

Estado global, navegación y estilos

Redux Toolkit, navegación en Next.js y Tailwind

Computación en Internet 3 — Universidad Icesi

Mapa de la sesión



Estado y Redux

Del estado local al store global



Navegación

push, replace y redirect



Estilos

Tailwind v4 y daisyUI 5



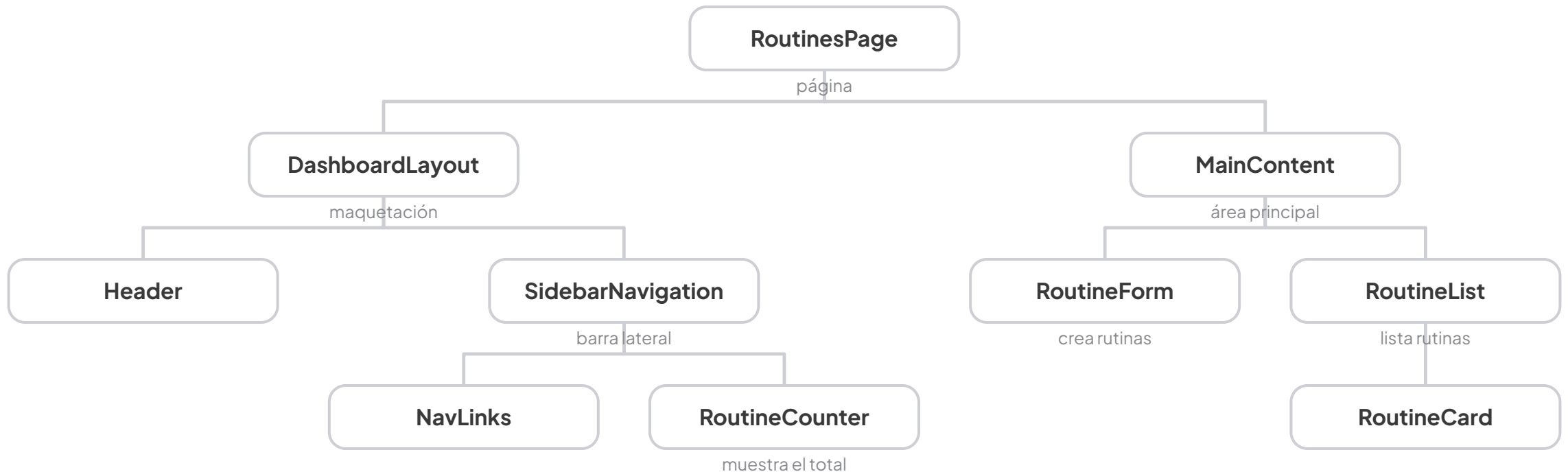
Trabajo en clase

Login, registro, header y navbar

El estado y su alcance

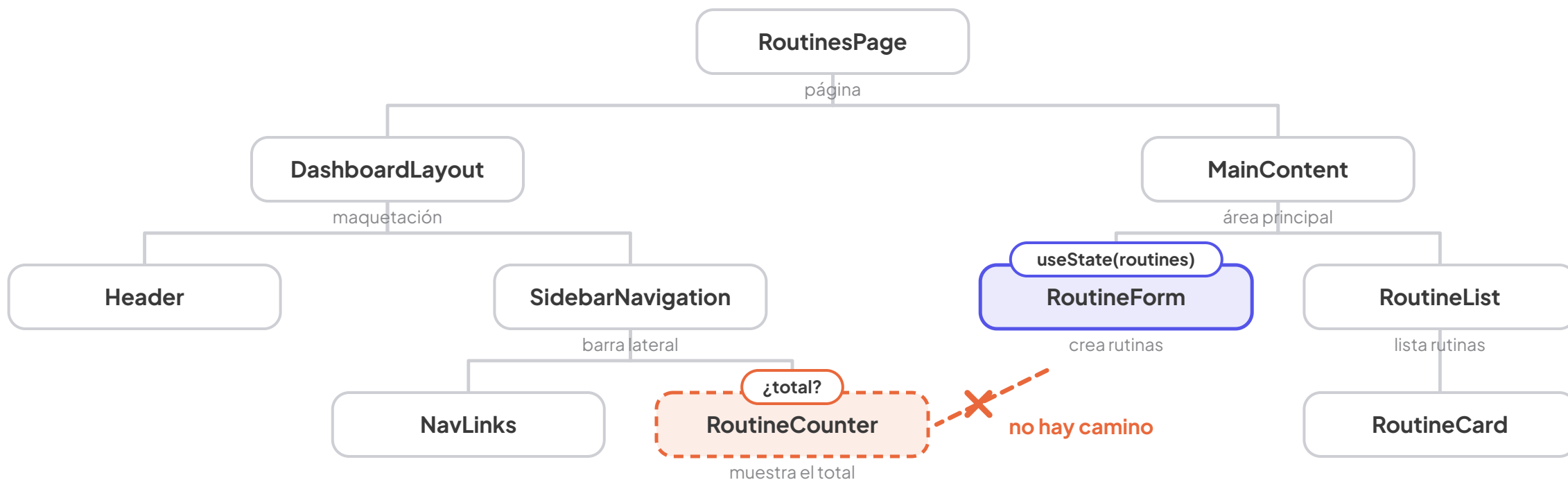
De local a global

Cómo viaja el estado por el árbol



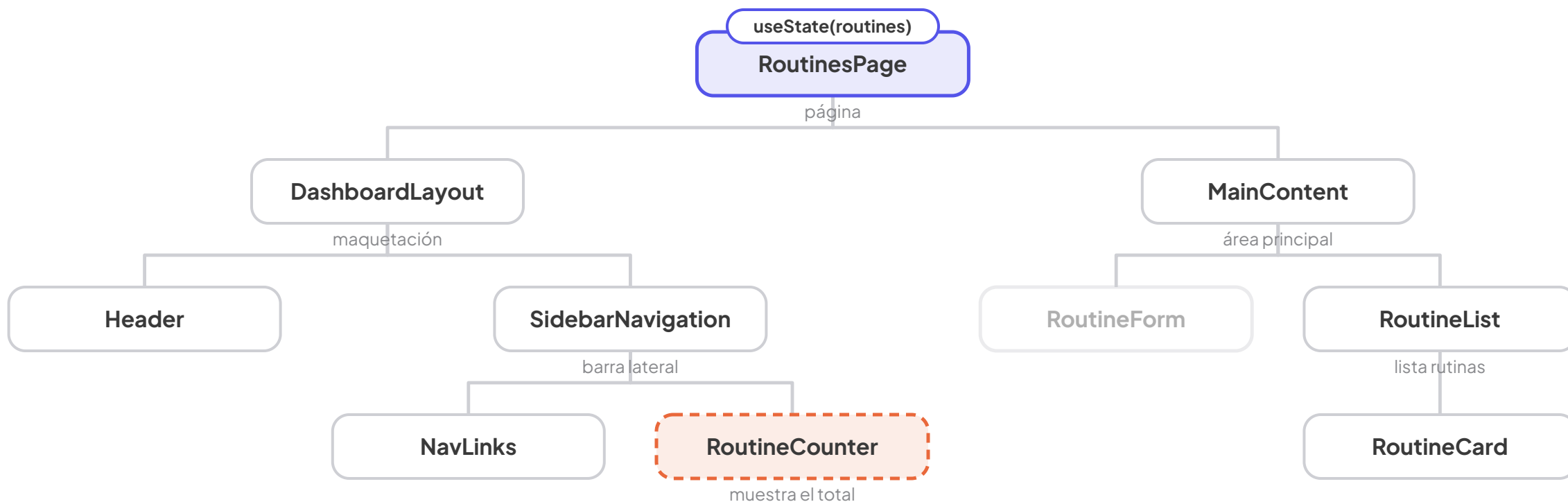
La página es un árbol: los datos solo bajan por las ramas, de padre a hijo.

