Spring Cloud

Victor Herrero Cazurro

Contenidos

1. Spring Cloud

1.1. ¿Que son los Microservicios?

Segun Martin Fowler y James Lewis, los microservicios son pequeños servicios autonomos que se comunican con APIs ligeros, tipicamente con APIs REST.

El concepto de autonomo, indica que el microservicio encierra toda la lógica necesaria para cubrir una funcionalidad completa, desde el API que expone que puede hacer hasta el acceso a la base de datos.

Las aplicaciones modernas presentan habitualmente la necesidad de ser accedidas por diversos tipos de clientes, browser, moviles, aplicaciones nativas, para ello deben implementar un API para ser consumidas, además de para ser integradas con otras aplicaciones a traves de servicios web o brokers de mensajeria.

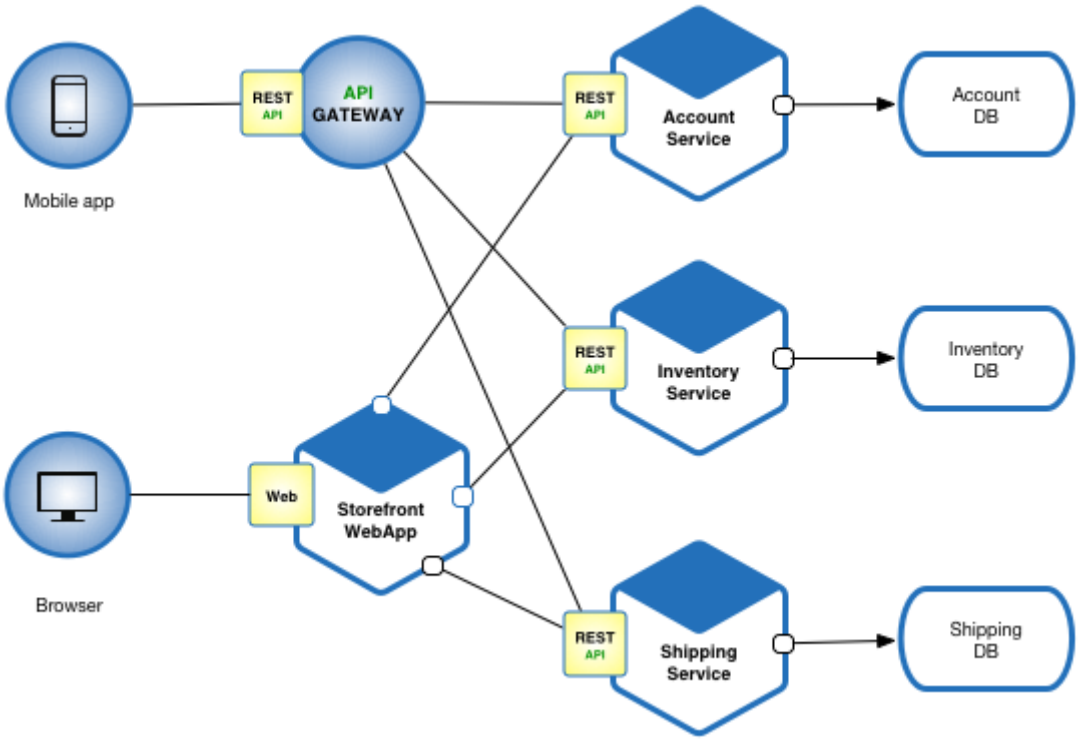
Tambien se busca que los nuevos miembros del equipo puedan ser productivos rapidamente, para lo cual se necesita que la aplicacion sea facilmente comprensible y modificable.

En ocasiones se aplican tecnicas de desarrollo agil, realizando despligue continuo, para lo cual la aplicacion debe poder pasar de desarrollo a producción de forma rapida.

Generalmente las aplicaciones necesitan ser escaladas y tienen criterios de disponiblidad, por lo que se necesitan ejecutar multiples copias de la aplicación.

Y siempre es interesante poder aplicar las nuevas tendencias en frameworks y tecnologias qeu mejoran los desarrollos, por lo que es interesante que los nuevos desarrollos no se vean obligados a seguir tecnologias de los antiguos.

Por estos motivos, será interesante aplicar una arquitectura con bajo acoplamiento y servicios colaborativos, el bajo acoplamiento lo conseguiremos empleando servicios HTTP y protocolos asincronos como AMQP y haciendo que cada microservicio tenga su propia base de datos.



1.2. Ventajas de los Microservicios

- Son ligeros, desarrollados con poco codigo.
- Permiten aprovechar de forma mas eficiente los recursos, ya que al ser cada microservicio una aplicación en sí, se pueden aumentar los recursos de dicha funcionalidad sin tener que aumnetar los recursos de otras piezas que quizas no los necesiten, por lo que los recursos son asignados de forma más granular.
- Permiten emplear tecnologias distintas, aprovechando las ventajas puntuales de cada una de ellas, sin que esas ventajas puedan lastrar otros componentes.
- Permiten realizar despliegues más rapidos, ya que evoluciones que se produzcan en un microservicio, al ser independientes no deben afectar a otras piezas del puzle y por tanto según esten terminados se pueden poner en producción, sin tener que esperar a hacer un despliegue de una version completa de toda la aplicación, esto unido a que los microservicios son **pequeños**, hace que el periodo desde que se comienza a desarrollar la mejora hasta la puesta en producción, sea corto y por tanto facilite la aplicación de **Continuous Delivery** (entrega continua).
- Mejoran el comportamiento de las aplicaciones frente a los fallos, ya que estos quedan mas aislados

1.3. Desventajas de los Microservicios

- Exigen mayor esfuerzo en el despliegue, control del versionado, copatibilidad entre versiones, actualización, monitorización, ..
- Es mas complejo realizar test de integracion.
- No existe un ambito transaccional entre todos los microservicios que participan.
- Con aplicaciones existentes, es mas complicado de aplicar.

