**<https://www.baeldung.com/cs/service-discovery-microservices#:~:text=A%20microservice%20needs%20to%20know,system%20that's%20difficult%20to%20maintain>.**

[**https://www.baeldung.com/eureka-self-preservation-renewal**](https://www.baeldung.com/eureka-self-preservation-renewal)

**A microservices-based application typically runs in virtualized or containerized environments. The number of instances of a service and its locations changes dynamically. We need to know where these instances are and their names to allow requests to arrive at the target microservice. This is where tactics such as Service Discovery come into play.**

**The Service Discovery mechanism helps us know where each instance is located. In this way, a Service Discovery component acts as a registry in which the addresses of all instances are tracked.**

**The Need for Service Discovery :**

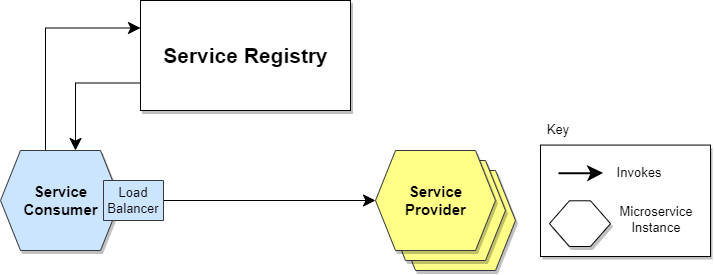
**A microservice needs to know the location (IP address and port) of every service it communicates with. If we don’t employ a Service Discovery mechanism, service locations become coupled, leading to a system that’s difficult to maintain.**

**How it works?**

**Service Discovery handles things in two parts. First, it provides a mechanism for an instance to register and say, “I’m here!” Second, it provides a way to find the service once it has registered**

**Client-Side Service Discovery**

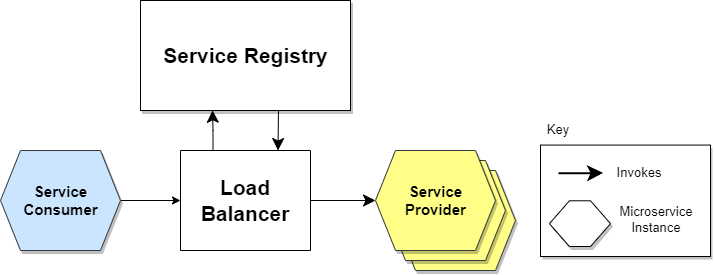
**When using Client-Side Discovery, the Service Consumer is responsible for determining the network locations of available service instances and load balancing requests between them**



**It’s an advantage because it saves an extra hop that we would’ve had with a dedicated load balancer. It’s a disadvantage because the Service Consumer must implement the load balancing logic**

**Server-Side Service Discovery**

**The alternate approach to Service Discovery is the Server-Side Discovery model, which uses an intermediary that acts as a Load Balancer.**

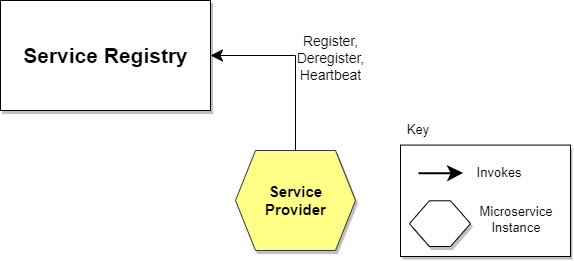


**Creating this level of abstraction makes the Service Consumer lighter, as it doesn’t have to deal with the lookup procedure**

**Service Registration Options:**

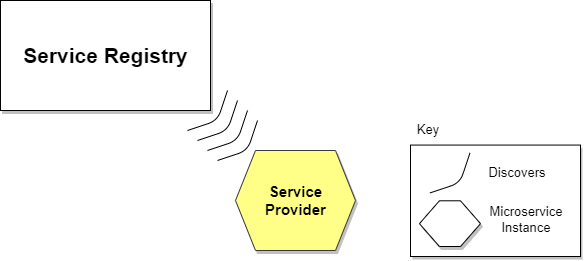
1.Self-Registration

When using the self-registration model, a service instance is responsible for registering and de-registering itself in the Service Registry. In addition, if necessary, a service instance sends heartbeat requests to keep its registration alive. The following diagram shows the structure of this pattern.



