Zoe Jimenez

DH  CTD

BACKEND II

MICROSERVICIOS

**Eureka**

Registrar automáticamente los microservicios y sus instancias, tan pronto como se encuentren ejecutando y eliminarlas del registro en cuanto dejan de ejecutarse o dejan de responder.

El cliente debe ser capaz de enviar una solicitud a los microservicios sin conocer su ubicación.

Las solicitudes a las instancias de un microservicio deben poder ser equilibradas mediante un balanceador de carga.

Se puede dividir en dos componentes.

|  |  |
| --- | --- |
| **Client** | **Server** |
| Encargado de publicar la información del microservicio en el que se encuentra. | Encargado de recopilar la información de todos los clientes. |

**Agregar a un proyecto:**

* Eureka Client:

Agregar dependencia:

* spring-cloud-starter-netflix-eureka-client

En Maven:

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Con agregar la dependencia se registra la app en Eureka server, hay que indicar donde se esta ejecutando (default: puerto 8761).

Dentro de application.properties:

eureka.client.service-url.defaultZone= <http://localhost:8761/eureka/>

Mas propiedades a configurar:

Los clientes para comunicarse forman la url con el nombre de la aplicación:

spring.application.name=mi-servicio

Para desactivar Eureka client en nuestro proyecto podemos agregar:

eureka.client.enabled=false

Eureka Server:

Crear un proyecto nuevo con Spring Boot, y agregar las dependencias:

* spring-cloud-starter-netflix-eureka-server
* spring-boot-starter-web
* spring-boot-starter-actuator

Con Maven:

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

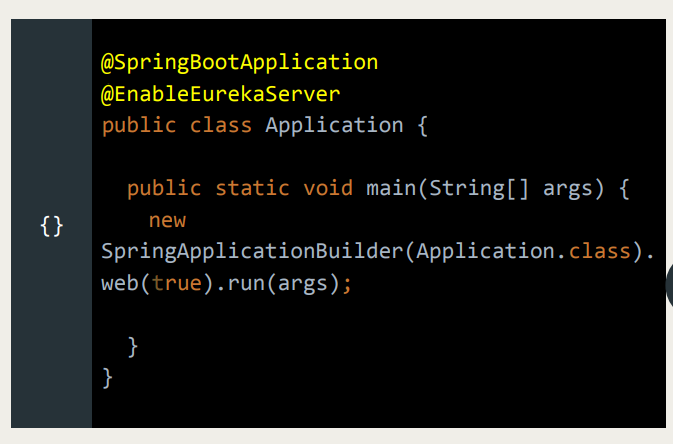
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Agregar anotación @EnableEurekaServer en la clase principal del proyecto.

Ej: 

En application.properties, Eureka server puede funcionar como cliente y servidor a la vez. Para evitar que se registre como cliente poner:

eureka.client.register-with-eureka=false eureka.client.fetch-registry=false

Configuramos el puerto:

server.port=8761

**Spring Boot Actuator (heartbeat)**

En un ecosistema de microservicios es necesario llevar un control por errores, para balanceo de carga o para analizar métricas (ver si se necesitan mas instancias de un servidor en particular, etc.).

Los errores mas comunes en las instancias son por excepción de aplicación, timeout (la respuesta llega después de un determinado umbral), y errores de instancia (la instancia se queda sin espacio en disco, sin memoria, etc.)

Spring Actuator genera los endpoints necesarios para el monitoreo, y las respuestas pueden ser sondeadas por balanceadores de carga, Eureka u otras aplicaciones de administración de tráfico.

|  |  |
| --- | --- |
| Endpoints útiles | |
| /health | Información sobre la salud de la aplicación. Un “estado” si estamos sin autenticación, o información más detallada al autenticarse. |
| /info | Información arbitraria. |
| /metrics | Información de métricas. |
| /trace | Información de seguimiento (default: ultimas peticiones HTTP). |
| /serviceregistry | Estado del registro en Eureka Server. |

Por defecto solo esta habilitado /health, para habilitar el resto hay que configurar application.properties:

**management.endpoints.web.exposure.include=serviceregistry,health,info**

Para habilitar todos:

**management.endpoints.web.exposure.include=\***

Eureka Server utiliza /health e /info.

