React context

Sistemas y Tecnologías Web (CC3062) - 2026

React Context API

Semestre 01, 2026

Propdrilling

Imaginemos una aplicación con esta estructura:

```
App
├── Header
│   └── UserAvatar
├── Sidebar
│   └── UserSettings
└── MainContent
    └── Dashboard
        └── WelcomeMessage
```

Todos necesitan saber quién está autenticado

"Solución"

La solución inmediata es pasar `user` como prop a cada componente

```
function App() {
  const user = { name: "María", role: "admin" };
```

```
  return (
    <>
      <Header user={user} />
      <Sidebar user={user} />
      <MainContent user={user} />
    </>
  );
}
```

No es escalable

Ejemplo

```
function MainContent({ user }) {
  return <Dashboard user={user} />;
}

function Dashboard({ user }) {
  return <WelcomeMessage user={user} />;
}

function WelcomeMessage({ user }) {
  return <h1>Hola, {user.name}!</h1>;
}
```

`MainContent` y `Dashboard` no usan `user`

Solo lo pasan hacia abajo — son intermediarios innecesarios

Definición

Pasar datos a través de componentes que no los necesitan

- Los datos viajan por 3, 4, 5 niveles de componentes intermedios

- Los componentes intermedios quedan "contaminados" con props que no les corresponden
- Si cambia la estructura del árbol, hay que actualizar cada nivel (frágil)
- Es difícil saber quién realmente usa el dato y quién solo lo pasa de largo

React Context

Context permite que un componente padre proporcione datos a cualquier componente del árbol debajo de él

Sin pasar props manualmente en cada nivel

```
App  ←── provee el valor
├── Header
│   └── UserAvatar  ←── recibe el valor directamente
├── Sidebar
│   └── UserSettings  ←── recibe el valor directamente
└── MainContent
    └── Dashboard
        └── WelcomeMessage  ←── recibe el valor directamente
```

CSS

Context funciona como la herencia de CSS

- En CSS: se define `color: blue` en el padre, todos los hijos lo heredan
- Con Context: se define un valor en el padre, todos los hijos pueden leerlo

React es explícito sobre qué contexto consumir

3 pasos

Paso 1 — Crear el contexto

```
// UserContext.js
import { createContext } from 'react';

export const UserContext = createContext(null);
```

- `createContext` vive fuera de cualquier componente
- El argumento es el valor por defecto (se usa si no hay Provider arriba)
- Convención: exportar el contexto para usarlo en otros archivos

Paso 2 — Proveer el contexto

```
// App.jsx
import { UserContext } from './UserContext';

function App() {
  const user = { name: "María", role: "admin" };

  return (
    <UserContext.Provider value={user}>
      <Header />
      <Sidebar />
      <MainContent />
    </UserContext.Provider>
  );
}
```

- `Provider` envuelve los componentes que necesitan el valor
- El prop `value` contiene el dato que se comparte
- Todos los componentes dentro del `Provider` pueden leer `user`

Paso 3 — Consumir el contexto

```
// WelcomeMessage.jsx
import { useContext } from 'react';
import { UserContext } from './UserContext';

function WelcomeMessage() {
  const user = useContext(UserContext);

  return <h1>Hola, {user.name}!</h1>;
}
```

- `useContext` lee el valor del contexto más cercano en el árbol
- No necesita recibir nada por props
- `MainContent` y `Dashboard` ya no tocan `user`

Buena práctica

En lugar de importar `UserContext` + `useContext` en cada componente, se encapsula en un hook:

```
// UserContext.js
import { createContext, useContext } from 'react';

export const UserContext = createContext(null);

export function useUser() {
  return useContext(UserContext);
}
```

```
// WelcomeMessage.jsx
import { useUser } from './UserContext';

function WelcomeMessage() {
  const user = useUser();  // un solo import, una sola línea
```

```
  return <h1>Hola, {user.name}!</h1>;
}
```

Si cambia la implementación del contexto, solo se actualiza un lugar

Ejemplo

Crear el contexto

```
// ThemeContext.js
import { createContext } from 'react';

export const ThemeContext = createContext('light');
```

Proveer el contexto

```
// App.jsx
import { useState } from 'react';
import { ThemeContext } from './ThemeContext';

function App() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={theme}>
      <button onClick={() => setTheme(t => t === 'light' ? 'dark' : 'light')}>
        Cambiar tema
      </button>
      <Navbar />
      <Main />
    </ThemeContext.Provider>
  );
}
```

Cuando `theme` cambia, todos los consumidores se actualizan automáticamente

Consumir el contexto

```
// Navbar.jsx
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function Navbar() {
  const theme = useContext(ThemeContext);

  return (
    <nav className={`navbar navbar--${theme}`}>
      Mi Aplicación
    </nav>
  );
}
```

```
// Button.jsx
function Button({ children }) {
  const theme = useContext(ThemeContext);

  return (
    <button className={`btn btn--${theme}`}>
      {children}
    </button>
  );
}
```

El resultado

Antes — prop drilling:

```
App → Navbar (pasa theme) → NavLinks (pasa theme) → Button ← usa theme
```

Después — Context:

```
App (provee theme)
           ↓
     Button ← lee theme directamente
```

`Navbar` y `NavLinks` ya no necesitan saber que `theme` existe

Concurrencia

Un componente puede hacer ambas cosas

Un componente puede leer el contexto del padre y proveer uno nuevo a sus hijos

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext';

function Section({ children }) {
  const level = useContext(LevelContext);  // Lee del padre

  return (
    <LevelContext.Provider value={level + 1}>  // Provee +1 a sus hijos
      <section>{children}</section>
    </LevelContext.Provider>
  );
}
```

Útil para contextos que se acumulan o transforman en cada nivel

Múltiples contextos

Pueden coexistir

```
function App() {
  return (
```

```
    <ThemeContext.Provider value="dark">
      <UserContext.Provider value={currentUser}>
        <Page />
      </UserContext.Provider>
    </ThemeContext.Provider>
  );
}
```

```
function Button() {
  const theme = useContext(ThemeContext);
  const user = useContext(UserContext);

  return (
    <button className={theme}>
      Hola, {user.name}
    </button>
  );
}
```

Cada contexto es independiente — no se interfieren entre sí

Cuándo usar Context

Casos de uso típicos

| Caso | Qué se comparte | |------|----------------| | Tema visual | `'light'` / `'dark'` | | Usuario autenticado | `{ name, role, token }` | | Idioma / locale | `'es'` / `'en'` | | Carrito de compras | `[items]` | | Estado global simple | Reducer + Context |

Si muchos componentes en distintos niveles necesitan el mismo dato, Context es la herramienta correcta

Cuándo NO usar Context

- Si son 1 o 2 niveles, las props siguen siendo la mejor opción