

**FUNDACION UNIVERSITARIA INTERNACIONAL  
DE LA RIOJA COLOMBIA**

**ESPECIALIZACION EN INGENIERIA DE SOFTWARE  
(DESARROLLO DE APLICACIONES WEB)**

**ACTIVIDAD 1  
DESARROLLO DE UN BACK-END UTILIZANDO JAVA Y SPRING**

**HAROLD FERNANDO ROJAS LOPEZ**

**10 DICIEMBRE 2024**

# TABLA DE CONTENIDO

<b>INTRODUCCIÓN</b>	<b>3</b>
<b>DESARROLLO DE LA ACTIVIDAD</b>	<b>4</b>
<b>Arquitectura general del sistema.</b>	<b>4</b>
Componentes principales.	4
API Gateway (Spring Cloud Gateway):	4
Eureka Server (Registro y Descubrimiento de Servicios):	4
Microservicio de Gestión de Rutas:	5
Microservicio de Gestión de Ciudades:	5
Base de Datos (MySQL):	5
<b>Flujo de Comunicación</b>	<b>5</b>
Registro en Eureka:	5
Configuración de servicios Actuator:	7
Comunicación y registros en Eureka:	7
Resolución de Servicios a través del API Gateway.	8
Enrutamiento hacia los microservicios	8
Rutas configuradas:	8
Integración con Actuator para la salud de servicios	9
Creación de las Bases de Datos para los Microservicios	10
Base de datos del Microservicio Gestión de Ciudades	10
Base de datos del Microservicio Gestión de Rutas	12
Servicios REST expuestos por los microservicios	13
Microservicio de Ciudades	13
Microservicio de Rutas	14
Comunicación bidireccional entre microservicios	16
Desde el microservicio de rutas al microservicio de ciudades	16
Desde el microservicio de ciudades al microservicio de rutas	17
Comparativa del Consumo de un Endpoint a través del Microservicio Directamente y del API Gateway	18
Acceso Directo al Microservicio	18
Consumo a través del API Gateway	19
Comparación General	20
<b>CONCLUSIONES</b>	<b>21</b>

# INTRODUCCIÓN

En el marco de la asignatura "*Desarrollo de aplicaciones web*", se lleva a cabo la actividad denominada "*Actividad 1: Desarrollo de un Back-End utilizando Java y Spring*". Este proyecto tiene como propósito la implementación de un sistema Back-End basado en una arquitectura de microservicios, utilizando tecnologías modernas como Java y el framework Spring. Este enfoque permite aplicar conceptos fundamentales en el diseño y consumo de APIs REST, así como explorar los beneficios y desafíos de las arquitecturas distribuidas.

El desarrollo se centra en la creación de dos microservicios independientes que exponen una serie de endpoints REST, implementando los verbos principales del protocolo HTTP: "GET, POST, PUT y DELETE". Además, los microservicios cuentan con métodos de búsqueda avanzados, permitiendo realizar consultas por atributos clave como el ID y el nombre de un atributo específico. Estos servicios utilizan MySQL como sistema de persistencia, garantizando un almacenamiento eficiente y estructurado de la información.

Para facilitar la escalabilidad y la comunicación entre servicios, se emplean componentes del ecosistema Spring como Spring Boot para la configuración y desarrollo simplificado, Spring Data para la gestión de persistencia, y Spring Cloud Eureka junto con Spring Cloud Gateway para el registro de servicios y el enrutamiento dinámico de las solicitudes. Este enfoque no solo asegura un sistema modular y fácilmente escalable, sino que también fomenta la reutilización de código y la separación de responsabilidades.

Con este proyecto, se busca consolidar los conocimientos adquiridos en el diseño de Back-Ends modernos, ofreciendo una solución robusta, eficiente y alineada con las necesidades actuales del desarrollo web. A lo largo del desarrollo, se pondrán en práctica principios de diseño, implementación y pruebas, abordando los desafíos técnicos que implica trabajar con arquitecturas de microservicios y APIs RESTful.

# DESARROLLO DE LA ACTIVIDAD

## Arquitectura general del sistema.

La arquitectura del sistema está diseñada bajo el patrón de *microservicios*, lo que significa que cada componente del sistema es autónomo, modular y cumple una función específica. La comunicación y la interacción entre los componentes se gestionan a través de un API Gateway y un servidor Eureka, facilitando el descubrimiento de servicios y el enrutamiento de peticiones.

A continuación, se describe cada componente y cómo interactúan entre sí:

## Componentes principales.

### API Gateway (Spring Cloud Gateway):

- Centraliza el acceso a los microservicios.
- Realiza el enrutamiento de las peticiones a los microservicios correspondientes basándose en configuraciones predefinidas (como rutas y filtros).
- Aplica reglas de transformación (como eliminar prefijos en las URLs).

### Eureka Server (Registro y Descubrimiento de Servicios):

- Actúa como un directorio donde los microservicios se registran automáticamente al iniciarse.
- Permite al API Gateway localizar los servicios dinámicamente por su nombre en lugar de usar direcciones IP o puertos específicos.
- Es esencial para manejar la escalabilidad y la alta disponibilidad en sistemas distribuidos.

## Microservicio de Gestión de Rutas:

- Gestiona las operaciones relacionadas con rutas (por ejemplo, creación, modificación, eliminación y consulta de rutas).
- Incluye búsquedas específicas, como por ID o por atributos como nombre o destino.
- Utiliza una base de datos MySQL para almacenar datos relacionados con las rutas.

## Microservicio de Gestión de Ciudades:

- Proporciona funcionalidades para gestionar información de ciudades (por ejemplo, agregar una ciudad, actualizar información, eliminar ciudades).
- Incluye búsquedas personalizadas, como por nombre o ID.
- También persiste sus datos en una base de datos MySQL.

