Microservicios

N-Layer architecture

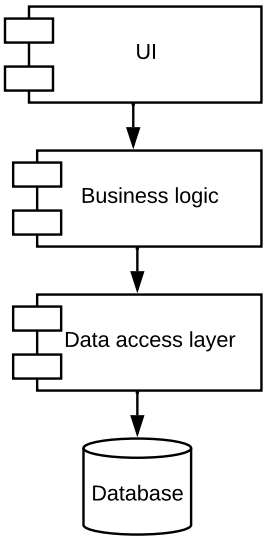
Un tipo común de arquitectura es la arquitectura basada en múltiples capas (N-layer), con esto se busca que cada capa tenga una responsabilidad definida dentro de una aplicación, un ejemplo común era la separación entre la capa de presentación, de lógica de negocio y de acceso a datos.

Microservices

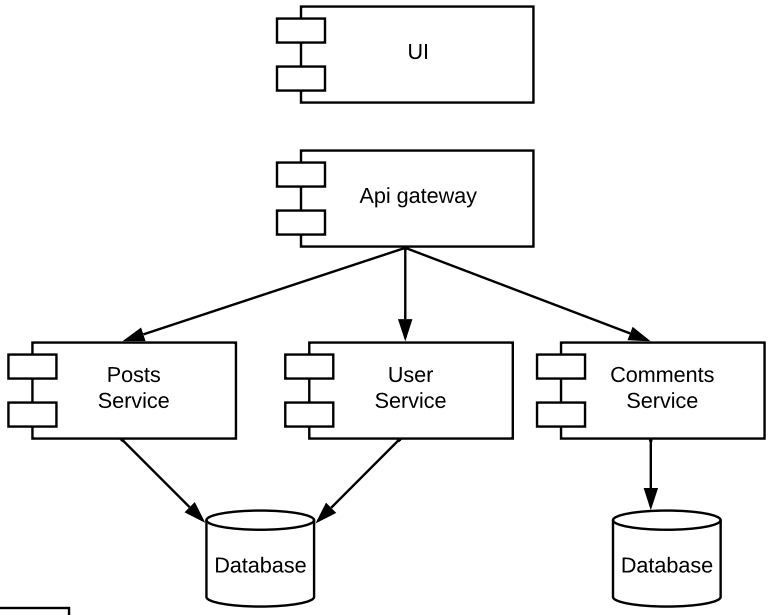
Los microservicios buscan descomponer aplicaciones monolíticas grandes en piezas más pequeñas, con bajo acoplamiento, distribuidas y fáciles de manejar.

< DEVS4J >
Hecho por Devs para Devs

Monolithic architecture



Microservices architecture



Características de los microservicios

Los microservicios tienen las siguientes características:

- La lógica de la aplicación se separa en **pequeños componentes** bien definidos con una responsabilidad limitada.
- Cada componente tiene una pequeña responsabilidad del dominio y es **desplegada de forma independiente** de las demás.
- Emplean **protocolos ligeros** de comunicación como HTTP o JSON.
- Al comunicarse con protocolos comunes los **detalles de implementación son irrelevantes**. Lo que resulta en que pueden ser escritos en diferentes lenguajes de programación.
- Al ser pequeñas piezas e independientes permiten a las empresas tener equipos de desarrollo más pequeños y bien definidos con una responsabilidad específica.

Objetivo de los sistemas

El objetivo es construir sistemas que sean:

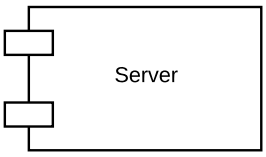
- **Flexibles**: Servicios desacoplados en los que se puedan agregar nuevas funcionalidades de forma rápida con el menor cambio posible.
- **Resiliente**: Fallos pueden ser localizados en pequeñas partes de la aplicación y pueden ser contenidos sin que la aplicación completa se caiga.
- **Escalable**: Servicios que puedan ser distribuidos de forma horizontal permitiendo que la aplicación crezca de forma adecuada.

Escalamiento vertical

Antes

RAM - 4GB
CORES - 2

Después



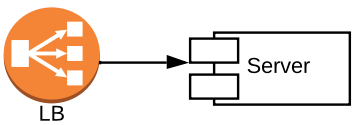
RAM - 8GB
CORES - 4

El escalamiento vertical re-dimensiona un servidor existente incrementando puntos como su memoria, cpu o almacenamiento.

La principal desventaja es que es un solo punto de fallo, un servidor grande puede hostear una aplicación monolítica compleja, pero si ese servidor falla, la aplicación completa se pone en riesgo.

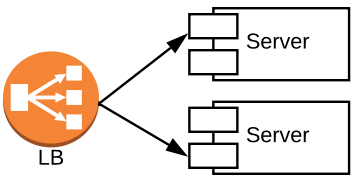
Escalamiento horizontal

Antes



RAM - 4GB
CORES - 2

Después



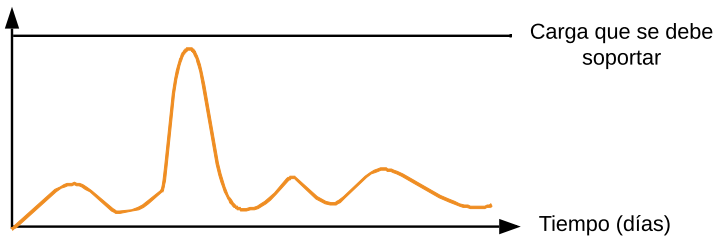
RAM - 4GB
CORES - 2

RAM - 4GB
CORES - 2

El **escalamiento horizontal** no modifica los servidores existentes si no que **agrega nuevos servidores pequeños** al balanceador de carga.

Esto permite que se **agreguen/remuevan** servidores como sea necesario, optimizando los recursos lo más posible(Es posible hacerlo de forma automática a través de métricas "autoscaling").

Uso de recursos



Cloud computing

Cloud computing

El cómputo en la nube es la entrega de productos de tecnología a través de internet permitiendo a las empresas enfocarse en su negocio sin convertirse en negocios de tecnología.

Modelos de Cloud Computing

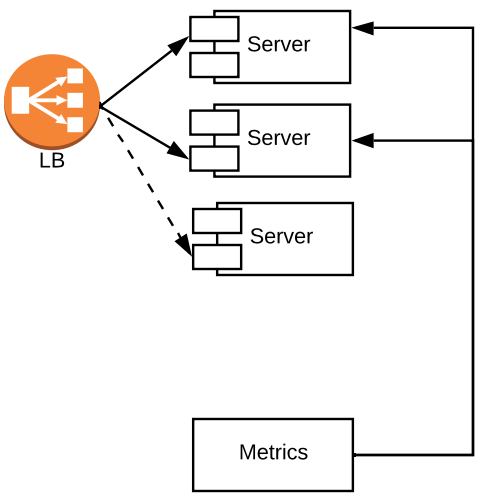
Los modelos de cloud computing permiten al usuario elegir el control que tendrá sobre la información y se tienen:

- **Infraestructure as a service (IaaS)**: El proveedor proveerá infraestructura como servidores, almacenamiento y redes.
- **Platform as a service**: Este modelo provee una plataforma que permite a los usuarios enfocarse en el desarrollo de una aplicación, algunas plataformas que se pueden encontrar en este modelo son: bases de datos, hosting, kafka clusters, etc.
- **Software as a service**:Este modelo permite a los usuarios utilizar una aplicación en específico sin tener que desplegar o mantenerla.
- **Container as a service**: Un modelo intermediario entre IaaS y PaaS, Con CaaS puedes desplegar microservicios de una forma ligera, a través de contenedores virtualmente portables.
- **Functions as a service**:También conocido como Serverless architecture, esto significa que permite ejecutar piezas de código sin necesidad de mantener ningún tipo de servidor, permite enfocarte en el desarrollo sin necesidad de pensar en escalamiento, administración o configuración, normalmente se cobra por el número de veces que se ejecuta esa pieza de código.

Cloud computing y microservicios

Uno de los conceptos principales de microservicios es que cada servicio se debe construir y desplegar de forma independiente con su propio artefacto, tarde o temprano tendrás que tomar una decisión sobre su despliegue en **servidores físicos**, **máquinas virtuales** o **contenedores virtuales**.

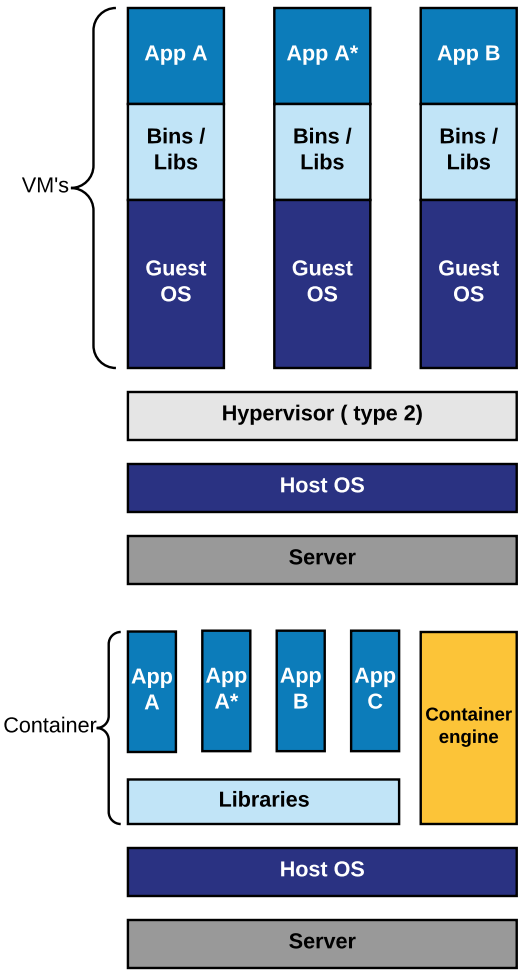
Autoscaling



Microservices

Los microservicios buscan descomponer aplicaciones monolíticas grandes en piezas más pequeñas, con bajo acoplamiento, distribuidas y fáciles de manejar.

VM vs Containers



Opciones de Despliegue

Servidor físico

- Limitados en escalabilidad y capacidad
Costosos

Máquinas virtuales (Nube):

- Ofrecidas por muchos proveedores
- Una máquina virtual por microservicio
- Fáciles de levantar y eliminar
- Virtualizan el hardware

Contenedores Virutales (Nube):

- Ventajas similares a una máquina virtual
- Más ligeros
- Varios contenedores por SO (virtualizan los SO)

Dimensionamiento

La definición correcta de un microservicio permite hacer cambios a una aplicación de forma simple y reduce el riesgo de que la aplicación no este disponible.

Recordemos que debe ser fácil para un equipo pequeño administrarlo.

Administración de ubicaciones

Los microservicios se inician y destruyen todo el tiempo, por lo que deben ser simples de ubicar, iniciar y apagar.

Resiliencia

Es normal que un microservicio presente fallas, pero debe ser resistente a las mismas o en caso de que no se pueda recuperar, debe fallar rápido para poder redirigir el tráfico a servicios denominados failover services.

Replicable

Se debe asegurar que cada instancia tenga la misma versión del código así como la misma configuración para que todos se conformen del mismo modo y no tengamos instancias que responden con una lógica de negocio y otras que respondan con una completamente diferente.

Escalable

El acoplamiento en la comunicación de los servicios debe ser mínimo con el fin de crecer o decrecer el número de instancias de forma simple y rápida.

Patrones de despliegue

Una de las partes clave de una arquitectura basada en microservicios es que cada instancia de un microservicio debe ser identica a todas las demás. Para conseguirlo debemos dejar de desplegar artefactos como jar / war a un servidor y en su lugar debemos compilar y construir una imágen que represente tu pieza de software, para entender como hacerlo debemos entender los siguientes temas:

- **Construcción y despliegue de pipelines**: Como crear un proceso de construcción y despliegue repetible a cualquier entorno de la organización.
- **Infraestructura como código**: Como proveer servicios con base a una pieza de código donde se incluya no solo la aplicación sino la definición del ambiente de ejecución.
- **Servidores inmutables**: Una vez que una imagen de un microservicio se ha creado esta no debe ser modificada despúes del despliegue.
- **Phoenix servers**: Se busca asegurar que los servidores se eliminen de forma regular y se vuelvan a crear a partir de una imágen inmutable con el fin de que se cambien configuraciones y la versión que se encuentra desplegada no cambie después de mucho tiempo.

