**UNIVERSIDAD DE LAS AMERICAS**

**FACULTAD DE INGENIERIA Y CIENCIAS APLICADAS**

**INGENIERIA DE SOFTWARE**

**INTEGRACION DE PROCESOS**

**Nombre:** Fabiana Vásconez

**EXAMEN PROGRESO 2**

**Estructura del Proyecto:**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**API**

El microservicio implementado con FastAPI permite gestionar solicitudes mediante una API REST, protegida por autenticación JWT (token Bearer). Se definen dos endpoints principales: uno para crear solicitudes (POST /solicitudes) y otro para consultarlas (GET /solicitudes/{id}). Para garantizar seguridad, se valida un JWT en cada petición protegida. A nivel de trazabilidad, se incluye un middleware que registra en el log cada petición entrante con un request\_id, tiempo de ejecución, método y ruta, lo cual facilita el seguimiento y debugging. Además, se registran eventos importantes como errores de autenticación o fallos en servicios externos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**JWT**

Se crea el Key que se utilizará mas adelantes para realizar las respectivas pruebas en postman

Captura de pantalla de computadora

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**API GATEWAY – KONG**

Este fragmento de configuración define una ruta en Spring Cloud Gateway llamada `solicitud\_route`, que enruta las solicitudes dirigidas a `/solicitudes/\*\*` hacia el servicio en `http://localhost:8000`. Incluye dos filtros clave para mejorar la resiliencia del sistema. El filtro `Retry` reintenta automáticamente hasta 2 veces si las respuestas devuelven códigos `BAD\_GATEWAY` (502) o `GATEWAY\_TIMEOUT` (504). El filtro `CircuitBreaker` monitorea el estado del servicio: si al menos el 50% de las llamadas fallan o son lentas (más de 2 segundos), el circuito se abre y las solicitudes se redirigen al endpoint de respaldo `/fallback`. La configuración también establece reglas para cómo se analiza el comportamiento (ventana deslizante de 10 llamadas, mínimo de 5 llamadas requeridas para evaluar fallas, espera de 60 segundos antes de intentar reabrir el circuito, y solo permite 2 llamadas de prueba en modo "half-open"). Todo esto garantiza que el sistema pueda recuperarse y responder apropiadamente ante fallas del servicio.

Escala de tiempo

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**ISTIO:**

Los archivos YAML que se utilizan con Istio definen recursos personalizados (CRDs) que permiten controlar y configurar el comportamiento del tráfico dentro del clúster mediante el Service Mesh. Por ejemplo, un archivo Gateway define cómo el tráfico externo entra al mesh, especificando el puerto, protocolo y host. El recurso VirtualService indica cómo enrutar las solicitudes entrantes a los servicios backend, permitiendo definir rutas, reintentos, políticas de tiempo de espera y fallbacks. El archivo DestinationRule complementa esto al definir políticas de conexión como circuit breaking, load balancing, y TLS por destino de servicio. Estos YAML trabajan juntos para implementar patrones de resiliencia, seguridad, y observabilidad, como retries, timeouts, circuit breakers y trazabilidad con herramientas como Jaeger o Kiali. Así, Istio permite separar la lógica de red de los servicios, centralizando el control del tráfico y la resiliencia en la capa de infraestructura.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente con confianza baja

Istio está enrutando el tráfico de ese pod:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Kiali:**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Prueba Afectando al servicio**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Prueba correcta:**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Una captura de pantalla de una computadora

Descripción generada automáticamente

Para implementar monitoreo en esta arquitectura basada en microservicios con FastAPI, Docker, Kubernetes e Istio, usaría una combinación de herramientas especializadas en observabilidad, métricas y trazabilidad distribuida. La solución se enfocaría en recolectar información del estado de salud, rendimiento y comportamiento de cada servicio.

**¿Cómo lo implementaría?**

1. Istio ya instalado puede capturar automáticamente métricas y trazas de red a nivel de malla de servicios gracias a su sidecar Envoy.
2. Desplegaría Prometheus para recopilar métricas (telemetría) y Grafana para visualizarlas en dashboards.
3. Integraría Jaeger o Zipkin como sistema de trazas distribuidas para visualizar el flujo de las peticiones entre servicios.
4. Agregaría logging estructurado en cada microservicio (como ya lo hiciste en FastAPI), y lo conectaría con Elasticsearch + Fluentd + Kibana (EFK) si se requiere análisis de logs centralizado.
5. Añadiría liveness/readiness probes en Kubernetes para que los pods reporten su estado de salud, y así poder actuar en caso de errores.
6. Configuraría alertas con Alertmanager para ser notificado ante anomalías (por ejemplo, tasa alta de errores o latencia).

**Herramientas que usaría:**

* **Prometheus** → recolectar métricas.
* **Grafana** → visualización de métricas.
* **Jaeger** → trazabilidad distribuida.
* **Kiali** → inspección de tráfico y salud en la malla Istio.
* **Istio Telemetry** → recolección automática de métricas y trazas vía sidecar.
* **EFK Stack** (opcional) → gestión de logs.

**Métricas y trazas a capturar:**

**Métricas técnicas:**

* Latencia promedio por servicio y endpoint.
* Tasa de errores 4xx/5xx.
* Cantidad de solicitudes por segundo (RPS).
* Uso de CPU y memoria por pod.
* Tiempo de respuesta de las llamadas a servicios externos (SOAP).
* Número de reintentos y activaciones del circuit breaker.

**Trazas:**

* Flujo completo de una solicitud desde la entrada en Istio Ingress hasta el servicio destino.
* Tiempo que cada microservicio tarda en procesar su parte.
* Cuellos de botella o servicios lentos.
* Propagación de contexto de trazas (usando headers como x-request-id, b3, etc.).