UNIVERSIDAD TECNOLOGICA DE TEHUACAN

PROGRAMA EDUCATIVO EN:

Tecnologías de la Información en Área Desarrollo
de Software Multiplataforma

NOMBRE DEL PROFESOR:

JOSÉ MIGUEL CARRERA PACHECO

NOMBRE DEL ESTUDIANTE:

Ciro Julian Cervantes Zamora

MATERIA:

DESARROLLO WEB PROFESIONAL

TEMA:

Propuesta Conceptual de Docker

TEHUACÁN, PUEBLA, ENERO-ABRIL 2026

1. Introducción

Este documento propone la estructura de los contenedores que ejecutarán el proyecto Rapiti. Se basa directamente en la investigación Docker realizada anteriormente, donde ya documentamos cómo funcionan los errores dentro de los contenedores, los logs y las políticas de reinicio.

El punto de partida es un Dockerfile base que ya existe en el proyecto. En esta propuesta lo adaptamos en dos versiones: una para el backend (Express) y otra para el frontend (React), y después los conectamos con un docker-compose.

Recordatorio importante

Esta es una propuesta conceptual. No se ejecuta ni se prueba en esta etapa.

El objetivo es que el equipo pueda revisar la estructura y tomar decisiones antes de implementar.

Todo lo que está aquí se fundamenta en la investigación del paso anterior.

2. Dockerfile base actual del proyecto

Este es el Dockerfile que actualmente existe en el repositorio de Rapiti. Es genérico y funciona para correr cualquier app de Node.js, pero no trata los problemas específicos del backend ni del frontend.

```
# Imagen base
FROM node:18-alpine

# Directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar archivos de dependencias
COPY package*.json ./

# Instalar dependencias
RUN npm install

# Copiar el resto del proyecto
COPY . .

# Exponer el puerto
EXPOSE 3000

# Comando para iniciar la app
CMD ["npm", "start"]
```

3. Dockerfile – Backend (Express)

El backend es donde corre Express y donde los errores 500 pueden hacer caer el contenedor si no están manejados. Este Dockerfile se basa en el original pero con dos ajustes clave.

3.1 Dockerfile propuesto

```
# Imagen base - igual que el original
FROM node:18-alpine

# Directorio de trabajo - igual que el original
WORKDIR /app

# Copiar solo package.json primero
# Esto aproveita el cache de Docker: si las dependencias no cambian,
# Docker no las reinstala en cada build
COPY package*.json ./

# Instalar dependencias - igual que el original
RUN npm install

# Copiar el resto del código
COPY . .

# Puerto del backend - el mismo que usa Express en Rapiti
EXPOSE 3001

# CAMBIO IMPORTANTE: usar 'node' directamente en lugar de 'npm start'
# npm start agrega un proceso extra entre Docker y Node.js.
# Cuando usamos node directamente, Docker puede comunicarse
# correctamente con el proceso y manejar los reintentos si cae.
CMD ["node", "app.js"]
```

3.2 ¿Qué cambió respecto al original y por qué?

Cambio	Explicación
EXPOSE 3001	El backend necesita un puerto diferente al frontend. Usamos 3001 para que no haya conflicto cuando ambos contenedores corren juntos.
CMD ["node", "app.js"]	En lugar de "npm start". Dentro de Docker, es la práctica recomendada usar node directamente. npm agrega un proceso intermedio que puede interferir con las señales de Docker (como cuando necesita reiniciar el contenedor).

¿Y los errores 500?

El Dockerfile del backend por sí solo no resuelve los errores 500.

Lo que resuelve los errores 500 tiene dos partes: el backend debe manejar los errores (responsabilidad de BE), y Docker debe saber cómo reiniciar el contenedor si cae (eso lo configura en el docker-compose con la restart policy).

En esta sección solo definimos cómo se construye el contenedor. La política de reinicio la definimos en la sección 5.

4. Dockerfile – Frontend (React)

El frontend es una aplicación React, que es una SPA (Single Page Application). Esto significa que toda la lógica de las rutas la maneja React en el navegador, no el servidor. El servidor solo necesita servir los archivos estáticos que produce el build de React.

Esto cambia bastante la estructura del Dockerfile respecto al original, por dos razones principales:

- React necesita un paso de build: npm run build. Este paso genera una carpeta de archivos estáticos optimizados (HTML, CSS, JS).
- Para servir esos archivos estáticos en producción no necesitamos Node.js. Usamos nginx, que es un servidor ligero y muy eficiente para esto.

4.1 El problema de las rutas en una SPA

Antes de ver el Dockerfile, es importante entender un problema específico de las SPAs que afecta directamente a cómo configuramos el servidor.

¿Qué pasa cuando un usuario va a /productos/5?

En una SPA como React, la ruta /productos/5 no existe en el servidor. Solo existe en React.

Si el servidor recibe esa petición y no la encuentra, responde con un 404.

Pero no queremos un 404. Queremos que el servidor devuelva el index.html y que React maneje esa ruta.

La solución es configurar nginx para que TODAS las rutas que no sean archivos estáticos (CSS, JS, imágenes) devuelvan el index.html.

Esto se llama 'try_files' en nginx.

