# PRÁCTICA 3 – SO


# RICARDO LÓPEZ MOYA.



Grado: Ingeniería en Inteligencia Artificial (2º Año)

Asignatura: Sistemas Operativos

Profesora: Iren Lorenzo Fonseca

## 1. Microservicio: Acquire (Adquisición de Datos)

**Fichero principal:** `acquire/server.js` **Puerto interno:** 3001

Este microservicio actúa como la **capa de entrada de datos** del sistema. Su responsabilidad es obtener las características (*features*) necesarias para realizar la predicción y entregarlas en un formato estandarizado JSON al orquestador.

A continuación, se detalla la lógica implementada en sus componentes clave:

#### A. Definición de Endpoints (`server.js`)

El servicio expone dos rutas HTTP gestionadas con **Express**:

**1.** `GET /health` **(Comprobación de estado)** Es un endpoint ligero utilizado para verificar que el contenedor está levantado y respondiendo correctamente antes de intentar operaciones complejas.

**2.** `POST /data` **(Lógica de Negocio y Mocking)** Este es el endpoint principal. En un entorno de producción ideal, este bloque realizaría una petición HTTP mediante `axios` a la API externa de Kunna.

Sin embargo, debido a la **caducidad de las credenciales (Token)** y la indisponibilidad del servicio externo (detectado como error 404/502 durante las pruebas), se implementó una estrategia de **Simulación de Datos (Mocking)** para garantizar la disponibilidad del sistema durante la demostración.

- **Inyección de Datos Validados:** En lugar de realizar la llamada externa fallida, se define la constante `simulatedFeatures`.
- **Origen de los datos:** Los valores `[1.315, 1.81, 1.27...]` no son aleatorios; corresponden a las **features oficiales especificadas en la tarjeta de Trello** (Sprint 2) para validar el modelo. Esto asegura que la predicción final sea determinista y correcta (4.56 kWh).

