

Estado Global con Zustand

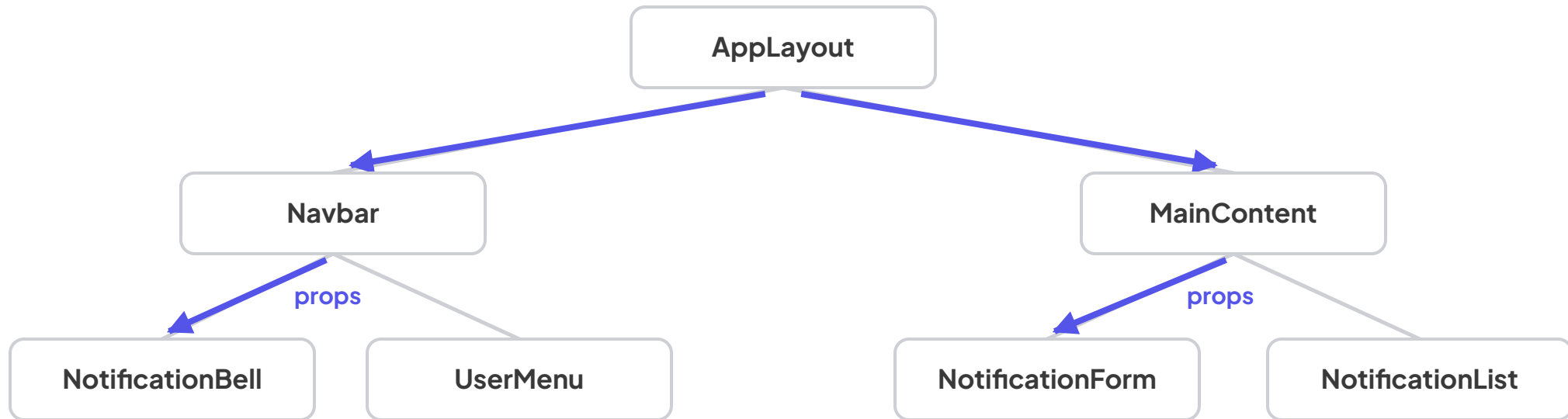
Desarrollo de Entornos Digitales Web · Semana 13

Universidad Icesi

1. Estado compartido

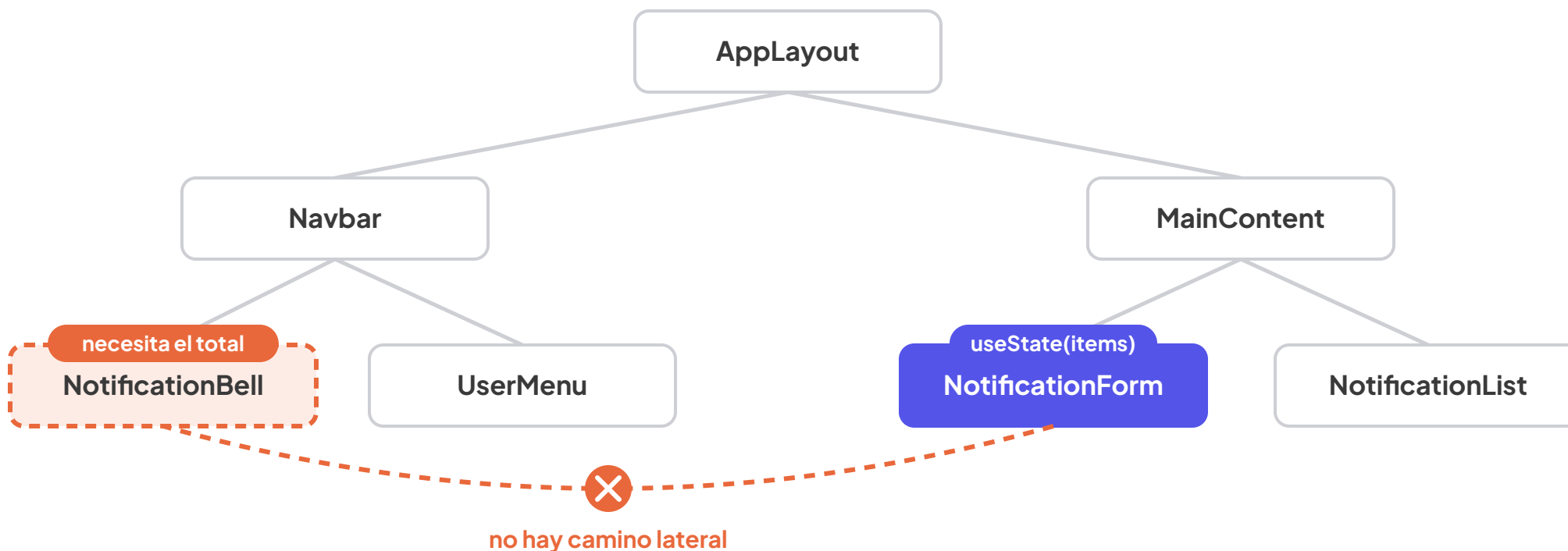
El problema que resuelve un store

Los datos solo bajan por el árbol



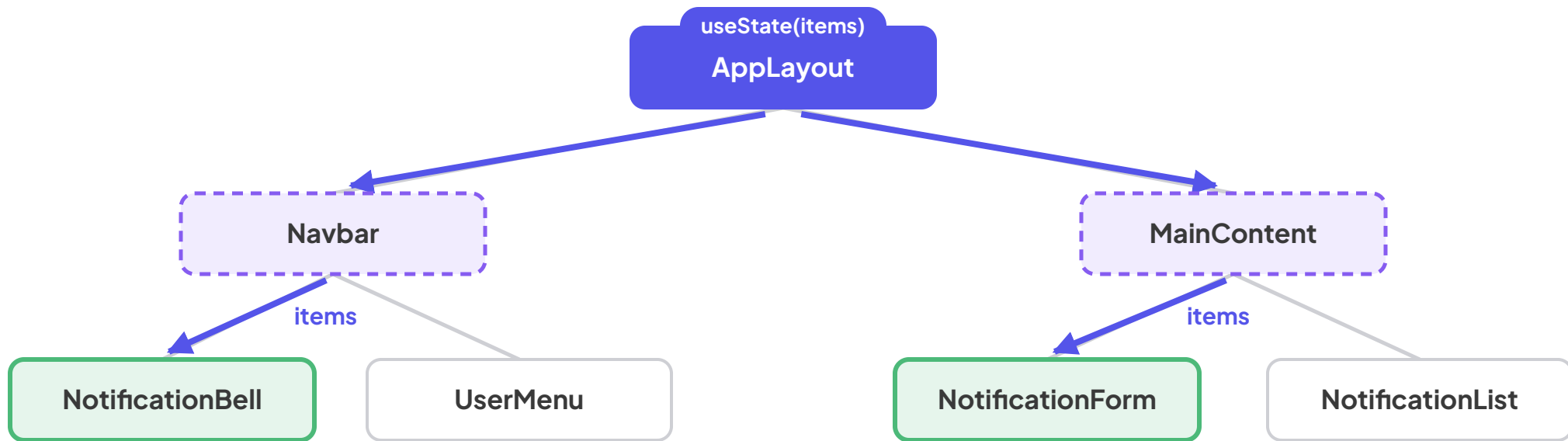
En React los datos viajan de padre a hijo como **props**. No existe un camino directo entre ramas distintas.

