## What does one mean by Service Registration and Discovery ? How is it implemented in Spring Cloud?

When we start a project, we usually have all the configurations in the properties file.  
As more and more services are developed and deployed, adding and modifying these properties become more complex. Some services might go down, while some the location might change. This manual changing of properties may create issues.  
**Eureka Service Registration and Discovery** helps in such scenarios. As all services are registered to the Eureka server and lookup done by calling the Eureka Server, any change in service locations need not be handled and is taken care of.

**Overview of Netflix components-**

|  |  |
| --- | --- |
| **Netflix Component Name** | **Functionality** |
| Eureka | Service Registration and Discovery |
| Ribbon | Dynamic Routing and client side Load Balancer |
| Hystrix | Circuit Breaker |
| Zuul | Edge Server intelligent routing |

## Why Microservices?

**Monolithic architecture** is like a big container where in all the software components of an application are assembled together and tightly coupled.

All the components in monolithic application is interconnected and interdependent.

Eg: In a e-commerce site there are three services 1.Customer Service 2.Product service and 3.Cart service

**Disadvantages of Monolithic architecture:**

1. large and Complex application : Very difficult to understand this complex application how to correctly implement a change

2. Slow development :

3. Blocks Contigous development : In order to a do small change in one component(like Cart service) we need to build and deploy all the components its irrelevant, time consuming and risky also.it discourages the frequent update

4. Unscalable : Means If we need to increase only one instance like Customerservice it is not possible in case of Monolithic application

5. Unreliable : Due to tightly coupled application once a system goes down the entire application will be down

6.Inflexible : To support new framework It is very difficult and time consuming to adopt a new environment

**MicroService** : In simple words, In Microservices each service is self contained and implements a single business logic so Unlike Monolithic application all the micro services is divided into separate module. Communication between microservices is stateless communication

All the microservice are small, independent and loosely coupled so each microservice is handled by a small team and deployed independently

Internal implementation of each microservices are hidden from other microservice

Apart from these typical microservice there are some new terms used in the world of Microservice

1. **Management** : Management component is responsible for placing services on nodes, identifying failures, rebalancing service across nodes

2. **Service discovery** : Mainly the task of this component is to maintain a list of services and which node they are located on. It enables to find the endpoint for a service.

3. **API GateWay** : It is basically the entry point for a client. First all the client request goes to API gateway then it forward the client to appropriate service on the backend. API gateway might interact with several services and get aggregate response .

**Features of Microservices:**

**Single responsibility principle:** The single responsibility principle is one of the principles defined as part of the [SOLID design pattern](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP). It implies that a unit, either a class, a function, or a microservice, should have one and only one responsibility.

At no point of time, one microservice should have more than one responsibility.

1.**Small focused**:

2.**Loosely Coupled** : Development and Deployment becomes really quick

3.**Language Neutral** : some Microservice might be written in Python for fast deployment and other can be written in Java for Speed purpose. It doesn’t effect the other microservices

4.**Bounded Context** : Each microservice doesnot need to understand the implementation of other microservices

Advantages Of Microservice:

1. **Independent development**

2. **Independent deployment** : Bug fixing and feature releases are less risky and more manageable

3. **Fault Isolation** : If our one service is down others will not affected at all

4**. Mixed Technology Stack** :

5. Scalability for a particular Service is possible unlike the entire services

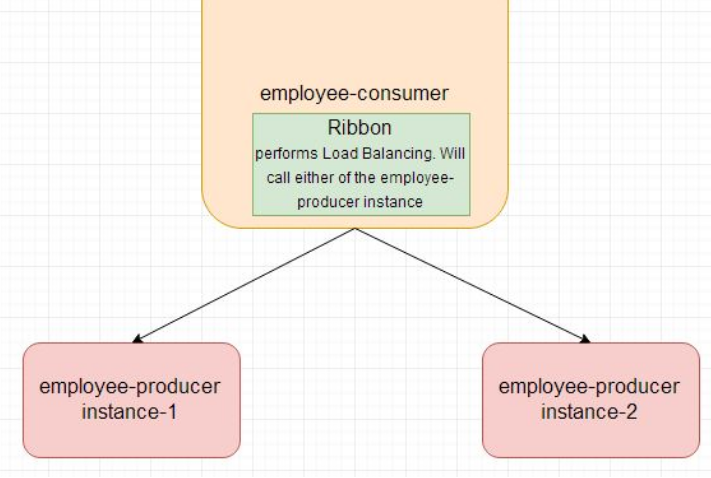
**Spring Cloud- Netflix Eureka + Ribbon Simple :**

In this post we make use of Netflix **Ribbon for Client Side Load Balancing**.

