## PARTE III — GESTION DE ESTADO GLOBAL

### El problema del estado compartido

Hasta ahora hemos usado `useState` dentro de un componente.

Esto funciona bien si el estado pertenece a un componente y a sus hijos inmediatos.

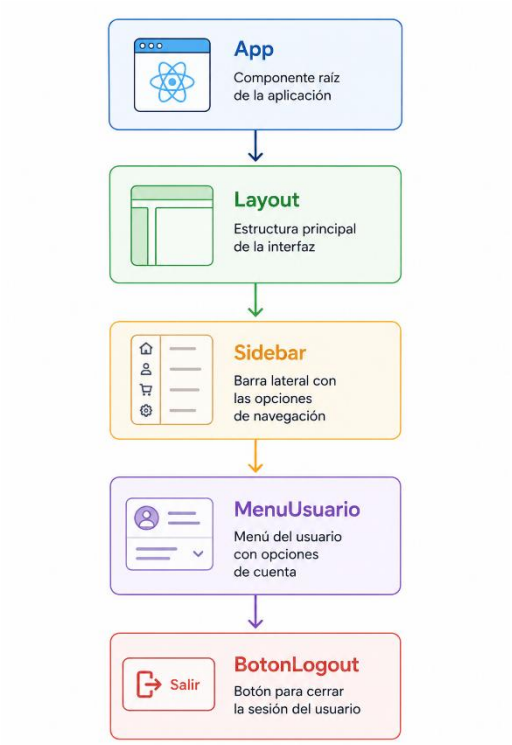
Pero en aplicaciones más grandes pueden aparecer diversos problemas. Suponga que tiene una aplicación con la siguiente estructura

```
App
├── Navbar
├── Productos
├── Carrito
├── Perfil
└── Footer
```

Donde varias partes necesitan conocer:

- usuario autenticado;
- rol del usuario;
- productos del carrito;
- tema claro/oscuro;
- cantidad de notificaciones.

Tal vez podría pensar que una solución posible es pasar **props** manualmente, pero si genera muchos niveles para pasarlos puede generar **prop drilling**. Con esta solución, por ejemplo, si necesita pasar el usuario desde la App hacia un componente denominado BotonLogout para poder desconectarse de la aplicación tal vez necesite enviar props por muchos componentes intermedios, como lo representa la siguiente figura:



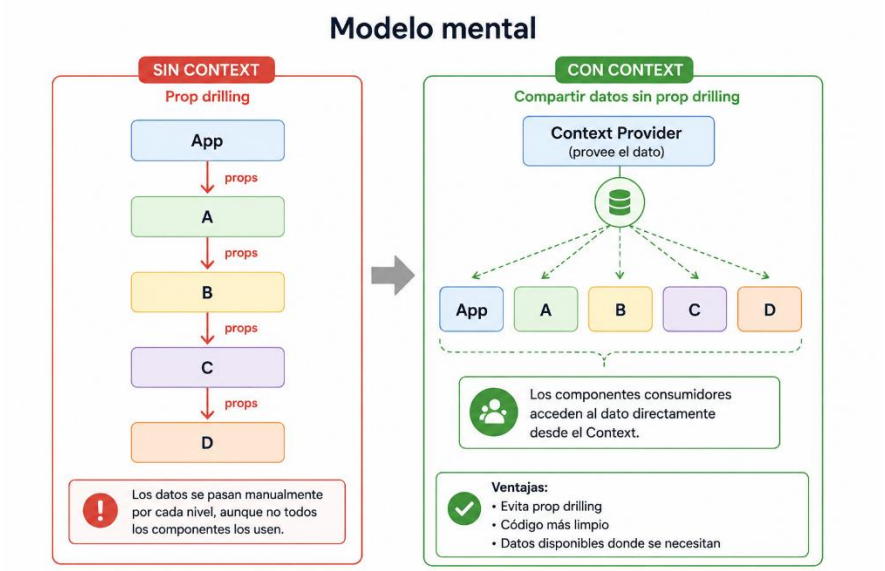
Esto constituye un mal diseño.