## Base de Datos (MySQL):

- Cada microservicio tiene su propia base de datos para mantener la independencia y modularidad de los datos.
- Este diseño sigue el principio de "base de datos independiente por microservicio".

# Flujo de Comunicación

A continuación describiremos el rol que cumple cada componente del sistema en la arquitectura de microservicios.

## Registro en Eureka:

Por defecto, el servidor Eureka se ejecuta en el puerto 8761, lo que permite a los microservicios registrarse y descubrirse entre sí mediante la dirección <http://localhost:8761/eureka/>.

En esta implementación, los microservicios Gestión de Rutas y Gestión de Ciudades están configurados como clientes de Eureka mediante las siguientes propiedades:

Configuración del cliente Eureka en los microservicios:

- *eureka.client.service-url.defaultZone*: Define la URL del servidor Eureka al que los microservicios se registran. En este caso, apunta a <http://localhost:8761/eureka/>.
- *eureka.client.fetchRegistry*: Activa la capacidad del cliente para obtener el registro completo de Eureka y utilizarlo para descubrir otros servicios. Este parámetro está habilitado (true).
- *eureka.client.registerWithEureka*: Permite que los microservicios se registren automáticamente en Eureka, lo que está activado (true).

```
# Eureka client configuration
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.client.fetchRegistry=true
eureka.client.registerWithEureka=true
```

*Figura 1. Propiedades de configuración para un cliente Eureka.*

Para que cada microservicio pueda actuar como cliente de Eureka y registrarse correctamente en el servidor Eureka, es necesario incluir la dependencia correspondiente en el archivo pom.xml. Esta dependencia habilita el cliente Eureka en el proyecto utilizando Spring Cloud. A continuación se muestra la dependencia que debe agregarse a los archivos POM de los microservicios *Gestión de Rutas* y *Gestión de Ciudades*:

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

*Figura 2. Dependencia maven para configurar el microservicio como cliente Eureka.*

Al iniciar, los microservicios se registran automáticamente en el servidor Eureka. Cada servicio proporciona su nombre único y detalles como la dirección del host y el puerto.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SERVICIO-GESTION-CIUDADES	n/a (1)	(1)	UP (1) - <a href="#">host.docker.internal/servicio-gestion-ciudades:0</a>
SERVICIO-GESTION-RUTAS	n/a (1)	(1)	UP (1) - <a href="#">host.docker.internal/servicio-gestion-rutas:0</a>

*Figura 3. Microservicios registrados en el servidor Eureka.*

```
Registered instance SERVICIO-GESTION-CIUDADES/host.docker.internal:servicio-gestion-ciudades:0 with status UP (replication=false)
Registered instance SERVICIO-GESTION-RUTAS/host.docker.internal:servicio-gestion-rutas:0 with status UP (replication=false)
```

Figura 4. Log del servidor Eureka informando los microservicios registrados.

Configuración del nombre y puerto de los microservicios:

- *spring.application.name*: Define el nombre con el que cada microservicio se registra en Eureka.
  - El microservicio de Gestión de Ciudades se registra como *servicio-gestion-ciudades*.
  - El microservicio de Gestión de Rutas se registra como *servicio-gestion-rutas*.
- *server.port=0*: Configura los microservicios para que Spring Boot asigne un puerto aleatorio en cada inicio. Esto proporciona flexibilidad, especialmente en entornos con múltiples instancias del mismo servicio.

Configuración de servicios Actuator:

Para garantizar que Eureka pueda realizar un seguimiento del estado de los microservicios y manejar correctamente el heartbeat, se ha configurado Actuator en ambos servicios con las siguientes propiedades:

- *management.endpoints.web.exposure.include*: Expone los endpoints [/actuator/health](#) y [/actuator/info](#) de Actuator.
- *management.endpoint.health.show-details*: Muestra siempre los detalles del estado del microservicio, lo que es útil para monitorear la disponibilidad y el estado del servicio desde Eureka.

Comunicación y registros en Eureka:

Cada microservicio envía automáticamente un heartbeat al servidor Eureka mediante el endpoint */actuator/health*. Esto permite a Eureka actualizar periódicamente el estado de cada microservicio, marcándolos como UP o DOWN dependiendo de su disponibilidad.

## Resolución de Servicios a través del API Gateway.

El API Gateway utiliza la configuración de discovery locator para consultar la lista de servicios disponibles en Eureka, resolviendo dinámicamente las instancias y sus direcciones.

