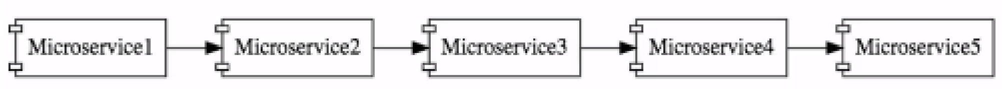
Microservices

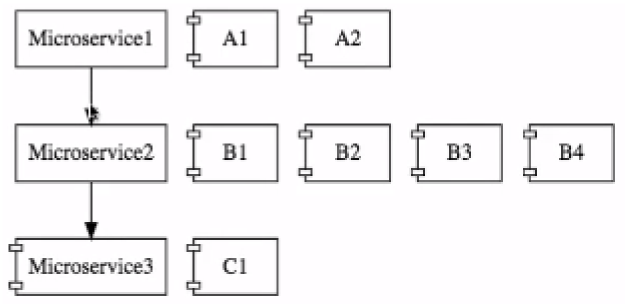
The Microservice architectural style is an approach to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms , often an HTTP resource API

It is service exposed as REST.

Small independent deployable units

Cloud enabled - means multiple instances of services present. If more load on a microservice we can bring more instances automatically. (Autoscaling property of cloud)





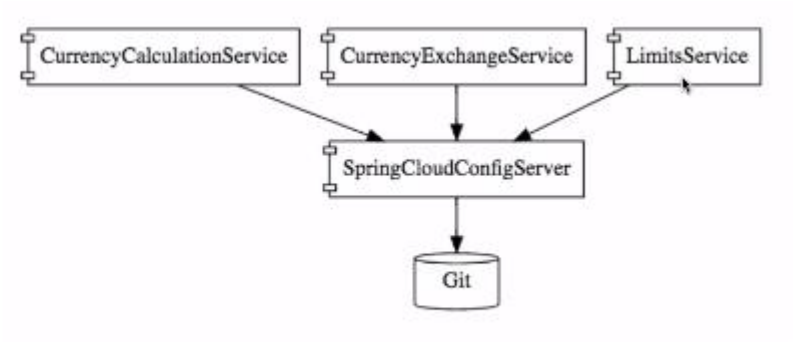
There are some challenges with Microservices-

- **<u>Bounded Context</u>** – As we know main motive of building microservices is instead of develop a huge application building a number of small microservices. How many microservice we should build? How to decide this boundary? What every microservice should do? It requires a lot of time and tremendous Business Knowledge.
- **<u>Configuration Management</u>** – An application might have a number of microservices, each in different environment and each having n numbers of instances to balance load in cloud oriented infrastructure. Let's say 100 of microservices in 5 environment and having 20 instances requires a ton of configurations.
- **<u>Dynamic scale up and scale down</u>** – At a time I need more instances of a microservice but minimum instance of another microservice. All this dynamic load balancing.
- **<u>Visibility</u>** – Suppose you have 20 number of microservices and there is a bug. How to find it? There should have a centralized log system and monitor which request failed and why? . A simple request might use 10s of microservice. How to find bug? How to know which microservice is failing?
- **<u>Fault tolerance</u>** - In microservice architecture one microservice calling another and that is calling another. There will be some fundamental microservice which will be called every time if that is down, Entire application may down due to the one microservice.

Introduction to Spring Cloud

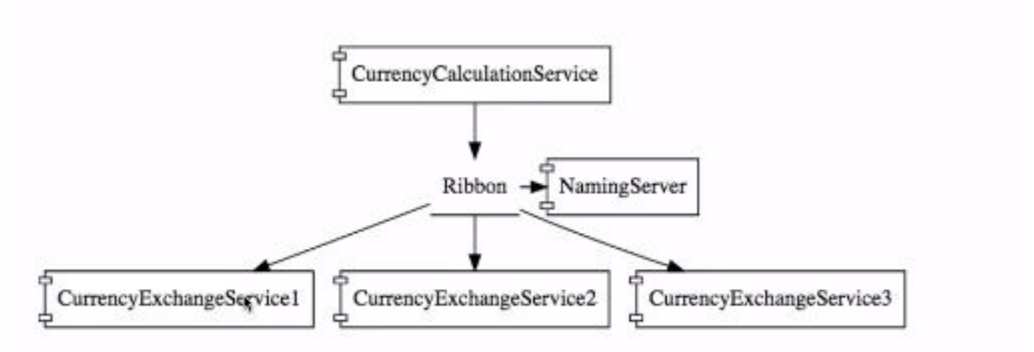
What challenges we discussed earlier can be resolved using Spring Cloud.

- Configuration Management – Spring Cloud Config Server provides an approach where we can store all configurations for different environment for different microservice in one place Git Repository. It will work like central storage and configuration.



Spring Cloud Config Server

- Dynamic scale up and Scale down – Can be achieved using Ribbon Load Balancing.
  We will be using



Ribbon Load Balancing

  - Naming Server (Eureka) – All instances of all microservice should register to naming server. It has special feature service registry where all microservice will be registered.
  - Ribbon (Client Side Load Balancing)- Helps to build dynamic relationship between microservices. It will provide current instance of CurrencyExchangeService (url)to CurrencyCalculationService.
  - Feign (Easier REST Client)- used to write rest client.
- Visibility and Monitoring –
  - Zipking Distributed Tracing- will be used to trace a request across multiple microservices.
  - Netflix API Gateway- There are some common features to all microservices like loggin, security etc. it should not be implemented to all microervices. API gateway provides very good support for it. Here we will be using Netflix API Gateway.
  - Fault Tolerance – To monitor if services are down. Hystrix will be used for it.

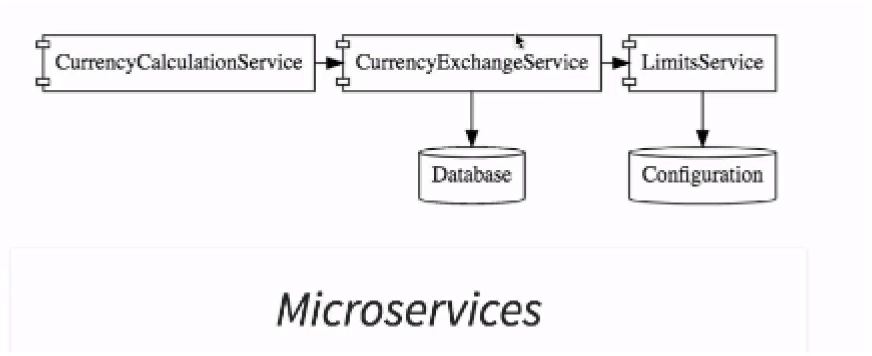
Advantages of Microservice

- New Technology Adaptation – it helps to adapt new technologies. In an application microservices may be in different programming language communicating with each other.
- Dynamic Scaling – If microservice is cloud enabled. It can be achieved easily.
- Faster Release Cycle – Since we are developing small components, microservice can be released at fast pace which is not possible in monolith applications means we can bring new features to market regularly.

Port Standardization

Since we will be using a lot of servers and services to it is better to standard the ports.