El prop drilling ocurre cuando pasamos datos por componentes que no los usan, solo para que lleguen más abajo.

## CONTEXT API

Context API permite compartir datos con múltiples componentes sin pasar props manualmente por cada nivel. La documentación clásica de React la presenta como un mecanismo para pasar datos a través del árbol de componentes sin pasar props manualmente en cada nivel.

La documentación oficial de React también explica que Context permite pasar información profundamente a otros componentes, y que puede combinarse con `reducers` para manejar estado más complejo. La siguiente figura representa el enfoque mental del funcionamiento de Context API como respuesta a la aplicación de props para pasar información entre componentes:




### Implementación en un uso clásico: La autenticación

Suponga que tiene la siguiente estructura de aplicación en React:

```
App
├── Navbar
├── Login
├── Admin
└── Footer
```

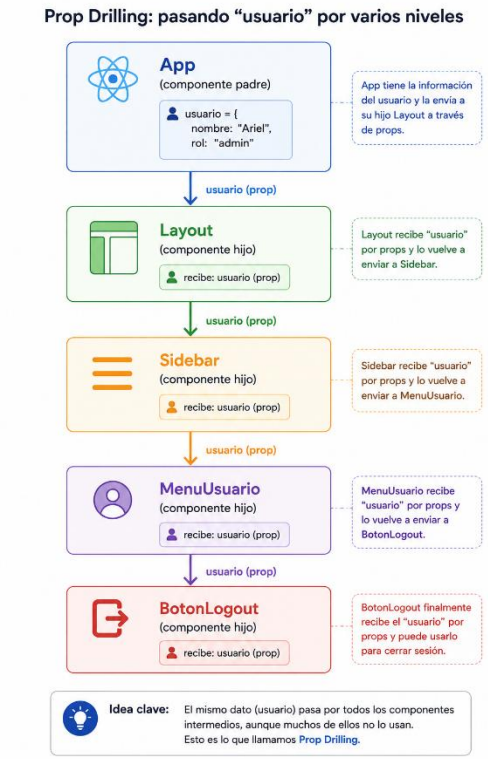
Imagine que un usuario inicia una sesión con el nombre de usuario “Ariel” y el rol “ADMIN”. La pregunta ahora es ¿Cómo lograríamos que cualquier componente conozca cual es el usuario actual?

Por ejemplo, es muy común que:

- Navbar necesite mostrar “Bienvenido Ariel”
- Admin necesite verificar constamente si el usuario actual es el administrador
- BotonLogout necesite ejecutar un logout() con la información del usuario (por ejemplo saber si está logueado o no)

Sin Context API podríamos intentar establecer una jerarquía que permita pasar props. Por ejemplo:

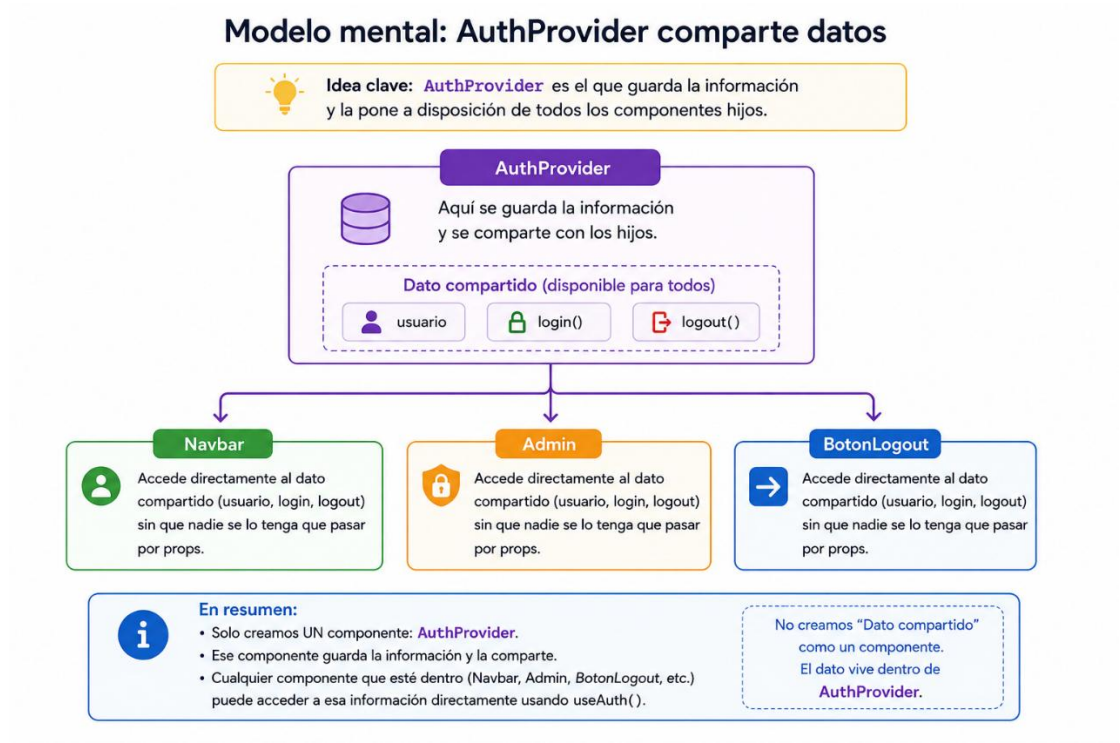

Pero ahora supongamos que nuestro `NavBar` tiene una navegación más profunda, es decir está conformado por más componentes interrelacionados en una jerarquía de tal forma que pasar la información requiera involucrar más componentes, por ejemplo:



Entonces aparece el problema del prop drilling, porque:

- Layout
- Sidebar
- MenuUsuario

no necesitan usar usuario y solo lo reciben para volver a enviarlo. Context API intenta resolver este problema, mediante el siguiente modelo conceptual:




La idea central es crear un contexto que funcione como una fuente compartida de información para todos los componentes que estén dentro de un proveedor. Tendremos entonces esta estructura:

```
src/
├── context/
│   └── AuthContext.jsx
│
├── components/
│   ├── Layout.jsx
│   ├── Sidebar.jsx
│   ├── MenuUsuario.jsx
│   ├── BotonLogout.jsx
│   ├── Navbar.jsx
│   └── RutaProtegida.jsx
│
├── pages/
│   ├── Login.jsx
│   └── Admin.jsx
│
├── App.jsx
└── main.jsx
```

## Paso 1 — Crear el archivo AuthContext.jsx

Creamos la carpeta `src/context/`

y dentro de ella el archivo `AuthContext.jsx`

El código será:

```
import { createContext, useContext, useState } from "react";

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [usuario, setUsuario] = useState(null);

  const login = () => {
    setUsuario({
      nombre: "Ariel",
      rol: "ADMIN"
    });
  };

  const logout = () => {
    setUsuario(null);
  };

  return (
    <AuthContext.Provider value={{ usuario, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => {
  return useContext(AuthContext);
};
```

Puedes observar que se importan tres herramientas de React:

- `createContext`
- `useContext`
- `useState`

La línea

```
const AuthContext = createContext();
```

Crea el contexto. Pero hay que ser precisos:

- createContext() no crea el usuario.
- createContext() no inicia sesión.
- createContext() no guarda datos por sí solo.

Lo que hace es crear un canal de comunicación. Ese canal permitirá que ciertos componentes accedan a una información compartida.

Con la línea

```
export const AuthProvider = ({ children }) => {
```

Se crea un componente llamado `AuthProvider`. Este componente será el encargado de:

1. Crear el estado de autenticación.
2. Definir funciones para modificar ese estado.
3. Compartir ese estado y esas funciones con sus componentes hijos.

Dentro de este componente definimos una variable de estado:

```
const [usuario, setUsuario] = useState(null);
```

Aquí aparece el estado real que posteriormente será compartido con los componentes hijos. Inicialmente:

```
usuario = null
```

Esto significa que no hay usuario autenticado.

Cuando alguien inicia sesión, usuario dejará de ser `null` y pasará a contener información.

