¿Cuándo conviene y cuándo no conviene usar Redux / NgRx?

Guía razonada para Angular 19 (sin dogmas)

Redux (y NgRx en Angular) no sustituye al servidor ni a la base de datos. Su objetivo es coordinar el estado compartido de la aplicación durante la ejecución y hacer explícitas y predecibles las transiciones de estado.

Idea clave: el servidor es la fuente de verdad persistente; el Store es la fuente de verdad de la UI mientras la aplicación vive.

1. Qué aporta Redux / NgRx

1. Estado compartido entre pantallas y componentes.
2. Coherencia: un único sitio donde se reflejan los cambios confirmados.
3. Predecibilidad: el estado solo cambia mediante acciones explícitas.
4. Trazabilidad: inspección clara de qué pasó y por qué (DevTools).
5. Separación de responsabilidades: UI, dominio y persistencia desacoplados.

2. Cuándo NO conviene usar Redux / NgRx

Si el coste mental y estructural de introducir Store es mayor que el problema que resuelve, no compensa.

1. Estado puramente local de UI (modales, pestañas, inputs).
2. Aplicaciones de una sola pantalla sin estado compartido.
3. Datos que siempre se recargan del servidor y no necesitas conservar.
4. Cuando ya usas una solución de caché/estado servidor más adecuada.
5. Equipos que no pueden mantener disciplina de inmutabilidad.

Checklist rápida de decisión

Si respondes sí a varias, NgRx suele compensar.

Señal	Interpretación práctica
El mismo dato aparece en varias pantallas Ej.: header + lista + detalle Vas a necesitar un estado compartido para evitar duplicar lógica o recargar por inseguridad.	Estado compartido Objetivo Un único sitio que represente la “verdad de la sesión” para esa parte de la UI.
Quieres volver tras crear/editar sin recargar Navegas y quieres continuidad El store puede actuar como memoria y reflejar el cambio confirmado por el servidor al instante.	Memoria de sesión Beneficio Menos llamadas HTTP y menos acoplamiento entre pantallas.
Hay estados intermedios importantes loading / saving / error Necesitas modelar transiciones: qué pasa mientras cargas, guardas o falla algo.	Transiciones explícitas Beneficio Acciones y reducers describen estados intermedios de forma predecible.
Cuesta explicar por qué la UI está así Debug complejo Si hay muchos “if” y estados duplicados, DevTools y acciones te ayudan a rastrear causas.	Trazabilidad Beneficio Puedes ver acción → estado anterior → estado nuevo y reproducir errores.

3. Cuándo SÍ conviene usar Redux / NgRx

Redux encaja cuando el problema no es renderizar datos, sino mantener coherencia entre decisiones y pantallas a lo largo del tiempo.

1. Estado de dominio compartido (carrito, usuario autenticado, permisos).
2. El mismo estado se modifica desde varios lugares.
3. Necesitas continuidad al navegar (sin recargas).
4. Quieres trazabilidad clara de los cambios.

Ejemplo clásico donde compensa: Carrito de compra

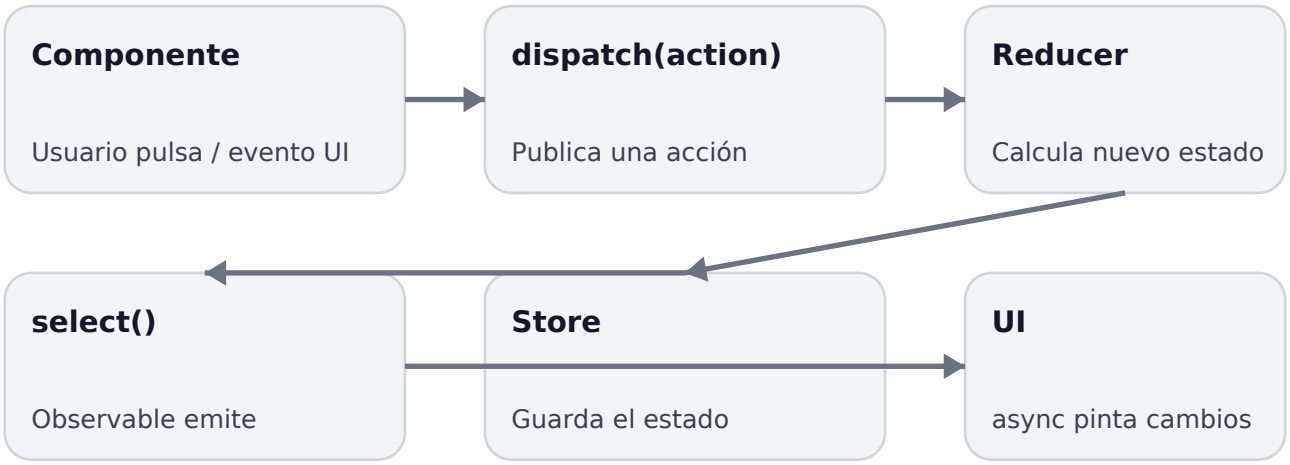
En un e-commerce típico existen múltiples puntos de interacción con el carrito: listado de productos, ficha de producto, mini-carrito en la cabecera, página de carrito y checkout. El carrito debe ser coherente en todos ellos.

1. Añadir/eliminar productos desde distintas pantallas.
2. Actualizar cantidades y totales automáticamente.
3. Mostrar contador y resumen en el header.
4. Mantener el estado al navegar sin recargar.
5. Gestionar estados intermedios (guardando, error de stock, etc.).

Aquí el backend sigue siendo la fuente de verdad del pedido final, pero el Store actúa como la verdad de la sesión: coordina toda la UI y reduce acoplamientos entre componentes.

Diagrama visual: flujo básico en NgRx

Este diagrama resume el recorrido típico: componente → dispatch → reducer → store → selector → UI.



Cierre

NgRx no existe para hacer la UI reactiva (Angular ya lo es), sino para coordinar estado compartido, hacer explícitas las transiciones y reducir la incertidumbre al escalar una aplicación.