Plantilla 

January 2023

Acerca de arc42

arc42, La plantilla de documentación para arquitectura de sistemas y de software.

Por Dr. Gernot Starke, Dr. Peter Hruschka y otros contribuyentes.

Revisión de la plantilla: 7.0 ES (basada en asciidoc), Enero 2017

© Reconocemos que este documento utiliza material de la plantilla de arquitectura arc42, https://www.arc42.org. Creada por Dr. Peter Hruschka y Dr. Gernot Starke.

Introducción y Metas

El proyecto consiste en la implementación de un **Sistema de Gestión de Logs Meteorológicos** utilizando un patrón de microservicios basado en mensajería. Su objetivo principal es recibir, procesar, validar y almacenar de forma robusta los datos de estaciones meteorológicas simuladas.

El modelo que seguira nuestra base de datos es el siguiente:

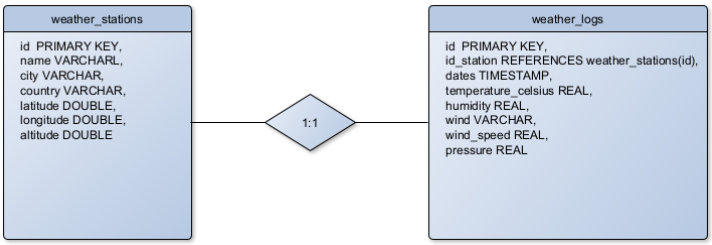

Figure 1: Modelo entidad-relacion

Vista de Requerimientos

El sistema debe cumplir los siguientes requerimientos funcionales:

- La ingesta de datos debe realizarse a través de un servicio Productor (**Python**) que simule o reciba logs en formato **JSON**.
- El Productor debe publicar los logs a un **Exchange** de **RabbitMQ** utilizando mensajes **durables**.
- Los datos validados deben persistirse en una base de datos **PostgreSQL** especificamente en la tabla creada para este proposito (`weather_logs`).
- El sistema debe estar completamente **contenereizado** (RabbitMQ, PostgreSQL, Productor, Consumidor) y orquestado con `docker-compose.yml` con reinicios automáticos.

Metas de Calidad

- **Confiabilidad:** Los mensajes deben ser **persistentes** y las colas **durables** para evitar la pérdida de datos ante caídas del sistema o del consumidor.
- **Procesamiento Controlado:** El consumo de mensajes debe ser ordenado y controlado mediante `prefetch_count=1`.
- **Mantenibilidad:** Se utilizarán buenas prácticas de código y documentación, con una clara separación de responsabilidades entre los microservicios.

Restricciones de la Arquitectura

El diseño de la arquitectura está condicionado por las siguientes restricciones técnicas:

- **Tecnología Principal:** Python 3.13+ y librerías estables (`pika`, `psycopg2`).
- **Durabilidad de Mensajería:** Los mensajes publicados deben ser `persistent` (`delivery_mode=2`).
- **Control de Flujo:** El consumo debe usar `prefetch_count=1` para garantizar un procesamiento ordenado.
- **Persistencia de Estado: PostgreSQL** y **RabbitMQ** deben ser *stateful* y utilizar **volúmenes Docker** para evitar la pérdida de datos y estado de colas.
- **Buenas Prácticas:** Seguir buenas prácticas de código, documentación y manejo robusto de excepciones (e.g., lógica de reconexión).

Alcance y Contexto del Sistema

Contexto de Negocio

El sistema actúa como el middleware central para la ingesta y almacenamiento de datos de observación meteorológica.

Alcance:

- Ingesta de datos JSON.
- Validación de rangos de logs.
- Persistencia asíncrona y robusta de datos validados.
- Orquestación completa de la infraestructura con Docker.

Fuera de alcance:

- Implementación de una Dead Letter Queue (DLQ) avanzada (es una extensión potencial).
- API REST para consulta de logs históricos (es una extensión potencial).
- Análisis predictivo o complejo de los datos.

Contexto Técnico

El sistema es un pipeline de datos *asíncrono* y *deacoplado*.

Sistemas externos relevantes:

- **RabbitMQ Broker:** Eje central de la comunicación desacoplada.