Spring framework 5 - HTTP

HTTP

HTTP (*Hyper Text Transfer Protocol*) es un protocolo de la capa de aplicación que define un conjunto de reglas para transferir archivos, texto, imágenes, sonidos, videos, etc.

Métodos HTTP

El protocolo HTTP define múltiples métodos listados a continuación:

-**GET** : Es utilizado para obtener información del servidor

-HEAD: Es igual que GET pero solo devuelve el *status* y los *headers*

-**POST**: Es utilizado para enviar información al servidor

-**PUT**: Reemplaza el contenido del recurso con el nuevo

-**PATCH**: Aplica modificaciones parciales al recurso especificado

-**DELETE**: Borra la información del recurso especificado

-TRACE: Es utilizado para *debugging* enviando un *echo* de vuelta al usuario

-**OPTIONS**: Describe las opciones de comunicación con el servidor

-CONNECT: Establece un tunel con el servidor basado en la URL

URI

URI (*Uniform Resource Identifier*) Es el nombre que se utiliza para identificar un recurso de forma única. Contiene 3 partes:

-**Schema**: Protocolo a utilizar para acceder al recurso

-**Host**: Ubicación del servidor (DNS, hostname o IP)

-**Path**: Ruta al recurso en el servidor

Petición HTTP

Una petición HTTP se conforma de lo siguiente:

-**Método HTTP**: Método HTTP, ejemplo GET

-**URI**:URI del recurso solicitado,ejemplo: http://www.twitter.com/search?q=devs4j

-**Versión**: Versión del protocolo utilizado, ejemplo 1.1

-**Headers**: Información adicional de la petición, ejemplo Accept:text/html.

-**Body**: Cuerpo de la petición HTTP

Respuesta HTTP

Una respuesta HTTP se conforma de lo siguiente:

-**Versión**: Versión del protocolo utilizado, ejemplo 1.1

-**Status code**: Estatus de la respuesta

-**Headers**: Información adicional de la respuesta, ejemplo Transfer-Encoding:chunked.

-**Body**: Cuerpo de la respuesta HTTP

Connectionless

HTTP es un protocolo al que se le conoce como *connectionless*. Esto significa que no se debe mantener una conexión viva hacia el servidor, de este modo el cliente y el servidor se conocen solo durante la petición.

Stateless

Al ser un protocolo *connectionless*, también es considerado como *stateless*, esto significa que cada petición es independiente a la anterior. Dado este comportamiento, ni el cliente ni el servidor retienen información durante diferentes peticiones.

Status HTTP

Un *status* HTTP es un número que representa el resultado de una petición. Se clasifica en rangos:

-**1xx (100-199)**: Informacional
-**2xx (200-299)**: Petición exitosa
-**3xx (300-399)**: Redirecciones
-**4xx (400-499)**: Error del lado del cliente
-**5xx (500-599)**: Error del lado del servidor

REST

REST (*Representational State Transfer*) Es un estilo de arquitectura que define un conjunto de reglas para comunicar aplicaciones, se basa en el protocolo HTTP.

Resource

La principal abstracción de información en REST es un recurso. Cualquier información que puede ser nombrada puede ser un recurso

Definición de recursos

Una mala práctica común al definir recursos es utilizar verbos en su definición. A continuación un ejemplo:

-**Método http**: GET
-**Recurso**: /getUsers
-**Descripción**: Devuelve una lista de usuarios

Esto es una mala práctica por los siguiente:

-Es redundante decir que el recurso es **/getUsers** y se utilizará el método HTTP GET

-El recurso **/getUsers** debería soportar otros métodos HTTP, al hacerlo se volvería contradictorio dado que el *endpoint* sería POST /getUsers

Peticiones exitosas

A continuación se presentan los *status* HTTP que se utilizan para representar **peticiones exitosas** de acuerdo a cada método HTTP:

-**200 OK**: La petición fue exitosa. Se puede aplicar a métodos como GET, HEAD, PUT y TRACE

-**201 CREATED**: La petición fue exitosa y se creó un nuevo recurso. Se genera en métodos POST.

-**204 NOT CONTENT**: No hay contenido que devolver. Se utiliza como resultado de un método DELETE.

Errores del lado del cliente

Algunos errores comunes del lado del cliente:

-**400 BAD REQUEST**: Error genérico que indica que existe un error del lado del cliente

-**401 UNAUTHORIZED**: Cuando el cliente no tiene suficientes privilegios para acceder a un recurso

-**404 NOT FOUND**: El recurso solicitado no existe

-**405 METHOD NOT ALLOWED**: El método HTTP no esta soportado en el recurso solicitado

-**429 TO MANY REQUESTS**: Demasiadas peticiones. Se utiliza cuando se tiene una cantidad límite de peticiones por cliente.

-**409 CONFLICT**: Existe un conflicto

DEVS4J
Hecho por Devs para Devs

Errores del lado del servidor

Algunos errores del lado del servidor:

-**500 INTERNAL SERVER ERROR**: Error genérico que significa que existe un error del lado del servidor

-**501 NOT IMPLEMENTED**: Indica que el servidor no soporta la funcionalidad solicidada

-**502 BAD GATEWAY**: Error que indica que el servidor recibió un error al invocar a otro servidor

-**503 SERVICE UNAVAILABLE**: El servidor no puede procesar la petición.

Versionamiento de API

Existen múltiples formas de versionar un API y existen discusiones activas sobre cual es la mejor opción:

-**Versionamiento en la URL**: La versión del API se define en la URL, ejemplo:

HTTP GET: /application/api/v1/users

-**Versionamiento en los headers**: La versión del API se define en los *headers*:

HTTP GET: /application/api/users
Accept: application/raidentrance.api.v2+json

-**Versionamiento en el dominio**: La versión del API se define en el dominio, ejemplo:

apiv1.devs4j.com/users

-**Versionamiento en un *request param***: La versión del API se define en un *request param*, ejemplo:

HTTP GET:
/application/api/users?version=1

Paginación, filtros y ordenamiento

Cuando se desarrollan servicios web REST, una buena práctica es proveer paginación, filtros y ordenamiento a través de ***query params***, a continuación se muestran algunos ejemplos:

Filtros

-GET /cars?color=red : Devuelve una lista de autos color rojo

-GET /cars?seats<=2 : Devuelve una lista de autos con 2 asientos o menos

Ordenamiento

-GET /cars?sort=-color&+model : Devuelve una lista de autos ordenada de forma descendente por color y ascendente por modelo

Paginación

-GET /cars?page=1&size=10 : Devuelve la página 1 de autos de un tamaño de 10

Spring framework 5 - REST

Configuración

Para habilitar el soporte para REST se debe incluir el módulo Spring MVC a través de la siguiente dependencia:

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

Spring provee una serie de herramientas de programación llamadas **DevTools**, para utilizarlas se debe incluir la siguiente dependencia:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-devtools</artifactId>
	<scope>runtime</scope>
	<optional>true</optional>
</dependency>
```

Al ejecutar la aplicación, se mantendrá en ejecución hasta que se detenga de forma explícita.

Spring MVC

Spring MVC es un módulo que soporta el patrón de diseño modelo vista controlador:

-**Model**: Representa un **POJO** de **Java** que transporta los datos

-**View**: Representa la visualización de los datos que contiene el *modelo*

-**Controller**: Controla el flujo de la información entre el modelo y la vista, mantiene el modelo y la vista separados.

Anotaciones

A continuación se presentan las anotaciones importantes al trabajar con **Spring MVC**:

-**@Controller**: **Stereotype** que define a una clase como un controlador de **Spring**

-**@RestController**: Meta anotación que representa a(**@Controller** y **@ResponseBody**)

-**@RequestMapping**: Se puede aplicar a nivel de clase y método y permite mapear peticiones **HTTP** a métodos llamados **handler methods**.

-**@GetMapping**: Es una anotación compuesta que actúa como **@RequestMapping**(method=RequestMethod.GET)

-**@PostMapping**: Es una anotación compuesta que actúa como **@RequestMapping**(method=RequestMethod.POST)

-**@PutMapping**: Es una anotación compuesta que actúa como **@RequestMapping**(method=RequestMethod.PUT)

-**@DeleteMapping**: Es una anotación compuesta que actúa como **@RequestMapping**(method=RequestMethod.DELETE)

-**@PatchMapping**: Es una anotación compuesta que actúa como **@RequestMapping**(method=RequestMethod.PATCH)

-**@PathVariable**: Indica que un parámetro de un método debe tomar el valor de alguno de la URL, es conocido como **path param**.

-**@RequestParam**: Indica que un parámetro de un método debe tomar el valor de un **request parameter**, es conocido como **query param**

-**@ResponseStatus**: Se aplica en métodos y excepciones y define el status HTTP que se devolverá

-**@Service**: Es utilizado por los **controllers** y contiene la lógica de negocio de la aplicación

@Controller, @RestController y @RequestMapping

A continuación se muestra un ejemplo sobre el uso de **@Controller** y **@RestController** :

```
@RestController
@RequestMapping("/roles")
public class RoleController {
	//handler methods
}
```

@RequestMapping define que las peticiones sobre la URL /roles serán interceptadas por los **handler methods** definidos en la clase. En este ejmplo se puede reemplazar **@RestController** por **@Controller** y tendrá la misma funcionalidad.

@RequestMapping en métodos

@RequestMapping se puede aplicar tanto en clases como en métodos. Veamos el siguiente ejemplo:

```
@RestController
@RequestMapping("/roles")
public class RoleController {
	@Autowired
	private RoleService roleService;


	@RequestMapping(value = "/{id}", method =
RequestMethod.GET)
	public ResponseEntity<List<Role>> getById(.....) {
		return new ResponseEntity<>(roleService.findAll(),
	HttpStatus.OK);
	}
}
```

En el ejemplo anterior, el método **getAll** atenderá las peticiones hacia la **URL** "/roles/{id}" sobre el método **HTTP** GET.

@PathVariable

En el ejemplo anterior, se explicó cómo generar una URL con base en múltiples **@RequestMapping**. Ahora se mostrará como añadir **@PathVariable**:

```
@RestController
@RequestMapping("/roles")
public class RoleController {
	@Autowired
	private RoleService roleService;


	@RequestMapping(value = "/{id}", method =
RequestMethod.GET)
	public ResponseEntity<Role>
getById(@PathVariable("id") int id) {
		return new
ResponseEntity<>(roleService.findById(id),
	HttpStatus.OK);
	}
}
```

En el ejemplo anterior, el método **getAll** atenderá las peticiones hacia la URL "/roles/{id}" sobre el método HTTP GET, una petición de ejemplo sería http://localhost:8080/roles/1.

En este ejemplo el valor 1 sería el valor de la **@PathVariable** "{id}" y se asignará en el parámetro **id** del método getById gracias a la anotación **@PathVariable**.

@GetMapping

Es posible simplificar el **endpoint** anterior a través de @GetMapping como se muestra a continuación:

```
@GetMapping(value = "/{id}")
public ResponseEntity<Role> getById(@PathVariable("id") int
id) {
	return new ResponseEntity<>(roleService.findById(id),
HttpStatus.OK);
}
```

Las anotaciones **@PostMapping**, **@PutMapping**, **@PatchMapping**, etc. funcionan exactamente igual.

@RequestParam

@RequestParam nos permite obtener **query params** desde la URL. Veamos un ejemplo:

```
@GetMapping
public ResponseEntity<List<Role>> getAll
(@RequestParam("page") int page,
	@RequestParam("size") int size) {

	log.info("page {} size {} ", page, size);
	return new ResponseEntity<>
	(roleService.findAll(), HttpStatus.OK);
}
```

En el ejemplo anterior el método **getAll** atenderá las peticiones hacia la URL "/roles" sobre el método HTTP GET. Una petición de ejemplo sería http://localhost:8080/roles?page=1&size=10.

@ResponseStatus

@ResponseStatus permite aplicar a métodos y excepciones para definir el **status http** a devolver. A continuación se muestra un ejemplo aplicado a una excepción:

```
@ResponseStatus(code=HttpStatus.NOT_FOUND,
reason="Resource not found")
public class ResourceNotFoundException extends
RuntimeException{

 private static final long serialVersionUID =
8668589127062335507L;

}
```

En caso de que se arroje una ResourceNotFoundException se devolverá un status HTTP 404.

Otras clases útiles

Algunas otras clases útiles al trabajar con REST son:

-**ResponseEntity**: Permite devolver un contenido, **status** y **headers** HTTP.

-**HttpStatus**: Enumeración que contiene los **status** HTTP y su descripción.

-**ResponseStatusException**: En caso de que no se desee crear una excepción propia, es posible arrojar una ResponseStatusException para definir el status HTTP, ejemplo:

```
throw new
ResponseStatusException(HttpStatus.BAD_REQUEST,
"Resource not found");
```

Configuraciones útiles

A continuación se presentan algunas configuraciones útiles al trabajar con Spring MVC:

```
server.tomcat.threads.max=2
server.port=8081
server.servlet.context-path=/rest
```

Patrones de microservicios

Routing patterns

Los "**routing patterns**" definen como una aplicación que quiere consumir un microservicio descubre donde se encuentra.

En aplicaciones empresariales es posible tener cientos de instancias de microservicios en ejecución.

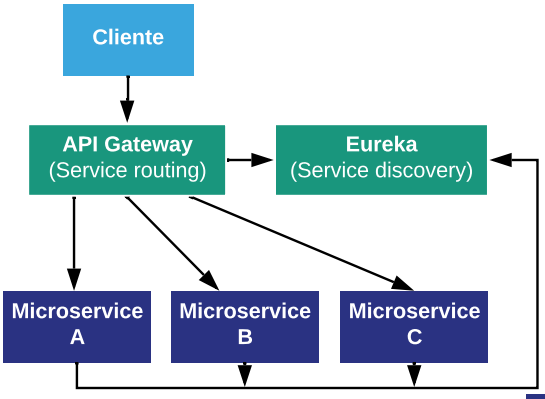
Dado lo anterior se recomienda tener un solo punto de acceso el cual debe definir tanto el enrutamiento, descubrimiento y seguridad de los servicios.

Service discovery / Service registry

Service discovery y service registry son dos características que van relacionadas dado que el **registro** permite mantener un **inventario** de los servicios así como su ubicación física y el **descubrimiento** permite a los clientes encontrar dichos servicios.

Service routing

Con un API Gateway puedes proveer un solo punto de acceso a tu aplicación de tal modo que la seguridad y las políticas de enrutamiento se apliquen de una forma uniforme a todos los microservicios.



Resiliencia

Dado que las arquitecturas basadas en microservicios son distribuidas, tienes que ser muy cuidadoso en como prevenir un problema en un servicio pequeño que pueda propagarse en todos los demás, para hacerlo existen 4 patrones:

- **Client side load balancing**: Mantiene en cache la ubicación de las instancias de los microservicios (Ip's) del lado del **cliente**, evitando tener un solo balanceador de carga y permitiendo tener diferentes estrategias de balanceo.
- **Circuit braker pattern**: Previene que un cliente invoque de forma continua a un servicio que está fallando o que tenga un problema de performance.
- **Fallback pattern**: Si un servicio falla permite definir un mecanismo en el que se proporcione una nueva alternativa.
- **Bulkhead pattern**: Los microservicios dependen unos de otros, este patron define como segregas el comportamiento erroneo de modo que no se propague e impacte de forma negativa a otros microservicios.

Patrones de seguridad

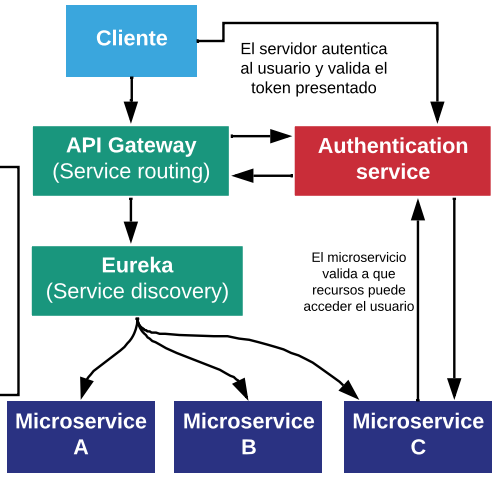
Para asegurar que todos los microservicios no están abiertos al publico, es importante aplicar los siguientes patrones de seguridad a la arquitectura para asegurar que solo peticiones autorizadas pueden invocar los servicios, a continuación los patrones:

- **Authentication**: Asegura que el cliente es quien dice ser.
- **Authorization**: Define si el cliente está intentando acceder a un servicio al que tiene acceso.
- **Credential management & Propagation**: Define el uso de segurad basada en tokens como OAuth2 y JWT (Json web tokens) para obtener un token que puede ser enviado a un servicio para autenticar y autorizar a un usuario.

OAuth 2

OAuth 2 es un framework de seguridad basado en **tokens** que permite a un usuario autenticarse con un servicio de autenticación externo, en caso de conseguir una autenticación exitosa obtendrá un token que se debe enviar en cada petición.

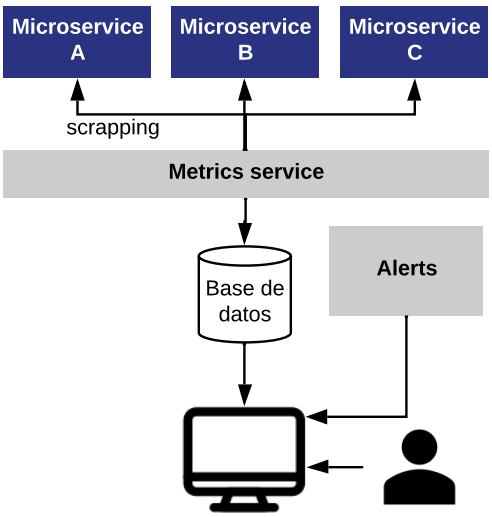
El principal objetivo es que las peticiones sean autenticadas sin exponer las credenciales.



Logging & Tracing

Al tener una arquitectura distribuida es muy complicado debugear, rastrear y monitorear las peticiones por lo que existen los siguientes patrones:

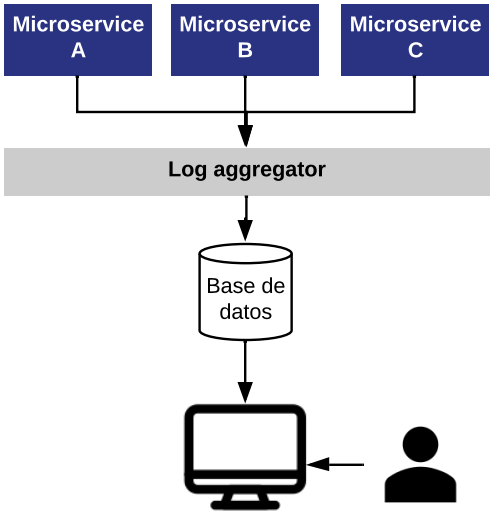
- **Log correlation**: Una petición puede resultar en **múltiples invocaciones** de múltiples microservicios, pero ¿Cómo saber en que punto se presentó una falla?. Con este patrón generaremos un **identificador único** que se propagará a todas las llamadas utilizadas.
- **Log aggregation**: Permite obtener todos los logs producidos por un microservicio con el fin de entender su fincionamiento, performance y características.
- **Microservice tracing**: Permitirá visualizar el flujo de las transacciones a través de todos los microservicios.



Patrones de métricas

Las patrones de métricas permiten definir como una aplicación va a ser monitoreada y como se generarán alertas ante fallas, este patrón define 3 diferentes componentes:

- **Metrics**: Como tu aplicación definirá sus métricas tanto de sistema como de aplicación.
- **Metrics service**: Define el lugar donde se almacenarán dichas métricas.
- **Metrics visualization suite**: Define el lugar donde podrás visualizar las métricas tanto de sistema como de aplicación.



DEVS4J
Hecho por Devs para Devs

Spring cloud / Config server

Spring cloud

Implementar todos los patrones de microservicios desde cero mencionados anteriormente sería algo bastante complejo y requeriría muchísimo trabajo.

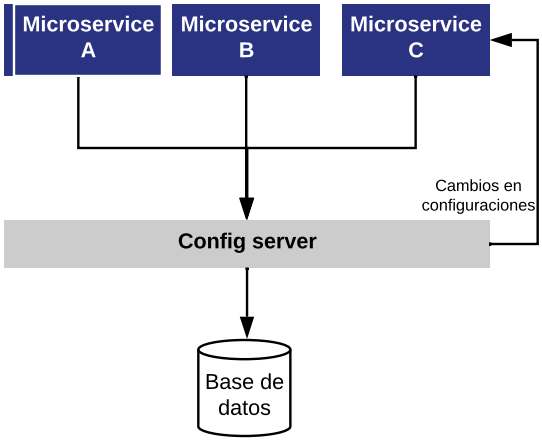
Spring cloud es una colección de herramientas robustas que simplifican la construcción de nuestras aplicaciones y dan una solución común.

Spring cloud config

Administra las **configuraciones** de las aplicaciones y **ambientes** a través de un servicio centralizado, esto asegura que sin importar el número de microservicios todos tengan la misma configuración.

La separación de la configuración permite cambiar el comportamiento de una aplicación sin necesidad de recompilar y re desplegar las aplicaciones.

Muchos desarrolladores almacenan las configuraciones en archivos de propiedades (YAML, JSON o XML's), administrar esos archivos de vuelve algo complejo



Configuración

Para construir un config server incluiremos las siguientes dependencias:

- Spring boot DevTools
- Spring boot Actuator
- Config Server

Una vez incluidas las dependencias cambiaremos el puerto al 8888 como se muestra a continuación:

application.yml

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/raidentrance/devs4j-config
          default-label: main
          clone-on-start: true
server:
  port: 8888
```

bootstrap.yml es un archivo especial de **Spring cloud** y se carga antes del archivo application.properties.

Cambios en el código

Una vez que se incluye el repositorio en el bootstrap.yml, el siguiente paso es habilitar el config server como se muestra a continuación:

```
@SpringBootApplication
@EnableConfigServer
public class Devs4jConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run
    (Devs4jConfigServerApplication.class, args);
    }
}
```

Creación de las configuraciones

En el repositorio de Github agregaremos los siguientes archivos de configuración:

devs4j-dragon-ball.properties

```
application.name = Devs4j configuration default
```

devs4j-dragon-ball-dev.properties

```
application.name = Devs4j configuration default DEV
```

devs4j-dragon-ball-prod.properties

```
application.name = Devs4j configuration default PROD
```

Configuración del lado del cliente

Para leer las configuraciones del config server debes incluir las siguientes entradas en el archivo **application.yml**:

```yaml
spring:
  application:
      name: devs4j-dragon-ball
  profiles:
      active: prod
  config:
      import:optional:configserver:
            http://localhost:8888
server:
  port: 8082
```

Uso de las propiedades

Una vez hecha la configuración del lado del cliente, el siguiente paso es utilizarlas en la aplicación como se muestra a continuación:

```
@Component
@ConfigurationProperties
public class DragonBallConfig {

    @Value("${application.name}")
    private String applicationName;

    public String getApplicationName() {
        return applicationName;
    }

    public void setApplicationName(String
applicationName) {
        this.applicationName =
        applicationName;
    }

}
```

Uso de las propiedades

Ya configurado el bean, puedes inyectarlo en cualquier lugar de la aplicación como se muestra a continuación:

```
@RequestMapping
("/api/v1/dragonball/greetings")
@RestController
public class GrettingsController {

    @Autowired
    private DragonBallConfig config;

    @GetMapping
    public ResponseEntity<String>
    getCharacters() {
        return new
        ResponseEntity<String>
        (String.format("Hello from %s",
        config.getApplicationName()),
        HttpStatus.OK);
    }
}
```

Actualizar propiedades

Puedes actualizar las propiedades agregando la anotación **@RefreshScope**, la cual habilita un endpoint llamado **POST /actuator/refresh,** a continuación un ejemplo:

```
@Component
@ConfigurationProperties
@RefreshScope
public class DragonBallConfig {}
```

NOTA: Debe estar habilitado **actuator** para que el endpoint que actualiza las configuraciones funcione, para hacerlo asegurate de tener la siguiente dependencia:

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

Una vez incluida la dependencia debes habilitar los management endpoints agregando la siguiente entrada en el archivo **application.yml**:

```yaml
management:
  endpoints:
    web:
      exposure:
        include: *
```

Service discovery

Service discovery

En un sistema distribuido es necesario conocer en que IP o DNS se encuentra un servicio, este concepto es conocido como "**service discovery**" y es necesario por las siguientes razones:

Escalamiento horizontal:Agregar nuevas instancias o removerlas requiere su registro para su uso.