Cómo viaja el estado por el árbol



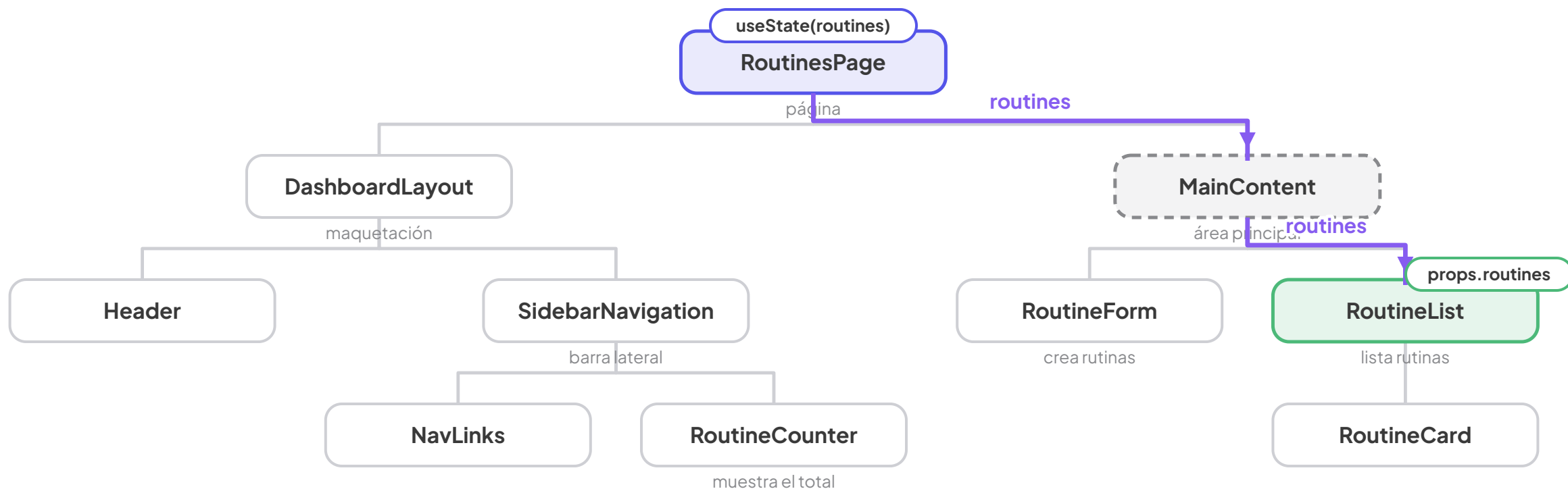
El estado vive en **RoutineForm**. **RoutineCounter** lo necesita y no hay camino entre ramas.

Cómo viaja el estado por el árbol



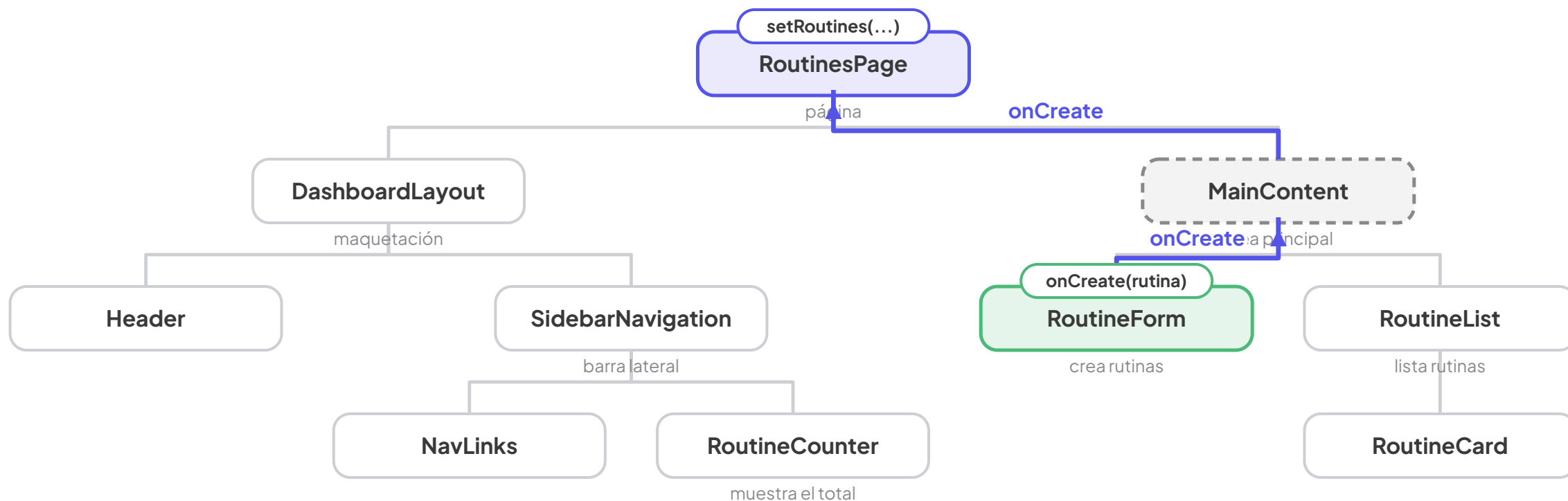
Solución: subir el estado al ancestro común más cercano.