- **PostgreSQL Database:** Componente de almacenamiento persistente.
- **Docker Engine/Compose:** Herramienta de contenedores y orquestación.

Interacciones técnicas:

- **Productor → RabbitMQ:** Envío de mensajes persistentes a un *Exchange* durable.
- **Consumidor RabbitMQ**: Recepción de mensajes mediante prefetch_count=1 y ACK manual.
- **Consumidor ↔ PostgreSQL:** Conexión gestionada con reconexión automática para insertar logs.

Vista de Bloques

Descripción general

El sistema está compuesto por cuatro bloques interconectados mediante una red Docker interna.

Bloques principales

Bloque	Tecnología	Responsabilidad Principal
Data Producer	Python (pika)	Simular logs JSON. Publicar mensajes **durables** a RabbitMQ, manejando fallos de conexión.
Message Broker	RabbitMQ	Enrutamiento (Exchange: `direct`) y gestión de la Cola (`durable`). Expone dashboard de administración.
Data Consumer	Python (pika, psycopg2)	Recepción con `prefetch_count=1` y `ACK manual`. Validación de rangos de datos. Persistencia robusta en PostgreSQL.
Database	PostgreSQL	Almacenamiento *stateful* de logs validados en la tabla `weather_logs`.

Interrelaciones entre bloques

- **Producer → Broker:** Envío asíncrono. El Productor depende de la disponibilidad del Broker.
- **Broker → Consumer:** Distribución controlada. El Broker retiene el mensaje hasta recibir el ACK.

- **Consumer ↔ Database:** Transacción de inserción. El Consumidor mantiene la conexión con lógica de reconexión.

Vista de Ejecución

Flujo de Ejecución Típico: Ingesta de Log

1. El **Producer** genera un log JSON (simulado).
2. El **Producer** publica el mensaje al *Exchange* de RabbitMQ, marcándolo como **persistente** (`delivery_mode=2`).
3. El **Consumer** toma el mensaje de la cola (por `prefetch_count=1`).
4. El **Consumer** valida los rangos de valores (T, H, P).
5. **Si válido:** El Consumer inserta el log en PostgreSQL y envía un **ACK manual** a RabbitMQ.
6. **Si inválido:** El Consumer registra el error (*log*) y decide si descarta el mensaje (ACK) o lo deja para reintento/DLQ (sin ACK).

Manejo de Robustez

- **Caída del Consumer:** Si el Consumer falla antes de enviar el ACK, el mensaje permanece en la cola (gracias a la durabilidad) y será reasignado a otro consumidor (o al mismo al reiniciarse).
- **Caída de DB:** El Consumidor implementa una lógica de **reconexión automática** a PostgreSQL. Solo enviará el ACK después de la inserción exitosa, previniendo la pérdida de datos durante interrupciones temporales de la base de datos.

Para esto implementamos los siguientes tests:

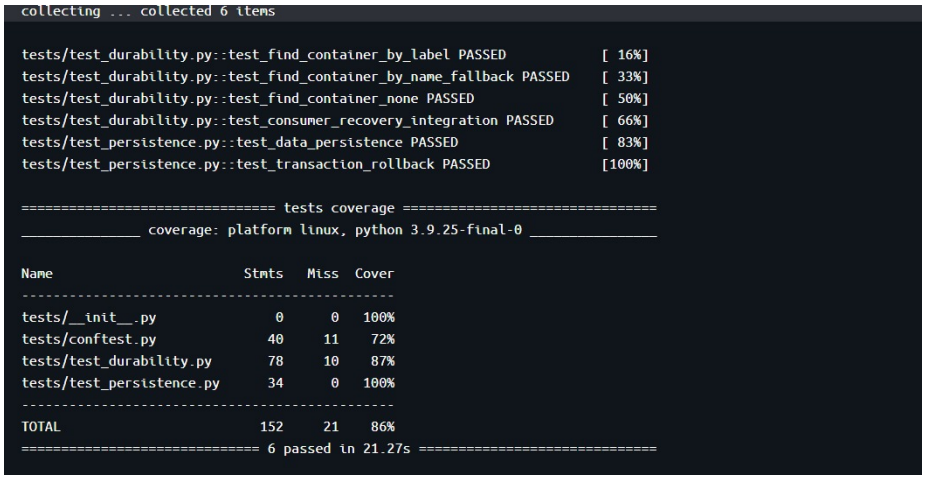

Figure 2: Tests y validaciones

Vista de Despliegue

Orquestación con Docker Compose

El sistema se despliega utilizando `docker-compose.yml` para definir y levantar la infraestructura en un solo comando.

```
services:
  # ============================
  #  RabbitMQ (mensajería)
  # ============================
  rabbitmq:
    image: rabbitmq:3.11-management
    container_name: rabbitmq
    environment:
      RABBITMQ_DEFAULT_PASS: ${RABBIT_PASS}
    ports:
      - "5672:5672"
    healthcheck:
      retries: 6
    restart: unless-stopped
    networks:
      - backend

  # ============================
  #  PostgreSQL (base de datos)
  # ============================
  db:
```

```
  image: postgres:15
  container_name: postgres
  environment:
    PGDATA: /var/lib/postgresql/data/pgdata
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  ports:
    - "5432:5432"
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./init:/docker-entrypoint-initdb.d
  restart: unless-stopped
  networks:
    - backend

# ============================
#  Adminer (UI para PostgreSQL)
# ============================
adminer:
  image: adminer
  container_name: adminer
  restart: always
  ports:
    - "8080:8080"
  depends_on:
    - db
  networks:
    - backend

# ============================
#  Producer (envía datos)
# ============================
producer:
  build:
    context: ./src/app/services/producers_service
  container_name: producer
  depends_on:
    - rabbitmq
  environment:
    - RABBIT_USER=${RABBIT_USER}
    - RABBIT_PASS=${RABBIT_PASS}
    - RABBITMQ_HOST=rabbitmq
  volumes:
    - ./logs:/app/logs
  restart: on-failure
```

```
  networks:
    - backend

# ==========================
#  Consumer (recibe y guarda)
# ==========================
consumer:
  build:
    context: ./src/app/services/consumers_service
  container_name: consumer
  depends_on:
    - rabbitmq
    - db
  environment:
    - RABBIT_USER=${RABBIT_USER}
    - RABBIT_PASS=${RABBIT_PASS}
    - RABBITMQ_HOST=rabbitmq
    - POSTGRES_HOST=db
  volumes:
    - ./logs:/app/logs
  restart: on-failure
  networks:
    - backend

# ==========================
#  Prometheus (monitoreo)
# ==========================
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - "9090:9090"
  depends_on:
    - consumer
    - producer
  networks:
    - backend

# ==========================
#  Grafana (visualización)
# ==========================
grafana:
  image: grafana/grafana:latest
  container_name: grafana
```

```
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
    ports:
      - "3000:3000"
    depends_on:
      - prometheus
    networks:
      - backend

  # ============================
  #  Tests
  # ============================
  tests:
    build:
      context: ./src/app/tests
      dockerfile: Dockerfile.test
    container_name: tests
    volumes:
      - ./src/app:/app
      - /var/run/docker.sock:/var/run/docker.sock
    depends_on:
      - rabbitmq
      - db
    environment:
      - POSTGRES_HOST=db
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=postgres
    networks:
      - backend

# ============================
# Volúmenes persistentes
# ============================
volumes:
  rabbitmq_data:
  postgres_data:

# ============================
# Red compartida
# ============================
networks:
  backend:
```

