Proyecto Final

Despliegue de aplicación con contenedores: Local y Nube

Infraestructuras paralelas y distribuidas

Estudiantes:

Tina María Torres - 2259729

Marlon Astudillo Muñoz - 2259462

Juan José Gallego Calderón - 2259433

Juan José Hernández Arenas - 2259500

Docente:

Carlos Andrés Delgado

Sede Tuluá

Diciembre 2024

Índice

Implementación de una Calculadora con Arquitectura de Microservicios utilizando Docker

1. Introducción

El presente informe documenta el desarrollo de una aplicación calculadora utilizando una arquitectura de microservicios y la tecnología de contenedorización Docker. El objetivo principal de este proyecto es demostrar la aplicación práctica de los principios de microservicios, separando la lógica de la interfaz de usuario en componentes independientes y facilitando su gestión y escalabilidad mediante el uso de contenedores Docker. Se justifica esta elección arquitectónica por las ventajas que ofrece en términos de modularidad, mantenibilidad, portabilidad y escalabilidad, en comparación con un enfoque monolítico tradicional. El alcance de este proyecto se centra en la implementación de las operaciones aritméticas básicas (suma, resta, multiplicación y división), excluyendo el despliegue en la nube debido a limitaciones técnicas encontradas durante el desarrollo. Este informe se estructura de la siguiente manera: se presenta un análisis y diseño de la arquitectura, seguido de la descripción de la implementación, las pruebas realizadas y, finalmente, las conclusiones obtenidas.

2. Análisis y Diseño

2.1 Arquitectura Inicial (Enfoque Monolítico)

La concepción inicial del proyecto consideraba una implementación monolítica de la calculadora, similar a una aplicación Java Swing. Este enfoque, si bien es válido para aplicaciones sencillas, presenta limitaciones significativas en escenarios donde se requiere escalabilidad, mantenimiento independiente de componentes o despliegues frecuentes. La tightly coupling entre la interfaz de usuario y la lógica de negocio dificulta la evolución y el mantenimiento del sistema a largo plazo.

2.2 Arquitectura Propuesta (Microservicios)

Para superar las limitaciones del enfoque monolítico, se adoptó una arquitectura de microservicios, dividiendo la aplicación en dos componentes principales:

- ★ **Frontend:** Responsable de la interacción con el usuario, implementado con tecnologías web (HTML, CSS, JavaScript).
- ★ **Backend:** Encargado de la lógica de la calculadora, implementado con Python y el framework Flask, exponiendo una API RESTful.

2.2.1 Descripción de los Microservicios

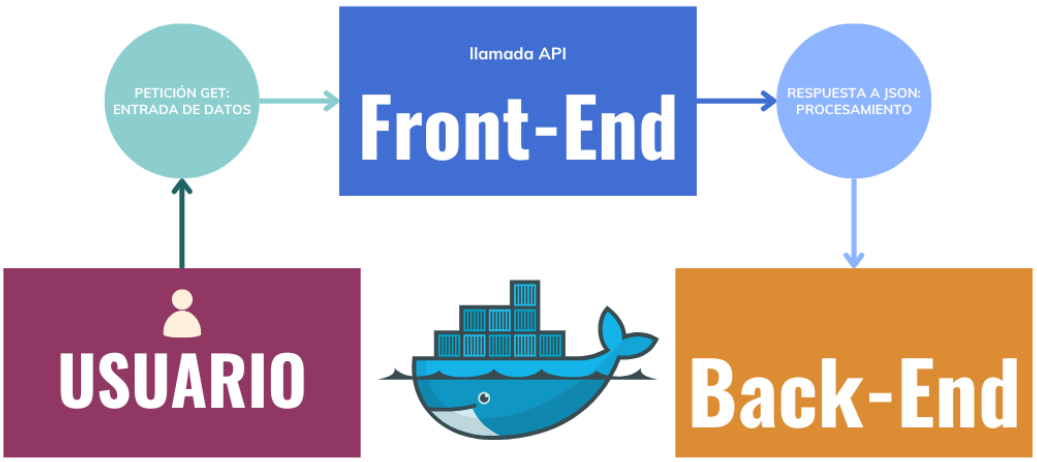
- ★ **Frontend:** Implementado utilizando HTML, CSS y JavaScript. Se encarga de presentar la interfaz de usuario al usuario, capturar las entradas y realizar las llamadas a la API del backend para obtener los resultados de las operaciones.
- ★ **Backend (Flask):** Implementado con Python y el framework Flask. Expone una API RESTful que recibe las solicitudes del frontend, realiza los cálculos y devuelve los resultados en formato JSON.