Cómo viaja el estado por el árbol



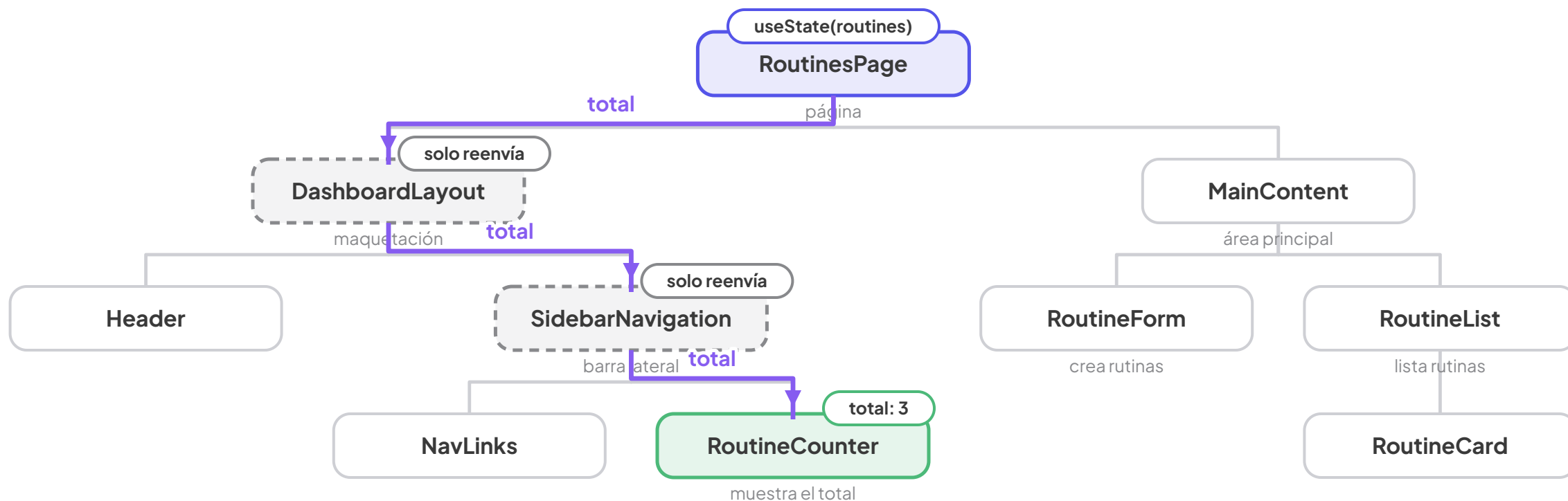
Los datos bajan como props.

Cómo viaja el estado por el árbol



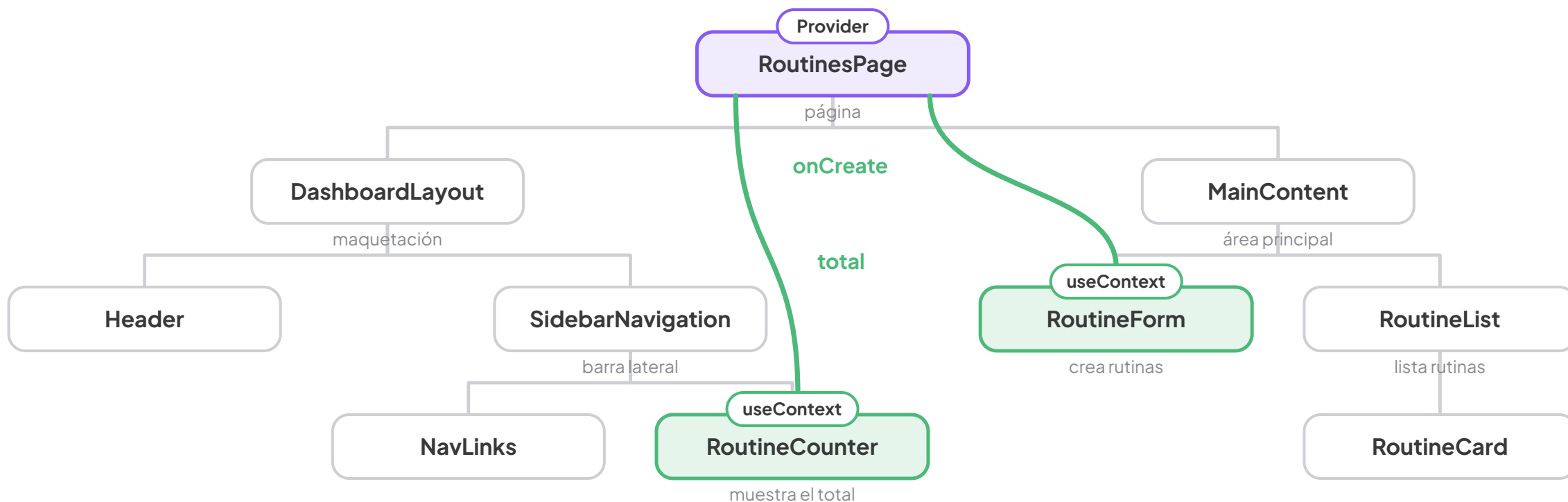
Los cambios suben como callbacks.

Cómo viaja el estado por el árbol



Si el consumidor está lejos, los intermedios solo reenvían: prop drilling.

Cómo viaja el estado por el árbol



Context: el consumidor lee directo, sin pasar por los intermedios.