**What is Load Balancing? Need for Netflix Ribbon**

Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource.

**In** [**previous example we developed two services employee-producer and employee-consumer**](https://www.javainuse.com/spring/spring_eurekaregister)**. Suppose other modules are also calling and consuming employee-producer module services. So the load at employee-producer is high. To deal with this we deploy multiple instances of employee-producer. Suppose two in this case. Now we will have to use a Load Balancer to route any incoming requests to either one of these two services.**

****

Code changes for employee producer-

In the application.properties add the instance id as follows-

**eureka.client.serviceUrl.defaultZone**=**http://localhost:8090/eureka  
spring.application.name**=**employee-producer  
eureka.instance.instanceid**=**${spring.application.name}:${random.value}  
server.port**=**8080**

We need to start the employee-producer instance twice. So start the employee-producer instance the first time.  
It will start on the default port 8080.  
Next in application.properties add the port as 8081 and start the employee-producer again.

So now we have two instances of employee producer running, one on port 8080 and the other on 8081

Code changes for employee-consumer module –

Modify the pom.xml to include the **spring cloud ribbon starter dependency**

**<dependency>**

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

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

**</dependency>**

Next modify the ConsumerControllerClient code. Previously we were autowiring an **DiscoveryClient** bean, replace it with **LoadBalancerClient**

**@Autowired**

**private LoadBalancerClient loadBalancer;**

@Controller

public class ConsumerControllerClient {

**@Autowired**

**private LoadBalancerClient loadBalancer;**

**public void getEmployee() throws RestClientException, IOException {**

**ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");**

**System.out.println(serviceInstance.getUri());**

**String baseUrl=serviceInstance.getUri().toString();**

baseUrl=baseUrl+"/employee";

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response=null;

try{

response=restTemplate.exchange(baseUrl,

HttpMethod.GET, getHeaders(),String.class);

}catch (Exception ex)

{

System.out.println(ex);

}

System.out.println(response.getBody());

}

private static HttpEntity<?> getHeaders() throws IOException {

HttpHeaders headers = new HttpHeaders();

headers.set("Accept", MediaType.APPLICATION\_JSON\_VALUE);

return new HttpEntity<>(headers);

}

}

**Hystrix for fault tolerance:**

If we have 9 microservices 1->2->3------🡪9

Suppose if the microservice 9 in the above diagram failed, then using the traditional approach we will propagate an exception. But this will still cause the whole system to crash anyways.  
This problem gets more complex as the number of microservices increase. The number of microservices can be as high as 1000. This is where hystrix comes into picture-  
We will be using two features of Hystrix-

* Fallback method : In fallback method it returns some value instead of exception
* Circuit Breaker

employee-consumer consuming the service exposed by the employee-producer. Now suppose due to some reason the employee-producer exposed service throws an exception. In this case using Hystrix we define a fallback method. This fallback method should have the same return type as the exposed service. In case of exception in the exposed service the fallback method will return some value

Step 1 : Add Hystrix dependency in Employee-producer module

**<dependency>**

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

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

**</dependency>**

Step 2 : Add this annotation with @SpringBootApplication

**@EnableCircuitBreaker**

Step 3:In Controller class define a fallback method

@RestController

public class TestController {

@RequestMapping(value = "/employee", method = RequestMethod.GET)

**@HystrixCommand(fallbackMethod = "getDataFallBack")**

public Employee firstPage() {

Employee emp = new Employee();

emp.setName("emp1");

emp.setDesignation("manager");

emp.setEmpId("1");

emp.setSalary(3000);

if(emp.getName().equalsIgnoreCase("emp1"))

throw new RuntimeException();

return emp;