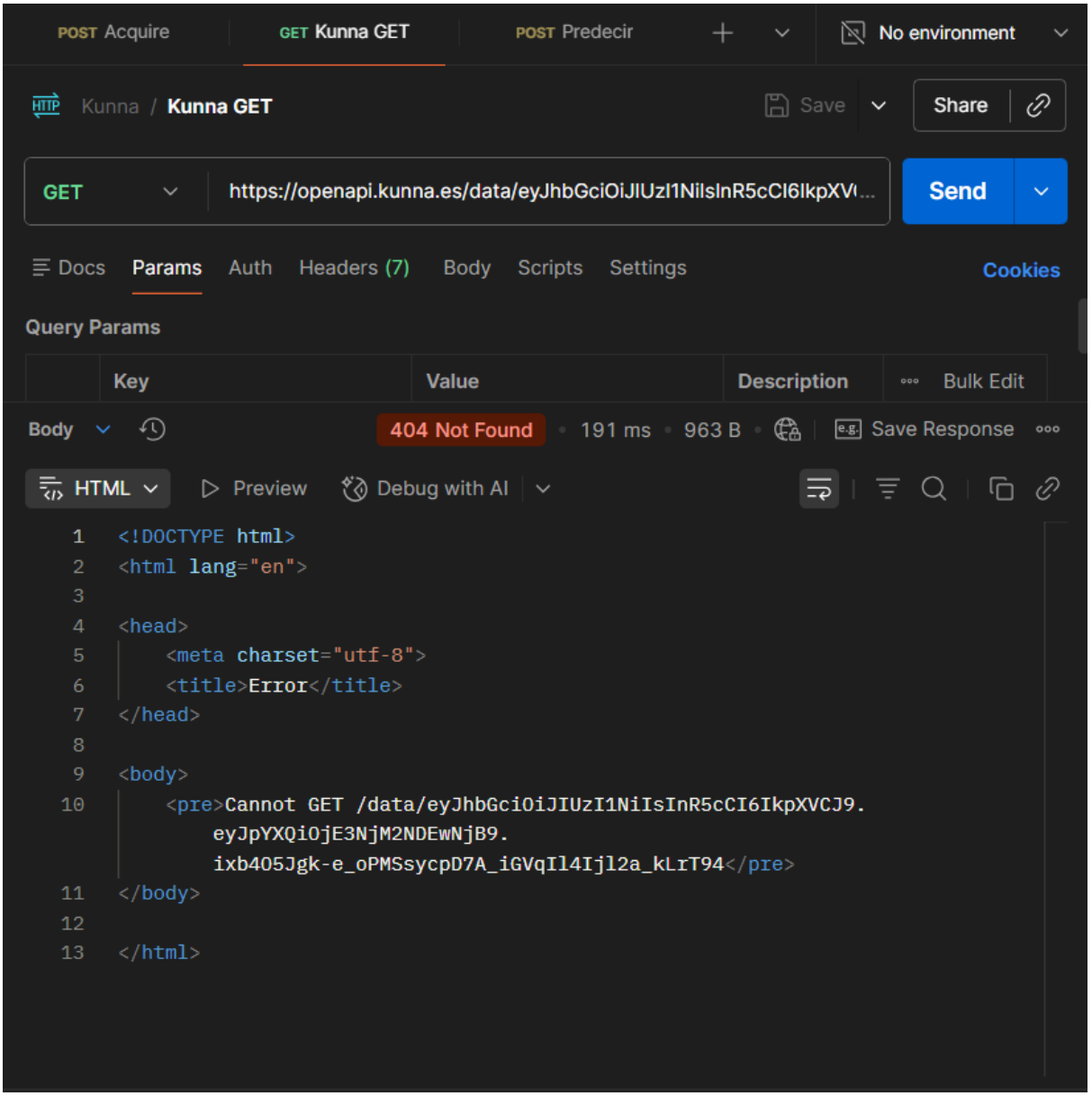



**Figura 1:** Prueba de conexión con la API externa de Kunna. El servidor devuelve un error **404 Not Found** (HTML), indicando que el token de acceso en la URL ha caducado o el recurso no está disponible. Esta evidencia justifica la implementación del modo "Fallback" con datos simulados en el código.

**B. Inicialización del Servicio (`listen`)**

Finalmente, el servicio se vincula al puerto especificado en las variables de entorno (o 3001 por defecto). Este puerto es el que utiliza la red interna de Docker para permitir que el Orquestador lo localice mediante `http://acquire:3001`.