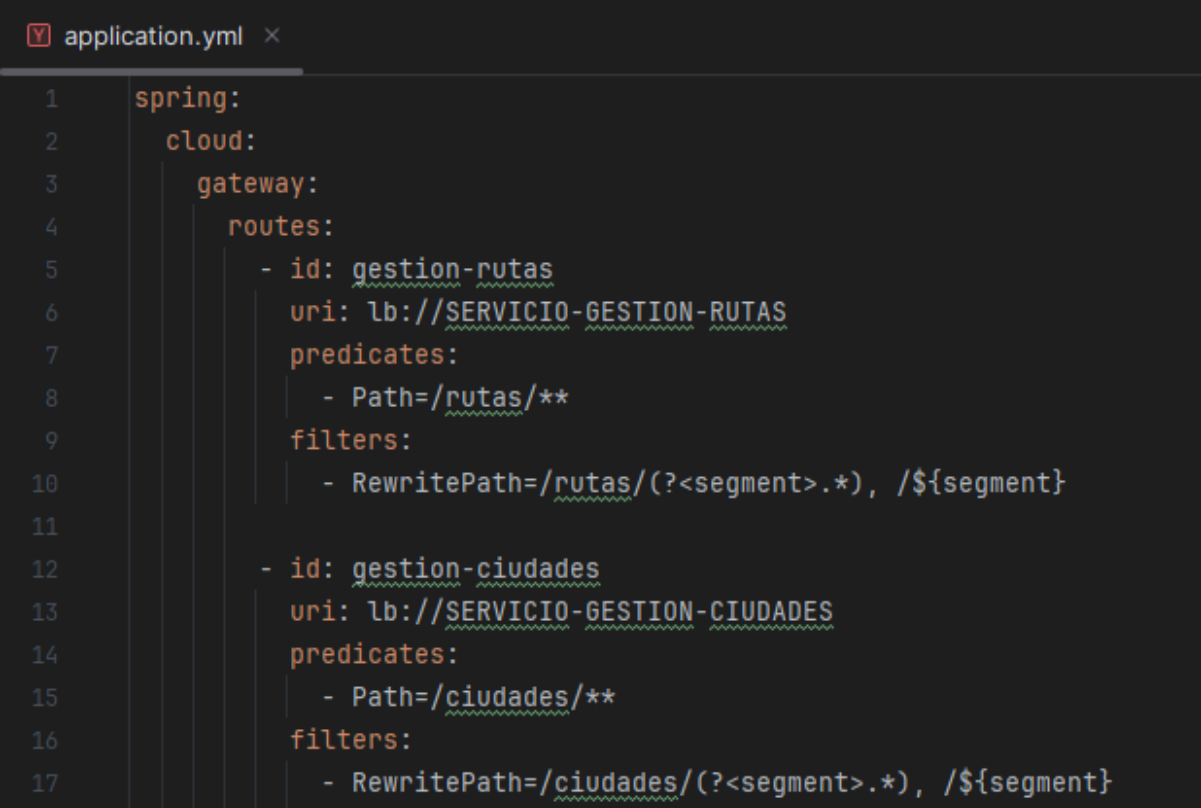
El API Gateway no necesita conocer las direcciones IP ni los puertos de los microservicios, ya que utiliza el nombre registrado en Eureka para resolver las instancias disponibles.

La propiedad *lb://* (*Load Balancer*) indica que las solicitudes se deben enviar al servicio correspondiente según su nombre (*SERVICIO-GESTION-RUTAS* o *SERVICIO-GESTION-CIUDADES*) y balancear entre las instancias disponibles.

### Enrutamiento hacia los microservicios

Cada ruta configurada en el API Gateway define un patrón de URL que se utiliza para enrutar solicitudes hacia los microservicios correspondientes.

Rutas configuradas:



```
1  spring:
2    cloud:
3      gateway:
4        routes:
5          - id: gestion-rutas
6            uri: lb://SERVICIO-GESTION-RUTAS
7            predicates:
8              - Path=/rutas/**
9            filters:
10              - RewritePath=/rutas/(?<segment>.*), /${segment}
11
12          - id: gestion-ciudades
13            uri: lb://SERVICIO-GESTION-CIUDADES
14            predicates:
15              - Path=/ciudades/**
16            filters:
17              - RewritePath=/ciudades/(?<segment>.*), /${segment}
```

Figura 5. Rutas configuradas en el ApiGateway



- Ruta para *SERVICIO-GESTION-RUTAS*:
  - *URL entrante*: [/rutas/\\*\\*](#)
  - *Acción*: Las solicitudes cuyo path comienza con [/rutas/](#) serán dirigidas al microservicio *SERVICIO-GESTION-RUTAS*.
  - *Filtro aplicado*: Se elimina el prefijo [/rutas/](#) antes de redirigir la solicitud al microservicio. Por ejemplo:
    - Entrada al Gateway: <http://localhost:9090/rutas/api/v1/rutas/1>
    - Reescritura y redirección: <http://<IP-del-microservicio>/api/v1/rutas/1>.
- Ruta para *SERVICIO-GESTION-CIUDADES*:
  - *URL entrante*: [/ciudades/\\*\\*](#)
  - *Acción*: Las solicitudes cuyo path comienza con [/ciudades/](#) serán dirigidas al microservicio *SERVICIO-GESTION-CIUDADES*.
  - *Filtro aplicado*: Se elimina el prefijo [/ciudades/](#) antes de redirigir la solicitud al microservicio. Por ejemplo:
    - Entrada al Gateway: <http://localhost:9090/ciudades/api/v1/ciudades/3>
    - Reescritura y redirección: <http://<IP-del-microservicio>/api/v1/ciudades/3>.

## Integración con Actuator para la salud de servicios

El API Gateway utiliza Spring Cloud LoadBalancer y consulta la salud de los microservicios a través de Actuator (configurado para exponer los endpoints [/actuator/health](#)).

Si un microservicio se encuentra fuera de servicio, Eureka actualiza su estado como *DOWN*, y el Gateway evita redirigir solicitudes hacia instancias no disponibles.

# Creación de las Bases de Datos para los Microservicios

En el desarrollo de los microservicios de Gestión de Ciudades y Gestión de Rutas, cada uno utiliza su propia base de datos en el motor MySQL, garantizando un diseño independiente y cohesivo en términos de almacenamiento y acceso a la información.

Ambos microservicios aprovechan los archivos *schema.sql* y *data.sql* para inicializar sus bases de datos. Al ejecutar las aplicaciones, Spring Boot detecta estos archivos y se asegura de que las bases de datos estén correctamente configuradas y pobladas con los datos iniciales.

A continuación, se detalla la creación de las bases de datos, tablas y datos iniciales.