Example

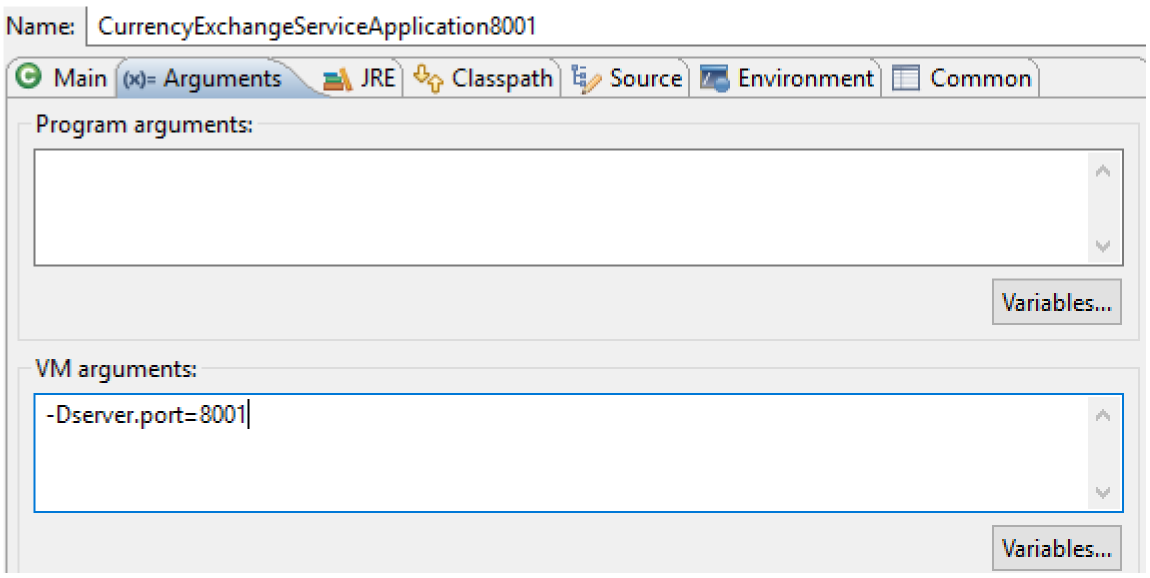


Setting Dynamic Port in Response

Go to Run Configurations

Create a duplicate Application you want to run

Set port number in VM argument as : -Dserver.port=8001

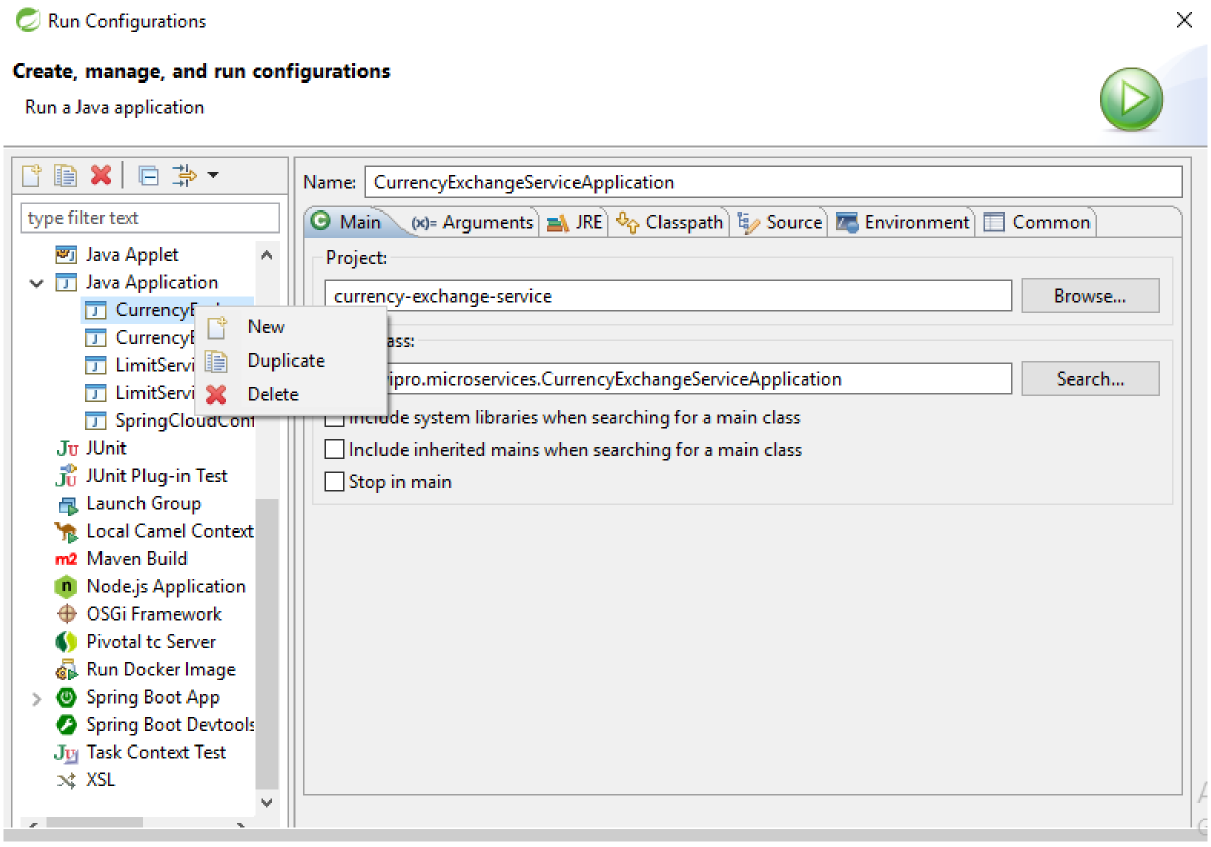


Name: CurrencyExchangeServiceApplication8001

Program arguments:

VM arguments:

```
-Dserver.port=8001
```


Run Configurations

Create, manage, and run configurations

Run a Java application

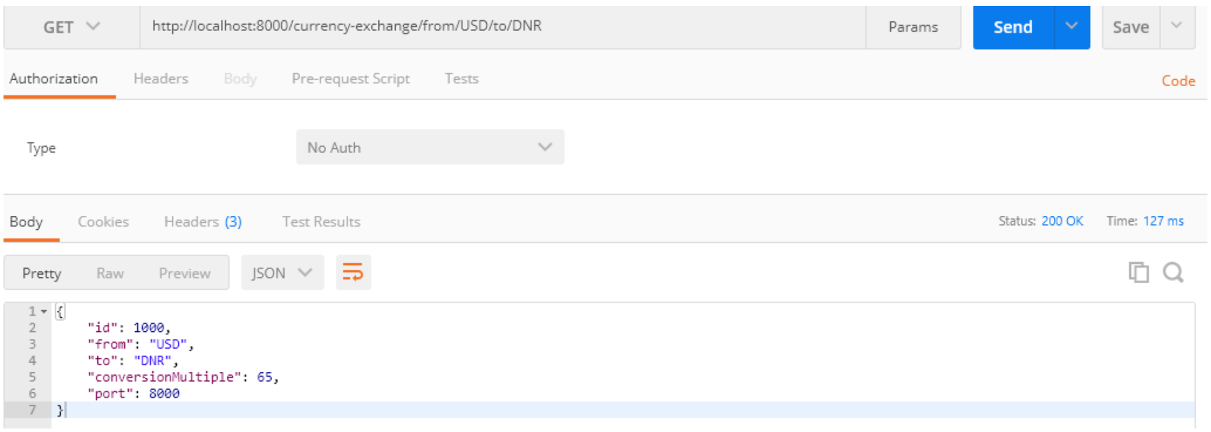
Name: CurrencyExchangeServiceApplication

Project: currency-exchange-service

ass: ipro.microservices.CurrencyExchangeServiceApplication

☐ Include system libraries when searching for a main class
☐ Include inherited mains when searching for a main class
☐ Stop in main

Output- (port number 8000- This is hard coded in ExchangeValue Bean)

