Universidad de Las Américas
Facultad de Ingenierías y Ciencias Aplicadas
Ingeniería De Software **Exámen Práctico**

1. **DATOS DEL ALUMNO:**
   Ariel Raura

2. **TEMA DE LA PRÁCTICA:**
   Examen Práctico Progreso 2.

3. **OBJETIVO DE LA ACTIVIDAD:**
   Diseñar e implementar una solución de integración funcional utilizando los conocimientos adquiridos durante el curso.

4. **DESARROLLO:**

   Identificación del problema:

   La universidad desea construir una nueva plataforma integrada para atender solicitudes académicas de los estudiantes (solicitudes de certificados, legalizaciones, homologaciones y equivalencias). Actualmente, existen 3 sistemas independientes:
   1. Sistema Académico (REST API – gestionado internamente)

   2. Sistema de Certificación (SOAP – externo, expuesto por un proveedor estatal)

   3. Sistema de Seguridad y Roles (interno, usa tokens JWT)

   Objetivos específicos

   1. Integrar servicios REST y SOAP en una solución funcional.

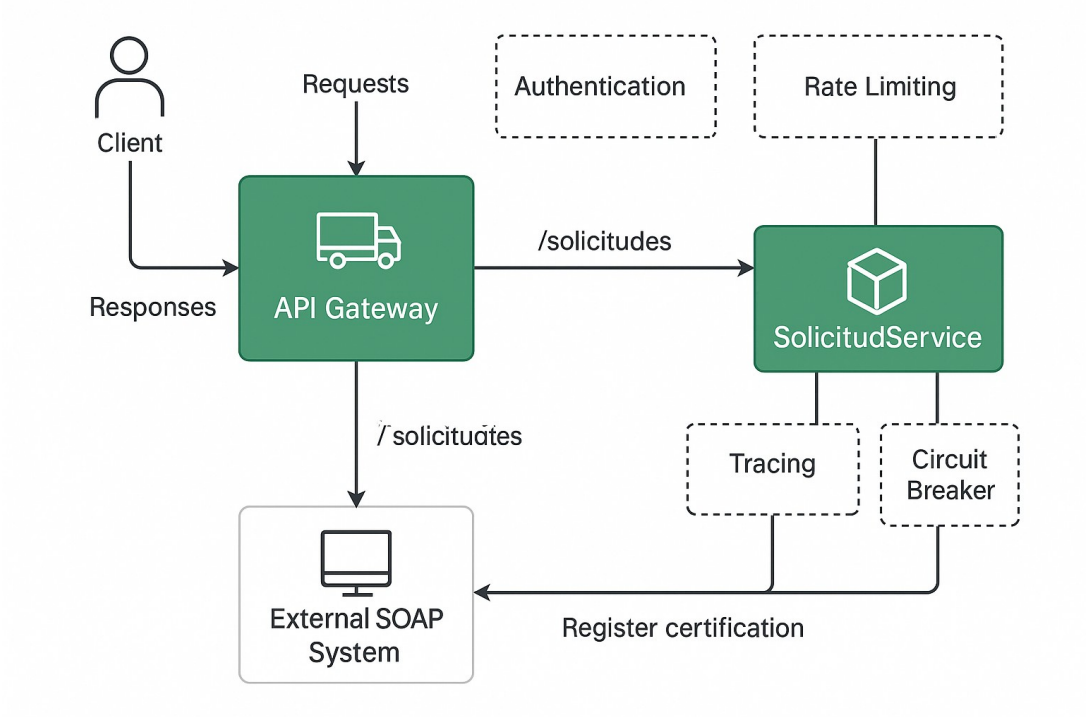
   2. Exponer todos los servicios a través de un API Gateway.

   3. Diseñar la solución considerando aspectos de trazabilidad, seguridad y resiliencia.

   4. Aplicar patrones como Circuit Breaking y Retry usando conceptos de Service Mesh (en forma de diseño o pseudocódigo si no se puede desplegar, con una evaluación menor que si se lo implementa).

Requerimientos

1. Diseño de arquitectura



2. Diseño de arquitectura

Para nuestro servicio decidí usar un proyecto construido en NodeJs con Express.

