



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

- Introducción a patrones de diseño (DAO, Singleton, Factory, etc.).
- Comunicación entre microservicios (REST, Feign Client, WebClient).
- Gestión de APIs con Swagger/OpenAPI.
- Integración con servicios externos (RESTful APIs).
- Manejo de errores y resiliencia (Retry, Circuit Breaker).

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Introducción a patrones de diseño

¿Qué son los patrones de diseño?

Los **patrones de diseño** son **soluciones reutilizables a problemas comunes de diseño de software**. No son código exacto ni librerías que se instalan, sino **plantillas de buenas prácticas** que ayudan a estructurar mejor el código para que sea:

- **Reutilizable**
- **Mantenible**
- **Desacoplado**
- **Fácil de entender y escalar**

Se hicieron populares gracias al libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (conocido como el libro de los "Gang of Four").

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Introducción a patrones de diseño

¿Por qué son importantes?

- **Facilitan el trabajo en equipo:** porque ofrecen un lenguaje común. Si dices “DAO” o “Singleton”, otro desarrollador sabe a qué te refieres.
- **Evitan reinventar la rueda:** aplican soluciones ya probadas.
- **Mejoran la calidad del diseño:** permiten mayor flexibilidad ante cambios.

Clasificación general

Los patrones se dividen en 3 grandes grupos:

Tipo	Qué resuelven	Ejemplo
Creacionales	Cómo crear objetos de forma flexible	Factory, Singleton
Estructurales	Cómo componer objetos en estructuras complejas	Adapter, Decorator
Comportamiento	Cómo se comunican objetos entre sí	Observer, Strategy

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Introducción a patrones de diseño

Patrones comunes en entornos Java

Patrón	¿Qué hace?	¿Para qué sirve en Java?
DAO (Data Access Object)	Separa la lógica de acceso a datos de la lógica de negocio	Ideal para mantener desacoplamiento entre capas
Singleton	Asegura que una clase tenga una sola instancia	Útil en clases compartidas como servicios de configuración o loggers
Factory	Crea objetos sin exponer la lógica de creación	Permite que la lógica decida qué clase crear según configuración o contexto
Service	Agrupar lógica de negocio reutilizable	Es la capa central en una arquitectura con Spring Boot (@Service)

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Introducción a patrones de diseño

Ejemplo visual con DAO

```
// Interfaz DAO
public interface ProductoDao {
    Producto findById(Long id);
    void save(Producto p);
}

// Implementación DAO con JPA
public class ProductoDaoImpl implements ProductoDao {
    @PersistenceContext
    private EntityManager em;

    public Producto findById(Long id) {
        return em.find(Producto.class, id);
    }

    public void save(Producto p) {
        em.persist(p);
    }
}
```

Así, si algún día cambias de base de datos (por ejemplo, de MySQL a MongoDB), no tienes que tocar toda tu lógica de negocio, solo tu DAO.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Introducción a patrones de diseño

Singleton

```
public class Logger {  
    private static final Logger instance = new Logger();  
  
    private Logger() {} // constructor privado  
  
    public static Logger getInstance() {  
        return instance;  
    }  
  
    public void log(String mensaje) {  
        System.out.println("LOG: " + mensaje);  
    }  
}
```

Asegurar una única instancia compartida en toda la aplicación.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Introducción a patrones de diseño

Factory

```
public interface ClienteRest {  
    String getTipo();  
}  
  
public class ClienteRESTExterno implements ClienteRest {  
    public String getTipo() { return "Cliente Externo"; }  
}  
  
public class ClienteRESTInterno implements ClienteRest {  
    public String getTipo() { return "Cliente Interno"; }  
}
```

java

```
// La fábrica  
public class ClienteRestFactory {  
    public static ClienteRest crearCliente(String tipo) {  
        if ("externo".equals(tipo)) {  
            return new ClienteRESTExterno();  
        } else {  
            return new ClienteRESTInterno();  
        }  
    }  
}
```

Crear objetos sin exponer su lógica interna.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Introducción a patrones de diseño

