@EnableEurekaServer**.

@SpringBootApplication

@EnableEurekaServer

**public** **class** NetflixEurekaNamingServerApplication {

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

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

}

}

* 1. Open the **application.properties** file and configure the **application name, port,**and**eureka server**.

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

server.port=8761

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

eureka.client.fetch-registry=**false**

# Connecting Microservices to Eureka naming server:

# Select the ****currency-conversion-service****project.

# Open the ****pom.xml**** file and add the ****eureka-client**** dependency****.****

**<dependency>**

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

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

**</dependency>**

1. Open **CurrencyConversionServiceApplication.java** file and enable **discovery client** by using the annotation **@EnableDiscoveryClient**.

@SpringBootApplication

@EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication   {

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

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

} }

1. Open the **application.properties** file and configure the URL for the Eureka naming 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>

1. Run the **CurrencyConversionServiceApplication.java**file.
2. Open the browser and **refresh** the Eureka server page. It shows the instances of registered microservices.



We see that an instance of currency-conversion-service is registered with the Eureka naming server, and running on port **8100**.

Similar steps follow for other services…..