Link al repositorio GIT

El proyecto completo está disponible en el siguiente repositorio público de GitHub:

https://github.com/Ariel454/solicitud-service.git

3. Exposición del servicio a través del API Gateway

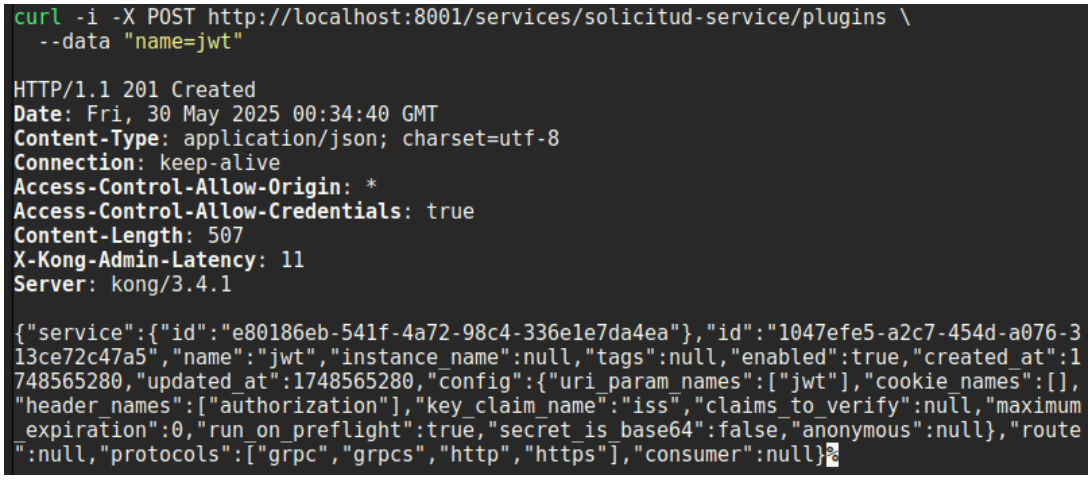
Usa una herramienta como WSO2 API Manager, Kong Gateway, o la de tu preferencia, o a su vez un mock si el entorno no lo permite.

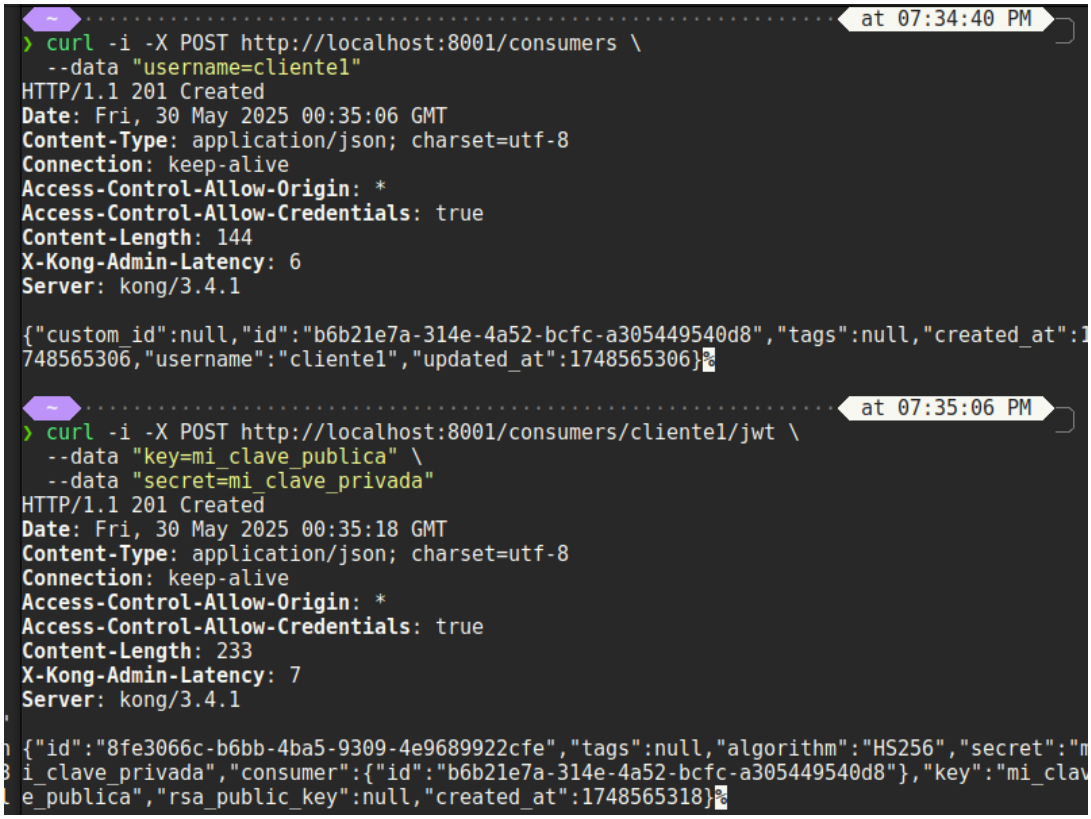
- Implementación instancia kong con docker:

- Registro del servicio rest en el gateway:

- Registra del endpoint /solicitudes

- Aplica una política de seguridad por token (API Key o JWT).





