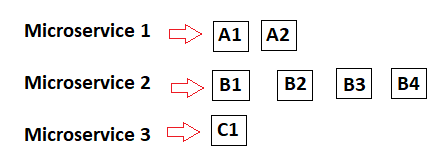
we will understand

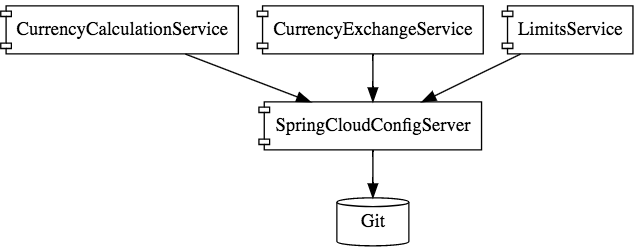
1. how to implement microservices using **Spring Cloud**.
2. how to establish communication between microservices, **enable** **load balancing**, **scaling up and down of microservices**.
3. **centralize the configuration of microservices**with **Spring Cloud Config Server**.
4. implement **Eureka Naming Server** and **Distributed tracing** with **Spring Cloud Sleuth** and **Zipkin**.
5. create fault tolerance microservices with **Zipkin**.

**Challenges**

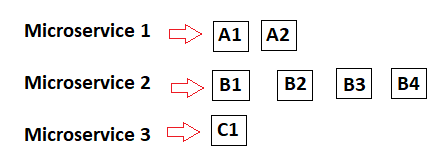
1. **Bounded context (deciding the design boundaries of the microservices)**: We will be developing lots of micro services from a monolithic application. So, while breaking down it to a micro-services, we should know what each one is doing, what should be the boundary for the application
2. **Configuration management** :



We have lot of micro services and each one is having many instances in each environmentrunning up, so every micro service is having lot of configuration. So, it is difficult to manage these configurations Solution for this is ***Spring cloud config server***.



1. **monitoring/visibility:** If all microservices are running and there is a bug, how we identify from which micro service the bug is. Solution is we should have the centralized logs .We should have monitoring on all those services.
2. **dynamic scale up and scale down:**



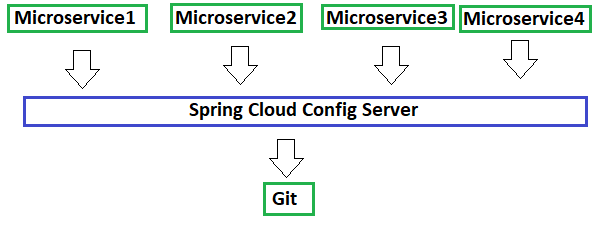
We need to make some instances up on running or down when needed. Also we need to distribute the load among the instanceswhich are up on running. When traffic is high then we should create new instances as required and when traffic is low then we should make some running instances down or we can delete those instances.

1. **pack of cards :**If all microservices are not designed properly then it will be like pack of cards. Generally one microservice calls another one as showed in below figure, if one service goes down, all will fail so that will be like pack of cards. For this solution is *FAULT TOLERENCE.*



**Solution for all above challenges id SPIRNG CLOUD**

1. **Spring Cloud Config server**(Solution for Configuration management)



Here all configuration of all microservices(along with their instances of each environment) will be stored in GIT and Spring cloud Config server exposes those configurations to all above microservices. It provides HTTP resource-based API for external configuration in the distributed system.

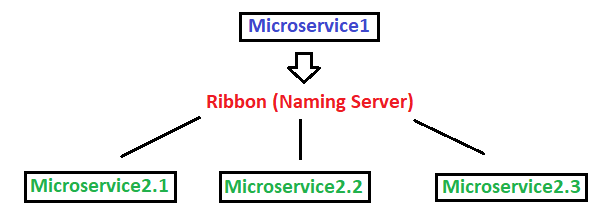
1. **DYNAMIC SCALE UP AND SCAL DOWN**

To overcome the challenges, following things to be fulfilled

1. Instances should be added or removed according to requirement.
2. Dynamically check what are all the instances up on running (the condition is services must be registered at some location from where we can get info)
3. MS1 should distribute Load among all the running instances(MS2 services)

Solution for this is

* 1. Naming server(Eureka) : this will register all the microservices and tells load balancer which are the services up on running
  2. Ribbon (Client side load balancing) : helps to distribute the load dynamically
  3. Feign (Easier Rest Client)



Naming server(Eureka) has 2 features

Service registration: registering each microservices

Service discovery : showing what are the microservices up on running.