4.2 Dockerfile propuesto

Este Dockerfile usa una técnica llamada multi-stage build. Tiene dos etapas: una para hacer el build de React, y otra para servir los archivos. Esto hace que el contenedor final sea mucho más ligero porque no incluye Node.js ni las herramientas de desarrollo.

```
# ─── ETAPA 1: Build ───
# Usamos node para instalar dependencias y hacer el build
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Este comando genera la carpeta /app/dist (o /app/build)
# con todos los archivos estáticos optimizados de React
RUN npm run build

# ─── ETAPA 2: Servir los archivos estáticos ───
# Usamos nginx como servidor. Es ligero y rápido.
FROM nginx:alpine

# Copiamos solo los archivos del build de la etapa 1
# nginx por defecto sirve archivos desde /usr/share/nginx/html
COPY --from=builder /app/dist /usr/share/nginx/html

# Copiamos la configuración de nginx que resuelve el problema
# de las rutas de la SPA (try_files)
COPY nginx.conf /etc/nginx/conf.d/default.conf

# nginx usa el puerto 80 por defecto
EXPOSE 80

# Comando para iniciar nginx
CMD ["nginx", "-g", "daemon off;"]
```

4.3 Archivo nginx.conf (configuración de nginx)

Este archivo le dice a nginx cómo servir los archivos y cómo manejar las rutas de la SPA:

```
server {
    listen 80;

    # Todos los archivos se sirven desde esta carpeta
    location / {
        root /usr/share/nginx/html;
        index index.html;

        # ESTO es lo clave para la SPA:
        # Si la ruta pedida no existe como archivo,
        # devolver index.html para que React la maneje.
        try_files $uri /index.html;
    }
}
```

4.4 ¿Qué cambió respecto al original y por qué?

Cambio	Explicación
Multi-stage build	La etapa 1 usa Node.js para hacer el build. La etapa 2 usa nginx para servir. El contenedor final no tiene Node.js, así que es mucho más ligero.
nginx en lugar de Node.js	Para servir archivos estáticos, nginx es mucho más eficiente que Node.js. No necesitamos un servidor de Node.js en producción para el frontend.
nginx.conf con try_files	Resuelve el problema de las rutas de la SPA. Sin esto, cualquier ruta que no sea un archivo físico devolvería un 404.
EXPOSE 80	nginx usa el puerto 80 por defecto, así que usamos ese en lugar de 3000.

5. docker-compose – Conectando los dos contenedores

El docker-compose es donde definimos cómo van a correr los dos contenedores juntos, cómo se comunican, qué puertos exponen hacia afuera, y las políticas de reinicio que investigamos anteriormente.

5.1 docker-compose propuesto

```
version: '3.8'

services:

  # ─── Frontend (React + nginx) ───
  frontend:
    # Busca el Dockerfile en la carpeta /frontend
    build: ./frontend
    # Puerto externo 3000 → puerto interno 80 (nginx)
    ports:
      - "3000:80"
    # Política de reinicio: unless-stopped
    # Se reiniicia automáticamente si cae,
    # pero no si alguien lo detiene a propósito
    restart: unless-stopped

  # ─── Backend (Express + Node.js) ───
  backend:
    # Busca el Dockerfile en la carpeta /backend
    build: ./backend
    # Puerto externo 3001 → puerto interno 3001 (Express)
    ports:
      - "3001:3001"
    # Política de reinicio: on-failure
    # Se reiniicia solo si cae por un error.
    # Si se detiene exitosamente, no lo reiniicia.
    restart: on-failure
```

5.2 ¿Cómo funcionan los puertos?

Es importante entender la notación de puertos en docker-compose porque en Rapiti los dos servicios necesitan puertos diferentes.

Notación	Significado
"3000:80"	El puerto 3000 de tu máquina se conecta al puerto 80 dentro del contenedor del frontend. Cuando vas a localhost:3000, nginx responde desde el puerto 80 interno.
"3001:3001"	El puerto 3001 de tu máquina se conecta al puerto 3001 dentro del contenedor del backend. Cuando el frontend necesita llamar a la API, lo hace a localhost:3001.

5.3 ¿Por qué cada servicio tiene restart diferente?

Razonamiento de las políticas de reinicio

Frontend (unless-stopped): El servidor nginx casi nunca falla por errores de código. Es muy estable. Usamos unless-stopped porque queremos que siempre esté disponible, pero permitir detenerlo manualmente si es necesario.

Backend (on-failure): El backend es donde puede ocurrir un error 500 no manejado que haga caer el contenedor. Usamos on-failure porque queremos que Docker lo reiniicie automáticamente si cae por un error, pero no en vano si se detiene por otra razón.

Ambos servicios son independientes: si el backend cae, el frontend sigue corriendo. El usuario puede ver la app, aunque las llamadas a la API fallen hasta que el backend se reiniicie.

6. Estructura de carpetas propuesta

Para que docker-compose encuentre los Dockerfiles correctamente, la estructura del proyecto debe verse así:

```
rapiti/
├── docker-compose.yml          ← el archivo que conecta todo
├── frontend/
│   ├── Dockerfile              ← Dockerfile del frontend (React + nginx)
│   ├── nginx.conf              ← configuración de nginx
│   ├── package.json
│   └── src/
│       └── ...                 ← código de React
└── backend/
    ├── Dockerfile              ← Dockerfile del backend (Express)
    ├── package.json
    └── app.js                  ← archivo principal de Express
```

7. Resumen

Esta propuesta transforma el Dockerfile genérico original en dos versiones adaptadas al contexto de Rapiti y las conecta con un docker-compose. Esto es lo que se propone:

1. El Dockerfile del backend se mantiene casi igual al original. El cambio principal es usar node directamente en CMD en lugar de npm start, para que Docker pueda comunicarse correctamente con el proceso.
2. El Dockerfile del frontend cambia más: usa un multi-stage build con nginx para servir los archivos estáticos de React, y configura try_files para que las rutas de la SPA funcionen sin dar 404.
3. El docker-compose conecta los dos contenedores, define los puertos y asigna las políticas de reinicio que ya investigamos: unless-stopped para el frontend y on-failure para el backend.
4. La estructura de carpetas separa frontend y backend en carpetas independientes, con el docker-compose en la raíz.