ChatGPT

Proyecto AnuSet89 – Diseño e implementación completa

Para cumplir con los requisitos, crearemos tres servicios Docker: un **frontend** en React+Vite, un **backend** en FastAPI, y un **servicio simulado de IA (WebUI Deepseek R1)**. Estos servicios se orquestan con Docker Compose y comparten un volumen `./data/` para persistencia. A continuación se describe la estructura general y se detallan los componentes clave con ejemplos de código comentados en español.

Estructura general del proyecto

El proyecto tiene esta organización de carpetas y archivos principales:

- **frontend/** – Aplicación React creada con Vite. Contiene el formulario (`Ritual.jsx`), hojas de estilo CSS, y configuración de Vite.
- **backend/** – API en FastAPI que recibe los datos del formulario, los valida y los guarda como archivos, además de comunicarse con la IA simulada.
- **webui/** – Servicio simulado de IA (Deepseek R1) que expone el endpoint `/v1/chat/completions` y devuelve respuestas sencillas.
- **data/** – Carpeta compartida (volumen Docker) donde el backend guarda los archivos `.json` y `.txt`.
- **docker-compose.yml** – Define y levanta los tres servicios: frontend (puerto 5173), backend (8000) y webui (3000).
- **assets adicionales** – Los recursos estáticos (imágenes, íconos) del frontend van en `frontend/public/` (por ejemplo, `logo.png`, `favicon.ico`).

```
Anuset89/
├── frontend/
│   ├── package.json
│   ├── vite.config.js
│   ├── public/
│   │   ├── index.html
│   │   ├── logo.png
│   │   └── favicon.ico
│   └── src/
│       ├── main.jsx
│       ├── App.jsx
│       ├── components/
│       │   └── Ritual.jsx
│       └── styles.css
├── backend/
│   ├── main.py
│   ├── requirements.txt
│   └── Dockerfile
├── webui/
```

```
│   ├── fake_ia.py
│   ├── requirements.txt
│   └── Dockerfile
├── data/              # Carpeta compartida por backend y webui
├── docker-compose.yml
└── README.md          # (Opcional, explicación y uso)
```

A continuación detallamos cada componente.

Frontend (React + Vite)

Se utiliza **React** junto con **Vite** para crear un frontend moderno y rápido. Vite provee un *dev server* veloz y soporte para React mediante el plugin oficial `@vitejs/plugin-react` [1]. En `vite.config.js` se activa el plugin de React:

```
// frontend/vite.config.js
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  // Activa el plugin oficial de React para Vite
  plugins: [react()],
})
```

package.json: Define las dependencias necesarias (React, ReactDOM, Vite, y el plugin de React). Por ejemplo:

```
// frontend/package.json
{
  "name": "anuset89-frontend",
  "version": "1.0.0",
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.0.0",
    "vite": "^4.0.0"
  }
}
```

Public assets: En `frontend/public/index.html` va el elemento raíz para la app React. También se colocan aquí imágenes o íconos necesarios, por ejemplo `logo.png`. Un HTML básico:

```
<!-- frontend/public/index.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8" />
  <title>AnuSet89</title>
  <link rel="icon" href="/favicon.ico" />
</head>
<body>
  <div id="root"></div>
  <!-- Vite inyectará los scripts de React aquí -->
</body>
</html>
```

Entrypoint: En `src/main.jsx` se monta la app de React en el DOM. Se importa el componente principal `App` y la hoja de estilos global `styles.css`:

```
// frontend/src/main.jsx
import React from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
import './styles.css'

// Se selecciona el div con id "root" y se renderiza la aplicación React
const container = document.getElementById('root')
createRoot(container).render(<App />)
```

App.jsx: Sirve como envoltorio de la aplicación. Aquí simplemente incluimos el componente del formulario `Ritual` y mostramos la respuesta de la IA cuando llegue:

```
// frontend/src/App.jsx
import React, { useState } from 'react'
import Ritual from './components/Ritual.jsx'

function App() {
  // Estado para almacenar la respuesta que venga del backend
  const [respuestaIA, setRespuestaIA] = useState('')

  // La función Ritual recibe una prop para actualizar la respuesta de la IA
  return (
    <div className="app-container">
      <h1>AnuSet89 - Configuración Ritual</h1>
      <Ritual onRespuesta={setRespuestaIA} />
      {respuestaIA && (
        <div className="response-box">
          <h2>Respuesta de la IA:</h2>
          <p>{respuestaIA}</p>
        </div>
```

```
      )}
    </div>
  )
}

export default App
```

