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:

Docker Investigacion

TEHUACÁN, PUEBLA, ENERO-ABRIL 2026

1. Introducción

Este es el documento integrador de DevOps para el proyecto Rapiti. Su función es unir en un solo lugar todo lo que se investigó y propuso en los tres documentos anteriores, y mostrar cómo Docker soporta la navegación, los errores y la estabilidad del sistema completo.

Este documento se estructura en tres grandes temas, que son exactamente lo que el rol de DevOps debe garantizar:

- Navegación: cómo Docker permite que todas las rutas de Rapiti funcionen correctamente.
- Errores: cómo Docker maneja los errores 404 y 500 sin que el sistema se caiga.
- Estabilidad: cómo Docker mantiene el sistema activo incluso cuando algo falla.

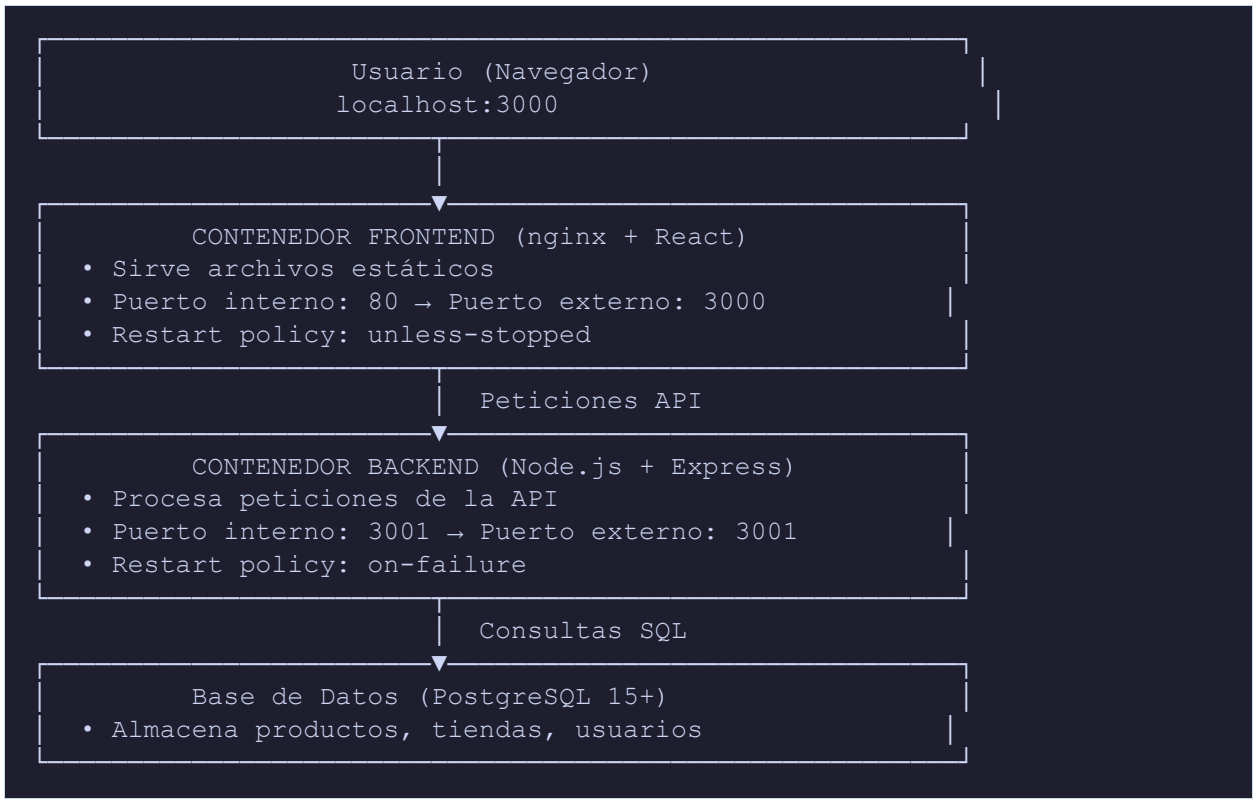
2. Documentos que se integran

Este documento integra los tres entregables previos de DevOps. Cada uno aportó una parte del panorama completo:

Documento	Contenido	Qué aporta al integrador
Doc 1 – Investigación Docker	Cómo funcionan los contenedores, errores, logs y restart policies.	El fundamento teórico. Sin esto, no podemos justificar las decisiones técnicas.
Doc 2 – Propuesta Docker	Dockerfiles del frontend y backend, docker-compose, estructura de carpetas.	La arquitectura concreta. Define cómo va a estar montado el entorno.
Doc 3 – Soporte a Tests	Cómo se ejecutan los tests de rutas dentro del entorno Docker.	La validación. Define cómo vamos a saber si todo funciona.

3. Arquitectura del sistema en Docker

Antes de hablar de navegación, errores y estabilidad, es importante tener claro cómo está montado el sistema. Rapiti tiene tres capas según lo que BE definió, y todas corren dentro de contenedores Docker.



