ChatGPT

Diseño del microservicio file_service

El microservicio **file_service** gestiona archivos (imágenes, PDFs, HTML, etc.) de forma centralizada. Se basa en **FastAPI**, almacena los archivos en disco local (`/data/files`) con proyección a S3/NFS, y publica eventos en Kafka. A continuación se detallan los componentes clave, estructura de proyecto, configuración y pasos de integración, manteniendo compatibilidad con el ecosistema de microservicios.

Tecnologías y librerías recomendadas

Se recomienda usar las siguientes herramientas y bibliotecas:

Tecnología/Librería	Uso o propósito
FastAPI	Framework web Python ASGI para crear las APIs REST de forma rápida y asíncrona.
Uvicorn	Servidor ASGI para ejecutar FastAPI en producción.
python-multipart	Soporte para *multipart/form-data* en uploads de archivos en FastAPI [1].
pydantic	Modelos de datos y validación. También para cargar configuración desde variables de entorno [2].
confluent-kafka	Cliente Python de alto nivel para Apache Kafka (Producer/Consumer) [3].
prometheus_client	Librería de métricas Prometheus: permite exponer `/metrics` en FastAPI [4].
pytest & pytest-asyncio	Framework de pruebas unitarias y asíncronas para Python.
python-jose / PyJWT	(Opcional) Decodificar JWT si se requiere extraer info de usuario.

Estas herramientas aseguran alto rendimiento, compatibilidad con Kafka y métricas. Por ejemplo, la documentación FastAPI señala que *python-multipart* es necesario para manejar archivos enviados como formulario [1]. El cliente de Kafka de Confluent brinda una API robusta para producir eventos [3]. Para métricas, la extensión `prometheus_client` permite montar `/metrics` en FastAPI de forma sencilla [4].

Estructura del proyecto

Se propone la siguiente estructura de carpetas y archivos para el microservicio:

```
file_service/
├── app/
│   ├── main.py
│   ├── config.py
│   ├── models.py
│   ├── events.py
│   ├── storage/
│   │   ├── __init__.py
│   │   ├── backend.py
│   │   └── local.py
│   ├── routes/
│   │   ├── __init__.py
│   │   └── files.py
│   └── tests/
│       ├── test_files_api.py
│       └── test_storage.py
├── Dockerfile
├── docker-compose.yml
├── requirements.txt
└── README.md
```

- **app/main.py**: Punto de entrada. Configura FastAPI, dependencias, rutas, métricas y healthchecks.
- **app/config.py**: Configuración basada en variables de entorno (usar `pydantic.BaseSettings`).
- **app/models.py**: Modelos de datos Pydantic: `FileMeta` y `FileStoredEvent`.
- **app/events.py**: Lógica de eventos Kafka (creación del productor, serialización de eventos).
- **app/storage/backend.py**: Interfaz `StorageBackend` (abstracta) con métodos genéricos de almacenamiento.
- **app/storage/local.py**: Implementación `LocalFSBackend` que guarda archivos en el sistema de archivos local.
- **app/routes/files.py**: Endpoints HTTP (`POST/GET/HEAD/DELETE /files`) usando FastAPI.
- **app/tests/**: Pruebas unitarias e integración (usando `pytest` y `TestClient`).
- **Dockerfile** y **docker-compose.yml**: Construcción de contenedor y servicio con volumen persistente `/data/files`.

Árbol de carpetas de ejemplo

```
file_service/
├── app/
│   ├── main.py           # Inicialización de FastAPI, métricas y healthcheck
│   ├── config.py         # Configuración via Pydantic BaseSettings
│   ├── models.py         # Modelos FileMeta y FileStoredEvent
│   ├── events.py         # Lógica para producir evento Kafka
│   ├── storage/
│   │   ├── backend.py    # Interface StorageBackend
│   │   └── local.py      # Clase LocalFSBackend
```

```
│   ├── routes/
│   │   └── files.py      # Endpoints /files
│   └── tests/            # Pruebas unitarias e integración
│       ├── test_files_api.py
│       └── test_storage.py
├── Dockerfile
├── docker-compose.yml
├── requirements.txt
└── README.md
```