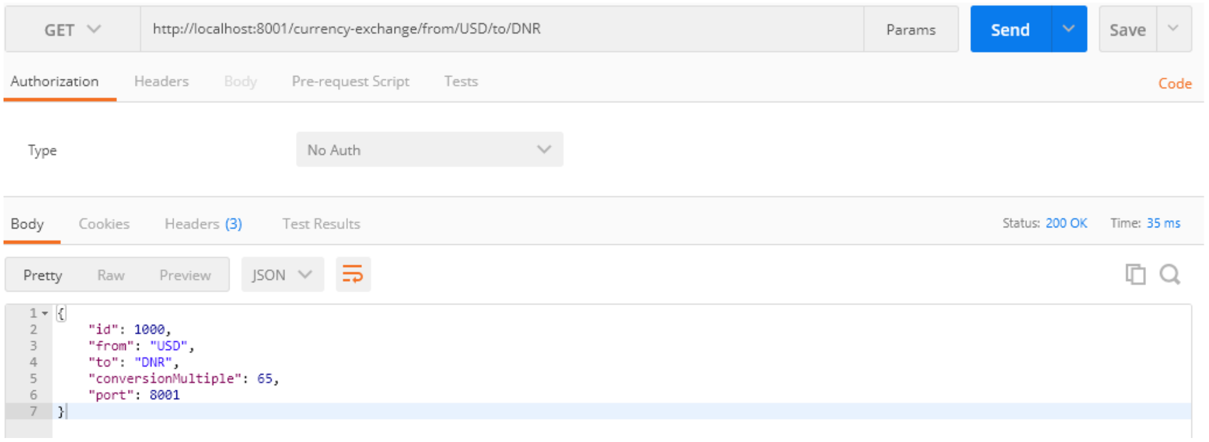

GET http://localhost:8000/currency-exchange/from/USD/to/DNR

Type: No Auth

Status: 200 OK Time: 127 ms

```
{
    "id": 1000,
    "from": "USD",
    "to": "DNR",
    "conversionMultiple": 65,
    "port": 8000
}
```

Port- 8081 set in VM arguments dynamically


GET http://localhost:8001/currency-exchange/from/USD/to/DNR

Type: No Auth

Status: 200 OK Time: 35 ms

```
{
    "id": 1000,
    "from": "USD",
    "to": "DNR",
    "conversionMultiple": 65,
    "port": 8001
}
```

Invoking Microservices using Feign Client

- Feign is the one of the component that spring cloud inherits from Netflix.
- It is a REST service client.
- Add the dependency in pom.xml
- Enable Feign to scan clients. It is enabled in Entry point of application.

```
        @EnableFeignClients("com.wipro.microservices")
/*Here we will pass package where it will scan for Feign Rest Clients
where we will we invoking other microservices.*/
```

- Create a Feign proxy (It is an Interface) to talk with external microservice like we have created JPA repository to talk with JPA.

```
@FeignClient(name="currency-exchange-service",url="localhost:8000")
/*
* if you notice here url of currency-exchange-service is hard coded
 * means it can talk to only one instance of it. But when load is more how it will
be balanced? In demo we will create 4 instance of it and balance the load.
 * Here comes Ribbon Load Balancing concept which we will be configuring in next
demo
 name- Name of microservice which will be invoked--this name you will find
application.properties file of that application
 *url-url of the microservice
  */
public interface CurrencyExchangeServiceProxy {

        @GetMapping("/currency-exchange/from/{from}/to/{to}")
        public CurrencyConversionBean retrieveExchangeValue(@PathVariable
("from")String from, @PathVariable ("to")String to);
        /*
         * if we use  CurrencyConversionBean retrieveExchangeValue(@PathVariable
String from, @PathVariable String to) Param missing exception is thrown;
         * if name is missing then java.lang.IllegalStateException: Either 'name'
or 'value' must be provided/or in @FeignClient is thrown
         * */
}
}
```

- Now invoke from Controller (CurrencyConverterController in our example)

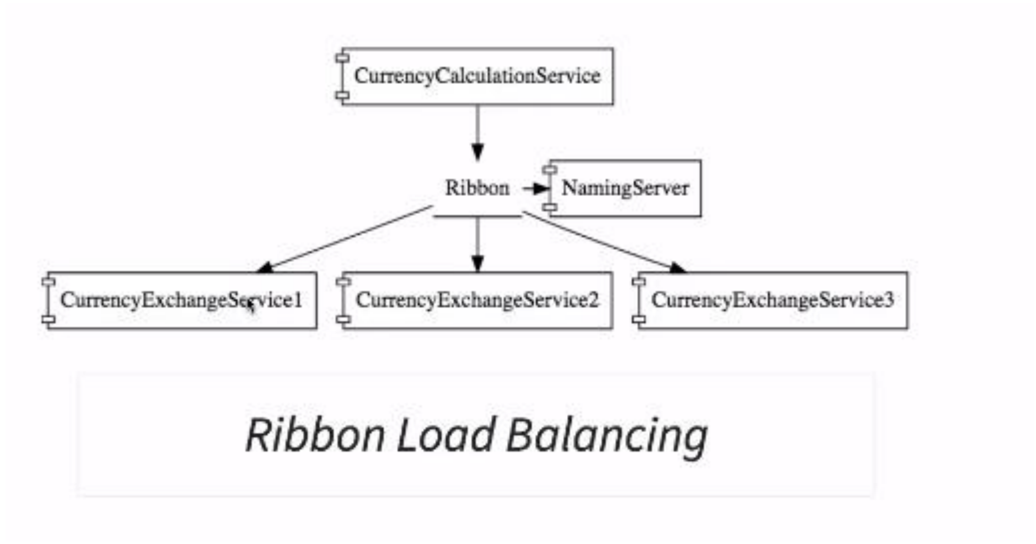
```
/*Invoking using Feign Client*/
        @GetMapping("/currency-conversion-
feign/from/{from}/to/{to}/quantity/{quantity}")
        public CurrencyConversionBean convertCurrencyFeign(@PathVariable String
from,
                        @PathVariable String to,
                        @PathVariable BigDecimal quantity)
        {
                CurrencyConversionBean response =
currencyExchangeServiceProxy.retrieveExchangeValue(from, to);
                return new CurrencyConversionBean(response.getId(), from, to,
response.getConversionMultiple(),
                                quantity,
quantity.multiply(response.getConversionMultiple()), response.getPort());
/*response.getPort()-- Port on which CurrencyExchangeService instance is picked up
                 * More clarity when we configure load balancing- port will change
dynamically
                 * */
        }
```

Ribbon Load Balancing

if you notice url of currency-exchange-service in previous example is hard coded

means it can talk to only one instance of it. But when load is more how it will be balanced?

Here comes Ribbon Load Balancing concept which we will be configuring in next demo.



In our example CurrencyConverterService will be taking to 4 instances of CurrencyExchangeService so we will be configuring Ribbon on CurrencyConverterService. Let's do set up for load balancing.

- Add Ribbon Dependency in pom.xml
- Enable Ribbon Client on Proxy Interface we created earlier

```
@RibbonClient(name="currency-exchange-service")//urls configured in properties
file
   public interface CurrencyExchangeServiceProxy {
```

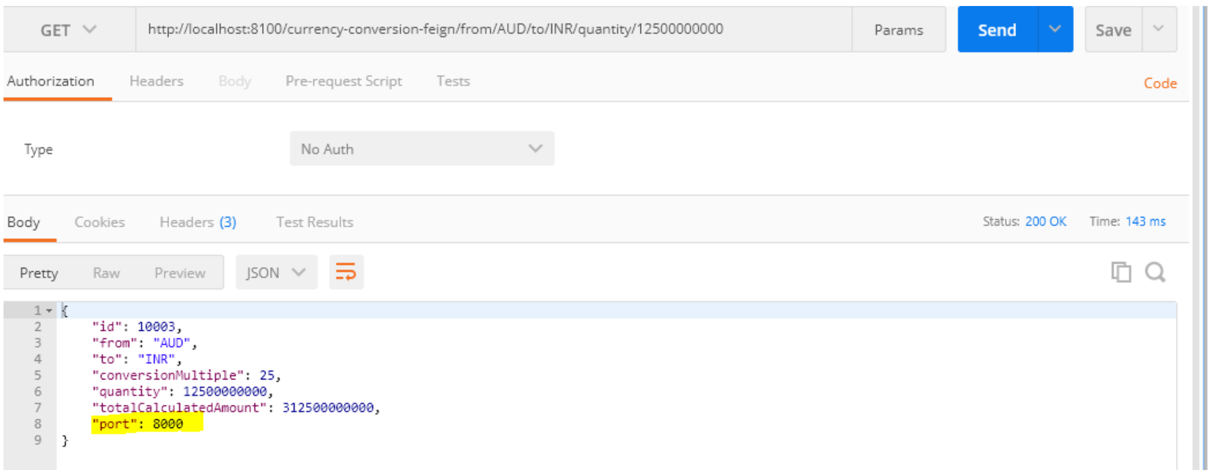
- Configure these in properties file of CurrencyConverterService

```
spring.application.name=currency-conversion-service
server.port=8100
#This microservice is communicating to currency-exchange-service
#and it has multiple instances for load balancing
#Let's configure it here separated by comma.
currency-exchange-
service.ribbon.listOfServers=http://localhost:8000,http://localhost:8001
```

Note : If we configure list of servers in properties file and forgot to remove hard code url from proxy interface, then server configured in proxy interface will only picked up. (Personal Experience)

Once the set up is done let's execute it and see the response.

