



Context (Estado global)

Módulo 7



Elaborado por: Jesua Luján

Jesua Hadai Luján



DEV.F.:



Objetivo: Entender el uso correcto del estado Global en las aplicaciones con React.

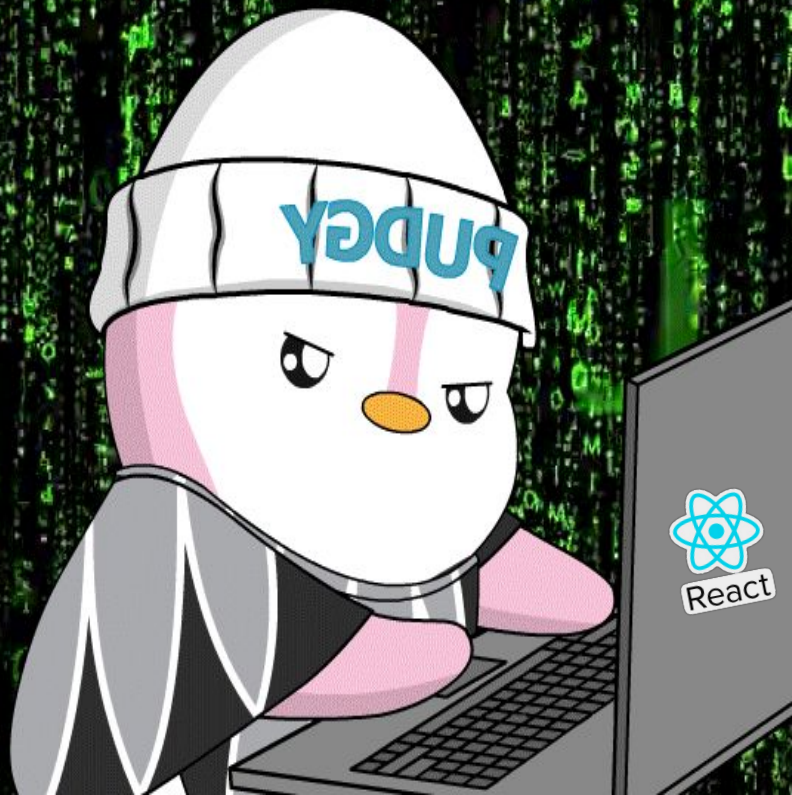
Contenido:

✓ Entender **cómo se comunican los componentes** en React. 🤔

✓ Identificar la problemática común de la **comunicación entre componentes.** 🧑

✓ Entender el concepto y la necesidad de uso de **contextos globales.** 🧑

✓ Aprender a utilizar el **hook de Context** en React. 😈





¿Cómo pasamos la
información de un
Componente a Otro?

Recordatorio



DEV.F.:



Comunicación

(props)

La comunicación entre componentes es de padres a hijos y **pasamos los datos a través de props** (propiedades).

JSX

```
<MyComponent  
  prop="My Value"  
>
```

Component

```
function MyComponent({ prop }) {  
  return <span>{prop}</span>;  
}
```

Output

```
<span>My Value</span>
```





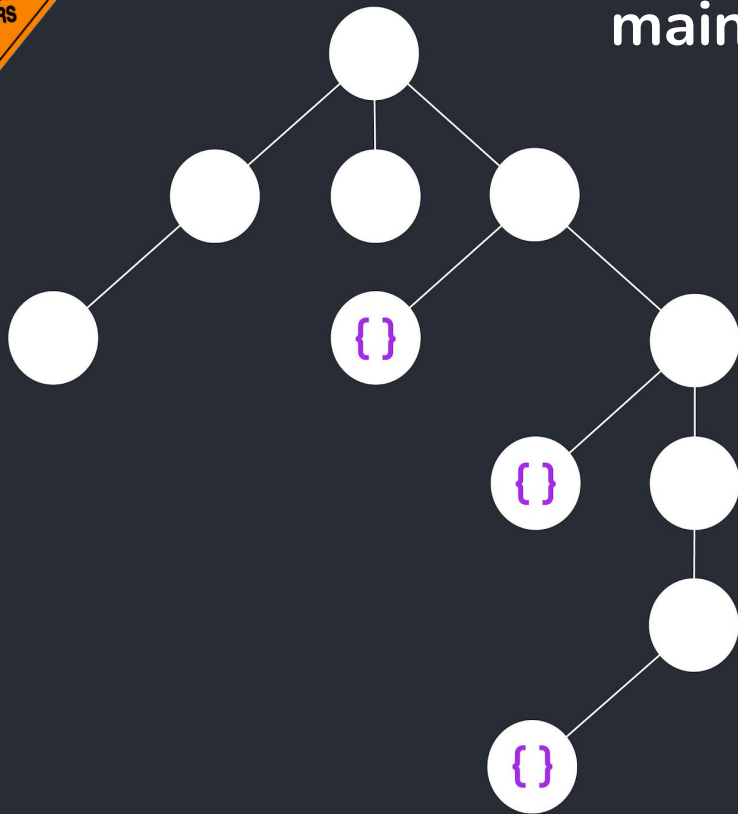
Flujo de comunicación

Entre Componentes



DEV.F.:

main



Árbol

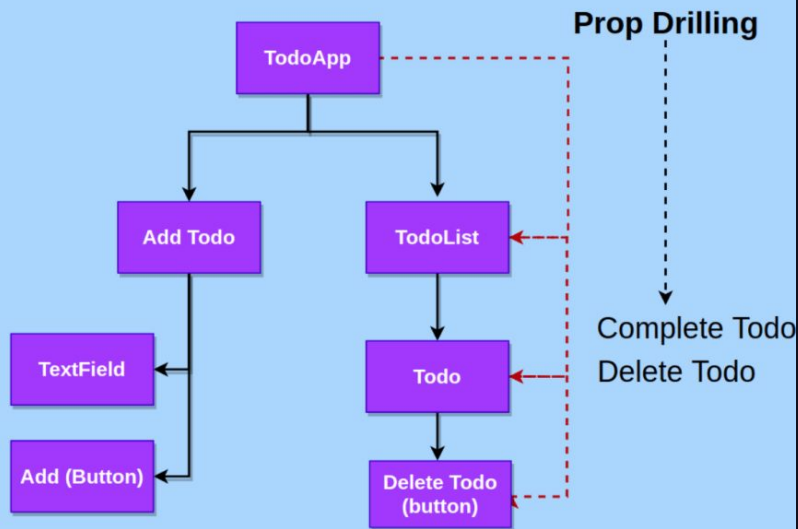
En React **todo es un árbol.**

El componente con mayor poder jerárquico comúnmente maneja la lógica y comparte por **props** a los **demás componentes hijos**, esto permite tener:

- Componentes con lógica
- Componente visuales.



How to handle Prop Drilling?