Cuándo subir de peldaño

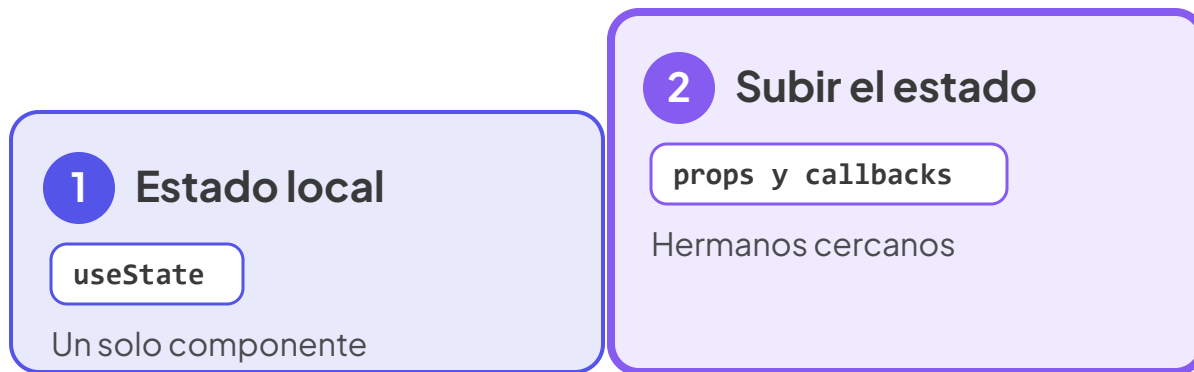
1 Estado local

`useState`

Un solo componente

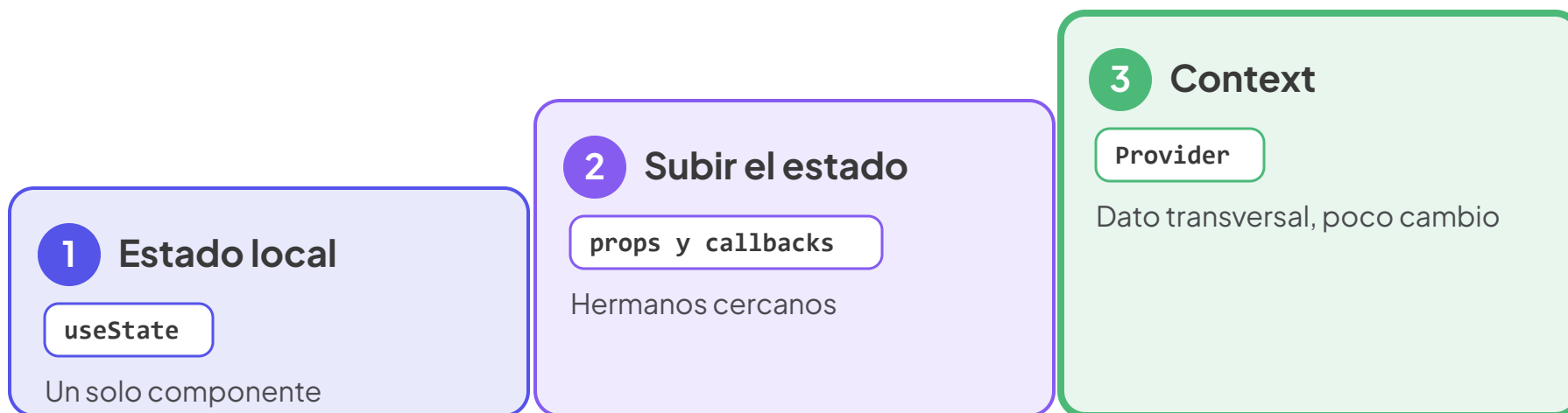
Empieza simple: el estado local alcanza para un componente.

Cuándo subir de peldaño



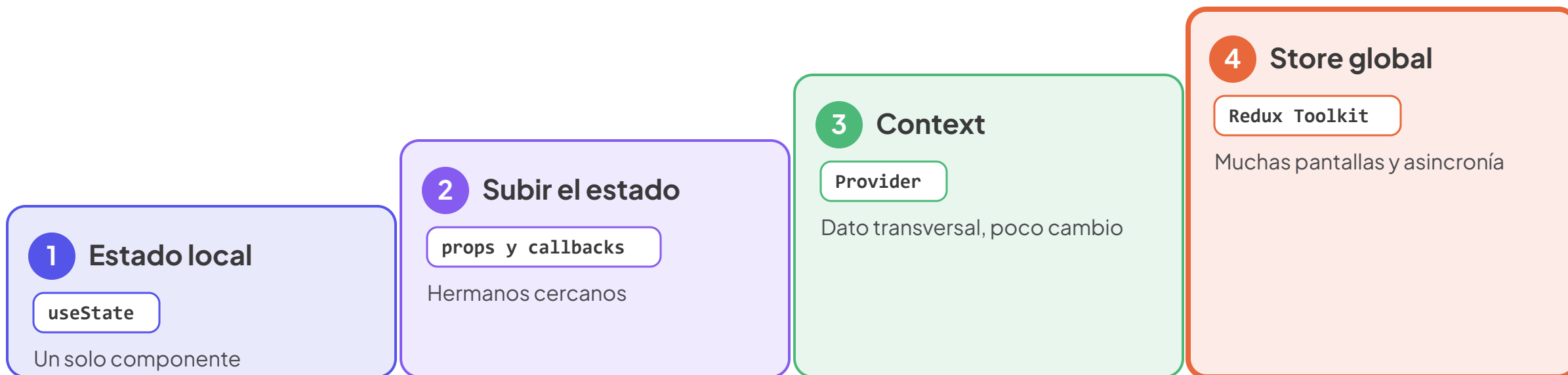
Si dos hermanos lo necesitan, sube al ancestro común.

Cuándo subir de peldaño



Si el dato está lejos y casi no cambia, Context evita el prop drilling.

Cuándo subir de peldaño



Si lo usan muchas pantallas y hay asincronía, un store global con Redux.

Redux Toolkit

Un solo lugar para el estado global

Los tres principios de Redux



Una sola fuente de verdad

Todo el estado de la app vive en un único store.

Los tres principios de Redux



Una sola fuente de verdad

Todo el estado de la app vive en un único store.



Estado de solo lectura

Nadie lo modifica directo: se pide el cambio con una action.

Los tres principios de Redux



Una sola fuente de verdad

Todo el estado de la app vive en un único store.



Estado de solo lectura

Nadie lo modifica directo: se pide el cambio con una action.



Funciones puras

Un reducer recibe estado y action y devuelve el estado nuevo.

Vocabulario de Redux



Store

La bóveda: guarda el estado de toda la app.

```
configureStore({ reducer })
```



Action

El formulario que dice qué pasó.

```
{ type: 'routines/create', payload }
```

Vocabulario de Redux

Store

La bóveda: guarda el estado de toda la app.

```
configureStore({ reducer })
```

Action

El formulario que dice qué pasó.

```
{ type: 'routines/create', payload }
```

dispatch

El buzón: la única forma de pedir un cambio.

```
dispatch(action)
```

Reducer

Función pura que calcula el estado nuevo.

```
(state, action) => newState
```

Vocabulario de Redux

Store

La bóveda: guarda el estado de toda la app.

```
configureStore({ reducer })
```

Action

El formulario que dice qué pasó.

```
{ type: 'routines/create', payload }
```

dispatch

El buzón: la única forma de pedir un cambio.

```
dispatch(action)
```

Reducer

Función pura que calcula el estado nuevo.

```
(state, action) => newState
```

Slice

Estado, reducers y actions de una feature.

```
createSlice({ name, ... })
```

Selector

Lee solo la parte del estado que necesitas.

```
useSelector(s => s.routines.items)
```

El flujo de datos en Redux

Componente

RoutineForm: clic en Guardar

El usuario pulsa "Guardar" en el formulario.

El flujo de datos en Redux



El componente no modifica nada: despacha una action.

El flujo de datos en Redux



El reducer calcula el estado nuevo a partir del estado y la action.