Por defecto, petición GET a /health responde:

{

“status”:”UP”

}

Petición GET a /info debe ser configurada, ejemplo:

info.app.name=mi-servicio  
info.app.description=Servicio probando Eureka  
info.app.version=1.0.0

Respuesta:

{  
"app" : {  
"version" : "1.0.0",  
"description" : "Servicio probando Eureka",  
"name" : "mi-servicio"  
}  
}

Según la versión de Actuator, el endpoint /info no lee la configuración en application.properties. Para habilitarlo:

**management.info.env.enabled = true**

Agregarlo a nuestro proyecto con Maven:

<dependency>

<groupId>org.springframework.boot</groupid>

<artifactId>spring-boot-starter-actuator</artifactid>

</dependency>

**Central configuration**

**(Configuración en sistemas distribuidos)**

Una organización que crece comienza a desarrollar microservicios que dan soporte a los distintos procesos de negocio. Estos microservicios que proveen la infraestructura, el procesamiento y el soporte de datos necesitan distintos tipos de configuraciones para funciona, por ejemplo;

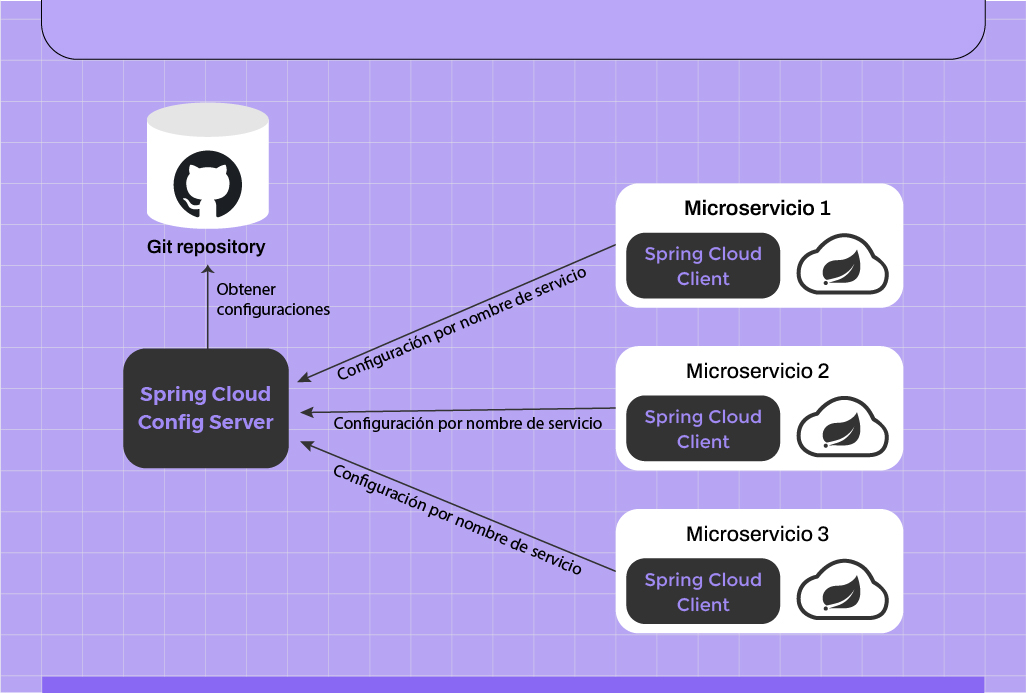
* **Localización** de otros microservicios.
* **Paths** de base de datos y endpoints de los servicios que se desplegarán.
* **Dirección IP** de servidores de almacenamiento, tales como FTP o S3.
* **Tipo de logging** de información por Log4j (INFO, ERROR, DEBUG).
* **Contraseñas** y/o secretos (en la práctica se aplica seguridad adicional en este caso).
* **Variables** de entorno.

¿Por qué configuración particular, y no constantes dentro de la app?

|  |  |
| --- | --- |
| **Desacoplar la información de configuración del código** | Un desarrollador no tiene porqué conocer la URL de la BBDD, ni las contraseñas, ni configuraciones generales manejadas por infraestructura o DevOps. Si Infraestructura debe cambiar una URL interna, es tedioso pedirle a cada equipo de desarrollo que redeploye, ya que los sistemas deben ser capaces de reacción a tales cambios de configuración. |
| **Separación de ambientes (producción, QA, desarrollo)** | Ambientes: producción (código que ve el usuario final), QA (lo mismo, pero para el equipo QA), dev (para el equipo de desarrollo). Cada ambiente va a manejar su configuración, se puede hacer con archivos, pero para configuraciones especificas (múltiples instancias) y determinado numero de microservicios es necesario centralizar las configuraciones. |
| **Evitar deploys por cambios de configuración** | La diferencia entre tener la configuración embebida en el código y tenerla separada, es que separada no necesita deploys (6 minutos), sino restarts (2 minutos). |

**Spring Cloud Config**

La solución que provee Spring Cloud es **Spring Cloud Config**. Tiene dos componentes básicos: **Spring Cloud Config Server** y **Spring Cloud Client**.