4. Navegación – Cómo Docker la soporta

Rapiti tiene rutas públicas y rutas privadas, según lo que BE y FE definieron. Docker necesita soportar todas ellas sin importar si son públicas o privadas. La clave está en cómo nginx maneja las rutas del frontend y cómo Express maneja las rutas de la API.

4.1 Rutas del frontend (React – SPA)

Todas las rutas del frontend pasan por nginx. Como Rapiti es una SPA, React es quien decide qué mostrar para cada ruta. Nginx solo necesita entregar el index.html en todos los casos.

Ruta	Tipo	Qué hace Docker	Resultado
/	Pública	nginx sirve index.html directamente	React muestra la página principal
/buscar	Pública	nginx sirve index.html (try_files)	React muestra la página de búsqueda
/login	Pública	nginx sirve index.html (try_files)	React muestra el formulario de login
/tienda/panel	Privada (JWT)	nginx sirve index.html (try_files)	React verifica JWT antes de mostrar el panel
/admin/*	Privada (JWT)	nginx sirve index.html (try_files)	React verifica JWT y rol admin

/ruta-no-existe	—	nginx sirve index.html (try_files)	React muestra su página 404 interna

Punto clave de Docker aquí

nginx no conoce ni se preocupa por las rutas de React. Su único trabajo es entregar archivos.

El try_files que configuramos en la propuesta de Docker es lo que hace que ninguna ruta del frontend genere un 404 a nivel de servidor.

La verificación de JWT (autenticación) la hace React en el navegador, no el servidor. Así que Docker no necesita hacer nada especial para las rutas privadas del frontend.

4.2 Rutas de la API (Express – Backend)

Las rutas de la API las maneja Express directamente. Aquí Docker soporta la navegación asegurando que el contenedor backend siga corriendo y respondiendo.

Ruta	Tipo	Qué hace Docker	Resultado
GET /api/productos	Pública	El contenedor backend responde	Express retorna la lista de productos
GET /api/buscar?q=	Pública	El contenedor backend responde	Express busca en PostgreSQL y retorna resultados
GET /api/tiendas	Pública	El contenedor backend responde	Express retorna la lista de tiendas
POST /api/auth/login	Pública	El contenedor backend responde	Express valida credenciales y retorna JWT
GET /api/tienda/productos	Privada	El contenedor backend responde	Express verifica JWT middleware antes de procesar
PUT /api/tienda/precios/:id	Privada	El contenedor backend responde	Express verifica JWT y actualiza el precio
POST /api/admin/*	Privada (admin)	El contenedor backend responde	Express verifica JWT con rol admin
/api/ruta-no-existe	—	El contenedor backend responde con 404	Express retorna error 404 sin caerse

5. Errores – Cómo Docker los maneja

Los dos errores principales del sistema son 404 y 500. Docker no genera estos errores, los genera la aplicación. Pero Docker es responsable de que estos errores no hagan caer el sistema.

5.1 Errores 404 – Recurso no encontrado

Un 404 en Rapiti puede ocurrir en dos lugares: en el frontend o en el backend. En ambos casos, Docker no necesita hacer nada especial porque un 404 es un error "normal" que no amenaza la estabilidad.

Dónde ocurre	Cómo lo maneja Docker
Frontend – ruta inexistente (ej: /producto/xyz)	nginx no genera un 404. Entrega el index.html gracias a try_files. React es quien muestra la página 404 al usuario. El contenedor frontend sigue corriendo sin problema.
Backend – endpoint inexistente (ej: /api/xyz)	Express responde con un 404 a través de su middleware. El proceso de Node.js sigue corriendo. El contenedor backend no se detiene.

5.2 Errores 500 – Error interno del servidor

Un 500 es más serio porque puede hacer caer el contenedor del backend si el error no está manejado correctamente. Aquí Docker juega un papel activo gracias a las políticas de reinicio.

Escenario	Cómo lo maneja Docker
Error síncrono en Express	Express lo captura automáticamente y responde con 500. El contenedor no cae. No necesita intervención de Docker.
Error asíncrono no manejado	Express no lo captura. El proceso de Node.js puede caer. El contenedor se detiene. Docker lo reiniicia automáticamente gracias a la política on-failure.
Error persistente (la app siempre cae)	Docker reiniicia el contenedor, pero sigue cayendo. La política on-failure tiene límite de reintentos para evitar un bucle infinito. Se necesita intervención manual.

¿Qué información tendremos cuando ocurra un error?

Los logs del contenedor (docker logs rapiti-be) van a mostrar exactamente qué error ocurrió y en qué línea.

Si el contenedor se detuvo, docker ps -a va a mostrar el exit code que indica por qué.

Si Docker lo reinició automáticamente, en los logs se verá que el proceso reinició. El usuario puede notar un segundo de interrupción pero la app vuelve a funcionar.

6. Estabilidad – Cómo Docker la garantiza

La estabilidad significa que el sistema sigue funcionando incluso cuando algo falla. Docker garantiza esto a través de tres mecanismos que definimos en los documentos anteriores.

