API GATEWAY

Información general	
Duración estimada en minutos:	120
Docente:	Carlos Andrés Florez Villarraga
Guía no.	04

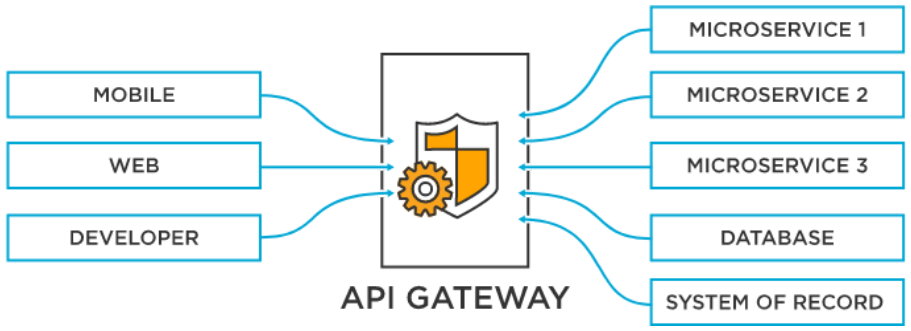
Información de la Guía

API Gateway es un servicio que se utiliza en la arquitectura de microservicios para gestionar, proteger y exponer las API de un conjunto de servicios web a través de una única interfaz.

En términos simples, API Gateway actúa como un intermediario entre el cliente y los servicios de backend. Los clientes envían solicitudes HTTP a través de API Gateway, que se encarga de enrutar la solicitud al servicio adecuado, realizar la autenticación y autorización necesarias, y entregar la respuesta al cliente.

API Gateway también puede proporcionar funciones adicionales, como la transformación de datos, el almacenamiento en caché, la limitación de la tasa, la monitorización y el registro de las solicitudes entrantes y salientes.

Finalmente es importante resaltar que API Gateway simplifica la gestión de las API al proporcionar una única interfaz para el cliente y los servicios de backend, al mismo tiempo que proporciona funciones de seguridad y control de acceso.



En Spring Boot, se puede usar Spring Cloud Gateway como API Gateway. Spring Cloud Gateway es un proyecto de Spring Cloud que proporciona una solución para implementar API Gateway en aplicaciones basadas en Spring Boot.

Para usar Spring Cloud Gateway se debe configurar las rutas y los filtros para enrutar y procesar las solicitudes entrantes a través de la API Gateway. Para hacerlo, se debe agregar las propiedades correspondientes en el archivo `application.properties` siguiendo la sintaxis:

```
spring.cloud.gateway.routes[<route-id>].uri=<uri>
spring.cloud.gateway.routes[<route-id>].predicates[<index>]=<predicate>
spring.cloud.gateway.routes[<route-id>].filters[<index>]=<filter>
```

Donde <route-id> es un identificador único para la ruta, <uri> es la URL de destino para enrutar las solicitudes, <predicate> es el predicado para la ruta (por ejemplo, Path=/api/**), y <filter> es el filtro para la ruta.

EJERCICIO

Tome evidencia de los siguientes puntos.

1. Cree un nuevo módulo con el nombre **`gateway`**, en su `build.gradle` agregue la siguiente configuración:

```
plugins {
   id 'java'
   id 'org.springframework.boot' version '3.0.2'
   id 'io.spring.dependency-management' version '1.1.0'
}

group = 'co.edu.eam'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
   mavenCentral()
   maven { url 'https://artifactory-oss.prod.netflix.net/artifactory/maven-oss-candidates' }
}

ext {
   set('springCloudVersion', "2022.0.1")
}

dependencies {
   implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
   implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
   imports {
       mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
   }
}

tasks.named('test') {
   useJUnitPlatform()
}
```

   En este archivo agregamos la configuración básica de Spring Boot y Spring Cloud junto con las dependencias de `netflix-eureka-client` y `spring-cloud-starter-gateway`.

2. Cree un paquete con un nombre acorde al proyecto que estamos construyendo (`co.edu.eam.biblioteca`), y dentro del paquete agregue una clase con el nombre: `GatewayServiceApplication` con la siguiente configuración:

```
@SpringBootApplication
public class GatewayServiceApplication {

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

```
    }
}
```

3. Cree, para este nuevo módulo, el archivo `application.properties` de la siguiente manera:

```
spring.application.name=gateway-service

server.port=8080
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

El servicio de API gateway debe tener un puerto fijo ya que por medio de él es que vamos a acceder a los demás microservicios, además, debe registrarse en Eureka (al igual que los demás servicios).

En ese mismo archivo agregue la siguiente configuración:

```
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.routes[0].id=cliente-service
spring.cloud.gateway.routes[0].uri=lb://cliente-service
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/cliente/**

spring.cloud.gateway.routes[1].id=libro-service
spring.cloud.gateway.routes[1].uri=lb://libro-service
spring.cloud.gateway.routes[1].predicates[0]=Path=/api/libro/**

spring.cloud.gateway.routes[2].id=prestamo-service
spring.cloud.gateway.routes[2].uri=lb://prestamo-service
spring.cloud.gateway.routes[2].predicates[0]=Path=/api/prestamo/**
```

Tal como se mencionó anteriormente, se debe definir la configuración de las rutas para que el API Gateway sepa cómo redireccionar las peticiones. En este caso estamos usando las rutas de los `RestController` de cada microservicio.

4. Modifique el `application.properties` del microservicio de préstamo para que no tenga un puerto fijo, ya no es necesario.

Esto se logra con la siguiente configuración (esta configuración es la que tenemos en el microservicio de libros y usuarios):

```
server.port=0
eureka.instance.instance-id=${spring.application.name}:${random.value}
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

5. Despliegue todos los microservicios, pero tenga en cuenta que primero se debe ejecutar el servicio de eureka y de último se debe ejecutar el servicio del gateway.

6. Desde Postman (o la aplicación que desee) haga una prueba para crear un libro, un usuario y un préstamo (en este caso verifique que se estén llamando correctamente los microservicios de clientes y libros). Dado que el servicio de gateway se ejecuta en el puerto 8080, entonces podemos acceder a todos los servicios desde dicho puerto.

7. Haga un archivo `docker-compose` que permita ejecutar todos los microservicios en el orden correcto para evitar tener que hacerlo de forma manual.

8. Investigue cómo se pueden ejecutar varias instancias de una misma imagen desde docker compose. Aplique lo investigado para ejecutar dos veces la imagen del microservicio del cliente. De esta forma podemos evidenciar el balance de carga automático que hace Spring Cloud.