2. Third-party Registration:

When using the third-party registration model, the service instances aren’t responsible for registration in the Service Registry. Instead, another system component known as the Service Register is responsible for registration. The Service Register keeps track of changes to running instances by polling the deployment environment or subscribing to events. When it detects a newly available service instance, it records it in its database. The Service Registry also de-registers terminated service instances. The following diagram illustrates this



**@EnableEurekaServer** is added to create Eureka Server

Also we need to add eureka server dependency And add below lines in app.properties file

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**@EnableEurekaClient :** To register with Eureka Server .Add this annotation on client services

Also add eureka client dependency in pom.xml and URL of Eureka Server in application.yaml

eureka:

client:

service-url:

defaultZone: <http://localhost:8761/eureka>

We can create multiple instances of services using below command

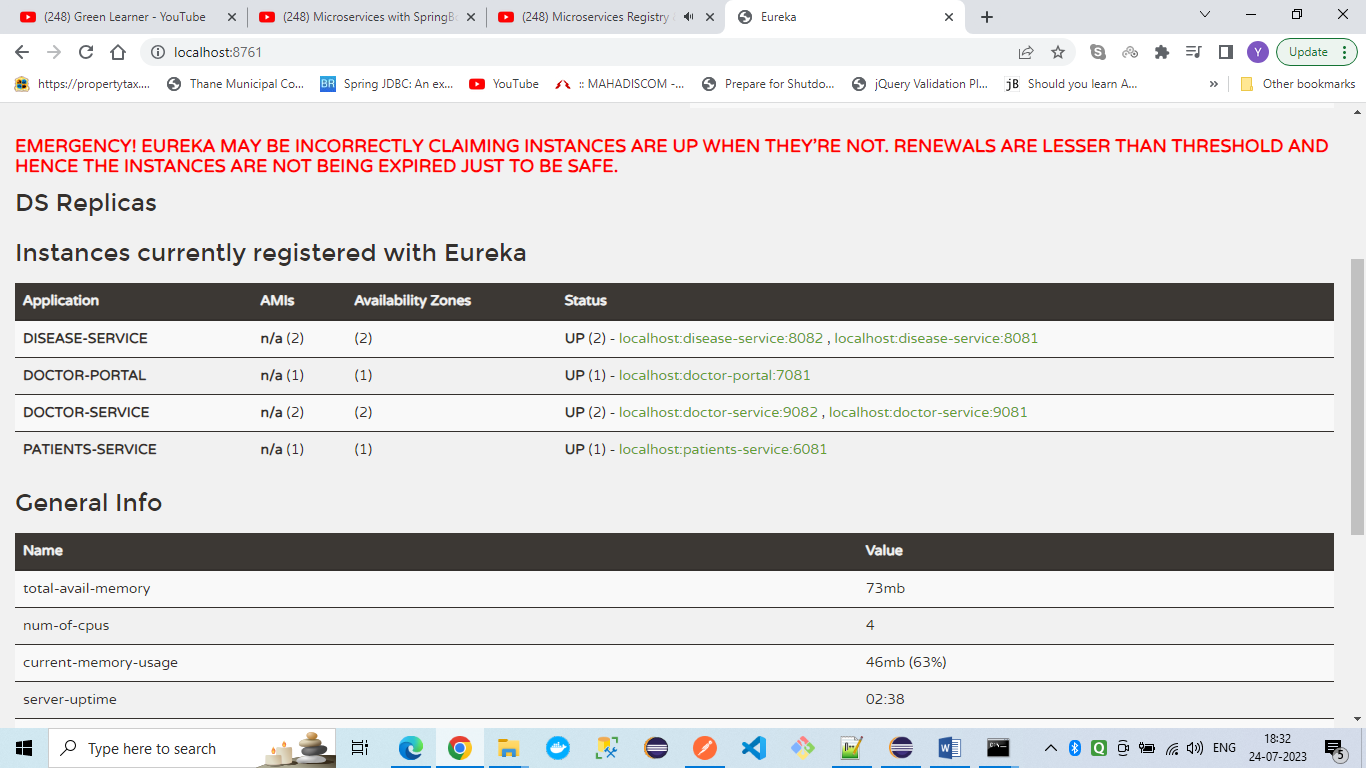
First create jar files using install package maven goal