El estado vive en NotificationForm



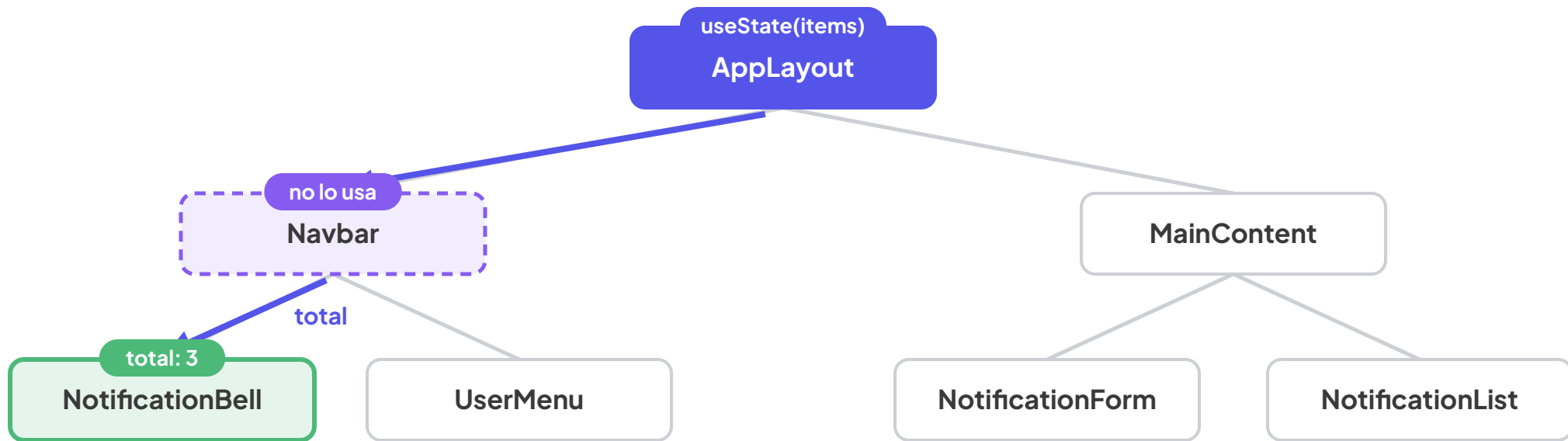
El formulario guarda la lista con `useState`. **NotificationBell** necesita el total, pero está en otra rama: el dato no le llega.

Subir el estado al ancestro común



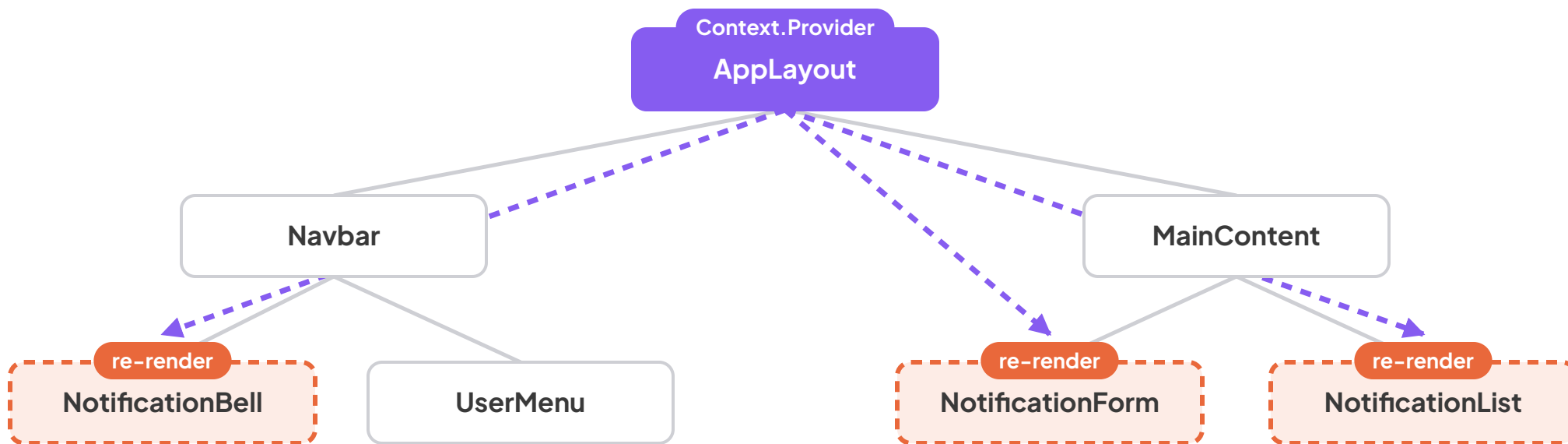
El ancestro común más cercano es **AppLayout**: el `useState` sube hasta ahí y los datos bajan como props.

Prop drilling: props que solo se reenvían



Navbar no usa el dato, solo lo reenvía. Renombrar la prop obliga a tocar todos los intermedios.

Context: un salto directo, con re-renders de más



El Provider evita los intermedios, pero cuando cambia su `value` se renderizan **todos** los consumidores, aunque usen solo una parte.

Tres formas de compartir la sesión

PROPS

```
<Navbar user={user} />
```

Costo: prop drilling; los intermedios quedan acoplados.

CONTEXT

```
<AuthContext.Provider>
```

Costo: todo consumidor se re-renderiza con cualquier cambio.

