INSTITUTO POLITÉCNICO NACIONAL
"ESCUELA SUPERIOR DE CÓMPUTO"

SISTEMAS DISTRIBUIDOS

Práctica 7:

"Microservicios"

Alumno: **Miguel Ángel Vázquez Cisneros**

Profesor: **Carreto Arellano Chadwick**

Grupo: **7CM1**

Índice

Antecedente

A lo largo de la evolución de los sistemas distribuidos, distintas tecnologías han surgido para facilitar la comunicación y el intercambio de datos entre múltiples dispositivos y aplicaciones. En sus primeras etapas, estos sistemas se basaban en modelos básicos de comunicación, como los sockets, que permitían la transferencia de información entre dos puntos de una red. Este enfoque, aunque eficiente para ciertas aplicaciones, presentaba limitaciones en términos de escalabilidad y mantenimiento, ya que cada conexión debía gestionarse manualmente y los protocolos de comunicación eran específicos de cada implementación.

Con el tiempo, los sistemas distribuidos evolucionaron hacia arquitecturas más flexibles, introduciendo modelos cliente-servidor en los que múltiples clientes podían conectarse simultáneamente a un servidor centralizado. Este paradigma permitió un mejor manejo de múltiples conexiones, dando lugar a aplicaciones más complejas que soportaban interacciones de N clientes con N servidores. Sin embargo, este modelo todavía dependía de implementaciones específicas y requería un conocimiento profundo de los protocolos subyacentes, lo que dificultaba la interoperabilidad entre distintas plataformas.

Para abordar estas limitaciones, surgieron los objetos distribuidos, una tecnología que facilitó la invocación remota de métodos en objetos ubicados en diferentes sistemas. Tecnologías como CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation) en Java y DCOM (Distributed Component Object Model) en entornos Microsoft permitieron que las aplicaciones interactuaran de manera más estructurada y modular. Sin embargo, la necesidad de librerías específicas y la compatibilidad entre distintos entornos seguían siendo desafíos importantes.

La necesidad de una solución más universal y accesible llevó al desarrollo de los servicios web, que han llegado a dominar el panorama actual de los sistemas distribuidos. Basados en protocolos estándares como HTTP y utilizando formatos de intercambio de datos como XML y JSON, los servicios web permiten la comunicación entre sistemas heterogéneos sin la necesidad de una implementación específica para cada plataforma. Tecnologías como SOAP (Simple Object Access Protocol) y REST (Representational State Transfer) han permitido la creación de APIs accesibles desde cualquier entorno con conexión a Internet, facilitando la integración de aplicaciones y la escalabilidad de los sistemas distribuidos.

Posteriormente surgieron los microservicios para aumentar el nivel de independencia, integración y escalabilidad de los sistemas. Aquellos sistemas que implementan sistemas distribuidos pueden aprovechar solo cierta funcionalidad e incluso combinarla con la funcionalidad de otro microservicio, utilizándolos como parte fundamental de su sistema y lógica de negocio, a diferencia de los servicios web, que generalmente no se podían integrar a fondo con el sistema en desarrollo.

Planteamiento del Problema

En el desarrollo de videojuegos multijugador o con interacción en red, es fundamental contar con una arquitectura eficiente que permita la comunicación entre los jugadores y el sistema central que gestiona el estado del juego. Tradicionalmente, estos sistemas han sido implementados utilizando arquitecturas cliente-servidor, donde un servidor mantiene el estado del mundo del juego y los clientes envían comandos para interactuar con él.

En esta práctica, se requiere desarrollar un simulador de videojuego basado en una arquitectura basada en Servicios Web utilizando Java. El servidor actúa como aquel que publica las direcciones o end-points que se encargan de la lógica de almacenando y actualizando la información clave de los jugadores, como su posición en el mapa, su nivel y su cantidad de vida. Además, administra la ubicación de enemigos dentro del mapa y la mecánica de combate. Por otro lado, los clientes representan al jugador, quien puede usar los métodos establecidos en el objeto para interactuar con el juego.

El problema principal a resolver es cómo diseñar y gestionar una comunicación eficiente entre los clientes y el servidor para que el juego pueda operar de manera fluida y los datos o estados de los jugadores queden perfectamente separados entre cada uno de ellos. Para ello, se deben considerar los siguientes aspectos:

- **Gestión del estado del jugador y del entorno**: Los servidores tienen que tener la información consistente sobre la posición de los enemigos y demás estados generales del juego.

