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:

ERRORES 404 Y 500

TEHUACÁN, PUEBLA, ENERO-ABRIL 2026

1. Introducción

Este documento forma parte de la fase de investigación del proyecto Rapiti, enfocada en entender cómo Docker maneja los errores dentro de los contenedores que ejecutarán el backend (Express.js) y el frontend (React). El objetivo no es escribir código, sino documentar cómo funcionan estos mecanismos para que el equipo pueda tomar decisiones informadas al momento de implementar.

Como DevOps, nuestra responsabilidad es garantizar que el sistema pueda ejecutarse y sobrevivir errores sin caerse. Para eso, necesitamos entender tres cosas fundamentales:

- ¿Qué pasa cuando un proceso dentro de un contenedor lanza un error?
- ¿Cómo capturamos y visualizamos esos errores desde afuera del contenedor?
- ¿Cómo impedimos que un error haga caer el contenedor de forma permanente?

2. Concepto clave: El contenedor y su proceso principal

Antes de hablar de errores, es importante entender cómo funciona un contenedor en lo más básico. Un contenedor Docker no es una máquina virtual. No ejecuta un sistema operativo completo. Solo ejecuta un único proceso principal, que se define en el Dockerfile mediante la instrucción CMD o ENTRYPOINT.

Regla fundamental

Si el proceso principal dentro del contenedor se detiene, el contenedor también se detiene.

No importa si hay otros procesos secundarios corriendo: si el proceso principal cae, todo cae con él.

En el contexto de Rapiti esto se traduce así:

- El contenedor del backend ejecuta Node.js como proceso principal. Node.js ejecuta la aplicación Express.
- El contenedor del frontend ejecuta el servidor estático que sirve la app de React.
- Si Node.js cae en el backend por un error no manejado, el contenedor del backend se detiene completamente.

3. Tipos de errores en Express.js

Express tiene dos categorías principales de errores, y la diferencia entre ellas es crítica para entender qué puede hacer caer un contenedor y qué no.

3.1 Errores síncronos (Express los captura automáticamente)

Cuando un error ocurre en código síncrono dentro de una ruta de Express, el framework lo captura por sí solo y responde al cliente con un error 500. El proceso de Node.js sigue corriendo, el contenedor no cae.

Ejemplo conceptual

Cliente hace una petición a una ruta.

Dentro de esa ruta, algo falla de forma síncrona (por ejemplo, se intenta acceder a un dato que no existe).

Express captura ese error automáticamente.

Express responde al cliente con un código 500.

El proceso sigue corriendo. El contenedor sigue en pie.

3.2 Errores asíncronos (Express NO los captura por defecto)

Este es el escenario peligroso. Cuando un error ocurre dentro de una función asíncrona (async/await o Promises) y no se maneja explícitamente, Express no lo captura. El error se convierte en una "promesa rechazada no manejada" (unhandled promise rejection), lo cual puede hacer caer el proceso de Node.js completo.

Ejemplo conceptual

Cliente hace una petición a una ruta que usa async/await.

Dentro de esa función asíncrona, algo falla (por ejemplo, una consulta a la base de datos falla).

El error NO es capturado por Express automáticamente.

Node.js recibe una promesa rechazada sin manejador.

El proceso de Node.js puede caer.

Si el proceso cae, el contenedor cae.

La solución a esto es usar middleware de manejo de errores en Express, que actúa como una red de seguridad al final de la cadena de middlewares. Esto no es responsabilidad directa de DevOps, sino de BE, pero es algo que DevOps debe entender para evaluar la estabilidad del contenedor.

4. Cómo Express genera errores 404 y 500

En el contexto de Rapiti, los dos errores principales que el sistema va a generar son 404 (recurso no encontrado) y 500 (error interno del servidor). Es importante entender cuándo ocurre cada uno y qué efecto tienen en el contenedor.

Tipo de error	Cuándo ocurre	¿Puede caer el contenedor?
404 – Not Found	Un cliente pide una ruta que no existe en el servidor (ej: /productos/xyz que no existe).	No. Es un error esperado que Express responde automáticamente. El proceso sigue corriendo.
500 – Internal Server Error	Algo falla internamente en el servidor mientras procesa una petición (ej: error en la base de datos, error en código).	Depende. Si el error es síncrono, no cae. Si es asíncrono y no está manejado, sí puede caer.

Punto clave para DevOps

Un error 404 nunca va a hacer caer un contenedor. Es parte del funcionamiento normal.

Un error 500 puede hacer caer el contenedor solo si el error no está correctamente manejado por el backend.

Nuestro trabajo como DevOps es implementar mecanismos (restart policies, logs) para que, incluso si el contenedor cae, el sistema se recupere.

5. Cómo ver los errores desde afuera del contenedor (Logs)

Cuando algo sale mal dentro de un contenedor, necesitamos una forma de saberlo. Docker captura automáticamente todo lo que la aplicación escribe en la salida estándar (stdout y stderr). Esto significa que si Express registra un error en la consola, nosotros podemos verlo desde afuera usando los logs de Docker.

