

Spring Cloud

Arquitectura de Microservicios

Cuando se habla de arquitectura de microservicios, se habla de **Cloud** o de arquitectura distribuida.

En esta arquitectura, se pueden producir problemas propios de la arquitectura relacionados con

- Monitorización de la arquitectura,
- Configuración de los microservicios,
- Descubrimiento de microservicios,
- Detección de caída de microservicios,
- Balanceo de carga,
- Seguridad centralizada,
- Logs centralizados, ...

Para ello, se suelen emplear herramientas que aplican patrones que permiten solventar/controlar dichos problemas, algunos de ellos son:

- **Configuración distribuida** → Se traduce en un **Servidor de Configuración** que permite centralizar las configuraciones de todos los microservicios que forman el sistema en un único punto, facilitando la gestión y posibilitando cambios en caliente.
- **Registro y autoreconocimiento de servicios** → Permite registrar instancias de servicios y exponerlos de forma integrada, es como unas paginas amarillas de servicios.
- **Enrutado** → Permite definir rutas dentro de la arquitectura para acceder a los microservicios.
- **Balanceo de carga (LoadBalancing)** → Necesidad de que los clientes puedan elegir cual de las instancias de un mismo servicio al que desean conectarse se va a emplear, todo de forma transparente. Esta funcionalidad se basa en obtener las instancias del Servidor de Registro y Descubrimiento.
- **Control de ruptura de comunicación con los servicios (CircuitBreaker)** → Permite controlar que la caída de un microservicio consultado, no provoque la caída de los microservicios que realizan la consulta, proporcionando un resultado estatico para la consulta.
- **Mensajería distribuida** → Permite emplear un bus de mensajería para desacoplar los microservicios entre si, además de poder propagar los cambios en la configuración de los microservicios.
- **Gestión centralizada de logs** → Permite controlar el estado de todas las instancias de todos los microservicios desde una misma aplicación, aunque cada microservicio tiene su propio sistema de logs, además de salvaguardar el tamaño en disco que requiera la aplicación o permitir una trazabilidad de las interacciones entre microservicios, que de otra forma seria compleja.
- **Monitorización** → Permite visualizar de forma centralizada el estado de todas las instancias de todos los microservicios.
- **Servidor de Seguridad centralizado** → Permite gestionar el acceso a los recursos de forma

centralizada, sin que los propios microservicios se encarguen de esa logica. Para ello se emplea OAuth.

[cloud arquitectura] | *cloud_arquitectura.png*

Orquestacion vs Coreografia

Se habla de **Orquestación**, cuando una aplicación, gestiona como invocar a otras aplicaciones, estableciendo los criterios y orden de invocación.

Se habla de **Coreografía**, cuando una aplicación produce un evento, que hace que otras aplicaciones realicen una acción (Bus de mensajería).

Servidor de Configuración

Las aplicaciones que contienen los microservicios se conectarán al servidor de configuración para obtener configuraciones.

El servidor se conecta a un repositorio **git** de donde saca las configuraciones que expone, lo que permite versionar fácilmente dichas configuraciones.

El OSS de Netflix proporciona para esta labor **Archaius**.

Para levantar un servidor de configuración, debemos incluir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Para definir una aplicación como **Servidor de Configuración** basta con realizar dos cosas

- Añadir la anotación **@EnableConfigServer** a la clase **@SpringBootApplication** o **@Configuration**.

```
@EnableConfigServer
@SpringBootApplication
public class ConfigurationApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigurationApplication.class, args);
    }
}
```

- Definir en las propiedades de la aplicación, la conexión con el repositorio **git** que alberga las configuraciones.

```
spring.cloud.config.server.git.uri=https://github.com/victorherrero/azurro/config-  
cloud  
spring.cloud.config.server.git.basedir=config
```

NOTE | La uri puede ser hacia un repositorio local.