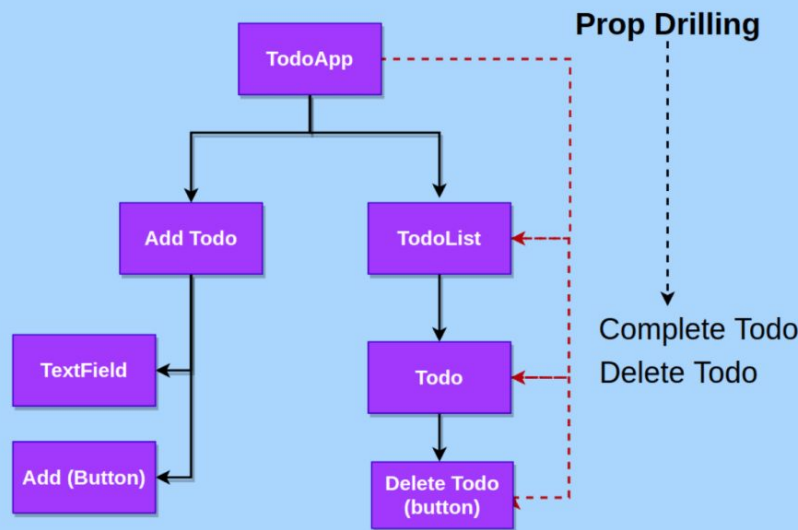


Prop Drilling

Prop Drilling es el término no oficial usado para decir que pasaremos datos a través de varios componentes hijos anidados, en un intento de entregar estos datos a un componente profundamente anidado.



How to handle Prop Drilling?



Prop Drilling

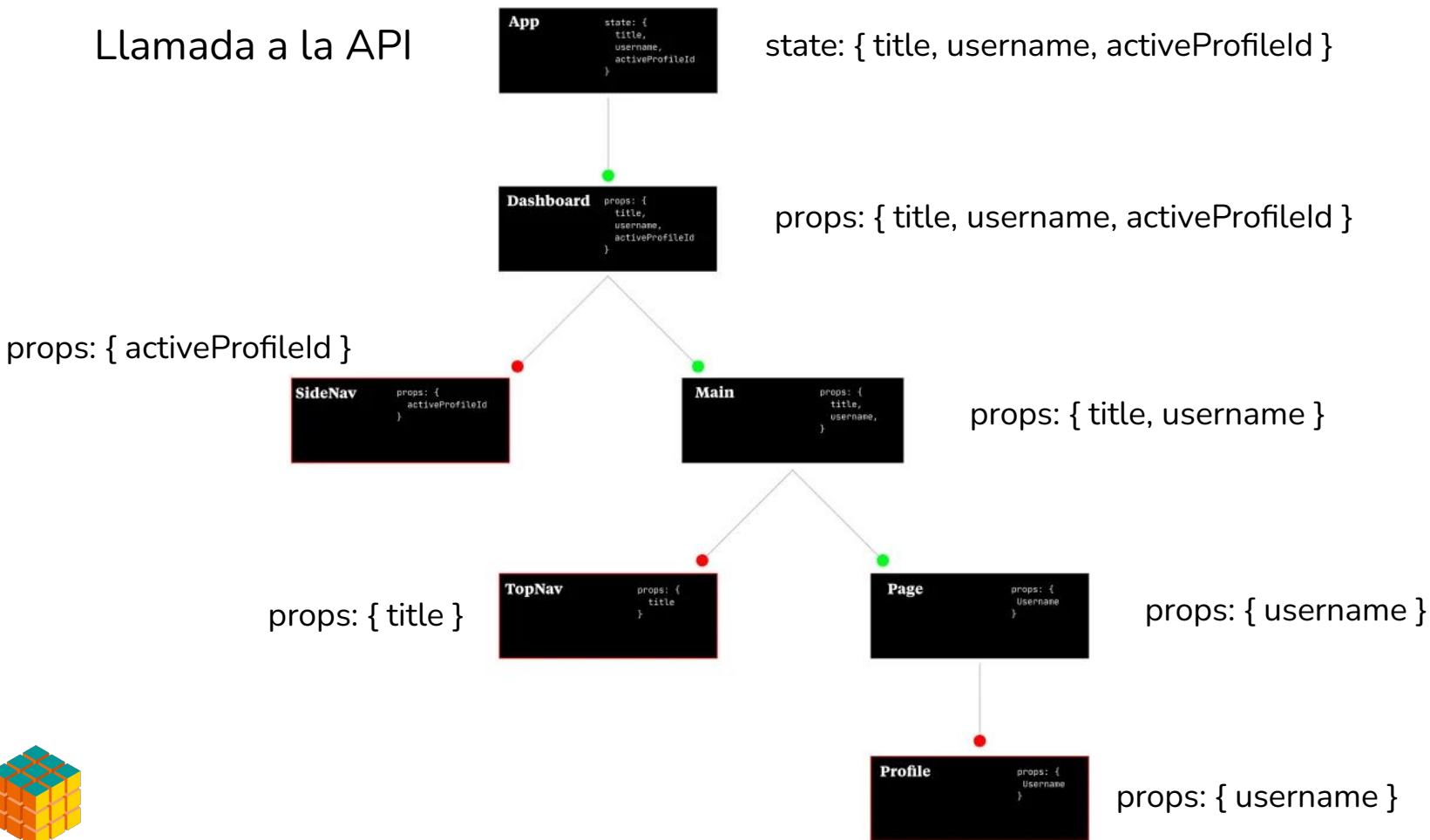
El problema con este enfoque es que la mayoría de los componentes a través de los cuales **se pasan estos datos no tienen ninguna necesidad real de estos datos.**

Simplemente se utilizan como medios para transportar estos datos a su componente de destino.

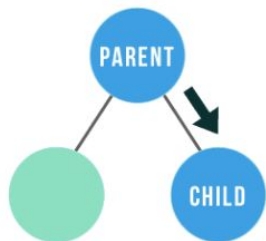
Esto puede causar importantes problemas de reutilización de componentes y de rendimiento de la aplicación.



Llamada a la API

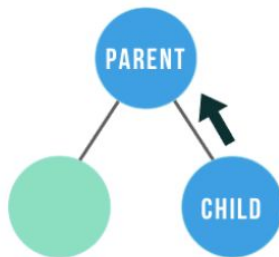


Use this as your *table of contents* to jump around and pick the strategy or strategies that apply to your use-case.