**Configuración de valores en Git**

Las configuraciones de los microservicios estarán especificadas en archivos YML dentro de un repo GitHub. Hay un archivo por microservicio que necesite ser configurado de forma externa. El uso de git permite:

* Control de versiones.
* Establecer las reglas de aprobación y rechazo de forma automática.
* Facilidad para analizar, aplicar y/o revertir cambios.
* Trazabilidad de los cambios.

Ej:

* **application.yml**: configuraciones generales para todos los servicios.
* **dh-cuenta-service.yml**: configuraciones para el microservicio llamado así.
* **dh-transaccion-service.yml**: configuraciones para el microservicio llamado así.

Dentro de un archivo:

* + **server.port:** puerto de escucha del microservicio específico.
  + **message:** propiedad personalizada con un valor.

**Configuración servidor de configs**

Servidor Cloud Config que tome la base de Git y lo deje disponible por servicios REST para los microservicios que lo necesiten.

**Pasos:**

* Agregar la dependencia de Spring Cloud Config

<dependency>

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

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

</dependency>

* Configurar el microservicio como servidor de configuraciones

En clase de startup anotación:

@EnableConfigServer

* Configurar el servidor para tomar los valores de Git

En nuestro YML de aplicación debemos especificar el Git de origen de configuración para el servidor de configuraciones dentro de la clave: spring.cloud.config.server.git.url.

server:

port: ${PORT:8888}

spring:

application:

name: config-server

cloud:

config:

server:

git:

url: https://github.com/duccisg/spring-cloud-example

Validación

Realizar la consulta de las configuraciones de los servicios declarados vía HTTP. Patrón:

**http://{url-servidor-config}/{nombre-servicio}/{profile-name}**

|  |  |
| --- | --- |
| {url-servidor-config} | Dirección del servidor que proporcionará las configuraciones, bajo Spring Cloud Config |
| {nombre-servicio} | Nombre del microservicio del cual se consulta la configuración. |
| {profile-name} | Establece el nombre del perfil de configuración a consultar. Este se utilizará junto al nombre del servicio para determinar las configuraciones a utilizar. |

**Configuración de clientes del servidor de configuraciones**

**Pasos:**

* Agregar la dependencia de Spring Cloud Config

<dependency>

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

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

</dependency>

* Especificar la ruta del servidor de configuraciones a utilizar

Crear el archivo de configs Bootstrap.yml indicando nombre del microservicio, que se usara para obtener la config del servidor según los nombres definidos en Git. Dentro del yml establecer la URL del servidor de configuración:

spring:

application:

name: dh-cuenta-service

cloud:

config:

url: ${CONFIG\_SERVER:http://localhost:8888}

* Utilizar las configuraciones obtenidas

Debemos agregar la dependencia de spring-web, agregando un endpoint REST donde mostrar las configuraciones obtenidas del servidor de configs.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-config-starter-web</artifactId>

</dependency>

Creamos una clase con un endpoint inyectando los valores de configuración previamente vistos en la configuración de Git:

**@RestController**

class AccountService {

**@Value**(“${message}”)

private String message;

**@Value**(“${global-message}”)

private String globalMessage;

**@RequestMapping**(method = RequestMethod.GET, path=”service”)

public Map<String,String> message() {

Map<String,String> response = new HashMap<>();

response.put(“message”, message);

response.put(“global-message”, globalMessage);

return response;

}

}

Por ultimo, accedemos al servicio de “cuentas” de acuerdo al puerto de uso definido en Git e ingresamos en el navegador la URL:

**http://localhost:8889/service**

{

**“global-message”**: ”Configuration server is alive!”,

**“message”**: ”Configuración custom con Spring Cloud Config!”

}

**Spring Cloud OpenFeign**

Invocaciones declarativas

Sirven para facilitar la integración entre microservicios mediante la creación de clientes HTTP de forma declarativa. Esto significa que se simplifica la invocación entre estos microservicios. El desarrollador simplemente necesita crear una interfaz y configurarla mediante anotaciones. No se necesita programar toda la lógica de conexión e invocación de una API, sino que simplemente se declara escribiendo anotaciones en los métodos que necesitamos invocar y/o exponer.