Resiliency: Se refiere a absorber el impacto de los problemas dentro de una arquitectura o servicio sin afectar al negocio. Si una instancia tiene un funcionamiento incorrecto, se remueve de inmediato para reducir el impacto.

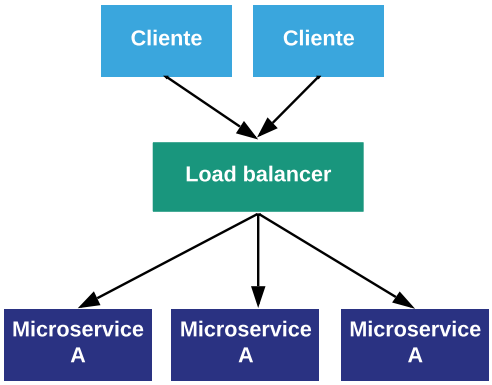
Balanceo de carga tradicional

En un sistema tradicional, se cuenta con un balanceador de carga que recibe peticiones de los clientes y basado en una tabla de rutas lo distribuye a uno o más servicios. Este esquema no es recomendado para aplicaciones basadas en microservicios por lo siuguiente:

- El balanceador de carga es un **punto único de falla**
- Los balanceadores **no** estan diseñados para realizar **registros / des registros rápidos** y simples.
- Es **difícil escalar** los balanceadores de carga (licencias y modelo de redundancia hot-swap)
- Todos los clientes siguen la **misma estategia de balanceo** de carga.

  Por otro lado los balanceadores de carga tradicionales siguen jugando un rol importante en la centralización de **SSL**

Balanceo de carga tradicional



Service discovery

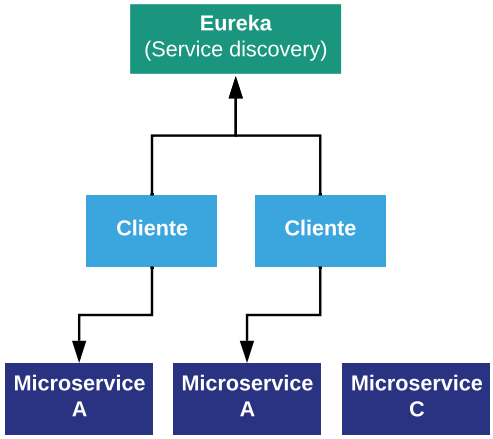
Un proyecto basado en microservicios puede utilizar service discovery para conseguir:

- **Alta disponibilidad**: Puede darse cuenta si un nodo no esta disponible para que otro pueda tomar su lugar de una forma muy rápida.
- **Uno a uno(Peer to peer)**: Cada nodo en el service discovery conoce el estado de una instancia.
- **Balanceo de carga**: Cada cliente tiene la lista de nodos en el cluster y puede realizar su propia estategia de balanceo.

Service discovery

- **Reciliencia**: El cliente puede poner la información de los nodos en cache de tal modo que si el servidor del service discovery no esta disponible las aplicaciones pueden seguir funcionando. El cliente deberá mantener actualizada su cache para actualizar el estado de salud de los nodos.
- **Tolerancia a fallos**: Service discovery necesita detectar cuando un servicio no está saludable pata removerlo de la lista de servicios y evitar causar un impacto(La detección no debe tener intervención humana).

Uso de service discovery



Eureka server

Para configurar un servidor de Eureka crearemos un proyecto nuevo de Spring boot que contenga las siguientes dependencias:

- **Eureka server**
- **Spring boot DevTools**
- **Spring actuator**

Al seleccionarlo se incluirán las siguientes dependencias:

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

Opcionalmente puedes incluir las dependencias de config server para externalizar la configuración.

Configuración

Una vez incluidas las dependencias el siguiente paso es incluir las siguientes entradas en el archivo application.properties:

```
spring.application.name=devs4j-registry
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

A continuación la explicación de cada una de ellas:

spring.application.name - Define el nombre de la aplicación que se está iniciando

server.port - Define el puerto en el que se iniciará el servidor (Iniciaremos varios servidores por lo que es importante cambiarlo)

eureka.client.register-with-eureka - Define si se registrará a si mismo en el registry

eureka.client.fetch-registry - Define que no se guardará en cache las direcciones, en caso de ser utilizada se actualiza cada 30 segundos.

Una vez incluidas las configuraciones anteriores el siguiente paso es habilitar el servidor de Eureka, para hacerlo agregaremos la anotación @EnableEurekaServer a nuestra clase aplicación:

```
@SpringBootApplication
@EnableEurekaServer
public class Devs4jServiceRegistryApplication {
      public static void main(String[] args) {
      SpringApplication.run
      (Devs4jServiceRegistryApplication.class, args);
      }
}
```

El paso final será construir nuestro proyecto e iniciar nuestra aplicación como cualquier otra aplicación creada con Spring boot y consultaremos en nuestro navegador la dirección **http://localhost:8761/**

UI

La interfaz gráfica mostrará las siguientes secciones:

- **System status** : Status del servidor de Eureka
- **DS replicas** : Replicas configuradas
- **Instancias registradas**: Muestra los servicios registrados en el service discovery
- **General info**: Información general sobre el servidor como memoria disponible, número de cpus, tiempo de funcionamiento, etc.
- **Información de la instancia**: Información util como la ip que se está utilizando y el estado del servidor

Service discovery

Registro de microservicios con Eureka

Una vez que nuestro servidor de Eureka está arriba funcionando, el siguiente paso es registrar nuestros microservicios en el, para hacerlo incluiremos la siguiente dependencia:

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

Una vez incluida la dependencia el siguiente paso es activar el registro, para hacerlo incluiremos la siguiente clase de configuración:

```
@EnableDiscoveryClient
@Configuration
public class DiscoveryConfiguration {
}
```

Por último agregaremos un nombre a nuestra aplicación en nuestro archivo **application**.properties:

```
spring.application.name=devs4j-registry
```

NOTA

Recuerda cambiar el puerto de la aplicación cliente para que no genere ningún conflicto para hacerlo agregar lo siguiente al archivo application.properties:

```
server.port=8761
```

Iniciando la aplicación

Una vez configurado el siguiente paso será iniciar nuestra aplicación, al hacerlo veremos los siguientes logs en la salida:

```
[36mo.s.c.n.e.s.EurekaServiceRegistry
Registering application
DEVS4J-DRAGON-BALL with eureka with
status UP
```

Además veremos que en la dirección **http://localhost:8761/** nuestra aplicación registrada en la sección "***Instances currently registered with Eureka***".

Eureka API

Puedes obtener información de las apis registradas utilizando el siguiente endpoint **http://localhost:8761/eureka/apps/** , la respuesta es por default en xml pero puedes agregar un header "**Accept : application/json**" para recibirla en formato JSON.

Utilizando service discovery en el cliente

Existen diferentes formas de consumir los microservicios, en el ejemplo del curso utilizaremos lo siguiente:

- **Spring discovery client**: Permitirá conectarnos y consumir información de Eureka
- **Netflix Feign client**: Permitirá definir que servicios consumiremos y el tipo de respuesta que recibiremos.

Utilizando service discovery en el cliente

Para que una aplicación consuma los microservicios utilizando el discovery client agregaremos las siguientes dependencias:

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

Una vez incluidas las dependencias habilitaremos los clientes de Feign como se muestra a continuación:

```
@SpringBootApplication
@EnableFeignClients
public class Devs4jClientApplication
implements ApplicationRunner {}
```

El siguiente paso será agregar las siguientes entradas en el archivo **application.properties**:

```
spring.application.name=devs4j-client
eureka.client.register-with-eureka=false
```

Eureka client

Puedes inyectar un EurekaClient para obtener información del servidor como se muestra a continuación:

```
@Autowired
private EurekaClient eurekaClient;

    private static final Logger log =
    LoggerFactory.getLogger
    (Devs4jClientApplication.class);


    @Override
    public void run(ApplicationArguments
args) throws Exception {
        Application application =
eurekaClient.getApplication
        ("devs4j-dragon-ball");

        log.info("Application
        {}",application.getName());

        List<InstanceInfo> instances =
application.getInstances();

        for (InstanceInfo instanceInfo :
instances) {
            log.info("Instance info {}",
instanceInfo.getIPAddr());
        }
```

Clientes Feign

A continuación se muestra un ejemplo de un Cliente Feign:

```
@FeignClient(name = "devs4j-dragon-ball")
public interface
DragonBallCharacterClient {

    @RequestMapping(method =
RequestMethod.GET, value =
"/api/v1/dragonball/characters")
    ResponseEntity<List<String>>
    getDragonBallCharacters();

}
```

Utilizando el cliente

Una vez creado el cliente el siguiente paso es utilizarlo, para hacerlo simplemente lo inyectaremos donde se necesite como se muestra a continuación:

```
@SpringBootApplication
@EnableFeignClients
public class Devs4jClientApplication
implements ApplicationRunner {

    @Autowired
    private DragonBallCharacterClient
dragonBallClient;

    private static final Logger log =
LoggerFactory.getLogger
(Devs4jClientApplication.class);

    @Override
    public void run(ApplicationArguments
args) throws Exception {
        log.info("Dragon ball
characters");
        ResponseEntity<List<String>>
characters =
dragonBallClient.getDragonBallCharacters();

    Objects.requireNonNull
    (characters.getBody())
    .stream().forEach(ch -> log.info(ch));
}

public static void main(String[] args) {
        SpringApplication.run
(Devs4jClientApplication.class, args);
    }
}
```

Client side load balancing

Para incluir balanceo de carga del lado del cliente incluiremos la siguiente anotación en nuestros clientes Feign:

```
@LoadBalancerClient(name =
"devs4j-dragon-ball", configuration =
LoadBalancerConfiguration.class)



@Configuration
public class LoadBalancerConfiguration {
    private static final Logger log =
    LoggerFactory.getLogger
    (LoadBalancerConfiguration.class);