}

**public Employee getDataFallBack() {**

**Employee emp = new Employee();**

**emp.setName("fallback-emp1");**

**emp.setDesignation("fallback-manager");**

**emp.setEmpId("fallback-1");**

**emp.setSalary(3000);**

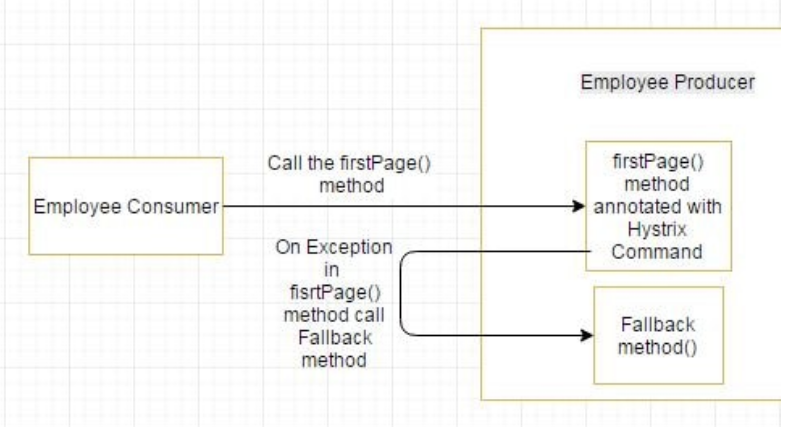
**return emp;**

**}**

}

**Circuit Breaker with Netflix Hystrix:**

If the exceptions keep on occuring in the firstPage method() then the Hystrix circuit will break and the employee consumer will skip the firtsPage method all together and directly call the fallback method.



The purpose of circuit breaker is to give time to the firstPage method or other methods that the firstPage method might be calling and is causing the exception to recover. It might happen that on less load the issue causing the exceptions have better chance of recovering

Previously using employee-consumer we were calling the employee producer only once. Now using for loop we will call it multiple times and check if the circuit trips and the fallback method gets called directly.

In Employee-Consumer main class:

@SpringBootApplication

public class SpringBootHelloWorldApplication {

public static void main(String[] args) throws RestClientException, IOException {

ApplicationContext ctx = SpringApplication.run( SpringBootHelloWorldApplication.class, args);

ConsumerControllerClient consumerControllerClient=ctx.getBean(ConsumerControllerClient.class);

System.out.println(consumerControllerClient);

**for(int i=0;i<100;i++)**

**consumerControllerClient.getEmployee();**

}

@Bean

public ConsumerControllerClient consumerControllerClient()

{

return new ConsumerControllerClient();

}

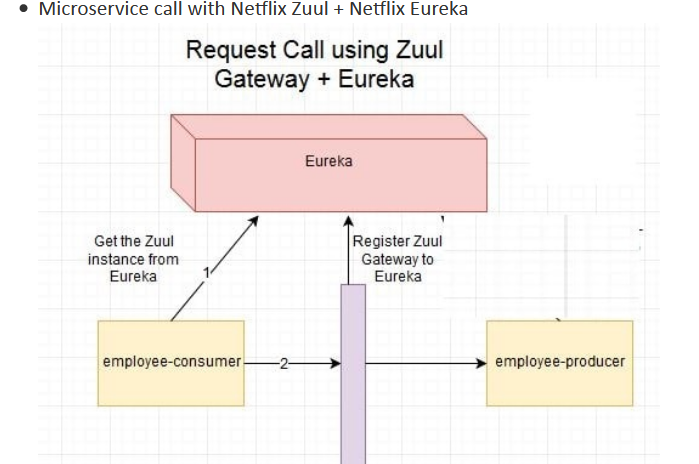
}

We can see that after some exceptions the **fallback method getting called directly and the hystrix annotated method skipped. So the hystrix circuit is open.**

**Dynamic routing with Netflix Zuul :**

Zuul is an api gateway between Employee-consumer and Employee-producer . Before that there was a direct call between consumer and producer. But after coming this API gateway under picture first consumer connected with zuul gateway and Zuul is responsible to redirect Producer service. so no direct call between consumer and producer.

Zuul is also registered with service Discovery Eureka. So Consumer first get the address of Zuul from Eureka server and call Zuul gateway. then Zuul gateway redirects its request to employee producer



Of these modules there will be no change in the [employee-producer and Eureka Server code we had developed in the Netflix Eureka Tutorial](https://www.javainuse.com/spring/spring_eurekaregister3). We will be creating a new module employee-zuul-service and modifying the [employee-consumer module code developed in the Netflix Eureka Tutorial](https://www.javainuse.com/spring/spring_eurekaregister4).

