INSTITUTO POLITÉCNICO NACIONAL
"ESCUELA SUPERIOR DE CÓMPUTO"

SISTEMAS DISTRIBUIDOS

Práctica 6:

"Servicios Web"

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.

Hoy en día, los servicios web representan una de las formas más eficientes y flexibles de desarrollar aplicaciones distribuidas, posibilitando la comunicación entre dispositivos y plataformas de manera estandarizada y escalable. Con el auge de arquitecturas como microservicios y la computación en la nube, estos servicios han demostrado ser fundamentales para la evolución de las aplicaciones modernas, proporcionando soluciones eficientes para la distribución de datos y el procesamiento descentralizado en entornos cada vez más interconectados.

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.

Propuesta de Solución

Para la simulación de un juego multijugador, se plantea la implementación de un servicio web 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 que gestionará el estado del juego y proporcionará una API accesible para que los clientes puedan conectarse y realizar acciones en tiempo real.

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 servicio web, 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 servicios web 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. A través de controladores (*Controllers*), se definieron los distintos endpoints que permiten la interacción entre los clientes y el servidor.

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.

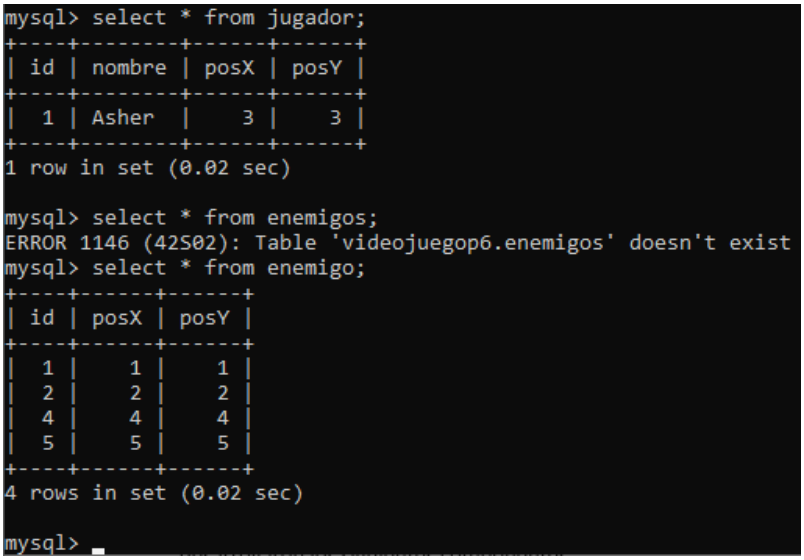
Desarrollo de la Solución

Para llevar a cabo la simulación del juego multijugador utilizando servicios web, se implementó un sistema distribuido basado en la arquitectura cliente-servidor. En este apartado se detalla el proceso de desarrollo, desde la configuración del entorno hasta la implementación de las funcionalidades principales.

1. Configuración del entorno

El desarrollo del sistema requirió la instalación y configuración de diversas herramientas tanto para el servidor como para el cliente:

- **Backend (Servidor):**
  - Se creó un proyecto en **Spring Boot** utilizando Maven para la gestión de dependencias.
  - Se configuró el servidor de aplicaciones embebido en Spring Boot para manejar peticiones HTTP en el puerto 8080.
  - Se definieron los paquetes correspondientes a controladores, servicios, repositorios y modelos.
  - Se estableció la conexión con **MySQL**, configurando las credenciales en `application.properties`.
  - Se utilizó **JPA (Java Persistence API)** para la manipulación de datos en la base de datos.
- **Base de Datos:**
  - Se creó la base de datos en MySQL con una estructura que incluye tablas para los jugadores y los enemigos.
  - Se definieron las entidades correspondientes en Java con anotaciones de **JPA** para mapearlas correctamente a la base de datos.



- **Cliente:**
  - Se desarrolló un cliente en Java que permite a los jugadores interactuar con el servidor a través de peticiones HTTP utilizando `HttpClient`.
  - Se diseñó un menú de comandos para que el usuario pueda realizar acciones dentro del juego de forma interactiva.

```java
public class Client {
    private static HttpRequest request;  10 usages
    private static HttpResponse<String> response;  2 usages
    private static String currentUser;  6 usages

    public static void main(String[] args) throws IOException, InterruptedException {
        HttpClient httpClient = HttpClient.newHttpClient();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Puedes comenzar a utilizar comandos:");
        while (true){
            System.out.print("Comando: ");
            String command = scanner.nextLine();
            String[] parts = command.split( regex: " ");
            switch (parts[0]){
                case "CHANGE_USER":
                    System.out.print("Ingrese el nombre de usuario: ");
                    currentUser = scanner.nextLine();
                    System.out.println("Ahora estás controlando a: " + currentUser);
                    request = null;
                    break;
                case "PRUEBA":
                    request = HttpRequest.newBuilder()
                            .uri(URI.create("http://localhost:8080/game/saludo"))
                            .GET()
                            .build();
                    break;
                case "REGISTER":
```

```java
                default:
                    System.out.print("Comando no reconocido");
                    request = null;
                    break;
            }
            if(request!=null){
                response = httpClient.send(request,HttpResponse.BodyHandlers.ofString());
                System.out.println("Respuesta del servidor: " + response.body());
            }
        }
    }
}
```