En el repositorio **git** deberan existir tantos ficheros de **properties** o **yaml** como aplicaciones configuradas, siendo el nombre de dichos ficheros, el nombre que se le dé a las aplicaciones

Por ejemplo si hay un microservicio que va a obtener su configuracion del servidor de configuración, configurado en el **application.properties** con el nombre

```
spring.application.name=microservicio
```

o **application.yaml**

```
spring:  
  application:  
    name:microservicio
```

Debera existir en el repositorio git un fichero **microservicio.properties** o **microservicio.yaml**.

Las propiedades son expuestas via servicio REST, pudiendose acceder a ellas siguiendo estos patrones

```
/application/{profile}[/{label}]  
/application-{profile}.yaml  
/label/{application}-{profile}.yaml  
/application-{profile}.properties  
/label/{application}-{profile}.properties
```

Donde

- **application** → será el identificador de la aplicacion **spring.application.name**
- **profile** → será uno de los perfiles definidos, sino se ha definido ninguno, siempre estará **default**
- **label** → será la rama en el repo Git, la por defecto **master**

Seguridad

Las funcionalidades del servidor de configuracion estan securizadas, para que cualquier usuario no pueda cambiar los datos de configuracion de la aplicación.

Para configurar la seguridad, hay que añadir la siguiente dependencia

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Cuando se arranca el servidor, se imprimirá un password en la consola

```
Using default security password: 60bc8f1a-477d-484f-aaf8-da7835c207ab
```

Que sirve como password para el usuario **user**. Si se desea otra configuracion se habra de configurar con Srping Security.

Se puede establecer con la propiedad

```
security:
  user:
    password: mipassword
```

Cientes del Servidor de Configuracion

Una vez definido el **Servidor de Configuración**, los microservicios se conectarán a el para obtener las configuraciones, para poder conectar estos microservicios, se debe añadir las dependencias

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Siempre que se añada dependencias de spring cloud, habra que configurar

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Y configurar a través del fichero **bootstrap.properties** donde encontrar el **Servidor de Configuración**. Se configura el fichero **bootstrap.properties**, ya que se necesita que los properties sean cargados antes que el resto de configuraciones de la aplicación.

```
spring.application.name=microservicio

spring.cloud.config.enabled=true
spring.cloud.config.uri=http://localhost:8888
```

El puerto 8888 es el puerto por defecto donde se levanta el servidor de configuración, se puede modificar añadiendo al **application.yml**

```
server:
  port: 8082
```

Dado que el **Servidor de Configuración** estará securizado se deberá indicar las credenciales con la sintaxis

```
spring.cloud.config.uri=http://usuario:password@localhost:8888
```

Si se quiere evitar que se arranque el microservicio si hay algún problema al obtener la configuración, se puede definir

```
spring.cloud.config.fail-fast=true
```

Una vez configurado el acceso del microservicio al **Servidor de Configuración**, habrá que configurar que hacer con las configuraciones recibidas.

```

@RestController
class HolaMundoController {

    @Value("${message:Hello default}")
    private String message;

    @RequestMapping("/")
    public String home() {
        return message;
    }
}

```

En este caso se accede a la propiedad message que se obtendra del servidor de configuración, de no obtenerla su valor será **Hello default**.

Actualizar en caliente las configuraciones

Dado que las configuraciones por defecto son solo cargadas al levantar el contexto, si se desea que los cambios en las configuraciones tengan repercusion inmediata, habrá que realizar configuraciones, en este caso la configuracion necesaria supone añadir la anotacion **@RefreshScope** sobre el componente a refrescar.

```

@RefreshScope
@RestController
class HolaMundoController {

    @Value("${message:Hello default}")
    private String message;

    @RequestMapping("/")
    public String home() {
        return message;
    }
}

```

Una vez preparado el microservicio para aceptar cambios en caliente, basta con hacer el cambio en el repo Git e invocar el servicio de refresco del microservicio del cual ha cambiado su configuracion

```
(POST) http://<usuario>:<password>@localhost:8080/refresh
```

Este servicio de refresco es seguro por lo que habrá que configurar la seguridad en el microservicio

Servidor de Registro y Descubrimiento

Permite gestionar todas las instancias disponibles de los microservicios.