- **Interacción en tiempo real**: Los clientes deben ser capaces de enviar comandos al servidor y recibir respuestas de manera eficiente para garantizar una experiencia de juego dinámica. Sin mencionar que las acciones de otros jugadores en el entorno, como la baja de un enemigo, se debe reflejar en todos los jugadores.

- **Sistema de combate**: Se deben definir reglas para el ataque, la detección de enemigos en el mapa y la actualización del estado del jugador en función de las batallas.

- **Comunicación por Servicios Web con REST**: Implementar una arquitectura basada en Servicios Web.

- **Integración de microservicios con Eureka Server**: Levantar un servidor en el cual los microservicios se registren para que se puedan utilizar e interactuar.

Propuesta de Solución

Para la simulación de un juego multijugador, se plantea la implementación de microservicios basado en la arquitectura REST, que permitirá la interacción de múltiples jugadores dentro de un entorno distribuido. La solución consistirá en el desarrollo de un servidor central con Eureka, que gestionará los microservicios que formen parte del sistema junto con un API Gateway que redireccionará las peticiones al microservicio que corresponda.

Cada jugador podrá registrarse en el sistema y enviar solicitudes al servidor para realizar diversas acciones dentro del juego, como moverse en un mapa virtual, atacar enemigos y actualizar su estado. Estas interacciones se manejarán mediante peticiones HTTP a los distintos endpoints del microservicio correspondiente, los cuales procesarán la información y actualizarán la base de datos centralizada que contendrá el estado de los jugadores y del entorno del juego.

La comunicación entre el cliente y el servidor se realizará a través de JSON, permitiendo una transmisión de datos ligera y eficiente. Además, se implementará una capa de persistencia con una base de datos relacional para garantizar que la información del juego se mantenga correctamente almacenada y pueda ser consultada o modificada en cualquier momento.

Esta solución aprovecha las ventajas de los microservicios para proporcionar un sistema escalable y accesible, en el que múltiples jugadores pueden interactuar simultáneamente sin depender de conexiones persistentes o configuraciones específicas de red. De este modo, se logra una simulación de juego multijugador eficiente y adaptable a distintos entornos tecnológicos.

Materiales y Métodos Empleados

Para el desarrollo de la práctica, se utilizaron diversas herramientas de software y técnicas de programación orientadas a la creación de aplicaciones cliente-servidor en Java utilizando sockets. El objetivo principal fue simular el entorno de un videojuego controlado. A continuación, se detallan los materiales y métodos empleados:

Materiales

1. **Entorno de desarrollo integrado (IDE): IntelliJ IDEA**

Se utilizó **IntelliJ IDEA** como el entorno de desarrollo integrado (IDE) principal, debido a sus potentes herramientas de edición, depuración y gestión de proyectos en Java. Esta plataforma proporciona características avanzadas como el autocompletado de código, análisis de errores en tiempo real y una interfaz intuitiva que facilita la creación y ejecución de aplicaciones Java complejas.

2. **Lenguaje de programación: Java**

El proyecto se desarrolló íntegramente en **Java**, aprovechando sus capacidades nativas de programación orientada a objetos y su robusta gestión de hilos. Java proporciona una API de concurrencia flexible y herramientas integradas para la creación de aplicaciones multihilo, lo que la convierte en una opción ideal para este tipo de prácticas.

3. **Spring Boot**

Framework para el desarrollo del servicio web REST.

4. **MySQL**

Base de datos relacional utilizada para almacenar la información de los jugadores y el estado del juego.

5. **Sistema operativo y hardware**
    - **Sistema operativo:** Windows 10 pro que es compatible con Java Development Kit (JDK).
    - **JDK:** Java Development Kit (JDK) versión 21.
    - **Hardware mínimo:**
        - Procesador Intel i5
        - 16 GB de memoria RAM.
        - Al menos 500 MB de espacio disponible en disco.

Métodos

Para llevar a cabo la simulación del juego multijugador utilizando servicios web, se siguieron una serie de metodologías y enfoques técnicos que garantizaron la correcta implementación y funcionamiento del sistema. A continuación, se describen los principales métodos empleados:

1. **Desarrollo del servidor web con Spring Boot**
   Se utilizó el framework Spring Boot para la creación del servidor web, ya que facilita la configuración y gestión de servicios REST.