2. Implementación del servidor web

El servidor web fue implementado utilizando **Spring Boot** y sigue la arquitectura REST. Se desarrollaron los siguientes componentes:

- **Modelo de Datos:**
  Se crearon las clases `Player` y `Enemy` para representar los elementos del juego. Estas clases fueron anotadas con `@Entity` y configuradas para su persistencia en la base de datos.

```java
package model;

import jakarta.persistence.*;
import lombok.*;

@Entity  8 usages
@Table(name = "Jugador")
public class Player {

    @Id  1 usage
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false)
    private Long id;
    @Column(name = "nombre", length=50, nullable = false)  3 usages
    private String name;
    @Column(nullable = false)  3 usages
    private int posX;
    @Column(nullable = false)  3 usages
    private int posY;

    public Player() {}  no usages

    public Player(String username) {  1 usage
        this.name = username;
        this.posX = 0;
        this.posY = 0;
    }

```

- **Repositorio (DAO):**
  Se definieron interfaces que extienden `JpaRepository` para interactuar con la base de datos sin necesidad de escribir consultas SQL manualmente.

```java
package model;

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

import java.util.Optional;

@Repository  3 usages
public interface PlayerRepository extends JpaRepository<Player,Long> {
    Optional<Player> findByName(String username);  3 usages
    long deleteByName(String username);  1 usage
}

```

- **Servicio:**
  Se implementó la lógica de negocio en una clase `PlayerService`, que maneja la creación, actualización y consulta de jugadores.

```java
@Service  2 usages
public class PlayerService {

    private final PlayerRepository playerRepository;  8 usages
    private final EnemyRepository enemyRepository;  4 usages

    public PlayerService (PlayerRepository playerRepository, EnemyRepository enemyRepository){  no usag
        this.playerRepository = playerRepository;
        this.enemyRepository = enemyRepository;
    }

    /*
    public List<Player> getAllPlayers() {
        return playerRepository.findAll();
    }*/


    public Player getPlayerByName(String username) {  1 usage
        Optional<Player> playerOpt = playerRepository.findByName(username);
        return playerOpt.orElse( other: null);
    }

    @Transactional  1 usage
    public String addPlayer(Player player) {
        if(player.getName()==null || player.getName().isEmpty()){
            return "Es posible que el nombre no se esté recibiendo correctamente";
        }
```

- **Controlador REST:**
  Se definieron endpoints en `PlayerController` para manejar las solicitudes HTTP. Los métodos principales incluyen:
  - **Registro de jugadores (`POST /game/addPlayer/{username}`)**
  - **Movimiento de jugadores (`PUT /game/move/{username}`)**
  - **Ataque a enemigos (`POST /game/attack/{username}`)**
  - **Consulta de jugadores (`GET /game/player/{username}`)**

```java
@RestController  no usages
@RequestMapping("/game")
public class PlayerController {
    private final PlayerService playerService;  7 usages

    public PlayerController(PlayerService playerService) {  no usages
        this.playerService = playerService;
    }

    @GetMapping("/saludo")  no usages
    public String welcome() {
        return "Saludos";
    }

    @GetMapping("/checkUser/{username}")  no usages
    public boolean existsUser(@PathVariable String username) {
        return playerService.getPlayerByName(username) != null;
    }

    @PostMapping("/addPlayer/{username}")  no usages
    public String addPlayer(@PathVariable String username) {
        Player newPlayer = new Player(username);
        System.out.println("newPlayer: " + newPlayer.getName());
        return playerService.addPlayer(newPlayer);
    }

```

3. Implementación del cliente

El cliente se diseñó para permitir la interacción con el servidor a través de comandos ingresados por el usuario. Se implementaron las siguientes funciones:

- **Registro de jugadores:** El usuario puede registrarse enviando una solicitud `POST` con su nombre.
- **Movimiento:** El usuario puede moverse por el mapa enviando una solicitud `PUT` con las nuevas coordenadas.
- **Ataque:** Se implementó una función que permite a un jugador atacar a un enemigo cercano mediante una solicitud `POST`.
- **Consulta de estado:** El usuario puede obtener información sobre su estado actual mediante una solicitud `GET`.