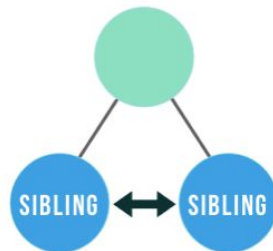
Parent to Child

1. Props
2. Instance Methods



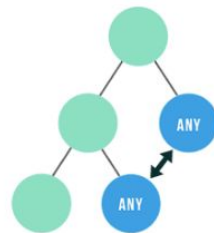
Child to Parent

3. Callback Functions
4. Event Bubbling



Sibling to Sibling

5. Parent Component



Any to Any

6. Observer Pattern
7. Global Variables
8. Context





DISCUSIÓN

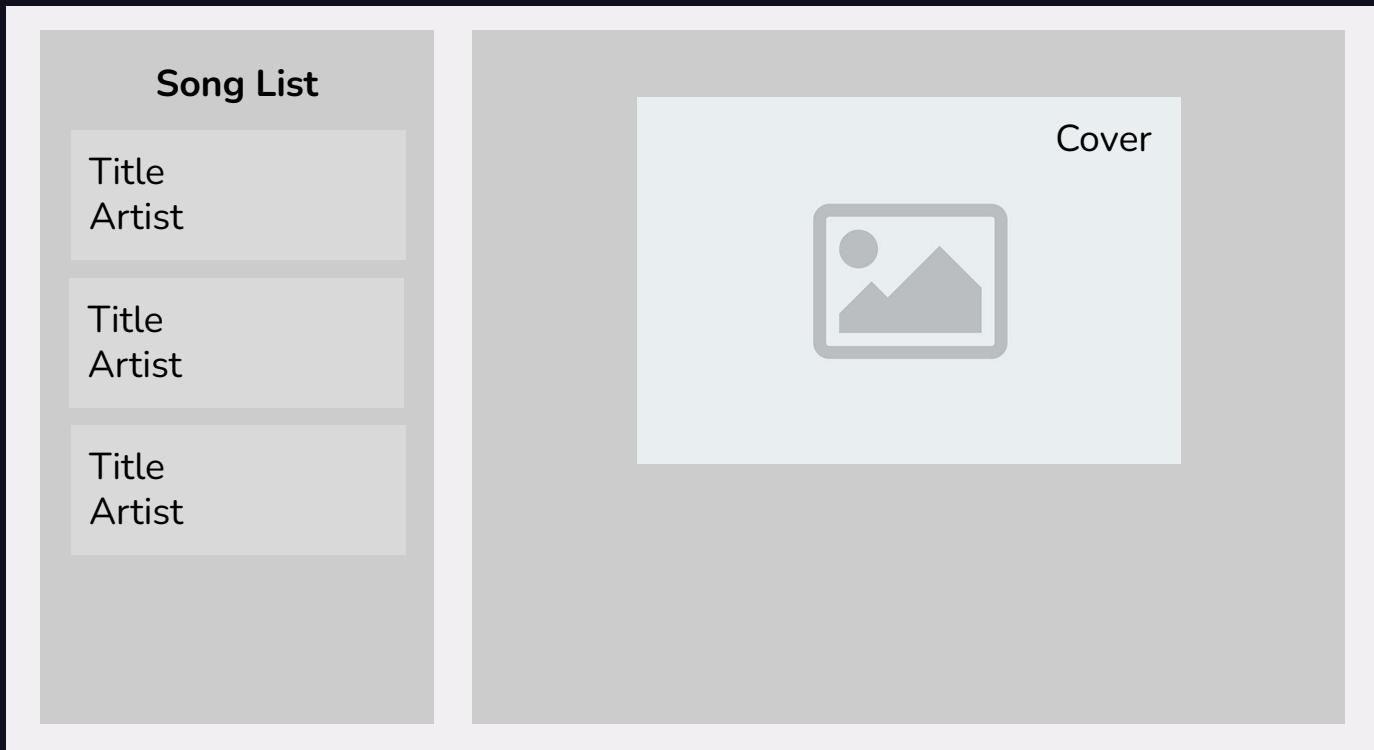
Arquitectura



DEV.F.:

Arquitectura de Reproductor Musical

¿Qué componentes identificas?

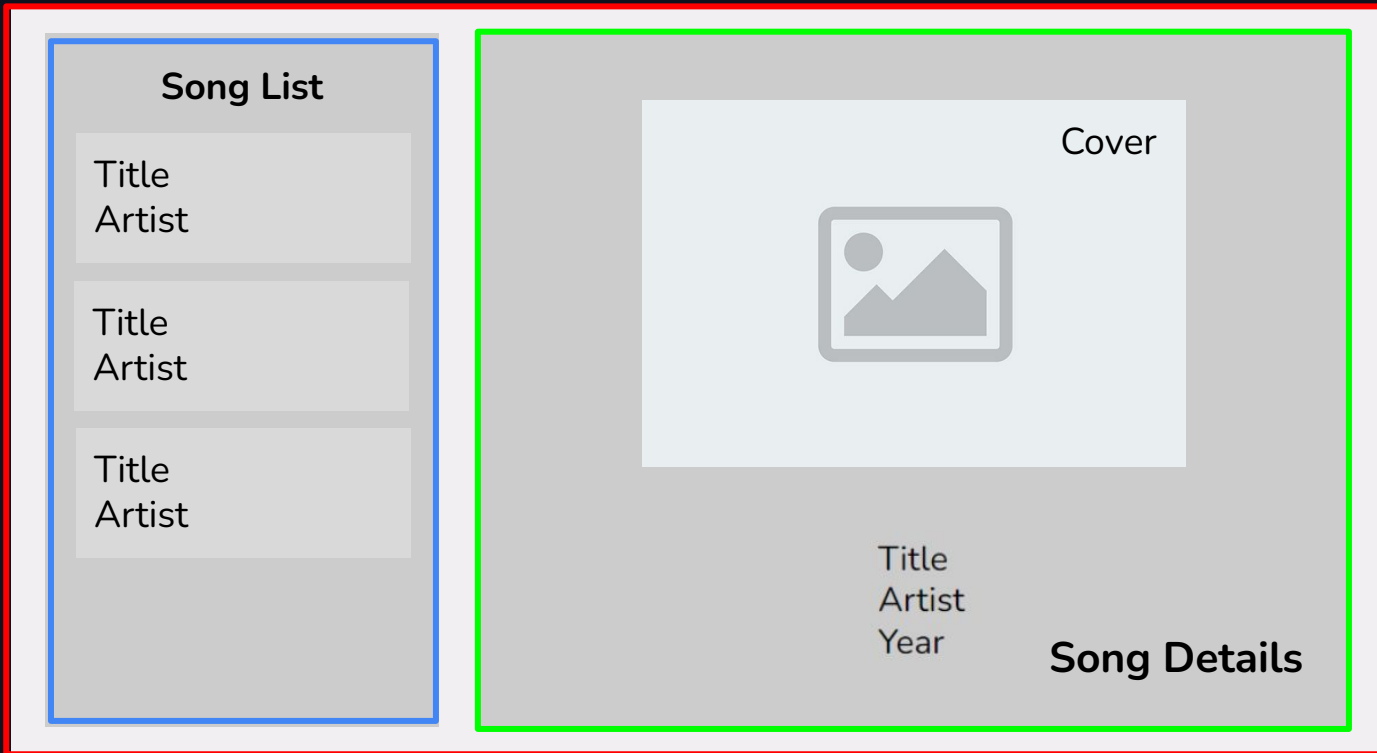


Arquitectura de Reproductor Musical

¿Qué componentes identificas?