## Base de datos del Microservicio Gestión de Ciudades

- *Conexión a la Base de Datos:* La base de datos se crea automáticamente al establecer conexión mediante la configuración:

```
spring.datasource.url=jdbc:mysql://localhost:3306/gestion_ciudades?createDatabaseIfNotExist=true
```

Esto asegura que la base de datos *gestion\_ciudades* será creada si no existe.

- *Tabla Ciudad:* La tabla ciudad almacena información básica sobre las ciudades principales. Su estructura incluye:
  - *id (BIGINT):* Identificador único de la ciudad.
  - *nombre (VARCHAR):* Nombre de la ciudad, con una restricción de unicidad para evitar duplicados.

Items	Queries	History	id	nombre
Search for item...			1	Bogotá
Functions			2	Medellín
Tables			3	Cali
ciudad			4	Barranquilla
			5	Cartagena
			6	Bucaramanga
			7	Pereira
			8	Santa Marta
			9	Manizales
			10	Cúcuta
			11	Caicedonia
			12	Armenia

Figura 6. Tabla ciudad en base de datos Mysql.

Script de creación (schema.sql):

```
1 CREATE DATABASE IF NOT EXISTS gestion_ciudades DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
2 USE gestion_ciudades;
3
4 CREATE TABLE IF NOT EXISTS ciudad (
5     id BIGINT AUTO_INCREMENT PRIMARY KEY,
6     nombre VARCHAR(255) NOT NULL UNIQUE
7 ) DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Figura 7. Script de creación de base de datos gestion\_ciudades y tabla ciudad.

- *Datos iniciales (data.sql)*: Se insertan por defecto las ciudades principales de Colombia, como Bogotá, Medellín y Cali, entre otras. Este es un ejemplo de los datos:

```
1 INSERT INTO ciudad (id, nombre)
2 SELECT 1, 'Bogotá'
3 WHERE NOT EXISTS (SELECT 1 FROM ciudad WHERE id = 1);
4
5 INSERT INTO ciudad (id, nombre)
6 SELECT 2, 'Medellín'
7 WHERE NOT EXISTS (SELECT 1 FROM ciudad WHERE id = 2);
```

Figura 8. Inserción de datos iniciales en la tabla ciudad.

## Base de datos del Microservicio Gestión de Rutas

- *Conexión a la Base de Datos:* La base de datos *gestion\_rutas* también se crea automáticamente mediante la siguiente configuración:

```
spring.datasource.url=jdbc:mysql://localhost:3306/gestion_rutas?createDatabaseIfNotExist=true
```

- *Tabla ruta:* La tabla *ruta* define las conexiones entre las ciudades y almacena las siguientes columnas:
  - *id (BIGINT):* Identificador único de la ruta.
  - *nombre (VARCHAR):* Descripción de la ruta, con una restricción de unicidad.
  - *ciudad\_id\_origen (BIGINT):* ID de la ciudad de origen.
  - *ciudad\_id\_destino (BIGINT):* ID de la ciudad de destino.
  - *distancia (DOUBLE):* Distancia en kilómetros entre las ciudades.

Script de creación (schema.sql):

```
1 CREATE DATABASE IF NOT EXISTS gestion_rutas DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
2 USE gestion_rutas;
3
4 CREATE TABLE IF NOT EXISTS ruta (
5     id BIGINT AUTO_INCREMENT PRIMARY KEY,
6     nombre VARCHAR(255) NOT NULL UNIQUE,
7     ciudad_id_origen BIGINT NOT NULL,
8     ciudad_id_destino BIGINT NOT NULL,
9     distancia DOUBLE NOT NULL
10 ) DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Figura 9. Creación de base de datos *gestion\_rutas* y de tabla *ruta*.

- *Datos iniciales (data.sql):* Se definen rutas importantes entre las principales ciudades de Colombia. Algunos ejemplos son:
  - Ruta Bogotá a Medellín (415 km).
  - Ruta Bogotá a Cali (465 km).
  - Ruta Barranquilla a Santa Marta (103 km).

Script de inserción de datos:

```
1 INSERT INTO ruta (id, nombre, ciudad_id_origen, ciudad_id_destino, distancia)
2 SELECT 1, 'Ruta Bogotá a Medellín', 1, 2, 415
3 WHERE NOT EXISTS (SELECT 1 FROM ruta WHERE id = 1);
4
5 INSERT INTO ruta (id, nombre, ciudad_id_origen, ciudad_id_destino, distancia)
6 SELECT 2, 'Ruta Bogotá a Cali', 1, 3, 465
7 WHERE NOT EXISTS (SELECT 1 FROM ruta WHERE id = 2);
```

Figura 10. Inserción de datos iniciales en la tabla ruta.

## Servicios REST expuestos por los microservicios

### Microservicio de Ciudades

El microservicio de ciudades permite gestionar y consultar información relacionada con las ciudades registradas en el sistema. Los servicios expuestos son:

Operación	Endpoint	Descripción
Crear una nueva ciudad	POST <a href="#">/api/v1/ciudades</a>	Permite agregar una ciudad al sistema proporcionando su nombre.
Obtener todas las ciudades	GET <a href="#">/api/v1/ciudades</a>	Devuelve una lista de todas las ciudades registradas.
Obtener una ciudad por ID	GET <a href="#">/api/v1/ciudades/{ciudadId}</a>	Devuelve los detalles de una ciudad específica usando su ID.
Buscar ciudades por nombre	GET <a href="#">/api/v1/ciudades/buscar</a>	Permite buscar ciudades cuyo nombre contenga una cadena de caracteres específica.
Actualizar una ciudad	PUT <a href="#">/api/v1/ciudades/{id}</a>	Actualiza el nombre de una ciudad existente proporcionando su ID.
Eliminar una ciudad	DELETE <a href="#">/api/v1/ciudades/{id}</a>	Elimina una ciudad específica usando su ID.
Verificar existencia de una ciudad por ID	GET <a href="#">/api/v1/ciudades/{ciudadId}/existe</a>	Devuelve true si una ciudad existe con el ID proporcionado, de lo contrario, false.