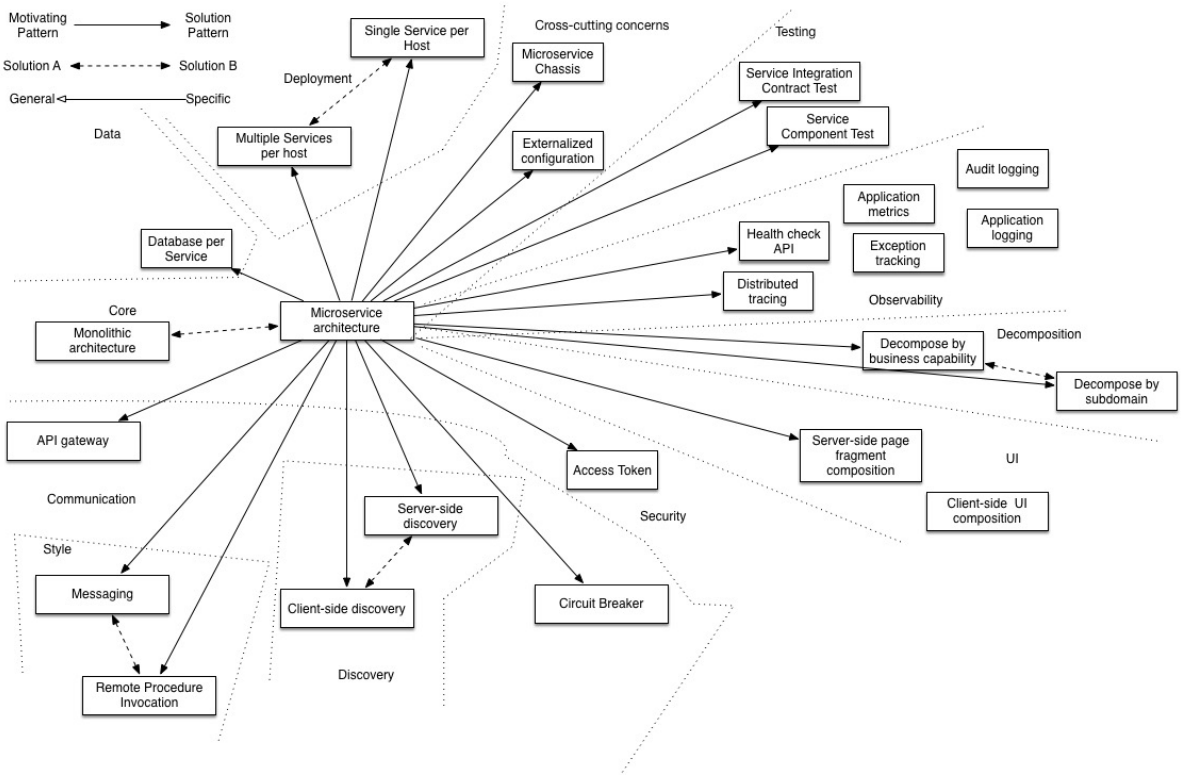
1.4. Arquitectura de Microservicios

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 microservicos,
- Detección de caida de microservicios,
- Balanceo de carga,
- Seguridad centralizada,
- Logs centralizados, ..

1.4.1. Patrones



Para solventarlos, se suelen emplear patrones, aqui tenemos algunos de los mas importantes agrupados

- **Principio de la responsabilidad unica**: Cada servicio debe tener responsabilidad sobre una sola parte de la funcionalidad total. La responsabilidad debe estar encapsulada en su totalidad por el servicio. Se dice que un **Servicio** debe tener solo una razón para cambiar.
- **Common Closure Principle**: Los componentes que cambian juntos, deben de estar empaquetados juntos, por lo que cada cambio afecta a un solo servicio.

Descomposicion

- **Descomposicion por capacidades de negocio**: Cada funcionalidad del negocio se encapsula en un unico servicio. Por ejemplo: **Gestor del catalogo de productos**, **Gestor del inventario**, **Gestor de compras**, **Gestor de envios**, …¬
- **Descomposicion por acciones**: Descomponer la aplicacion en microservicios por las acciones/casos de uso que ha de implementar. Por ejemplo: **Servicio de Envios**.
- **Descomposicion por recursos**: Descomponer la aplicacion en microservicios por los recursos que manejan. Por ejemplo: **Servicio de Usuario**.

Despliegue

- **Multiples servicios por servidor**: Se aprovechan mas los recursos, en cambio es mas dificil medir los recursos que necesita cada servicio, a parte de posibles conflictos en cuanto a los recursos compartidos.
- **Un Servicio por servidor**: Es mas sencillo redesplegar, gestionar y monitorizar el servicio, no existiendo conflictos con otros servicios, como pega es el no aprovechamiento optimo de los recursos, dado que el contenedor, consume parte de los recursos.
- **Un servicio por VM**: Facilita el despliegue dado que permite definir los requisitos del servicio de forma aislada, sin tener sorpresas, tambien permite gestionar aspectos del despliegue como memoria, CPU, …¬ El escalado es mas facil, como pega se tiene el tiempo que se tarda en montar la VM.
- **Un servicio por Contenedor**: Se trata de encapsular el despliegue del servicio dentro de un contenedor como Docker, los pros y contras son similares a los de la VM.
- **Serverless deployment**: Se basa en el empleo de una infraestructura de servicios, que abstrae el concepto de entorno de despliegue, basicamente se pone la aplicacion en el servicio y este reserva recursos y la pone en funcionamiento. Algunas son: AWS Lambda, Google Cloud Functions o Azure Functions.
- **Service deployment platform**: Se basa en el empleo de una plataforma de despliegue, que abstrae el servicio, porporcionando mecanismos de alta disponibilidad por nombre de servicio. Algunos ejemplo de plataformas son: Docker Swarm, Kubernetes, Cloud Foundry o AWS Elastic Beanstalk.