CurrencyExchangeService from port 8000



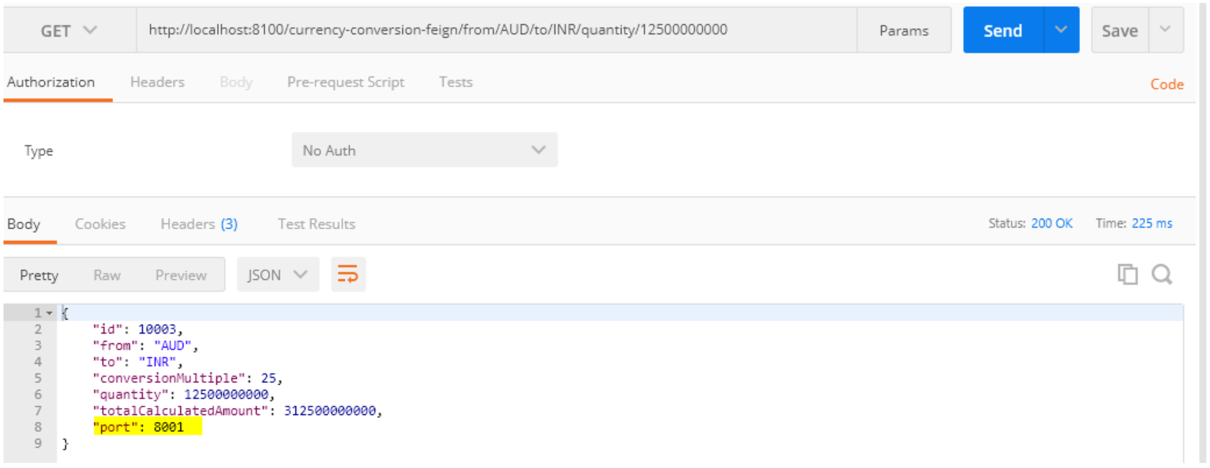
http://localhost:8100/currency-conversion-feign/from/AUD/to/INR/quantity/12500000000

Type: No Auth

Status: 200 OK Time: 143 ms

```
{
    "id": 10003,
    "from": "AUD",
    "to": "INR",
    "conversionMultiple": 25,
    "quantity": 12500000000,
    "totalCalculatedAmount": 312500000000,
    "port": 8000
}
```

CurrencyExchangeService from port 8001



http://localhost:8100/currency-conversion-feign/from/AUD/to/INR/quantity/12500000000

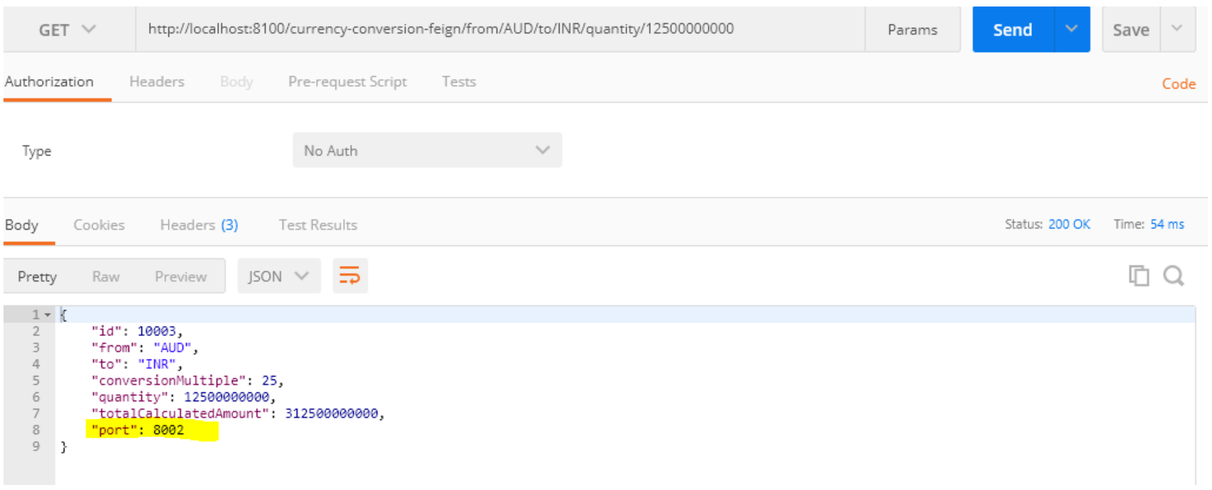
Type: No Auth

Status: 200 OK Time: 225 ms

```
{
    "id": 10003,
    "from": "AUD",
    "to": "INR",
    "conversionMultiple": 25,
    "quantity": 12500000000,
    "totalCalculatedAmount": 312500000000,
    "port": 8001
}
```

When I tried to launch the application, getting response only from ports 8000 and 8001. Not from 8002 though CurrencyExchangeSerivice is up on this port. (Personal Experience)

Reason- This port was not enabled in properties file.



http://localhost:8100/currency-conversion-feign/from/AUD/to/INR/quantity/12500000000

Type: No Auth

Status: 200 OK Time: 54 ms

```
{
    "id": 10003,
    "from": "AUD",
    "to": "INR",
    "conversionMultiple": 25,
    "quantity": 12500000000,
    "totalCalculatedAmount": 312500000000,
    "port": 8002
}
```

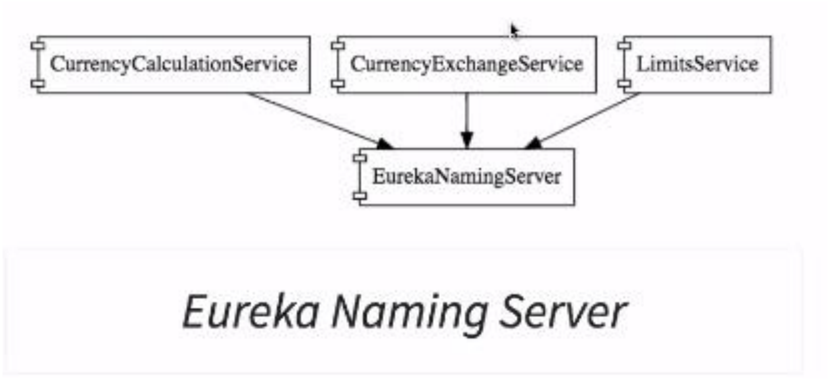
Why Naming Server is Required?

Consider our previous example where we learned load balancing and created three instances of CurrencyExchangeService. Suppose when load is increased more and I need to launch one more instance of it then what I need to do is configure one more port 8003 in properties file.

When our load is very low one or two servers is enough to handle the load why to keep all 4 servers busy. These are some of the issue. If we have to configure this manually every time then it is big maintenances headache. To overcome this naming server comes into picture. The best practice is based on load increase and decrease instances of the microservice dynamically.

All the instances of all the microservices would register with the Eureka Naming Server. This is called service registration. Whenever a microservice wants to talk with another microservice it will ask to naming server that what instances of the other microservices is running now and this is called service discovery.