2. **Uso de una arquitectura RESTful**
   Se implementó una arquitectura basada en REST para la comunicación entre el cliente y el servidor. Cada acción del juego (como mover un jugador, registrar un nuevo usuario o atacar a un enemigo) se implementó como un endpoint accesible a través de peticiones HTTP con los métodos adecuados (GET, POST, PUT, DELETE).

3. **Persistencia de datos con JPA y MySQL**
   Para almacenar y administrar la información del juego, se utilizó una base de datos relacional en MySQL. Se empleó Java Persistence API (JPA) junto con Spring Data JPA para facilitar la gestión de las entidades y el acceso a los datos de manera eficiente.

4. **Intercambio de datos en formato JSON**
   La comunicación entre el cliente y el servidor se llevó a cabo utilizando JSON como formato de intercambio de datos. Esto permitió que las peticiones y respuestas fueran ligeras y fácilmente procesables por cualquier cliente que desee interactuar con la API.

5. **Manejo de peticiones con HttpClient en el cliente**
   En la parte del cliente, se utilizó la clase `HttpClient` para enviar solicitudes HTTP al servidor y recibir las respuestas. Dependiendo de la acción a realizar, se enviaban peticiones con parámetros en la URL o con un cuerpo JSON.

6. **Control de flujo y validaciones**
   Para garantizar el correcto funcionamiento del sistema, se implementaron validaciones en el servidor antes de procesar las solicitudes. Por ejemplo, se verificaba que un jugador existiera antes de permitirle moverse o atacar, y se aseguraba que los datos enviados por el cliente fueran válidos.

7. **Registro y actualización del estado del juego**
   Se diseñó un esquema de base de datos que permitiera mantener actualizada la información de los jugadores, sus posiciones y acciones dentro del juego. Cada

cambio en el estado de un jugador se reflejaba en la base de datos y podía ser consultado mediante peticiones al servidor.

Este conjunto de métodos permitió la implementación de un sistema distribuido funcional, en el que múltiples clientes pueden interactuar con un servidor centralizado que gestiona el estado del juego en tiempo real.

Desarrollo de la Solución

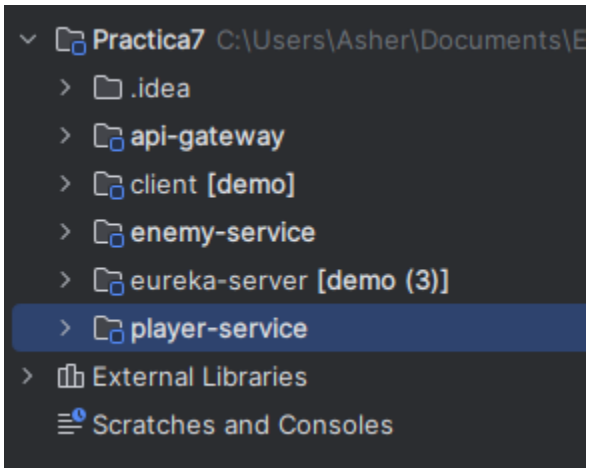
1. Diseño general del sistema

Antes de implementar, se definieron las siguientes reglas de interacción:

- Los microservicios serían independientes y se comunicarían **únicamente por HTTP**.
- Se usaría **Eureka como servicio de descubrimiento**.
- La **comunicación entre servicios** (sin pasar por el API Gateway) se realizaría con **OpenFeign**, aprovechando el balanceador de carga mediante `lb://`.
- El API Gateway serviría de punto de entrada para los clientes externos, mientras que la comunicación interna entre microservicios usaría nombres lógicos vía Eureka

2. Estructura de directorios

La arquitectura de directorios que se utilizó fue la siguiente, teniendo en cuenta que cada carpeta es un proyecto independiente de los demás que puede correr por su cuenta, sin embargo, algunos dependen de que cierto servicio o servidor ya esté activo.



3. Implementación de microservicios

a) Como en prácticas anteriores, tenemos la clase mapeada como una entidad para poderla persistir en la base de datos con Spring Data.

```java
@Entity  5 usages
@Table(name = "Enemigo")
public class EnemyEntity {
    @Id  1 usage
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false)
    private Long id;
    @Column(nullable = false)  2 usages
    private int posX;
    @Column(nullable = false)  2 usages
    private int posY;

    public EnemyEntity(){}  no usages

    public Long getId() { return id; }

    public int getPosX() { return posX; }

    public int getPosY() { return posY; }

    public void setPosX(int posX) { this.posX = posX; }

    public void setPosY(int posY) { this.posY = posY; }
}
```