2.2.2 API REST

La comunicación entre el frontend y el backend se realiza mediante una API RESTful. A continuación, se detalla la especificación de los endpoints:

Endpoint	Método	Parámetros	Respuesta (JSON)	Códigos de Estado
/sumar	GET	a (float), b (float)	{"resultado": valor} o {"error": "Mensaje"}	200, 400, 500
/restar	GET	a (float), b (float)	{"resultado": valor} o {"error": "Mensaje"}	200, 400, 500
/multiplicar	GET	a (float), b (float)	{"resultado": valor} o {"error": "Mensaje"}	200, 400, 500
/dividir	GET	a (float), b (float)	{"resultado": valor} o {"error": "Mensaje"}	200, 400, 500

2.2.3 Flujo de Datos



- ❖ **Entrada del usuario en el Frontend:** El usuario interactúa con la interfaz web (frontend) e ingresa los números que desea operar y la operación a realizar (suma, resta, etc.).
- ❖ **Preparación de la Petición:** El frontend, utilizando JavaScript, toma los datos ingresados por el usuario y los formatea para enviarlos al backend. Generalmente, esto se hace creando una petición HTTP, que puede ser de tipo GET o POST. En nuestro caso, al usar parámetros en la URL (ej. `/sumar?a=5&b=3`), se trata de una petición GET. Los datos se incluyen en la URL como pares clave-valor.

- **Envío de la Petición a la API del Backend:** El frontend envía la petición HTTP a la API REST que expone el backend. Esta petición contiene la información necesaria para que el backend realice la operación.
- **Recepción de la Petición en el Backend:** El backend (Flask) recibe la petición HTTP. El framework Flask se encarga de extraer los parámetros de la URL (en el caso de una petición GET) o del cuerpo de la petición (en el caso de una petición POST).
- **Procesamiento de la Petición en el Backend:** El backend toma los datos recibidos, realiza la operación matemática correspondiente (suma, resta, multiplicación o división) y genera un resultado.
- **Formato de la Respuesta:** El backend formatea el resultado en un formato que el frontend pueda entender, generalmente JSON (JavaScript Object Notation). JSON es un formato ligero de intercambio de datos, fácil de leer y de procesar tanto por máquinas como por humanos. Por ejemplo, una respuesta podría ser {"resultado": 8}.
- **Envío de la Respuesta al Frontend:** El backend envía la respuesta en formato JSON al frontend a través de la conexión HTTP.
- **Recepción de la Respuesta en el Frontend:** El frontend recibe la respuesta JSON del backend.
- **Presentación del Resultado al Usuario:** El frontend, utilizando JavaScript, extrae el resultado del JSON recibido y lo muestra al usuario en la interfaz web.

2.2.3.1 Ejemplo concreto (suma)

1. El usuario ingresa "5" y "3" en los campos correspondientes y selecciona la operación "sumar".
2. El frontend crea la URL /sumar?a=5&b=3.
3. El frontend envía una petición GET a esa URL.
4. El backend recibe la petición y extrae los valores a=5 y b=3.
5. El backend realiza la suma: 5 + 3 = 8.
6. El backend crea el JSON {"resultado": 8}.
7. El backend envía este JSON al frontend.
8. El frontend recibe el JSON.
9. El frontend muestra "8" en la pantalla.

3. Implementación

3.1 Backend (Flask)

El backend se implementó utilizando Python y el framework Flask. Su función principal es exponer una API RESTful que gestiona las operaciones de la calculadora. Cada operación (suma, resta, multiplicación, división) se implementa como un endpoint separado.