Esquema de Base de Datos

El script `init.sql` define la tabla principal `weather_logs`:

```
-- Script de inicialización: init.sql
-- CORREGIDO: crear estaciones primero, luego logs que referencian estaciones.
CREATE TABLE IF NOT EXISTS weather_stations (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    city VARCHAR(150),
    country VARCHAR(150),
    latitude DOUBLE PRECISION,
    longitude DOUBLE PRECISION,
    altitude DOUBLE PRECISION
);

CREATE TABLE IF NOT EXISTS weather_logs (
    id SERIAL PRIMARY KEY,
    id_station INTEGER NOT NULL REFERENCES weather_stations(id),
    dates TIMESTAMP WITH TIME ZONE NOT NULL,
    temperature_celsius REAL,
    humidity REAL,
    wind VARCHAR(5),
    wind_speed REAL,
    pressure REAL,
    -- CHECK básico para evitar valores absurdos
    CHECK (temperature_celsius IS NULL OR (temperature_celsius > -100 AND temperature_celsiu
);

INSERT INTO weather_stations (name, city, country, latitude, longitude, altitude) VALUES
('Estacion Norte', 'Ciudad A', 'Colombia', 4.700, -74.050, 2550),
('Estacion Sur',  'Ciudad B', 'Chile', 4.500, -74.100, 2450),
('Estacion Este', 'Ciudad C', 'Argentina', 4.650, -73.950, 2600),
('Estacion Oeste', 'Ciudad D', 'Romania', 4.720, -74.200, 2400),
('Estacion Central','Ciudad E', 'Colombia', 4.680, -74.080, 2500);

-- Asegurar que la constraint existe también para tablas ya creadas (idempotente)
DO $$
BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM pg_constraint WHERE conname = 'chk_temperature_range'
    ) THEN
        ALTER TABLE weather_logs
          ADD CONSTRAINT chk_temperature_range
          CHECK (temperature_celsius IS NULL OR (temperature_celsius > -100 AND temperature_
    END IF;
```

```
END$$;
```

Y como despliegue final tenemos nuestras graficas hechas con ayuda de grafana y prometheus:

Figure 3: Demostracion final de uso

Conceptos Transversales (Cross-cutting)

- **Seguridad:** Uso estricto de **variables de entorno** de Docker para inyectar credenciales (DB y RabbitMQ) y evitar hardcoding.
- **Monitoreo (Logging):** Implementación de registros de eventos en cada microservicio (`logging` de Python) para trazar la recepción, la validación y el estado de las transacciones (ACK/No-ACK).
- **Escalabilidad:** La arquitectura de mensajería desacoplada permite la **escalabilidad horizontal** de los `consumer` sin afectar al productor ni a la base de datos (mediante la distribución de carga de la cola).

Requerimientos de Calidad

- **Rendimiento:** El Consumidor debe ser capaz de procesar logs de forma rápida y el ACK debe ser enviado inmediatamente después de la inserción exitosa.
- **Confiabilidad (Durabilidad):** Se verificará que los logs en cola no se pierdan después de reiniciar los contenedores `db` y `broker` (prueba de persistencia).
- **Confiabilidad (Integridad):** La lógica de validación de rangos debe ser rigurosa; un dato fuera de rango no debe ser insertado como válido.

Riesgos y Deuda Técnica

- **Riesgo de Loop de Mensajes:** La gestión básica de errores sin una **Dead Letter Queue (DLQ)** implica que un mensaje constantemente inválido podría ser reentregado indefinidamente, consumiendo recursos.
- **Deuda Técnica (Monitoreo):** La dependencia del *logging* para la verificación limita la visualización en tiempo real. La integración de **Prometheus/Grafana** es necesaria para el monitoreo productivo.
- **Riesgo de Desfase de Versiones:** La compatibilidad de librerías Python con versiones futuras (especialmente `pika`) puede requerir mantenimiento periódico.

Glosario

- **ACK (Acknowledgement):** Confirmación enviada por el Consumidor a RabbitMQ indicando que el mensaje fue procesado exitosamente.
- **Durable/Persistent:** Propiedad aplicada a Colas y Mensajes que asegura que sobreviven a un reinicio del Broker.
- **DLQ (Dead Letter Queue):** Cola especializada para mensajes que fallaron el procesamiento un número máximo de veces.
- **prefetch_count=1:** Configuración que limita al Consumidor a procesar solo un mensaje a la vez, garantizando el control de flujo.
- **Stateful:** Componente (como una DB o Broker) cuyo estado de datos debe ser preservado entre reinicios mediante volúmenes.

Link del repositorio

He aqui el link de nuestro repositorio donde fue desarrollado nuestro programa: https://github.com/SteinDevlop/Sistemas-de-Mensajes