Informe de Proyecto: Aplicación "Postits" con Arquitectura de 3 Contenedores Docker

Tarea de Despliegue de Aplicaciones

Mark Romero Guillen
Arequipa, Perú

4 de septiembre de 2025

Índice

1. Introducción

El presente informe detalla la configuración y despliegue de una aplicación web simple denominada "Postits". El objetivo principal de este proyecto es demostrar la implementación de una arquitectura de tres capas (frontend, backend, base de datos) utilizando contenedores Docker, orquestados mediante Docker Compose.

La aplicación permite a los usuarios crear, visualizar y eliminar notas de texto (o "postits") que se persisten en una base de datos. El desarrollo y la ejecución se realizaron en un entorno Pop!_OS con Docker Desktop.

2. Arquitectura General

La aplicación sigue un patrón de arquitectura de tres niveles, donde cada nivel es un servicio independiente encapsulado en su propio contenedor Docker.

1. **Frontend:** Es la capa de presentación, responsable de la interfaz de usuario (UI). Se trata de una página web estática (HTML, CSS, JavaScript) que interactúa con el backend a través de peticiones HTTP (API REST).

2. **Backend (API):** Es la capa de lógica de negocio. Expone una API RESTful que el frontend consume. Se encarga de procesar las peticiones, aplicar la lógica de la aplicación y comunicarse con la base de datos para almacenar o recuperar datos.

3. **Base de Datos:** Es la capa de persistencia. Su única función es almacenar los datos de la aplicación de forma segura y duradera.

La comunicación entre los contenedores se gestiona a través de una red privada virtual creada por Docker Compose, lo que permite que los servicios se comuniquen entre sí utilizando sus nombres de servicio como si fueran nombres de host.

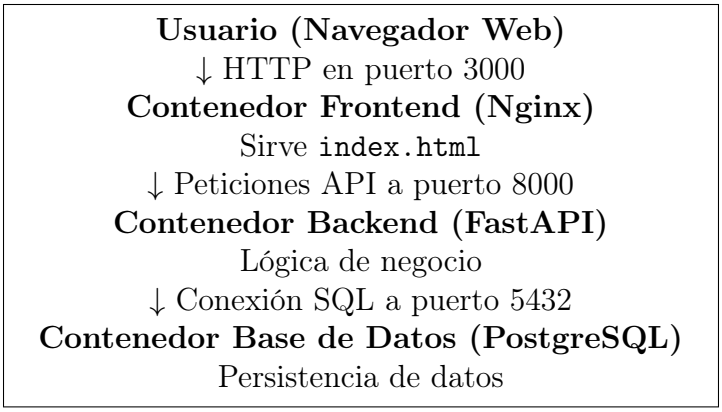

Figura 1: Flujo de comunicación en la arquitectura de 3 niveles.

3. Análisis de Componentes

3.1. Frontend

- **Tecnología:** HTML5, CSS3 y JavaScript puro (vanilla JS).

- **Servidor Web:** Nginx (versión Alpine para un tamaño de imagen mínimo).
- **Función:** Proporciona la interfaz gráfica para que el usuario interactúe con la aplicación. El código JavaScript realiza llamadas `fetch` al servicio backend para obtener, crear y eliminar posts.

El `Dockerfile` para el frontend es extremadamente simple. Su única tarea es copiar el archivo estático `index.html` en el directorio raíz del servidor Nginx.

```
# Imagen muy pequena para estaticos
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
# nginx expone 80 por defecto
```

Listing 1: Dockerfile del Frontend

3.2. Backend (API)

- **Tecnología:** Python 3.12 con el framework FastAPI.
- **Servidor ASGI:** Uvicorn.
- **Conexión a BD:** `psycopg`, el driver moderno para PostgreSQL en Python.
- **Función:** Expone los siguientes endpoints para gestionar los posts:
  - `GET /posts`: Lista todos los posts.
  - `POST /posts`: Crea un nuevo post.
  - `DELETE /posts/{id}`: Elimina un post específico.
  - `GET /healthz`: Endpoint de chequeo de salud.

El código Python incluye una función `wait_for_db` que realiza una espera activa, intentando conectarse a la base de datos en un bucle antes de iniciar el pool de conexiones. Esto es crucial en un entorno de contenedores para evitar que la API falle si se inicia antes de que la base de datos esté completamente lista.