Home

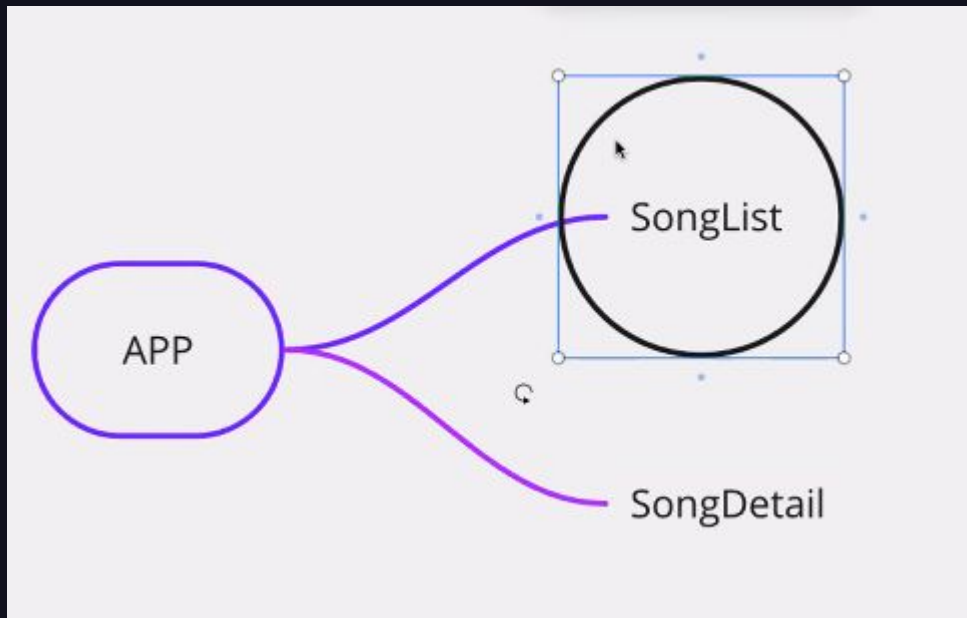
Al hacer clic en una canción, se actualizará la información en la parte derecha de la aplicación.



Mientras en la parte derecha no haya una canción elegida, mostrará una leyenda que solicitará al usuario que elija una canción.



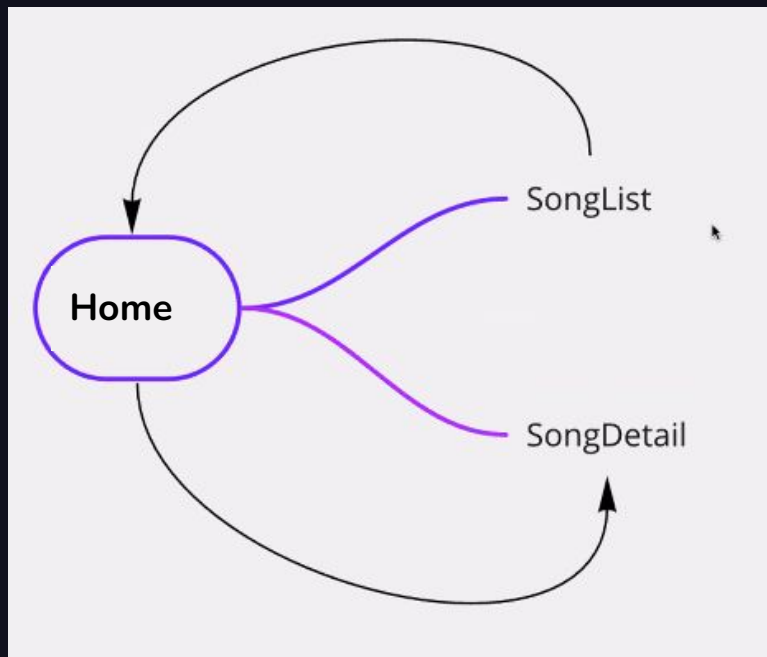
Información de API en SongList



Si donde mandamos a llamar la información de la API es en **SongList**, entonces el otro componente **SongDetail** no tiene la forma de tener acceso directo a esa información.



Enviando información desde SongList...



Una posible solución es: Que **SongList** envíe la información de la canción seleccionada a Home y de ahí a **SongDetail**, pero esto no es viable porque la información da más vueltas y complica la lógica.





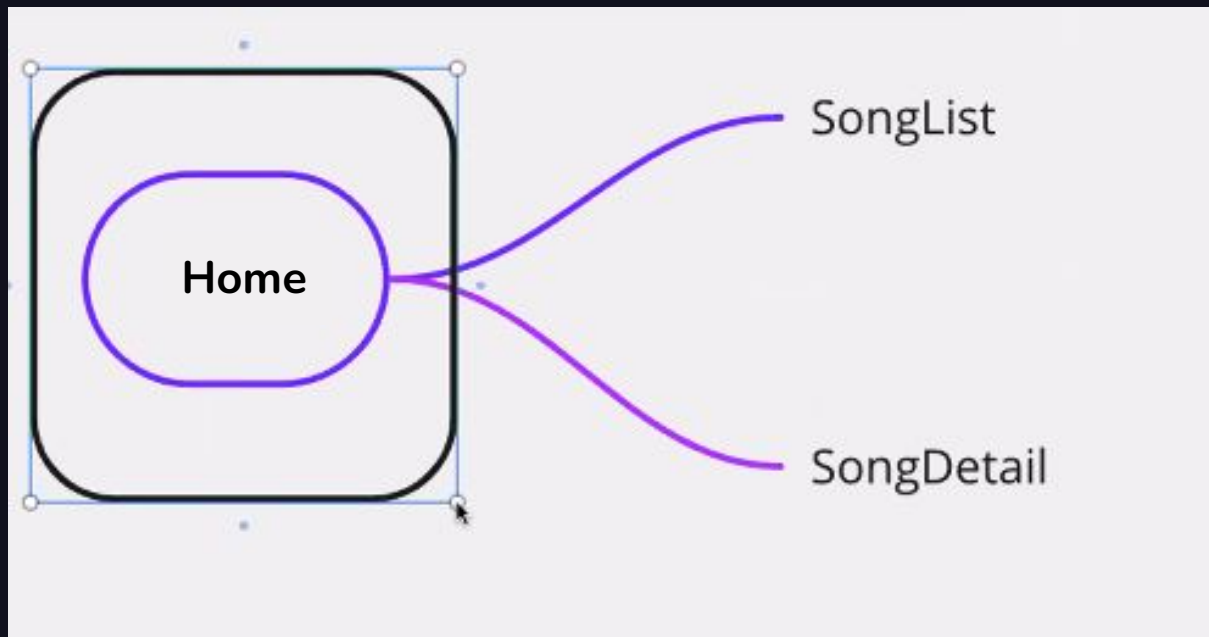
***NO SE PUEDEN COMPARTIR
DATOS ENTRE COMPONENTES
DE LA MISMA JERARQUÍA.***

**SOLO DE PADRE A HIJO Y
VICEVERSA**



DEV.F.:

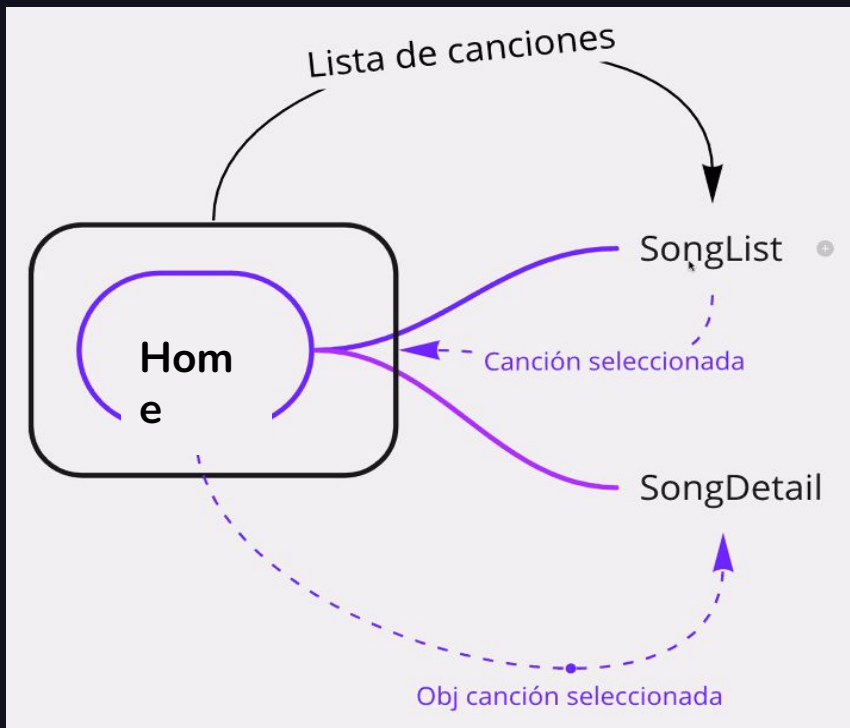
Información de API en Home



La solución más viable es mover la lógica de quien pide la información al padre de ambos: **Home**



Analizando la Lógica en Home



Al mover la lógica a Home, se manejaría con props:

1. **Home** obtiene la información de la API.
2. **SongList** recibe la lista de canciones de **Home**.
3. **SongList** envía de regreso a **Home** la canción seleccionada.
4. **Home** notifica a **SongDetail** que una canción fue seleccionada.
5. **SongDetail** pinta esta información.

Este enfoque hace mucho ir y venir, el problema es que existe mucha circulación de información y **Home** realmente no necesita conocer toda esta información que manejan sus hijos.





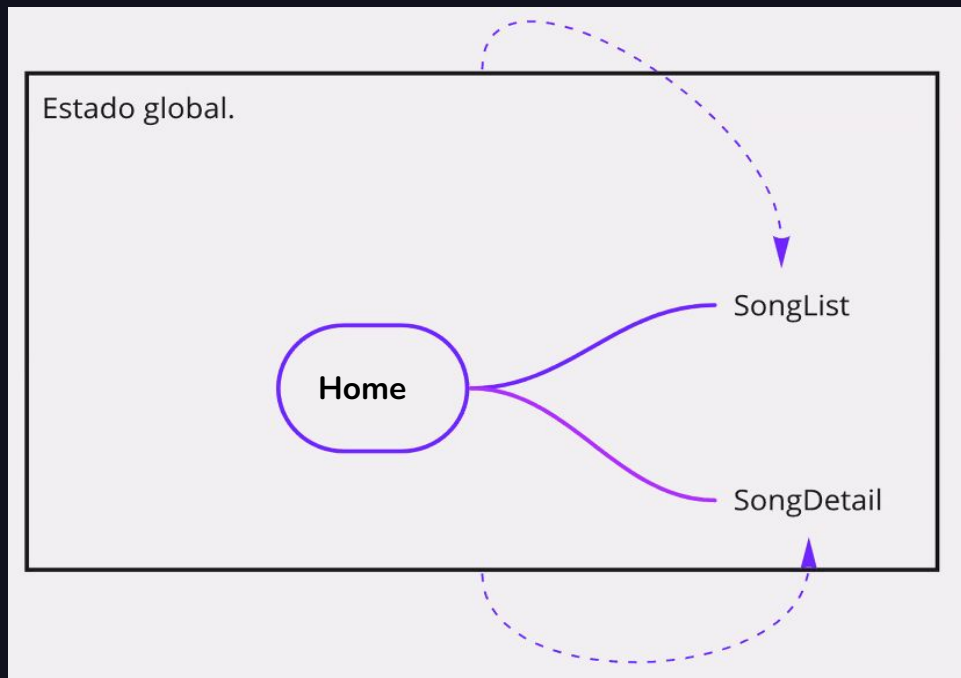
Solución

Estado Global



DEV.F.:

Estado Global



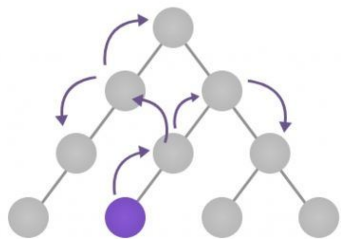
***si se solicita la lista
de canciones se
pide
al estado global***

*Podemos crear un estado global que tenga de habilidad de
compartir las listas de canciones y la canción seleccionada.*



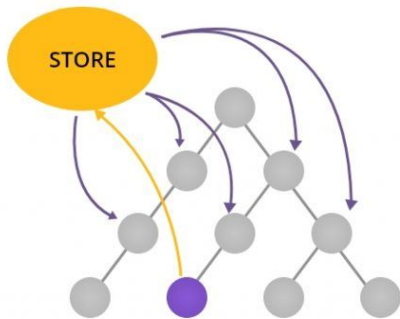


props



● Component initiating change

estado global



*Nota: Podemos usar props cuando un componente quiere **pasar props de padre a hijo directamente***



DEV.F.



Estado Global

Un estado global nos permite compartir valores y funcionalidades a través del árbol de componentes sin necesidad de usar props, además:

- Un estado global permite tener un estado donde **cualquier componente puede acceder**.
- No solo puede compartir información, sino **funcionalidades completas**.
- **No afectas a componentes** pasandole props que no necesitan.

un estado global no reemplaza las props

PROPS DRILLING



REACT CONTEXT



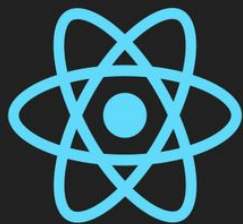


Estado Global

React Context



DEV.F.



React Context API



¿Qué es React Context?

React Context es una forma de **compartir datos globales** entre componentes sin tener que pasar props manualmente en cada nivel del árbol.