LOCALSTORAGE DIRECTO

```
session.getToken()
```

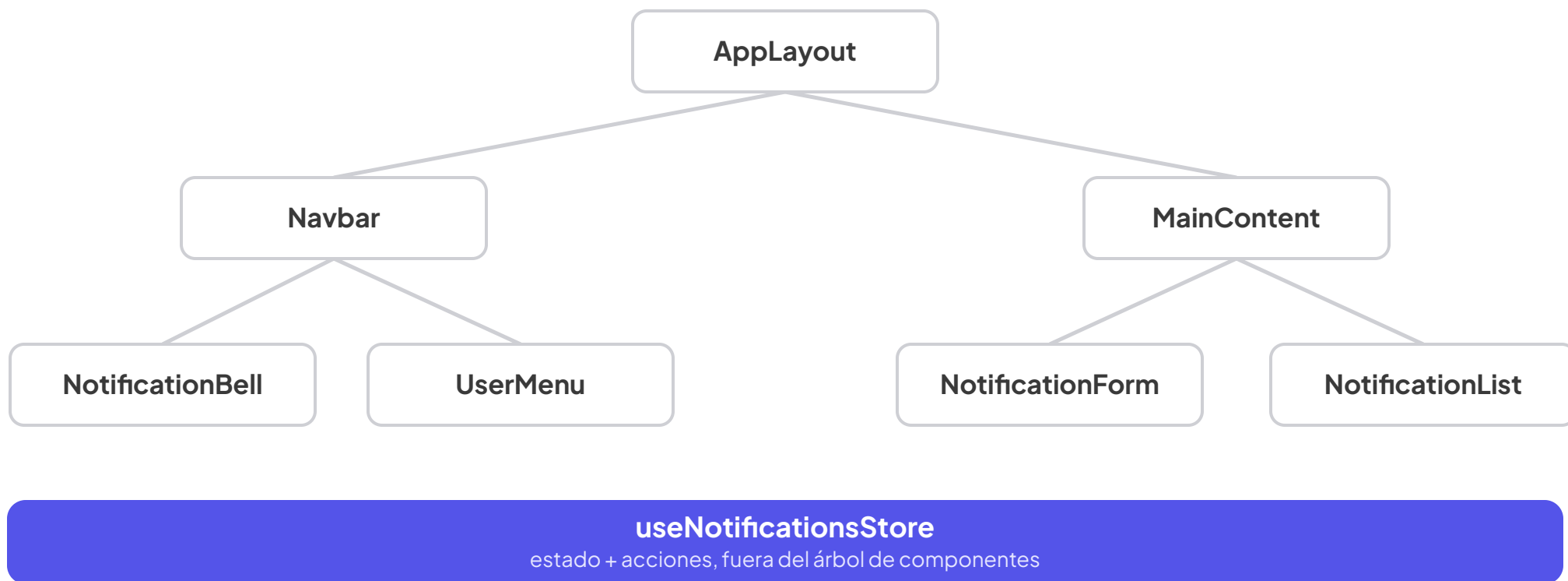
Costo: nadie se entera cuando el valor cambia.



Qué es un store global

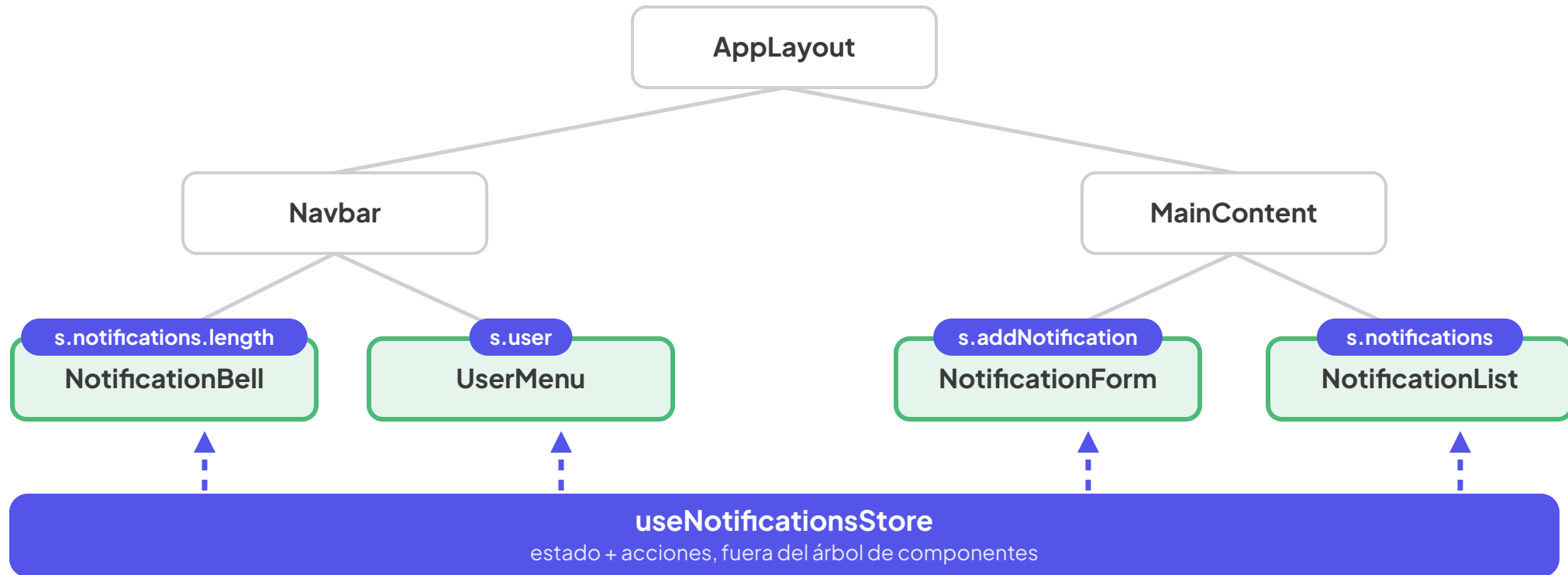
-  Un objeto que vive **fuera** del árbol de componentes
-  Guarda el **estado** compartido y las **acciones** que lo cambian
-  Cada componente se **suscribe** solo a lo que usa
-  Cuando algo cambia, se renderizan **solo los suscritos**

El store vive fuera del árbol



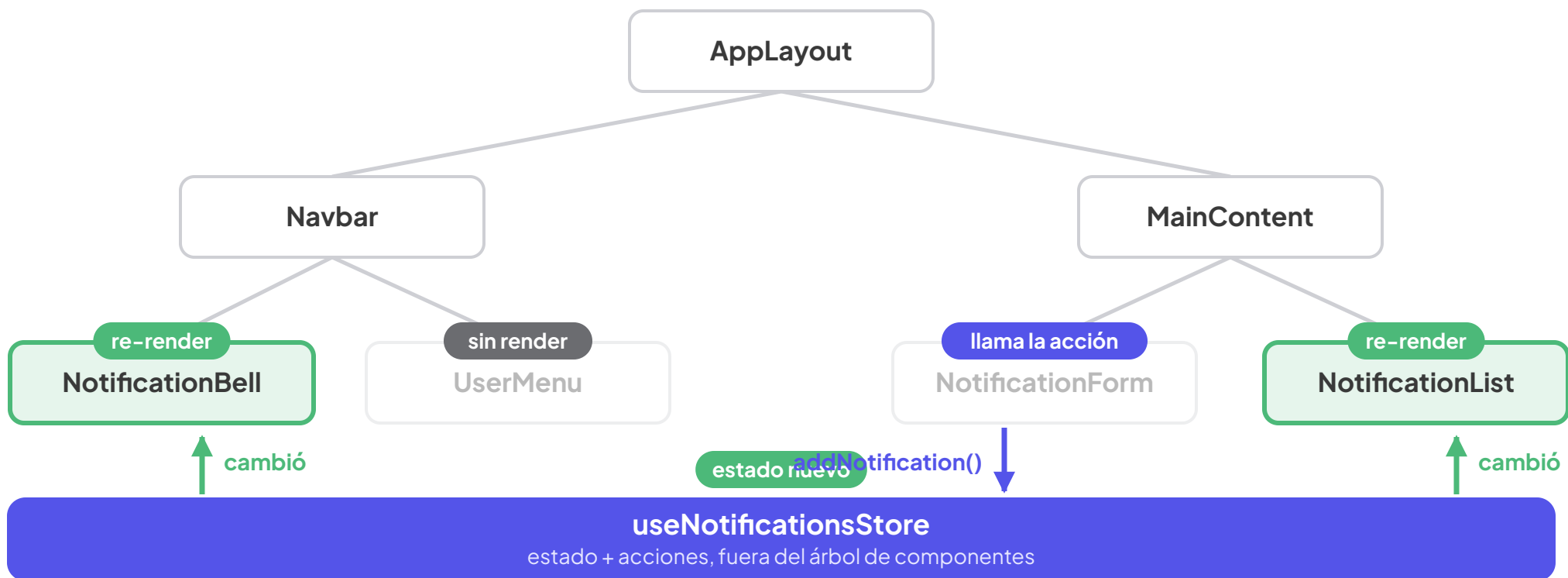
El store no es hijo de nadie: no depende de la jerarquía de componentes, así que **cualquier componente puede llegar a él**.