Módulos y código clave

A continuación se describen los módulos principales con fragmentos de ejemplo:

- **Interfaces de almacenamiento (`app/storage/backend.py`)**: define métodos genéricos para guardar, obtener, borrar archivos. Se usan `abc.ABC` para crear el contrato. Por ejemplo:

```python
# app/storage/backend.py
from abc import ABC, abstractmethod

class StorageBackend(ABC):
    @abstractmethod
    def save_file(self, key: str, data: bytes) -> None:
        """Almacena datos binarios con una clave única."""
        pass

    @abstractmethod
    def get_file(self, key: str) -> bytes:
        """Retorna datos binarios del archivo identificado por key."""
        pass

    @abstractmethod
    def delete_file(self, key: str) -> None:
        """Elimina el archivo del almacenamiento."""
        pass
```

- **LocalFSBackend (`app/storage/local.py`)**: Implementa `StorageBackend` usando sistema de archivos local. Maneja la ruta base (`FILE_BASE_PATH`) y crea el directorio si no existe. Ejemplo:

```python
# app/storage/local.py
import os
from pathlib import Path
from .backend import StorageBackend

class LocalFSBackend(StorageBackend):
```

```
    def __init__(self, base_path: str):
        self.base_path = Path(base_path)
        self.base_path.mkdir(parents=True, exist_ok=True)

    def save_file(self, key: str, data: bytes) -> None:
        path = self.base_path / key
        path.write_bytes(data)

    def get_file(self, key: str) -> bytes:
        path = self.base_path / key
        return path.read_bytes()

    def delete_file(self, key: str) -> None:
        path = self.base_path / key
        if path.exists():
            path.unlink()
```

- **Modelos de datos (`app/models.py`)**: Define `FileMeta` y `FileStoredEvent` con `pydantic`:

```
# app/models.py
from datetime import datetime
from pydantic import BaseModel
from typing import List

class FileMeta(BaseModel):
    user_id: str
    tags: List[str]
    sha256: str
    created_at: datetime

class FileStoredEvent(BaseModel):
    key: str
    meta: FileMeta
    timestamp: datetime
```

- **Rutas de la API (`app/routes/files.py`)**: Define los endpoints. Ejemplo de `/files`:

```
# app/routes/files.py
from fastapi import APIRouter, File, UploadFile, HTTPException
from starlette.responses import StreamingResponse
from app.config import settings
from app.storage.local import LocalFSBackend
from app.events import publish_file_stored_event
import hashlib
import uuid
```

```
router = APIRouter()
storage = LocalFSBackend(settings.FILE_BASE_PATH)

@router.post("/files")
async def upload_file(file: UploadFile = File(...), user_id: str =
Depends(get_current_user)):
    content = await file.read()
    key = str(uuid.uuid4())
    sha256 = hashlib.sha256(content).hexdigest()
    storage.save_file(key, content)
    meta = {"user_id": user_id, "tags": [], "sha256": sha256, "created_at":
datetime.utcnow()}
    # Public URL de ejemplo: FILE_PUBLIC_URL/{key}
    url = f"{settings.FILE_PUBLIC_URL}/{key}"
    # Publicar evento Kafka
    publish_file_stored_event(key, meta)
    return {"key": key, "url": url}
```

- **Producción de eventos (`app/events.py`)**: Configura `confluent_kafka.Producer` y envía eventos `file_stored`. Por ejemplo:

```
# app/events.py
import json
from confluent_kafka import Producer
from app.config import settings

producer = Producer({"bootstrap.servers": settings.KAFKA_BOOTSTRAP})

def publish_file_stored_event(key: str, meta: dict):
    event = {"key": key, "meta": meta, "timestamp":
datetime.utcnow().isoformat()}
    producer.produce(settings.KAFKA_TOPIC, key=key.encode(),
value=json.dumps(event).encode())
    producer.flush()  # Asegura envío sincrónico
```