Service

```
@Service
public class ProductoService {

    private final ProductoDao productoDao;

    public ProductoService(ProductoDao productoDao) {
        this.productoDao = productoDao;
    }

    public Producto obtenerProducto(Long id) {
        return productoDao.findById(id);
    }

    public void guardarProducto(Producto producto) {
        productoDao.save(producto);
    }
}
```

```
// Controlador que llama al service
@RestController
@RequestMapping("/productos")
public class ProductoController {

    private final ProductoService productoService;

    public ProductoController(ProductoService productoService) {
        this.productoService = productoService;
    }

    @GetMapping("/{id}")
    public Producto obtener(@PathVariable Long id) {
        return productoService.obtenerProducto(id);
    }
}
```

Centralizar y encapsular la lógica de negocio.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Introducción a patrones de diseño

Ejemplo:

<https://github.com/Joselota/Relatorias/tree/main/CursoJavaAvanzado/src>

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Comunicación entre microservicios

¿Qué es la comunicación entre microservicios?

En una arquitectura de **microservicios**, cada módulo del sistema (ej. usuarios, productos, pedidos) se implementa como un **servicio independiente** que:

- Tiene su propia lógica
- Se despliega por separado
- Y necesita **comunicarse con otros microservicios** para realizar tareas completas

Esa **comunicación** se puede dar de varias formas, siendo la más común:

Comunicación vía HTTP/REST

- Los microservicios se comunican **como si fueran APIs web**, usando el protocolo HTTP.
- Envían y reciben datos en formato **JSON**.
- Ejemplo: Servicio A hace una petición GET <http://servicio-b/api/productos/1>.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Comunicación entre microservicios

¿Cómo se implementa en Java?

Hay varias formas de implementar esa comunicación HTTP entre microservicios. Aquí te explico las tres principales en entornos Spring Boot:

1. REST con `RestTemplate` (obsoleto)

- Fue durante años la forma estándar de hacer llamadas HTTP desde Java.
- Ya no se recomienda usarlo para nuevas aplicaciones, pero sigue presente en muchos sistemas.

```
1 RestTemplate restTemplate = new RestTemplate();  
2 Producto p = restTemplate.getForObject("http://servicio/api/productos/1", Producto.class);  
3
```

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Comunicación entre microservicios

2. Feign Client – cliente declarativo

Concepto:

Feign te permite crear una interfaz Java que “representa” un microservicio externo. Spring se encarga del resto.

¿Qué lo hace especial?

- No escribes código HTTP a mano.
- Es ideal para **llamadas internas entre microservicios**, especialmente si usas **Spring Cloud**.

```
@FeignClient(name = "producto-service")
public interface ProductoClient {
    @GetMapping("/productos/{id}")
    Producto getProducto(@PathVariable Long id);
}
```

Es como decirle a Spring: “Hazme una clase que sepa hablar con el servicio de productos”.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Comunicación entre microservicios

3. WebClient – cliente reactivo y moderno

Concepto:

WebClient es el sucesor de RestTemplate, diseñado para aplicaciones **reactivas y asíncronas**.

¿Qué lo hace especial?

- No bloquea el hilo mientras espera la respuesta (ideal para sistemas de alto rendimiento).
- Devuelve Mono o Flux (objetos reactivos de Project Reactor).
- Es más **manual y flexible** que Feign.

```
WebClient client = WebClient.create("http://localhost:8080");
client.get()
    .uri("/productos")
    .retrieve()
    .bodyToMono(Producto.class)
    .subscribe(System.out::println);
```

Es como tener un cliente HTTP programable, más potente, pero que tú controlas línea a línea.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Comunicación entre microservicios

¿Cuál usar?

Situación	Mejor opción
Llamadas simples entre microservicios	✓ Feign
Integraciones con APIs externas complejas	✓ WebClient
Requiere control sobre headers, errores	✓ WebClient
Uso de Spring Cloud y Eureka	✓ Feign
Necesitas llamadas reactivas y no bloqueantes	✓ WebClient

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Comunicación entre microservicios

Resumen


- **Todos permiten que un microservicio hable con otro por HTTP.**
- **Feign** es más declarativo y automático.
- **WebClient** es más flexible y potente para escenarios complejos o reactividad.
- **RestTemplate** está en desuso.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Gestión de APIs con Swagger/OpenAPI

 Objetivo:

Documentar APIs REST para facilitar la interoperabilidad y exploración de endpoints.

 ¿Qué es OpenAPI (Swagger)?

Es una especificación para describir APIs RESTful de forma estandarizada.

Springdoc/OpenAPI permite generar la documentación automáticamente.