Formulario (Ritual.jsx): Este componente presenta un formulario con campos personalizados (por ejemplo `nombre`, `edad`, etc.). Al enviarse, hace una petición `fetch` al backend local. Un ejemplo de este formulario es:

```
// frontend/src/components/Ritual.jsx
import React, { useState } from 'react'

function Ritual({ onRespuesta }) {
  // Estado local del formulario
  const [formData, setFormData] = useState({
    nombre: '',
    edad: '',
    elemento: '',
    mantra: ''
  })

  // Maneja los cambios en los inputs
  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    })
  }

  // Al enviar el formulario, se envía JSON al backend
  const handleSubmit = async (e) => {
    e.preventDefault()  // Evita recarga de página
    try {
      // Realiza la petición POST al backend
      const response = await fetch('http://localhost:8000/api/ritual', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(formData)  // Envía datos como JSON
      })
      if (!response.ok) throw new Error('Error en la petición')
      const data = await response.json()
      // Actualiza la respuesta recibida (supone campo `respuesta_ia`)
      onRespuesta(data.respuesta_ia)
    } catch (error) {
      console.error('Error al enviar el formulario:', error)
    }
  }
```

```
  return (
    <form className="ritual-form" onSubmit={handleSubmit}>
      {/* Campo Nombre */}
      <label>
        Nombre:
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
          required
        />
      </label>

      {/* Campo Edad */}
      <label>
        Edad:
        <input
          type="number"
          name="edad"
          value={formData.edad}
          onChange={handleChange}
          required
        />
      </label>

      {/* Campo Elemento */}
      <label>
        Elemento Favorito:
        <input
          type="text"
          name="elemento"
          value={formData.elemento}
          onChange={handleChange}
        />
      </label>

      {/* Campo Mantra */}
      <label>
        Mantra:
        <textarea
          name="mantra"
          value={formData.mantra}
          onChange={handleChange}
        />
      </label>

      <button type="submit">Enviar Ritual</button>
    </form>
  )
}
```

```
export default Ritual
```

Notas:
- Se usa la API moderna **Fetch** para enviar datos al servidor de forma asíncrona [2]. Esta API basada en promesas reemplaza a `XMLHttpRequest` y permite configurar encabezados como `Content-Type: application/json`.
- No se muestra ninguna selección de modelo IA en el frontend; la respuesta de la IA es transparente al usuario.
- El componente recibe una función `onRespuesta` (prop) para comunicar la respuesta de la IA (enviado por el backend) hacia el componente padre.

CSS (styles.css): Se añaden estilos básicos para el formulario y contenedores. Por ejemplo:

```
/* frontend/src/styles.css */
body {
  font-family: Arial, sans-serif;
  background-color: #f0f0f0;
  margin: 0;
  padding: 0;
}

.app-container {
  max-width: 600px;
  margin: 2rem auto;
  padding: 1rem;
  background-color: #fff;
  border-radius: 8px;
  text-align: center;
}

.ritual-form {
  display: flex;
  flex-direction: column;
  align-items: stretch;
}

.ritual-form label {
  margin: 0.5rem 0;
  text-align: left;
}

.ritual-form input,
.ritual-form textarea {
  width: 100%;
  padding: 0.5rem;
  margin-top: 0.25rem;
  border: 1px solid #ccc;
  border-radius: 4px;
}
```

```
.ritual-form button {
  margin-top: 1rem;
  padding: 0.75rem;
  background-color: #0078d4;
  color: #fff;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.ritual-form button:hover {
  background-color: #005a9e;
}

.response-box {
  margin-top: 1.5rem;
  padding: 1rem;
  background-color: #e6f7ff;
  border: 1px solid #91d5ff;
  border-radius: 4px;
  text-align: left;
}
```

En resumen, el frontend está compuesto por un formulario React (en `Ritual.jsx`) que envía los datos por `fetch` al backend en `POST /api/ritual`. Todo el código del frontend está documentado con comentarios en español para explicar cada parte.

Backend (FastAPI)