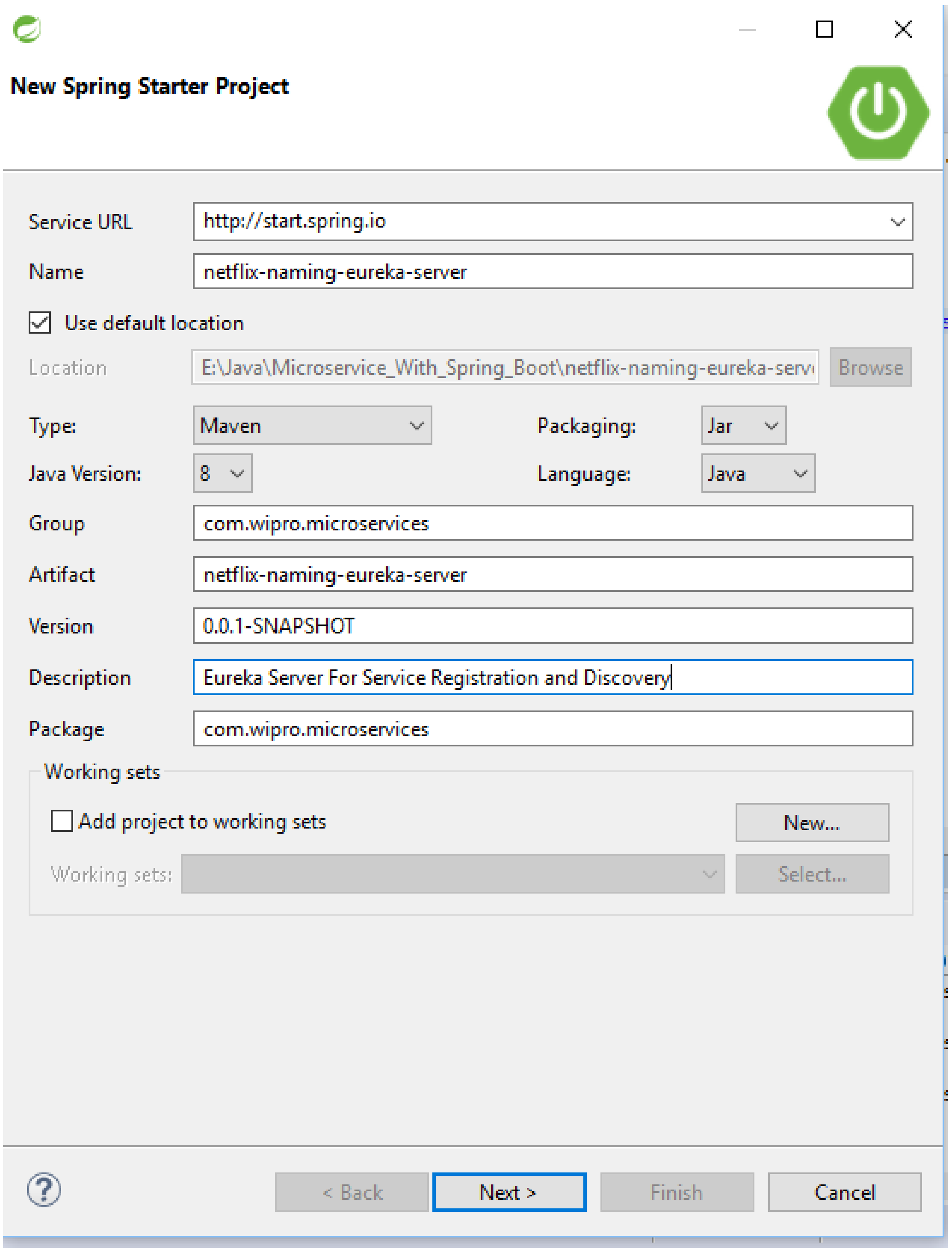
The two-important feature of Naming server is service registration and service discovery. At start up of every application they would register with naming server.



How to set Eureka Naming Server?

A lot of naming servers are present but we are going to use Eureka provided by Netflix.

Let’s Create the eureka server.



New Spring Starter Project

Service URL	http://start.spring.io
Name	netflix-naming-eureka-server

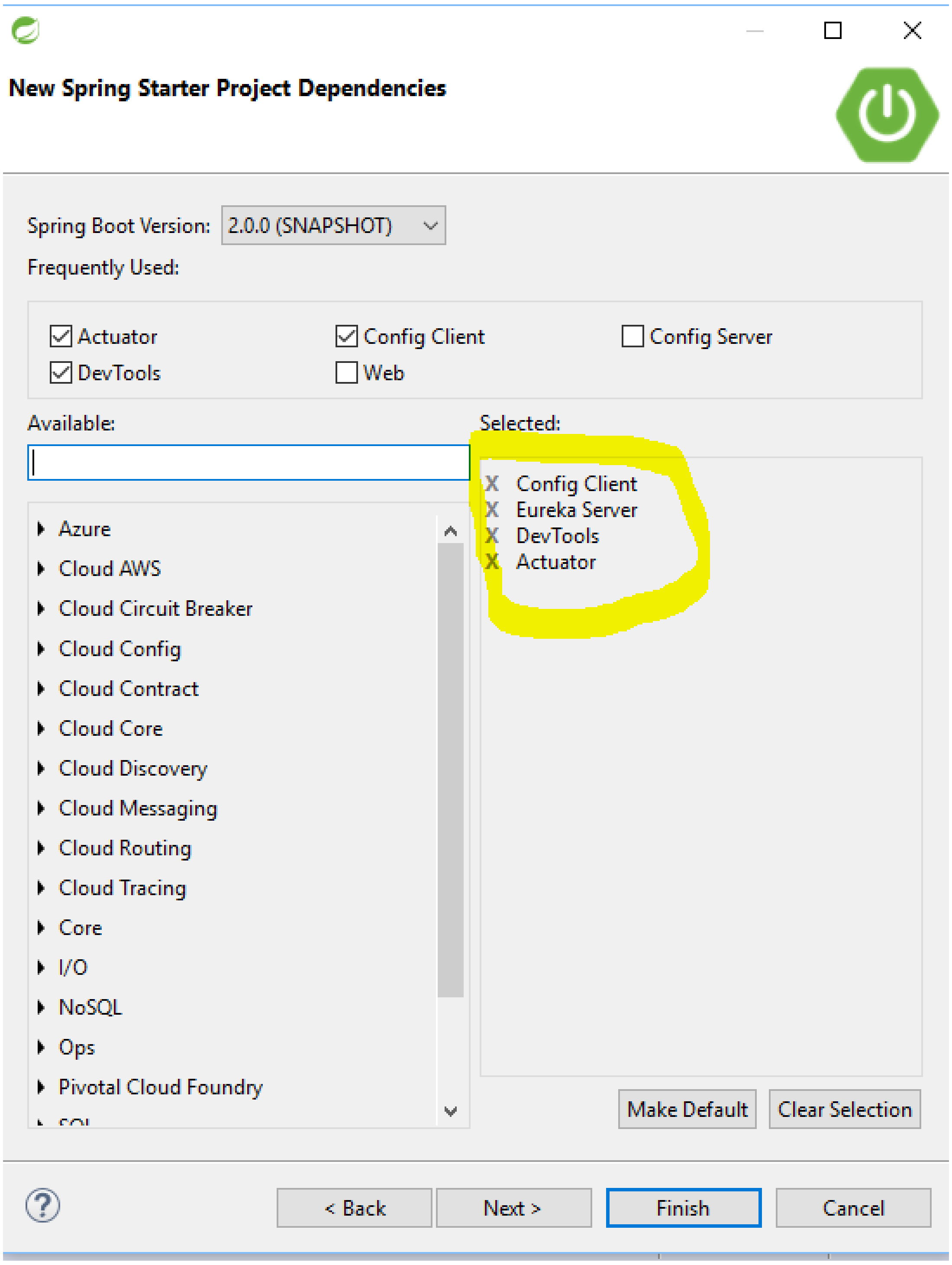
[✓] Use default location

Location	E:\Java\Microservice_With_Spring_Boot\netflix-naming-eureka-serv
Type:	Maven
Packaging:	Jar
Java Version:	8
Language:	Java
Group	com.wipro.microservices
Artifact	netflix-naming-eureka-server
Version	0.0.1-SNAPSHOT
Description	Eureka Server For Service Registration and Discovery
Package	com.wipro.microservices