    @Bean
    public ServiceInstanceListSupplier
discoveryClientServiceInstanceListSupplier(
            ConfigurableApplicationContext
context) {
        log.info("Configuring Load
balancer to prefer same instance");
        return ServiceInstanceListSupplier
        .builder()
        .withBlockingDiscoveryClient()
        .build(context);
    }
}
```

Spirng gateway + Resiliencia

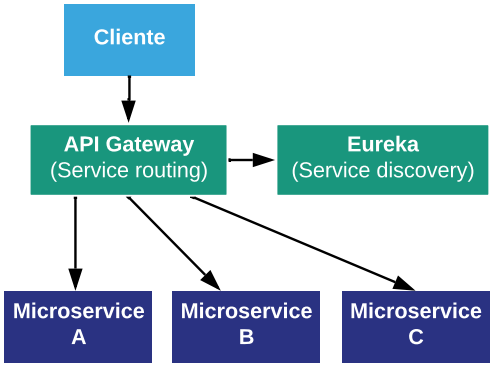
DEVS4J
Hecho por Devs para Devs

Gateway

Con un API Gateway puedes proveer un solo punto de acceso a tu aplicación de tal modo que la seguridad y las políticas de enrutamiento se apliquen de una forma uniforme a múltiples servicios en tus microservicios.

Así como en Spring boot se utiliza AOP para implementar cross cutting concerns, en un gateway aplicaremos lo mismo pero a nivel de microservicios, a continuación se muestran cross-cutting concerns que normalmente se implementan en un gateway:

- **Enrutamiento estático**: Permite registrar todos los servicios detrás de una url, es simple dado que solo se debe conocer el endpoint y nadamás.
- **Enrutamiento dinámico**: Se registrarán una serie de servicios y dependiendo de los datos que se encuentran en la petición realizará un enrutamiento inteligente.
- **Autenticación y autorización**: Como todas las peticiones pasarán por el service gateway es el lugar natural para realizar las validaciones de seguridad.
- **Colección de métricas y logs**: Como todas las peticiones pasarán por el service gateway es el lugar natural para analizar métricas y logs y evitar que el comportamiento se deba duplicar en todos los microservicios



Configuración

Para construir un gateway incluiremos las siguientes dependencias:

```
<dependency>
    <groupId>
    org.springframework.cloud
    </groupId>
    <artifactId>
    spring-cloud-starter-gateway
    </artifactId>
</dependency>
<dependency>
    <groupId>
    org.springframework.cloud
    </groupId>
    <artifactId>
spring-cloud-starter-netflix-eureka-client
    </artifactId>
</dependency>
<dependency>
    <groupId>
    org.springframework.cloud
    </groupId>
    <artifactId>
spring-cloud-starter-circuitbreaker
-reactor-resilience4j
    </artifactId>
</dependency>
```

Dependencias

A continuación se explican las dependencias incluidas:

- **spring-cloud-starter-gateway** :Dependencia de Spring cloud para configurar un gateway simple
- **spring-cloud-starter-netflix -eureka-client**: Si se utiliza Eureka como service discovery, nuestro gateway será un cliente más por lo que se deberá registrar
- **spring-cloud-starter-circuitbreaker-reactor-resilience4j**: Más adelante se aplicarán patrones de resiliencia por lo que es necesario incluir la dependencia de resilience4j.

Configuración del gateway

Una vez incluidas las dependencias el siguiente paso es configurar nuestro gateway a continuación se muestra un ejemplo:

Configuración en el archivo **application.properties**:

```
spring.application.name=devs4j-gateway
server.port=9090
```

Configuración en el gateway:

```
@Configuration
public class GatewayConfig {

    @Bean
    @Profile("localhostRouter-noEureka")
    public RouteLocator
configLocalNoEureka(RouteLocatorBuilder
builder) {
        return builder.routes()
.route(r->r.path("/api/v1/dragonball/*")
.uri("http://localhost:8080"))
.route(r->r.path("/api/v1/gameofthrones/*")
.uri("http://localhost:8081"))
.build();
    }
}
```

La configuración anterior indica lo siguiente:

- El tráfico que ingrese en el path "/api/v1/dragonball/*" se redireccionará con el mismo path a la url "http://localhost:8080"
- El tráfico que ingrese en el path "/api/v1/gameofthrones/*" se redireccionará con el mismo path a la url "http://localhost:8081"

Esta configuración es funcional pero tiene un pequeño problema, no utiliza **Eureka** por lo que si se registran nuevas instancias a alguno de los microservicios no serán incluidas en este enrutamiento.

Resiliencia

Todos los sistemas distribuidos experimentan **fallos**, es responsabilidad de los desarrolladores preparar a las aplicaciones para responder a ellos.

Los patrones de reciliencia se enfocan en proteger al cliente de los fallos en un recurso remoto (Por errores o un performance bajo).

Estos patrones permiten al cliente fallar rápido y no consumir los recursos para prevenir que el problema siga empeorando o se siga propagando.

Configuración del gateway con Eureka

A continuación veremos el mismo ejemplo pero ahora utilizando **Eureka**:

Configuración en el archivo **application.properties**:

```
spring.application.name=devs4j-gateway
server.port=9090
eureka.client.register-with-eureka=false
```

Configuración en el gateway:

```
@Configuration
public class GatewayConfig {

    @Bean
    @Profile("localhostRouter-eureka")
    public RouteLocator
configLocalNoCB(RouteLocatorBuilder
builder) {
        return builder.routes()
.route(r ->r.path("/api/v1/dragonball/*")
.uri("lb://dragon-ball"))
.route(r->r.path("/api/v1/gameofthrones/*")
.uri("lb://game-of-thrones"))
.build();   }
}
```

Client side load balancing

Todos los sistemas distribuidos experimentan **fallos**, es responsabilidad de los desarrolladores preparar a las aplicaciones para responder a ellos.

Los patrones de reciliencia se enfocan en proteger al cliente de los fallos en un recurso remoto (Por errores o un performance bajo).

Estos patrones permiten al cliente fallar rápido y no consumir los recursos para prevenir que el problema siga empeorando o se siga propagando.

Circuit breaker

Cuando se invoca a un microservicio circuit braker monitorea las peticiones, si la petición toma mucho, circuit braker aparece y mata la llamada.

El objetivo de esto es que la falla sea rápida y prevenga efectos colaterales o se malgasten recursos.

Fallback

Una vez que circuit braker detecto que un servicio del que se tiene dependencia falla en lugar de propagar un error fallback ejecuta otra alternativa (Puede ser un servicio de failover).

Los servicios de failover normalmente son servicios que apuntan a otras fuentes de datos o replicas.

Distribucción de logs

Sleuth

Sleuth es una herramienta poderosa para mejorar los logs de cualquier aplicación.

Diagnosticar un problema de una aplicación que funciona en modo miltithread es muy complicado y la mayoría de las veces se resuelve pasando un identificador único a cada método para identificar los logs, lo cual no es una solución muy limpia.

Sleuth permite identificar los logs que pertenecen a un job, hilo o petición sin mucho esfuerzo con frameworks como Logback y SLF4j.

Configuración

Para trabajar con Sleuth incluiremos la siguiente dependencia:

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

A demás de la dependencia debemos asegurar que se tiene configurado el nombre de nuestra aplicación en el archivo **application.properties (yml)**:

```
spring:
  application:
      name: devs4j-dragon-ball
```

Terminología

A continuación se presenta una terminología general de sleuth:

- **Span**: Es la unidad base de trabajo, que contiene información como:
  - Descripción
  - Timestamp
  - Tags
  - Span id
  - Process ID (IP's)
- **Trace**: Un conjunto de spans forma una estructura

Comportamiento default

Spring Sleuth permite definir comportamientos personalizados pero puede ser utilizado en su funcionalidad por defecto, a continuación el ciclo de vida de un Span:

- **start**: Cuando inicias un span su nombre es asignado y el timestamp es asignado
- **end**: El momento en el que span finaliza
- **continue**:El span continua, por ejemplo en otro Thread.
- **create explicit parent**: Permite crear un span nuevo asignando un parent explícito.

Descarga Zipkin

Puedes descargar Zipkin en el siguiente enlace:

https://zipkin.io/pages/quickstart.html

Una vez descargado ejecutarás el siguiente comando:

```
java -jar .\zipkin-server-2.23.7-exec.jar
```

Creando y terminando spans

Puedes crear manualmente spans utilizando la interfaz Tracer, como se muestra en el siguiente ejemplo:

```
@Autowired
private Tracer tracer;


void foo() {
    Span newSpan =
tracer.nextSpan().name("newSpan");
    try (Tracer.SpanInScope ws =
this.tracer.withSpan(newSpan.start())) {
        // You can log an event on a span
        log.info("Log 3");
    }finally {
        // Termina un span
        newSpan.end();
    }
}
```

Habilitar @Async

Para probar el funcionamiento del logging en múltiples Threads, el primer paso será configurar un Threadpool como se muestra a continuación:

```
@Configuration
@EnableAsync
public class ThreadConfig extends
AsyncConfigurerSupport {

    @Autowired
    private BeanFactory beanFactory;

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor
threadPoolTaskExecutor = new
ThreadPoolTaskExecutor();
threadPoolTaskExecutor.setCorePoolSize(1);
threadPoolTaskExecutor.setMaxPoolSize(1);
threadPoolTaskExecutor.initialize();
        return new
LazyTraceExecutor(beanFactory,
threadPoolTaskExecutor);
    }
}
```

Habilitar @Async

A continuación se muestra el método asíncrono:

```
@Async
public void fooAsync() {
    log.info("Async start");
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    log.info("Async end");
}
```

Una vez hecho anterior ejecutaremos el método y notaremos que los logs que se encuentran en el método asíncrono se generarán con el mismo traceId pero con un nuevo span id.

Lo anterior es posible dado que utilizamos LazyTraceExecutor en caso de no utilizarlo se creará un nuevo traceId y un nuevo span id.

Abrir Zipkin

Una vez descargado e iniciado puedes abrir Zipkin utilizando la siguiente URL:

http://127.0.0.1:9411/

DEVS4J
Hecho por Devs para Devs

Sin LazyThreadExecutor

Modifiquemos el Executor que definimos para entender la diferencia:

```
@Configuration
@EnableAsync
public class ThreadConfig extends
AsyncConfigurerSupport {

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor
threadPoolTaskExecutor = new
ThreadPoolTaskExecutor();
threadPoolTaskExecutor.setCorePoolSize(1);
threadPoolTaskExecutor.setMaxPoolSize(1);
threadPoolTaskExecutor.initialize();
        return threadPoolTaskExecutor;
    }
}
```

Al realizar el cambio anterior notaremos que se generó tanto un traceId como un spanId nuevo.

Configuración de Zipkin

Para trabajar con Zipkin incluiremos la siguiente dependencia:

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

A demás de la dependencia incluiremos la siguiente entrada en nuextro archivo **application.yml**:

```
spring:
  zipkin:
      baseUrl: http://localhost:9411
```

Para concluir ejecutaremos varias veces nuestros endpoints y abriremos **Zipkin** en nuestro navegador

Monitoreo y alertas

Configuración

Todas las clases necesarias para trabajar con micrometer las tendremos al incluir la dependencia:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
     <version>1.7.1</version>
</dependency>
```

Si se trabajará con Spring Framework + Prometheus como es en el caso de este curso se puede incluir solo la dependencia:

```
<dependency>
      <groupId>io.micrometer</groupId>
      <artifactId>micrometer-registry-prometheus
      </artifactId>
      <scope>runtime</scope>
</dependency>

<plugin>
    <artifactId>maven-compiler-plugin
    </artifactId>
    <version>3.7.0</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

@Timed

Puedes anotar un método con **@Timed** para medir el tiempo de ejecución de un método definido en un **@Controller**. Al agregar la anotación se generarán 2 series con los siguientes nombres:

-`${name}_count`- Número total de todas las peticiones
-`${name}_sum`- Tiempo total de todas las peticiones

Puedes calcular el tiempo promedio con el siguiente query de Prometheus:

```
rate(timer_sum[10s])/rate(timer_count[10s])
```

Puedes calcular el **throughput**(Peticiones por segundo) como se muestra a continuación:

```
rate(timer_count[10s])
```

NOTA: No es posible utilizar @Timed fuera del contexto web en un método regular.

Long task timer

Los long task timers son un tipo especial de timers que permiten medir el tiempo de un evento que sigue en ejecución. Un timer regular no calcula la duración hasta que el proceso se completa.

En una aplicación de Spring, es común ejecutar procesos largos con la anotación **@Scheduled**, puedes aplicar un long task timer como se muestra a continuación:

```
@Timed(value = "long_time", longTask =
true)
@Scheduled(fixedDelay = 360000)
void longTimeProcess() {