Consultar rutas relacionadas con una ciudad	GET <a href="#">/api/v1/ciudades/{id}/rutas</a>	Devuelve las rutas en las que una ciudad es origen o destino.
---	--	---

Tabla 1. Endpoints expuestos por el microservicio gestion-ciudades.

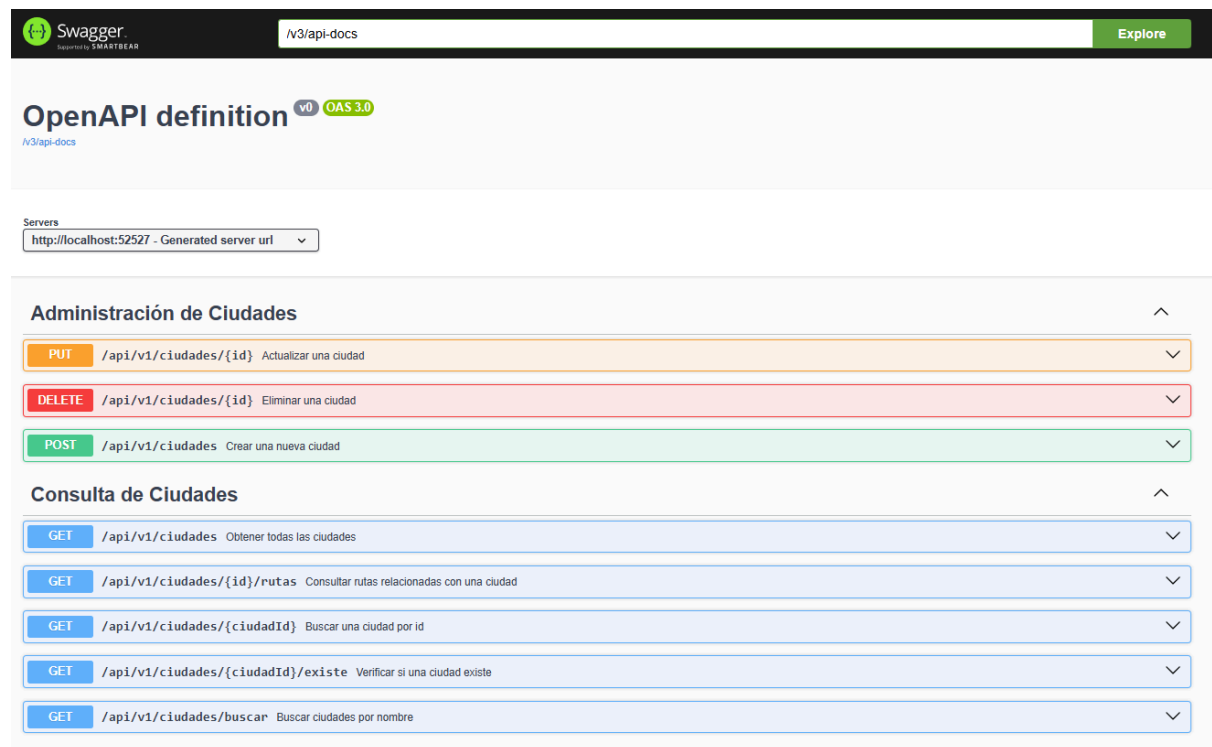


Figura 11. Swagger de los servicios expuestos por el microservicio gestion-ciudades.

## Microservicio de Rutas

El microservicio de rutas gestiona y consulta información sobre las rutas disponibles en el sistema. Los servicios expuestos son:

Operación	Endpoint	Descripción
Crear una nueva ruta	POST <a href="#">/api/v1/rutas</a>	Permite agregar una nueva ruta proporcionando su nombre, ciudad de origen, ciudad de destino y distancia.
Obtener todas las rutas	GET <a href="#">/api/v1/rutas</a>	Devuelve una lista de todas las rutas registradas.
Buscar rutas por nombre	GET <a href="#">/api/v1/rutas/buscar</a>	Permite buscar rutas cuyo nombre contenga una cadena de caracteres específica.

Actualizar una ruta	PUT <a href="#">/api/v1/rutas/{id}</a>	Actualiza los datos de una ruta existente proporcionando su ID.
Eliminar una ruta	DELETE <a href="#">/api/v1/rutas/{id}</a>	Elimina una ruta específica usando su ID.
Buscar una ruta por ID	GET <a href="#">/api/v1/rutas/{id}</a>	Devuelve los detalles de una ruta específica usando su ID.
Buscar rutas por ciudad de origen	GET <a href="#">/api/v1/rutas/buscar-origen</a>	Devuelve una lista de rutas cuyo origen coincide con la ciudad proporcionada.
Buscar rutas por ciudad de destino	GET <a href="#">/api/v1/rutas/buscar-destino</a>	Devuelve una lista de rutas cuyo destino coincide con la ciudad proporcionada.

Tabla 2. Endpoints expuestos por el microservicio gestion-rutas.

