# 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:

Test de Rutas

**TEHUACÁN, PUEBLA, ENERO-ABRIL 2026**

## 1. Introducción

Este documento define cómo se van a ejecutar las pruebas de rutas dentro del entorno Docker que ya propusimos. Su objetivo es que cuando el equipo llegue a la fase de implementación, ya sepa exactamente cómo levantar el entorno, cómo ejecutar los tests y qué hacer si algo falla.

Este documento se coordina con dos áreas del equipo:

- BE (Backend): define qué rutas deben existir, cuáles no deben existir, y cómo simular un error interno.
- QA: define los casos de prueba específicos y qué es un resultado correcto vs incorrecto.

## 2. Punto de partida: el entorno Docker

Para poder ejecutar los tests, primero necesitamos que el entorno Docker esté corriendo. Esto significa que los dos contenedores (frontend y backend) deben estar activos.

**¿Cómo se levanta el entorno?**

Todo se hace desde la carpeta raíz del proyecto, donde está el docker-compose.yml.

El comando que levanta ambos contenedores es: docker-compose up

Si quieres ver los logs en tiempo real mientras corren, usar: docker-compose up -d (esto los levanta en segundo plano y puedes usar docker logs para ver los logs por separado).

Cuando ya estén activos, el frontend eará disponible en localhost:3000 y el backend en localhost:3001.

### 2.1 Verificar que el entorno está listo

Antes de ejecutar cualquier test, necesitamos confirmar que ambos contenedores están corriendo. Esto se hace con un comando simple:

```
# Ver los contenedores que están activos
docker ps

# Lo que debe ver:
# CONTAINER ID   IMAGE      COMMAND          STATUS     PORTS
# abc123         rapiti-fe  nginx -g ...     Up 2 min   0.0.0.0:3000-
>80/tcp
# def456         rapiti-be  node app.js      Up 2 min   0.0.0.0:3001-
>3001/tcp
```

## 3. Tipos de rutas que se prueben

Según lo que BE y QA definen, hay tres tipos de rutas que necesitan probarse. Como DevOps, no definimos los casos de prueba específicos (eso es de QA), pero sí necesitamos entender estos tipos para preparar el entorno correctamente.

| Tipo de ruta | Descripción |
|---|---|
| Rutas válidas | Rutas que existen y deben funcionar correctamente. Por ejemplo, la página principal, la lista de productos, etc. El sistema debe responder normalmente. |
| Rutas inexistentes | Rutas que no existen en el sistema. Por ejemplo, /productos/xyz123 donde ese producto no existe. El sistema debe responder con un 404 sin caerse. |
| Simulación de error interno | Rutas que generan un error 500 a propósito, para probar que el sistema se recupera. Esto requiere que BE tenga una ruta especial de prueba que force un error. |

## 4. Cómo se ejecutan los tests

Los tests de rutas son básicamente peticiones HTTP que se hacen al sistema y se verifican las respuestas. Como el entorno está en Docker, las peticiones se hacen contra localhost con los puertos que definimos en el docker-compose.

### 4.1 Pruebas al frontend (puerto 3000)

Estas pruebas verifican que la navegación del usuario funciona correctamente dentro de la app de React.

```
# Ruta válida - la página principal debe cargar
curl http://localhost:3000/

# Ruta válida - una página interna de la app
curl http://localhost:3000/productos

# Ruta inexistente - NO debe dar 404 en el servidor.
# Debe devolver el index.html para que React la maneje.
# (Esto es gracias al try_files que configuramos en nginx)
curl http://localhost:3000/esta-ruta-no-existe
```

**¿Por qué la ruta inexistente no da 404 en el frontend?**

Como ya documentamos en la propuesta de Docker, el frontend es una SPA.

nginx está configurado con try_files para devolver el index.html en cualquier ruta que no sea un archivo estático.

React es quien decide si esa ruta es válida o no dentro de la aplicación.

Si React no reconoce la ruta, React muestra la página 404 al usuario, pero el servidor nunca respondió con un código HTTP 404.

### 4.2 Pruebas al backend (puerto 3001)

Estas pruebas verifican que la API responde correctamente a las peticiones del frontend.

```
# Ruta válida – un endpoint de la API que existe
curl http://localhost:3001/api/productos

# Ruta inexistente – un endpoint que no existe en Express
# Debe responder con 404 sin caer el contenedor
curl http://localhost:3001/api/esta-ruta-no-existe

# Simulación de error 500 – una ruta que BE prepara a propósito
# para forzar un error interno y probar la estabilidad
curl http://localhost:3001/api/simular-error
```

## 5. Qué pasa si un test falla

"Fallar" en este contexto significa que el sistema no respondió como se esperaba. Hay tres escenarios posibles cuando un test falla, y cada uno tiene un proceso diferente de diagnóstico.