- Una política de rate limiting.

```
~ ····· at 07:35:18 PM
❯ curl -i -X POST http://localhost:8001/services/solicitud-service/plugins \
  --data "name=rate-limiting" \
  --data "config.minute=10" \
  --data "config.policy=local"

HTTP/1.1 201 Created
Date: Fri, 30 May 2025 00:35:49 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Content-Length: 760
X-Kong-Admin-Latency: 8
Server: kong/3.4.1

{"service":{"id":"e80186eb-541f-4a72-98c4-336e1e7da4ea"},"id":"b51a1219-0821-483b-a877-0
24680a65a2a","name":"rate-limiting","instance_name":null,"tags":null,"enabled":true,"cre
ated_at":1748565349,"updated_at":1748565349,"config":{"header_name":null,"path":null,"er
ror_code":429,"error_message":"API rate limit exceeded","sync_rate":-1,"second":null,"mi
nute":10,"hour":null,"day":null,"month":null,"year":null,"redis_port":6379,"redis_server
_name":null,"redis_timeout":2000,"redis_ssl":false,"redis_ssl_verify":false,"redis_usern
ame":null,"redis_host":null,"policy":"local","redis_password":null,"redis_database":0,"f
ault_tolerant":true,"limit_by":"consumer","hide_client_headers":false},"route":null,"pro
tocols":["grpc","grpcs","http","https"],"consumer":null}%
```

- Resultados:

```
~ ·····
❯ curl http://localhost:8000/solicitudes/123

{"message":"No autorizado"}%
```

```
~/Doc/U/9/INTEG/solicitud-service on main ?4 ····· at 07:38:24 PM
❯ node -e "console.log(require('jsonwebtoken').sign(
  { iss: 'mi_clave_publica', sub: 'usuario123', exp: Math.floor(Date.now()/1000) + 60*60
 },
  'mi_clave_privada'
));"

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJtaV9jbGF2ZV9wdWJsaWNhIiwic3ViIjoidXN1YXJ
pbzEyMyIsImV4cCI6MTc0ODU2OTEwNSwiaWF0IjoxNzQ4NTY1NTA1fQ.5ouJKeQkNe1eKYC385bZkb_158LrAzH8
pq1Zu-hgz_k
```


curl -i -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJtaV9
jbGF2ZV9wdWJsaWNhIiwic3ViIjoidXN1YXJpbzEyMyIsImV4cCI6MTc0ODU2OTEwNSwiaWF0IjoxNzQ4NTY1NTA
1fQ.5ouJKeQkNeleKYC385bZkb_158LrAzH8pq1Zu-hgz_k" \
http://localhost:8000/solicitudes/123
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 33
Connection: keep-alive
X-RateLimit-Remaining-Minute: 9
RateLimit-Limit: 10
RateLimit-Remaining: 9
RateLimit-Reset: 53
X-RateLimit-Limit-Minute: 10
X-Powered-By: Express
ETag: W/"21-hL562QJwikTjAOqL99rcTOY+s9g"
Date: Fri, 30 May 2025 00:40:07 GMT
X-Kong-Upstream-Latency: 7
X-Kong-Proxy-Latency: 1
Via: kong/3.4.1
{"id":"123","estado":"procesado"}

```
Petición 9:
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 33
Connection: keep-alive
X-RateLimit-Remaining-Minute: 1
RateLimit-Limit: 10
RateLimit-Remaining: 1
RateLimit-Reset: 19
X-RateLimit-Limit-Minute: 10
X-Powered-By: Express
ETag: W/"21-hL562QJwikTjAOqL99rcTOY+s9g"
Date: Fri, 30 May 2025 00:41:41 GMT
X-Kong-Upstream-Latency: 1
X-Kong-Proxy-Latency: 1
Via: kong/3.4.1

{"id":"123","estado":"procesado"}
--------------------

Petición 10:
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 33
Connection: keep-alive
X-RateLimit-Remaining-Minute: 0
RateLimit-Limit: 10
RateLimit-Remaining: 0
RateLimit-Reset: 19
X-RateLimit-Limit-Minute: 10
X-Powered-By: Express
ETag: W/"21-hL562QJwikTjAOqL99rcTOY+s9g"
Date: Fri, 30 May 2025 00:41:41 GMT
X-Kong-Upstream-Latency: 1
X-Kong-Proxy-Latency: 0
Via: kong/3.4.1

{"id":"123","estado":"procesado"}
--------------------

Petición 11:
HTTP/1.1 429 Too Many Requests
Date: Fri, 30 May 2025 00:41:41 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
X-RateLimit-Remaining-Minute: 0
RateLimit-Limit: 10
RateLimit-Remaining: 0
X-RateLimit-Limit-Minute: 10
RateLimit-Reset: 19
Retry-After: 19
Content-Length: 41
X-Kong-Response-Latency: 1
Server: kong/3.4.1

{
  "message":"API rate limit exceeded"
}
--------------------
```