El backend se implementa con **FastAPI**, un framework Python moderno de alto rendimiento diseñado para APIs (utiliza tipeo y Pydantic para validación automática) [3]. En `backend/main.py` definimos la aplicación y los endpoints:

```
# backend/main.py
from fastapi import FastAPI
from pydantic import BaseModel
from datetime import datetime
import json
import requests  # Para llamadas HTTP internas
import os

app = FastAPI()

# Modelo de datos recibido del formulario. Pydantic se encarga de validar.
class RitualData(BaseModel):
    nombre: str
    edad: int
    elemento: str
```

```
    mantra: str

# Endpoint que recibe datos JSON del frontend.
@app.post("/api/ritual")
async def procesar_ritual(data: RitualData):
    """
    Procesa los datos del ritual:
    - Valida automáticamente con Pydantic (FastAPI lo hace al recibir la
petición) [4].
    - Guarda los datos en archivos JSON y TXT.
    - Llama al servicio de IA simulado y retorna su respuesta.
    """
    # Prepara un nombre de archivo único con fecha y hora
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    base_filename = f"{timestamp}_{data.nombre}".replace(" ", "_")
    json_path = os.path.join("data", f"{base_filename}.json")
    txt_path = os.path.join("data", f"{base_filename}.txt")

    # Serializar datos a JSON y guardarlos
    with open(json_path, "w", encoding="utf-8") as f_json:
        json.dump(data.dict(), f_json, ensure_ascii=False, indent=4)

    # Guardar versión de texto plano de los datos
    with open(txt_path, "w", encoding="utf-8") as f_txt:
        f_txt.write(f"Nombre: {data.nombre}\n")
        f_txt.write(f"Edad: {data.edad}\n")
        f_txt.write(f"Elemento: {data.elemento}\n")
        f_txt.write(f"Mantra: {data.mantra}\n")

    # Preparar un prompt básico para la IA usando los datos recibidos
    prompt = (
        f"Datos del ritual:\n"
        f"- Nombre: {data.nombre}\n"
        f"- Edad: {data.edad}\n"
        f"- Elemento: {data.elemento}\n"
        f"- Mantra: {data.mantra}\n"
        f"Por favor, genera una respuesta de bienvenida."
    )

    # Llamada POST al endpoint local de la IA simulada (Deepseek R1)
    # En Docker Compose, se puede usar el nombre de servicio 'webui' para
referirse al contenedor de la IA
    # (véase documentación: cada servicio es alcanzable por su nombre dentro
de la red de Compose [5]).
    try:
        response = requests.post(
            "http://webui:3000/v1/chat/completions",
            json={"prompt": prompt}
        )
        ia_result = response.json()
        respuesta_ia = ia_result.get("message", "")
```

```
    except Exception as e:
        respuesta_ia = "Error al contactar con la IA simulada."
        print("Error en llamada a IA:", e)

    # Devolver la respuesta de la IA al frontend
    return {"respuesta_ia": respuesta_ia}
```

Puntos clave:
- FastAPI interpreta el cuerpo JSON y lo convierte en una instancia de `RitualData`, validando tipos automáticamente [4]. Si falta algún campo o el tipo es incorrecto, responde con un error claro.
- Los datos se guardan en `./data/` como JSON (usando `json.dump`) y como archivo de texto con formato simple. La función `json.dump()` es el método estándar en Python para escribir objetos en archivos JSON [6].
- Para llamar al servicio de IA, utilizamos la librería `requests` para hacer `POST` a `http://webui:3000/v1/chat/completions`. Como se ejecutarán en Docker Compose, la red permite referenciar el servicio por nombre `webui` [5]. Se envía un JSON con el campo `prompt`.
- Finalmente, se extrae la respuesta (`message`) de la IA simulada y se retorna en JSON al frontend.

Dependencias: En `backend/requirements.txt` se incluyen las librerías necesarias:

```
fastapi
uvicorn[standard]
requests
```

El **Dockerfile del backend** utiliza una imagen oficial de Python y arranca con Uvicorn, según prácticas recomendadas [7]:

```
# backend/Dockerfile
FROM python:3.11-slim               # Imagen oficial ligera de Python
WORKDIR /app

# Copia las dependencias y las instala
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copia el código fuente
COPY . .

# Expone el puerto 8000 para FastAPI
EXPOSE 8000

# Ejecuta Uvicorn al iniciar el contenedor
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Esto crea el servicio de backend en el contenedor, escuchando en el puerto 8000. La ruta `/api/ritual` estará documentada automáticamente por FastAPI (con Swagger en `/docs`).

Servicio WebUI Simulado (IA Deepseek R1)

Este servicio imita una API de chat completions (como OpenAI), pero de manera trivial. Expone `POST /v1/chat/completions` y devuelve un mensaje de bienvenida basado en el prompt. Un ejemplo usando **Flask** sería:

```
# webui/fake_ia.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/v1/chat/completions", methods=["POST"])
def completions():
    data = request.get_json()
    prompt = data.get("prompt", "")
    # Extraemos el nombre después de "Nombre: " si está en el prompt
    nombre = "AnuSet89"
    for line in prompt.splitlines():
        if line.lower().startswith("nombre:"):
            nombre = line.split(":", 1)[1].strip()
            break
    mensaje = f"Bienvenida, {nombre} ha sido configurada"
    return jsonify({"message": mensaje})

if __name__ == "__main__":
    # Flask por defecto usa puerto 5000; aquí especificamos 3000
    app.run(host="0.0.0.0", port=3000)
```

Explicación:
- El endpoint recibe un JSON con `prompt`. Para este ejemplo, busca en el prompt la línea con `Nombre: ...` y extrae el nombre del usuario.
- Devuelve un JSON con el campo `message` que contiene la respuesta simulada (ej: `"Bienvenida, {nombre} ha sido configurada"`).
- Se ejecuta en el puerto 3000 (`app.run(host="0.0.0.0", port=3000)`), expuesto a través de Docker.

El **Dockerfile del servicio IA** instala Flask y arranca esta aplicación:

```
# webui/Dockerfile
FROM python:3.11-slim
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

EXPOSE 3000
CMD ["python", "fake_ia.py"]
```

Donde `webui/requirements.txt` contiene `flask`.

Docker Compose y despliegue

El archivo `docker-compose.yml` define los tres servicios y sus puertos, así como el volumen compartido:

```
# docker-compose.yml
version: '3.9'
services:
  frontend:
    build: ./frontend
    container_name: anuset89_frontend
    ports:
      - "5173:5173"
    depends_on:
      - backend

  backend:
    build: ./backend
    container_name: anuset89_backend
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data       # Comparte la carpeta data con el contenedor
    depends_on:
      - webui

  webui:
    build: ./webui
    container_name: anuset89_webui
    ports:
      - "3000:3000"
    volumes:
      - ./data:/app/data       # También comparte la carpeta data (aunque en
este ejemplo sólo el backend escribe ahí)
```

Aspectos clave:
- El servicio **frontend** expone el puerto `5173` (el que Vite usa por defecto) y depende del backend.
- El servicio **backend** escucha en el puerto `8000` (configurado en Uvicorn) y monta el volumen `./data` en `/app/data` (ajustar rutas según Dockerfile).
- El servicio **webui** escucha en el puerto `3000` y también monta `./data` en su `/app/data`.
- Todos los servicios están en la misma red por defecto, por lo que pueden comunicarse usando el nombre de cada servicio (por ejemplo, `http://webui:3000` desde el backend) [5].
- Docker Compose simplifica el despliegue multi-contenedor, permitiendo mapear puertos y volúmenes como se muestra arriba [8].

Con esta configuración, basta ejecutar `docker-compose up --build` en la raíz del proyecto para que se construyan las imágenes y se levanten los tres servicios en localhost (5173, 8000, 3000). El

formulario del frontend en `http://localhost:5173` se comunicará con el backend local en el puerto 8000, y el backend llamará a la IA simulada en el puerto 3000.

Resumen y consideraciones adicionales

Este proyecto sigue las especificaciones solicitadas: - **Todo el código va comentado en español**, facilitando su comprensión.
- No se requiere selección de modelos IA en el frontend; la interacción con la IA ocurre de forma transparente.
- Se incluyen estilos CSS personalizados y puede añadirse cualquier recurso estático en `frontend/public/`.
- El backend utiliza validación Pydantic (FastAPI) y persiste datos en archivos, además de invocar la IA simulado.
- El entorno completo corre en *localhost* gracias a Docker Compose, aislando cada servicio y compartiendo el volumen `data/`.

Por ejemplo, la documentación de FastAPI destaca cómo la mera declaración del modelo en la función maneja la validación de JSON [4]. Asimismo, la documentación de Docker explica que en Compose cada contenedor se registra con el nombre del servicio, lo que permite la comunicación interna sencilla [5]. Además, el uso del plugin oficial de React para Vite está ejemplificado en la guía de Vite, asegurando un entorno React funcional [1].

En conjunto, esta solución cumple con el flujo completo: el usuario llena un formulario en React, el backend guarda esos datos y envía un prompt a la IA simulada, y luego muestra la respuesta de la IA al usuario. Todo ello correrá exclusivamente en local usando los servicios Docker indicados.

Referencias: Se han empleado prácticas comunes documentadas en las guías oficiales de FastAPI, JSON en Python y Docker Compose [4] [6] [8] [5] [1] para asegurar un diseño correcto y robusto.

---

[1] @vitejs/plugin-react - npm
https://www.npmjs.com/package/@vitejs/plugin-react

[2] Using the Fetch API - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[3] [7] [8] Setting up FastAPI with Docker. FastAPI and Docker are a powerful... | by Kwaku Ofosu-Agyeman | Medium
https://medium.com/@kwakuayemang.2000/setting-up-fastapi-with-docker-cd14b3402f10

[4] Request Body - FastAPI
https://fastapi.tiangolo.com/tutorial/body/

[5] Networking | Docker Docs
https://docs.docker.com/compose/how-tos/networking/

[6] Reading and Writing JSON to a File in Python | GeeksforGeeks
https://www.geeksforgeeks.org/reading-and-writing-json-to-a-file-in-python/