El flujo de datos en Redux



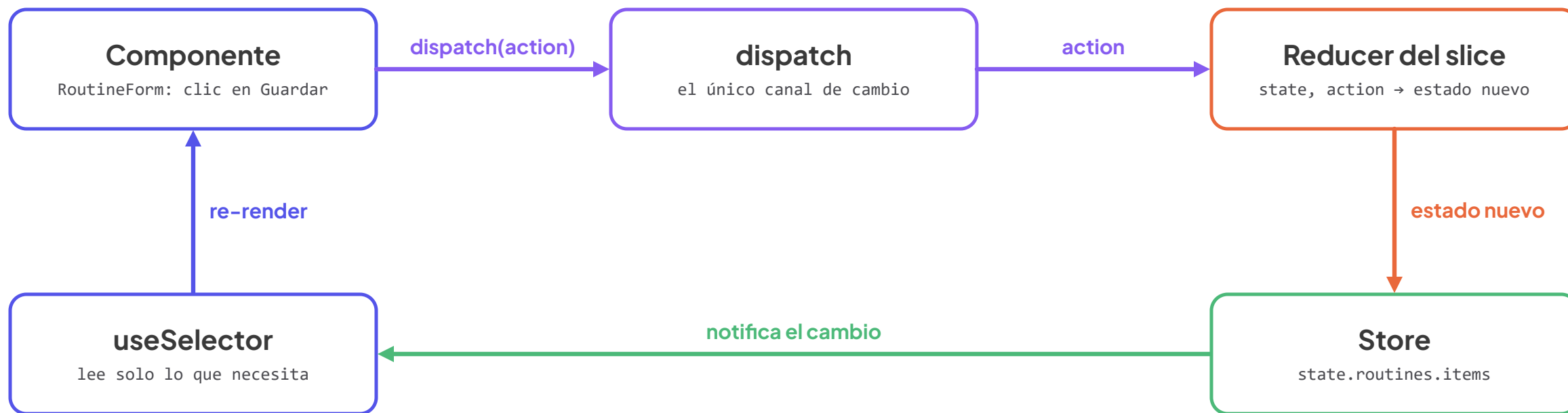
El store guarda el estado nuevo: sigue siendo la única fuente de verdad.

El flujo de datos en Redux



Los selectores revisan si lo que leen cambió.

El flujo de datos en Redux



Solo los componentes afectados se vuelven a renderizar.

Un slice por dentro

routines.slice.ts

```
const routinesSlice = createSlice({
  name: 'routines',
  initialState: { items: [], isLoading: false, errorMessage: null },
  reducers: {
    clearError: (state) => { state.errorMessage = null; },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchRoutines.pending, (s) => { s.isLoading = true; })
      .addCase(fetchRoutines.fulfilled, (s, a) => { s.items = a.payload; });
  },
});

export const { clearError } = routinesSlice.actions;
```

name e initialState

name es el prefijo de las actions.
initialState es el estado de partida.

Un slice por dentro

routines.slice.ts

```
const routinesSlice = createSlice({
  name: 'routines',
  initialState: { items: [], isLoading: false, errorMessage: null },
  reducers: {
    clearError: (state) => { state.errorMessage = null; },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchRoutines.pending, (s) => { s.isLoading = true; })
      .addCase(fetchRoutines.fulfilled, (s, a) => { s.items = a.payload; });
  },
});

export const { clearError } = routinesSlice.actions;
```

reducers

Cambios síncronos. Genera actions: clearError().

Un slice por dentro

```
routines.slice.ts

const routinesSlice = createSlice({
  name: 'routines',
  initialState: { items: [], isLoading: false, errorMessage: null },
  reducers: {
    clearError: (state) => { state.errorMessage = null; },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchRoutines.pending, (s) => { s.isLoading = true; })
      .addCase(fetchRoutines.fulfilled, (s, a) => { s.items = a.payload; });
  },
});

export const { clearError } = routinesSlice.actions;
```

extraReducers

Responde a los thunks: pending, fulfilled y rejected.

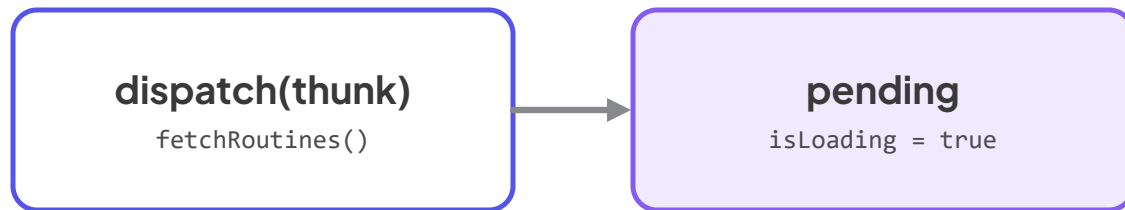
El thunk: dónde vive la API

dispatch(thunk)

`fetchRoutines()`

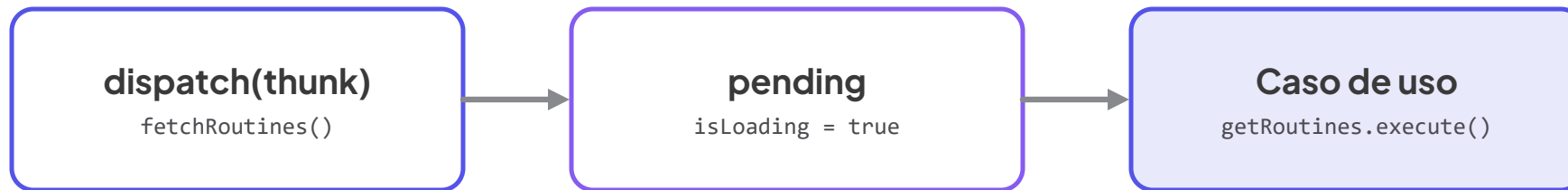
Un thunk es una función asíncrona que se despacha como una action.