5.1 Comandos principales para ver logs

Comando	Qué hace
docker logs <nombre_contenedor>	Muestra todos los logs del contenedor desde que inició. Funciona incluso si el contenedor ya se detuvo.
docker logs -f <nombre_contenedor>	Muestra los logs en tiempo real (como hacer tail -f en un archivo). Útil para monitorear en vivo.
docker logs --tail 50 <nombre_contenedor>	Muestra solo las últimas 50 líneas de log. Útil cuando hay mucho texto.
docker logs -t <nombre_contenedor>	Muestra los logs con marca de tiempo. Ayuda a saber exactamente cuándo ocurrió el error.

5.2 Cómo saber si un contenedor se detuvo

Comando	Qué hace
docker ps	Muestra solo los contenedores que están actualmente corriendo.
docker ps -a	Muestra todos los contenedores, incluyendo los que ya se detuvieron. En la columna STATUS aparece "Exited" con un código de salida.

5.3 Códigos de salida (Exit Codes)

Cuando un contenedor se detiene, Docker le asigna un código de salida que indica por qué se detuvo. Estos códigos son clave para entender qué pasó.

Código de salida	Significado
0	El proceso terminó exitosamente. No hay error.
1	Error genérico en la aplicación. Algo salió mal en el código.
137	El contenedor fue terminado por el sistema operativo, generalmente por falta de memoria (OOM – Out of Memory).

6. Políticas de reinicio (Restart Policies)

Una de las herramientas más importantes que tiene Docker para mantener la estabilidad es las políticas de reinicio. Estas le dicen a Docker qué hacer automáticamente cuando un contenedor se detiene. En lugar de que un error haga caer el sistema permanentemente, Docker puede reiniciar el contenedor de forma automática.

6.1 Políticas disponibles

Política	Comportamiento
no (por defecto)	Docker NO reinicia el contenedor cuando se detiene. Es la configuración por defecto si no se especifica otra.
always	Docker siempre reiniicia el contenedor, sin importar por qué se detuvo. Includes errores y salidas exitosas.
on-failure	Docker solo reiniicia el contenedor si se detuvo por un error (código de salida diferente a 0). Si se detuvo exitosamente, no lo reiniicia.
unless-stopped	Similar a 'always', pero si alguien detiene el contenedor manualmente (docker stop), no lo reiniicia hasta que el daemon de Docker se reinicie. Es la política recomendada para producción.

¿Cuál usar en Rapiti?

Para el backend (Express): on-failure es la más adecuada. Si el proceso cae por un error, Docker lo reiniicia automáticamente. Pero si se detiene por razón legítima, no lo reiniicia en vano.

Para el frontend (React – servidor estático): unless-stopped es una buena opción, ya que este contenedor raramente falla pero debe estar siempre disponible.

Importante: on-failure permite configurar un máximo de reintentos (ej: on-failure:3). Esto evita que si hay un error persistente, el contenedor no se reinicie en bucle infinito.

6.2 Limitación importante

Una política de reinicio solo funciona si el contenedor arrancó exitosamente al menos una vez (generalmente Docker espera 10 segundos para considerar que el contenedor inició correctamente). Si el contenedor no puede iniciar en absoluto, la política de reinicio no va a solucionar el problema.

7. Buenas prácticas de manejo de errores en Docker

Basándose en la investigación realizada, estas son las buenas prácticas que el equipo de Rapiti debe considerar al momento de implementar los contenedores:

1. Implementar manejo de errores en Express (responsabilidad de BE). Si los errores asíncronos no están manejados, el contenedor puede caer. Esto es lo más importante para la estabilidad.
2. Usar políticas de reinicio apropiadas. No dejar los contenedores con la política por defecto (no). Configurar on-failure o unless-stopped según corresponda.
3. Siempre registrar errores en la consola. Si Express no registra los errores, los logs de Docker no tendrán información útil cuando algo salga mal. El equipo de BE debe asegurarse de que los errores se escriban en stdout/stderr.
4. No envolver Node.js con gestores de procesos extras dentro del contenedor. Herramientas como PM2 no están diseñadas para correr dentro de Docker. Docker mismo maneja los reintentos con las políticas de reinicio.
5. Usar health checks en docker-compose. Un health check es una verificación automática que Docker hace periódicamente para saber si el contenedor sigue funcionando correctamente. Si falla, Docker puede actuar.
6. Separar los contenedores. Si el backend cae, el frontend no debe caer con él. Docker Compose permite que cada servicio corra independientemente.
7. Monitorear los logs regularmente. No esperar a que algo falle en producción para revisar los logs. Durante el desarrollo, revisar los logs es parte normal del flujo.

8. Resumen ejecutivo

Esta investigación cubre los fundamentos necesarios para que el equipo DevOps pueda proponer una arquitectura de contenedores que sea estable frente a errores. Los puntos principales son:

Conclusiones principales

Un contenedor Docker se detiene cuando su proceso principal cae. En Rapiti, esto significa que si Node.js cae, el contenedor del backend cae.

Express captura errores síncronos automáticamente, pero los errores asíncronos necesitan manejo explícito del backend.

Los logs de Docker (docker logs) son la herramienta principal para ver qué pasó dentro de un contenedor cuando algo sale mal.

Las políticas de reinicio (restart policies) permiten que Docker reescriba automáticamente los contenedores cuando caben, evitando tiempo de inactividad.

La política on-failure con límite de reintentos es la más apropiada para el backend de Rapiti.

El aislamiento entre contenedores (backend y frontend separados) es clave para que un error en uno no derribe al otro.