Swagger [v3/api-docs](#) [Explore](#)

OpenAPI definition v0 **OAS 3.0**

[v3/api-docs](#)

Servers

[http://localhost:52373](#) - Generated server url

### Consulta de Rutas

- GET [/api/v1/rutas/{id}](#) Buscar una ruta por ID
- GET [/api/v1/rutas](#) Obtener todas las rutas
- GET [/api/v1/rutas/buscar](#) Buscar rutas por nombre
- GET [/api/v1/rutas/buscar-origen](#) Buscar rutas por ciudad de origen
- GET [/api/v1/rutas/buscar-destino](#) Buscar rutas por ciudad de destino

### Administración de Rutas

- PUT [/api/v1/rutas/{id}](#) Actualizar una ruta
- DELETE [/api/v1/rutas/{id}](#) Eliminar una ruta
- POST [/api/v1/rutas](#) Crear una nueva ruta

Figura 12. Swagger de los endpoints expuestos por el microservicio gestion-rutas.

## Comunicación bidireccional entre microservicios

Desde el microservicio de rutas al microservicio de ciudades

Cuando se crea o consulta una ruta, el microservicio de rutas consulta al microservicio de ciudades para obtener los nombres de las ciudades de origen y destino correspondientes a los IDs proporcionados.

A continuación, se ejecuta el endpoint que lista todas las rutas, los nombres de las ciudades origen y destino son consultadas llamando al endpoint obtener una ciudad por Id del microservicio gestion ciudades.

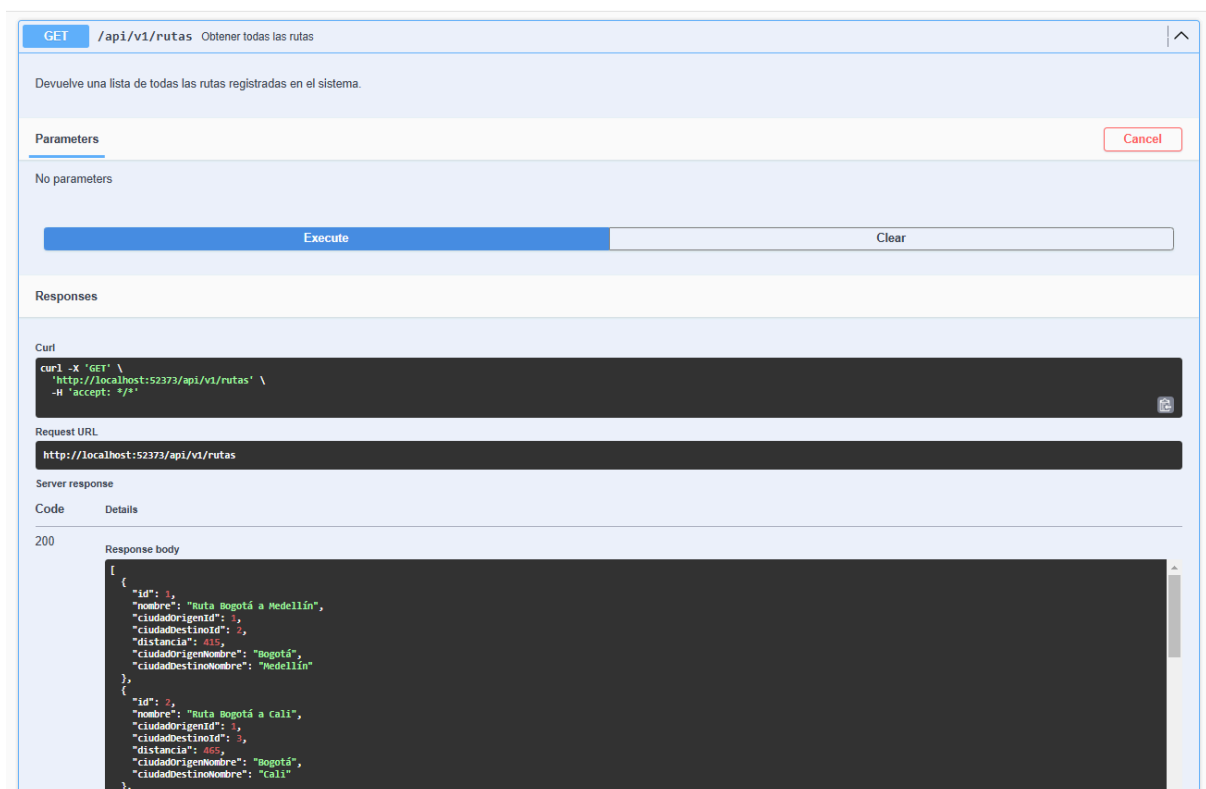


Figura 13. Llamado al endpoint para obtener todas las rutas.

Resaltar que la comunicación que se realiza desde el microservicio de rutas al microservicio de ciudades pasa a través del *ApiGateway*. El microservicio de de rutas tiene una propiedad en el archivo `application.properties` la cual es una URL que apunta al puerto del *ApiGateway* y al path ciudades, el cual internamente es enrutado al microservicio de ciudades.

```
microservicio.ciudades.url=http://localhost:9090/ciudades
```



En el log del *Apigateway* podemos visualizar la URL original que este recibe y el enrutamiento que realiza el servicio de gestion-ciudades.

```
Mapping [Exchange: GET http://localhost:9090/ciudades/api/v1/ciudades/1] to Route[id='gestion-ciudades', uri-lb://SERVICIO-GESTION-CIUDADES, order=0, predicate=Paths: [/ciudades/**], match trailing slash: true, gateway]
[c1c35907-35] Mapped to org.springframework.cloud.gateway.handler.FilteringWebHandler@3cfc55e0c
Sorted gatewayFilterFactories: [[GatewayFilterAdapter{delegate=org.springframework.cloud.gateway.filter.RemoveCachedBodyFilter@20a3e10c}, order = -2147483648], [GatewayFilterAdapter{delegate=org.springframework.cloud.g
```

Desde el microservicio de ciudades al microservicio de rutas

Al consultar rutas relacionadas con una ciudad (`/api/v1/ciudades/{id}/rutas`), el microservicio de ciudades realiza una solicitud al microservicio de rutas para obtener las rutas en las que la ciudad consultada es origen o destino.