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Gestión de APIs con Swagger/OpenAPI

📌 Integración básica con Spring Boot:

```
<dependency>  
  <groupId>org.springdoc</groupId>  
  <artifactId>springdoc-openapi-ui</artifactId>  
  <version>1.7.0</version>  
</dependency>
```

Con esto accedes a:

<http://localhost:8080/swagger-ui.html>



Beneficios:

- Visualización interactiva de endpoints
- Pruebas sin Postman
- Auto-documentación de DTOs, parámetros, errores



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Gestión de APIs con Swagger/OpenAPI

¿Qué es OpenAPI / Swagger?

OpenAPI (anteriormente conocido como Swagger) es un **estándar** para describir APIs RESTful en un formato **legible por humanos y por máquinas** (YAML o JSON). Permite:

- Documentar todos los endpoints de forma estructurada
- Probar las APIs sin necesidad de Postman
- Generar código cliente o servidor automáticamente

Swagger UI es una herramienta visual que lee esa especificación OpenAPI y te permite:

- Explorar todos los endpoints de tu API
- Ejecutar peticiones directamente desde el navegador
- Ver ejemplos, parámetros, respuestas y errores

👉 En proyectos con **Spring Boot**, usamos **Springdoc OpenAPI** para generar esta documentación automáticamente a partir de las anotaciones `@RestController`, `@GetMapping`, `@RequestBody`, etc.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java



Gestión de APIs con Swagger/OpenAPI

Ejemplo:

<https://github.com/Joselota/Relatorias/tree/main/CursoJavaAvanzado/swagger-demo>

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Gestión de APIs con Swagger/OpenAPI

The screenshot displays the Swagger UI for an API. At the top, the Swagger logo is visible, along with the text 'Supported by SMARTBEAR'. The API path is '/v3/api-docs', and there is an 'Explore' button. Below this, the 'OpenAPI definition' is shown, with a version 'v0' and 'OAS 3.1' label. The 'Servers' section shows a dropdown menu with the selected server 'http://localhost:8080 - Generated server url'. The main section, titled 'Productos' with the description 'API para gestión de productos', lists five API endpoints:

- GET** `/productos/{id}` Obtener producto por ID
- PUT** `/productos/{id}` Actualizar producto
- DELETE** `/productos/{id}` Eliminar producto
- GET** `/productos` Listar todos los productos
- POST** `/productos` Crear nuevo producto

Below the endpoints, the 'Schemas' section is visible, showing a schema for 'Producto' with the type 'object'.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Integración con servicios externos (RESTful APIs).

¿Qué significa integrar con un servicio externo?

Cuando hablamos de integración con servicios externos en el contexto de microservicios o aplicaciones Java, nos referimos a:

Consumir datos o funcionalidades expuestas por un sistema ajeno a nuestra aplicación, como APIs públicas, socios B2B o sistemas internos desacoplados.

Ejemplos comunes:

- Consultar clima desde `api.weather.com`
- Validar RUT en un servicio del Registro Civil
- Obtener tasas de cambio desde un API bancaria
- Usar una API de pagos como Transbank, Stripe, MercadoPago



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Integración con servicios externos (RESTful APIs).

¿Cómo se implementa en Java con Spring Boot?

Opción moderna: **WebClient**

WebClient Example ▾

```
WebClient client = WebClient.create("https://api.externo.com");

Producto producto = client.get()
    .uri("/productos/1")
    .retrieve()
    .bodyToMono(Producto.class)
    .block();
```

WebClient es parte de **Spring WebFlux**

Soporta llamadas reactivas y asíncronicas







Es útil cuando quieres controlar headers, autenticación, timeouts



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Integración con servicios externos (RESTful APIs).