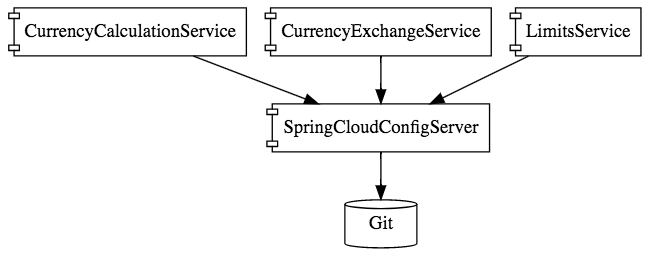
**What is Service Registry in Microservices?**

Service Registry is the process of registering a microservice with Eureka Server. In a nutshell, it acts as a kind of database which stores the details of all microservices involved in the entire application. However, this is also a regular Spring Boot application. In order to enable the service registry, we apply annotation @EnableEurekaServer on the top of the Application’s main class. Moreover, on using Spring Cloud’s annotation @EnableEurekaServer, other microservices can register here and communicate with each other via service discovery.

**What is Service Discovery in Microservices?**

Service Discovery is the process of discovering other microservices in the network to make intra-communication happen. However, a microservice first connects with Eureka to discover other microservice to make communicating with each other. Using Service Discovery, one microservice can connect with the other microservice via Eureka.

All images



### Step 21

**Rest CLient and Feign Client**

****

Disadvantage of **RestTemplate**:

write lot of code to call the microservices

Feign simplifies the above problem and it also has client side load balancing component called ribbon

Steps to use Feign Client

Add **@EnableFeignClient** to starter class and just like we create a repo to call DB, here we need to create a interface (with method ) which calls the other microservices.

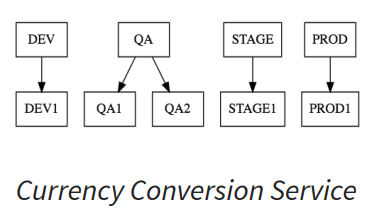
Add **@FeignClient(name="",url="")**

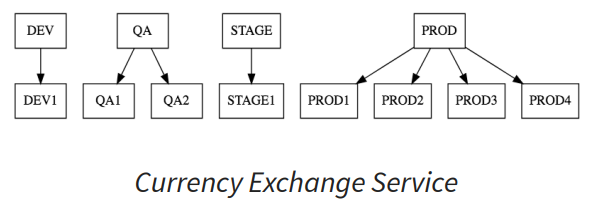
name=> name of the service we are calling from this interface

url=> url of the service we are calling from this interface

this interface contains the method which is same like the method of controller of calling service.

**Services with their instances**

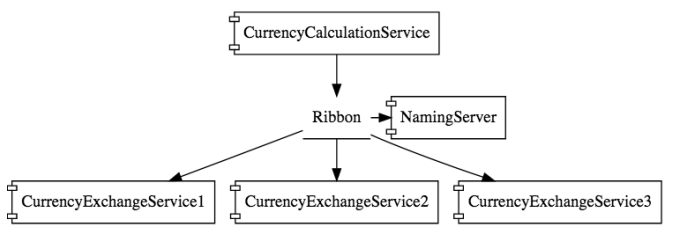




### Step 22 - Setting up client side load balancing with Ribbon

**Why Ribbon?**

Using Feign only we can call only one instance at a time. For ex, (in above figure) PROD1 of CCS can call only one of PROD1/ PROD2/ PROD3/ PROD4 of CES . But if we have to call multiple instances then use **RIBBON**, it will distribute the load among all instances of calling services Given below

****

**Implementation of Ribbon**

Ribbon will distribute the load among all the instances of the called service. For this we need to to following steps

1. add property in property file and write no. Of instances like below

**<nameOFCalledService>.ribbon.listOfServers=http://localhost:8000,http://localhost:8001**

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

1. add @RibbonClient(name="nameOfCalledService") annotation

### Step 23 - Running client side load balancing with Ribbon

**Drawback of Ribbon**

We cannot increase or decrease the instances when we are using Ribbon. Because we need to add or remove the instances in application.properties file. We cannot add or delete instances and distribute the load dynamically using ribbon. That’s why Eureka naming server comes to picture

### Step 25 - Setting up Eureka Naming Server

Refer link for more info: <https://javatechonline.com/how-to-register-discover-microservices-using-netflix-eureka/#eurekaclientregister-with-eurekafalse>

Steps

1. create a project with dependencies
2. Add @EnableEurekaServer on starter class
3. Add following properties in configuration file

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

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

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

Default value of property ‘eureka.client.register-with-eureka’ is true. Please note that this property is mandatory to include in Eureka Server in order to make its value as false. However, this is optional to add in case of other microservices/applications that are not Eureka server. Moreover, every microservice project is connected to Spring Cloud project that provides default value to true. Therefore, We should include ‘eureka.client.register-with-eureka=false’ for one time only in case of Eureka server as Eureka Server itself can’t be registered.

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

This property indicates that Eureka Server is supported to fetch instance details of microservice to make intra-communication between microservices happen. If one microservice wants to communicate with another microservice by using Eureka then inside microservice we should add this property (eureka.client.fetch-registry) and set it to true. However, inside Eureka Server we should include this property with a value as false. However, Eureka server will never try to fetch registry as it is itself having a registry. Hence the value of this property in case of Eureka server will be false. Moreover, every microservice project is connected to Spring Cloud project that provides default value to true.