### 5.1 El sistema respondió, pero con el código incorrecto

Por ejemplo, se esperaba un 200 pero se recibió un 404, o se esperaba un 404 pero se recibió un 500. En este caso el contenedor sigue corriendo, solo la respuesta no es la correcta.

**Qué hacer (DevOps)**

El contenedor sigue activo, así que el problema es en la lógica del código.

Ver los logs del backend para entender qué pasó: docker logs rapiti-be

Reportar al equipo (BE o QA según corresponda) con la información de los logs.

No necesita intervención de DevOps para resolver, solo para diagnosticar.

### 5.2 El contenedor se detuvo después del test

Esto puede pasarle al backend si un error 500 no está manejado y hace caer el proceso de Node.js. El test de la simulación de error 500 puede generar exactamente esta situación.

**Qué hacer (DevOps)**

Primero verificar el estado: docker ps -a. El contenedor mostrará "Exited" con un código de salida.

Ver los logs del contenedor detenido: docker logs rapiti-be. Esto funciona incluso si el contenedor ya no está corriendo.

Si la política de reinicio está configurada (on-failure), Docker debería reiniciar el contenedor automáticamente. Verificar si lo hizo.

Si el contenedor no se reinició, significa que el error es persistente y necesita ser corregido por BE.

Reportar al equipo con los logs y el exit code del contenedor.

### 5.3 El entorno no responde en general

En rare occasion puede pasar que ninguno de los contenedores responde. Esto puede ser un problema de recursos o de configuración.

**Qué hacer (DevOps)**

Verificar el estado de todos los contenedores: docker ps -a

Si están detenidos, intentar reiniciarlos: docker-compose up -d

Si no encienden, revisar los logs de cada contenedor por separado.

Si el problema persiste, puede ser necesario hacer un rebuild: docker-compose build y luego docker-compose up -d

## 6. Tabla resumen de escenarios de prueba

Esta tabla une los tres tipos de rutas con lo que debe pasar en cada caso y el papel que juega DevOps. Los colores indican la categoría:

- Verde: pruebas de funcionamiento normal
- Amarillo: pruebas de errores esperados (404)
- Rosa: pruebas de errores internos (500)

| Categoría | Ruta probada | Resultado esperado | ¿Qué hace DevOps? |
|---|---|---|---|
| Ruta válida (FE) | localhost:3000/ | HTTP 200 – la página carga correctamente | Verificar que el contenedor frontend está activo |
| Ruta válida (FE) | localhost:3000/productos | HTTP 200 – la página se carga | Verificar que el contenedor frontend está activo |
| Ruta válida (BE) | localhost:3001/api/productos | HTTP 200 – la API responde con datos | Verificar que el contenedor backend está activo |
| Ruta inexistente (FE) | localhost:3000/no-existe | HTTP 200 – nginx devuelve index.html, React muestra 404 | Confirmar que nginx tiene try_files configurado |
| Ruta inexistente (BE) | localhost:3001/api/no-existe | HTTP 404 – Express responde con error | Verificar logs si el contenedor tiene algún problema |

| | | | |
|---|---|---|---|
| Error interno (BE) | localhost:3001/api/simular-error | HTTP 500 – Express responde con error interno | Verificar que el contenedor no cae. Si cae, revisar logs y restart policy |

## 7. Alineación con el equipo

Este documento no existe en aislamiento. Se conecta con lo que los otros roles del equipo están definiendo en paralelo:

| Rol del equipo | Cómo se conecta con DevOps |
|---|---|
| Backend (BE) | BE define las rutas que existen, las que no existen, y prepara la ruta de simulación de error 500. DevOps necesita esas rutas para saber contra qué URLs hacer las pruebas. |
| QA | QA define los casos de prueba concretos y qué es un resultado correcto vs incorrecto. DevOps define el entorno donde esos tests se ejecutan. |
| Frontend (FE) | FE define la estructura de la app de React y cómo maneja las rutas. DevOps necesita saber eso para configurar nginx correctamente (que ya lo hicimos en la propuesta de Docker). |
| Tech Lead (TL) | TL revisa que todos los documentos estén alineados y que nadie se pise tareas. DevOps reporta cualquier inconsistencia que encuentre entre los documentos del equipo. |

## 8. Resumen

Este documento define la parte de DevOps en la ejecución de tests:

1. El entorno se levanta con docker-compose up desde la carpeta raíz del proyecto. Ambos contenedores deben estar activos antes de ejecutar cualquier prueba.
2. Los tests son peticiones HTTP contra localhost:3000 (frontend) y localhost:3001 (backend), usando los puertos que definimos en el docker-compose.
3. Si un test falla, el proceso de diagnóstico depende de si el contenedor sigue activo o no. Los logs de Docker son la herramienta principal para entender qué pasó.
4. La política de reinicio (on-failure) del backend debe hacerlo reiniciar automáticamente si cae por un error 500, reduciendo el tiempo de inactividad.
5. Este documento se coordina con BE (rutas), QA (casos de prueba) y FE (estructura de la app).