1. ¿Qué es un Dockerfile?

Un **Dockerfile** es un archivo de texto que contiene instrucciones para construir una imagen de Docker. Define el entorno necesario para ejecutar una aplicación dentro de un contenedor, incluyendo la instalación de dependencias, la configuración del sistema y la ejecución de comandos. Las instrucciones más comunes en un Dockerfile son:

- `FROM`: Especifica la imagen base.
- `RUN`: Ejecuta comandos durante la construcción de la imagen.
- `COPY`: Copia archivos desde el host al contenedor.
- `CMD`: Define el comando predeterminado que se ejecutará cuando el contenedor arranque.

---

2. Estructura del proyecto

El proyecto tendrá la siguiente estructura:

```
proyecto-flask/
├── app.py # Código principal de la aplicación Flask
├── requirements.txt # Dependencias de Python
├── Dockerfile # Archivo para construir la imagen Docker
└── db.sqlite3 # Base de datos SQLite (se generará automáticamente)
```

---

3. Implementación

Archivo `requirements.txt`

Este archivo define las dependencias necesarias para la aplicación. Incluye Flask y SQLite.

```
Flask==2.3.2
```

Archivo `app.py`

Este es el código principal de la aplicación Flask. Define las tres APIs (`presión`, `temperatura` y `humedad`) y genera datos aleatorios cada 60 segundos. OJO!!!!!! es python, riguroso en la sangría (indentación). Copy/paste puede dar lugar a errores.

```
from flask import Flask, jsonify
import sqlite3
import random
import threading
import time
```

```
app = Flask(__name__)
DATABASE = "db.sqlite3"

# Función para inicializar la base de datos
def init_db():
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS readings (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            tipo TEXT NOT NULL,
            valor REAL NOT NULL,
            timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
        )
    """)
    conn.commit()
    conn.close()

# Función para insertar datos en la base de datos
def insert_reading(tipo, valor):
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("INSERT INTO readings (tipo, valor) VALUES (?, ?)", (tipo, valor))
    conn.commit()
    conn.close()

# Función para obtener los datos más recientes
def get_latest_reading(tipo):
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("SELECT valor, timestamp FROM readings WHERE tipo = ? ORDER BY
timestamp DESC LIMIT 1", (tipo,))
    result = cursor.fetchone()
    conn.close()
    return result

# Generar datos aleatorios cada 60 segundos
def generate_data():
    while True:
        pressure = round(random.uniform(900, 1100), 2)  # Presión entre 900 y 1100 hPa
        temperature = round(random.uniform(-10, 40), 2)  # Temperatura entre -10 y 40 °C
        humidity = round(random.uniform(0, 100), 2)  # Humedad entre 0% y 100%

        insert_reading("pressure", pressure)
        insert_reading("temperature", temperature)
        insert_reading("humidity", humidity)
```

```
        time.sleep(60)

# Rutas de la API
@app.route("/api/pressure", methods=["GET"])
def api_pressure():
    reading = get_latest_reading("pressure")
    if reading:
        return jsonify({"tipo": "presión", "valor": reading[0], "timestamp": reading[1]})
    return jsonify({"error": "No hay datos disponibles"}), 404

@app.route("/api/temperature", methods=["GET"])
def api_temperature():
    reading = get_latest_reading("temperature")
    if reading:
        return jsonify({"tipo": "temperatura", "valor": reading[0], "timestamp": reading[1]})
    return jsonify({"error": "No hay datos disponibles"}), 404

@app.route("/api/humidity", methods=["GET"])
def api_humidity():
    reading = get_latest_reading("humidity")
    if reading:
        return jsonify({"tipo": "humedad", "valor": reading[0], "timestamp": reading[1]})
    return jsonify({"error": "No hay datos disponibles"}), 404

# Inicializar la base de datos y comenzar a generar datos
if __name__ == "__main__":
    init_db()
    threading.Thread(target=generate_data, daemon=True).start()
    app.run(host="0.0.0.0", port=5000)
```

Archivo `Dockerfile`

Este archivo define cómo construir la imagen Docker basada en Alpine Linux.

```
# Usar Alpine como imagen base
FROM alpine:latest

# Instalar dependencias básicas
RUN apk add --no-cache python3 py3-pip bash

# Crear directorio de trabajo
WORKDIR /app
```

```
# Copiar archivos necesarios
COPY requirements.txt .
COPY app.py .

# Instalar dependencias de Python
RUN pip3 install --no-cache-dir -r requirements.txt

# Exponer el puerto 5000
EXPOSE 5000

# Comando para iniciar la aplicación
CMD ["python3", "app.py"]
```

---

4. Documentación de la API

La API tiene tres endpoints accesibles mediante el método GET:

Endpoint	Descripción
`/api/pressure`	Devuelve la última lectura de presión en formato JSON.
`/api/temperature`	Devuelve la última lectura de temperatura en formato JSON.
`/api/humidity`	Devuelve la última lectura de humedad en formato JSON.

Ejemplo de respuesta:

```
{
"tipo": "presión",
"valor": 1013.25,
"timestamp": "2023-10-01T12:34:56"
}
```

---

5. Construcción y ejecución del contenedor

Paso 1: Construir la imagen Docker

En el directorio del proyecto, ejecuta:

```
docker build -t flask-alpine-app .
```

Paso 2: Ejecutar el contenedor

Inicia el contenedor mapeando el puerto 5000:

```
docker run -d -p 5000:5000 --name flask-app flask-alpine-app
```

Paso 3: Probar la API

Accede a las APIs desde tu navegador o usando `curl`:

curl http://localhost:5000/api/pressure
curl http://localhost:5000/api/temperature
curl http://localhost:5000/api/humidity

---

6. Resultado esperado

- La aplicación generará datos aleatorios de presión, temperatura y humedad cada 60 segundos y los almacenará en la base de datos SQLite.
- Las APIs estarán disponibles para consultar los datos más recientes.
- Todo funcionará dentro de un contenedor ligero basado en Alpine Linux.