- **Healthcheck (`/healthz`)**: Verifica acceso a disco y conexión a Kafka. Por ejemplo, en `main.py`:

```
@app.get("/healthz")
def health_check():
    # Verificar acceso a disco escribiendo/leyendo un archivo temporal
    try:
        test_path = Path(settings.FILE_BASE_PATH) / ".health"
        test_path.write_text("ok")
        test_path.unlink()
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Storage error: {e}")
```

```
    # Verificar Kafka (ping)
    try:
        producer.list_topics(timeout=5)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Kafka error: {e}")
    return {"status": "ok"}
```

- **Métricas Prometheus (`/metrics`)**: Usando `prometheus_client`, se monta en FastAPI. Por ejemplo:

```
from prometheus_client import make_asgi_app
metrics_app = make_asgi_app()
app.mount("/metrics", metrics_app)  # Exponer métricas en /metrics [4]
```

Configuración y variables de entorno

Toda la configuración se maneja con variables de entorno, cargadas en `app/config.py` mediante `pydantic.BaseSettings`. Esto facilita la coherencia con otros servicios y la sobrecarga en pruebas [2]. Ejemplo de `Settings`:

```
# app/config.py
from pydantic import BaseSettings

class Settings(BaseSettings):
    FILE_BASE_PATH: str
    FILE_PUBLIC_URL: str
    KAFKA_BOOTSTRAP: str
    KAFKA_TOPIC: str = "file_stored"
    MAX_UPLOAD_MB: int = 10

    class Config:
        env_file = ".env"  # Opcional: cargar desde .env
```

Variables clave: - `FILE_BASE_PATH` (p.ej. `/data/files`): ruta base en disco.
- `FILE_PUBLIC_URL`: URL base pública para acceder archivos.
- `KAFKA_BOOTSTRAP`: brokers de Kafka (e.g. `kafka:9092`).
- `KAFKA_TOPIC`: nombre del tópico (e.g. `file_stored`).
- `MAX_UPLOAD_MB`: tamaño máximo permitido por subida (configurable en FastAPI o middleware).

Se recomienda usar `@lru_cache` en un *dependency* de FastAPI para instanciar una sola vez `Settings`, según la documentación oficial [2].

Compatibilidad e integración con otros servicios

Para mantener coherencia con el ecosistema:

- **Rutas y endpoints**: Respetar los paths definidos (`/files`, `/healthz`, `/metrics`) y métodos HTTP. Otros servicios podrán invocar estos endpoints o el API Gateway lo hará.
- **Eventos Kafka**: Publicar el evento `file_stored` en el tópico compartido (configurado en `KAFKA_TOPIC`). Se debe usar el esquema de mensaje acordado (`FileStoredEvent`), serializando a JSON. El consumidor (otro microservicio) esperará este formato.
- **JWT**: El gateway envía JWT en la cabecera `Authorization`. El servicio puede usarlo para extraer el `user_id` o `sub`. Por ejemplo, crear una dependencia que decodifique el JWT (sin validar si ya confía en el gateway) y retorne `user_id`. Así, cada archivo queda asociado al usuario.
- **Variables de entorno**: Mantener nombres estándar. Usar las variables indicadas facilita la integración en Kubernetes o Docker Compose.
- **Métricas y monitoreo**: Exponer `/metrics` y `/healthz` permite que Prometheus y Kubernetes comprueben salud del servicio.
- **Persistencia de archivos**: Definir un volumen persistente (`/data/files`) compartido por la instancia del servicio. En producción en k3s, usar un PersistentVolumeClaim con un storage class (p.ej. NFS, Ceph, etc.).

Se recomienda documentar el contrato de eventos y rutas en la wiki del proyecto. Así, cualquier cambio futuro en el esquema o rutas será conocido por todos los equipos. Además, versionar la imagen Docker y los endpoints (v1) en caso de evolución.