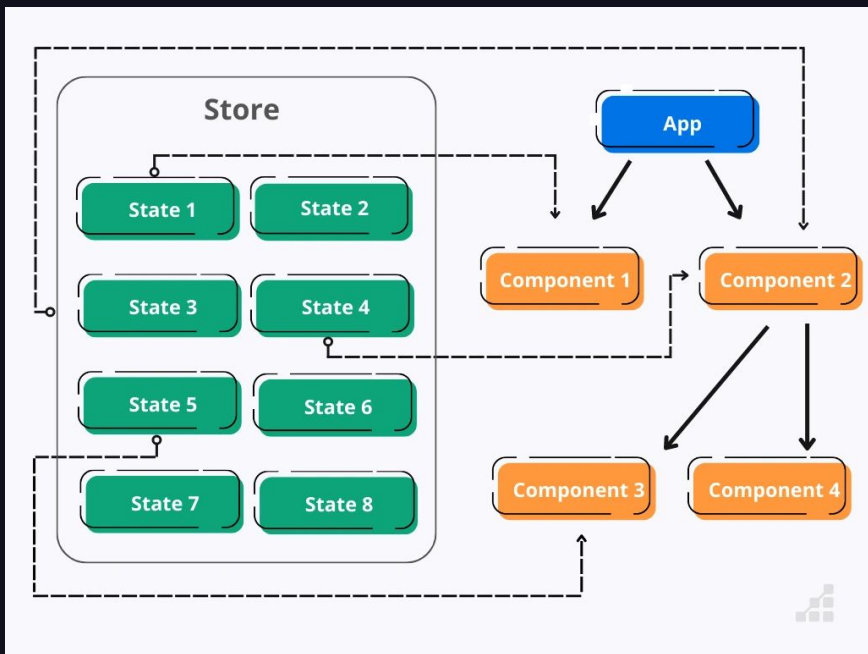
- Es como una mochila invisible que todos los componentes pueden abrir... si están dentro del contexto correcto



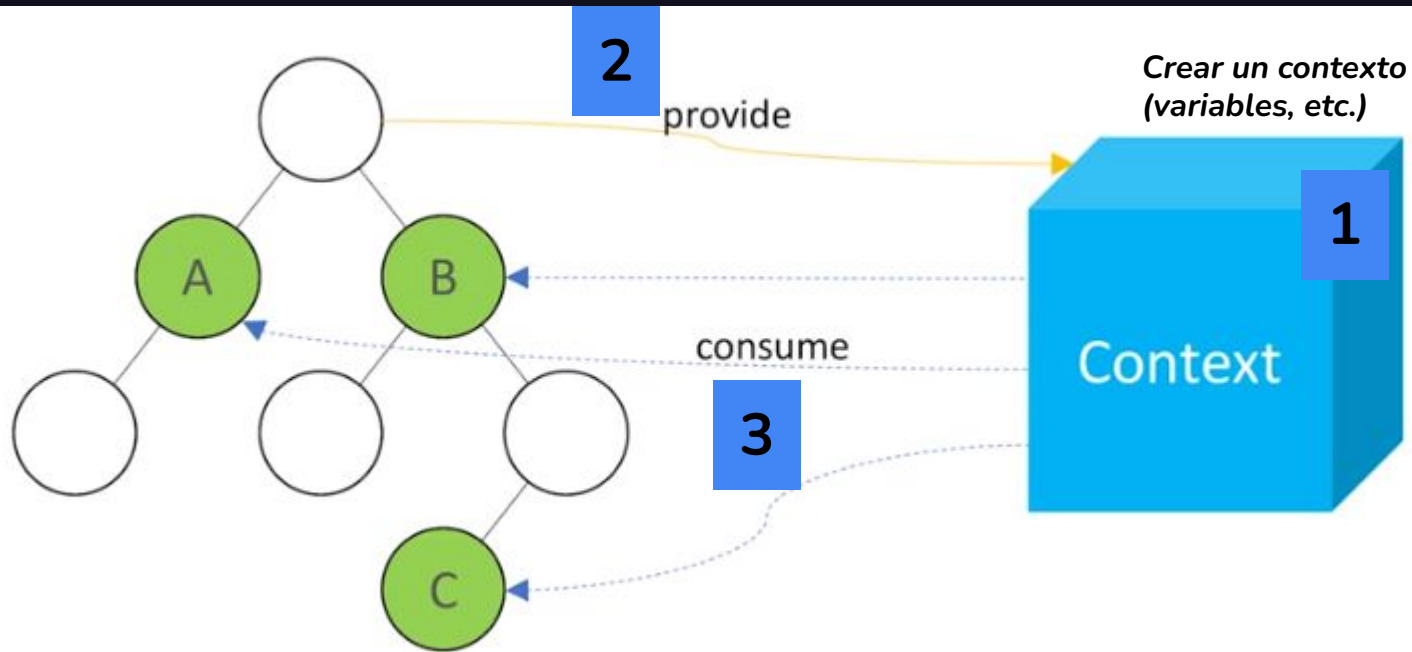


¿Cuándo usarlo?

- Autenticación (**estado del usuario logueado**).
- Tema (**dark/light mode**).
- Idioma (**internacionalización**).
- Carrito de compras.
- Config global de app.



Elementos de React Context





¿Cómo se usa?

```
import { createContext } from 'react';  
  
const ThemeContext = createContext('light');
```

1. Creando el contexto *createContext*

Para crear un contexto hay que usar la función de React ***createContext()***, qué recibe como parámetro el valor predeterminado de dicho contexto, que pueden ser valores o funciones.

Guardaremos el contexto dentro de un ***const*** para referenciarlo a futuro por dicho nombre.





¿Cómo se usa?

```
function App() {  
  const [theme, setTheme] = useState('light');  
  // ...  
  return (  
    <ThemeContext value={theme}>  
      <Page />  
    </ThemeContext>  
  );  
}
```



DEV.F.

2. Proveedor del contexto

Context.Provider

El *Context.Provider* es un componente que recibe un *prop value* que serán los valores a compartir.

Todos los componentes renderizados dentro de este componente tendrán acceso a los valores del contexto.

Es decir, provee a todos los elementos hijos del componente *Context.Provider* de acceso a la información del Contexto.



¿Cómo se usa?

```
function Button() {  
  // ☒ Recommended way  
  const theme = useContext(ThemeContext);  
  return <button className={theme} />;  
}
```