**Step 1: Create a Zuul-proxy microservice**

**In pom.xml give the entry of zuul**

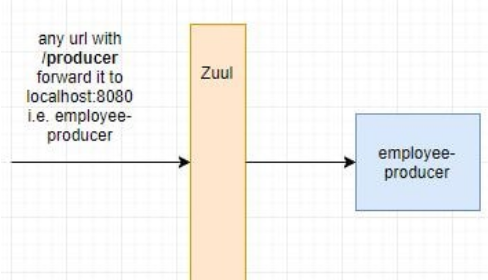
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-zuul</**artifactId**>  
</**dependency**>

**Step 2: in application.properties :**

zuul.routes.producer.url=http://localhost:8080 eureka.client.serviceUrl.defaultZone=http://localhost:8090

server.port=8079

spring.application.name=employee-zuul-service



Here zuul.routes.producer.url will route incoming traffic to request for /producer to the employee-producer microservice. Similar routes can be added for other microservices as well.

Step 3 : Next we define the 4 types of filters supported by Zuul-

* **pre**
* **post**
* **route**
* **error**

Finally we annotate the Spring Boot Main class with **@EnableZuulProxy**.With this the module will act as a service proxy or gateway.

Also we create the beans for the filters defined above.

@SpringBootApplication

@EnableDiscoveryClient

**@EnableZuulProxy**

public class SpringBootHelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootHelloWorldApplication.class, args);

}

@Bean

public PreFilter preFilter() {

return new PreFilter();

}

@Bean

public PostFilter postFilter() {

return new PostFilter();

}

@Bean

public ErrorFilter errorFilter() {

return new ErrorFilter();

}

@Bean

public RouteFilter routeFilter() {

return new RouteFilter();

}

}

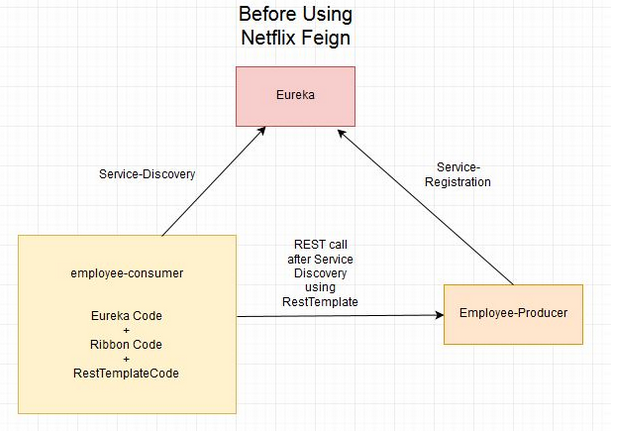
**Code changes for employee-consumer :**

The changes we make for the consumer module are

* We fetch the Zuul Service instance instead of the Employee Producer service we were doing earlier.  
  So in code we have **discoveryClient.getInstances("EMPLOYEE-ZUUL-SERVICE")** instead of **discoveryClient.getInstances("EMPLOYEE-PRODUCER")**
* Append the URL to be hit with /producer since we have defined so in the applicatio.properties above.  
  **baseUrl = baseUrl + "/producer/employee"**

**Spring Cloud- REST call using Netflix Feign Client :**

Feign is a java to http client binder. Feign's first goal was reducing the complexity of binding Denominator uniformly to http apis regardless of restfulness. Previous examples in the employee-consumer we consumed the REST services exposed by the employee-producer using **REST Template.**

****

But we had to write a lot of code to perform following-

* For Load balancing using Ribbon.
* Getting the Service instance and then the Base URL.
* Make use of the REST Template for consuming service.

The previous code was as below :

@Controller

public class ConsumerControllerClient {

**@Autowired**

**private LoadBalancerClient loadBalancer;**

**public void getEmployee() throws RestClientException, IOException {**

**ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");**

**System.out.println(serviceInstance.getUri());**

**String baseUrl=serviceInstance.getUri().toString();**

baseUrl=baseUrl+"/employee";

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response=null;

try{

response=restTemplate.exchange(baseUrl,

HttpMethod.GET, getHeaders(),String.class);

}catch (Exception ex)