Docker y despliegue

Ejemplo básico de **Dockerfile**:

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

En **docker-compose.yml**, se monta un volumen persistente para los archivos y se vincula a Kafka:

```
version: '3.8'
services:
  file_service:
    build: .
    ports:
      - "8000:8000"
    environment:
```

```
      - FILE_BASE_PATH=/data/files
      - KAFKA_BOOTSTRAP=kafka:9092
      - FILE_PUBLIC_URL=http://localhost:8000
      - MAX_UPLOAD_MB=50
    volumes:
      - file_data:/data/files
    depends_on:
      - kafka
  kafka:
    image: confluentinc/cp-kafka:latest
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    ports:
      - "9092:9092"
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

volumes:
  file_data:
```

En **k3s/Kubernetes**, usar un Deployment basado en la imagen construida y un PersistentVolumeClaim para `/data/files`. También definir un ConfigMap o Secret para variables de entorno.

Pruebas unitarias e integración

Se recomienda usar **pytest** y el **TestClient** de FastAPI para pruebas unitarias [5]. Algunos pasos prácticos:

1. **Instalar dependencias de test**: En el entorno virtual o Dockerfile, incluir `pytest`, `httpx`, `pytest-asyncio`.
2. **Pruebas de storage**: En `app/tests/test_storage.py`, crear tests que verifiquen `LocalFSBackend` (por ejemplo, guardar y leer datos en un directorio temporal).
3. **Pruebas de API**: En `app/tests/test_files_api.py`, usar `TestClient(app)` para hacer requests a los endpoints:

```
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_upload_and_download(tmp_path, monkeypatch):
    # Configurar storage en ruta temporal
    monkeypatch.setenv("FILE_BASE_PATH", str(tmp_path))
    # Simular un archivo
```

```
    file_content = b"datos de prueba"
    response = client.post("/files", files={"file": ("test.txt", file_content)})
    assert response.status_code == 200
    data = response.json()
    key = data["key"]
    # Descargar el archivo
    resp = client.get(f"/files/{key}")
    assert resp.content == file_content
```

1. **Pruebas health/metrics**: Verificar que `/healthz` retorne 200 cuando el servicio está operando (esto implica tener Kafka accesible, puede usarse un *mock* o una instancia real de prueba). Probar que `/metrics` existe.

2. **Comandos de ejecución**:

3. Ejecución de tests unitarios:

```
pytest --maxfail=1 --disable-warnings -q
```

4. En Docker Compose, levantar servicios y probar en conjunto:

```
docker-compose up -d
pytest  # corre tests con Kafka local
```

5. **Mocking de Kafka**: Para pruebas unitarias puras, puede *mockearse* `confluent_kafka.Producer` o usar una configuración de Kafka embebido. Para pruebas de integración, usar Kafka real en Docker (ver *depends_on* arriba) o un cluster de prueba.

El manual de FastAPI sugiere usar `TestClient` (Starlette) y pytest para API testing [5]. Asegúrese de aislar entornos de prueba (por ejemplo, cambiar `FILE_BASE_PATH` a un directorio temporal) usando fixtures de pytest.

Resumen

Este diseño provee un servicio de archivos basado en FastAPI, con almacenamiento local extensible a S3/NFS (mediante la interfaz `StorageBackend`), eventos Kafka para notificar subidas (`file_stored`) y métricas para monitoreo. Se utilizan variables de entorno para configuración, se exponen endpoints REST definidos y se incluye un healthcheck. La estructura del proyecto y las pruebas permiten escalabilidad y mantenibilidad dentro de una arquitectura de microservicios. Los fragmentos de código arriba muestran la implementación crítica (gestión de archivos, eventos, métricas) de forma ilustrativa.

Fuentes: Documentación oficial de FastAPI (subida de archivos) [1], guía Prometheus/FastAPI (montar `/metrics`) [4], y enfoque de configuración con Pydantic [2] y testing con TestClient [5].

1 Request Files - FastAPI
https://fastapi.tiangolo.com/tutorial/request-files/

2 Settings and Environment Variables - FastAPI
https://fastapi.tiangolo.com/advanced/settings/

3 GitHub - confluentinc/confluent-kafka-python: Confluent's Kafka Python Client
https://github.com/confluentinc/confluent-kafka-python

4 FastAPI + Gunicorn | client_python
https://prometheus.github.io/client_python/exporting/http/fastapi-gunicorn/

5 Testing - FastAPI
https://fastapi.tiangolo.com/tutorial/testing/