GET /api/v1/ciudades/{id}/rutas Consultar rutas relacionadas con una ciudad

Consulta las rutas de las cuales esta ciudad es un origen o destino.

Parameters

Name	Description
id * required integer(\$int64) (path)	1

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:52527/api/v1/ciudades/1/rutas' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:52527/api/v1/ciudades/1/rutas
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "id": 1,   "nombre": "Ruta Bogotá a Medellín",   "ciudadOrigenId": 1,   "ciudadDestinoId": 2,   "distancia": 415,   "ciudadOrigenNombre": "Bogotá",   "ciudadDestinoNombre": "Medellín" }, {   "id": 2,   "nombre": "Ruta Bogotá a Cali",   "ciudadOrigenId": 1,   "ciudadDestinoId": 3,   "distancia": 465,   "ciudadOrigenNombre": "Bogotá",   "ciudadDestinoNombre": "Cali" }</pre>

Figura 14. Llamada al endpoint para consultar rutas relacionadas con una ciudad.

Al ingresar el id de la ciudad, este endpoint se comunica a través del *ApiGateway* con el microservicio de rutas, para obtener las rutas donde esta ciudad es un origen o un destino.

En la salida del log del *ApiGateway* podemos visualizar cómo se realiza el enrutamiento al microservicio de gestión rutas.

```
Mapping [Exchange: GET http://localhost:9090/rutas/api/v1/rutas/buscar-destino?ciudadId=1] to Route[id='gestion-rutas', uri-lb://SERVICIO-GESTION-RUTAS, order=0, predicate=Paths: [/rutas/**], match trailing slash: true]
Mapping [Exchange: GET http://localhost:9090/rutas/api/v1/rutas/buscar-origen?ciudadId=1] to Route[id='gestion-rutas', uri-lb://SERVICIO-GESTION-RUTAS, order=0, predicate=Paths: [/rutas/**], match trailing slash: true]
[d9c8f074-66] Mapped to org.springframework.cloud.gateway.handler.FilteringWebHandler@3cfc55e0c
[db0c959d-65] Mapped to org.springframework.cloud.gateway.handler.FilteringWebHandler@3cfc55e0c
```

A su vez en esta llamada para obtener las rutas las cuales la ciudad es un origen o destino, el microservicio de rutas llama de vuelta al microservicio de ciudades para obtener los nombres.

## Comparativa del Consumo de un Endpoint a través del Microservicio Directamente y del API Gateway

En una arquitectura de microservicios, los endpoints pueden ser consumidos de dos maneras principales: directamente desde el microservicio correspondiente o a través del API Gateway.

A continuación, se presenta una comparativa de ambas opciones en diferentes aspectos clave:

### Acceso Directo al Microservicio

Consiste en realizar la solicitud HTTP directamente al microservicio que expone el endpoint.

#### *Ventajas:*

- *Menor latencia:* Al eliminar el paso intermedio del API Gateway, la comunicación es más rápida en términos de transferencia de datos.
- *Simplicidad en ambientes locales o de pruebas:* Permite probar endpoints de manera más directa sin depender de configuraciones externas.

#### *Desventajas:*

- *Gestión compleja de múltiples endpoints:* En aplicaciones con múltiples microservicios, los clientes deben conocer y manejar las URLs específicas de cada servicio, lo que complica la integración y el mantenimiento.
- *Falta de seguridad centralizada:* Cada microservicio necesita implementar individualmente mecanismos de autenticación, autorización y rate-limiting, lo que incrementa la duplicación de esfuerzos.
- *Dependencia en cambios:* Si cambia la URL o el puerto del microservicio, los clientes deben actualizar manualmente estas referencias.

Ejemplo:

Una consulta al endpoint GET `/api/v1/ciudades/{ciudadId}` del microservicio de Ciudades.

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** `/api/v1/ciudades/{ciudadId}` (Description: Buscar una ciudad por id)
- Description:** Devuelve la ciudad asociada al id
- Parameters:**
  - Name:** ciudadId (required)
  - Description:** integer (\$int64) (path)
  - Value:** 1
- Buttons:** Execute, Clear, Cancel
- Responses:**
  - Curl:**

```
curl -X 'GET' \
'http://localhost:52527/api/v1/ciudades/1' \
-H 'accept: */*'
```
  - Request URL:** `http://localhost:52527/api/v1/ciudades/1`
  - Server response:**

Code	Details
200	<p><b>Response body</b></p> <pre>{   "id": 1,   "nombre": "Bogotá" }</pre>

Figura 15. Llamada al endpoint buscar una ciudad por id directamente al microservicio.

## Consumo a través del API Gateway

Aquí, todas las solicitudes son canalizadas a través del API Gateway, que actúa como punto único de entrada para el sistema.

### Ventajas:

- **Abstracción de la infraestructura:** Los clientes no necesitan conocer las URLs internas de los microservicios. Solo interactúan con una URL unificada, como <http://gateway-service>.
- **Seguridad centralizada:** El API Gateway puede gestionar la autenticación, autorización y restricciones de acceso para todos los microservicios.
- **Balanceo de carga y resiliencia:** Redirige las solicitudes a las instancias disponibles del microservicio correspondiente, garantizando alta disponibilidad.
- **Comunicación bidireccional simplificada:** En casos de microservicios con dependencia mutua (como Rutas y Ciudades), el Gateway coordina las solicitudes sin exponer las implementaciones internas.
- **Facilidad para escalar y actualizar:** Cambios en las URLs de los microservicios no afectan a los clientes, ya que la configuración se maneja en el Gateway.