Cross cutting concerns (Conceptos Transversales)

- **Chassis**: Establece la conveniencia de usar una herramienta que gestione la configuracion de los aspectos trasversales del desarrollo como son: Externalizar configuraciones como credenciales de acceso o localizacion de los servicios como son Bases de datos o Brokers de mensajeria, configuracion del framework de Logging, Servicios de monitorizacion del estado de la aplicacion, Servicios de metricas que permiten ajustar el rendimiento, Servicios de traza distribuida.
- **Externalizar configuraciones**: Establece la necesidad de externalizar la configuracion de tal forma que la aplicación pueda obtener su configuracion en la fase de arranque, para independizar su despliegue del entorno, que la aplicación

no tenga que cambiar en cada despliegue. 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.

Formas de comunicacion

- **Remote Procedure Invocation**: Permite definir una interface comunicacion entre las aplicaciones clientes y los servicios, asi como entre los propios servicios cuando han de colaborar para satisfacer la peticion del cliente. La implementacion mas habitual es REST.
- **Messaging**: Permite la comunicacion asincrona y desacoplada entre los servicios. Algunas implementaciones son Apache Kafka y RabbitMQ.
- **Domain-specific protocol**: Describe el uso de un protocolo especifico para el dominio tratado, como puede ser SMTP para los correos electronicos.

Exponer APIs

API Gateway

Es el punto de entrada para los clientes, este puede simplemente actuar como un enrutador o bien componer una respuesta basada en peticiones a varios servicios.

- **Ventajas**:
  - ¬ Aísla a los clientes de cómo la aplicación se divide en microservicios
  - ¬ Aísla a los clientes del problema de determinar las ubicaciones de las instancias de servicio
  - ¬ Proporciona un API óptimo para cada cliente
- Reduce el número de peticiones. Ya que es capaz de componer la respuesta a la petición recupernado informacion de varios microservicios
- Simplifica el cliente moviendo la lógica para llamar múltiples servicios desde el cliente a la puerta de enlace API
- Se traduce de un protocolo API público estándar y fácil de usar a cualquier protocolo que se use internamente.
- **Desventajas**:
  - ¬ Mayor complejidad: la puerta de enlace API es otra parte móvil que debe desarrollarse, implementarse y administrarse

- ¬ Aumento del tiempo de respuesta debido a incluir otro componente en la peticion, este será solo perceptible en llamadas a un unico microservicio, cuando se involucren dos o mas, el aunmente de tiempo se compensa al realizar una unica peticion.

Backend for front-end

Caso particular del anterior, donde se proporciona un **API Gatway** para cada tipo de cliente: movil, web, . . .

Enrutador

Permite exponer todos los servicios que los clientes necesitan, independientemente del tipo de cliente.

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.

Descubrir Servicios

- **Service registry**: Servicio que mantiene las ubicaciones de las distintas instancias de los microservicios y que es consultable.
- **Client-side discovery**: La forma en la que un cliente (API Gateway u otro mecroservicio) obtiene la referencia a un microservicio, es a traves del servicio de registro, donde todas las instancias de todos los microervicios se han de registrar para que el resto los encuentre.
- **Server-side discovery**: El cliente no accede directamente al servicio, sino que lo accede a traves de un enrutador, que consulta al servicio de registro.
- **Self registration**: Capacidad de los servicios para registrarse en el servicio de registro de forma autonoma, para quitar responsabilidad y complejidad al servicio de registro.
- **3rd party registration**: Cuando el registro lo realiza una herramienta independiente.

Confiabilidad (Reliability)

CircuitBreaker

El patrón **Control de ruptura de comunicación con los servicios** permite controlar que la caida de un microservicio consultado, no provoque la caida de los microservicios que realizan la consulta, evitando así una potencial caida de servicios en cascada.

Para ello proporciona un resultado estatico para la consulta.

- **Ventajas**:
- El servicio maneja los fallos de otros servicios a los que invoca
- **Desventajas**:
- Será complicado elegir el **timeout** adecuado sin crear falsos errores (en caso de ser excesivamente estrictos) o crear latencia en la consecución de la tarea (en caso de ser excesivamente permisivo).

Gestion de Datos

Database per Service

Cada servicio tiene su propia base de datos, que solo es accesible mediante el API que proporciona el servicio, para ello se pueden tener tablas privadas por servicio, esquemas por servicio o incluso base de datos por servicio.

- **Ventajas**:
  - ¬ Bajo acoplamiento entre servicios, el despliegue de una app, no implica que otra se haya desplegado.
  - ¬ Los cambios en el modelo solo afectan a una app, por lo que se pueden hacer mas libremente.
  - ¬ Cada servicio puede emplear la tecnologia de persistencia mas adecuada para su caso de uso
- **Desventajas**:
  - ¬ No será posible trabajar con transacciones distribuidas ya que REST no es transaccional (Se puede implementar el patrón **saga**)
  - ¬ Complejidad de implementar consultas con **joins**, ya que la composición del join se debera realizar en una aplicación.

Shared database

Al contrario que la anterior, indica compartir una misma base de datos por todos los microservicios.

- **Ventajas**:
  - ¬ Proporciona posibilidad de realizar **joins**.
- **Desventajas**:
  - ¬ Necesidad de que los equipos coordinen cambios en el esquema.
  - ¬ Problemas de rendimiento al acceder tantos microservicios a la misma base de datos.
  - ¬ Que algunos microservicios no puedan adaptar sus necesidades a esa situación.
  - ¬ Mas facilidad en saltarse el principio de responsabilidad única.

Patrón Saga

Permite mantener la consistencia de los datos, que queda afectada por el echo de que cada microservicio tiene su propia base de datos y no es posible aplicar transacciones distribuidas.

La idea es que cuando un microservicio necesite actualizar sus datos, pero esa actualizacion necesite de un segundo microservicio, el primer microservicio deja la modificacion marcada como pendiente, a la espera de los datos del segundo microservicio y lanza un evento que hace ejecutarse el segundo microservicio, este realiza su tarea y lanza otro envento para comunicar al primer microservicio el estado en el que ha quedado su tarea, realizando este la modificacion pendiente que tenia.

Será necesario tener eventos de **commit** y de **rollback**

- **Ejemplo**
  - ¬ El **ServicioPedido** inserta en la BD de pedidos, un nuevo **Pedido** con estado **pendiente** y lanza el evento **PedidoCreadoEvento**
  - ¬ El **ServicioCliente** recibe el evento **PedidoCreadoEvento** y comprueba que el **Cliente** tiene credito para pagar el **Pedido**. Si lo tiene lanza un evento **CreditoReservadoEvento** y sino lo tiene lanza un **CreditoExcedidoEvento**.