3. Consumir valores del contexto

useContext hook

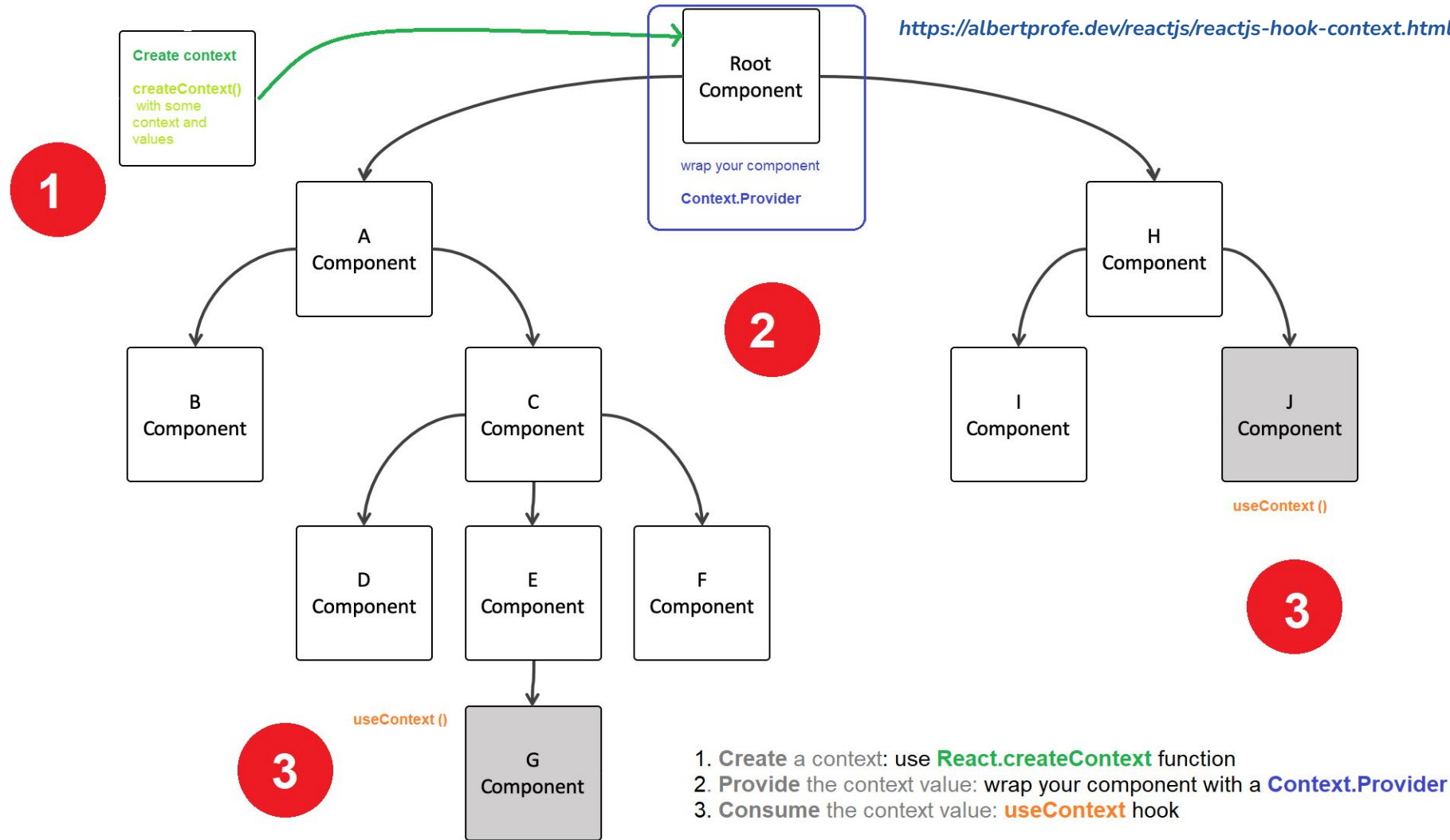
Podemos consumir los valores del contexto y almacenar los valores en una variable usando el hook *useContext*.

useContext recibe como argumento el contexto que se desea consumir.

Al guardarlo en un *const*, podemos acceder directamente a los valores del contexto indicado.



DEV.F





v19.1

Search

<https://react.dev/reference/react/createContext> 🔥

Ctrl K

Learn

Reference

react@19.1

Overview

Hooks

Components

APIs

act

cache

captureOwnerStack

createContext

lazy

memo

startTransition

API REFERENCE > APIS >

createContext

`createContext` lets you create a [context](#) that components can provide or read.

```
const SomeContext = createContext(defaultValue)
```

• Reference

- `createContext(defaultValue)`
- `SomeContext` Provider
- `SomeContext.Consumer`

• Usage

- [Creating context](#)
- [Importing and exporting context from a file](#)

• Troubleshooting



● Advertencia de sentido común



¡No abuses de Context!

Para valores que cambian muy seguido (como **inputs, sliders, etc.**), puede ser mejor usar otros estados locales o un **state manager**.. React Context **no está optimizado para updates frecuentes.**





Alternativas

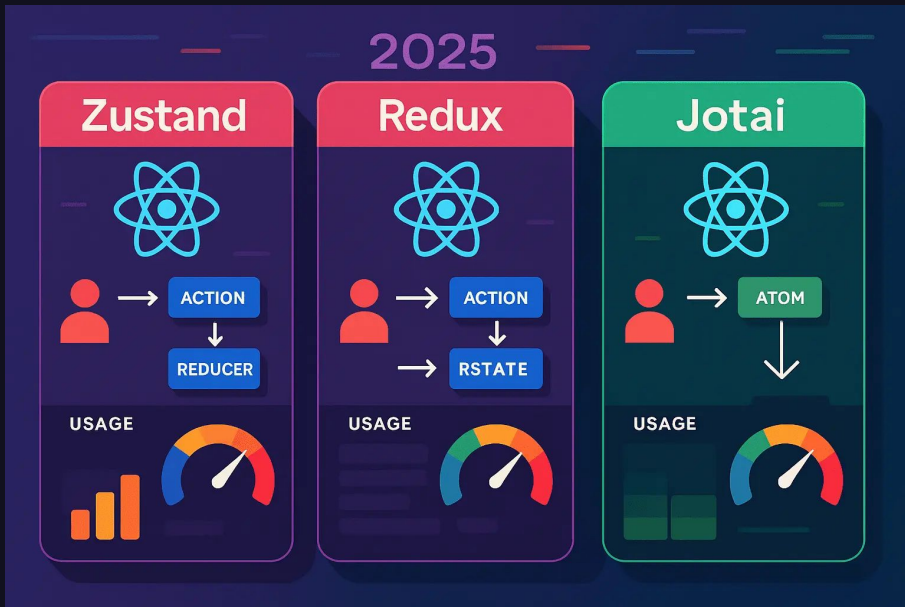
React Context



DEV.F.:



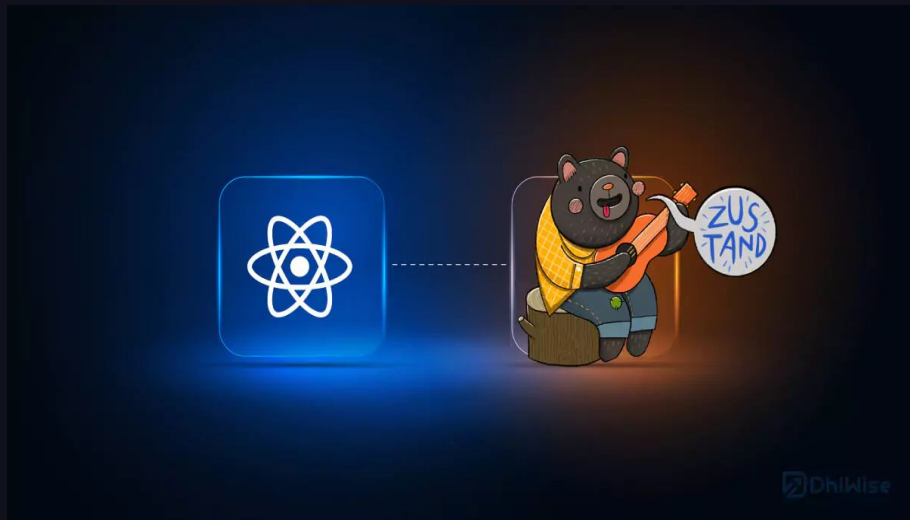
● Alternativas



En el universo de React, cuando `useState` ya no se da abasto y `context` se siente como pasar props por una madriguera de conejo... entran los **State Managers**.