6.1 Aislamiento entre contenedores

Frontend y backend corren en contenedores separados. Si el backend cae por un error 500, el frontend sigue activo y el usuario puede seguir navegando en la app. Solo las peticiones a la API van a fallar hasta que el backend se reiniicie.

Ejemplo concreto en Rapiti

El usuario está en la página principal (/) y el backend cae.

La página que ya cargó sigue visible porque está en el navegador.

Si el usuario intenta buscar un producto, la petición a /api/buscar va a fallar.

Pero la app no se cierra ni da un error catastrófico. React puede mostrar un mensaje amable.

Mientras tanto, Docker reiniicia el backend automáticamente.

En pocos segundos la API vuelve a funcionar y el usuario puede buscar normalmente.

6.2 Políticas de reinicio

Las políticas de reinicio son la principal herramienta de Docker para mantener la estabilidad. En Rapiti las configuramos así:

Contenedor	Política y por qué
Frontend (nginx)	unless-stopped: nginx es muy estable y raramente falla. Esta política asegura que siempre esté disponible, pero permite detenerlo manualmente si es necesario para mantenimiento.
Backend (Express)	on-failure: el backend es donde pueden ocurrir errores 500. Esta política reiniicia el contenedor automáticamente solo cuando cae por un error, sin entrar en un bucle infinito gracias al límite de reintentos.

6.3 Monitoreo con logs

Docker captura automáticamente todos los logs de las aplicaciones. Esto nos permite saber qué pasó sin tener que adivinar. En Rapiti esto significa:

- Si Express registra un error 500, aparece en docker logs rapiti-be.
- Si el contenedor se detiene, docker ps -a muestra el exit code.
- Si Docker reinició el contenedor, los logs muestran el momento en que reinició.
- Si hay un problema con la base de datos (PostgreSQL), Express va a registrar el error de conexión en los logs.

7. Flujo completo de un usuario en Rapiti

Para cerrar el documento, veamos cómo Docker soporta todo el recorrido de un usuario típico en Rapiti, desde que abre la app hasta que busca un producto.

```
1. El usuario abre localhost:3000 en el navegador
   → nginx (contenedor frontend) sirve el index.html
   → React se carga en el navegador

2. React pide los productos a la API
   → Petición GET a localhost:3001/api/productos
   → Express (contenedor backend) procesa la petición
   → Express consulta PostgreSQL
   → Express responde con los datos
   → React muestra los productos en la pantalla

3. El usuario busca un producto específico
   → React hace GET a localhost:3001/api/buscar?q=leche
   → Express busca en la base de datos
   → Express responde con los resultados
   → React los muestra

4. El usuario va a una ruta que no existe (/producto/xyz123)
   → nginx sirve index.html (try_files)
   → React detecta que la ruta no existe
   → React muestra su página 404 interna
   → Ningún contenedor cae

5. El usuario va a /login y inicia sesión
   → React muestra el formulario
   → POST a localhost:3001/api/auth/login
   → Express valida y retorna un JWT
   → React guarda el JWT y redirige al panel

6. El usuario accede a /tienda/panel (ruta privada)
   → nginx sirve index.html (try_files)
   → React verifica que tiene JWT válido
   → React hace GET a localhost:3001/api/tienda/productos
   → Express verifica el JWT en middleware
   → Express retorna los productos de la tienda
```

8. Resumen ejecutivo

Este documento integrador cierra el trabajo de DevOps en esta fase. Las conclusiones principales son:

1. Docker soporta la navegación de Rapiti a través de dos contenedores independientes: nginx para el frontend y Express para el backend. Cada uno maneja sus rutas de forma diferente pero complementaria.
2. Los errores 404 no representan una amenaza para la estabilidad del sistema. nginx los evita en el frontend con try_files, y Express los responde en el backend sin caerse.
3. Los errores 500 son el principal riesgo de estabilidad. Docker los maneja con la política de reinicio on-failure, que reiniicia automáticamente el contenedor backend cuando cae por un error.
4. El aislamiento entre contenedores garantiza que un fallo en el backend no derribe el frontend. El usuario puede seguir navegando mientras el backend se recupera.
5. Los logs de Docker son la herramienta principal de monitoreo. Capturan automáticamente todo lo que ocurre dentro de los contenedores, permitiendo diagnosticar cualquier problema sin necesidad de código adicional.
6. Los tres documentos anteriores (investigación, propuesta, soporte a tests) se ven reflejados en este integrador: la investigación justifica las decisiones, la propuesta las implementa, y los tests las validan.

Estado de los entregables de DevOps

✓ Doc 1 – Investigación Docker: Completado

✓ Doc 2 – Propuesta de Dockerfiles y docker-compose: Completado

✓ Doc 3 – Soporte a Tests: Completado

✓ Doc 4 – Documento Integrador: Completado (este documento)

Todos los entregables de DevOps para esta fase están listos para revisión del Tech Lead.