Working sets

[ ] Add project to working sets

Working sets:


New Spring Starter Project Dependencies

Spring Boot Version: 2.0.0 (SNAPSHOT)

Frequently Used:

[✓] Actuator [✓] Config Client [ ] Config Server
[✓] DevTools [ ] Web

Available:

- Azure
- Cloud AWS
- Cloud Circuit Breaker
- Cloud Config
- Cloud Contract
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- I/O
- NoSQL
- Ops
- Pivotal Cloud Foundry

Selected:

- X Config Client
- X Eureka Server
- X DevTools
- X Actuator

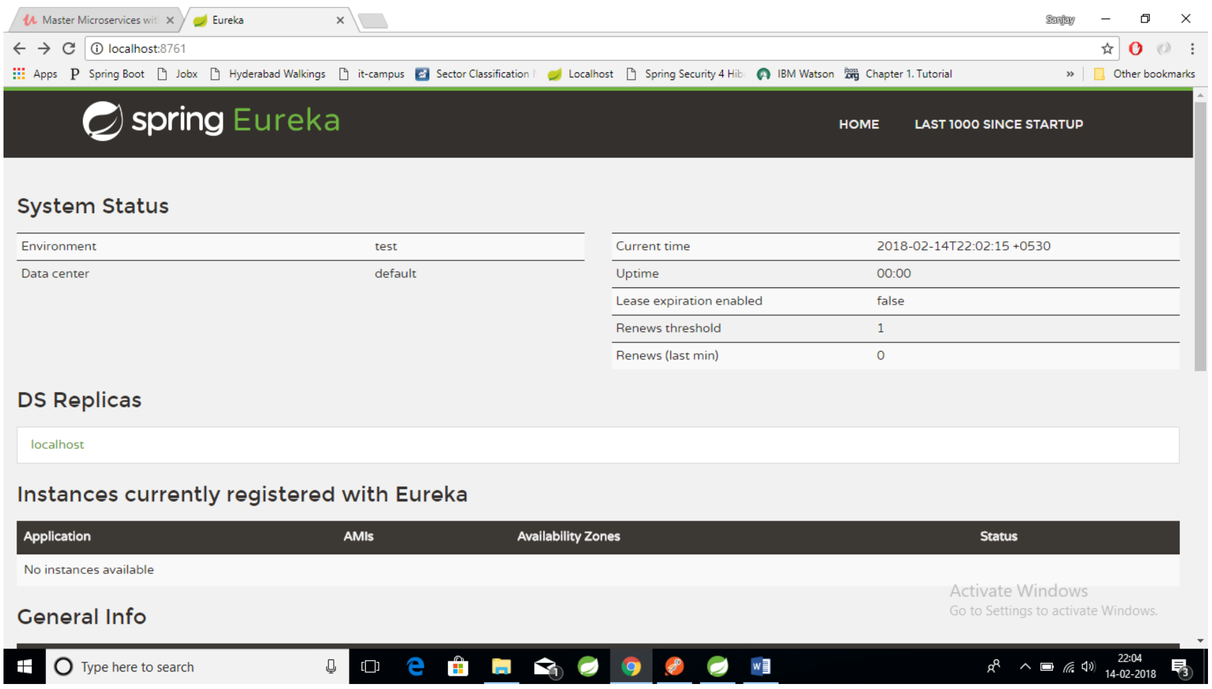
Once created component Enable Eureka Server on main class

```
@EnableEurekaServer
@SpringBootApplication
    public class NetflixNamingEurekaServerApplication {
```

Configure settings in application.properties file.

```
spring.application.name=netflix-naming-eureka-server
server.port=8761
eureka.client.register-with-eureka=false
    eureka.client.fetch-registry=false
```

Now we can launch the server : http://localhost:8761/



System Status

Environment	test
Data center	default

Current time	2018-02-14T22:02:15 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Now we will connect our microservice with Eureka Naming Server

Connecting Currency Conversion Microservice to Eureka

- Add Dependency to pom.xml

```
<!-- Eureka  Dependency-->
                  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.3.RELEASE</version>
       </dependency>
```

- Now we would want that the microservice should register with naming server and this we will be doing by adding an annotation

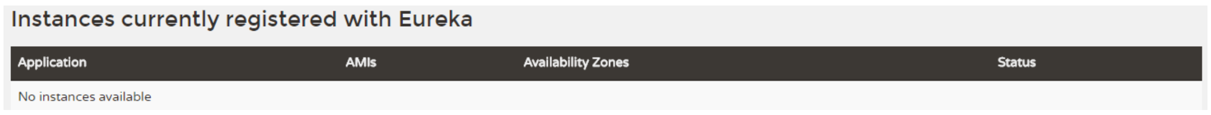
```
@SpringBootApplication
@EnableFeignClients("com.wipro.microservices")
@EnableDiscoveryClient
/*Here we will pass package where it will scan for Feign Rest Clients
where we will we invoking other microservices.*/
public class CurrencyConversionServiceApplication {

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

- Configure properties file of Currency Conversion Microservice

Configure url of Eureka

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

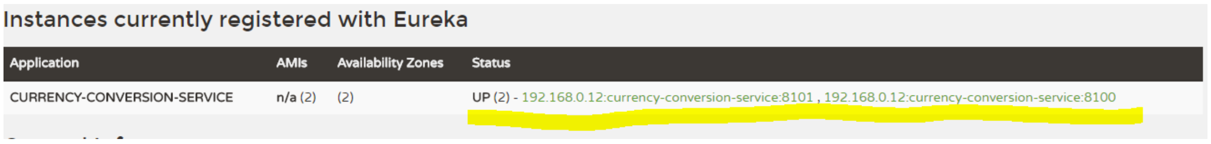
- Before configuring eureka we can see no microservice is registered with eureka



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

- Once you restarts the server you can see it is registered with the naming server.

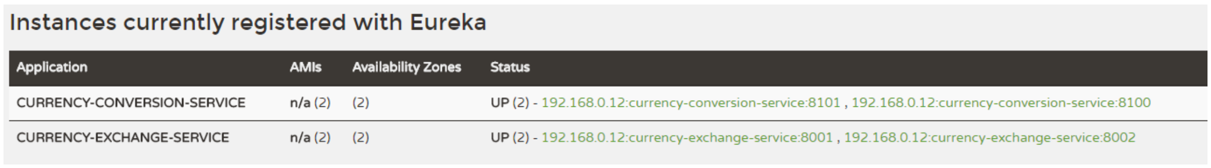


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.12:currency-conversion-service:8101 , 192.168.0.12:currency-conversion-service:8100

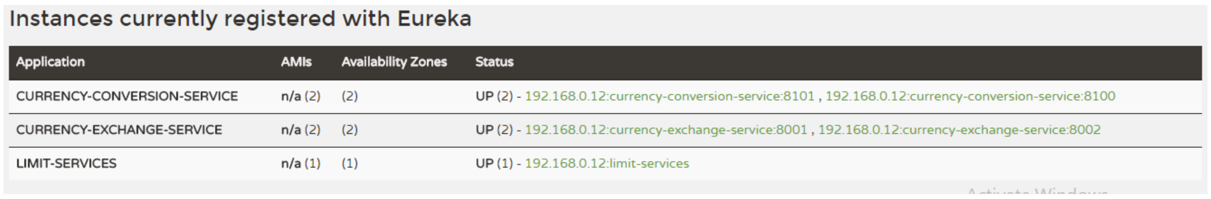
Here we can see two instances are up.

Similarly Currency Exchange and Limit Microservice will be registered with eureka



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.12:currency-conversion-service:8101 , 192.168.0.12:currency-conversion-service:8100
CURRENCY-EXCHANGE-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.12:currency-exchange-service:8001 , 192.168.0.12:currency-exchange-service:8002

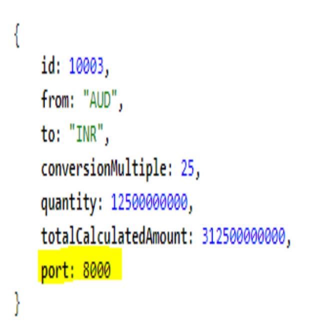


Instances currently registered with Eureka

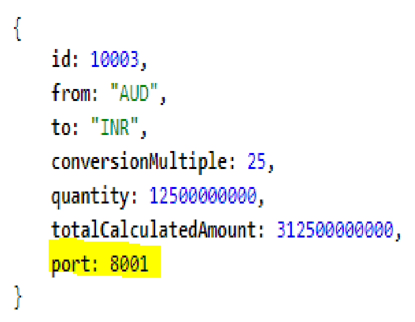
Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.12:currency-conversion-service:8101 , 192.168.0.12:currency-conversion-service:8100
CURRENCY-EXCHANGE-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.12:currency-exchange-service:8001 , 192.168.0.12:currency-exchange-service:8002
LIMIT-SERVICES	n/a (1)	(1)	UP (1) - 192.168.0.12:limit-services

Distributing calls using Eureka and Ribbon

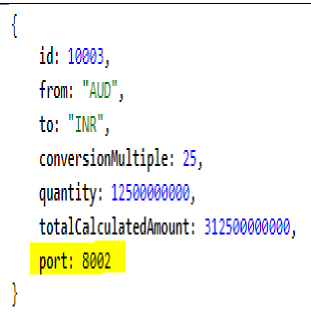
If you remember in last demo (Currency Conversion Service) to divide load of Currency Exchange Service we configured three servers in Currency Conversion Service so that load can be balanced. Now comment that. But create three instances (Please note these three instances are created manually in eclipse for demo purpose, Here we have just removed hard coded port in properties file) of Currency Conversion Service and try to hit the Currency Conversion Service microservice. We will be getting response from all the three servers 8000,8001 and 8002.