Then go that location and use below command :

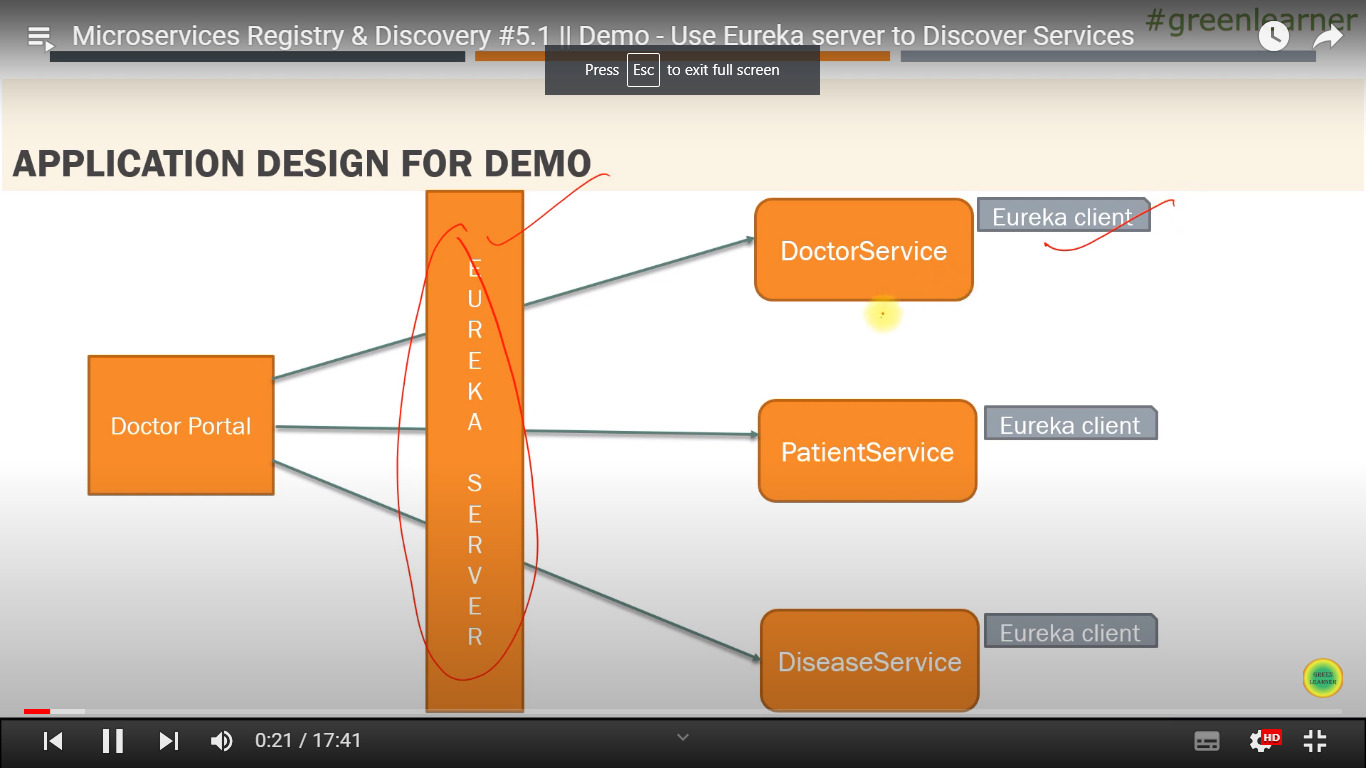
java -jar deseases-service-0.0.1-SNAPSHOT.jar --server.port=8082