El `Dockerfile` del backend utiliza una imagen base de Python Alpine para mantener un tamaño reducido. Instala las dependencias de `requirements.txt` y luego ejecuta la aplicación con Uvicorn.

```
# Imagen pequena, sin apt-get
FROM python:3.12-alpine

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1 \
    PIP_NO_CACHE_DIR=1

WORKDIR /app

# Instalar deps Python (todas en wheel/binario)
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copiar codigo
```

```
COPY app.py .

# Variables por defecto (puedes sobreescribir en compose)
ENV PORT=8000
EXPOSE 8000

# Arranque del servidor ASGI
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Listing 2: Dockerfile del Backend

3.3. Base de Datos

- **Tecnología:** PostgreSQL 16 (versión Alpine).
- **Persistencia:** Se utiliza un **volumen de Docker** llamado `db_data`. Este volumen mapea el directorio de datos de PostgreSQL dentro del contenedor (`/var/lib/postgresql/data`) a un espacio de almacenamiento gestionado por Docker en el host. Esto garantiza que los datos no se pierdan si el contenedor se detiene o se elimina.

4. Orquestación con Docker Compose

El archivo `docker-compose.yml` es el núcleo de la orquestación. Define los tres servicios (`db`, `api`, `frontend`), sus configuraciones, y cómo se relacionan entre sí.

```
version: "3.8"

services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: postits
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d postits"]
      interval: 5s
      timeout: 3s
      retries: 20
    # ...

  api:
    build:
      context: ./backend
    environment:
      DATABASE_URL: postgresql://postgres:password@db:5432/postits
    depends_on:
      db:
        condition: service_healthy
    # ...

  frontend:
    build:
```

```
      context: ./frontend
    ports:
      - "3000:80"
    depends_on:
      - api
    # ...

volumes:
  db_data:
```

Listing 3: Fragmento de docker-compose.yml

4.1. Interacción y Dependencias

La interacción entre contenedores se logra mediante características clave de Docker Compose:

- **Red Interna:** Docker Compose crea una red por defecto para el proyecto. Todos los servicios dentro de esta red pueden comunicarse entre sí usando su nombre de servicio como si fuera un DNS. Por ejemplo, el servicio `api` se conecta a la base de datos en la URL `postgresql://...@db:5432`, donde `db` es el nombre del servicio de la base de datos.
- **Control de Arranque con `depends_on` y `healthcheck`:** El servicio `api` tiene una dependencia explícita del servicio `db`. La condición `service_healthy` le indica a Docker Compose que no debe iniciar el contenedor `api` hasta que el `healthcheck` del contenedor `db` sea exitoso. El `healthcheck` del servicio `db` utiliza el comando `pg_isready` para verificar que PostgreSQL no solo esté en ejecución, sino que también esté listo para aceptar conexiones. Esto previene errores de conexión al iniciar la pila de servicios.
- **Mapeo de Puertos:**
  - El Frontend (`frontend`) mapea el puerto 3000 del host al puerto 80 del contenedor Nginx. El usuario accede a la aplicación a través de http://localhost:3000.
  - El Backend (`api`) mapea el puerto 8000 del host al 8000 del contenedor. Esto permite que el JavaScript del frontend (ejecutándose en el navegador del usuario) pueda hacer peticiones a http://localhost:8000.
  - La Base de Datos (`db`) mapea el puerto 5432 para permitir la conexión directa desde el host con herramientas de gestión de bases de datos si fuera necesario para depuración.

5. Configuración y Ejecución

Para ejecutar este proyecto, se deben seguir los siguientes pasos:

1. **Prerrequisitos:** Tener instalado Docker y Docker Compose (Docker Desktop en Pop!_OS los incluye a ambos).
2. **Estructura de Archivos:** Organizar los archivos en la siguiente estructura:

```
.
|-- docker-compose.yml
|-- backend/
|   |-- Dockerfile
|   |-- app.py
|   |-- requirements.txt
|-- frontend/
    |-- Dockerfile
    |-- index.html
```

3. **Levantar los Servicios:** Abrir una terminal en el directorio raíz del proyecto y ejecutar el siguiente comando:

   ```
   docker-compose up --build
   ```

   El flag `-build` asegura que las imágenes de los contenedores se construyan a partir de los `Dockerfile` la primera vez o si han sufrido cambios.

4. **Acceder a la Aplicación:** Una vez que todos los contenedores estén en funcionamiento, se puede acceder a la interfaz de la aplicación abriendo un navegador web en la dirección http://localhost:3000.

6. Conclusión

Este proyecto demuestra de manera efectiva la creación y orquestación de una aplicación web moderna de tres capas utilizando Docker y Docker Compose. La separación de responsabilidades en contenedores independientes (frontend, backend, base de datos) facilita el desarrollo, el despliegue y el mantenimiento.

Aspectos clave como la comunicación en red interna, el control del orden de arranque con `healthchecks`, y la persistencia de datos mediante volúmenes han sido implementados correctamente, resultando en una arquitectura robusta y escalable, lista para entornos de desarrollo y producción.

7. Código Fuente

El código fuente completo de este proyecto, incluyendo los archivos `Dockerfile`, el código de la aplicación y el archivo `docker-compose.yml`, está disponible en el siguiente repositorio de GitHub:

github.com/marcksdbgg/notas_contenedores