```
{
   id: 10003,
   from: "AUD",
   to: "INR",
   conversionMultiple: 25,
   quantity: 12500000000,
   totalCalculatedAmount: 312500000000,
   port: 8000
}
```



```
{
   id: 10003,
   from: "AUD",
   to: "INR",
   conversionMultiple: 25,
   quantity: 12500000000,
   totalCalculatedAmount: 312500000000,
   port: 8001
}
```



```
{
   id: 10003,
   from: "AUD",
   to: "INR",
   conversionMultiple: 25,
   quantity: 12500000000,
   totalCalculatedAmount: 312500000000,
   port: 8002
}
```


If we kill server 8002 it will start picking from 8001 and 8000.Application will not down. Ribbon automatically take from 8000 and 8001.

So increasing and decreasing instances will not affect the microservices.

Introduction to API Gateways

There are some common features we need to implement to all microservices like

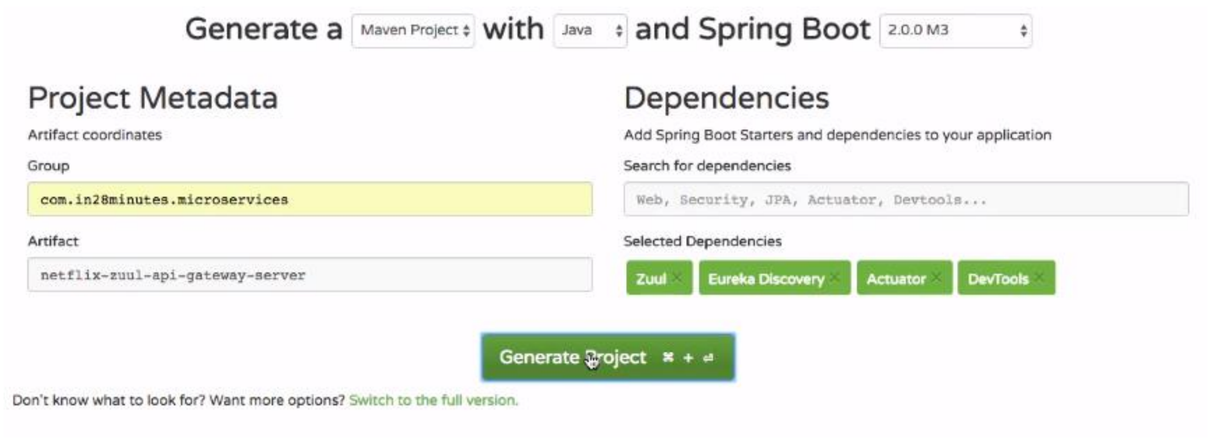
- Authentication, Authorization and Security
- Rate Limits (Restricting certain client to access the service certain times in certain time)
- Fault Toleration- if there is a service and I am dependent and it is not up, it should give some default response.
- Service Aggregation- Combining a number of microservices together and showing as one to external client.

In our previous example microservices were communicating to each other directly. From now onwards they will be routed via API gateways and these gateway will provide common features to the microservices.

In Our example we will be using Zuul API provided by Netflix.

How to do set up for Zuul API Server?

1. Create a Component



Generate a Maven Project with Java and Spring Boot 2.0.0 M3

Project Metadata
Artifact coordinates
Group: com.in28minutes.microservices
Artifact: netflix-zuul-api-gateway-server

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies: Zuul, Eureka Discovery, Actuator, DevTools

Generate Project

Don't know what to look for? Want more options? Switch to the full version.

Eureka Discovery to see zuul instance is up or down

2. Enable Zuul proxy and register it with name server.
3. Configure properties file

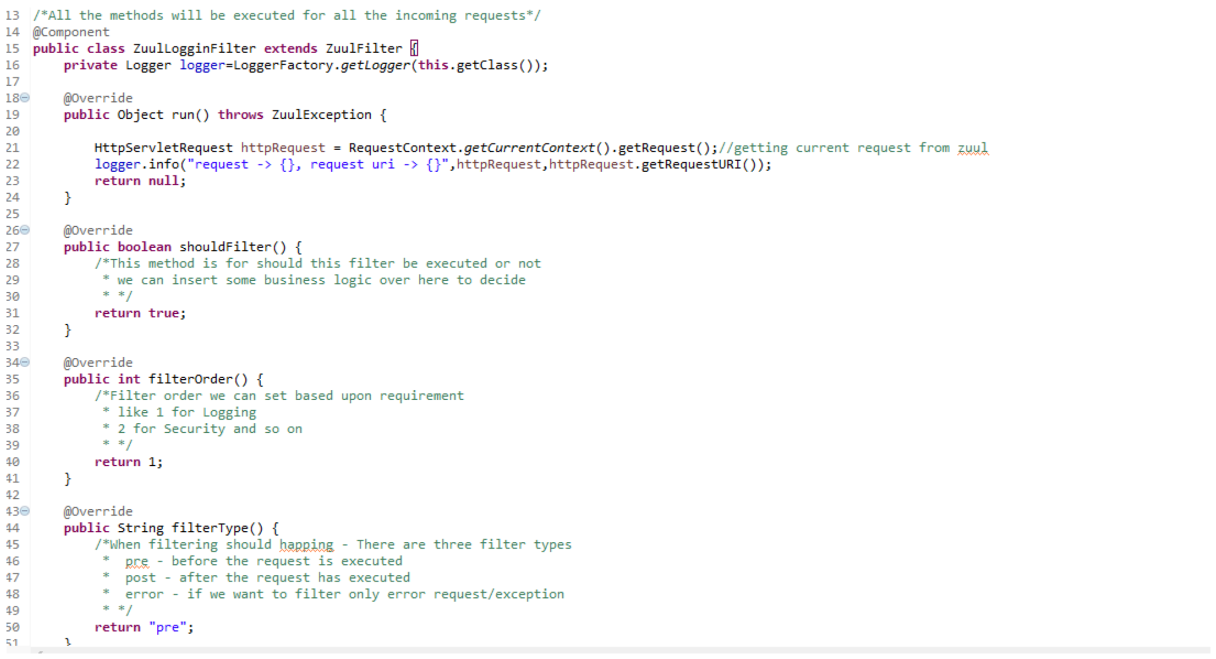
```
spring.application.name=netflix-zuul-api-gateway-server
server.port=8765
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

Now API gateway server is ready however we did not configure what it should do when it intercepts a request. Let's do now.