Los microservicios enviarán su estado al servidor Eureka a través de mensajes **heartbeat**, cuando estos mensajes no sean correctos, el servidor desregistrará la instancia del microservicio.

Los clientes del servidor de registro, buscarán en el las instancias disponibles del microservicio que necesiten.

Es habitual que los propios microservicios, a parte de registrarse en el servidor, sean a su vez clientes para consumir otros micoservicios.

Se incluyen varias implementaciones en Spring Cloud para serrvidor de registro/descubrimiento, Eureka Server, Zookeeper, Consul ...

Para configurarlo hay que incluir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

Se precisa configurar algunos aspectos del servicio, para ello en el fichero **application.yml** o **application.properties**

```
server:
  port: 8084 #El 8761 es el puerto para servidor Eureka por defecto

eureka:
  instance:
    hostname: localhost
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Para arrancar el servicio Eureka, unicamente es necesario lanzar la siguiente configuración.

```
@SpringBootApplication
@EnableEurekaServer
public class RegistrationServer {

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

Registrar Microservicio

Lo primero para poder registrar un microservicio en el servidor de descubrimiento es añadir la dependencia de maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Para registrar el microservicio habrá que añadir la anotación **@EnableDiscoveryClient**

```
@EnableAutoConfiguration
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingServer {

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

Y se ha de configurar el nombre de la aplicación con el que se registrará en el servidor de registro Eureka.

```
spring:
  application:
    name: holamundo

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/ # Ha de coincidir con lo definido en
el Eureka Server
```

El tiempo de refresco de las instancias disponibles para los clientes es de por defecto 30 sg, si se desea cambiar, se ha de configurar la siguiente propiedad


```
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
```

NOTE

Puede ser interesante lanzar varias instancias del mismo microservicio, para que se registren en el servidor de Descubrimiento, para ello se pueden cambiar las propiedades desde el script de arranque

```
mvn spring-boot:run -Dserver.port=8081
```

Localizacion de Microservicio registrado en Eureka con Ribbon

El cliente empleará el API de **RestTemplate** al que se proxeara con el balanceador de carga **Ribbon** para poder emplear el servicio de localización de **Eureka** para consumir el servicio.

Se ha de definir un nuevo Bean en el contexto de Spring de tipo **RestTemplate**, al que se ha de anotar con **@LoadBalanced**

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Una vez proxeadado, las peticiones empleando este **RestTemplate**, no se harán sobre el **EndPoint** del servicio, sino sobre el nombre del servicio con el que se registro en **Eureka**.

```
@Autowired
private RestTemplate restTemplate;

public MessageWrapper<Customer> getCustomer(int id) {
    Customer customer = restTemplate.exchange( "http://customer-
service/customer/{id}", HttpMethod.GET, null, new
ParameterizedTypeReference<Customer>() { }, id).getBody();
    return new MessageWrapper<>(customer, "server called using eureka with rest
template");
}
```

Si el servicio es seguro, se pueden emplear las herramientas de **RestTemplate** para realizar la autenticación.

```
restTemplate.getInterceptors().add(new BasicAuthorizationInterceptor("user",
"mipassword"));

ResponseEntity<String> respuesta = restTemplate.exchange("http://holamundo",
HttpMethod.GET, null, String.class, new Object[]{});
```

Para que **Ribbon** sea capaz de enlazar la URL que hace referencia al identificador del servicio en **Eureka** con el servicio real, se debe configurar donde encontrar el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Y configurar la aplicación para que pueda consumir el servicio de **Eureka**

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

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

Uso de Ribbon sin Eureka

Se puede emplear el balanceador de carga Ribbon, definiendo un pool de servidores donde encontrar el servicio a consultar, no siendo necesario el uso de Eureka.