El thunk: dónde vive la API



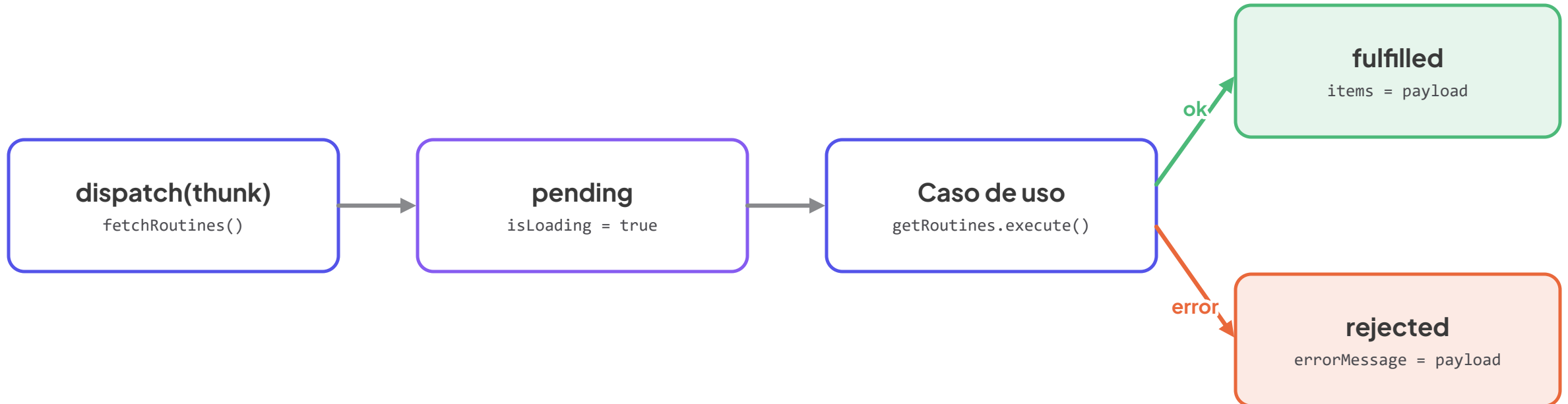
Primero se despacha pending: la interfaz puede mostrar "cargando".

El thunk: dónde vive la API



Después trabaja el caso de uso: el thunk nunca llama a fetch directo.

El thunk: dónde vive la API



Según el resultado se despacha fulfilled o rejected, con el dato o el error en el payload.

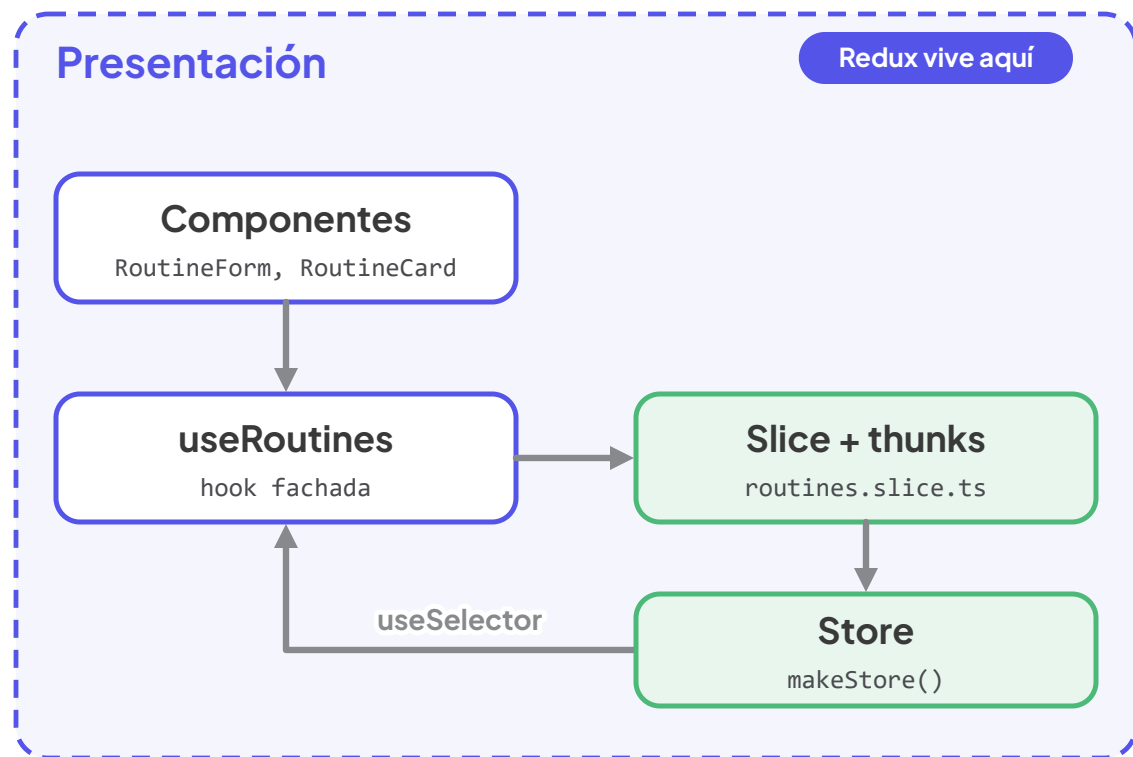
Redux, Zustand y Context

	Context	Zustand	Redux Toolkit
Boilerplate	Mínimo	Casi nulo	Moderado
Re-renders	Todos los consumidores	Por selector	Por selector
DevTools	No	Opcionales	Nativas
Estructura	Ninguna	Libre	Impone una convención
Úsalo para	Tema o idioma	Apps medianas	Equipos y apps grandes