Vamos a simular el inicio de sesión mediante la función `login()`:

```
const login = () => {
  setUsuario({
    nombre: "Ariel",
    rol: "ADMIN"
  });
};
```

Cuando se ejecuta `login()`, el estado cambia de null a un objeto que contiene el nombre “Ariel ” y el rol “ADMIN”. Por lo tanto, cualquier componente que consuma este contexto podrá saber que el usuario actual es Ariel y que tiene rol ADMIN.

La función `logout()` cierra la sesión. Cuando se ejecuta el estado vuelve a `null`. De esta manera, se indica que no hay ningún usuario autenticado.

```
const logout = () => {
  setUsuario(null);
};
```

El Provider

La siguiente línea del componente es la parte central del patrón de comportamiento:

```
return (
  <AuthContext.Provider value={{ usuario, login, logout }}>
    {children}
  </AuthContext.Provider>
);
```

El componente `<AuthContext.Provider>` es el proveedor de información.

La propiedad `value={{ usuario, login, logout }}` define qué información estará disponible para los componentes hijos.

En este caso comparte:

- Usuario: la variable de estado
- login(): función de autenticación
- logout(): función para cerrar la sesión

¿Qué significa `{children}`?

`children` representa todo lo que esté dentro de `AuthProvider`.

Por ejemplo, si en `main.jsx` escribimos:

```
<AuthProvider>
  <App />
</AuthProvider>
```

Entonces `children = <App />`

Pero como `App` contiene otros componentes, todo el árbol queda envuelto:

```
AuthProvider
└─ App
   ├─ Navbar
   ├─ Layout
   │  └─ Sidebar
   │     └─ MenuUsuario
   │        └─ BotonLogout
   └─ Routes
      ├─ Login
      └─ Admin
```

Esto significa que todos esos componentes pueden acceder al contexto.

El hook useAuth

Al final del componente está presente el siguiente código:

```
export const useAuth = () => {
  return useContext(AuthContext);
};
```

Esta función crea un hook personalizado. En lugar de escribir en cada componente:

`useContext(AuthContext)`

escribiremos:

`useAuth()`

Esto hace que el código sea más claro y expresivo.

Por ejemplo, la siguiente línea:

```
const { usuario, logout } = useAuth();
```

Se debe interpretar como que se desea usar la información de autenticación. ¿Por qué?

Recuerde que:

```
<AuthContext.Provider
value={{
  usuario,
  login,
  logout
}}
>
```

Indica qué es lo que se va a compartir con los componentes hijos; por lo tanto, `useAuth()` devuelve realmente esto:

```
{
  usuario,
  login,
  logout
}
```

Es decir, devuelve un objeto estructurado de la siguiente manera:

```
{
  usuario: {
    nombre: "Ariel",
    rol: "ADMIN"
  },

  login: [función],

  logout: [función]
}
```

O sea, un objeto usuario y las funciones login() y logout(). Por lo tanto:

```
const { usuario, logout } = useAuth();
```

aplica la desestructuración de objetos. Esto significa que estos códigos son equivalentes:

| | |
|---|---|
| `const { usuario, logout } = useAuth();` | `const auth = useAuth();`<br><br>`const usuario = auth.usuario;`<br>`const logout = auth.logout;` |

Al ejecutar `useAuth()` se obtiene el objeto compartido por `AuthProvider`, luego se extrae `usuario` y `logout` y los guarda en una variable local.

Este uso quedará más claro cuando sea consumido por los componentes, que lo veremos en los siguientes pasos.