    - El **ServicioPedido** recibe tanto el evento **CreditoReservadoEvento** que actualiza el **Pedido** a estado **aprobado**, como el evento **CreditoExcedidoEvento** que actualiza el **Pedido** a estado **cancelado**.

- **Ventajas**:
    - Permite que una aplicación mantenga la consistencia de los datos en múltiples servicios sin usar transacciones distribuidas
- **Desventajas**:
    - Mas complejo de desarrollar. Se han de programar comandos de compensación (rollback) que deshagan explícitamente los cambios realizados por cada tarea de la **transaccion**.

API Composition

Describe la creación de un nuevo servicio, que se encarga de consultar a cada servicio propietario de los datos, realizando un **join** en memoria.

Un API Gateway puede realizar esta operacion.

- **Ventajas**:
    - Forma simple de consultar datos en una arquitectura de microservicio.
- **Desventajas**:
    - Algunas consultas pueden ser ineficaces haciendo un uso exhaustivo de la memoria.

CQRS (Command Query Responsibility Segregation)

Describe la separación de las operaciones sobre los datos en dos partes **command** (actualizan el estado) y **query** (consultan el estado). Esto permite aprovechar dos ventajas, la primera es poder escalar independientemente las consultas y la administracion, generalmente las consultas exigen muchos mas recursos y la segunda poder emplear una tecnologia mas adecuada para cada caso, por ejemplo una RDBS para la administracion y una NoSQL para las consultas. Se habrá de implementar un sistema que permita la sincronización de los sistemas, de tal forma que el sistema de las **query** se mantenga actualizado, para ello una de las aproximaciones mas empleadas es **Event Sourcing**, que permite que dicha actualizacion de provoque bajo la subscripcion a los eventos generados por los **command**.

Event sourcing

Describe como se persiste el estado de una entidad, guardando un listado de eventos que han llevado a la entidad al estado actual, recomoniendo el estado cada vez que es necesario, para mejorar el rendimiento ante entidades que modifican su estado con asiduidad, se realizan snapshot que sirven para recomponer el estado aplicando unicamente los ultimos eventos.

- **Application events**: Describe como se insertan registros en una tabla **events** por cada operacion y un proceso los publica en un **message broker**.
- **Traza de seguimiento de transacciones (Transaction log tailing)**: Describe la publicacion del log generado por cada transaccion como evento para trasladar los cambios al resto de microservicios.

Seguridad

- **Access Token**: Define la autenticación centralizada en el **API Gateway**, ya que es el que interaciona con el cliente, que obtiene un **token** que traslada al resto de microservicios para que estos puedan asegurar que el cliente que realiza la petición tiene permisos para invocarlos.

Observacion

- **Gestión centralizada de logs**: Define la generación de una traza formada por todas las trazas generadas por los microservicios participantes en una petición.
- **Metricas de aplicacion**: Define la creación de servicios que permiten obtener el estado de los microservicios.

1.4.2. Orquestacion vs Coreografia

Se habla de **Orquestación**, cuando una aplicacion, gestiona como invocar a otras aplicaciones, estableciendo los criterios y orden de invocacion (API Gateway).

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

1.5. Servidor de Configuracion

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

El servidor se conecta a un repositorio **git** de donde saca las configuraciones que expone, lo que permite versionar facilmente 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/victorherrerocazurro/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 **yml** 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.yml**

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

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

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

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{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**

1.5.1. 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 configrar con Srping Security.

Se puede establecer con la propiedad

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

1.5.2. Clientes 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 Configuracion**. 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 configuracion, se puede modificar añadiendo al **application.yml**

```
server:
  port: 8082
```

Dado que el **Servidor de Configuración** estará securizado se debera 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 algun problema al obtener la configuración, se puede definir

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

Una vez configurado el acceso del microservicio al **Servidor de Configuracion**, 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**.

1.5.3. 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, habrà 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

1.6. Servidor de Registro y Descubrimiento

Permite gestionar todas las instancias disponibles de los microservicios.

Los microservicios enviaran su estado al servidor Eureka a traves 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, Zookeper, 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);
    }
}
```

1.6.1. Registrar Microservicio

Lo primero para poder registrar un microservicio en el servidor de descubrimineto 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
```

1.7. 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 proxeado, 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 ientificador del servicio en **Eureka** con el servicio real, se debe configurar donde encntrar 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);
  }
}
```

1.7.1. 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
```

1.8. Simplificación de Clientes de Microservicios con Feign

Feign abstrae el uso del API de RestTemplate para consultar los microservicios, encapsulandolo 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 aplicación **@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
```

1.8.1. 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");
    }
}
```

1.8.2. 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/
```

1.9. 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 siguientee 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/
```

1.9.1. Seguridad

En el caso de enrutar hacia servicios seguros, se puede configrar **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 desacivada 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
```

1.10. Circuit Breaker

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

Para ello Spring Cloud integra **Hystrix**.

Para emplearlo, se ha de añadr 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 via de ejecución alterantiva **fallback**, para así evitar que se propague el error en la invocacion (try-catch).

Cuando se detectan una serie de errores en la respuesta, el proxy abre la conexión deribando al **falback**, aunque mantiene una validación periodica del estado de la respuesta, cerrando nuevamente la conexión cuando la respuesta vuelve a ser la correcta, este proceso puede ser monitorizado ya que se genera un stream con el estado de las conexiones.

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** debera retornar el mismo tipo de dato que el método proxeado, generalmente retornará una cache con el resultado de la última petición que se resolvio de forma correcta.

NOTE	No deberan 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);
  }
}
```

1.10.1. 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 traves 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>
```

1.10.2. 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>
```

1.11. Configuracion Distribuida en Bus de Mensajeria

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

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

Para instalar RabbitMQ, se necesita instalar Erlang a parte de RabbitMQ

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

1.11.1. Servidor

Se necesitará incluir las siguientes dependencias en el servidor de configuracion

```
<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 configuracion de la ubicacion 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 idealmente deberá acceder el repositorio de código donde se alberguen las configuraciones, para que cuando se produzca un commit nuevo, invocar el servicio, por ejemplo, GitHub proporciona **WebbHooks** para ello.

