GESTIÓN DE SERVICIOS WAKANDA

Para la creación de un sistema de gestión de servicios, de una ciudad inteligente se diseñó e implemento una arquitectura de microservicios. El sistema está diseñado para ser escalable, resiliente y observable.

Tecnologías:

Python: Se utilizo como lenguaje de programación principal.

FastAPI: Se ha utilizado tanto para los microservicios como para el Frontend y el Gateway debido a su alto rendimiento.

Docker: Cada servicio se empaqueta en su propio contenedor, garantizando aislamiento y reproducibilidad.

Docker Compose: Gestiona el ciclo de vida de la aplicación multi-contenedor, definiendo redes internas y volúmenes para persistencia de datos.

Librerías:

- httpx: Para realizar peticiones HTTP asíncronas entre servicios.
- uvicorn: Servidor ASGI para ejecutar las aplicaciones FastAPI.
- jinja2: Motor de plantillas para el renderizado del Frontend.
- prometheus-client & opentelemetry: Para la instrumentación de métricas y trazas.

Microservicios:

En el sistema disponemos de 6 microservicios autónomos. Cada servicio simula datos que se reportan a una central y operan de forma aislada dentro de su propio contenedor Docker. Responsabilidad de cada microservicio:

- **Traffic Service:**
  Monitoriza el flujo vehicular en tiempo real, simula sensores de calzada que cuentan vehículos y comunica el estado del tráfico.
  Alertas: Calcula la densidad de tráfico y emite estados como FLOWING, MODERATE o CONGESTED dependiendo si el número de vehículos supera ciertos valores establecidos.

- **Energy Service:**
  Gestiona la infraestructura de la red eléctrica, simula "Transformadores Inteligentes". Monitoriza el voltaje (V) y el porcentaje de carga.
  Alertas: Detecta anomalías críticas como CRITICAL_OVERLOAD (Sobrecarga > 90%) o WARNING_LOW_VOLTAGE (Caídas de tensión).

- **Water Service:**
  Control de calidad y suministro de agua, sensores IoT que miden el pH y la turbidez (NTU).
  Alertas: Identifica agua no potable o fallos en bombas mediante estados como UNSAFE_PH_LEVEL o PUMP_FAILURE.

- **Waste Service:**
  Gestión eficiente de residuos urbanos, contenedores inteligentes de diferentes tipos (Orgánico, Plástico, Vidrio) que reportan su nivel de llenado.
  Alertas: Genera avisos de recogida prioritaria (CRITICAL_FULL_PICKUP_REQUIRED) cuando un contenedor supera el 90% de capacidad.

- **Security Service:**
  Centro de mando de videovigilancia, simula cámaras de CCTV con analítica de vídeo para contar personas.
  Alertas: Detecta aglomeraciones peligrosas (WARNING_CROWD) o incidentes de seguridad pública (DANGER).

- **Health Service:**
  Gestión de recursos sanitarios de emergencia, monitoriza el estado de hospitales y centros de salud en la zona.
  Alertas: Controla la ocupación de camas UCI y la disponibilidad de ambulancias, emitiendo alertas como CRITICAL_BED_SHORTAGE.

(Persistencia en la Simulación)

Para garantizar una simulación realista sin la complejidad de una base de datos externa, los servicios implementan un patrón de Persistencia en Memoria

La primera vez que se consulta una zona (ej. "Zona A"), el servicio determina aleatoriamente cuántos sensores tiene esa zona (ej. 3 semáforos, 2 hospitales).

Este número de sensores se guarda en una variable global (ZONE_CONFIG dictionary). Las consultas posteriores a la misma zona siempre devolverán el mismo número de dispositivos, aunque los valores de sus lecturas (temperatura, tráfico, etc.) varíen dinámicamente en cada petición para simular el paso del tiempo.

API Endpoints

Todos los microservicios exponen una API RESTful estandarizada. Estructura de los endpoints:

- **Health Check:**
  Utilizado por el orquestador y sistemas de monitoreo para verificar que el servicio está vivo. | Método: GET | Ruta: / |

  Ejemplo de Respuesta:
  { "service": "Traffic Service", "status": "active" }

- **Endpoint de Datos de Zona:**
  Es el endpoint principal que devuelve la telemetría de los sensores.
  | Método: GET | Ruta: /{service_name}/zone/{zone_id} |

  Parámetros: zone_id: Identificador de la zona (ej: "A", "B", "CENTRAL").