Redux en el proyecto

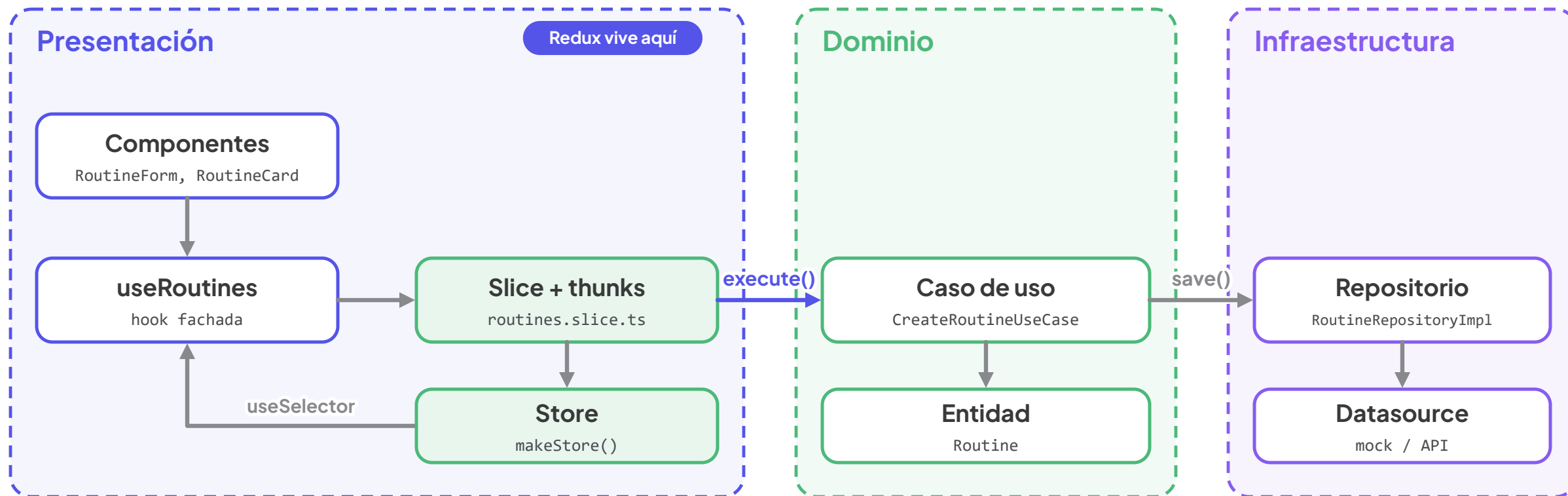
Dentro de Clean Architecture

Dónde vive Redux en la arquitectura



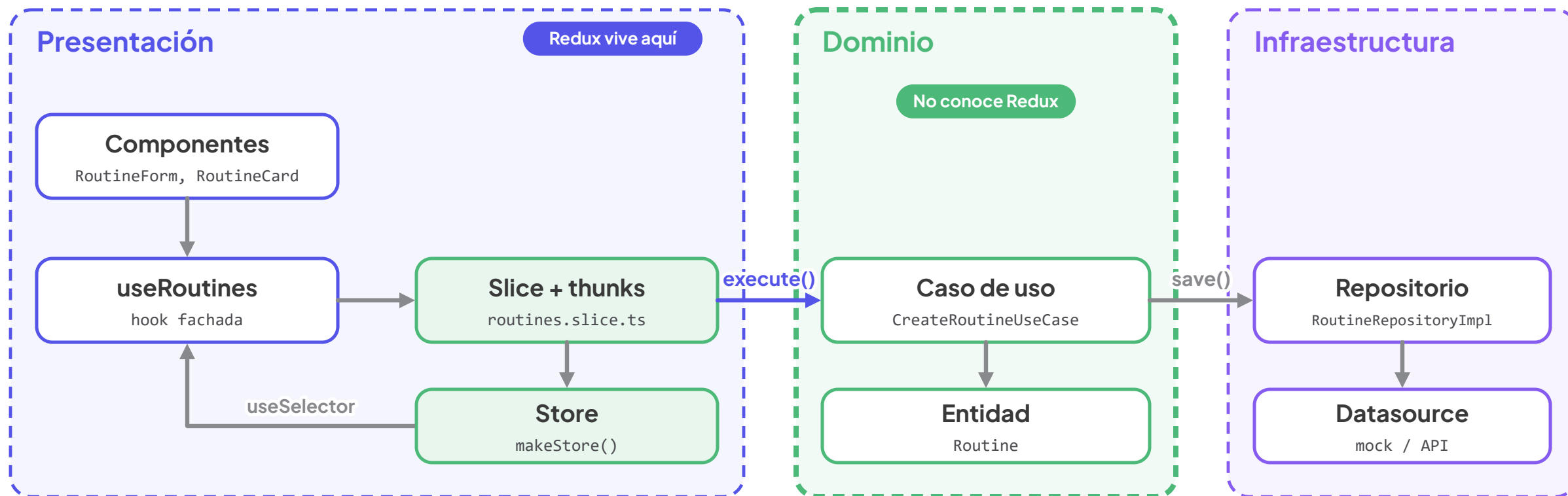
Redux es un detalle de la capa de Presentación.

Dónde vive Redux en la arquitectura



El thunk entra al dominio por el caso de uso, igual que antes.

Dónde vive Redux en la arquitectura

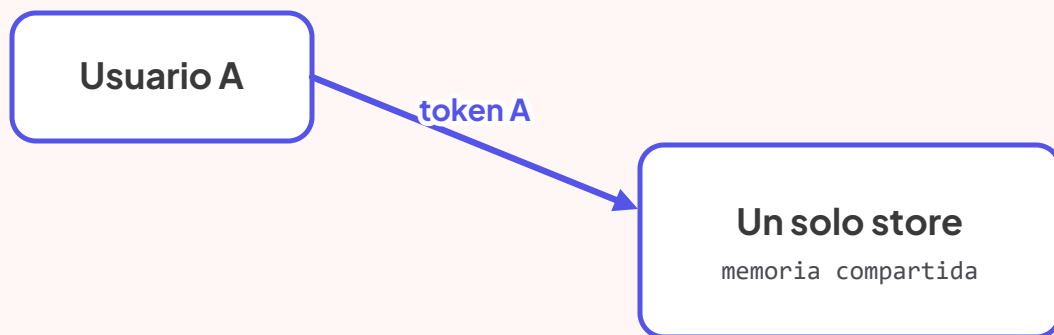


El dominio no importa Redux, React ni Next: no cambió.

Por qué makeStore y no un singleton

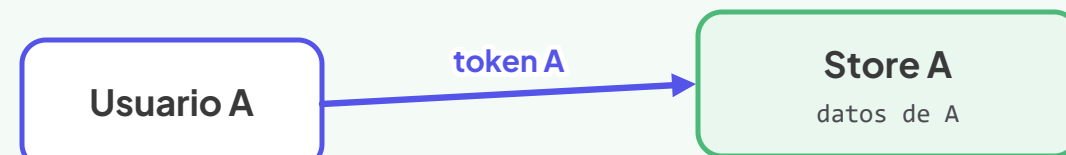
Singleton

```
export const store = configureStore(...)
```



Fábrica

```
export const makeStore = () => configureStore(...)
```

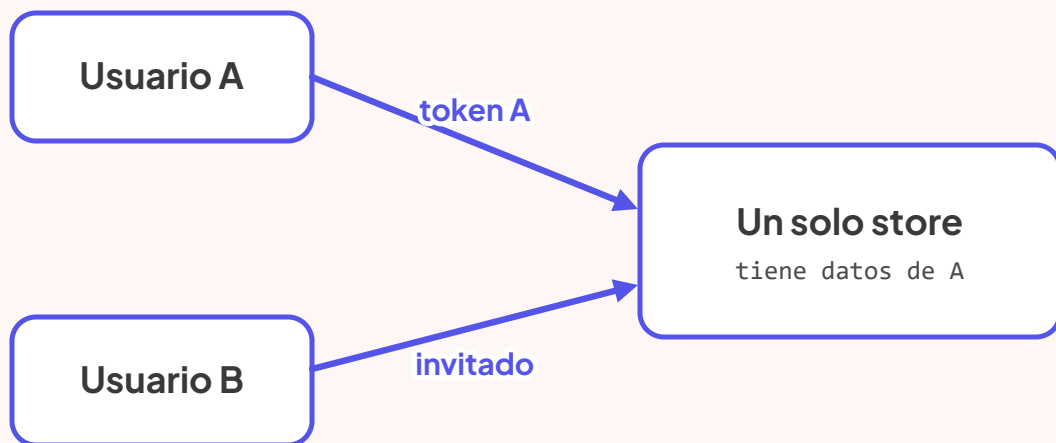


Llega la petición del usuario A y se llena el store.