Existen 3 alternativas que podrías usar: **Zustand, Redux y Jotai**.





DEV.F.



Zustand: el león simple pero poderoso

Zustand (alemán para "estado") es como **React Context pero sin complicaciones**, más rápido y sin boilerplate.



¿Por qué usar Zustand?

- Sencillo de implementar
- API minimalista
- Sin Providers
- Reactivo y liviano
- Se puede persistir y resetear fácilmente



Zustand: el león simple pero poderoso

```
// store/useUserStore.js
import { create } from 'zustand'

export const useUserStore = create((set) => ({
  user: null,
  setUser: (user) => set({ user }),
  logout: () => set({ user: null }),
}))
```



Casos de uso

- Autenticación
- Carrito de compras
- UI global (modales, temas, loaders)




```
// Componente
import { useUserStore } from '@store/useUserStore'

const Navbar = () => {
  const { user, logout } = useUserStore()

  return user ? <button onClick={logout}>Cerrar sesión</button> : <span>No logeado</span>
}
```

✅ **No hay Provider.** Se puede usar en cualquier parte de la app.



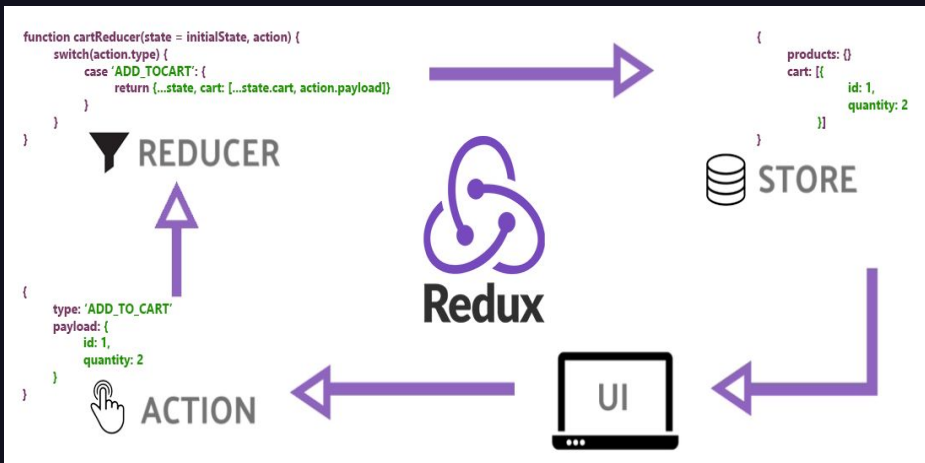


■ Redux: el veterano robusto y estructurado

Redux es un clásico. Usa un único store global, acciones y reducers. Ideal para **apps grandes y complejas**, pero puede ser un poco complejo

🧠 ¿Por qué usar Redux?

- Perfecto para proyectos grandes o en equipo
- Compatible con middlewares (thunk, saga)
- Herramientas avanzadas como Redux DevTools





■ Redux: el veterano robusto y estructurado

```
// store/userSlice.js
import { createSlice } from '@reduxjs/toolkit'

const userSlice = createSlice({
  name: 'user',
  initialState: { user: null },
  reducers: {
    setUser: (state, action) => { state.user = action.payload },
    logout: (state) => { state.user = null }
  }
})

export const { setUser, logout } = userSlice.actions
export default userSlice.reducer
```



Casos de uso

- Dashboards
- Aplicaciones con APIs complejas
- Necesidad de tiempo real o WebSockets



```
// Componente
import { useSelector, useDispatch } from 'react-redux'
import { logout } from '@store/userSlice'

const Navbar = () => {
  const user = useSelector((state) => state.user.user)
  const dispatch = useDispatch()

  return user ? <button onClick={() => dispatch(logout())}>Cerrar</button> : <span>Anonimo</span>
}
```

⚠ Es más robusto, pero también más boilerplate.





```
re like any other atom
[mode] = useAtom(darkModeAtom)
=> setDarkMode(!darkMode)
```

```
mode ? 'dark' : 'light' mode!</h1>
{!darkMode}>toggle theme</button>
```

Jotai

The ultimate React
State Management



```
import
// Co
con
cor
.value)
value {text} on
}
const innerpage = {} => {
```



DEV.F.



Jotai: el minimalista reactivo

Jotai se basa en **átomos reactivos**. Es como usar useState pero compartido. No necesitas reducers, ni context, ni configuración complicada.



¿Por qué usar Jotai?

- Super liviano y reactivo
- Cada átomo es un estado independiente
- Ideal para apps que requieren mucha composición o microestado



Jotai: el minimalista reactivo

```
// store/userAtom.js  
import { atom } from 'jotai'  
  
export const userAtom = atom(null)
```



Casos de uso

- Dashboards con widgets
- Formularios complejos
- Control de UI modular



```
// Componente
import { useAtom } from 'jotai'
import { userAtom } from '@store/userAtom'




const Navbar = () => {
  const [user, setUser] = useAtom(userAtom)

  return user ? <button onClick={() => setUser(null)}>Logout</button> : <span>Invitado</span>
}
```

✓ Sencillo, sin estructura obligada. Cada átomo es como un `useState` global.



¿Cuál elegir?

Criterio	Zustand 	Redux 	Jotai 
Boilerplate	Muy bajo	Alto	Muy bajo
Curva de aprendizaje	Fácil	Media/Alta	Fácil
Tamaño del bundle	Ligero	Pesado	Muy ligero
Escalabilidad	Alta	Muy alta	Media
Estructura	Flexible	Estricta	Súper flexible
Reactividad	Alta	Manual	Reactiva



Boilerplate

Definición:

Boilerplate es el **código repetitivo o plantilla básica** que necesitas para empezar un proyecto. No aporta lógica nueva, pero es **necesario para que el proyecto funcione**.

Ejemplo:

- El setup de Vite + React (`main.jsx`, `App.jsx`, etc.) es un boilerplate.
- En Redux, los reducers, actions, types, dispatch... mucho boilerplate.

Equivalencia:

Es como la receta de la abuela: la seguís cada vez igual, pero no cambia el sabor base.



WebSockets

Definición:

WebSockets permiten una **conexión en tiempo real bidireccional** entre el cliente y el servidor. A diferencia de HTTP, no necesitas hacer una petición cada vez que querés nuevos datos: el servidor puede enviarte datos apenas estén listos.



Ideal para:

- Chats en vivo
- Juegos en línea
- Dashboards en tiempo real
- Notificaciones push



Analógico:

Es como una radio abierta: si alguien habla, todos los oyentes lo reciben **sin tener que pedirlo**.