4. Implementación de Circuit Breaking y Retry

• Define una configuración (real o pseudocódigo YAML) para aplicar:

• Retry automático al servicio SOAP (máximo 2 intentos).

• Circuit Breaker si hay más de 3 fallos en 60 segundos.

• Puedes usar herramientas como Istio o Spring Cloud Gateway

En nuestro `application.yml` definimos un gateway de Spring Cloud llamado `gateway-cb-retry` con una ruta `soap_route` que redirige todas las peticiones que empiecen con `/soap/**` al servicio SOAP local (`http://localhost:9000`). En los filtros de esa ruta hemos encadenado primero `Retry` con `retries: 2`, `series: SERVER_ERROR` y un "backoff" que espera 1 segundo antes del primer reintento y hasta 2 segundos antes del segundo, de modo que cualquier error 5xx del SOAP provoca como máximo dos reintentos automáticos. A continuación, aplicamos `CircuitBreaker` bajo el nombre `soapCircuit`, cuyas políticas (definidas en `resilience4j.circuitbreaker.instances.soapCircuit`) usan una ventana deslizante de conteo (`slidingWindowSize: 3`) con umbral de fallo del 100 % y `waitDurationInOpenState: 60s`, de forma que si se acumulan tres fallos en un minuto el circuito se abre y todas las llamadas posteriores se redirigen a `fallbackUri: forward:/fallbackSoapResponse`, donde nuestro `FallbackController` devuelve un JSON con el mensaje "Servicio SOAP no disponible, intente más tarde". Esta configuración implementa exactamente el retry automático (dos intentos) y el circuit breaker tras tres fallos en sesenta segundos

```
# Puerto en el que arranca el Gateway
server:
port: 8080

spring:
application:
# Nombre de la aplicación, visible en logs y en registros de Spring Cloud
name: gateway-cb-retry

cloud:
gateway:
```

```
# Definición de rutas gestionadas por el Gateway
routes:
- id: soap_route
# URI del servicio SOAP al que redirigiremos (puede ser un mock local en el puerto 9000)
uri: http://localhost:9000
predicates:
- Path=/soap/** # Coincide con cualquier petición que comience con /soap/
filters:
# Filtro de reintentos automáticos
- name: Retry
args:
retries: 2 # Permitir hasta 2 reintentos adicionales tras el primer fallo
series: SERVER_ERROR # Solo volver a intentar si se recibe un error 5xx
backoff:
firstBackoff: 1s # Esperar 1 segundo antes del primer reintento
maxBackoff: 2s # Esperar hasta 2 segundos como máximo antes del segundo reintento
# Filtro de Circuit Breaker
- name: CircuitBreaker
args:
name: soapCircuit # Identificador del Circuit Breaker (coincide con la sección resilience4j)
fallbackUri: forward:/fallbackSoapResponse
# Cuando el circuito esté abierto (muchos fallos), redirige a /fallbackSoapResponse

- id: solicitud_route
# URI del microservicio SolicitudService (por ejemplo, en el puerto 3000)
uri: http://localhost:3000
predicates:
- Path=/solicitudes/** # Coincide con cualquier petición que comience con /solicitudes/
filters:
# Solo Circuit Breaker para este microservicio (sin retry)
- name: CircuitBreaker
args:
name: solicitudCircuit # Identificador del Circuit Breaker para SolicitudService
fallbackUri: forward:/fallbackSolicitud
# Redirige a /fallbackSolicitud cuando el circuito se abra

redis:
# Configuración de Redis (opcional, solo si se usase para algo como rate limiting)
# Si no se utiliza Redis en el proyecto, este bloque se puede eliminar.
host: localhost
port: 6379

# Configuraciones para Resilience4j (gestiona los circuit breakers)
```

```
resilience4j:
circuitbreaker:
configs:
soap-only:
slidingWindowType: COUNT_BASED # Ventana deslizante basada en conteo de llamadas
slidingWindowSize: 3 # Seis llamadas en la “ventana” (tamaño 3 = 3 llamadas para
evaluar)
minimumNumberOfCalls: 3 # Mínimo de 3 llamadas necesarias para evaluar la tasa de
fallo
failureRateThreshold: 100 # Umbral del 100% de fallos para abrir el circuito
waitDurationInOpenState: 60s # El circuito permanecerá abierto durante 60 segundos
antes de permitir probar de nuevo
instances:
soapCircuit:
# Asigna la configuración “soap-only” al circuit breaker llamado soapCircuit
baseConfig: soap-only

logging:
level:
root: INFO # Nivel de logging general de la aplicación
org.springframework.cloud.gateway: DEBUG # Nivel DEBUG para Spring Cloud Gateway
(útil para ver detalles de filtros)
io.github.resilience4j: DEBUG # Nivel DEBUG para Resilience4j (para ver estados de circuit
breaker)

management:
endpoints:
web:
exposure:
# Exponer estos endpoints en el actuator para monitoreo: health, info, métricas y
prometheus
include: health,info,metrics,prometheus
```

```
package com.ejemplo.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
* Clase principal que arranca la aplicación Spring Boot
```

```
* con Spring Cloud Gateway y Resilience4j configurados.
*/
@SpringBootApplication
public class GatewayCbRetryApplication {

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