Cada componente se suscribe con un selector



El **selector** declara de qué dato depende el componente. Sin props y sin Provider.

Una acción cambia el estado y avisa a los interesados



Solo los componentes cuyo dato cambió se vuelven a dibujar: **UserMenu ni se entera**.

¿Cuándo usar un store global?

SÍ CONVIENE

- ✓ Estado que usan **componentes distantes** (sesión, carrito)
- ✓ Datos que sobreviven a que un componente se desmonte
- ✓ Acciones que se llaman desde varias pantallas

NO HACE FALTA

- ✗ El valor de un input mientras se escribe
- ✗ Datos del servidor que se piden en cada pantalla: eso es **server state**
- ✗ Algo que solo usa un componente y sus hijos directos

2. Zustand

Conceptos y flujo de datos

Zustand en cuatro ideas

CREATE

Crea el store y devuelve **un hook** que usan los componentes.

ESTADO + ACCIONES

Viven **en el mismo objeto**. Una acción es solo una función.

SET

Cambia el estado. **Fusiona** lo que devuelves con lo que ya había.

SELECTOR

Cada componente elige **qué dato lee** y solo se renderiza si cambia.

Anatomía de un store: create

notifications.store.ts

```
1 import { create } from 'zustand';
2
3 type State = {
4   notifications: NotificationItem[];
5   addNotification: (title: string) => void;
6   clear: () => void;
7 };
8
9 export const useNotificationsStore = create<State>()(
10   (set) => ({
11     notifications: [],
12     addNotification: (title) =>
13       set((s) => ({
14         notifications: [
15           { id: Date.now(), title, read: false },
16           ...s.notifications,
17         ],
18       })),
19     clear: () => set({ notifications: [] }),
20   }),
21 );
```

-  **create** recibe una función que describe el store
-  Devuelve el hook **useNotificationsStore**
-  Con TypeScript: `create<State>()(...)`, con los dos paréntesis

El hook es lo **único** que importan los componentes.

Anatomía de un store: el estado

notifications.store.ts

```
1 import { create } from 'zustand';
2
3 type State = {
4   notifications: NotificationItem[];
5   addNotification: (title: string) => void;
6   clear: () => void;
7 };
8
9 export const useNotificationsStore = create<State>()(
10   (set) => ({
11     notifications: [],
12     addNotification: (title) =>
13       set((s) => ({
14         notifications: [
15           { id: Date.now(), title, read: false },
16           ...s.notifications,
17         ],
18       })),
19     clear: () => set({ notifications: [] }),
20   }),
21 );
```

-  El **tipo** describe los datos y las acciones
-  El estado inicial se escribe como una propiedad más
-  Los datos se leen con un **selector**, no con el objeto completo

Separa en el tipo los **datos** de las **acciones**.

Anatomía de un store: las acciones

notifications.store.ts

```
1 import { create } from 'zustand';
2
3 type State = {
4   notifications: NotificationItem[];
5   addNotification: (title: string) => void;
6   clear: () => void;
7 };
8
9 export const useNotificationsStore = create<State>(()(
10   (set) => ({
11     notifications: [],
12     addNotification: (title) =>
13       set((s) => ({
14         notifications: [
15           { id: Date.now(), title, read: false },
16           ...s.notifications,
17         ],
18       })),
19     clear: () => set({ notifications: [] }),
20   }),
21 );
```

- ⚙️ Una acción es una **función dentro del store**
- ➔ Se nombra con un verbo: `addNotification`, `clear`
- ✅ No hay `dispatch`, objetos *action* ni reducers

Menos piezas que Redux: **una función** hace todo el trabajo.

set con una función

notifications.store.ts

```
1 import { create } from 'zustand';
2
3 type State = {
4   notifications: NotificationItem[];
5   addNotification: (title: string) => void;
6   clear: () => void;
7 };
8
9 export const useNotificationsStore = create<State>()(
10   (set) => ({
11     notifications: [],
12     addNotification: (title) =>
13       set((s) => ({
14         notifications: [
15           { id: Date.now(), title, read: false },
16           ...s.notifications,
17         ],
18       })),
19     clear: () => set({ notifications: [] }),
20   }),
21 );
```

- 🔄 Úsala cuando el nuevo valor **depende del anterior**
- ☰ Recibe el estado actual: `(s) => ...`
- ✅ Devuelve **solo lo que cambia**: notifications

Siempre un arreglo **nuevo**: nunca `push` sobre el anterior.

set con un objeto

notifications.store.ts

```
1 import { create } from 'zustand';
2
3 type State = {
4   notifications: NotificationItem[];
5   addNotification: (title:string)=> void;
6   clear: ()=> void;
7 };
8
9 export const useNotificationsStore = create<State>(()(
10   (set)=> ({
11     notifications: [],
12     addNotification: (title)=>
13       set((s)=> ({
14         notifications: [
15           { id: Date.now(), title, read: false },
16           ...s.notifications,
17         ],
18       })),
19     clear: ()=> set({ notifications: [] }),
20   }),
21 );
```

- ➔ Úsalo cuando el valor **no depende del anterior**
- ✓ Aquí vaciamos la lista: `{ notifications: [] }`
- ⚡ Las demás propiedades del store **se conservan**

Dos formas de set : con función (depende del estado) o con objeto.

