1. Introduction

We have already been through the [Microservice demo](http://www.jcombat.com/spring/introduction-to-microservices" \t "_blank) and it was so pleasing to see the wonderful response it received. Hopefully we are all good with the basics now. This tutorial will guide us all through the actual Microservices implementation, so we are well-directed every time we are asked to create it.

2. Understanding Service Registry

Traditionally, when we consume a REST service, we are usually provided with the network location of the service instance (the REST service URLs that are usually static). However, this tradition has changed with the Microservices architecture coming into picture. Lets understand, how!

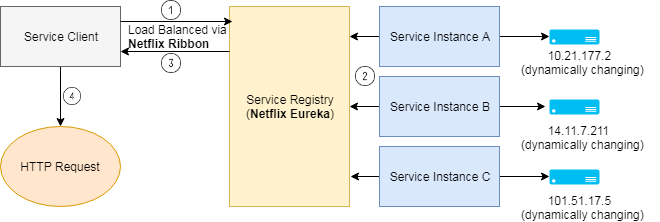
In Microservices architecture, network locations of the service instances are dynamically assigned/changed because of autoscaling, failures and upgrades. To comply with this dynamicity, **service discovery mechanism** comes in.

So there are two main service discovery patterns –

* client‑side discovery
* server‑side discovery

Client‑side discovery

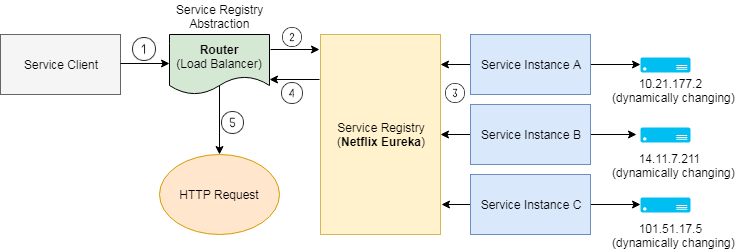
In client‑side service discovery, client queries the Service Registry and determine the available service instances using a separate load balancing mechanism. Load balancing usually works with Service Registry to load balance requests across the available service instances. Each HTTP request is done using one of the load balanced instance that is returned.

[](http://www.jcombat.com/wp-content/uploads/2017/12/CLient-side.png)

One of the major drawback is the tight coupling of client with Service Discovery and the Service discovery logic needs to be written at the client end.

Server‑side discovery

The client queries the Discovery Service via an abstraction layer, which is actually the load balancer or we can say a router, that queries the Service Registry and routes the HTTP request to an available service instance. Details of Service Discovery are abstracted (or hidden) from the client, as a result of which the client don’t need to write the discovery logic, which lead to more ideal loose coupling of client with the Service Discovery.

[](http://www.jcombat.com/wp-content/uploads/2017/12/Service-Side-Discovery.png)

**So what is Service Registry?**

**Service registry** is the key part of service discovery mechanism. Service registry is a database of available service instances. Service registry assures that its highly available and up to date with the network locations of the services instances. Clients can cache network locations obtained from the service registry, but as the network locations keep changing, the cache data soon becomes out of date. To cope with this, it is the responsibility of Service Registry server (consists of a cluster of servers) to maintain the consistency and keep refreshing the network location of the service instances (usually refresh is done every 30 seconds).

One of most common example of a Service Registry is **Netflix Eureka**. It provides a REST API to register (using a POST request) and query (using an HTTP GET request) service instances.

Netflix achieves the high availability by running one or more Eureka servers in each [Amazon EC2 availability zone](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html). Each Eureka in turn runs on an EC2 instance that has elastic IP address. DNS TEXT records are used to store the Eureka cluster configuration (maps to the list of network locations of Eureka servers), so that when Eureka server starts up, DNS is queried to retrieve the Eureka cluster configuration (network locations of Eureka servers) and assigns itself an unused elastic IP address.

**Hashicorp’s Consul** and **Apache Zookeeper** are other examples of Service Registry.

**Netflix Ribbon** is an IPC (Inter Process Communication) client that works with Eureka to load balance requests across the available service instances. The **@LoadBalanced**annotation configures the RestTemplate to use **Ribbon**, which has been configured to use the Eureka client to query service discovery and fetch available service instances.

3. Microservices Implementation

We will check out a simple demo on Microservices using Spring Boot. Hope we get more close to the Microservices concept.

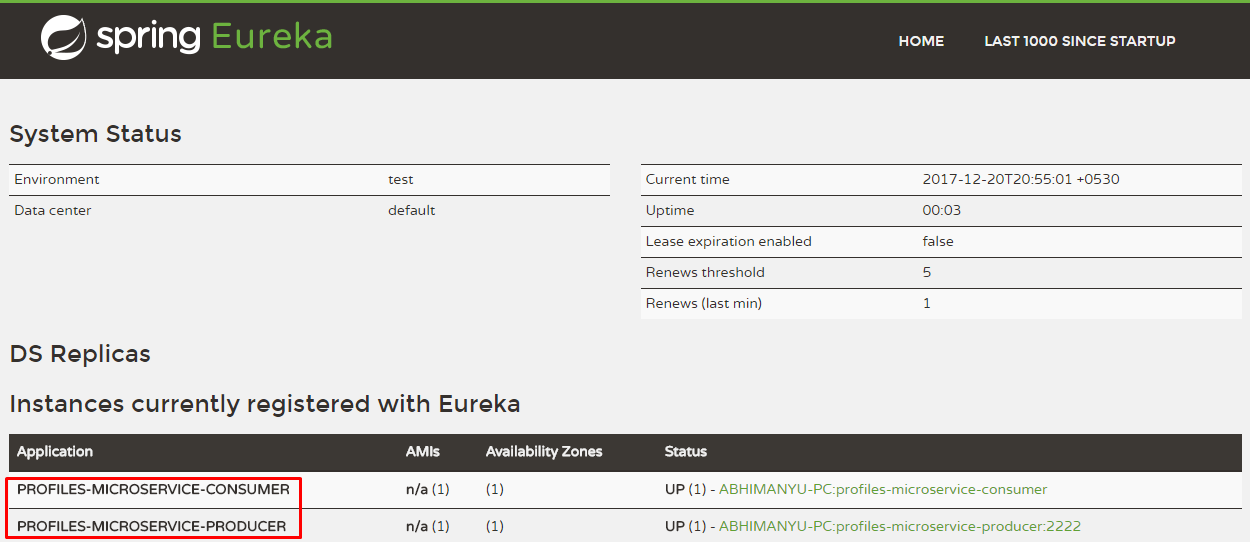
To create a Microservices system, we need to ensure of the below steps –

1. **Creation of Eureka Discovery Server**
   * With Spring Boot, just one annotation **@EnableEurekaServer** does the job.
2. **Creating Producer Microservice**
   * Register itself with Discovery Service
   * **@EnableDiscoveryClient** activates the Netflix Eureka DiscoveryClient implementation
3. **Create Consumer Microservice** which finds the producer service instance registered with Discovery Service
   * Register itself with Discovery Service
   * **@EnableDiscoveryClient** activates the Netflix Eureka DiscoveryClient implementation
   * Requests for DiscoveryClient instance of Producer Microservice using a smart **RestTemplate**.
4. We can then **test end-to-end result** by starting the Eureka service first. Once the Eureka service starts up, start the discovery clients one after the other.

4. Testing the application

Once the Discovery service and Discovery client are started, we can check if the Discovery clients got properly registered themselves with the Eureka Discovery server.

http://localhost:1111/

[](http://www.jcombat.com/wp-content/uploads/2017/12/micros.png)