F:\Service-Registry-Discovery-Eureka-GreenLearner\doctor-service\target>java -jar doctor-service-0.0.1-SNAPSHOT.jar --server.port=9082

Go to Eureka dashboard & check



Now Next task is Creating Doctor Portal and access the microservices like doctor service , patient service without knowing actual address of those services .



**@EnableDiscoveryClient :**

Used to contact Eureka Server to connect to other services which are registered with it.

@Autowired

EurekaClient eurekaClient;

InstanceInfo nextServerFromEureka = eurekaClient.getNextServerFromEureka("DOCTOR-SERVICE", false);

String baseUrl = nextServerFromEureka.getHomePageUrl();//http://localhost:port

Doctor Portal App is using service ID like “DOCTOR-SERVICE” to call the spring boot services which are registered with Eureka Server.

We can have multiple instances of each micro service so at runtime we get response from any instance.

Eureka server continuously checking the heartbeats with the services registered with it.

So every 30 secs server checks the health of services .If its down then it will check again.

This happens 3 time i.e 90 sec till this period even if some service instance is down ,

In Eureka Server Dashboard we will get to see the instance status as UP.

To avoid it follow below steps :

<https://dzone.com/articles/evicting-instances-from-eureka>

**eureka.server.enableSelfPreservation to false**

Load Balancing using Spring Cloud Load Balancer + Feign Client \_ Eureka

**Refer Currency-conversion-service & currency-exchange –service in microservices\_v2\_in28Minutes**

We have to use feign-client & Eureka .Feign client internally uses SpringCloudLoadBalancer .

We have to add below dependencies :

netflix-eureka-client

spring-cloud-starter-config

spring-cloud-starter-openfeign

User below annotation on SpringBootMain class:

**@EnableFeignClients**

Create one proxy class to call the third party API and annotate it with:

@FeignClient(name="currency-exchange")

Feign client automatically does load balancing .It will search for all instances registered with eureka with “currency-exchange” name. & then distributes the request in round robin manner.

If we want to have our customer Load Balancer then we need to do following things:

Create One class (here CurrencyExchangeLoadBalancer) & add below logic:

**@LoadBalancerClient**(value="currency-exchange" , configuration=**MyCustomLoadBalancer**.class)

public class CurrencyExchangeLoadBalancer {

@LoadBalanced

@Bean

public Feign.Builder feignBuilder(){

return Feign.builder();

}

}

Create one more class (here **MyCustomLoadBalancer)** & add below logic

public class MyCustomLoadBalancer {

@Bean

ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(Environment environment,

LoadBalancerClientFactory loadBalancerClientFactory) {

String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY\_NAME);

return new RandomLoadBalancer(loadBalancerClientFactory

.getLazyProvider(name, ServiceInstanceListSupplier.class),

name);

/\* return new RoundRobinLoadBalancer(loadBalancerClientFactory

.getLazyProvider(name, ServiceInstanceListSupplier.class),

name);\*/

}

}

Here you can write logic for RandomLoadbalancer or RoundRobinLoadBalancer.

Create Multiple instances of Currency Exchange microservice with below command:

F:\spring\_microservices\_in28Minutes\_udemy\spring-microservices-v2-main\03.microservices\currency-exchange-service\target>java -jar currency-exchange-service-0.0.1-SNAPSHOT.jar **--server.port=8002**

And execute currency conversion API which internally will call currency exchange service.