```
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": []} }'
```

1.11.2. 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 **RabitMQ**, al que habrá que configurar las siguientes propiedades

application.properties

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

1.12. OAuth2

1.12.1. Caracteristicas

- Emplea Https en vez de la Criptografia de OAuth 1.
- Permite flujos con aplicaciones nativas y no solo con Navegadores.
- Posibilidad de asignacion de Privilegios a los Token basados en la caracteristica SCOPE (LEER, ESCRIBIR, ..)
- Caducidad de los Token, **Refresh Token**.

1.12.2. Que se quiere conseguir

- Diferenciar cuando el acceso a los datos privados los hace el dueño de los datos y cuando lo hace una aplicación en nombre del usuario.
- Permitir al usuario dar distintos privilegios de acceso a las distintas aplicaciones que acceden en nombre del usuario a sus datos privados.
- Poder revocar los privilegios concedidos.

1.12.3. Actores

- **Resource Owner** - Usuario propietario de los **datos seguros**.
- **Resource Server** - Servidor que alberga los **datos seguros**, el servidor que realiza la autenticación por OAuth puede ser uno de ellos, por ejemplo permitiendo el acceso a los datos del perfil del usuario.

- **Authorization Server** - Plataforma OAuth, controla que clientes pueden acceder a los **datos seguros**.
- **Client** - Aplicacion que quiere acceder a los **datos seguros** de un usuario, residentes en otra aplicación y cuyo usuario (propietario) es usuario de las dos aplicaciones tanto del cliente como del servidor de recursos.

1.12.4. Caracteristicas

El **Resource Owner** es usuario del **Authorization Server**.

La aplicacion **Client**, se registra en el **Authorization Server**, para delegar en él, el proceso de Login.

El usuario del **Authorization Server**, otorga permisos a la aplicacion **Client** para acceder a determinados **Resource** a través de los SCOPE, esta asignación de SCOPE es revocable.

La aplicación **Client** solicitará una serie de SCOPE, pero el usuario le otorgará los que el decida, de no otorgarle todos los SCOPE, la aplicación cliente puede no funcionar correctamente.

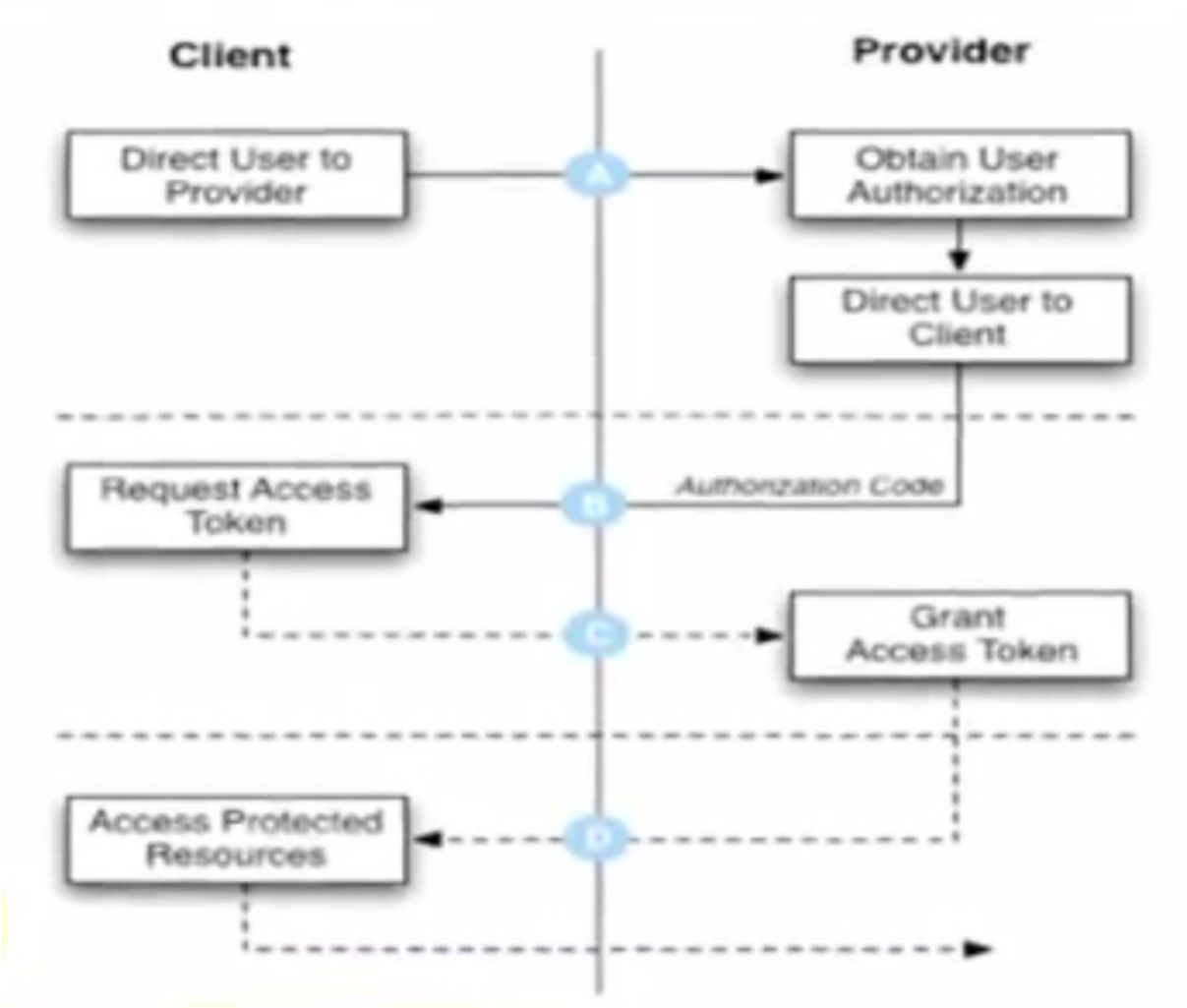
Los SCOPE, son algo similar a los ROLES (READ, WRITE, …¬.), que son definidos por el servidor de recursos. Se definen de forma anologa a los ROLES asociados a las URLs.

1.12.5. Grants

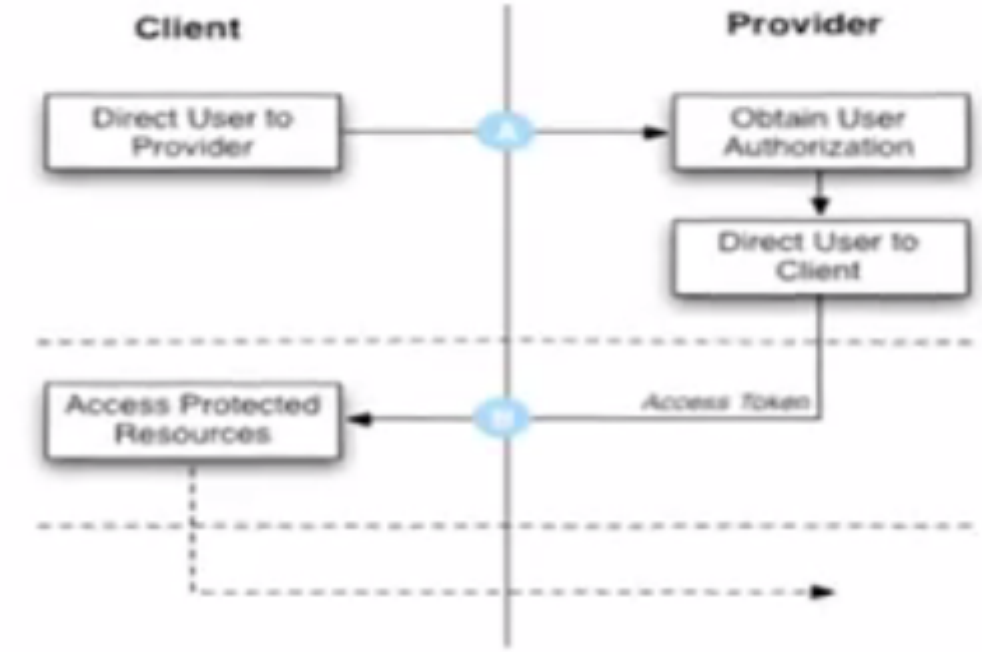
Describen el dialogo que se produce entre los distintos actores en distintos escenarios.

En **Oauth2** se describen 5 formas por las cuales los clientes pueden obtener el **AccessToken**

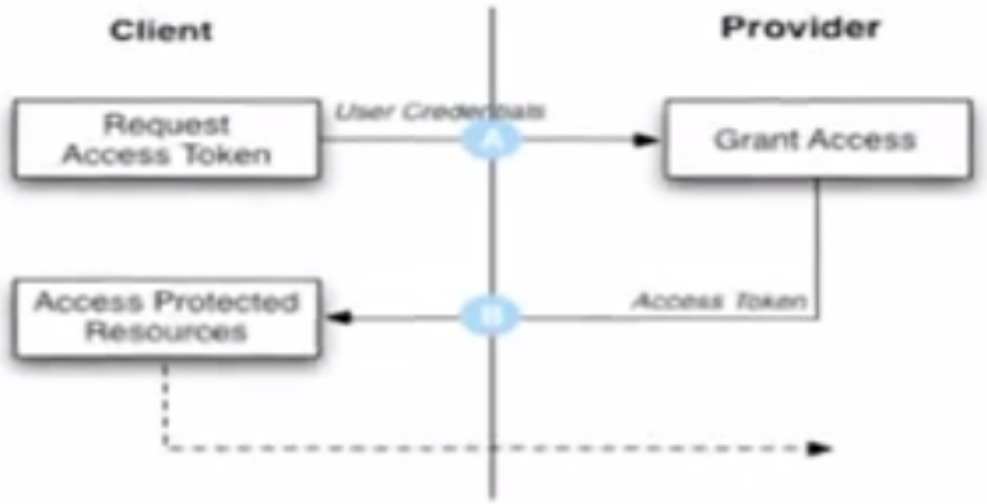
- **Authorization Code Grant**: Es el más habitual en las aplicaciones web.



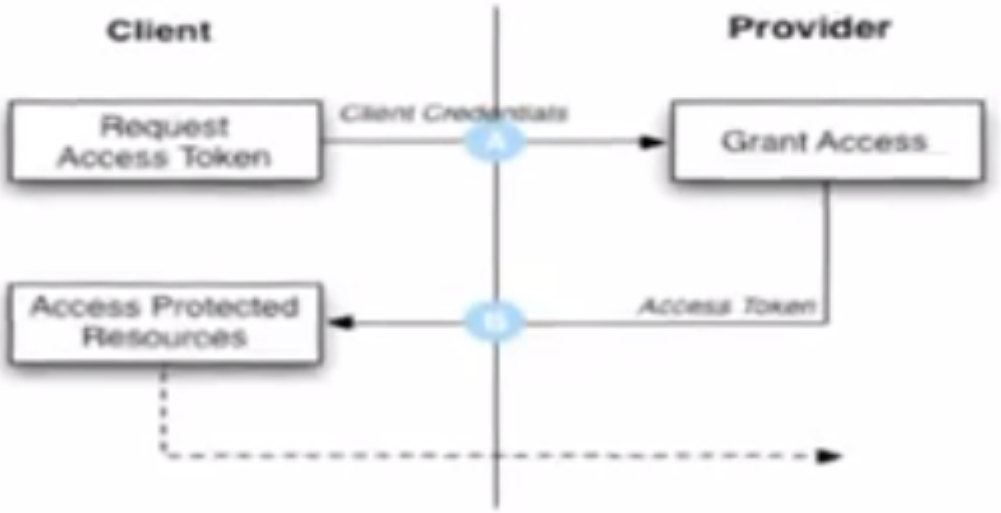
- **Implicit Grant**: Aplicaciones Javascript, que reducen el dialogo.



- **Resource Owner Credential Grant** o **Password Grant**: Para aplicaciones muy confiables, dado que se les da el user/password.



- **Client Credential Grant**: Empleada en comunicaciones entre aplicaciones, donde que el usuario epecifique los permisos no es necesario.



- **Refresh token grant**: Permite obtener un AccessToken nuevo cuando el antiguo caduca.

Authorization Code Grant

1- Se define el **Authorization Server**.

2- Se define la aplicación **Client**, que se ha de dar de alta en el **Authorization Server**, indicando un **Id** y un **Secret**, que representan el login de la aplicacion **Client**. Esta aplicación, deberá almacenar estos datos porque se los tendrá que enviar posteriormente al **Authorization Server**.

3- De forma analoga, el **Resource Owner** se registrará en el **Authorization Server** con un **login** y **password**.

