Unidad Académica de Economía

Carrera

Licenciatura en Sistemas Computacionales

Grupo y Semestre

Semestre 6

Nombre estudiante

Ricardo Matos Vizcarra

Actividad

Reto IA 5

Maestro

Eligardo Cruz Sánchez

Materia

Programación Distribuida del lado Cliente

Documento ADR

Contexto

Soy desarrollador frontend junior trabajando en "EcoMarket", un e-commerce para productores locales. Dentro del desarrollo estoy trabajando en un API REST donde la estamos diseñando para EcoMarket y al mismo tiempo estamos viendo la construcción del Cliente HTTP.

Decisión

1. “Use fetch() nativo en lugar de axios”
2. “Maneje errores con try/catch genérico”
3. “Utilice Mock Server en vez de un servidor de navegador (como MSW)”

Alternativas Consideradas

- **Decisión 1:** Alternativa de Axios en vez de fetch(), Axios ya trae hecho (timeouts, protección contra XSRF, transformación automática de JSON).

  La alternativa de tener Axios en vez de fetch es algo que me llama mas la atención ya que tiene mas opciones de manejo de timeouts y errores de HTTP.

- **Decisión 2:** Alternativas de Try/catch:
  - **El patrón “Golang” (Errores como Valores):** Los errores son valores que retornan las funciones. Se puede imitar esto en JavaScript para eliminar el bloque anidado. La ventaja es que el código plano es fácil de leer y te obliga a verificar si hubo error antes de usar la data.
  - **El patrón .catch() Encadenado (Promise Chaining**): Si usas await, a menudo olvidamos que podemos mezclarlo con .catch() para manejar el error en una sola línea y mantener un “valor por defecto” o un estado de fallo sin romper la ejecución. La ventaja es para operaciones de lectura donde un fallo no debe tumbar toda la app, sino mostrar algo vacío o por defecto.
  - **React Query / TanStack (Si usas Frontend):** La mejor alternativa al try/catch manual es no escribirlo tu mismo, las libreiras como TanStack Query o SWR manejan el estado por ti, su ventaja es que desaparece la lógica imperativa. Tú solo dices “Qué ,pstrar si hay error”, no “Como atrapar el error”.

- **Decisión 3:** Alternativa es MSW (Mock Service Worker), se mira la diferencia de arquitectura además de que si es necesario tener que colaborar no va a ser muy útil tenerlo en un lugar local como tu computadora en un entorno como Linux / Ubuntu para alguien que esta en Windows o Mac.

Consecuencias

- **Decisión 1:** Imagina que el servidor de EcoMarket devuelve un error 404 o 500 debido a una caída temporal. fetch() no rechaza la promesa en códigos de estado HTTP de error; solo la rechaza en fallos de red. Tu bloque .then() se ejecuta intentando parsear un JSON que no existe, lanzando un error de sintaxis confuso (Unexpected token < in JSON...) en lugar de un error de API claro. ¿Cómo planeas escalar el manejo de estos "falsos positivos" en 50 endpoints diferentes sin repetir código?

- **Decisión 2:** Un usuario en EcoMarket intenta pagar. Falla. Tu catch genérico captura el error y muestra un mensaje: "Ocurrió un error".
    - ¿Fue porque el usuario no tiene internet?
    - ¿Fue porque su tarjeta fue rechazada (402 Payment Required)?
    - ¿O fue porque tú cometiste un error de tipo (TypeError) al leer la respuesta? Tu analítica de errores recibe 1,000 reportes de "Error genérico" y no tienes idea de si es un bug de código o un servidor caído. El usuario reintenta 10 veces frustrado porque no sabe qué está mal.

- **Decisión 3:** Estás desarrollando una funcionalidad crítica de EcoMarket mientras viajas y no tienes buena conexión, o el servidor de mocks externo se cae / tiene latencia. O peor aún: tus tests E2E en el CI fallan aleatoriamente porque el Mock Server tardó 10ms más de lo esperado en levantar el puerto. Además, tu código de producción hace peticiones a /api/products, pero tu código de desarrollo tiene que apuntar a localhost:3000. Accidentalmente, comiteas la URL de localhost y rompes el entorno de staging.