  Ejemplo de Respuesta:

```
{
  "id": "T-ZONA-A-01",
  "zone": "A",
  "timestamp": "2025-11-13T10:00:00",
  "vehicle_count": 45,
  "status": "NORMAL"
},
{
  "id": "T-ZONA-A-02",
  "zone": "A",
  "timestamp": "2025-11-13T10:00:00",
  "vehicle_count": 120,
  "status": "CONGESTED"
}
```

- **Endpoint de Métricas:**
  Expone datos técnicos para ser recolectados por Prometheus.
  | Método: GET | Ruta: /metrics |

  Formato: Texto plano compatible con el estándar OpenMetrics (ej: app_request_count_total 150).

Arquitectura de Conectividad e Infraestructura:

Para el desarrollo del sistema se siguió una arquitectura orientada a microservicios. Para lograr que componentes independientes funcionen como un sistema unificado y robusto, hemos implementado patrones de diseño de software avanzados para el descubrimiento, enrutamiento y orquestación.

Detalles de la infraestructura:

- **Docker Compose:**
  Toda la infraestructura se define en el archivo docker-compose.yml. Este orquestador gestiona el ciclo de vida de los contenedores y define la topología de la red.
  Se ha creado una red tipo bridge personalizada (Red Virtual).
  Seguridad: Los microservicios de negocio (Tráfico, Energía, etc.) no exponen sus puertos al host anfitrión (Windows). Viven aislados dentro de esta red virtual.
  Docker proporciona un servidor DNS interno que permite a los contenedores comunicarse usando sus nombres de servicio.
  Ejemplo, el Gateway puede hacer un ping traffic_service sin necesidad de conocer su dirección IP numérica, la cual cambia en cada reinicio.

- **Service Discovery:**
  En un entorno distribuido dinámico, las direcciones IP de los servicios son efímeras. Para evitar el acoplamiento fuerte, se implemento un componente Service Registry (Puerto 8002).

  El flujo de descubrimiento funciona en dos fases:

  1. Auto-Registro (Startup): Cuando un microservicio (ej. water_service) arranca, ejecuta un script de inicio (on_event("startup")) que envía una petición POST al Registry indicando su nombre y su ubicación interna.

  2. Descubrimiento (Runtime): Cuando el Gateway necesita contactar con un servicio, no "adivina" la dirección. Consulta al Registry (GET /discover/{service_name}) para obtener la URL válida y actualizada.

- **API Gateway**
  El componente Gateway API (Puerto 8080) actúa como la única puerta de entrada para el tráfico externo (Frontend, Scripts de prueba etc.). Implementa el patrón Reverse Proxy y cumple funciones críticas de resiliencia (enrutamiento inteligente, abstracción, resiliencia y timeouts).
  Para evitar fallos en cascada (que un servicio lento cuelgue todo el sistema), el Gateway implementa un timeout de 5 segundos en sus clientes HTTP asíncronos (httpx). Si un servicio no responde en ese tiempo, el Gateway corta la conexión y devuelve un error controlado (503 Service Unavailable), permitiendo que el resto del sistema siga funcionando.

Observabilidad y Monitorización

- Prometheus (Métricas): Recolecta métricas numéricas (series temporales).
  Viaja cada 5 segundos a los endpoints /metrics de todos los servicios para leer datos como app_request_count (número de peticiones) o app_request_latency (tiempo de respuesta).

- Grafana (Visualización): Interfaz gráfica para humanos.
  Se conecta a Prometheus y muestra un Dashboard Unificado ("Wakanda Monitor"). Permite visualizar la carga del sistema en tiempo real, diferenciando el tráfico por cada microservicio mediante consultas PromQL.

- Jaeger (Trazabilidad Distribuida): Rastreo de peticiones ("Tracing").
  Gracias a OpenTelemetry, podemos ver la "historia" de una petición desde que entra al Gateway hasta que llega al microservicio final. Es vital para depurar cuellos de botella y errores de comunicación.

Frontend

El Frontend es una aplicación web construida también con FastAPI y Jinja2 Templates, servida en el puerto 3001.