Usar el store en dos componentes que no se conocen

Emitir: solo selecciona la acción

NotificationForm.tsx

```
1 'use client';
2
3 export function NotificationForm() {
4   const addNotification = useNotificationsStore(
5     (s) => s.addNotification,
6   );
7
8   return (
9     <button onClick={() => addNotification('Hola')}>
10       Emitir
11     </button>
12   );
13 }
```

Leer: selecciona un dato derivado

NotificationBell.tsx

```
1 'use client';
2
3 export function NotificationBell() {
4   const unread = useNotificationsStore(
5     (s) => s.notifications
6       .filter((n) => !n.read).length,
7   );
8
9   return <span className="badge">{unread}</span>;
10 }
```

Sin props, sin Provider y sin padre común que guarde el estado.

set fusiona de forma superficial

merge

```
1 //Estado actual
2 { notifications: [], unread: 0 }
3
4 //Devuelves solo lo que cambia
5 set({ unread: 1 });
6
7 //Resultado: lo demás se conserva
8 { notifications: [], unread: 1 }
9
10 //Para reemplazar TODO el estado
11 set({ unread: 1 }, true);
```

- ✓ No necesitas escribir `...state`
- ⚠ La fusión es **superficial**: un objeto anidado se reemplaza completo
- 🔄 El segundo argumento `true` reemplaza todo el estado

El estado es inmutable: nunca mutes

Mal: muta el arreglo

```
addNotification: (title)=>
  set((s)=> {
    s.notifications.push(item);
    return s;
  }),
```

La referencia no cambia: Zustand no detecta el cambio y **nadie se re-renderiza**.

Bien: arreglo nuevo

```
addNotification: (title)=>
  set((s)=> ({
    notifications: [item, ...s.notifications],
  })),
```

Nueva referencia: el selector ve un valor distinto y **avisa a los componentes**.

get: leer el estado dentro de una acción

auth.store.ts

```
1 export const useAuthStore = create<AuthState>()(
2   (set, get) => ({
3     token: null,
4
5     isAuthenticated: () => get().token !== null,
6
7     logout: () => {
8       if (get().token) set({ token: null });
9     },
10  }),
11 );
```

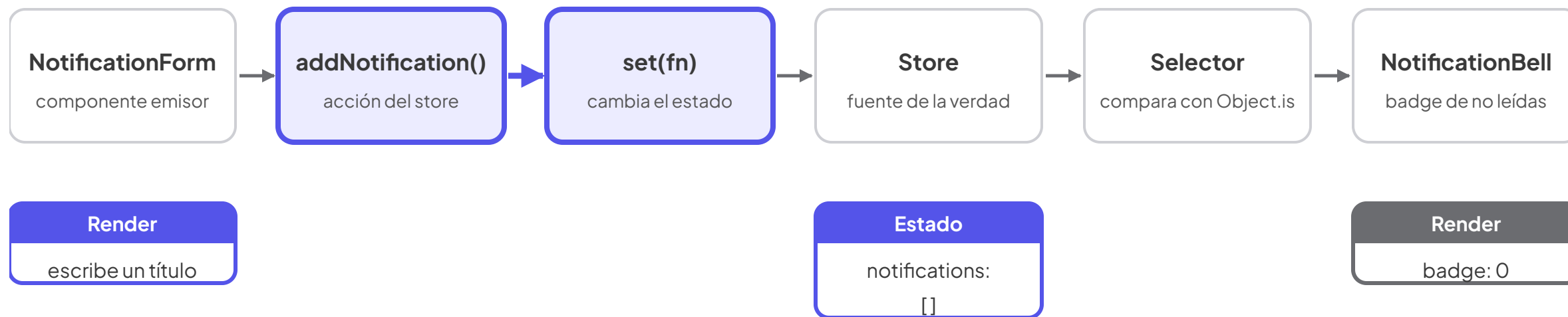
-  `get()` devuelve el estado **actual**
-  Es el segundo parámetro, junto a `set`
-  No suscribe a nadie: solo lee en ese momento

Flujo de un cambio: el componente llama la acción



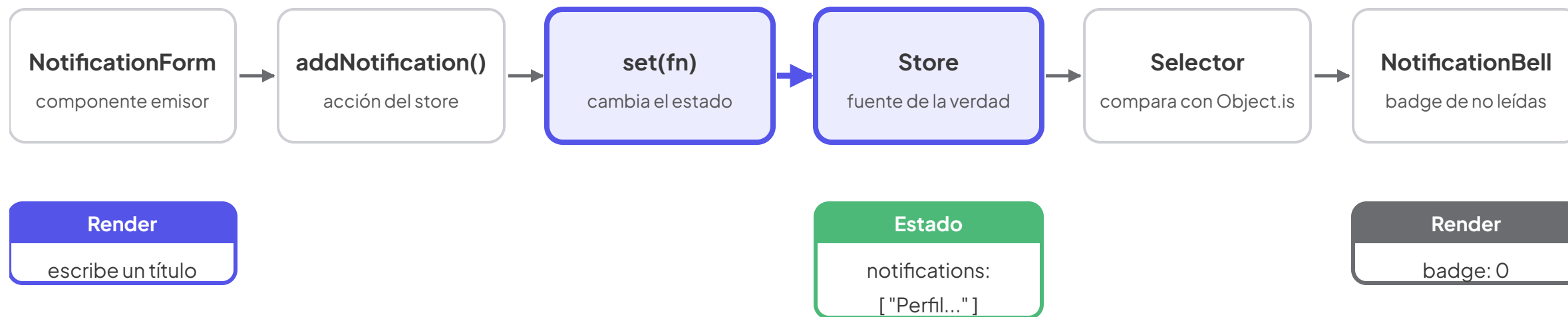
El formulario llama `addNotification("Perfil actualizado")` como cualquier función. No hay `dispatch` ni objeto *action*.

Flujo de un cambio: la acción llama a set()



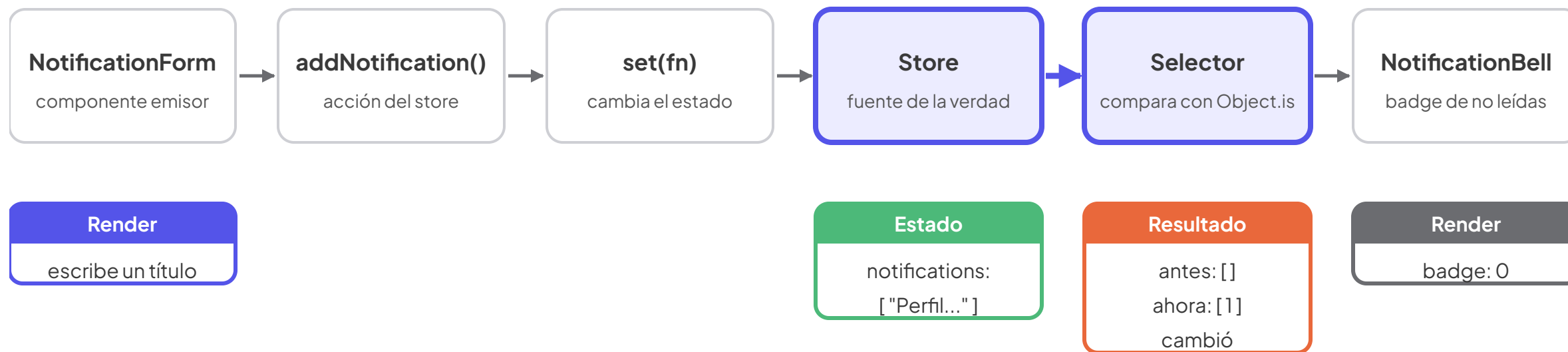
Dentro de la acción, `set` recibe una función con el estado actual y devuelve **solo la parte que cambia**.