### Paso 2 — Envolver la aplicación con AuthProvider

Ahora debemos asegurarnos de que toda la aplicación quede dentro del proveedor. Tal como se anticipo en el paso anterior, deberíamos hacer esto:

main.jsx

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { AuthProvider } from "./context/AuthContext.jsx";
import App from "./App.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <AuthProvider>
      <App />
    </AuthProvider>
  </StrictMode>
);
```

Con esto no necesitaremos pasar manualmente usuario, login y logout manualmente desde App mediante props a cada componente.

### Paso 3 — Consumir el contexto desde NavBar

Ahora sí, tenemos un ejemplo de como consumir desde un componente hijo, en este caso de App. Observemos a NavBar.jsx

```
import { Link } from "react-router";
import { useAuth } from "../context/AuthContext";

const Navbar = () => {
  const { usuario } = useAuth();

  return (
    <nav>
      <Link to="/">Inicio</Link>
      <Link to="/admin">Admin</Link>

      {usuario ? (
        <span>Bienvenido {usuario.nombre}</span>
      ) : (
        <span>No autenticado</span>
      )}
    </nav>
  );
};

export default Navbar;
```

Mediante

```
const { usuario } = useAuth();
```

`Navbar` no recibe `usuario` por props. `Navbar` accede directamente al contexto.

Esto puede interpretarse así:

`Navbar` pregunta al `AuthProvider`, ¿cuál es el usuario actual? Si existe `usuario`, muestra:

*Bienvenido Ariel*

Ya que renderiza una etiqueta de párrafo que referencia a la variable local que tiene alojado usuario después de aplicar la desestructuración.

Si no existe usuario, muestra:

*No autenticado*

### Paso 4 — Consumir el contexto desde BotonLogout

De manera similar, en BotonLogout.jsx tenemos:

```
import { useAuth } from "../context/AuthContext";

const BotonLogout = () => {
  const { usuario, logout } = useAuth();

  if (!usuario) return null;

  return (
    <button onClick={logout}>
      Cerrar sesión
    </button>
  );
};

export default BotonLogout;
```

Mediante

```
const { usuario, logout } = useAuth();
```

El botón accede a:

- usuario
- logout()

Si no hay usuario el botón no se muestra. Esto lo realiza:

```
if (!usuario) return null;
```

Si hay usuario, se muestra el botón y habilita el acceso a la función `logout()`. Cuando el usuario hace clic:

```
onClick={logout}
```

se ejecuta la función que limpia el estado:

```
usuario = null
```

### Paso 5 — Consumir el contexto desde RutaProtegida

De manera similar, en RutaProtegida.jsx tenemos:

```
import { Navigate } from "react-router";
import { useAuth } from "../context/AuthContext";

const RutaProtegida = ({ children }) => {
  const { usuario } = useAuth();

  if (!usuario) {
    return <Navigate to="/login" />;
  }

  return children;
};

export default RutaProtegida;
```

¿Qué sucede aquí? `RutaProtegida` pregunta ¿Existe usuario?

Si no existe:

```
return <Navigate to="/login" />;
```

redirige a la página de `login`.

Si existe:

```
return children;
```

muestra el contenido protegido.

¿Cuál es el contenido protegido al que hace referencia `children`?

Recordemos que en main.jsx

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { AuthProvider } from "./context/AuthContext.jsx";
import App from "./App.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <AuthProvider>
      <App />
    </AuthProvider>
  </StrictMode>
);
```

Se hace referencia a App. Veamos que contiene App:

```
<Route
  path="/admin"
  element={
    <RutaProtegida>
      <Admin />
    </RutaProtegida>
  }
/>
```

Por lo tanto, el recurso protegido es <Admin> debido a que está contenido dentro de RutaProtegida.

Por lo tanto, en el ejemplo

```
children = <Admin />
```

Entonces, cuando RutaProtegida ejecuta:

```
return children;
```

y realmente está devolviendo:

```
return <Admin />;
```

Es decir, en caso de existir el usuario se muestra la página de administración.

Expresar el elemento protegido mediante `children` permite extender la protección a otros elementos diferentes, porque podría ser:

- <Perfil />
- <Carrito />
- <PanelDocente />
- <Configuracion />

En síntesis, Context API permite centralizar información compartida, como el usuario autenticado, y acceder a ella desde distintos componentes sin pasar props por toda la jerarquía. En este ejemplo, AuthProvider guarda y comparte usuario, login() y logout(); useAuth permite consumir esa información; y RutaProtegida utiliza el estado global para decidir si mostrar Admin o redirigir al login.