b) Realizamos el repositorio para poder interactuar con la BD

```java
package com.example.enemy_service;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository  2 usages
public interface EnemyRepository extends JpaRepository<EnemyEntity,Long> {
}
```

c) Se implementa el controlador del servicio

```java
@RestController  no usages
@RequestMapping("/enemies")
public class EnemyController {
    private final EnemyService enemyService;  3 usages

    public EnemyController(EnemyService enemyService) { this.enemyService = enemyService; }

    @RequestMapping("/isInEnemy")  no usages
    public boolean isInEnemy(@RequestParam int x,@RequestParam int y) { return enemyService.isInEnem

    @RequestMapping("/delete")  no usages
    public String delete(@RequestParam int x, @RequestParam int y) { return enemyService.delete(x,y)
```

d) Se implementan los métodos del controlador como tal, en una clase service

```java
@Service  2 usages
public class EnemyService {
    private final EnemyRepository enemyRepository;  4 usages

    public EnemyService(EnemyRepository enemyRepository){  no usages
        this.enemyRepository = enemyRepository;
    }

    public boolean isInEnemy(int x, int y){  1 usage
        List<EnemyEntity> lst = enemyRepository.findAll();
        for(EnemyEntity enemy: lst){
            if(enemy.getPosX()==x && enemy.getPosY()==y){
                return true;
            }
        }
        return false;
    }

    public String delete(int x, int y){  1 usage
        List<EnemyEntity> lst = enemyRepository.findAll();
        for(EnemyEntity enemy: lst){
            if(enemy.getPosX()==x && enemy.getPosY()==y){
                enemyRepository.delete(enemy);
```

e) Clase principal

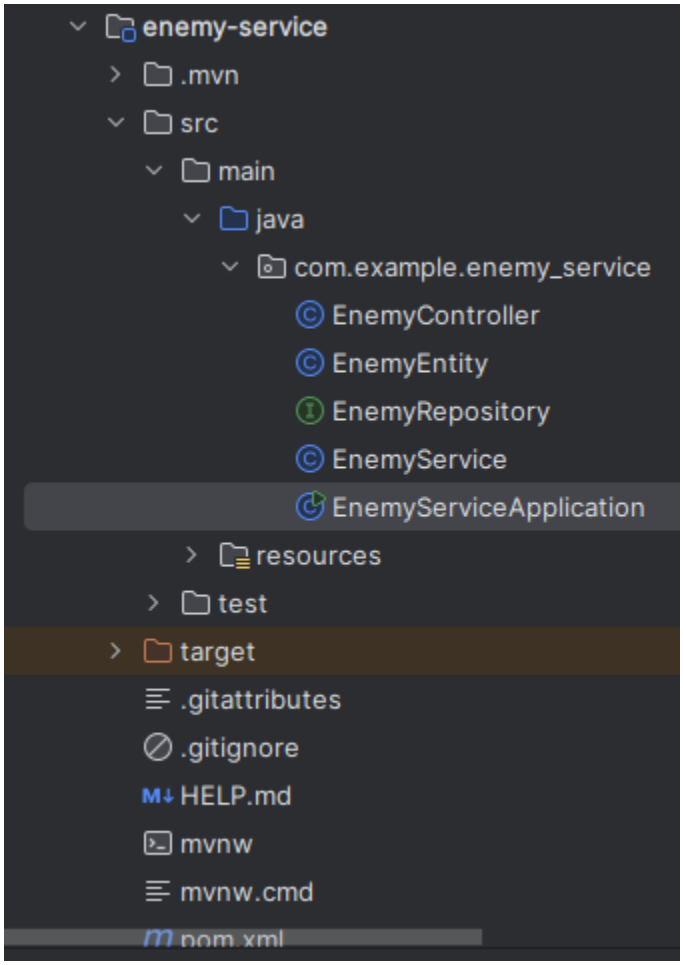
```java
@SpringBootApplication
@RestController
public class EnemyServiceApplication {

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

    @GetMapping("/enemies/test")  no usages
    public String test() {
        return "El servicio de jugadores está funcionando";
    }

}
```

De tal forma que la estructura de los microservicios se basa en la siguiente:



4. Registro de microservicios en Eureka

Para evitar problemas de resolución de nombre (como `DESKTOP-P2Q9TU5.mshome.net`), se añadió la siguiente configuración a todos los `application.yml` (PlayerService, EnemyService y Gateway)

```yaml
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
    ip-address: 127.0.0.1
```