**Spring Cloud OpenFeign** sirve para hacer invocaciones declarativas. Algunas ventajas:

* Autodescubrimiento: utilizando Eureka, podemos hacer llamadas a los microservicios utilizando el nombre en lugar de una URL formada por IP y puerto.
* Balanceo de carga: podemos integrar Feign con un balanceador de carga como Spring Cloud LoadBalancer.

<dependency>

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

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

</dependency>

@SpringBootApplication

**@EnableFeignClients**

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Pasos para enviar peticiones a otro microservicio

* Creamos DTO para guardar la respuesta

public class ProductDTO {

private Integer id;

private String title;

private String color;

}

* Creamos la interfaz donde configurar Feign

**@FeignClient**(name = "product-service")

public interface ProductClient {

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

List<ProductDTO> getProducts();

}

@FeignClient(name = “producto-service”)

El valor pasado es el nombre del cliente que estamos creando, y es utilizado también para buscar en el registro de Eureka qué dirección/es están asociadas a ese nombre.

*Envío de peticiones a servicios externos*

@FeignClient(name = "ip-service", url =”http://ipwhois.app/json”)

Test

Para ejecutar los métodos de la interfaz debemos inyectar la misma en un servicio utilizando Spring Boot. Ejemplo:

@Service

public class ProductService{

@Autowired

private ProductClient productFeingClient;

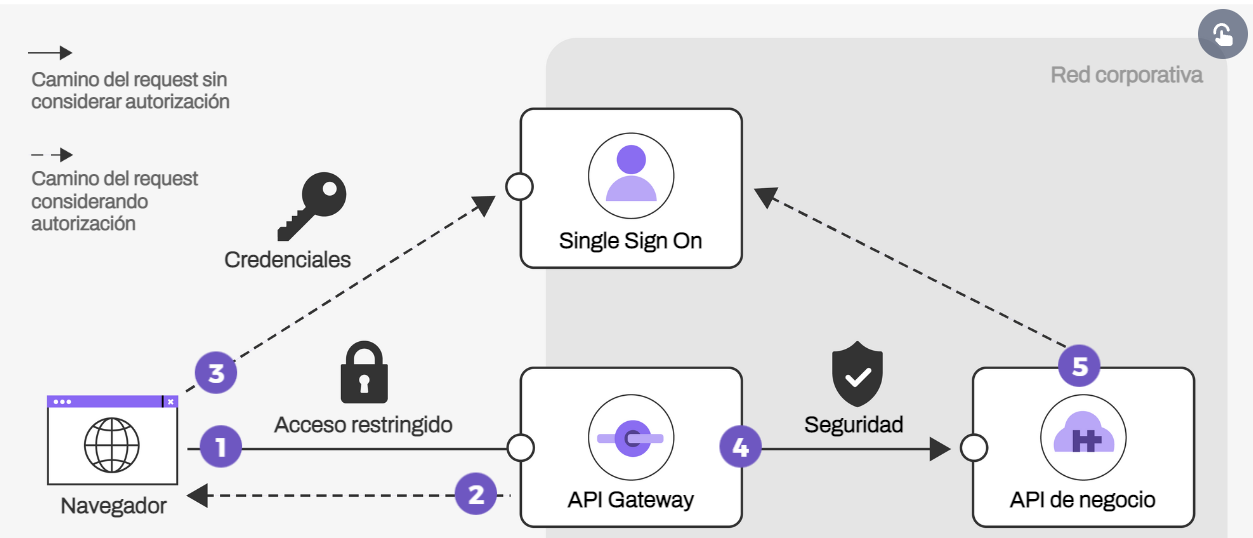
public List<ProductDTO> fetchAllProducts() {

return productFeingClient.getProducts();

}

}

**Spring security - OAuth2**



Roles:

* **Resource owner:** es la persona/entidad que da acceso a sus datos. Literalmente, el propietario del recurso.
* **Client:** aplicación que interactúa con el propietario del recurso. En el caso de una app web, sería la aplicación del navegador.
* **Resource server:** la API que está expuesta en Internet necesita protección de datos. Para obtener acceso a su contenido, se necesita un token emitido por el authorization server.
* **Authorization server:** es responsable de autenticar al usuario y emitir tokens de acceso. Es quien tiene la información del resource owner (usuario). Autentica e interactúa con el usuario después de la identificación del cliente.
* **Autenticación:** es el proceso que determina que un usuario es quien dice ser.
* **Autorización:** es el proceso que determina que un determinado usuario puede ejecutar determinada acción.