Implementing Zuul Logging Filter- If any request comes through API gateway, log it.

Here is the steps –

create ZuulLoggingFilter class



```java
/*All the methods will be executed for all the incoming requests*/
@Component
public class ZuulLogginFilter extends ZuulFilter {
    private Logger logger=LoggerFactory.getLogger(this.getClass());

    @Override
    public Object run() throws ZuulException {

        HttpServletRequest httpRequest = RequestContext.getCurrentContext().getRequest();//getting current request from zuul
        logger.info("request -> {}, request uri -> {}",httpRequest,httpRequest.getRequestURI());
        return null;
    }

    @Override
    public boolean shouldFilter() {
        /*This method is for should this filter be executed or not
         * we can insert some business logic over here to decide
         * */
        return true;
    }

    @Override
    public int filterOrder() {
        /*Filter order we can set based upon requirement
         * like 1 for Logging
         * 2 for Security and so on
         * */
        return 1;
    }

    @Override
    public String filterType() {
        /*When filtering should happing - There are three filter types
         *  pre - before the request is executed
         *  post - after the request has executed
         *  error - if we want to filter only error request/exception
         * */
        return "pre";
    }
```

Execute the request through Zuul API Gateway

Suppose we want to execute currency exchange microservice through Zuul API Gateway then just we need to change the url like-

http://localhost:8765/{application_name}/{uri}

http://localhost:8765/- uri of Zuul API server

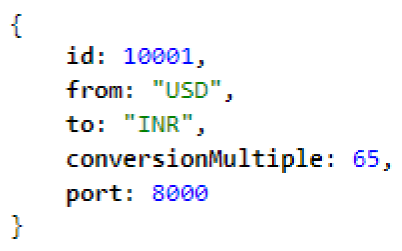
application name- currency-exchange-service

uri-http://localhost:8000/currency-exchange/from/USD/to/INR

after combining-

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

Execute it on browser-



```
{
    id: 10001,
    from: "USD",
    to: "INR",
    conversionMultiple: 65,
    port: 8000
}
```

Zuul API Log

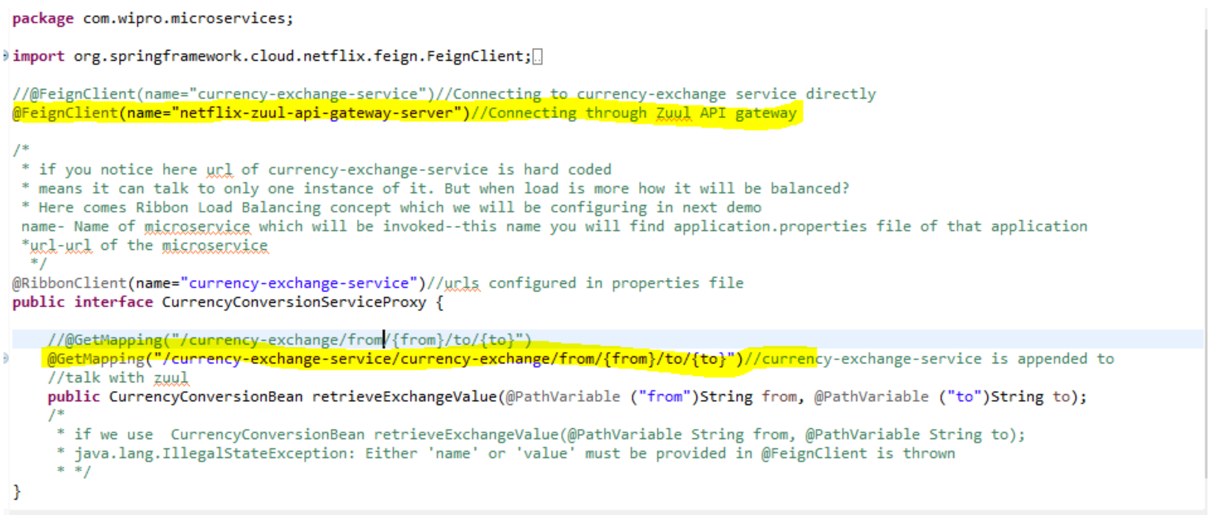
```
2018-02-17 12:13:17.012  INFO 11728 --- [nio-8765-exec-3] c.wipro.microservices.ZuulLogginFilter   : request -> org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@4be674fd, request uri -> /currency-exchange-service/currency-exchange/from/USD/to/INR
```

Invoking Microservices using Zuul API Gateway

Previously we learned how to communicate with microservices directly using feign client.

Now let’s try it using Zuul API Gateway

Modify Proxy-



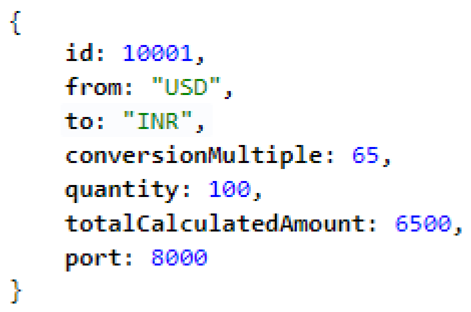
```
package com.wipro.microservices;

import org.springframework.cloud.netflix.feign.FeignClient;

//@FeignClient(name="currency-exchange-service")//Connecting to currency-exchange service directly
@FeignClient(name="netflix-zuul-api-gateway-server")//Connecting through Zuul API gateway

/*
 * if you notice here url of currency-exchange-service is hard coded
 * means it can talk to only one instance of it. But when load is more how it will be balanced?
 * Here comes Ribbon Load Balancing concept which we will be configuring in next demo
 name- Name of microservice which will be invoked--this name you will find application.properties file of that application
 *url-url of the microservice
  */
@RibbonClient(name="currency-exchange-service")//urls configured in properties file
public interface CurrencyConversionServiceProxy {

    //@GetMapping("/currency-exchange/from/{from}/to/{to}")
    @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")//currency-exchange-service is appended to
    //talk with zuul
    public CurrencyConversionBean retrieveExchangeValue(@PathVariable ("from")String from, @PathVariable ("to")String to);
    /*
     * if we use  CurrencyConversionBean retrieveExchangeValue(@PathVariable String from, @PathVariable String to);
     * java.lang.IllegalStateException: Either 'name' or 'value' must be provided in @FeignClient is thrown
     * */
}
```

Note : Zuul uses application name in the url to talk with Eureka and find the url of the service.

Note : Service name is appended to url only while executing the it on browser. It is not appended to service url. Only in proxy it is appended.

Execute-

http://localhost:8765/currency-conversion-service/currency-conversion-feign/from/USD/to/INR/quantity/100



```
{
    id: 10001,
    from: "USD",
    to: "INR",
    conversionMultiple: 65,
    quantity: 100,
    totalCalculatedAmount: 6500,
    port: 8000
}
```

Here Zuul Loggin filter will be invoked twice first time for currency-exchange-service and second time for currency-conversion-service.

Distributed Tracing

It is a centralized system to trace the defects/bugs microservices. Using it we can view what happened with a specific request. To achieve it we are going to use spring cloud sleuth with Zipking.

Every request will be assigned with a unique ID by Spring Cloud Sleuth. All the logs from these all services will put in a MQ server (Rabbit MQ Server) and would be sent out to Zipkin Server.

How to apply Spring cloud Sleuth?

It can be applied to all the microservices.

In our example we are implementing it in currency-exchange-service, conversion service and zuul api server.

First add dependency

```
<!-- Sleuth -->
		<dependency>
			<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-starter-sleuth</artifactId>
		</dependency>
```

And then to trace all the request we will create a Always Sampler in main application.