Buenas prácticas para consumir APIs externas

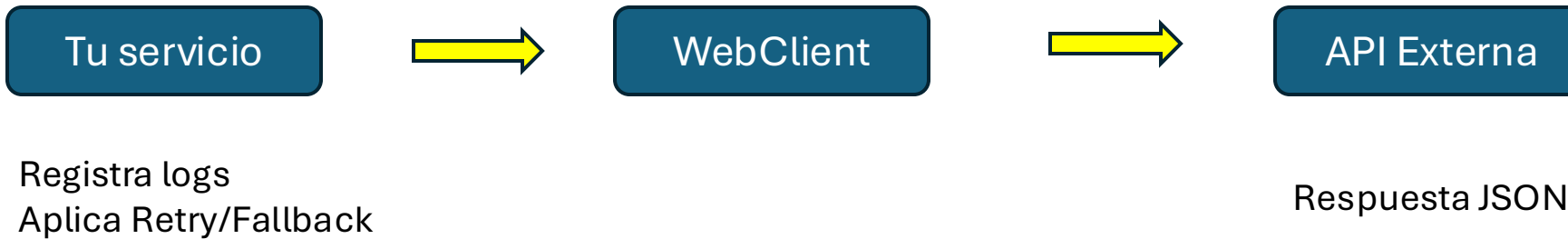
Tema	Buena práctica
 Autenticación	Usar API Key, OAuth2 o JWT
 Timeouts	Definir tiempo de espera (evita bloqueos)
 Retry	Reintentar automáticamente en fallos leves
 Circuit Breaker	Evitar saturar el sistema si la API externa está caída
 Logs y trazabilidad	Registrar todas las llamadas (idempotencia)
 Encapsulamiento	Crear una clase de servicio dedicada (ej. ApiClienteExternoService)



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Integración con servicios externos (RESTful APIs).

Flujo de integración con una API externa



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

○ Manejo de errores y resiliencia (Retry, Circuit Breaker).

¿Por qué es clave el manejo de errores y resiliencia?

En un sistema distribuido (microservicios), **los fallos son inevitables**:

- APIs externas pueden estar caídas
- Redes lentas o con latencia
- Timeout en respuestas
- Respuestas erróneas o mal formadas

Tu sistema **no debe caerse por esto**, sino **recuperarse, aislar el error o manejarlo con elegancia**.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

○ Manejo de errores y resiliencia (Retry, Circuit Breaker).

Principales patrones de resiliencia

Patrón	¿Qué hace?
Retry	Reintenta una llamada fallida automáticamente (por un tiempo o número máximo).
Circuit Breaker	Evita saturar servicios que están fallando; abre el “circuito” y deja de llamar por un tiempo.
Fallback	Devuelve una respuesta alternativa en caso de fallo.
Timeout	Cancela una petición si tarda demasiado.

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Manejo de errores y resiliencia (Retry, Circuit Breaker).

¿Cómo se aplica esto en Spring Boot?

Usamos la librería **Resilience4j**, que se integra fácilmente con WebClient, Feign y otros.

Ejemplo con `@Retry` y `fallback`

1. Agrega la dependencia

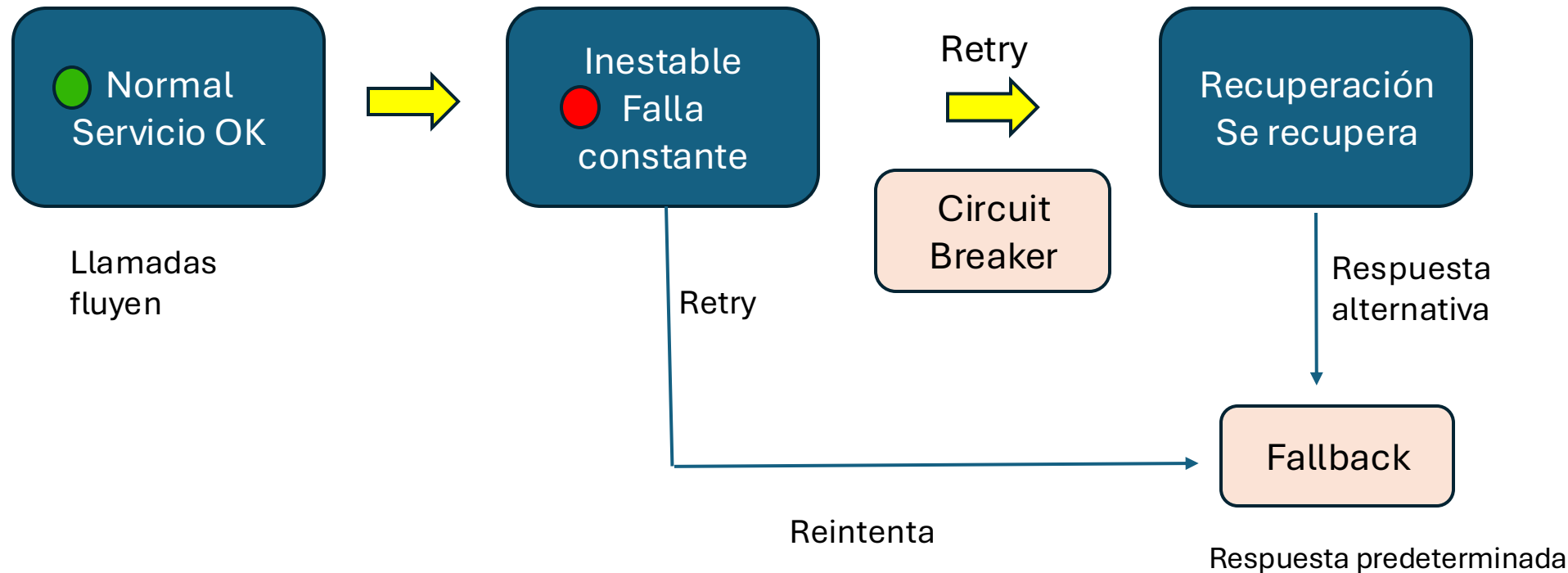
```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot3</artifactId>
  <version>2.1.0</version>
</dependency>
```