3.1.1 Estructura del Código

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
from decimal import Decimal
import psycopg2
import os

app = Flask(__name__)

# Permite solo solicitudes desde localhost
CORS(app, resources={r"/api/*": {"origins": "http://localhost"}})

def get_db_connection():
    return psycopg2.connect(
        dbname=os.getenv('DB_NAME', 'calculos'),
        user=os.getenv('DB_USER', 'postgres'),
        password=os.getenv('DB_PASSWORD', 'postgres'),
        host=os.getenv('DB_HOST', 'db'),
        port=os.getenv('DB_PORT', '5432')
    )

@app.route('/api/calcular', methods=['POST'])
def calcular():
    try:
        data = request.get_json()
        operacion = data['operacion']
        resultado = eval(operacion)
        resultado_decimal = Decimal(str(resultado))

        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO calculos (operacion, resultado) VALUES (%s, %s)",
            (operacion, resultado_decimal)
        )
        conn.commit()
        cursor.close()
        conn.close()

        return jsonify({"resultado": float(resultado_decimal)})

    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/api/historial', methods=['GET'])
def historial():
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM calculos ORDER BY fecha DESC")
        rows = cursor.fetchall()
        cursor.close()
        conn.close()

        historial = [
            {
                "id": row[0],
                "operacion": row[1],
                "resultado": float(row[2]),
                "fecha": row[3].strftime('%Y-%m-%d %H:%M:%S')
            }
            for row in rows
        ]

        return jsonify(historial)

    except Exception as e:
        return jsonify({"error": str(e)}), 500


if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

1.Importaciones:

- **from flask import Flask, request, jsonify:** Importa las clases necesarias de Flask para crear la aplicación web y manejar las solicitudes y respuestas.
- **from flask_cors import CORS:** Importa CORS (Cross-Origin Resource Sharing - Intercambio de Recursos de Origen Cruzado) para permitir que el frontend, que probablemente se ejecuta en un dominio diferente (ej. http://localhost:3000), pueda realizar solicitudes al backend.
- **from decimal import Decimal:** Importa la clase Decimal para manejar números decimales con mayor precisión que los tipos float nativos de Python. Esto es crucial para evitar errores de redondeo en cálculos financieros o que requieran alta precisión.
- **import psycopg2:** Importa la biblioteca psycopg2, que es el adaptador de PostgreSQL más popular para Python. Se utiliza para interactuar con la base de datos PostgreSQL.
- **import os:** Importa el módulo os para acceder a variables de entorno. Esto se utiliza para configurar la conexión a la base de datos sin tener que codificar las credenciales directamente en el código.

2. Configuración de la Aplicación:

- **app = Flask(__name__):** Crea una instancia de la aplicación Flask. __name__ es una variable especial que representa el nombre del módulo actual.
- **CORS(app, resources={r"/api/*": {"origins": "http://localhost"}}):** Configura CORS para la aplicación. resources={r"/api/*": {"origins": "http://localhost"}} especifica que solo se permiten solicitudes desde el origen http://localhost para todas las rutas que comiencen con /api/. Esto es importante por seguridad, ya que evita que otros sitios web maliciosos realicen solicitudes a tu backend.

3. Función de Conexión a la Base de Datos:

- **def get_db_connection():** Define una función que establece una conexión a la base de datos PostgreSQL.Utiliza os.getenv() para obtener las credenciales y la información de conexión de las variables de entorno:
  - **DB_NAME:** Nombre de la base de datos (por defecto 'calculos').
  - **DB_USER:** Nombre de usuario de la base de datos (por defecto 'postgres').
  - **DB_PASSWORD:** Contraseña de la base de datos (por defecto 'postgres').
  - **DB_HOST:** Host de la base de datos (por defecto 'db').
  - **DB_PORT:** Puerto de la base de datos (por defecto '5432').

Esto permite configurar la conexión a la base de datos sin modificar el código, lo cual es una buena práctica.