```
package com.ejemplo.gateway;

import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Mono;

/**
* Controlador que maneja las rutas de fallback cuando los Circuit Breakers
* detectan que el servicio está caído o no puede responder.
*/
@RestController
public class FallbackController {

/**
* Método llamado cuando el Circuit Breaker de soapCircuit está abierto
* o tras los reintentos fallidos al servicio SOAP. Devuelve un JSON con
* mensaje de servicio no disponible.
*/
@RequestMapping(
value = "/fallbackSoapResponse",
produces = MediaType.APPLICATION_JSON_VALUE
)
@ResponseStatus(HttpStatus.SERVICE_UNAVAILABLE) // HTTP 503
public Mono<String> soapFallback() {
// Aquí se define la respuesta estándar en caso de fallo prolongado del SOAP
return Mono.just("{\"message\":\"Servicio SOAP no disponible, intente más tarde\"}");
}

/**
```

```
* Método llamado cuando el Circuit Breaker de solicitudCircuit está abierto.
* Se ejecuta cuando el microservicio SolicitudService devuelve fallos
* o no responde. Devuelve un JSON con mensaje de servicio no disponible.
*/
@RequestMapping(
value = "/fallbackSolicitud",
produces = MediaType.APPLICATION_JSON_VALUE
)
@ResponseStatus(HttpStatus.SERVICE_UNAVAILABLE) // HTTP 503
public Mono<String> solicitudFallback() {
// Respuesta genérica cuando SolicitudService no está disponible
return Mono.just("{\"message\":\"Servicio SolicitudService no disponible en este
momento\"}");
}
}
```

El proyecto springcloud inicializado se encuentra en el siguiente repositorio:

https://github.com/Ariel454/gateway-cb-retry.git

Resultados:

Retry automático:

```
at 08:28:10 PM
❯ curl -v http://localhost:8080/soap/anything

* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /soap/anything HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 503 Service Unavailable
< Content-Type: application/json;charset=UTF-8
< Content-Length: 61
<
* Connection #0 to host localhost left intact
{"message":"Servicio SOAP no disponible, intente más tarde"}%
```

Circuit Breaker test

```
for i in 1 2 3; do
  curl -i http://localhost:8080/soap/anything
done

HTTP/1.1 503 Service Unavailable
Content-Type: application/json;charset=UTF-8
Content-Length: 61

{"message":"Servicio SOAP no disponible, intente más tarde"}HTTP/1.1 503 Service Unavail
able
Content-Type: application/json;charset=UTF-8
Content-Length: 61

{"message":"Servicio SOAP no disponible, intente más tarde"}HTTP/1.1 503 Service Unavail
able
Content-Type: application/json;charset=UTF-8
Content-Length: 61

{"message":"Servicio SOAP no disponible, intente más tarde"}%
```

Circuit breaker con el flujo normal

```
curl -s "http://localhost:8080/actuator/metrics/resilience4j.circuitbreaker.state?tag=name:soap
rcuit" | jq .