```
customer-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:8290,localhost:8490
```

Simplificación de Clientes de Microservicios con Feign

Feign abstrae el uso del API de RestTemplate para consultar los microservicios, encapsulándolo todo con la definición de una interface.

Para activar su uso, lo primero será añadir la dependencia con Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

El siguiente paso sera activar el autodescubimiento de las configuraciones de **Feign**, como la anotacion **@FeignClient**, para lo que se ha de incluir la anotacion en la configuracion de la aplicaci3n **@EnableFeignClients**

```
@SpringBootApplication
@EnableFeignClients
public class Application {

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

Luego se definen las interaces con la anotacion **@FeignClient**

```
@FeignClient(name="holamundo")
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}
```

Solo resta asociar el nombre que se ha dado al cliente **Feign** con un sevicio real, para ello en el fichero **application.yml** y gracias a **Ribbon**, se pueden definir el pool de servidores que tienen el servicio a consumir.

```
holamundo:
  ribbon:
    listOfServers: http://localhost:8080
```

Acceso a un servicio seguro

Si al servicio al que hay que acceder es seguro, se pueden realizar configuraciones extras como el usuario y password, haciendo referencia a los **Beans** definidos en una clase de configuracion particular

```
@FeignClient(name="holamundo", configuration = Configuracion.class)
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}

@Configuration
public class Configuracion {
    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "mipassword");
    }
}
```

Uso de Eureka

En vez de definir un pool de servidores en el cliente, se puede acceder al servidor **Eureka** facilmente, basta con tener la precaución de emplear en el **name** del Cliente **Feign**, el identificador en **Eureka** del servicio que se ha de consumir.

Añadir la anotacion **@EnableDiscoveryClient** para poder buscar en **Eureka**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {

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

Y configurar la direccion de **Eureka**, no siendo necesario configurar el pool de **Ribbon**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Servidor de Enrutado

Permite definir paths y asociarlos a los microservicios de la arquitectura, será por tanto el componente expuesto de toda la arquitectura.

Spring Cloud proporciona **Zuul** como Servidor de enrutado, que se acopla perfectamente con

Eureka, permitiendo definir rutas que se enlacen directamente con los microservicios publicados en **Eureka** por su nombre.

Se necesita añadir la dependencia Maven.

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

Lo siguiente es activar el Servidor **Zuul**, para lo cual habrá que añadir la anotación **@EnableZuulProxy** a una aplicación Spring Boot.

```
@SpringBootApplication
@EnableZuulProxy
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Solo restarán definir las rutas en el fichero **application.yml**

Estas pueden ser hacia el servicio directamente por su url

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      url: http://localhost:8080/
```

Con lo que se consigue que las rutas hacia **zuul** con path **/holamundo/** se redireccionen hacia el servidor <http://localhost:8080/>

NOTE	Se ha de crear una clave nueva para cada enrutado, dado que la propiedad routes es un mapa, en este caso la clave es holamundo .
-------------	--

O hacia el servidor de descubrimiento **Eureka** por el identificador del servicio en **Eureka**

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      #Para mapeo de servicios registrados en Eureka
      serviceId: holamundo
```

Para esto último, habra que añadir la dependencia de Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Activar el descubrimiento en el proyecto añadiendo la anotación **@EnableDiscoveryClient**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class Application {

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

E indicar en las propiedades del proyecto, donde se encuentra el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Seguridad

En el caso de enrutar hacia servicios seguros, se puede configurar **Zuul** para que siendo el que reciba los token de seguridad, los propague a los servicios a los que enruta, esta configuración por defecto viene desactivada dado que los servicios a los que redirecciona o tienen porque ser de la misma arquitectua y en ese caso, no sería seguro.

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      #Para mapeo de las url directas a un servicio
      url: http://localhost:8080/

      #No se incluye ninguna cabecera como sensible, ya que todas las definidas
      como sensibles, no se propagan
      sensitive-headers:
        custom-sensitive-headers: true
      #Se evita añadir las cabeceras de seguridad a la lista de sensibles.
      ignore-security-headers: false
```

Circuit Breaker

La idea de este componente es la de evitar fallos en cascada, es decir que falle un componente, no por error propio del componente, sino porque falle otro componente de la arquitectura al que se invoca.