4. Endpoint de Cálculo (/api/calcular):

- **@app.route('/api/calcular', methods=['POST']):** Define un manejador de ruta para el endpoint /api/calcular, que solo acepta solicitudes POST. Las solicitudes POST generalmente se utilizan para enviar datos al servidor.
- **try...except:** Bloque para manejar posibles excepciones durante la ejecución del código.

- **data = request.get_json():** Obtiene los datos JSON enviados por el frontend en el cuerpo de la solicitud.
- **operacion = data['operacion']:** Extrae la cadena que representa la operación matemática del JSON recibido.
- **resultado = eval(operacion):** Utiliza la función eval() de Python para evaluar la expresión matemática.
- **resultado_decimal = Decimal(str(resultado)):** Convierte el resultado a un objeto Decimal para mantener la precisión decimal.
- **Conexión y operación con la base de datos:**
  - **conn = get_db_connection():** Establece una conexión a la base de datos.
  - **cursor = conn.cursor():** Crea un cursor para ejecutar comandos SQL.
  - **cursor.execute("INSERT INTO calculos (operacion, resultado) VALUES (%s, %s)", (operacion, resultado_decimal)):** Ejecuta una consulta SQL para insertar la operación y el resultado en la tabla calculos.
  - **conn.commit():** Guarda los cambios en la base de datos.
  - **cursor.close():** Cierra el cursor.
  - **conn.close():** Cierra la conexión a la base de datos.
- **return jsonify({"resultado": float(resultado_decimal)}):** Devuelve una respuesta JSON con el resultado del cálculo, convertido a float para que sea compatible con JavaScript en el frontend.

5. Endpoint de Historial (/api/historial):

- **@app.route('/api/historial', methods=['GET']):** Define un manejador de ruta para el endpoint /api/historial, que acepta solicitudes GET.Similar al endpoint /api/calcular, utiliza un bloque try...except y establece una conexión a la base de datos.
- **cursor.execute("SELECT * FROM calculos ORDER BY fecha DESC"):** Ejecuta una consulta SQL para obtener todas las entradas de la tabla calculos, ordenadas por fecha de forma descendente (las más recientes primero).
- **rows = cursor.fetchall():** Obtiene todos los resultados de la consulta.
- Recorre las filas obtenidas y crea una lista de diccionarios, donde cada diccionario representa un registro del historial:
  - **"id":** El ID del registro.
  - **"operacion":** La operación realizada.
  - **"resultado":** El resultado del cálculo (convertido a float).
  - **"fecha":** La fecha y hora del cálculo, formateada como una cadena.
- **return jsonify(historial):** Devuelve una respuesta JSON con la lista del historial.

6. Ejecución de la Aplicación:

- **if __name__ == '__main__'::** Asegura que el código dentro de este bloque solo se ejecute cuando el script se ejecuta directamente (no cuando se importa como un módulo).
- **app.run(debug=True, host='0.0.0.0', port=5000):** Inicia el servidor de desarrollo de Flask.

- **debug=True:** Activa el modo de depuración, que proporciona información útil para el desarrollo, como la recarga automática del servidor al modificar el código.
- **host='0.0.0.0':** Hace que el servidor escuche en todas las interfaces de red, lo cual es necesario para que sea accesible desde dentro de un contenedor Docker.
- **port=5000:** Especifica el puerto en el que se ejecutará el servidor.