Dashboard: Cuando un usuario consulta una zona (ej: "Zone A"), el Frontend realiza 6 peticiones en paralelo (asyncio.gather) al Gateway para obtener el estado de Tráfico, Energía, Agua, Residuos, Seguridad y Salud simultáneamente.

Toolbar: Incluye accesos directos integrados a las herramientas de observabilidad (Grafana, Jaeger, etc.).

Pruebas

Para garantizar la fiabilidad del código, se han implementado dos niveles de pruebas:

-Tests Unitarios (Pytest):

Se ejecutan dentro del contenedor (docker compose exec traffic_service pytest).

Verifican la lógica interna de los servicios de forma aislada.

Comprueban que los endpoints responden 200 OK y que la persistencia de datos por zona funciona correctamente.

Pytest Traffic_service (resultado):

========== test session starts ==========

platform linux -- Python 3.9.25, pytest-8.4.2, pluggy-1.6.0

rootdir: /app

plugins: anyio-4.12.0

collected 3 items

tests/test_main.py ... [100%]

========== warnings summary ==========

main.py:39

/app/main.py:39: DeprecationWarning:

on_event is deprecated, use lifespan event handlers instead.

Read more about it in the

[FastAPI docs for Lifespan Events](https://fastapi.tiangolo.com/advanced/events/).

@app.on_event("startup")

../usr/local/lib/python3.9/site-packages/fastapi/applications.py:4576

/usr/local/lib/python3.9/site-packages/fastapi/applications.py:4576: DeprecationWarning:

on_event is deprecated, use lifespan event handlers instead.

Read more about it in the

[FastAPI docs for Lifespan Events](https://fastapi.tiangolo.com/advanced/events/).

return self.router.on_event(event_type)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html

========== 3 passed, 2 warnings in 0.46s ==========

Transient error StatusCode.UNAVAILABLE encountered while exporting traces to jaeger:4317, retrying in 1.00s.

-Tests de Integración (Script de Python):

Un "Bot" automatizado (integration_test.py) que se ejecuta externamente.

Simula el comportamiento de usuarios reales navegando aleatoriamente por diferentes zonas. Genera tráfico constante contra el API Gateway, validando que el enrutamiento, el descubrimiento de servicios y la tolerancia a fallos funcionan en conjunto.

Este script es el que alimenta de datos las gráficas de Grafana en tiempo real.

integration_test.py (resultado):

[21:51:50] ✅ WASTE | Zone E | 4 items | 87ms

[21:51:51] ✅ WATER | Zone CENTRAL | 1 items | 30ms

[21:51:52] ✅ TRAFFIC | Zone E | 3 items | 29ms

[21:51:53] ✅ WATER | Zone B | 2 items | 22ms

[21:51:53] ✅ WATER | Zone C | 3 items | 19ms

[21:51:54] ✅ HEALTH | Zone INDUSTRIAL | 1 items | 36ms

[21:51:54] ✅ ENERGY | Zone INDUSTRIAL | 3 items | 34ms

[21:51:55] ✅ HEALTH | Zone E | 1 items | 24ms

CONCLUSION

El desarrollo del proyecto se buscó la eficacia de una arquitectura de microservicios para gestionar sistemas complejos. La implementación ha cumplido todos los objetivos técnicos, destacando cuatro logros principales:

Arquitectura Modular y Desacoplada: Gracias a Docker y al diseño de servicios independientes, hemos logrado un sistema donde cada componente (Tráfico, Energía, etc.) puede evolucionar sin afectar a los demás.

Resiliencia Operativa: La implementación del API Gateway y el Service Registry ha demostrado ser robusta. Las pruebas de integración confirmaron que el sistema aísla los errores: si un servicio cae, el resto de la plataforma sigue funcionando, garantizando la alta disponibilidad.

Observabilidad Total: La integración del stack Prometheus, Grafana y Jaeger proporciona control absoluto sobre el sistema, permitiendo visualizar la carga en tiempo real y trazar el origen exacto de cualquier latencia o fallo.

Fiabilidad Validada: Mediante tests unitarios y scripts de carga automatizados, se ha verificado que la infraestructura soporta tráfico concurrente y mantiene la coherencia de los datos simulados.

En conclusión, se ha construido una base tecnológica sólida, moderna y escalable, preparada para futuras ampliaciones o despliegues en entornos de producción.