Desventajas:

- *Posible incremento de latencia:* Al ser un intermediario, añade un pequeño retraso en la comunicación.
- *Punto único de fallo:* Aunque se mitiga con estrategias de alta disponibilidad, cualquier problema en el Gateway afecta a todo el sistema.

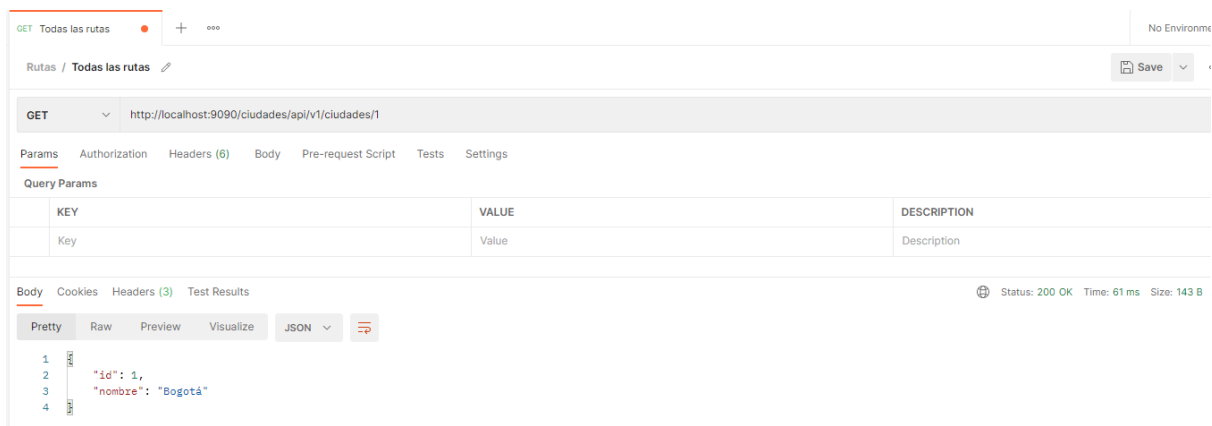


Figura 16. Llamada al endpoint buscar una ciudad por id a través del ApiGateway.

Comparación General

Aspecto	Directo al Microservicio	A través del API Gateway
Simplicidad inicial	Alta (para entornos pequeños)	Alta (para entornos grandes)
Escalabilidad	Limitada	Alta
Manejo de seguridad	Distribuido en cada microservicio	Centralizado en el Gateway
Rendimiento	Menor latencia	Ligera latencia adicional
Mantenimiento	Complejo con muchos servicios	Simplificado
Resiliencia	Limitada	Mejorada (balanceo y fallback)

Tabla 3. Comparativa general.

# CONCLUSIONES

La experiencia con Eureka, el API Gateway y microservicios en general ha dejado claro que, si bien estas tecnologías requieren una curva de aprendizaje, son indispensables para manejar sistemas complejos en entornos dinámicos. Eureka simplifica enormemente el descubrimiento de servicios, mientras que el API Gateway centraliza y optimiza la interacción con los clientes. Este proyecto no solo refuerza la relevancia de estas herramientas, sino también su capacidad para mejorar la escalabilidad, disponibilidad y flexibilidad de las aplicaciones modernas.

## *1. Profundización en los conceptos de microservicios*

La implementación de microservicios permitió entender mejor cómo funciona una arquitectura distribuida, destacando:

- La importancia de la independencia y autonomía de cada servicio para facilitar su despliegue, escalabilidad y mantenimiento.
- La necesidad de herramientas que centralicen y gestionen de manera eficiente la interacción entre servicios, como el API Gateway y Eureka.

## *2. Entendimiento del rol del API Gateway*

El uso del API Gateway reafirmó su importancia como punto central para la interacción entre clientes y microservicios:

- Proporciona una abstracción de la infraestructura interna, lo que simplifica las integraciones y mejora la seguridad al manejar autenticación y autorización desde un único lugar.
- Facilita la transformación de datos y otras tareas, como la limitación de tasas y el balanceo de carga, sin sobrecargar a los microservicios.
- A pesar de introducir una capa adicional, el impacto en latencia es mínimo frente a los beneficios en control, eficiencia y resiliencia.

## *3. Integración con Eureka para el descubrimiento de servicios*

La inclusión de Eureka en la arquitectura fue un elemento crucial que destacó su valor en entornos dinámicos:

- Eureka actúa como un registro centralizado de servicios, permitiendo que los microservicios se descubran mutuamente sin necesidad de configuraciones estáticas. Esto eliminó la dependencia de URLs o puertos fijos.
- La integración con el API Gateway simplificó el ruteo dinámico hacia microservicios registrados, mejorando la flexibilidad del sistema.
- En escenarios donde hay cambios frecuentes (por ejemplo, despliegues, escalado de instancias o fallos), Eureka demostró ser indispensable para garantizar la disponibilidad y la resiliencia del sistema.

#### *4. Mejora en habilidades técnicas*

El proyecto permitió reforzar habilidades clave en tecnologías relacionadas con microservicios:

- Spring Cloud Eureka: su configuración, registro y consumo de servicios, así como su rol en el balanceo de carga.
- API Gateway: configuración de ruteo dinámico, seguridad y gestión de tráfico.
- Comprensión de cómo estas herramientas se combinan para crear un sistema resiliente, flexible y mantenible.