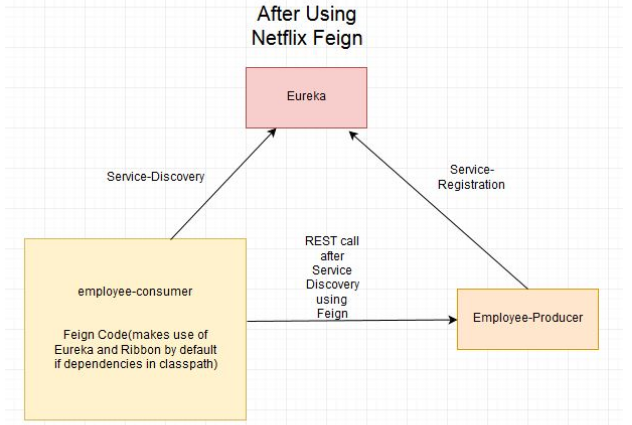
{

System.out.println(ex);

}

System.out.println(response.getBody());

}



The previous code, there are chances of exceptions like NullPointer and is not optimal. We will see how the call is made much easier and cleaner using Netflix Feign. If the Netflix Ribbon dependency is also in the classpath, then Feign also takes care of load balancing by default.

**W**e had implemented Load Balancing using Netflix Ribbon. The [netflix ribbon code](https://www.javainuse.com/spring/spring_ribbon) here will be the starting point. Will only be making changes in the employee-consumer module by adding the Netflix Feign code. The employee-producer and Eureka Server code will remain the same.

We first add the netflix feign dependency in the pom as follows-

<dependency>

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

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

</dependency>

**<dependency>**

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

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

**</dependency>**

We next define a Feign Client by creating an interface with @FeignClient annotation. We also specify the name value as "employee-producer". This value is the name of the service registered using Eureka for discovery. We define the method call to be made to consume the REST service exposed by the employee-producer module.

package com.javainuse.services;

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

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import com.javainuse.controllers.Employee;

**@FeignClient(name="employee-producer")**

public interface RemoteCallService {

@RequestMapping(method=RequestMethod.GET, value="/employee")

public Employee getData();

}

Next we autowire the RemoteCallService in the ConsumerControllerClient class. Then using it make the REST call. Load Balancing is automatically taken care by Feign Client.

package com.javainuse.controllers;

import java.io.IOException;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.client.RestClientException;

import com.javainuse.services.RemoteCallService;

@Controller

public class ConsumerControllerClient {

@Autowired

private RemoteCallService loadBalancer;

public void getEmployee() throws RestClientException, IOException {

try {

**Employee emp = loadBalancer.getData();**

System.out.println(emp.getEmpId());

} catch (Exception ex) {

System.out.println(ex);

}

}

}

Finally we annotate the Spring Boot Main class with **@EnableFeignClients**

package com.javainuse;

import java.io.IOException;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestClientException;

import com.javainuse.controllers.ConsumerControllerClient;

@SpringBootApplication

**@EnableFeignClients**

public class SpringBootHelloWorldApplication {

public static void main(String[] args) throws RestClientException, IOException {

ApplicationContext ctx = SpringApplication.run(SpringBootHelloWorldApplication.class, args);

ConsumerControllerClient consumerControllerClient = ctx.getBean(ConsumerControllerClient.class);

System.out.println(consumerControllerClient);

consumerControllerClient.getEmployee();

}

@Bean

public ConsumerControllerClient consumerControllerClient() {

return new ConsumerControllerClient();

}

}

As we had done in previous posts- Start the following Spring Boot Applications-

* eureka-server
* employee-producer
* employee-consumer

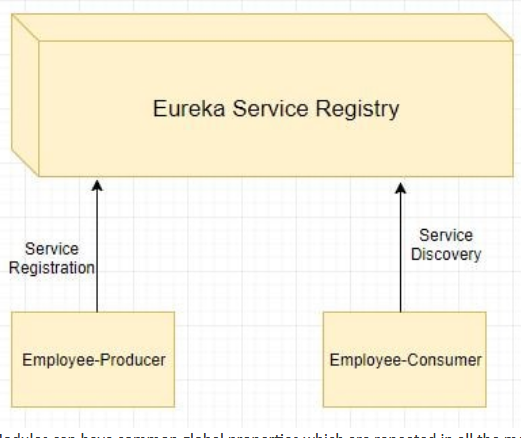
**Spring Cloud Config Simple Example :**