Flujo de un cambio: set reemplaza el estado



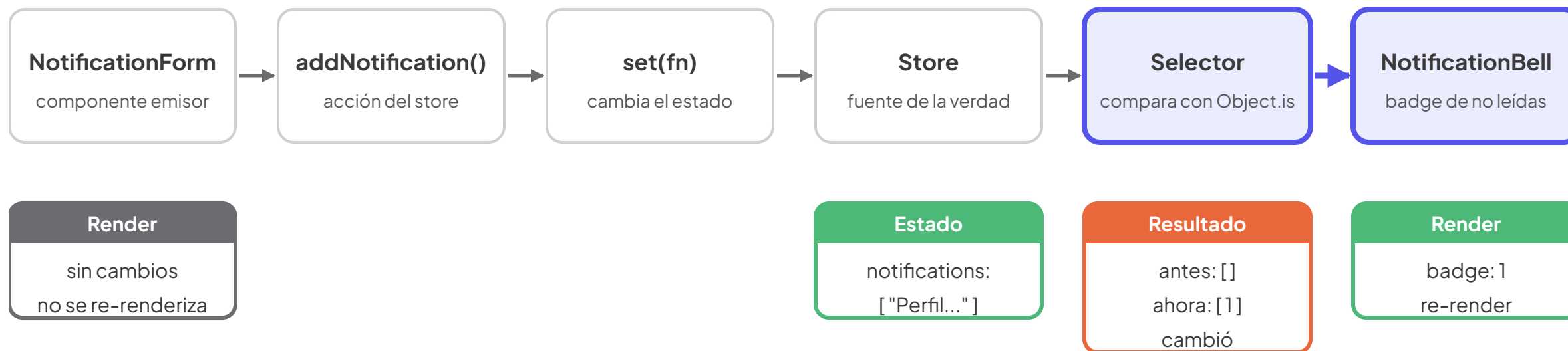
Zustand fusiona el resultado con el estado existente y **cambia la referencia** de `notifications`.

Flujo de un cambio: el store avisa a los suscriptores



Después de cada `set`, el store vuelve a ejecutar el selector de cada componente suscrito y compara con `Object.is`.

Flujo de un cambio: solo se renderiza quien cambió



La cuenta pasó de 0 a 1 y **NotificationBell** se dibuja de nuevo. El formulario solo seleccionó una función, que no cambió, y no se entera.

Por qué el formulario no se re-renderiza

SELECCIONA LA ACCIÓN

```
(s) => s.addNotification
```

Las funciones del store se crean **una sola vez**: la referencia nunca cambia y **no hay render**.

SELECCIONA LA LISTA

```
(s) => s.notifications
```

Cada `addNotification` crea un arreglo nuevo: la referencia cambia y **el componente se vuelve a renderizar**.

Zustand también funciona fuera de React

getState, setState y subscribe

```
1 //Leer el estado actual (sin suscribirse)
2const { token } = useAuthStore.getState();
3
4// Modificarlo desde cualquier archivo
5 useNotificationsStore
6   .getState().addNotification('Guardado');
7 useNotificationsStore.setState({ notifications: [] });
8
9 //Escuchar cambios (devuelve cómo cancelar)
10const unsubscribe = useAuthStore.subscribe(
11   (state) => console.log(state.token),
12 );
```

-  El hook trae `getState`, `setState` y `subscribe`
-  Útil en un **cliente HTTP**, una utilidad o una prueba
-  Ejemplo: `apiFetch` lee el token con `getState()`

Context, Zustand y Redux Toolkit

Criterio	React Context	Zustand	Redux Toolkit
Piezas que aprendes	createContext, Provider, useContext	create, set, selector	store, slice, reducer, action, dispatch
Provider obligatorio	Sí	No (versión simple)	Sí
Re-renders	A todos los consumidores	Por selector	Por selector
Efectos asíncronos	Los resuelves tú	Acción async normal	createAsyncThunk
Depuración	React DevTools	Middleware devtools	Redux DevTools nativo
Convención al equipo	Ninguna	Ninguna	Alta

El mismo cambio en las dos herramientas

Redux Toolkit

```
//slice
reducers: {
  add: (state, action: PayloadAction<string>)=> {
    state.items.unshift(newItem(action.payload));
  },
},

//componente
const dispatch = useAppDispatch();
const items = useAppSelector(
  (s)=> s.notifications.items,
);
dispatch(add('Hola'));
```

Zustand

```
//store
addNotification: (title)=>
  set((s)=> ({
    notifications: [newItem(title), ...s.notifications],
  })),

//componente
const notifications = useNotificationsStore(
  (s)=> s.notifications,
);
const addNotification = useNotificationsStore(
  (s)=> s.addNotification,
);
addNotification('Hola');
```

3. Configuración en Next.js

Instalación, estructura y el cuidado con el servidor



Sin Provider



Sin configuración



TypeScript incluido



React 18 o superior

Instalar Zustand

```
terminal
```

```
npm install zustand
```



Requiere **React 18** o superior: con React 19 y Next.js 16 no hay más que hacer



No se registra un `Provider`, ni un plugin, ni se toca `next.config`



Trae sus propios **tipos de TypeScript**

Dónde viven los stores

estructura por features

```
src/
├── features/
│   ├── notifications/
│   │   ├── components/
│   │   │   ├── NotificationForm.tsx
│   │   │   └── NotificationBell.tsx
│   │   └── store/
│   │       └── notifications.store.ts
└── shared/
    ├── store/
    │   ├── session.store.ts
    │   └── ui.store.ts
```

- Un store pertenece a la **feature** que lo necesita
- Va a `shared/store` si lo usan **varias features** o la capa shared
- Archivo `<nombre>.store.ts` ; hook `use<Nombre>Store`
- Un store por responsabilidad**, no uno por aplicación

Por qué la sesión va en shared



api.ts necesita el token

El cliente HTTP vive en `shared/lib` y lo adjunta a cada petición.



shared no puede depender de una feature

Si el store viviera en `features/auth`, la capa compartida importaría de una feature.



La solución: shared/store/session.store.ts

Es la misma razón por la que `session.ts` estaba en `shared/lib` en la semana 12.