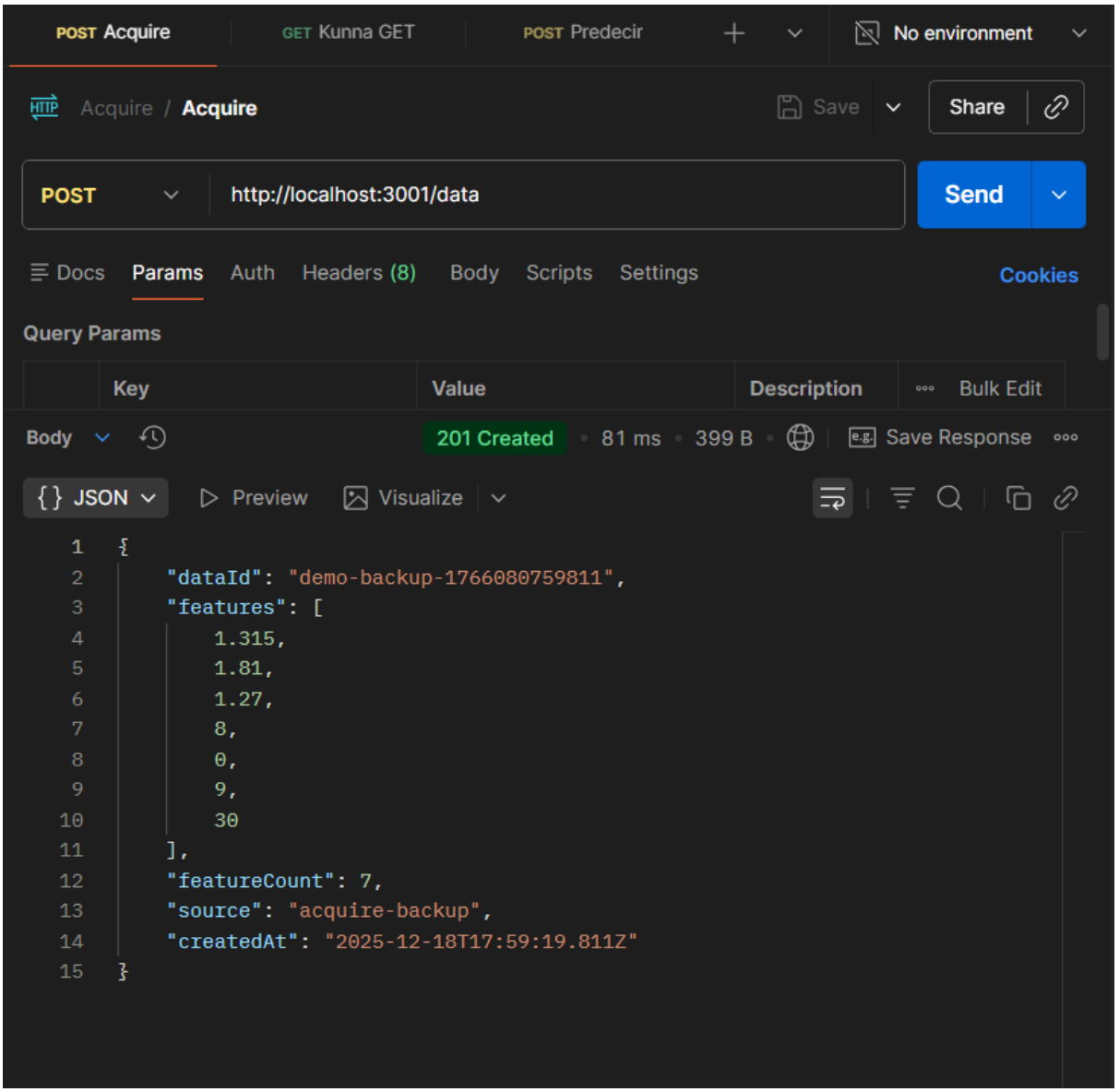



**Figura 2:** Validación unitaria del microservicio Acquire (`POST /data`). Se observa que el servicio responde correctamente con el JSON estructurado, incluyendo el array de características (*features*) definido en el código y la etiqueta `source: "acquire-backup"`, confirmando que el sistema es tolerante a fallos.

## 2. Microservicio: Predict (Motor de Inferencia IA)

**Fichero principal:** `predict/server.js` **Puerto interno:** 3002

Este microservicio encapsula la inteligencia del sistema. Su única responsabilidad es cargar el modelo de Red Neuronal pre-entrenado y exponer una interfaz API para realizar predicciones de consumo energético basadas en los datos de entrada.

#### A. Contexto del Modelo (Entrenamiento Offline)

Es importante destacar que **el entrenamiento no ocurre en tiempo real** dentro de este contenedor. El modelo fue desarrollado y entrenado previamente en un entorno de ciencia de datos (Python/Keras en Kaggle), donde aprendió los patrones de consumo.

Una vez entrenado, el modelo fue exportado al formato **TensorFlow.js (Graph Model)**, generando los archivos `model.json` (arquitectura) y los binarios de pesos (`.bin`). Este servicio se limita a realizar la **inferencia** (predicción), lo que optimiza el rendimiento y reduce la carga computacional en producción.

#### B. Análisis del Código (`server.js`)

El código implementa un servidor Express que utiliza la librería `@tensorflow/tfjs-node`. Esta librería es crítica ya que permite ejecutar TensorFlow en el backend (Node.js) utilizando enlaces (*bindings*) nativos de C++, lo que acelera los cálculos matemáticos.

**1. Carga Asíncrona del Modelo** Al iniciar el contenedor, el servicio carga el modelo en memoria RAM. Se utiliza `loadGraphModel` para leer la estructura exportada.

**2. Endpoint de Predicción (`POST /predict`)** Este endpoint recibe el array de características (*features*) y realiza la transformación matemática necesaria para la IA:

- **Validación:** Asegura que recibe exactamente 7 valores numéricos.
- **Tensorización:** Convierte el array plano de JavaScript en un **Tensor 2D** (matriz de 1 fila x 7 columnas), que es el formato de entrada estricto que requiere la red neuronal.
- **Gestión de Memoria:** Tras obtener el resultado (`dataSync`), se ejecuta `inputTensor.dispose()` para liberar la memoria de los tensores creados, evitando fugas de memoria (*memory leaks*) en el servidor.