3.1.2 Base de Datos (PostgreSQL)

Se utiliza PostgreSQL como sistema de gestión de bases de datos relacional para almacenar el historial de cálculos. La tabla principal se llama **cálculos** y tiene la siguiente estructura:

```
CREATE TABLE calculos (
    id SERIAL PRIMARY KEY,
    operacion VARCHAR(255),
    resultado NUMERIC,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- ➔ **id SERIAL PRIMARY KEY:** Define una columna llamada id de tipo SERIAL. SERIAL es un tipo de dato especial en PostgreSQL que crea una secuencia automática para generar números enteros únicos. PRIMARY KEY indica que esta columna es la clave primaria de la tabla, lo que significa que identifica de forma única cada fila en la tabla.
- ➔ **operacion VARCHAR(255):** Define una columna llamada operacion de tipo VARCHAR(255). VARCHAR almacena cadenas de texto de longitud variable, con un máximo de 255 caracteres en este caso. Esta columna almacenará la expresión matemática que se calculó (por ejemplo, "2 + 2").
- ➔ **resultado NUMERIC:** Define una columna llamada resultado de tipo NUMERIC. Este tipo de dato es ideal para almacenar números con precisión arbitraria, evitando los problemas de redondeo que pueden ocurrir con los tipos FLOAT o REAL. Almacenará el resultado del cálculo.
- ➔ **fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP:** Define una columna llamada fecha de tipo TIMESTAMP. Este tipo de dato almacena la fecha y la hora. DEFAULT CURRENT_TIMESTAMP asigna automáticamente la fecha y hora actual al registrar una nueva fila, lo que registra el momento exacto en que se realizó el cálculo.

El backend (Flask) interactúa con esta base de datos a través de la biblioteca psycopg2. Cuando se realiza un nuevo cálculo, el backend inserta una nueva fila en la tabla cálculos con la operación, el resultado y la fecha actual. Cuando se solicita el historial, el backend consulta la tabla cálculos y devuelve los registros en formato JSON al frontend.

3.1.3 Dockerización del Backend

Para facilitar el despliegue y la portabilidad del backend, se utiliza Docker para contenerizar la aplicación. A continuación, se describe el *Dockerfile* utilizado:

```dockerfile
# Usar una imagen base oficial de Python
FROM python:3.9

# Establecer el directorio de trabajo en el contenedor
WORKDIR /app

# Copiar los archivos necesarios al contenedor
COPY backend.py /app/
COPY requirements.txt /app/

# Instalar dependencias
RUN pip install --no-cache-dir -r requirements.txt

# Exponer el puerto donde correrá el backend
EXPOSE 5000

# Comando para ejecutar la aplicación
CMD ["python", "backend.py"]
```

- ➔ **FROM python:3.9:** Esta línea especifica la imagen base que se utilizará para construir la imagen del contenedor. Se utiliza la imagen oficial de Python 3.9, que proporciona un entorno preconfigurado con Python.
- ➔ **WORKDIR /app:** Establece el directorio de trabajo dentro del contenedor a /app. Todos los comandos posteriores se ejecutarán dentro de este directorio.
- ➔ **COPY backend.py /app/:** Copia el archivo backend.py (tu script principal de Flask) al directorio /app dentro del contenedor.
- ➔ **COPY requirements.txt /app/:** Copia el archivo requirements.txt al directorio /app dentro del contenedor. Este archivo contiene la lista de dependencias de tu proyecto.
- ➔ **RUN pip install --no-cache-dir -r requirements.txt:** Ejecuta el comando pip install para instalar las dependencias listadas en requirements.txt. El flag --no-cache-dir evita el uso de la caché de pip, asegurando que se instalen las versiones más recientes de las dependencias.
- ➔ **EXPOSE 5000:** Expone el puerto 5000 del contenedor. Este es el puerto en el que se ejecuta la aplicación Flask.
- ➔ **CMD ["python", "backend.py"]:** Define el comando que se ejecutará cuando se inicie el contenedor. En este caso, ejecuta el script backend.py con Python.

3.1.4 Orquestación con Docker Compose

Para simplificar la gestión de los contenedores de la base de datos, el backend y el frontend, se utiliza Docker Compose. A continuación, se describe el archivo *docker-compose.yml* utilizado:

```yaml
version: '3.8'

services:
  db:
    image: postgres:13
    container_name: postgres_container
    environment:
      POSTGRES_DB: calculos
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: backend_container
    environment:
      - DB_HOST=db  # Dirección del servicio PostgreSQL
      - DB_PORT=5432
      - DB_NAME=calculos
      - DB_USER=postgres
      - DB_PASSWORD=postgres
    ports:
      - "5000:5000"
    depends_on:
      - db