4- El **Resource Owner** accede a la aplicacion **Client** y está lo redirecciona al **Authorization Server**, enviando a dicho servidor los siguientes datos

- **response_type**: Indica que es lo que espera la aplicacion cliente del proceso de login del usuario en el **Authorization Server**. En este caso se indica que se quiere un **autorization code**, para ello se indica el valor **code**
- **client_id**: con el identificador de la aplicacion cliente en el **Authorization Server**
- **redirect_uri**: con la url a la que el **Authorization Server** deberá redireccionar al usuario una vez terminado el login y aprovación de scopes.
- **scope**: listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **state**: con el **CSRFToken**

Si el usuario aprueba los **scopes**, el **Authorization Server** lo redirecciona al **redirect_uri** o a uno especificado por la aplicacion cliente de forma generica, enviando

- **code**: con el **authorization code**
- **state**: con el mismo **CSRFToken** que se recibio

5- Cuando la aplicacion **Client** recibe la petición, realiza una peticion al **Authorization Server** enviando

- **grant_type**: Indica en tipo de Grant a emplear, en este caso con valor **authorization_code**.
- **client_id**: con el identificador de la aplicacion cliente en el **Authorization Server**.
- **client_secret**: con la clave secreta asociada al cliente.
- **redirect_uri**: con la misma uri que paso por parametros al **Authorization Server** cuando redirecciono al usuario.

- **code**: con el **authorization code** que obtuvo de la anterior redireccion.

Y el **Authorization Server** le retorna un JSON con

- **token_type**: con valor **Bearer**.
- **expires_in**: con un entero que indica cuando expira el Token.
- **access_token**: con el **accessToken**.
- **refresh_token**: con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado.

6- Por último se produce el acceso a los **Resource**, ya que la aplicacion **Client** tiene el **accessToken**, le pide con este Token los datos privados al **Resource Server**, el cual se los da, dado que internamente es capaz de validarlo contra el **Authorization Server**.

Implicit grant

Similar al anterior, pero dado que es el empleado por lo browser y estos no son capaces de asegurar guardar un **secret** de la aplicacion **Client**, el flujo se simplifica, dado que en la redireccion del usuario hacia el **Authorization Server** se obtendrá el **accesstoken**, no existen los **authorization code** y ni tampoco **refresh_token** dado que el browser no lo podria almacenar de forma segura.

1- Se define el **Authorization Server**.

2- Se define la aplicación **Client**, que se ha de dar de alta en el **Authorization Server**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **Authorization Server**.

3- El usuario se registra en el **Authorization Server**.

4- El usuario accede a la aplicacion **Client** y está lo redirecciona al **Authorization Server**, enviando a dicho servidor los siguientes datos

- **response_type** con el valor **token**.
- **client_id** con el id de la aplicacion cliente.
- **redirect_uri** con la url a la que el **Authorization Server** deberá redireccionar al usuario una vez terminado el login y aprovación de scopes.
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

- **state** con el **CSRFToken**.

Si el usuario aprueba los **scopes**, el **Authorization Server** lo redirecciona a **redirect_uri** enviando

- **token_type** con el valor **Bearer**.
- **expires_in** con un entero que indica cuando expira el Token.
- **access_token** con el **accessToken**.
- **state** con el mismo **CSRFToken** que se recibio.

Resource owner credentials grant

Es una forma de obtener el **accessToken** en la que hay una completa confianza en la aplicación **Client**, dado que se le dan **login/password**.