**What is Spring Cloud Config ?Need for it?**

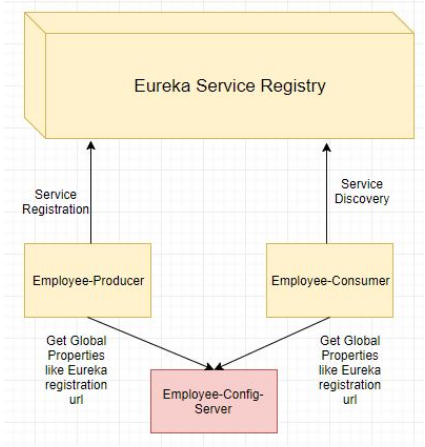
Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Modules can have common global properties which are repeated in all the modules. For example we have properties related to Database, Messaging Queues etc. For example in our employee-consumer and employee-producer we are having the following property for registering to Eureka Server.

eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka



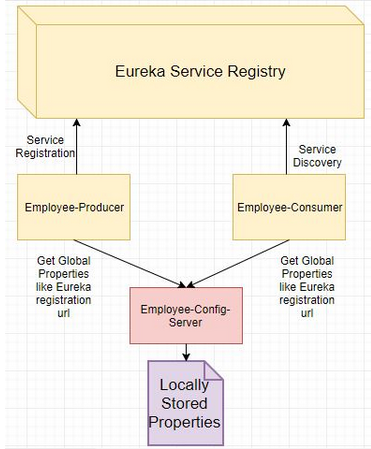
We can **externalize this property using Spring Cloud Config**

****

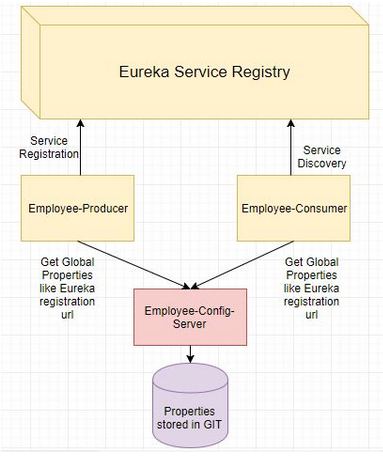
How Spring Cloud Config Works?

Spring Cloud Config Server can be either configured in following 2 ways-

* **Using Local File System -** Properties to be externalized are stored in the local file system of the Spring Cloud Config Server.

****

**Using GIT Repo -** Properties to be externalized are stored in the GIT Repo.

****

**L**ets Begin-

We will be making use of the [employee-producer and the eureka-server code we developed in a previous tutorial](https://www.javainuse.com/spring/spring_eurekaregister3).  
In this tutorial we will be-  
1. We will be creating a new module name employee-config-server having the externalized Eureka registration property.  
2. Next we will the make code changes to the employee-producer module to get the required Eureka registration property from the employee-config-server.

The pom.xml will be as follows with the **spring-cloud-config-server**.

We add two things : **<dependency>**

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

**<artifactId>spring-cloud-config-server</artifactId>**

**</dependency>**

**And <resources>**

**<resource>**

**<filtering>true</filtering>**

**<directory>src/main/resources</directory>**

**<includes>**

**<include>\*.properties</include>**

**<include>common-config/</include>**

**</includes>**

**</resource>**

**</resources>**

Next define the properties in application.properties. The **spring.profiles.active=native** property tells the config module to look for the externalized properties locally.

spring.profiles.active=native

server.port=8888

spring.cloud.config.server.native.search-locations=classpath:/common-config

Next create a folder named **common-config**, and inside it create the application.properties. This is where we will store the global common properties to be used by other microservices.

eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka

Finally we annotate the Spring Boot Main class with **@EnableConfigServer**. With this the module will act as a config server.

package com.javainuse.employeeconfig;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

**@EnableConfigServer**

public class EmployeeConfigApplication {

public static void main(String[] args) {

SpringApplication.run(EmployeeConfigApplication.class, args);

}

}

[**http://localhost:8888/application/default**](http://localhost:8888/application/default) **-->**

{"name":"application","profiles":["default"],"label":null,"version":null,"state":null,"propertySources":[{"name":"classpath:/common-config/application.properties","source":{"eureka.client.serviceUrl.defaultZone":"http://localhost:8090/eureka"}}]}

Code changes for employee-producer

The changes we make for the consumer module are

* Add the Spring Cloud Server Config dependency in the pom.xml  
  The pom.xml will be as follows-

**<dependency>**

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

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

**</dependency>**