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: frontend_container
    ports:
      - "80:80"
    depends_on:
      - backend

volumes:
  db_data:
```

- ➔ **services:** Define los servicios que se ejecutarán.
- ➔ **db:** Define el servicio de la base de datos PostgreSQL.
- ➔ **image:** postgres:13: Utiliza la imagen oficial de PostgreSQL versión 13.
- ➔ **container_name: postgres_container:** Asigna el nombre postgres_container al contenedor de la base de datos.
- ➔ **environment:** Define las variables de entorno para la configuración de PostgreSQL:
  - ◆ **POSTGRES_DB: calculos:** Nombre de la base de datos.
  - ◆ **POSTGRES_USER: postgres:** Nombre de usuario de la base de datos.
  - ◆ **POSTGRES_PASSWORD: postgres:** Contraseña del usuario de la base de datos.
- ➔ **ports: - "5432:5432":** Mapea el puerto 5432 del contenedor al puerto 5432 del host. Esto permite acceder a la base de datos desde el host (si fuera necesario, por ejemplo, con un cliente de PostgreSQL).
- ➔ **volumes: - db_data:/var/lib/postgresql/data:** Monta un volumen llamado db_data en el directorio /var/lib/postgresql/data dentro del contenedor. Esto asegura la persistencia de los datos de la base de datos entre reinicios del contenedor.
- ➔ **backend:** Define el servicio del backend (Flask).
- ➔ **build:** Indica que se debe construir la imagen.
  - ◆ **context: ./backend:** Especifica el directorio que contiene el Dockerfile y los archivos necesarios para construir la imagen (en este caso, el directorio ./backend).
  - ◆ **dockerfile: Dockerfile:** Especifica el nombre del archivo Dockerfile (en este caso, Dockerfile).
- ➔ **container_name: backend_container:** Asigna el nombre backend_container al contenedor del backend.
- ➔ **environment:** Define las variables de entorno para el backend, incluyendo la información de conexión a la base de datos. Es importante notar que DB_HOST=db se refiere al *nombre del servicio* db definido en este mismo archivo docker-compose.yml. Docker Compose se encarga de resolver este nombre a la dirección IP del contenedor de la base de datos.
- ➔ **ports: - "5000:5000":** Mapea el puerto 5000 del contenedor al puerto 5000 del host.

- ➔ **depends_on: - db:** Define una dependencia entre el backend y la base de datos. Esto asegura que el contenedor de la base de datos se inicie *antes* que el contenedor del backend, evitando errores de conexión.
- ➔ **frontend:** Define el servicio del frontend.
- ➔ **build:** Indica que se debe construir la imagen.
    - ◆ **context: ./frontend:** Especifica el contexto de construcción para el frontend.
    - ◆ **dockerfile: Dockerfile:** Especifica el nombre del archivo Dockerfile para el frontend.
- ➔ **container_name: frontend_container:** Asigna el nombre frontend_container al contenedor del frontend.
- ➔ **ports: - "80:80":** Mapea el puerto 80 del contenedor al puerto 80 del host.
- ➔ **depends_on: - backend:** Define una dependencia entre el frontend y el backend. Esto asegura que el backend esté en funcionamiento antes de que se inicie el frontend.
- ➔ **volumes:** db_data:: Define el volumen nombrado db_data que se utiliza para la persistencia de datos de la base de datos.

3.2 Frontend

El frontend se implementó con HTML, CSS y JavaScript. Su función es proporcionar una interfaz de usuario para ingresar los números y mostrar el resultado, se centrará en JavaScript donde se muestra mayor funcionalidad.