}
```

Puede ser utilizado para notificar si el tiempo de ejecución supera algún threshold definido.

Puedes calcular la duración con :

```
longTaskTimer{statistic="duration"}
```

Prometheus

Las méticas se pueden consultar siguiendo la estructura de prometheus, para esto consultaremos utilizando la siguiente url:

/actuator/prometheus

Hecho lo anterior veremos una salida como la siguiente:

```
# HELP get_characters_seconds   # TYPE
get_characters_seconds summary
get_characters_seconds_count
{exception="None",method="GET",
status="200",uri="/characters",} 1.0
```

Descarga de prometheus

Puedes descargar prometheus del siguiente enlace:

https://prometheus.io/download/

Una vez descargado, lo debes descomprimir entrar a la carpeta y ejecutar:

```
./prometheus --config.file=prometheus.yml
```

Una vez iniciado puedes acceder a la siguiente URL:

http://localhost:9090/graph

Modificaremos el archivo prometheus.yml para que utilice las métricas que estamos generando en nuestra aplicación como se muestra a continuación:

```
scrape_configs:
  - job_name: 'prometheus'
    metrics_path: /actuator/prometheus

    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8080']
```

Una vez hecho lo anterior entra de nuevo a la Url y verás las métricas de tu aplicación

Descarga grafana

Puedes descargar grafana en :

https://grafana.com/grafana/download

Una vez descargado entra al folder en la carpeta de bin y ejecuta:

```
./grafana-server
```

Y abre la URL http://localhost:3000/

Selecciona en el side bar:

- Configuración
- Datasources
- Add datasource
- Selecciona Prometheus
    -Name = Prometheus
    -Url = http://localhost:9000

Creación de un dashboard

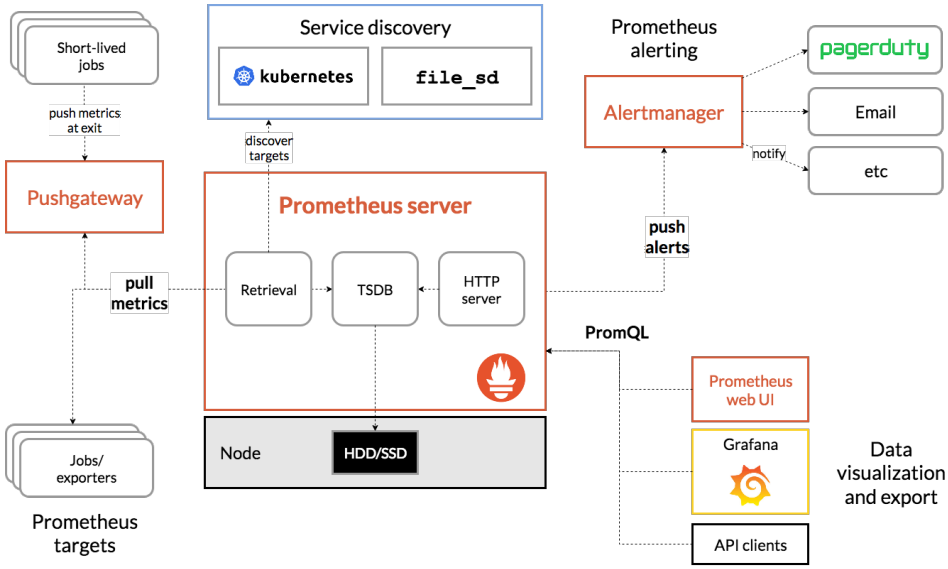
Para crear un dashboard de grafana selecciona el símbolo + de la barra de la izquierda y sigue los siguientes pasos:

- Crea un panel

- En la sección de Metrics, selecciona Prometheus y coloca el query que deseas utilizar, en el ejemplo utilizaremos :

rate(get_characters_seconds_count[5m])

- En el panel de la derecha puedes definir el título y el tipo de gráfica que deseas representar



DEVS4J
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Docker

Conceptos básicos

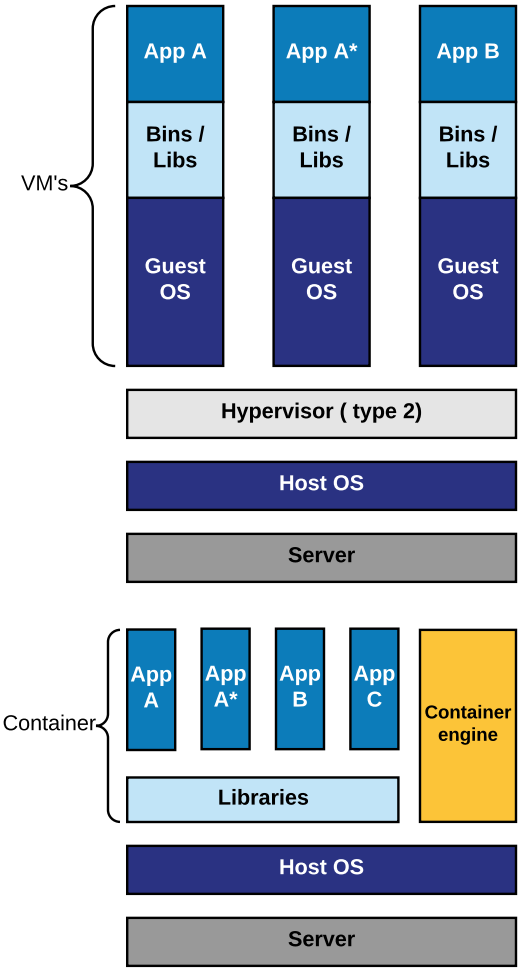
La idea detrás de Docker es empaquetar una aplicación con todas sus dependencias en una sola y estandarizada unidad de despliegue.

Docker envuelve todo en un completo sistema de archivos que contiene todo lo que tu aplicación Java necesita para ejecutarse una máquina virtual, un servidor de aplicaciones, el código empaquetado y todas las bibliotecas en tiempo de ejecución.

Empacar todo en una imágen garantiza que es portable.

Con Docker puedes ejecutar aplicaciones Java sin tener que instalar Java en el servidor destino, en caso de querer hacer algún tipo de limpieza solo debes destruir la imágen y construirla de nuevo y no pasa nada.

Virtualization vs containerization



Beneficios

-Muy buen **performance**
-Tiempo de creación corto
-Uso compartido de recursos como OS Kernel, bibliotecas, etc.
-Elimina conflictos de bibliotecas de software
-Tiene la misma promesa de Java, escribelo una vez y ejecutalo en cualquier lugar
-Los desarrolladores pueden ejecutar las mismas imagenes de producción en sus ambientes locales.

www.twitter.com/devs4j

Docker

Docker es un engine de contenedores open source basado en Linux que cuenta con los siguientes componentes:

- **Docker daemon** - Permite crear y administrar imágenes de Docker.
- **Docker client** - Los usuarios interactúan con Docker a través de un cliente cuya responsabilidad es enviar instrucciones a el daemon.
- **Docker registry** - Un registry es la ubicación donde las imágenes de Docker se almacenan. Estos registros pueden ser públicos o privados. Docker Hub es el lugar por defecto para los registros públicos.
- **Docker images** - Son templates de solo lecura con múltiples instrucciones que indican como crear un contenedor de Docker. Las imágenes se pueden descargar de un registry, puedes crear imágenes utilizando **Dockerfiles**.
- **Docker containers** - Un container es una instancia de una imágen.
- **Docker volumes** - Un volumen es el mecanismo para almacenar información generada por Docker en un contenedor.
- **Docker networks** - Las networks permiten adjuntar a los contenedores las redes que se requieran. Podemos ver las networks como las formas de comunicación con un contenedor aislado.

Dockerfile

Un Dockerfile es un **archivo de texto** plano que contiene una colección ordenada de **cambios a un sistema** con sus correspondientes **parámetros** para el uso dentro de un contenedor en ejecución.

Cada instrucción crea una nueva **layer** (capa) en la imagen.

Un Dockerfile es utilizado para crear una imágen.

Una vez que el Dockerfile es creado, es posible ejecutar **docker build** para construir una imágen. Una vez que todo está listo puedes utilizar **docker run** para crear los contenedores.

Comandos en el Dockerfile

A continuación se presentan algunos comandos útiles al crear un archivo Dockerfile:

- **FROM** - Define la imágen base para iniciar el proceso de construcción.
- **LABEL** - Agrega metadatos (llave-valor) a una imágen.
- **ARG** - Define **variables** que el usuario puede pasar a la construcción utilizando el comando **docker build**.
- **COPY** - Copia nuevos archivos, directorios de una fuente y los agrega a un sistema de archivos de la imágen, por ejemplo COPY ${JAR_FILE} app.jar
- **VOLUME** - Crea un punto de montaje en el contenedor. Cuando se crean múltiples contenedores utilizando la misma imágen se creará un nuevo volumen cada vez aislado de los anteriores.
- **RUN** - Toma el comando y sus argumentos para ejecutar un contenedor desde una imágen, usualmente se utiliza para instalar paquetes de software dentro del contenedor.

www.facebook.com/devs4j

Comandos en el Dockerfile

- **CMD** - Provee argumentos al ENTRYPOINT. Este comando es similar a docker run, pero este se ejecuta después de que el contenedor es creado.
- **ADD** - Copia y agrega los archivos de una fuente a un destino dentro del contenedor.
- **ENTRYPOINT** - Configura un contenedor que funcionará como un ejecutable.
- **ENV** - Asigna las variables de entorno.

Construcción de una imágen

Para iniciar vamos a construir una imagen de Docker agregando un pluging de Maven a nuestro archivo pom.xml. Este plugin permitirá administrar las imágenes de Docker y contenedores desde nuestro archivo pom.xml. a continuación un ejemplo:

```
<properties>
    <java.version>11</java.version>
    <docker.image.prefix>devs4j
    </docker.image.prefix>
</properties>

<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin
    </artifactId>
    <version>1.4.13</version>
    <configuration>
        <repository> ${docker.image.prefix}/
${project.artifactId}
    </repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>
target/${project.build.finalName}.jar
</JAR_FILE>
    </buildArgs>
  </configuration>
  <executions>
    <execution>
      <id>default</id>
      <phase>install</phase>
      <goals>
        <goal>build</goal>
        <goal>push</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

El plugin anterior define lo siguiente:

- Asigna el nombre del repositorio remoto, en este caso se utilizaron las variables **docker.image.prefix** y **project.artifactId**.
- Asigna una etiqueta al repositorio utilizando la versión del proyecto

Docker File

A continuación se muestra un archivo de ejemplo de un Dockerfile:

```
#Imágen base a utilizar
FROM openjdk:11-slim

#Información del equipo de contacto
LABEL maintainer="contacto@devs4j.com"

#Archivo Jar de la aplicación(Será asignado por el #pluging de Maven)
ARG JAR_FILE

#Agregar el Jar al contenedor
COPY ${JAR_FILE} /app.jar

#Ejecución de la aplicación
ENTRYPOINT ["java","-jar","/app.jar"]
```

Ejecución

Puedes construir tu imágen de Docker utilizando:

```
mvn package dockerfile:build
```

Puedes ejecutar tu imágen de Docker utilizando:

```
docker run -p 8080:8080
devs4j/devs4j-docker-example:0.0.1-SNAPSHOT
```

Spring framework 5 - Springdoc + Cache

Configuración

Para configurar Springdoc agregaremos las siguiente dependencia a nuestra aplicación:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.6.9</version>
</dependency>
```

Al hacerlo podremos acceder a nuestra documentación en formato json :

```
http://localhost:8080/v3/api-docs
```

Para ver la ui utilizaremos:

```
http://localhost:8080/swagger-ui.html
```

Descripciones propias

Puedes definir una descripción y un tipo de dato de retorno a través de las anotaciones:

```
@Operation(summary = "Get a list of characters from GoT")
    @ApiResponses(value = {
      @ApiResponse(responseCode = "200", description =
"Found the characters",
        content = { @Content(mediaType =
"application/json", array
=@ArraySchema(schema=@Schema(implementation =
String.class))) }),
      @ApiResponse(responseCode = "404", description =
"Characters not found",
        content = @Content) })
```

Exclusión de controllers

Si quieres excluir algunos controllers de la documentación puedes agregar la siguiente entrada al application.properties:

```
springdoc.packagesToScan=
com.devs4j.got.controller
```

También puedes hacerlo a través de los paths como se muestra a continuación:

```
springdoc.pathsToMatch=/v1, /api/balance/**
```

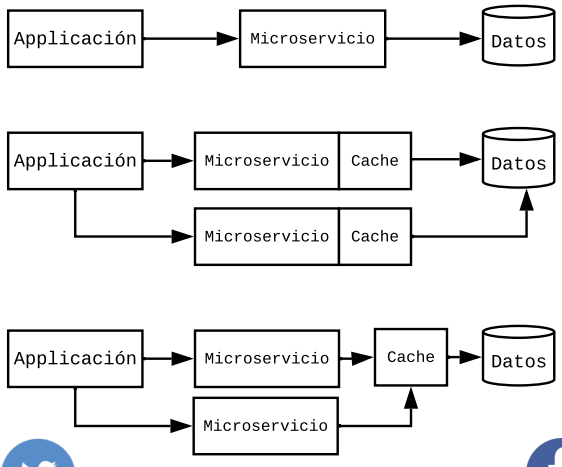
Puedes consultar las configuraciones disponibles en:

```
https://springdoc.org/#properties
```

Configuración de cache

Para configurar Spring cache debes agregar la siguiente dependencia:

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



Configuración de los beans

Una vez agregada la dependencia se debe configurar del siguiente modo:

```
@EnableCaching
@Configuration
public class CacheConfig {

 @Bean
 public CacheManager getManager() {
      return new ConcurrentMapCacheManager("codes");
 }
}
```

Uso de cache

Una vez configurado se deben agregar los objetos al cache, para esto utilizaremos la siguiente anotación como se muestra a continuación:

```
@Cacheable("codes")
 public List<String> getCp() {
      ./Code
}
```

Remover valores con @CacheEvict

Es posible remover valores de un cache a través de la anotación @CacheEvict:

```
@CacheEvict("users")
 public User getUserById(IntegeruserId){
  .....
}
```

Cache en memoria

Cuando se utiza **ConcurrentMapCacheManager** el cache se realiza en la máquina que está ejecutando la aplicación.

Si se colocarán en cache muchos datos esto puede causar problemas en la aplicación.

Uso de un sistema de cache externo

Para el caso de que se deseen tener muchos datos en cache, es posible almacenarlos en un sistema de cache externo, en este ejemplo se utilizará redis, puedes descargarlo en la siguiente dirección:

https://redis.io/download

Utilizando Docker

docker pull redis

Iniciar redis

Para iniciar redis debes seguir los siguientes pasos:

```
- Descarga redis
- Descomprimelo
- Ejecuta el comando make
- Ejecuta el comando src/redis-server

Utilizando Docker

docker run -d -p 6379:6379 --name devs4j-redis
redis
```

Configuración de redison

Para configurar la integración con redis agregar la siguiente dependencia:

```
<dependency>
 <groupId>org.redisson</groupId>
 <artifactId>redisson</artifactId>
 <version>3.11.5</version>
</dependency>
```

Agregar el bean de redison al contexto

Agregaremos al contexto de spirng :

```
 @Bean(destroyMethod ="shutdown")
publicRedissonClient redisson() {
      Configconfig=newConfig();
      config.useSingleServer().
      setAddress("redis://127.0.0.1:6379");
 returnRedisson.create(config);
}
```

Modificaremos el Cache Manager

El siguiente paso es indicar a spring que utilizará redis en lugar del ConcurrentMapCache, como se muestra a acontinuación:

```
@Bean
 publicCacheManager
cacheManager(RedissonClientredissonClient)
{
  Map<String,
CacheConfig>config=newHashMap<>();
  config.put("testMap",newCacheConfig());
  returnnewRedissonSpringCacheManager
(redissonClient);
 }
```

Nota importante

Los objetos que se escriban en el cache deben implementar la interfaz Serializable

Revisar información en Redis

Para consultar la información almacenada en redis puedes utilizar el cliente que provee, ejecutando el comando en la carpeta raíz de redis:

-> src/redis-cli

Comandos útiles:

DEL key - Borra una llave

EXISTS key - Determina si la llave existe

HGETALL key - Obtiene el valor de un map

Utilizando docker

docker exec -it devs4j-redis sh

redis-cli

Spring cloudstream

Producer - Supplier

Kafka topic - in

Procesor - Function

Kafka topic - out

Consumer - Consumer

Descarga kafka

Para descargar apache kafka debes acceder a la siguiente url :

https://kafka.apache.org/downloads

Asegurate de descargar la versión que dice binary downloads.

Inicia kafka

Para iniciar tu servidor de kafka deberás ejecutar los siguientes comandos:

```
$ bin/zookeeper-server-start.sh
config/zookeeper.properties

$ bin/kafka-server-start.sh
config/server.properties
```

Esto iniciará tanto zookeeper como kafka.

Creando un topic

Los mensajes se procesan en topics, para crear uno deberás ejecutar el siguiente comando:

```
$ bin/kafka-topics.sh --bootstrap-server
localhost:9092 --create --topic devs4j-topic
--partitions 5 --replication-factor 1
```

Este comando recibe los siguientes parámetros:

- `bootstrap-server` = Kafka server
- `topic` = Nombre del topic a crear
- `partitions` = Número de particiones
- `replication-factor` = Número de réplicas por broker

Listando topics

Puedes listar los topics disponibles ejecutando:

```
$ bin/kafka-topics.sh --list
--bootstrap-server localhost:9092
```

Salida de ejemplo:

```
devs4j-topic
```

Ver definición de un topic

Si deseas consultar como se definió un topic puedes describirlo con el siguiente comando:

```
$ bin/kafka-topics.sh --describe --topic
devs4j-topic --bootstrap-server
localhost:9092
```

Salida de ejemplo:

```
Topic: devs4j-topic       PartitionCount: 5  ReplicationFactor: 1

    Topic: devs4j-topic     Partition: 0  Leader: 0   Replicas: 0  Isr: 0
    Topic: devs4j-topic     Partition: 1  Leader: 0   Replicas: 0  Isr: 0
    Topic: devs4j-topic     Partition: 2  Leader: 0   Replicas: 0  Isr: 0
    Topic: devs4j-topic     Partition: 3  Leader: 0   Replicas: 0  Isr: 0
    Topic: devs4j-topic     Partition: 4  Leader: 0   Replicas: 0  Isr: 0
```

De la salida podemos observar lo siguiente:

- Tiene 5 particiones, que
- Solo se tiene una replica
- Solo hay un líder el cual es quien contiene el topic especificado.

Crear un producer

Para iniciar un producer ejecutaremos el siguiente comando:

```
$ bin/kafka-console-producer.sh --topic
devs4j-topic --bootstrap-server localhost:9092
```

Crear un consumer

Para iniciar un consumer ejecutaremos el siguiente comando:

```
$ bin/kafka-console-consumer.sh --topic
devs4j-topic --from-beginning --bootstrap-server
localhost:9092
```

El parámetro `--from-beginning` permite especificar si queremos recibir solo los mensajes nuevos o queremos leer todos desde el inicio.

Configuración

Crear proyecto que incluya los siguientes starter dependencies:

```
<dependency>
    <groupId>org.springframework.cloud
    </groupId>
    <artifactId>spring-cloud-stream
    </artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud
    </groupId>
    <artifactId>spring-cloud-stream-binder-kafka
    </artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka
    </groupId>
    <artifactId>spring-kafka
    </artifactId>
</dependency>
```

Functions

Para realizar transformaciones a los eventos utilizaremos funciones como se muestra a continuación:

```
@Bean
public Function<String, String> toUpperCase(){
    return data-> data.toUpperCase;
    //return String::toUpperCase;

}
```

Al crear el método anterior, se crearán 2 topic:

- toUpperCase-In
- toUpperCase-Out

Uno definirá la entrada de la función y el otro mostrará la salida de la misma.

Pruebas

Para probar el comportamiento de nuestra función utilizaremos un command line consumer / producer como se muestra a continuación:

```
kafka-console-producer.sh --broker-list
localhost:9092 --topic toUpperCase-in-0


kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic toUpperCase-out-0
```

Suppliers

Ahora es tiempo de crear nuestro propio producer, para hacerlo crearemos un supplier como se muestra a continuación:

```
@Bean
public Supplier <Flux<Long>> producer()
{
    return ()->Flux.interval
    (Duration.ofSeconds(1)).log();
}
```

Flux Es un componente de reactor que sirve para generar una secuencia de números.

Más información sobre el tema en https://spring.io/reactive donde podrás ver la información sobre reactor.

El supplier anterior publicará una cada segundo en el segundo en el que se encuentra.

Function

En este ejemplo elevaremos el número que recibimos al cuadrado:

```
@Bean
public Function<Flux<Long>,Flux<Long>>
processor(){
    return flx->flx.map(nbr->nbr*nbr);
}
```

Consumer

Para leer el producto de la función escribiremos un consumer como se muestra a continuación:

```
@Bean
public Consumer<Long> consumer(){
    return (number)-> log.info("Value
{}", number);
}
```

Definición de entradas y salidas

Para definir en que topics se publicarán los mensajes o de que topic se leerá se podrán utilizar las siguientes configuraciones en el archivo application.properties:

```
spring.cloud.stream:
    bindings:
        producer-out-0:
            destination: numbers
        processor-in-0:
            destination: numbers
        processor-out-0:
            destination: sqares
        consumer-in-0:
            destination: sqares
```

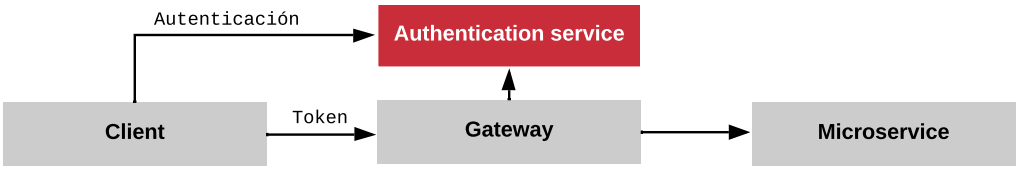
Producer simple

Puedes enviar mensajes de forma arbitraria como se muestra a continuación:

```
@Autowired
private StreamBridge streamBridge;

public void
delegateToSupplier(@RequestBody String
body) {    streamBridge.send("toStream-out-0",
body);
}
```

JSON Web Tokens



Configuración

Para configurar nuestro authentication service incluiremos las siguientes dependencias:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client
    </artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.4.4</version>
</dependency>

<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt</artifactId>
   <version>0.9.1</version>
</dependency>
```

Configuración de seguridad

Al ser el servicio de seguridad, todos deberían poder ejecutarlo, para incluir esas reglas crearemos la siguiente clase de configuración de seguridad:

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain
filterChain(HttpSecurity http)
    throws Exception {
        http.csrf().disable()
    .authorizeRequests().anyRequest()
    .permitAll();
        return http.build();
    }

}
```

Una vez que las reglas de seguridad fueron definidas, el siguiente paso es crear el bean para el cifrado:

```
@Configuration
public class PasswordEncoderConfig {
    @Bean
    public PasswordEncoder encoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Dto's

Crearemos los siguientes modelos:

```
public class TokenDto {

    private String token;

    public TokenDto(String token) {
        super();
        this.token = token;
    }
....
}
```

Dto's

```
public class UserDto {
    private String username;
    private String

    public UserDto() {
    }
    ....
}
```

Entities

A demás de los dto crearemos la siguiente entidad:

```
@Entity
public class UserEntity {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private int id;
    private String username;
    private String password;

    public UserEntity() {
    }

    public UserEntity(String username,
String password) {
        super();
        this.username = username;
        this.password = password;
    }

    public UserEntity(int id, String
username, String password) {
        super();
        this.id = id;
        this.username = username;
        this.password = password;
    }
...
}
```