{
  "name": "resilience4j.circuitbreaker.state",
  "description": "The states of the circuit breaker",
  "baseUnit": null,
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 1.0
    }
  ],
  "availableTags": [
    {
      "tag": "state",
      "values": [
        "closed",
        "disabled",
        "half_open",
        "forced_open",
        "open",
        "metrics_only"
      ]
    },
    {
      "tag": "group",
      "values": [
        "none"
      ]
    }
  ]
}
```

5. Monitoreo y trazabilidad

Herramienta de monitoreo
Podríamos usar Prometheus para raspar métricas expuestas por Kong, Spring Cloud Gateway y el microservicio Docker, y luego visualizarlas en Grafana. Ahí se montaría un dashboard que muestre, por ejemplo, el percentil p95 de latencia, la tasa de errores 4xx/5xx, el uso de CPU/memoria y los contadores de rate-limiting.

Trazado distribuido
Se podría instrumentar cada petición con OpenTelemetry (o Spring Cloud Sleuth). Kong inyectaría un `traceId` y un `spanId`; luego Spring Cloud Gateway generaría spans para sus filtros (Retry, CircuitBreaker) y el servicio Node.js añadiría spans para la validación de JWT y la llamada al SOAP. Todas las trazas serían enviadas a Jaeger (o Zipkin) para inspeccionar el recorrido completo y detectar posibles cuellos de botella.

Log centralizado
Imaginemos que cada componente emite logs en JSON. Mediante Logstash (o Filebeat) los enviaríamos a Elasticsearch, y con Kibana podríamos buscarlos y analizarlos. Esto permitiría, por ejemplo, filtrar por `traceId`, identificar picos de latencia o peticiones bloqueadas por rate-limiting o JWT inválido, y configurar alertas automáticas.

Métricas clave

- **Gateway (Kong/Spring Cloud):** tasa de peticiones, latencias (`proxy_latency`, `upstream_latency`), recuento de retries y estado del circuit breaker.
- **SolicitudService (Node.js):** número de peticiones, histogramas de latencia, errores 500, rechazos JWT y métricas de la llamada SOAP.
- **Infraestructura:** uso de CPU/memoria por contenedor (vía cAdvisor o kube-state-metrics), reinicios de pods y disponibilidad de nodos.

Alertas y SLOs
Podríamos definir en Prometheus Alertmanager reglas como:

- “Circuit breaker en estado OPEN por más de X segundos”
- “Tasa de error 5xx > 5 % en 5 min”
- “p95 de latencia supera el SLO”

Cuando se cumpliera alguna regla, se enviarían notificaciones (Slack, correo) para garantizar una respuesta rápida ante incidentes.

El flujo de una petición a `/solicitudes` con trazabilidad quedaría así:

1. **Kong Gateway** recibe la petición y valida el JWT con su plugin. Internamente crea un **TraceID** único y arranca un span de validación (`Kong:validate_token`). Luego abre otro span (`Kong:proxy_request`) cuando reenvía la llamada a Spring Cloud Gateway.
2. **Spring Cloud Gateway** hereda ese TraceID y genera un span al recibir la solicitud (`SCG:received_request:/solicitudes`). Inmediatamente antes de llamar al microservicio aplica el filtro de circuito, abriendo un span `SCG:apply_CircuitBreaker(solicitudCircuit)`. Una vez superado el filtro, forwards la petición a SolicitudService.
3. **SolicitudService (Node.js)** propaga el mismo TraceID, arranca un span de autenticación interna (`Express:authenticateJWT`) para verificar la firma del token y luego otro span (`Express:sendSOAPRequest`) cuando invoca al servicio SOAP externo.
4. **Servicio SOAP** (mock o real) recibe la petición y marca dos spans propios: `SOAP:receiveRequest` y `SOAP:sendResponse` al devolver el resultado.
5. En la **respuesta**, cada componente cierra su span correspondiente: Spring Cloud Gateway registra `SCG:receive_response` (incluyendo latencia y posibles reintentos), y finalmente Kong cierra `Kong:proxy_response` antes de enviar el resultado al cliente. Gracias a este encadenamiento de spans podemos ver, en Jaeger o Zipkin, el tiempo que pasó en cada filtro, en el microservicio y en la llamada SOAP, lo que facilita localizar cuellos de botella o errores en el flujo distribuido.