```
    <script>
        // Función para enviar la operación y obtener el resultado
document.getElementById('calcular-form').addEventListener('submit', async function (e) {
    e.preventDefault();

    const operacion = document.getElementById('operacion').value;

    try {
        // Realiza la solicitud al backend
        const response = await fetch('/api/calcular', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ operacion: operacion })
        });

        // Verifica si la respuesta es correcta
        if (!response.ok) {
            throw new Error(`Error: ${response.statusText}`);
        }

        // Procesa la respuesta JSON
        const data = await response.json();

        // Actualiza el resultado en la página
        if (data.resultado !== undefined) {
            document.getElementById('resultado').textContent = data.resultado;

            // Recarga el historial para reflejar los nuevos cálculos
            await cargarHistorial();
        } else if (data.error) {
            alert(`Error en el cálculo: ${data.error}`);
        }
    } catch (error) {
        console.error(error);
        alert("No se pudo conectar al servidor.");
    }
});
```

```
        // Función para cargar el historial desde el backend
        async function cargarHistorial() {
            try {
                const response = await fetch('/api/historial');
                if (!response.ok) {
                    throw new Error(`Error al cargar historial: ${response.statusText}`);
                }
                const historial = await response.json();

                const historialContainer = document.getElementById('historial');
                historialContainer.innerHTML = ''; // Limpiar historial

                historial.forEach(item => {
                    const li = document.createElement('li');
                    li.textContent = `${item.operacion} = ${item.resultado} (Fecha: ${item.fecha})`;
                    historialContainer.appendChild(li);
                });
            } catch (error) {
                console.error(error);
                alert("No se pudo cargar el historial");
            }
        }


        // Cargar el historial al cargar la página
        window.onload = cargarHistorial;
```

3.2.1 Explicación del JavaScript:

- **Manejo del envío del formulario:**
  - **document.getElementById('calcular-form').addEventListener('submit', ...):** Añade un listener al evento submit del formulario.
  - **e.preventDefault():** Evita que la página se recargue al enviar el formulario.
  - **fetch('/api/calcular', { ... }):** Realiza una petición POST al endpoint /api/calcular del backend, enviando la operación en formato JSON.
  - **Manejo de la respuesta:**
    - Verifica si la respuesta es exitosa (response.ok).
    - Parsea la respuesta JSON.
    - Muestra el resultado en el elemento con el id resultado.
    - Llama a cargarHistorial() para actualizar el historial.
    - Maneja errores mostrando alertas al usuario.
- **Función cargarHistorial():**
  - **fetch('/api/historial'):** Realiza una petición GET al endpoint /api/historial para obtener el historial.
  - Muestra el mensaje de carga antes de la petición y lo oculta después, independientemente de si la petición fue exitosa o no.
  - Limpia el contenido anterior del historial (historialContainer.innerHTML = '';).
  - Recorre los elementos del historial recibidos del backend y crea elementos <li> para cada uno, añadiéndolos a la lista <ul> con el id historial.
  - Maneja errores mostrando una alerta al usuario.
- **window.onload = cargarHistorial;:** Llama a la función cargarHistorial() cuando se carga la página para mostrar el historial inicial.

3.2.2 Configuración de Nginx (dentro del Frontend)

Este archivo de configuración de Nginx define cómo el servidor web maneja las peticiones entrantes. Se utiliza principalmente para servir los archivos estáticos del frontend y para redirigir las peticiones a la API del backend (Flask).

```
server {
    listen 80;

    server_name localhost;

    # Directorio raíz donde está el archivo HTML
    root /usr/share/nginx/html;


    # Archivo predeterminado para servir
    index index.html;

    # Manejo de archivos estáticos (Frontend)
    location / {
        try_files $uri /index.html;
    }

    # Proxy para el backend (Calcular)
    location /api/calcular {
        proxy_pass http://backend_container:5000/api/calcular;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # Proxy para el backend (Historial)
    location /api/historial/ {
        proxy_pass http://backend_container:5000/api/historial;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }


    # Manejo de errores (opcional)
    error_page 502 /502.html;
    location = /502.html {
        root /usr/share/nginx/html;
        internal;
    }
}
```

- ➔ **server { ... }:** Define un bloque de configuración para un servidor virtual.
- ➔ **listen 80:** Indica a Nginx que escuche en el puerto 80, que es el puerto HTTP por defecto.
- ➔ **server_name localhost:** Define el nombre del servidor. En este caso, localhost significa que el servidor responderá a las peticiones realizadas a localhost o 127.0.0.1.
- ➔ **root /usr/share/nginx/html;:** Define el directorio raíz donde se encuentran los archivos estáticos del frontend (HTML, CSS, JavaScript).
- ➔ **index index.html:** Especifica el archivo que se servirá por defecto cuando se acceda al directorio raíz.
- ➔ **location / { ... }:** Define un bloque de configuración para todas las peticiones a la raíz (/).
- ➔ **try_files $uri /index.html:** Intenta servir el archivo solicitado ($uri). Si el archivo no existe, sirve el archivo **index.html.** Esto es útil para aplicaciones de una sola página (SPA) que utilizan enrutamiento del lado del cliente.
- ➔ **location /api/calcular { ... }:** Define un bloque de configuración para las peticiones a /api/calcular. Este es el bloque que configura el proxy inverso para el backend.
- ➔ **proxy_pass http://backend_container:5000/api/calcular:** Redirige las peticiones a **http://backend_container:5000/api/calcular. backend_container** es el *nombre del servicio* definido en el docker-compose.yml. Docker se encarga de resolver este nombre a la dirección IP del contenedor del backend.
- ➔ **proxy_set_header Host $host:** Pasa el encabezado Host original a la petición al backend.
- ➔ **proxy_set_header X-Real-IP $remote_addr:** Pasa la dirección IP real del cliente al backend.
- ➔ **proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for:** Pasa la cadena de direcciones IP de los proxies que ha atravesado la petición.
- ➔ **location /api/historial/ { ... }:** Define un bloque de configuración similar al anterior, pero para las

peticiones a /api/historial.

- ➔ **error_page 502 /502.html:** Define una página de error personalizada para el código de estado 502 (Bad Gateway).
- ➔ **location = /502.html { ... }:** Define la ubicación de la página de error 502.

4. conclusiones

4.1 Conclusión General:

Se ha desarrollado una aplicación web completa para realizar cálculos matemáticos y mantener un historial de los mismos. La arquitectura implementada se basa en un enfoque de microservicios, donde el frontend, el backend y la base de datos se ejecutan en contenedores separados, orquestados por Docker Compose. Nginx actúa como un proxy inverso, gestionando las peticiones del usuario y redirigiéndolas al backend correspondiente.

4.2 Conclusiones Específicas:

- **Backend (Flask):** El backend se encarga de la lógica de negocio, recibiendo las operaciones del frontend, realizando los cálculos (con la importante **advertencia de seguridad sobre el uso de `eval()`**) y gestionando la interacción con la base de datos. Se utiliza Flask como framework web, lo que facilita la creación de APIs RESTful.
- **Frontend (HTML, CSS, JavaScript):** El frontend proporciona una interfaz de usuario intuitiva para ingresar operaciones y visualizar resultados e historial. Se utilizan tecnologías web estándar como HTML, CSS y JavaScript, y se realizan peticiones asíncronas al backend mediante `fetch`.
- **Nginx:** Nginx actúa como un servidor web y proxy inverso, mejorando el rendimiento y la escalabilidad de la aplicación. Sirve los archivos estáticos del frontend y redirige las peticiones a la API al backend.
- **Docker Compose:** Docker Compose simplifica la gestión de los contenedores, definiendo las dependencias entre ellos y facilitando el despliegue de la aplicación en diferentes entornos.
- **Base de Datos (PostgreSQL):** PostgreSQL almacena el historial de cálculos de forma persistente, permitiendo que la información se conserve entre sesiones. Se utiliza un esquema de tabla simple pero efectivo para este propósito.

4.3 Puntos Fuertes:

- **Arquitectura modular:** La separación en microservicios facilita el mantenimiento, la escalabilidad y la reutilización de componentes.
- **Uso de Docker y Docker Compose:** La contenedorización asegura la portabilidad y la consistencia del entorno de ejecución.
- **Proxy inverso con Nginx:** Mejora el rendimiento y la seguridad al actuar como intermediario entre el usuario y el backend.

- **Persistencia de datos:** El uso de una base de datos permite mantener el historial de cálculos.