Repository

Una vez creada la entidad crearemos el siguiente repositorio:

```
@Repository
public interface UserRepository extends
JpaRepository<UserEntity, Integer> {

    public Optional<UserEntity>
findByUsername(String username);

}
```

Application.yml

```
spring:
  zipkin:
      baseUrl: http://localhost:9411
  application:
      name: devs4j-auth
server:
  port: 8084
jwt:
  secret: secret
```

JwtProvider

EL JwtProvider será responsable de crear, validay y obtener información de un json web token como se muestra a continuación:

```
@Component
public class JwtProvider {

    @Value("${jwt.secret}")
    private String secret;

    @PostConstruct
    public void init() {
        secret = Base64.getEncoder()
        .encodeToString(secret.getBytes());
    }

    public String createToken(UserEntity user) {
        Map<String, Object> claims =
Jwts.claims().setSubject(user.getUsername());

        claims.put("id", user.getId());
        Date now = new Date();
        Date expiration = new Date(now.getTime()
+ 3600*1000);

        return Jwts.builder()
            .setClaims(claims)
            .setIssuedAt(now)
            .setExpiration(expiration)
            .signWith(SignatureAlgorithm.HS256,
            secret)
            .compact();
    }

    public boolean validate(String token) {
        try {
            Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public String getUsernameFromToken(String
token) {
        try {
            return Jwts.parser()
                .setSigningKey(secret)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
        } catch (Exception e) {
            return "bad token";
        }
    }
}
```

Configuración del mapper

Se debe incluir la siguiente dependencia para asegurar que model mapper sigue funcionando:

```
@Configuration
public class MappingConfig {

    @Bean
    public ModelMapper getMapper() {
        return new ModelMapper();
    }

}
```

JSON Web Tokens

AuthService

Se implementarán métodos para crear, validar y hacer login como se muestra a continuación:

```
@Service
public class AuthUserService {

    @Autowired
    private UserRepository repository;

    @Autowired
    private PasswordEncoder encoder;

    @Autowired
    private JwtProvider provider;

    @Autowired
    private ModelMapper mapper;

    public UserDto save(UserDto dto) {
        Optional<UserEntity> result =
repository.findByUsername(dto.getUsername());

        if (result.isPresent()) {
            throw new
ResponseStatusException(HttpStatus.CONFLICT,
                    String.format("User %s already
exists", dto.getUsername()));
        }

        UserEntity entity = repository.save(new
UserEntity(dto.getUsername(),
encoder.encode(dto.getPassword())));
        return mapper.map(entity, UserDto.class);
    }

    public TokenDto login(UserDto user) {
        Optional<UserEntity> result =
repository.findByUsername(user.getUsername());

        if (!result.isPresent()) {
            throw new
ResponseStatusException(HttpStatus.UNAUTHORIZED);
        }

        if (encoder.matches(user.getPassword(),
result.get().getPassword())) {
            return new
TokenDto(provider.createToken(result.get()));
        }
        throw new
ResponseStatusException(HttpStatus.UNAUTHORIZED);
    }

    public TokenDto validate(String token) {
        if (!provider.validate(token)) {
            throw new
ResponseStatusException(HttpStatus.UNAUTHORIZED);
        }
        String username =
provider.getUsernameFromToken(token);
        Optional<UserEntity> result =
repository.findByUsername(username);
        if (!result.isPresent()) {
            throw new
ResponseStatusException(HttpStatus.UNAUTHORIZED);
        }
        return new TokenDto(token);
    }
}
```

Controller

```
@RequestMapping("/auth/")
public class AuthController {

    @Autowired
    private AuthUserService service;

    @PostMapping("/login")
    public ResponseEntity<TokenDto>
login(@RequestBody UserDto dto) {
        TokenDto token = service.login(dto);
        return ResponseEntity.ok(token);
    }

    @PostMapping("/validate")
    public ResponseEntity<TokenDto>
validate(@RequestParam String token) {
        TokenDto tokenDto =
service.validate(token);
         return ResponseEntity.ok(tokenDto);
    }

    @PostMapping("/create")
    public ResponseEntity<UserDto>
create(@RequestBody UserDto dto) {
        UserDto userEntity = service.save(dto);
        return ResponseEntity.ok(userEntity);
    }
}
```

Probando AuthService

Una vez incluidos los cambios anteriores al auth service lo iniciaremos y tendremos disponibles los siguientes endpoints:

```
POST http://localhost:9090/auth/create
{
   "username":"devs4j",
   "password":"HolaMundo"
}

POST http://localhost:9090/auth/login
{
   "id": 1,
   "username": "devs4j",
   "password": "HolaMundo"
}

POST
http://localhost:9090/auth/validate?token={token}
```

Agregando Dto's al Gateway

Ahora agregaremos de igual modo el Dto del Token al proyecto del gateway:

```
public class TokenDto {

    private String token;

    public TokenDto() {
    }

    public TokenDto(String token) {
        super();
        this.token = token;
    }
...
}
```

AuthFilter

Agregaremos el siguiente filter a nuestro proyecto de spring gateway:

```
@Component
public class AuthFilter implements GatewayFilter
{

    @Autowired
    private WebClient.Builder webClient;

    public Mono<Void> onError(ServerWebExchange
exchange, HttpStatus status) {
        ServerHttpResponse response =
exchange.getResponse();
        response.setStatusCode(status);
        return response.setComplete();
    }

    @Override
    public Mono<Void> filter(ServerWebExchange
exchange, GatewayFilterChain chain) {
        if (!exchange.getRequest().getHeaders()
        .containsKey(HttpHeaders.AUTHORIZATION)) {
            return onError(exchange,
HttpStatus.BAD_REQUEST);
        }

        String tokenHeader =
exchange.getRequest().getHeaders()
        .get(HttpHeaders.AUTHORIZATION).get(0);
        String chunks[] = tokenHeader.split(" ");
        if (chunks.length != 2  ||
!chunks[0].equals("Bearer")) {
            return onError(exchange,
HttpStatus.BAD_REQUEST);
        }
        return webClient.build().post()
    .uri("http://devs4j-auth/auth/validate?token="
        + chunks[1]).retrieve()
        .bodyToMono(TokenDto.class)
        .map(t -> {
            return exchange;
        }).flatMap(chain::filter);
    }
}
```

DEVS4J
Hecho por Devs para Devs

Configuración del cliente

Crearemos un cliente para consumir los servicios del auth-service:

```
@Configuration
public class WebClientConfig {

    @Bean
    @LoadBalanced
    public WebClient.Builder
builder(){
        return WebClient.builder();
    }

}
```

Modificación de las rutas

Por último modificaremos las rutas como se muestra a continuación:

```
@Autowired
private AuthFilter filter;

@Bean
@Profile("localhostRouter")
public RouteLocator
configLocal(RouteLocatorBuilder
builder) {
        return builder.routes()
                .route(r ->
    r.path("/api/v1/dragonball/*")
    .filters(f->{
        f.circuitBreaker
        (c->c.setName("failoverCB")
    .setFallbackUri
    ("forward:/api/v1/db-failover/
    dragonball/characters")
    .setRouteId("dbFailover"));
    f.filter(filter);
    return f;
    })
    .uri("lb://devs4j-dragon-ball"))
    .route(r ->
r.path("/api/v1/gameofthrones/*")
    .filters(f->f.filter(filter))
    .uri("lb://game-of-thrones"))
    .route(
    r->r.path("/api/v1/db-failover
    /dragonball/characters")
    .uri("lb://failover"))
    .route(r->r.path("/auth/**")
    .uri("lb://devs4j-auth"))
    .build();
}
```

Pruebas

Probaremos ejecutando las siguientes apis:

POST http://localhost:9090/auth/create

```
{
   "username":"devs4j",
   "password":"HolaMundo"
}
```

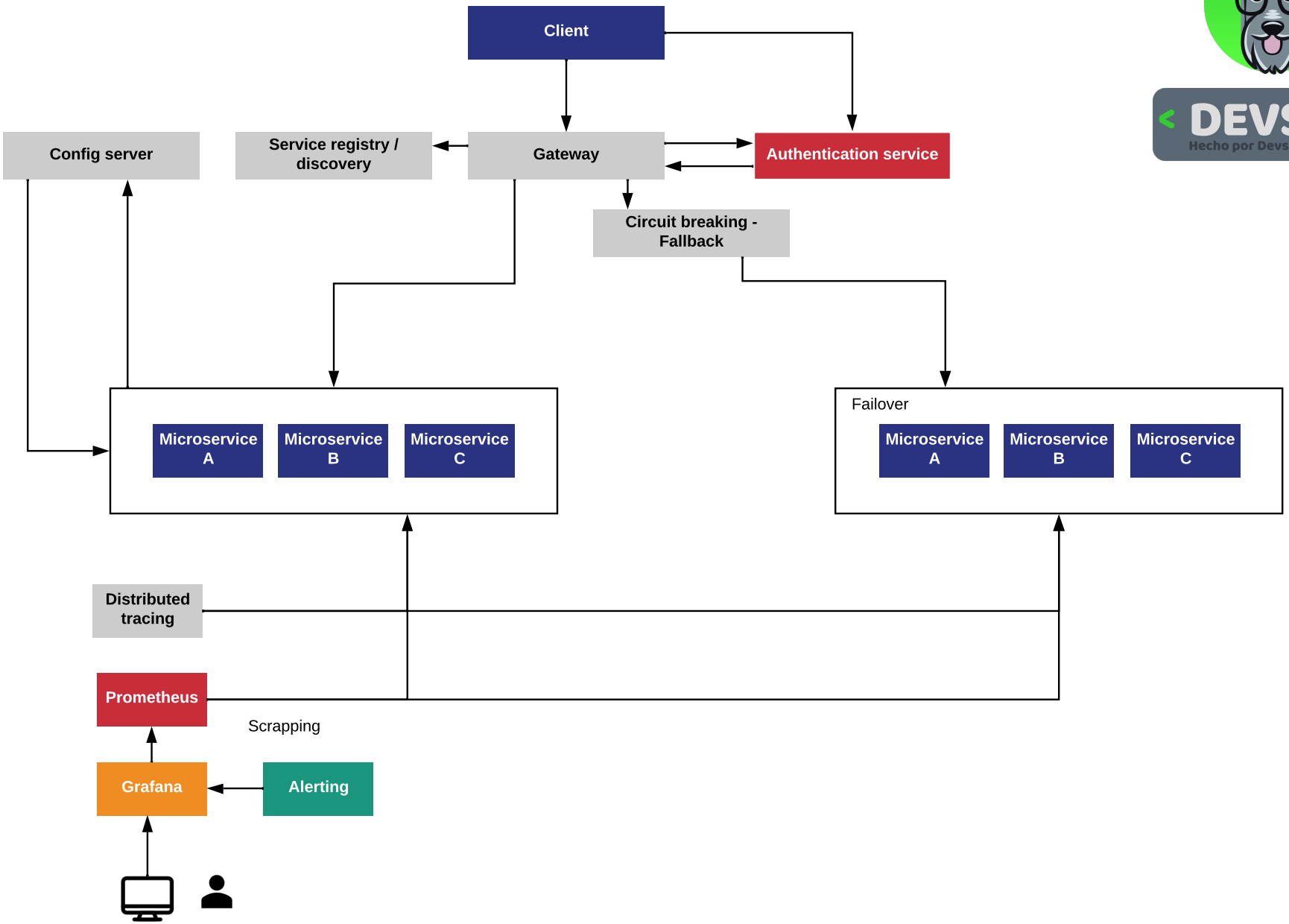
POST http://localhost:9090/auth/login

```
{
   "id": 1,
   "username": "devs4j",
   "password": "HolaMundo"
}
```

GET http://localhost:9090/api/v1/gameofthrones/characters

Bearer Token = **{token}**

Arquitectura final



Ejemplo práctico