4. Pruebas y validaciones

Una vez implementados el servidor y el cliente, se realizaron diversas pruebas para verificar el correcto funcionamiento del sistema:

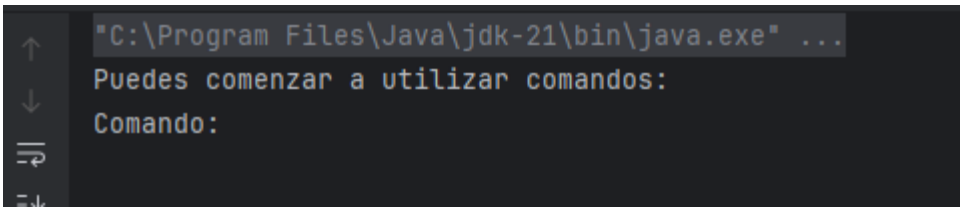
- **Pruebas de conectividad:** Se verificó que el cliente pudiera enviar y recibir información del servidor correctamente.
- **Validaciones de entrada:** Se probaron escenarios en los que los jugadores intentaban moverse a posiciones inválidas o atacar sin estar cerca de un enemigo.
- **Persistencia de datos:** Se confirmó que los datos de los jugadores y sus movimientos se almacenaban correctamente en la base de datos.
- **Pruebas concurrentes:** Se simularon múltiples clientes interactuando simultáneamente con el servidor para evaluar su rendimiento y estabilidad.

Resultados

Primero se inicia el programa del servidor que usa Spring Boot para iniciar un servidor local en el puesto 8080 a través de TomCat

```
  : HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA p
n : Initialized JPA EntityManagerFactory for persistence unit 'default'
n : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed du
  : Tomcat started on port 8080 (http) with context path '/'
  : Started PractóServWebApplication in 7.135 seconds (process running for 7.809)
```

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.

```
Puedes comenzar a utilizar comandos:
Comando: REGISTER
Ingrese el nombre de usuario del nuevo jugador:Paco
Respuesta del servidor: Jugador añadido exitosamente
Comando: CHANGE_USER
Ingrese el nombre de usuario: Paco
Ahora estás controlando a: Paco
Comando:
```

MOVE X Y

Mueve al usuario a las coordenadas seleccionadas

```
Comando: MOVE 4 6
Respuesta del servidor: Paco fue movido a (4,6) de manera exitosa
Comando:
```

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

El desarrollo de esta práctica permitió comprender y aplicar los principios fundamentales de los sistemas distribuidos mediante la implementación de un servicio web. A través de este ejercicio, se pudo observar cómo la arquitectura cliente-servidor facilita la comunicación entre múltiples usuarios, gestionando eficientemente la interacción entre ellos y manteniendo la coherencia en la información del juego. El uso de **Spring Boot** simplificó significativamente el proceso de implementación del servidor, proporcionando herramientas que agilizan la gestión de peticiones HTTP, la persistencia de datos y la comunicación con la base de datos relacional.

Durante la simulación del juego multijugador, se evidenció la importancia de diseñar un sistema capaz de manejar múltiples solicitudes simultáneas sin comprometer el rendimiento. A través del uso de JSON como formato de intercambio de datos, se logró una comunicación eficiente y liviana entre el cliente y el servidor, asegurando que la información fuera transmitida y procesada correctamente en cada interacción. Además, la implementación de un cliente en **Java** permitió evaluar la correcta recepción y procesamiento de las solicitudes, garantizando así la funcionalidad esperada del sistema.

A lo largo de la evolución de los sistemas distribuidos, han surgido distintas tecnologías para facilitar la comunicación entre aplicaciones. En sus inicios, soluciones como **sockets** fueron ampliamente utilizadas, permitiendo la conexión directa entre dos puntos de una red. Sin embargo, su uso conllevaba una complejidad considerable, ya que cada conexión debía ser gestionada manualmente y las aplicaciones debían compartir protocolos específicos. Posteriormente, tecnologías como **RMI** y **CORBA** introdujeron el concepto de objetos distribuidos, facilitando la invocación remota de métodos en diferentes sistemas. Aunque estas soluciones representaron un avance significativo, su implementación era compleja y solía requerir configuraciones específicas que limitaban su interoperabilidad entre distintas plataformas.

Con la llegada de los **servicios web**, se logró una solución mucho más flexible y escalable. A diferencia de las tecnologías anteriores, los servicios web están basados en estándares abiertos como **HTTP** y **JSON**, lo que permite su integración con prácticamente cualquier sistema, independientemente del lenguaje de programación o del entorno en el que se implemente. Además, su arquitectura sin estado facilita la escalabilidad, permitiendo que múltiples clientes interactúen con el servidor sin necesidad de mantener conexiones persistentes.

En este contexto, la práctica permitió demostrar que los servicios web representan una alternativa eficiente y moderna para la implementación de sistemas distribuidos. Su capacidad de integración con diversas tecnologías, su facilidad de mantenimiento y su escalabilidad los posicionan como la opción preferida en el desarrollo de aplicaciones actuales. Frente a las soluciones anteriores, los servicios web no solo han simplificado la comunicación entre aplicaciones, sino que también han permitido la creación de sistemas más robustos y adaptables a las necesidades del mundo digital actual.