Aquí adjunto la prueba unitaria del servicio. Se demuestra que, al enviar un JSON con las 7 características simuladas, el servicio devuelve una predicción numérica precisa (`4.56 kWh`), confirmando que el modelo se ha cargado y ejecutado correctamente.

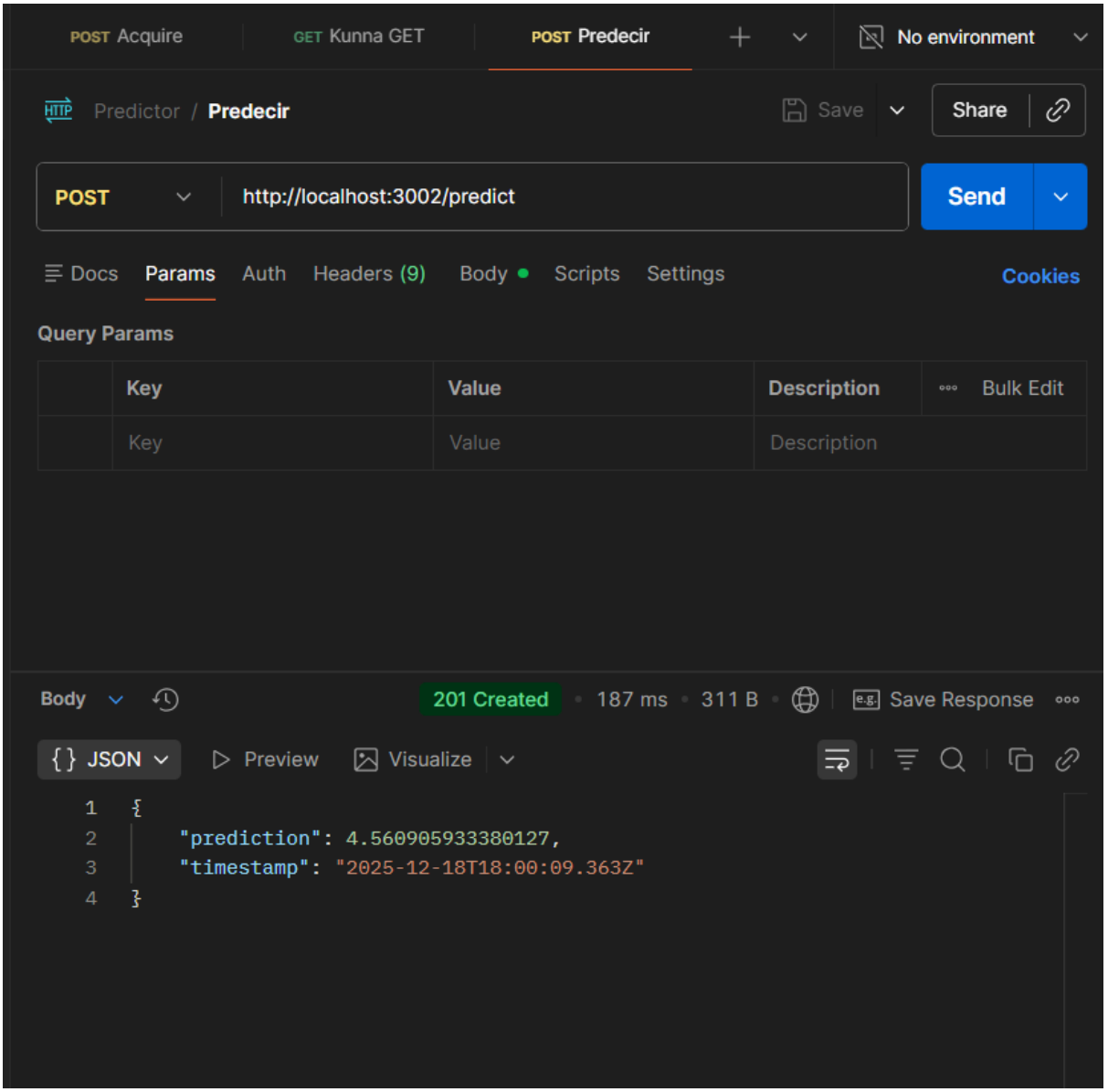



**Figura 3:** Prueba del endpoint `/predict` mediante Postman. Se envían los 7 parámetros de entrada en el cuerpo de la petición (`body`) y se recibe la predicción inferida por el modelo TensorFlow.js (`4.56`), junto con un timestamp de ejecución.