Un store bien tipado: datos y acciones

notifications.store.ts (tipos)

```
1 type NotificationsData = {
2   notifications: NotificationItem[];
3 };
4
5 type NotificationsActions = {
6   addNotification: (title: string) => void;
7   markRead: (id: number) => void;
8   markAllRead: () => void;
9   removeNotification: (id: number) => void;
10};
11
12export type NotificationsState =
13  NotificationsData & NotificationsActions;
14
15 const initialState: NotificationsData = {
16   notifications: [],
17 };
```

-  Separa en el tipo los **datos** de las **acciones**
-  Las acciones se nombran con **verbos**
-  `initialState` se reutiliza para **reiniciar** el store (logout, pruebas)

El store completo

notifications.store.ts

```
1 export const useNotificationsStore =
2   create<NotificationsState>()((set) => ({
3     ...initialState,
4
5     addNotification: (title) =>
6       set((s) => ({
7         notifications: [newItem(title), ...s.notifications],
8       })),
9
10    markRead: (id) =>
11      set((s) => ({
12        notifications: s.notifications.map((n) =>
13          n.id === id ? { ...n, read: true } : n,
14        ),
15      })),
16
17    removeNotification: (id) =>
18      set((s) => ({
19        notifications: s.notifications.filter((n) => n.id !== id),
20      })),
21    }));
```

- ➔ `newItem(title)` crea `{ id, title, read: false }`
- ↺ `map` reemplaza **solo** el elemento que cambia
- 🔍 `filter` quita uno y devuelve un arreglo nuevo
- ✓ Cada acción devuelve un **estado nuevo**, nunca muta

Los hooks solo funcionan en Client Components

Client Component: funciona

NotificationBell.tsx

```
'use client';

export function NotificationBell() {
  const unread = useNotificationsStore(
    (s) => s.notifications.length,
  );
  return <span>{unread}</span>;
}
```

La directiva va en el **componente**, no en el store.

Server Component: falla

page.tsx

```
// app/(app)/dashboard/page.tsx
export default async function Page() {
  const n = useNotificationsStore(
    (s) => s.notifications,
  ); // Error: hooks en el servidor
  return <p>{n.length}</p>;
}
```

El store es estado **del navegador**: un Server Component puede renderizar un Client Component que lo use, pero nunca leerlo.

El servidor comparte los módulos entre peticiones



Solo eventos del usuario (clics, formularios)

Seguro: esas acciones corren en el navegador, donde cada pestaña tiene su propia copia del store.



Meter en el store datos del servidor del request

Inseguro: un usuario podría ver datos del anterior, porque el módulo es el mismo en el proceso.



Escribir en el store durante el render

Inseguro: el render también ocurre en el servidor.

Un store por petición: la fábrica

counter-store.ts

```
1 import { createStore } from 'zustand';
2
3 const createCounterStore = (initialCount: number) =>
4   createStore<CounterState>()((set) => ({
5     count: initialCount,
6     increment: () =>
7       set((s) => ({ count: s.count + 1 })),
8   }));
```

-  `createStore` (de `zustand`) crea un store **sin hook**
-  Cada llamada a la fábrica devuelve una **instancia independiente**
-  Recibe los datos iniciales que llegan del servidor

Es la misma idea que `makeStore` en Redux: una fábrica en vez de una instancia global.

Un store por petición: Provider y hook

counter-provider.tsx

```
1  const CounterContext =
2    createContext<StoreApi<CounterState> | null>(null);
3
4  export function CounterProvider({initialCount, children}) {
5    const storeRef = useRef<StoreApi<CounterState>>(null);
6    if (!storeRef.current) {
7      storeRef.current = createCounterStore(initialCount);
8    }
9    return (
10     <CounterContext.Provider value={storeRef.current}>
11       {children}
12     </CounterContext.Provider>
13   );
14 }
15
16 export function useCounterStore<T>(sel: (s: CounterState) => T) {
17   const store = useContext(CounterContext)!;
18   return useStore(store, sel);
19 }
```

-  El `Provider` crea el store **una vez** con `useRef`
-  El hook lee el store del contexto y usa `useStore`
-  El Context solo **transporta** la referencia: la reactividad sigue siendo por selector



Stores del navegador



Sesión y tema



Notificaciones



create simple basta

Decisión práctica



Empieza con la versión simple: `create` a nivel de módulo



Pasa al patrón por petición solo si necesitas **hidratar el store con datos del servidor**



No lo uses "por si acaso": agrega un Provider y líneas que no necesitas todavía

4. Capacidades

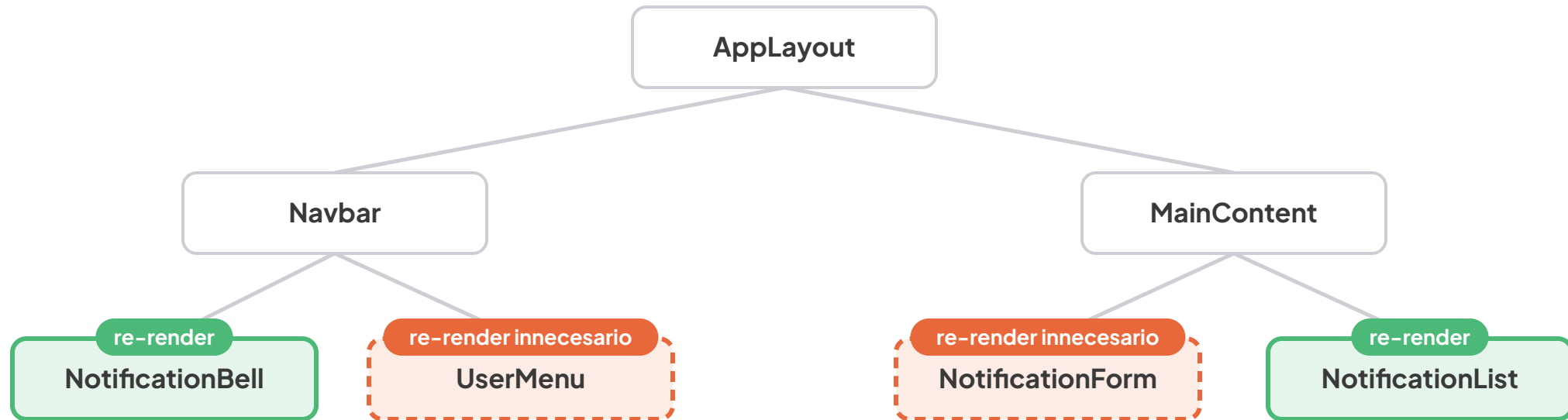
Selectores y acciones asíncronas

Selectores: cuándo se renderiza un componente

Después de cada `set`, Zustand ejecuta el selector y compara con `Object.is` (igualdad por referencia).