5. Exposición de endpoints en `EnemyService`

Se creó un controlador REST para exponer los métodos necesarios:

a) Eliminar un enemigo

```java
@DeleteMapping("/enemies/{id}")
public ResponseEntity<Void> deleteEnemy(@PathVariable Long id) {
    boolean eliminado = enemyService.deleteEnemyById(id);
    return eliminado ? ResponseEntity.ok().build() : ResponseEntity.notFound().build();
}
```

b) Verificar si hay un enemigo en cierta posición

```java
@GetMapping("/enemies/at")
public boolean isEnemyAt(@RequestParam int x, @RequestParam int y) {
    return enemyService.existsEnemyAtPosition(x, y);
}
```

6. Comunicación desde `PlayerService` usando OpenFeign

a) Se agrega la dependencia de Feign

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

b) Se agrega la anotación para que el proyecto acepe el proyecto de Feign con una anotación en la clase principal. Además, se crea la interfaz de cliente

```
@FeignClient(name = "enemy-service")
public interface EnemyServiceClient {

    @DeleteMapping("/enemies/{id}")
    void deleteEnemy(@PathVariable("id") Long id);

    @GetMapping("/enemies/at")
    boolean isEnemyAt(@RequestParam("x") int x, @RequestParam("y") int y);
}
```

7. Uso del cliente desde la lógica de jugador

Dentro del `PlayerService`, se inyectó `EnemyServiceClient` para su uso directo:

```java
@Autowired
private EnemyServiceClient enemyClient;

public void move(int x, int y) {
    if (enemyClient.isEnemyAt(x, y)) {
        System.out.println("¡Cuidado! Hay un enemigo en esta posición.");
    } else {
        // lógica de movimiento...
    }
}

public void attack(Long enemyId) {
    try {
        enemyClient.deleteEnemy(enemyId);
    } catch (FeignException.NotFound e) {
        System.out.println("No se encontró al enemigo.");
```

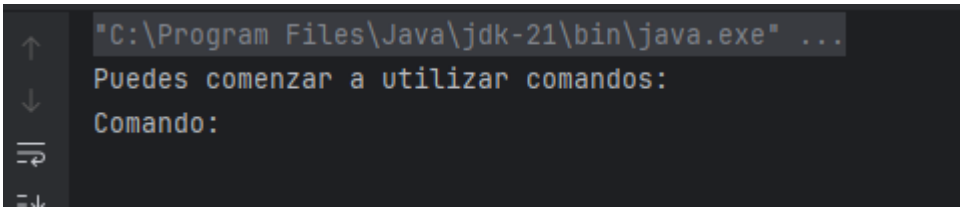
8. Verificación del funcionamiento

- Se realizaron peticiones desde el cliente, a través del Gateway, que eran correctamente redirigidas a los microservicios.
- Se verificó en los logs que el enrutamiento se hiciera mediante `lb://enemy-service`, y que el nombre no se resolviera a un host inválido.
- Se comprobó que `PlayerService` podía consultar y modificar datos de `EnemyService` correctamente.
- Se validó el correcto uso del patrón cliente Feign como alternativa a `RestTemplate`.

Resultados

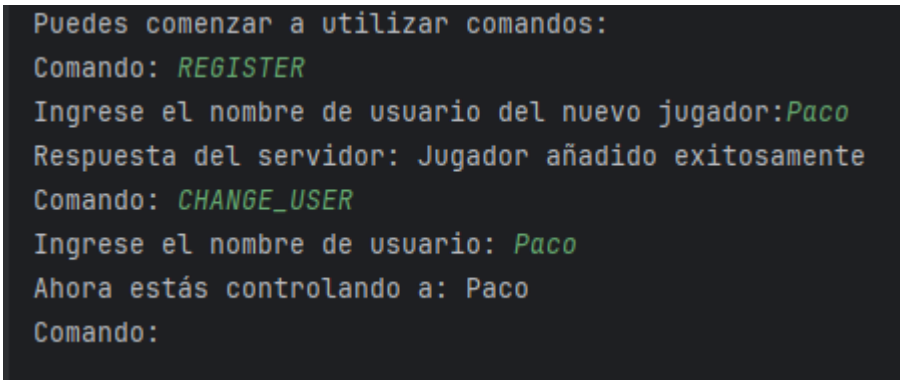
Para que todo funcione de forma correcta primero se debe iniciar el servidor de Eureka, luego se inician los microservicios de Player y Enemy y, por último, el Api Gateway.