Por qué makeStore y no un singleton

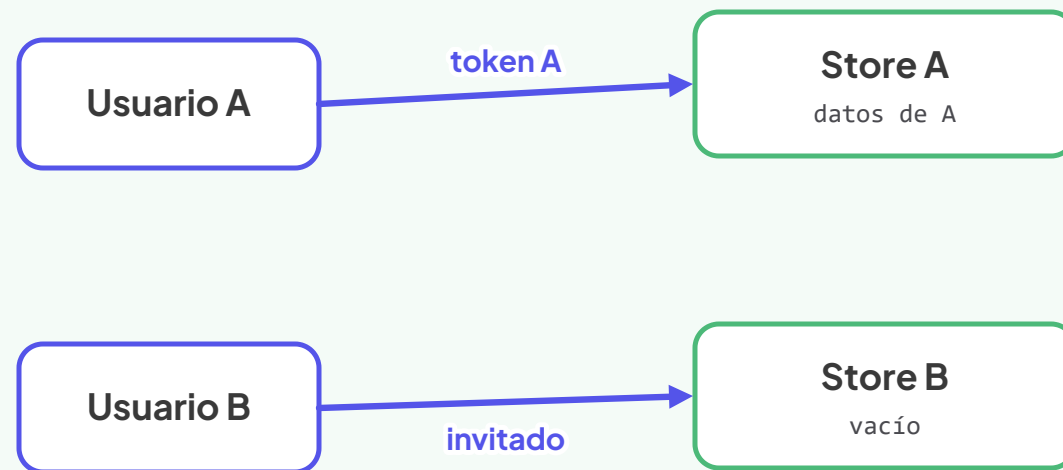
Singleton

```
export const store = configureStore(...)
```



Fábrica

```
export const makeStore = () => configureStore(...)
```

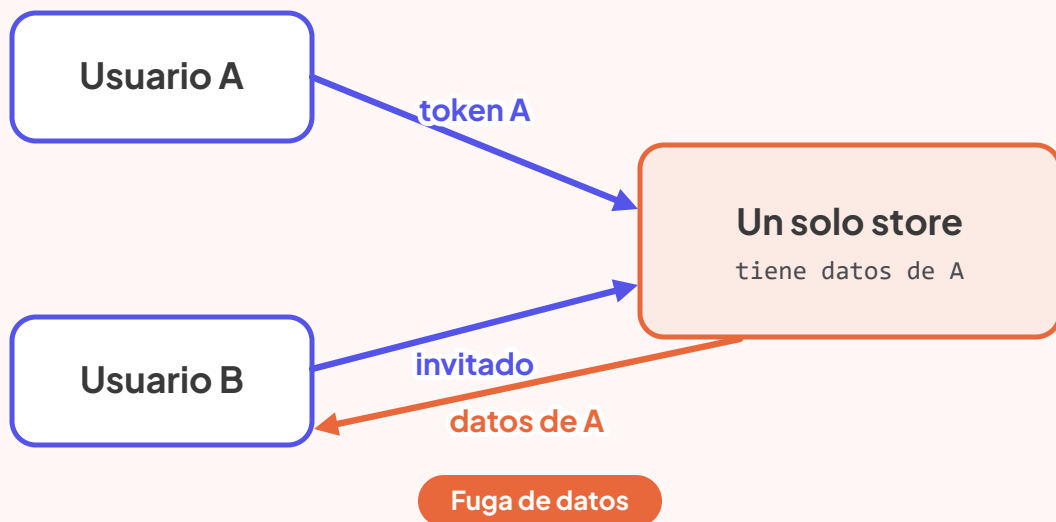


Llega a la vez la petición del usuario B.

Por qué makeStore y no un singleton

Singleton

```
export const store = configureStore(...)
```



Fábrica

```
export const makeStore = () => configureStore(...)
```



Singleton: B puede ver datos de A. makeStore: cada petición, su propio store.

Los archivos de la integración

■ src/store (transversal) ■ dentro de la feature ■ modificado

store.ts

makeStore y los tipos RootState y AppDispatch.

hooks.ts

useAppDispatch y useAppSelector, ya tipados.

StoreProvider.tsx

El Provider de React, en un componente cliente.

routines.slice.ts

Estado, thunks y reducers de la feature.

use-routines.ts

La fachada: modificada, con la misma API.

app/layout.tsx

Envuelve la app con StoreProvider.

El slice: el thunk llama al caso de uso

routines.slice.ts

```
export const fetchRoutines = createAsyncThunk(
  'routines/fetchAll',
  async (_, { rejectWithValue }) => {
    try {
      const entities = await routineContainer.getRoutines.execute();
      return entities.map(RoutineViewModelMapper.toViewModel);
    } catch (err) {
      return rejectWithValue(err instanceof Error ? err.message : 'Error');
    }
  }
);
```

El thunk llama al caso de uso, no a fetch. Al estado entran ViewModels planos.

store.ts: una fábrica, no una constante

src/store/store.ts

```
export const makeStore = () =>
  configureStore({
    reducer: { routines: routinesReducer },
  });

export type AppStore = ReturnType<typeof makeStore>;
export type RootState = ReturnType<AppStore['getState']>;
export type AppDispatch = AppStore['dispatch'];
```

Cada petición del servidor crea su propio store.

StoreProvider.tsx: el puente hacia React

src/store/StoreProvider.tsx

```
'use client';

export function StoreProvider({ children }: { children: ReactNode }) {
  const [store] = useState<AppStore>(makeStore);

  return <Provider store={store}>{children}</Provider>;
}
```

'use client' porque Provider usa Context. useState(makeStore) lo crea una sola vez.

layout.tsx: envolver la aplicación

```
app/layout.tsx
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <StoreProvider>{children}</StoreProvider>
      </body>
    </html>
  );
}
```

El layout sigue siendo Server Component; StoreProvider es una hoja de cliente.

use-routines.ts: la API no cambia

use-routines.ts (extracto)

```
export function useRoutines() {  
  const dispatch = useAppDispatch();  
  const routines = useAppSelector((s) => s.routines.items);  
  const isLoading = useAppSelector((s) => s.routines.isLoading);  
  
  const createRoutine = async (params: CreateRoutineParams) => {  
    const result = await dispatch(createRoutineMutation(params));  
    return createRoutineMutation.fulfilled.match(result);  
  };  
  
  return { routines, isLoading, createRoutine };  
}
```

Misma API que con useState: los componentes no cambiaron una línea de JSX.