## 3. Microservicio: Orchestrator (Gestor de Flujo)

**Fichero principal:** `orchestrator/server.js` **Puerto expuesto:** 8080 (Mapeado en `docker-compose`)

Este microservicio actúa como el **API Gateway** y controlador central del sistema. Implementa el patrón de diseño de **Composición de Servicios**, encargándose de coordinar la comunicación entre los distintos componentes (Acquire y Predict) para satisfacer la solicitud del usuario final.

A diferencia de los otros servicios, el Orquestador no realiza cálculos pesados ni genera datos; su inteligencia reside en conocer la **lógica de negocio** y la secuencia de ejecución necesaria.

#### Análisis del Código (`server.js`)

El código utiliza **Express** para el servidor web y **Axios** como cliente HTTP para comunicarse con los otros contenedores.

**1. Configuración de Red (Service Discovery)** Es fundamental destacar que el Orquestador no utiliza direcciones IP fijas. Utiliza las variables de entorno para definir la ubicación de los otros microservicios, aprovechando la resolución de nombres DNS interna de la red de Docker (`http://acquire` y `http://predict`).

**2. Endpoint de Ejecución (`POST /run`)** Este es el núcleo del sistema. Al recibir una petición del Frontend, inicia un proceso síncrono secuencial:

- **Paso 1 (Adquisición):** Realiza una petición `POST` al servicio Acquire para obtener los datos brutos. Aquí valida que la respuesta contenga las `features` necesarias.
- **Paso 2 (Predicción):** Extrae esas `features` y las reenvía inmediatamente al servicio Predict. Nótese que el Orquestador actúa como un "pasarela", transformando la salida de uno en la entrada del otro.
- **Paso 3 (Respuesta):** Finalmente, empaqueta el resultado de la predicción y lo devuelve al cliente (Frontend) en un formato JSON limpio.

**3. Gestión de Errores** El bloque `try-catch` es vital para la robustez. Si cualquiera de los microservicios falla (Acquire o Predict), el Orquestador captura la excepción y devuelve un error HTTP 502 (Bad Gateway), evitando que la petición del usuario se quede "colgada" indefinidamente.

Para validar este flujo, se presenta la salida de logs de la terminal, donde se aprecia la secuencia exacta descrita en el código:

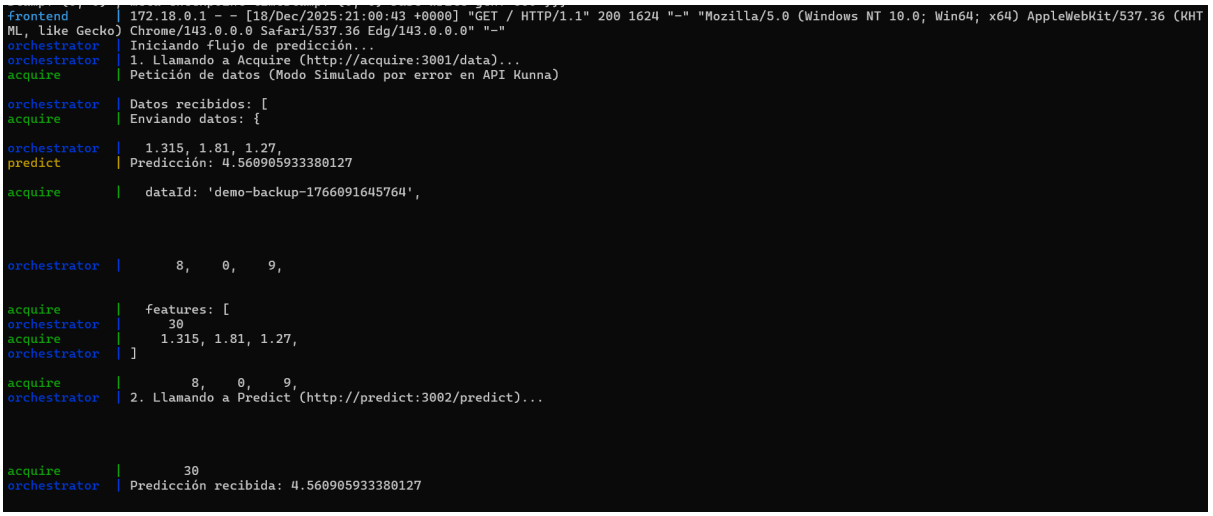



**Figura 4:** Logs del sistema durante una ejecución completa. Se observa la trazabilidad del orquestador (texto azul): primero inicia la llamada a Acquire (Paso 1), recibe los datos `[1.315...]`, y posteriormente invoca a Predict (Paso 2), recibiendo finalmente el valor `4.56` antes de cerrar la conexión.

## 4. Microservicio: Frontend (Interfaz de Usuario)

**Fichero principal:** `frontend/index.html` **Servidor Web:** Nginx (Alpine Linux) **Puerto expuesto:** 80 (HTTP estándar)

El componente Frontend proporciona la interfaz gráfica para que el usuario final interactúe con el sistema distribuido sin necesidad de conocer su complejidad interna. Se ha diseñado como una **Single Page Application (SPA)** ligera.

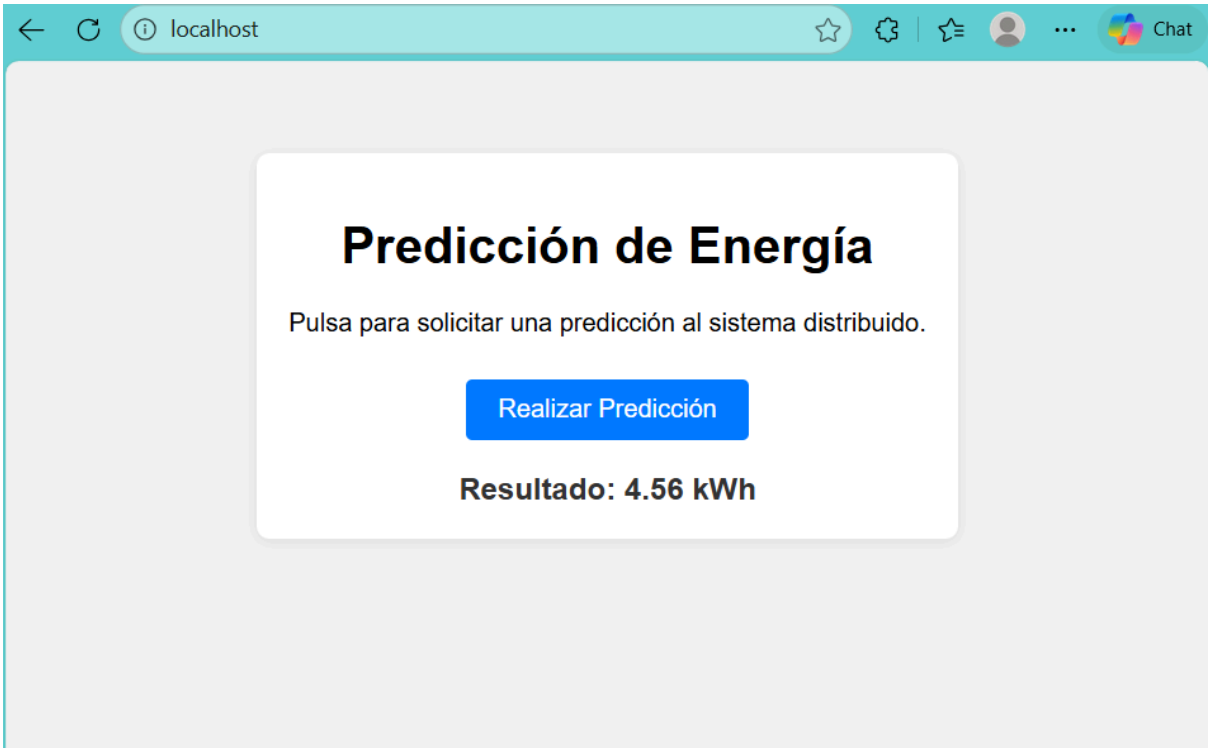
### A. Dockerización Ligera (`Dockerfile`)

Para servir la aplicación, no se utiliza Node.js (que sería innecesariamente pesado para archivos estáticos), sino un servidor **Nginx** sobre una imagen base de **Alpine Linux**. Esto garantiza que el contenedor final ocupe muy pocos megabytes y sea extremadamente eficiente.