<https://github.com/Joselota/Relatorias/tree/main/CursoJavaAvanzado/resilience4j-demo>

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Manejo de errores y resiliencia (Retry, Circuit Breaker).

Flujo visual del manejo de resiliencia en sistemas distribuidos usando Retry, Circuit Breaker y Fallback



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Manejo de errores y resiliencia (Retry, Circuit Breaker).

Buenas Prácticas de Resiliencia

1. Usa Circuit Breaker en puntos críticos

- **¿Por qué?** Previene la saturación de servicios cuando un sistema está fallando.
- **Tip:** Configura un *threshold* de errores y un *timeout* de reapertura.
- *Ejemplo:* `@CircuitBreaker(name = "catalogService", fallbackMethod = "fallbackCatalog")`

2. Implementa Retry con backoff


- **¿Por qué?** Reintentar inmediatamente puede agravar el problema.
- **Tip:** Usa `@Retry` con `waitDuration` progresivo (exponencial o fijo).
- *Ejemplo:* `@Retry(name = "retryService", fallbackMethod = "fallback")`

Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

Manejo de errores y resiliencia (Retry, Circuit Breaker).

Buenas Prácticas de Resiliencia

3. Agrega lógica de Fallback clara y útil

- **¿Por qué?** Brinda una respuesta alternativa ante fallas.
- **Tip:** El fallback debe evitar propagar la excepción original y entregar un valor seguro.
-  *Ejemplo:* devolver lista vacía, mensaje genérico, datos cacheados.

4. Evita fallback silencioso

- **¿Por qué?** Podrías no enterarte de un error grave si no logueas la excepción.
- **Tip:** Loguea dentro del fallback la causa (`Throwable ex`) al menos en nivel WARN.



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

○ Manejo de errores y resiliencia (Retry, Circuit Breaker).

Buenas Prácticas de Resiliencia

5. Observabilidad: monitorea tus políticas

- **¿Por qué?** Necesitas saber cuándo se activan circuit breakers y cuántos retries se están ejecutando.
- **Tip:** Usa Actuator + Micrometer para exponer métricas de Resilience4j a Prometheus / Grafana.

6. Haz pruebas de resiliencia

- **¿Por qué?** No sabrás si tu app resiste hasta que lo compruebes.
- **Tip:** Simula fallas de red, APIs lentas o caídas de servicios (Chaos Engineering).

7. No uses Retry en operaciones no idempotentes

- **¿Por qué?** Repetir una operación como un pago puede causar errores financieros.
- **Tip:** Asegúrate de que el endpoint externo soporte reintentos seguros.



Módulo 5.- Patrones de Integración e Interoperabilidad en Entornos Java

○ Manejo de errores y resiliencia (Retry, Circuit Breaker).

Buenas Prácticas de Resiliencia

Herramientas recomendadas

- Resilience4j: Circuit Breaker, Retry, RateLimiter, Bulkhead
- Spring Boot Actuator: monitoreo de endpoints
- Sleuth + Zipkin o OpenTelemetry: trazabilidad distribuida