### Step 26 - Connecting Currency Conversion Microservice to Eureka

Step 1:

In order to make your application/microservice acts as a Eureka discovery client, you need to apply **@EnableEurekaClient** at the main class of your application.

Step 2:

**eureka.client.service-url.default-zone=http://localhost:8761/eureka #Register with Eureka**

It indicates that this service is getting registered with the Eureka Server.

### Step 27 - Connecting Currency Exchange Microservice to Eureka

Same as above

### Step 28 - Distributing calls using Eureka and Ribbon

**Drawback of ribbon:**

W need to hard code number of instance in configuration files as below

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

In order to dynamically create or delete instances, we use eureka naming server. We will make ribbon to talk to naming server.

### Step 29 - A review of implementing Eureka, Ribbon and Feign

<https://javatechonline.com/microservices-in-java/#What_is_Service_Registration_Discovery>

## ZULL API GATEWAY

#### API Gateways and Distributed Tracing

### Step 30 - Introduction to API Gateways

**What is API Gateway?**

1. API Gateway is the ***single entry & exit point of all the microservices*** in the application. Since every microservice has its own IP & port, and we can’t provide multiple IP & port details to the client / end user, there must be a single entry and exit point.
2. It is also a type of microservice, that calls all other microservices using Eureka and it should also be registered with the Eureka Server like other microservices.
3. It generates **a class(proxy)** based on the service Id provided with the path(URL) using a load balancer client. Then it selects one Service Instance from Eureka and makes the HTTP call. This is obviously required because Eureka Server itself can’t communicate with any microservice.
4. helps in implementing Security, applying filters, SSO(Single Sing On), dynamic routing etc.

**Zuul Proxy Server**

1. Use to implement API Gateway
2. It handles all the requests and does the dynamic routing of microservices. Dynamic routing is nothing but choosing one microservice instance and make HTTP call based on the load.
3. Called as **Zuul Server or Edge Server**.
4. Add @EnableZuulProxy annotation on our main application class to make our Spring Boot application act as a Zuul Proxy server.

### Step 31 - Setting up Zuul API Gateway

**Step 1**: Create a component

Add @EnableZuulProxy and @EnableDiscoveryClient.

**Step 2**: Decide what it should do when it intercepts a request

**Step 3**: All requests should be configured to pass through it

### Step 32 - Implementing Zuul Logging Filter

Here we will write what to do when it intercepts request

@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;

}

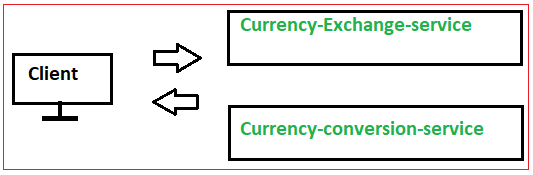
}

### In above example w have 4 methods which have some significance mentioned below

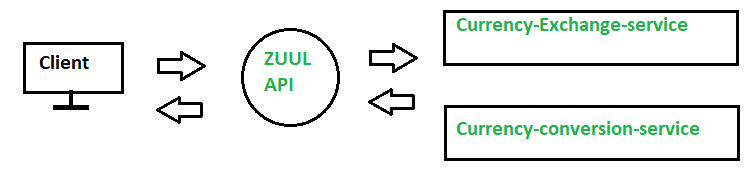
1. shouldFilter() : it tell whether we should filter or not. It is conveyed by returning true or false
2. filterType() : it returns one value out of pre/post/error which means this filter acts before the req/after the req/only when error comes.
3. run(): real logic comes here
4. filterOrder() : it returns a number like 1/0/-1/2/3 which specifies the order of filters.

### Step 33 - Executing a request through Zuul API Gateway

**Before Zuul API**



**After Zuul API**



Here we just need to call through url of zuul to microservie

Zuul url : http://localhost:8765

Microservice url: <http://localhost:8000/currency-exchange/from/USD/to/INR>

Final url : [**http://localhost:8765/{application-name}/{uri}**](http://localhost:8765/%7bapplication-name%7d/%7buri%7d)

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

### Step 34 - Setting up Zuul API Gateway between microservice invocations

We just need to write ZUUL API application,in annotation @FeignClient(name=””) so that we can go through API first then it goes to respective service.

### Step 35 - Introduction to Distributed Tracing

### Step 36 - Implementing Spring Cloud Sleuth

### Step 37 - Introduction to Distributed Tracing with Zipkin

### Step 38 - Installing Rabbit MQ

### Step 39 - Setting up Distributed Tracing with Zipkin

### Step 40 - Connecting microservices to Zipkin

### Step 41 - Using Zipkin UI Dashboard to trace requests