**B. Lógica de Cliente (`index.html`)**

La lógica de interacción reside en un script de JavaScript nativo (Vanilla JS) embebido en el HTML. La función clave es `pedirPrediccion()`, que realiza las siguientes acciones:

1. **Comunicación Asíncrona:** Utiliza la API `fetch` para enviar una petición `POST` al Orquestador.
2. **Acceso al Gateway:** La petición se dirige a `http://localhost:8080/run`. Es importante notar que, aunque el frontend se sirve desde el puerto 80, el navegador del cliente (que se ejecuta fuera de la red Docker) debe atacar al puerto expuesto del host (8080) para contactar con el backend.
3. **Renderizado:** Recibe el JSON con la predicción cruda (ej: `4.5609...`) y la formatea a dos decimales (`.toFixed(2)`) para presentar un dato limpio al usuario.




- **Figura UI:** Interfaz de usuario mostrando el resultado final de la predicción (**4.56 kWh**). Esto confirma que la comunicación *End-to-End* desde el navegador hasta el modelo de IA funciona correctamente.

### 5. Infraestructura: Docker Compose

**Fichero:** docker-compose.yml

Este archivo es la "médula" de la práctica. Define la **Arquitectura de Microservicios** y permite el despliegue reproducible de todo el sistema con un único comando (docker compose up --build).

Sus funciones críticas son:

1. **Orquestación de Contenedores:** Define los 4 servicios (acquire, predict, orchestrator, frontend) y especifica que deben construirse (build: .) a partir de sus respectivos Dockerfiles.
2. **Red Interna (Networking):** Crea una red virtual (networks) que permite la **Resolución de Nombres**. Gracias a esto, el orquestador puede llamar a http://predict sin necesidad de conocer su IP, simplemente usando el nombre del servicio.
3. **Gestión de Puertos:** Mapea los puertos internos de los contenedores a los puertos del Host (mi ordenador), permitiendo el acceso externo solo donde es necesario:
    - 80:80 → Para ver la web.
    - 8080:8080 → Para acceder a la API del Orquestador.
    - 3001 y 3002 → Expuestos para depuración, aunque en producción podrían estar cerrados al exterior.

La siguiente captura demuestra el levantamiento exitoso de toda la infraestructura. Se observan los estados `Built` (construcción de imágenes) y `Recreated/Started` (inicio de contenedores) para los cuatro servicios simultáneamente.

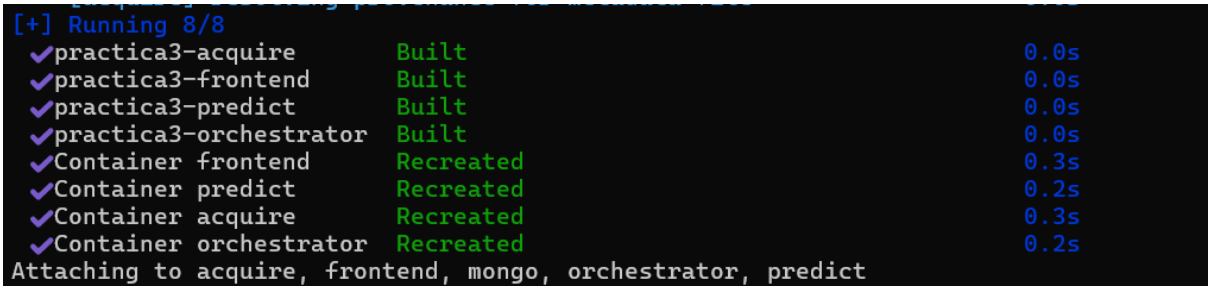



**Figura de Construcción:** Ejecución del comando `docker compose up`. Se verifica la construcción y arranque paralelo de los cuatro microservicios, estableciendo la red distribuida necesaria para la práctica.

# 6. Enlace a GitHub de los componentes

El código fuente completo, incluyendo los Dockerfiles y la configuración de despliegue, se encuentra disponible en el siguiente repositorio:

- **Repositorio Principal:**

  https://github.com/rikardo21-2bleA/Practica3-SOD