Ahora iniciamos tantos clientes como queramos



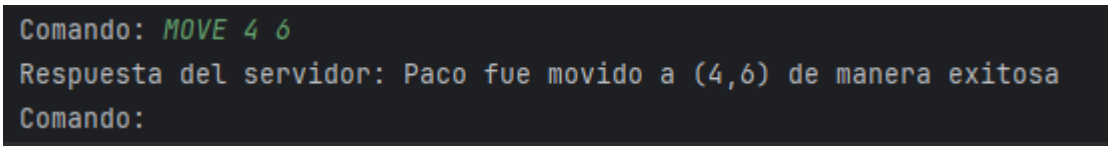
Primero necesitamos o bien, registrar un jugador o usar alguno usando su username.

Una vez que los clientes se han conectado al servidor, pueden comenzar a usar comandos como MOVE X Y, ATTACK, STATUS, REGISTER y CHANGE_USER. Como se mencionó anteriormente, primero de debe registrar un jugador con REGISTER, o bien, si ya se tiene un usuario, se usa CHANGE_USER para usar a ese personaje.



MOVE X Y

Mueve al usuario a las coordenadas seleccionadas



Si en las coordenadas seleccionadas hay un enemigo, el enemigo ataca al jugador infringiéndole algo de daño

```
Comando: MOVE 1 1
Respuesta del servidor: Paco fue movido a (1,1) de manera exitosa. Está parado sobre un enemigo
Comando:
```

ATTACK

Si el jugador está arado sobre un enemigo, puede atacarlo y acabar con él. Al derrotar a un enemigo el jugador sube de nivel

```
Comando: ATTACK
Respuesta del servidor: Ataque realizado con éxito. Haz eliminado al enemigo
Comando:
```

STATUS

Muestra las estadísticas actuales del jugador

```
Respuesta del servidor: El jugador Paco está en la posición (1,1)
Comando:
```

Conclusiones

La implementación de esta práctica basada en una arquitectura de microservicios permitió comprender de manera integral los conceptos clave que fundamentan el desarrollo de sistemas distribuidos modernos. A través de la separación de responsabilidades en distintos servicios como `player-service`, `enemy-service`, `api-gateway` y `eureka-server`. Se evidenció cómo esta arquitectura favorece la escalabilidad, mantenibilidad y evolución independiente de cada componente del sistema.

El uso de **Eureka** como servidor de descubrimiento permitió que los distintos microservicios se registraran y descubrieran dinámicamente entre sí. Esta característica es esencial en entornos donde los servicios pueden escalar o cambiar de dirección IP, ya que permite mantener una comunicación estable sin necesidad de conocer previamente las ubicaciones exactas de cada uno.

El **API Gateway** sirvió como punto de entrada unificado al sistema, redirigiendo las solicitudes entrantes al microservicio correspondiente mediante rutas dinámicas definidas por patrones. Además, se experimentó con la funcionalidad de balanceo de carga gracias a la integración del prefijo `lb://`, lo cual permite distribuir las peticiones entre múltiples instancias de un mismo servicio, mejorando la disponibilidad y tolerancia a fallos.

Una de las partes más enriquecedoras de la práctica fue la integración de **Feign Client**, una herramienta que facilita la comunicación entre microservicios al permitir hacer peticiones HTTP mediante simples interfaces Java, ocultando la complejidad de la implementación del cliente HTTP. Gracias a esto, se logró que, por ejemplo, `player-service` pudiera interactuar directamente con `enemy-service` para consultar si en una posición determinada existía un enemigo o incluso para eliminarlo.

Asimismo, se enfrentaron desafíos técnicos comunes al trabajar con microservicios, como la resolución de nombres de host en ambientes locales, el registro incorrecto en Eureka debido a configuraciones por defecto del sistema operativo, y errores derivados de fallas en la configuración de red o dependencias. Estos problemas permitieron profundizar en la configuración detallada de `application.yml` en cada servicio, como el uso de `prefer-ip-address` o la especificación de la IP manual para asegurar una correcta resolución y comunicación entre servicios.

La presente práctica no solo permitió afianzar conocimientos teóricos sobre microservicios, sino que también brindó una experiencia práctica completa que abarcó desde la configuración de servicios hasta la resolución de errores reales. Todo esto demostró que, si bien la arquitectura de microservicios puede ser compleja al principio, su dominio ofrece grandes beneficios en términos de flexibilidad, escalabilidad y robustez para sistemas distribuidos modernos.