Para ello Spring Cloud integra **Hystrix**.

Para emplearlo, se ha de añadir la dependencia Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

La idea de este framework, es proxear la llamada del cliente del servicio sensible a caerse proporcionando una vía de ejecución alternativa **fallback**, para así evitar el error en la invocación.

Para ello se ha de anotar el método que haga la petición al cliente con **@HystrixCommand** indicando el método de **fallback**

```
@RestController
class HolaMundoClienteController {

    @Autowired
    private HolaMundoCliente holaMundoCliente;

    @HystrixCommand(fallbackMethod="fallbackHome")
    @RequestMapping("/")
    public String home() {
        return holaMundoCliente.holaMundo() + " con Feign";
    }

    public String fallbackHome() {
        return "Hubo un problema con un servicio";
    }
}
```

El método de **Fallback** deberá retornar el mismo tipo de dato que el método proxead.

NOTE

No deberán aplicarse las anotaciones sobre los controladores, dado que los proxys entran en conflicto

Para activar estas anotaciones se ha de añadir **@EnableCircuitBreaker**.

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

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

Monitorizacion: Hystrix Dashboard

Se ha de crear un nuevo servicio con la dependencia de Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

Y activar el servicio de monitorizacion con la anotacion **@EnableHystrixDashboard**

```
@SpringBootApplication
@EnableHystrixDashboard
public class Application {

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

Se accederá al panel de monitorizacion en la ruta <http://<host>:<port>/hystrix> y allí se indicará la url del servicio a monitorizar <http://<host>:<port>/hystrix.stream>

Para que la aplicación configurada con **Hystrix** proporcione información a través del servicio **hystrix.stream**, se ha de añadir a dicha aplicación **Actuator**, con Maven.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Monitorizacion: Turbine

Se puede añadir un servicio de monitorización de varios servicios a la vez, llamado **Turbine**, para ello se ha de añadir la dependencia


```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
```

Configuración Distribuida en Bus de Mensajería

Se trata de emplear un bus de mensajería para trasladar el evento de refresco a todos los nodos de los microservicios que emplean una configuración distribuida.

Se precisa por tanto de un bus de mensajería, en este caso Spring Cloud apuesta por implementaciones **AMQP** frente a otras alternativas como podrían ser **JMS**. Y más concretamente **RabbitMQ**.

Para instalar RabbitMQ, se necesita instalar [Erlang](#) a parte de [RabbitMQ](#)

La configuración por defecto de **RabbitMQ** es escuchar por el puerto 5672

Servidor

Se necesitará incluir las siguientes dependencias en el servidor de configuración

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Y la siguiente configuración de la ubicación de RabbitMQ

application.yml

```
server:
  port: 8180

spring:
  cloud:
    bus:
      enabled: true #Habilitamos la publicacion en el bus

  #Indicamos donde esta el repositorio con las configuraciones
  config:
    server:
      git:
        uri: https://github.com/victorherrerocazurro/RepositorioConfiguraciones

#Se necesita conocer donde esta rabbitMQ para enviar los eventos de cambio de
#propiedades
rabbitmq:
  host: localhost
  port: 5672
```

A partir de este punto el servidor aceptará el refresco de las propiedades a traves del bus, empleando el servicio **/monitor**, al cual podrán acceder los repositorio Git a traves de **Hook**

```
curl -v -X POST "http://localhost:8100/monitor" -H "Content-Type: application/json" -H
"X-Event-Key: repo:push" -H "X-Hook-UUID: webhook-uuid" -d '{"push": {"changes": []}}
}'
```

Cliente

Se necesitará incluir la siguiente dependencia en los clientes que servidor de configuración

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

Como bus por defecto se emplea **RabbitMQ**, al que habrá que configurar las siguientes propiedades

application.properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```