1- Se define el **Authorization Server**.

2- Se define la aplicación **Client**, que se ha de dar de alta en el **Authorization Server**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **Authorization Server**.

3- El usuario se registra en el **Authorization Server**.

4- El usuario accede a la aplicacion **Client** y está le pregunta cual es su **login/password**

5- La aplicacion cliente realiza una peticion al **Authorization Server**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **password**.
- **client_id** con el identificador de la aplicacion cliente en el **Authorization Server**.
- **client_secret** con la clave secreta asociada al cliente.
- **scope** listado de scopes separados por comas que la aplicacion **Client** requiere al usuario.
- **username** el login obtenido del usuario.
- **password** la contraseña obtenida del usuario.

El **Authorization Server** responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**
- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

Client credentials grant

Es el mas simple de los **Grant** de OAuth2

1- Se define el **Authorization Server**.

2- Se define la aplicación **Client**, que se ha de dar de alta en el **Authorization Server**, obteniendo un **Id** y un +secret*, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **Authorization Server**.

3- La aplicacion **cliente** realiza una peticion POST al **Authorization Server**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **client_credentials**.
- **client_id** con el identificador de la aplicacion **Client** en el **Authorization Server**.
- **client_secret** con la clave secreta asociada a la aplicacion **Client**.
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**.
- **expires_in** con un entero que indica cuando expira el Token.
- **access_token** con el **accessToken**.

Refresh token grant

Se emplea cuando se puede regenerar el **accessToken** sin la intervencion del usuario

1- La aplicacion **cliente** realiza una peticion POST al **Authorization Server**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **refresh_token**
- **refresh_token** con el **refreshToken** que se obtuvo con el ahora caducado **accessToken**
- **client_id** con el identificador de la aplicacion **Client** en el **Authorization Server**
- **client_secret** con la clave secreta asociada a la aplicacion **Client**.
- **scope** listado de scopes separados por comas que la aplicacion **Client** requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**.
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**.
- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado.

1.13. Trazas distribuidas

Existen varias soluciones para centralizar el conjunto de trazas producidas por una arquitectura de microservicios

- **Zipkin**
- **ELK**
- **Loggy**
- **Graylog**
- **Splunk**

En algunos casos las herramientas se basan en generar un nuevo **Appender** para los logs, en todo caso todos coinciden en enviar los logs a un servicio que los centraliza.

1.13.1. Zipkin

Es un software desarrollado por **Twitter** que permite resolver el problema que se presenta en las arquitecturas de microservicios para el seguimiento de las trazas producidas por una petición que atraviesa varios microservicios.

Tiene los siguientes modulos:

- **Collector**: El encargado de recibir las trazas de los microservicios, que valida y envia al modulo de **Storage**.
- **Storage**: El encargado de almacenar e indexar las trazas. Se soportan: Cassandra, ElasticSearch y MySQL.
- **Search**: Permite realizar busquedas de trazas a traves de un Api JSON. El principal consumidor de este modulo es el de **Web UI**.
- **Web UI**: Aplicación web para visualizar las trazas.

Para instalarlo, se puede descargar el jar desde aquí, o emplear una imagen de Docker