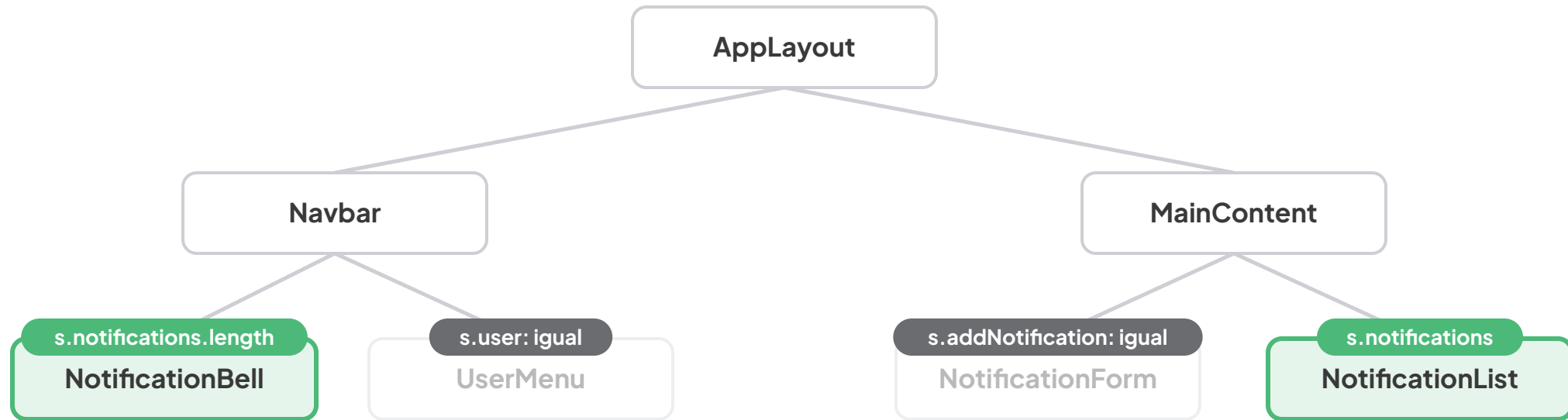
Selector	Qué devuelve	¿Cuándo se renderiza?
<code>useStore()</code>	El store completo	Con cualquier cambio
<code>(s) => s.notifications</code>	Un arreglo	Cuando se reemplaza por otro arreglo
<code>(s) => s.addNotification</code>	Una función	Nunca: la referencia no cambia
<code>(s) => s.notifications.length</code>	Un número	Solo si cambia la cantidad

Sin selector: se renderizan todos los suscritos



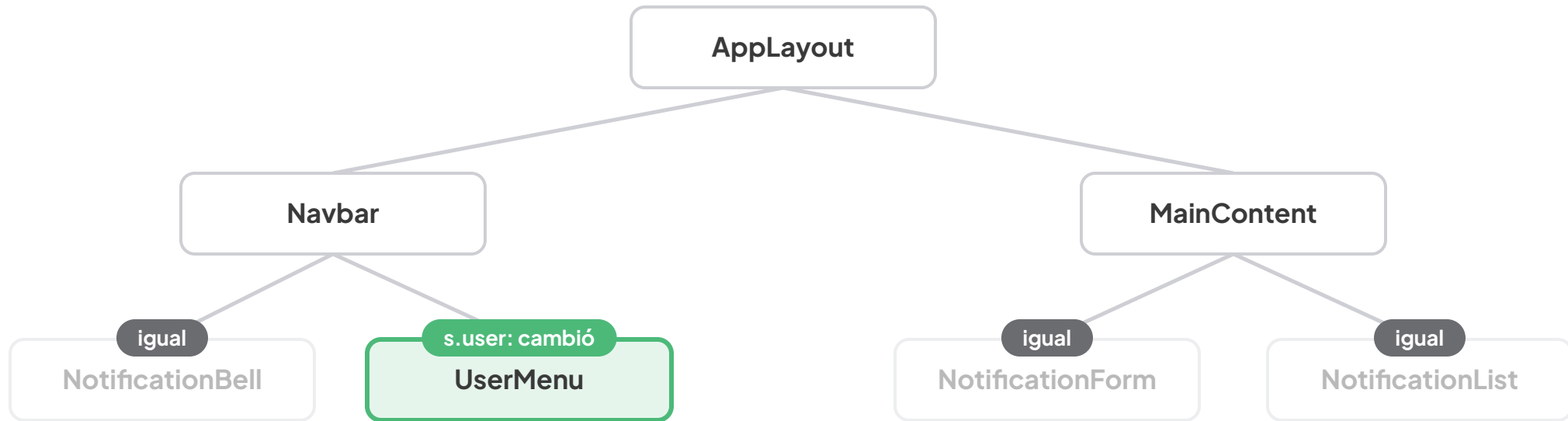
Llega una notificación. Solo NotificationBell y NotificationList necesitan el dato, pero **UserMenu y NotificationForm** (en naranja) también se renderizan.

Con un selector por dato: se renderizan dos



Zustand vuelve a evaluar cada selector. El conteo y la lista cambiaron; el usuario y la función no. **Los otros dos ni se enteran.**

Cambia el usuario: solo se renderiza UserMenu



Otra acción modifica otra parte del store. Ahora solo el componente suscrito a `s.user` se vuelve a dibujar. **El resto del árbol no se toca.**

useShallow: seleccionar varios valores

Bucle en Zustand 5

```
// Cada ejecución crea un objeto NUEVO:  
// Object.is lo ve distinto y vuelve a renderizar  
const { user, token } = useAuthStore(  
  (s) => ({ user: s.user, token: s.token }),  
);
```

Termina en un error de **actualización infinita**.

Con useShallow

```
import { useShallow } from 'zustand/react/shallow';  
  
const { user, token } = useAuthStore(  
  useShallow((s) => ({ user: s.user, token: s.token })),  
);
```

Compara el objeto **propiedad por propiedad**. La alternativa: un selector por valor.

Estado derivado: calcúlalo, no lo guardes

estado derivado

```
// Bien: devuelve un número (primitivo)
const unread = useNotificationsStore(
  (s) => s.notifications.filter((n) => !n.read).length,
);

// Cuidado: filter devuelve un arreglo NUEVO
// en cada ejecución del selector.
// Si necesitas la lista filtrada:
const all = useNotificationsStore((s) => s.notifications);
const unreadItems = all.filter((n) => !n.read);
```



Un valor **primitivo** solo cambia cuando cambia de verdad



Un arreglo o un objeto nuevo en el selector **siempre** se ve distinto



Para listas filtradas: selecciona el arreglo y filtra en el componente

Acciones asíncronas: una función async normal

auth.store.ts

```
1 export const useAuthStore = create<AuthState>()((set)=> ({
2   token: null,
3   isLoading: false,
4   error: null,
5
6   login: async (credentials) => {
7     set({ isLoading: true, error: null });
8     try {
9       const token = await authService.login(credentials);
10      set({ token, isLoading: false });
11      return true;
12    } catch (err) {
13      set({ isLoading: false, error: messageOf(err) });
14      return false;
15    }
16  },
17 }));
```



`isLoading` y `error` viven en el store



La interfaz solo llama `login()` y reacciona al resultado



No hace falta `createAsyncThunk`

El mismo ciclo que un thunk, sin la maquinaria

PENDING

```
set({ isLoading: true })
```

Antes de la petición: enciende el indicador y limpia el error.

FULFILLED

```
set({ token, isLoading: false })
```

Éxito: guarda el resultado y apaga el indicador.

REJECTED

```
set({ error, isLoading: false })
```

Fallo: guarda un mensaje controlado para mostrarlo.