```
> docker run -d -p 9411:9411 --name=zipkin openzipkin/zipkin
```

El acceso a la herramienta web, se hace a través del puerto **9411**

```
http://localhost:9411/zipkin
```

Para enviar las trazas a **Zipkin** en aplicaciones Spring Boot, se ha de añadir la siguiente depedencia

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

1.13.2. Sleuth

API que permite generar el **TraceId** y el **SpanId** y añadirlo a las cabeceras de las comunicaciones.

Es empleado junto con herramientas como **Zipkin** o **ELK**.

Por incluirse en el classpath automaticamente afecta a las comunicaciones realizadas con:

- RestTemplate.

- Netflix Zuul.
- Spring MVC controllers.
- Apache Kafka, RabbitMQ, etc.

Para emplear **Sleuth** unicamente hay que añadir la siguiente depedencia

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

Será necesario definir un **Sampler** para indicar cuando una traza ha de ser enviada al servicio.

```
@Bean
public AlwaysSampler alwaysSampler() {
    return new AlwaysSampler();
}
```

Se dispone de las siguientes opciones

- **AlwaysSampler**
- **IsTracingSampler**
- **NeverSampler**
- **PercentageBasedSampler**

Tambien es necesario que las aplicaciones esten identificadas a través del **spring.application.name**

Se puede definir el endpoint de **Zipkin** con la propiedad **spring.zipkin.baseUrl=http://192.168.99.100:9411/**

Con esto, lo único que es necesario es añadir logs a la aplicación

```
@RestController
class ZipkinController {

    @Autowired
    private RestTemplate restTemplate;

    private static final Logger LOG = Logger.getLogger(
ZipkinController.class.getName());

    @GetMapping(value = "/zipkin")
    public String service() throws RestClientException,
URISyntaxException {
        LOG.info("Primer Servicio");

        ResponseEntity<String> response = restTemplate.getForEntity(new
URI("http://localhost:8082/zipkin2"), String.class);

        return "Primer Servicio -> " + response.getBody();
    }
}
```

Para apreciar la monitorización puede ser interesante introducir retardos

```
@RestController
class ZipkinController {

    @Autowired
    private RestTemplate restTemplate;

    private static final Logger LOG = Logger.getLogger(
ZipkinController.class.getName());

    @GetMapping(value = "/zipkin2")
    public String service() {
        LOG.info("Segundo Servicio");

        LOG.info("Se incluye el retardo");

        try {
            Thread.sleep(20 * 1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        LOG.info("Finaliza la espera");

        ResponseEntity<String> response = restTemplate.getForEntity(new
URI("http://localhost:8083/zipkin3"), String.class);

        return "Primer Servicio -> " + response.getBody();
    }
```

1.14. Spring Boot Admin

Herramienta de monitorizacion de aplicaciones **Spring Boot**, que emplea los servicios publicados con **Spring Boot Actuator**.

1.14.1. Servidor

Para incluirlo en un proyecto se ha de añadir la siguiente dependencia

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

Y habilitar el api con la anotacion **@EnableAdminServer**.

Se puede configurar tambien el path para acceder al servicio

```
spring:
  boot:
    admin:
      context-path : admin
```

1.14.2. Cliente

Para incluirlo en un proyecto se ha de añadir la siguiente dependencia

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

Y añadir en la configuracion **application.properties** la url del servidor.

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:8080
```

Para tener una visualizacion de los datos completa que proporciona **Actuator** se puede añadir la siguiente configuracion a los microservicios.

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    health:
      show-details: ALWAYS
```

1.14.3. Discovery

Si se tiene un servidor de registro y descubrimineto como **Eureka**, se puede delegar en el, el descubrimiento de los servicios, con lo que no es necesario añadir la dependencia de **spring-boot-admin-starter-client** a los microservicios.

Bastará con añadir la dependencia de **Eureka Client** al servidor de monitorizacion

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

Y activar el descibrimiento con la anotacion @**EnableDiscoveryClient**.

Esta podria ser una configuracion valida para el caso de tener en una misma aplicacion, los servicios de registro y monitorizacion.

```
<dependencies>
    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-starter-server</artifactId>
    </dependency>
    <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-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Fichero Application.java

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

Fichero application.yml

```yaml
spring:
  boot:
    admin:
      context-path : admin

server:
  port: 8084

eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
    health-check-url-path: /actuator/health
  client:
    registryFetchIntervalSeconds: 5
    serviceUrl:
      defaultZone: http://localhost:${server.port}/eureka/

management:
  endpoints:
    web:
      exposure:
        include: "*"
  endpoint:
    health:
      show-details: ALWAYS
```