Purwadhika
Digital Technology School

Full Stack AI Software Development

Application Data Flow and State Management

Job Connector Program

Outline

Introduction to Application Data Flow

Understanding how data moves in a React application through **unidirectional data flow**, and why managing state properly is essential for building scalable and maintainable apps.

Global State Management

Exploring strategies for sharing data across multiple components using **React Context** and understanding the problem of **prop drilling**.

Local State Management

Learning how to manage component-specific data using useState and when to use each approach effectively.

Zustand State Management

Implementing a lightweight global store using **Zustand** — creating, updating, and consuming global state efficiently with simple code.

State Management

What is State?

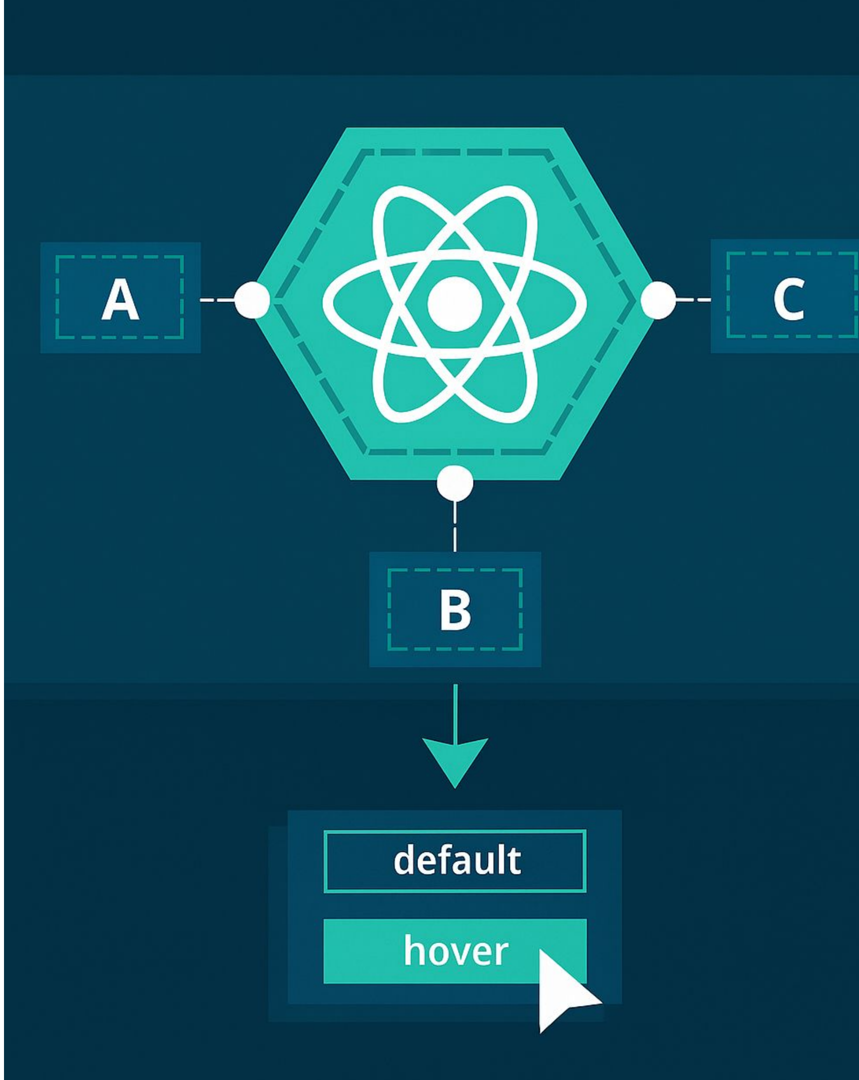
State represents **dynamic** data that affects what you see on the UI. When state changes, the UI updates.

Why State Matters

- Makes UI interactive
- Controls component behavior
- Stores temporary data (ex: form input)

Common State Examples

- Login status
- Theme mode (light/dark)
- Shopping cart items
- Input text values



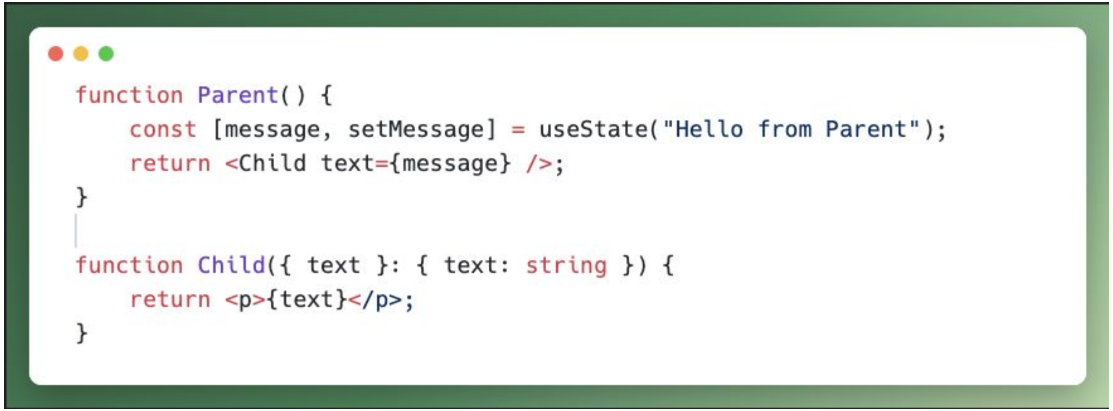
React Unidirectional Data Flow

React follows a **One-way data flow** mechanism:

- **Parent → Child** using Props
- **Child → Parent** using callback functions

Key Concepts

- **State**: **Internal data** owned by the component
- **Props**: **External data** passed to a component



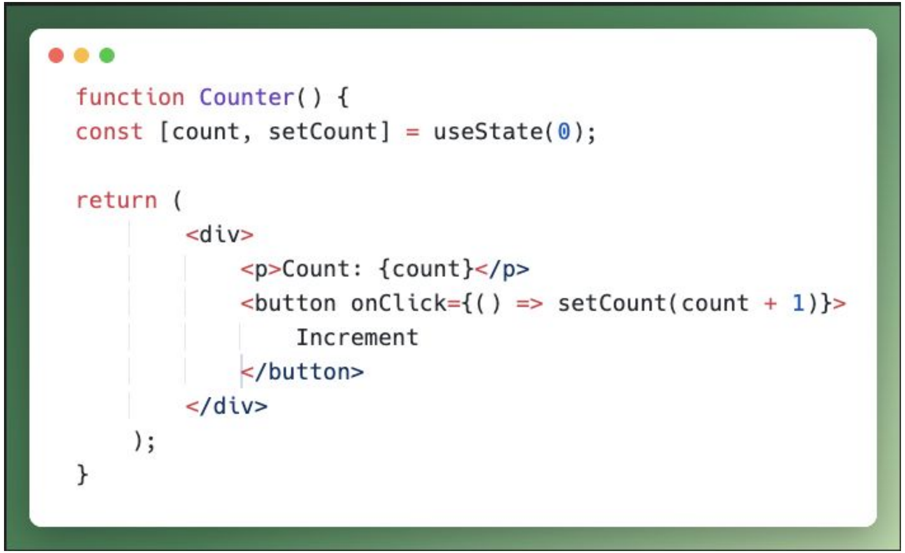
```
function Parent() {
    const [message, setMessage] = useState("Hello from Parent");
    return <Child text={message} />;
}

function Child({ text }: { text: string }) {
    return <p>{text}</p>;
}
```

Local State

Local state exists **inside a component only.** And can’t use at other component.

When To Use Local State

- UI toggle (modal, dropdown)
- Form values
- Simple component-level behavior



```
function Counter() {
const [count, setCount] = useState(0);

return (
      <div>
         <p>Count: {count}</p>
         <button onClick={() => setCount(count + 1)}>
            Increment
         </button>
      </div>
   );
}
```

Problem Example — Prop Drilling

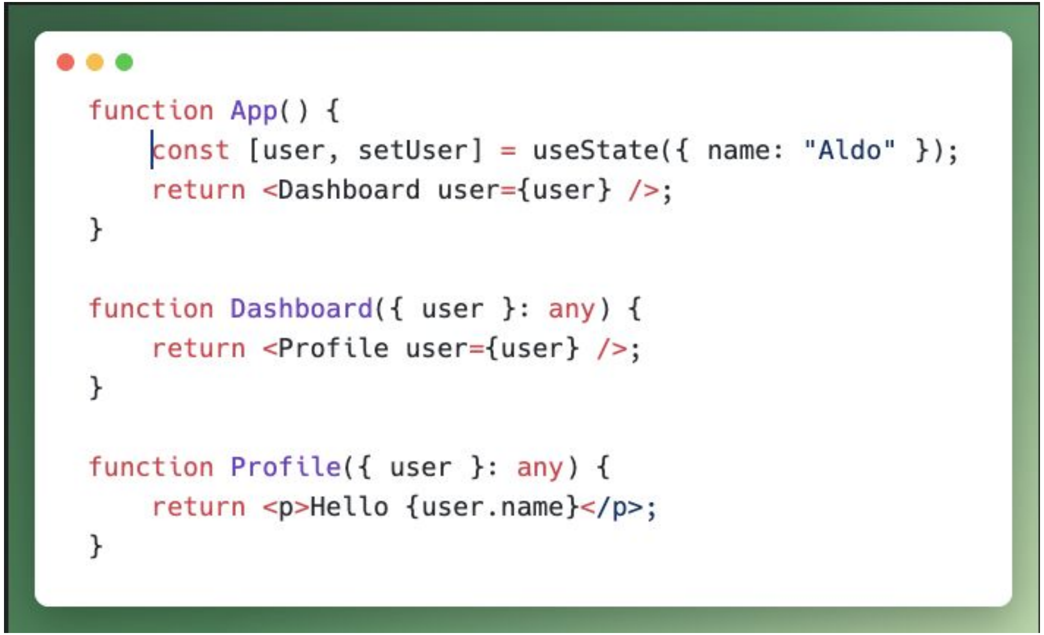
```
App
└─── Dashboard (needs user)
        └─── Profile (needs user)
                └─── ProfileHeader (uses user)
```

```
App
│
│——----passes user ➜ Dashboard
│          └─── passes user ➜ Profile
│                  └─── passes user ➜ ProfileHeader
```

Issue: Each component must receive **user** even if it doesn't use it.

Problem Example — Prop Drilling

When passing props too deep through components.



```
function App() {
    const [user, setUser] = useState({ name: "Aldo" });
    return <Dashboard user={user} />;
}

function Dashboard({ user }: any) {
    return <Profile user={user} />;
}

function Profile({ user }: any) {
    return <p>Hello {user.name}</p>;
}
```

This becomes messy — we need global state.

What Is Global State?

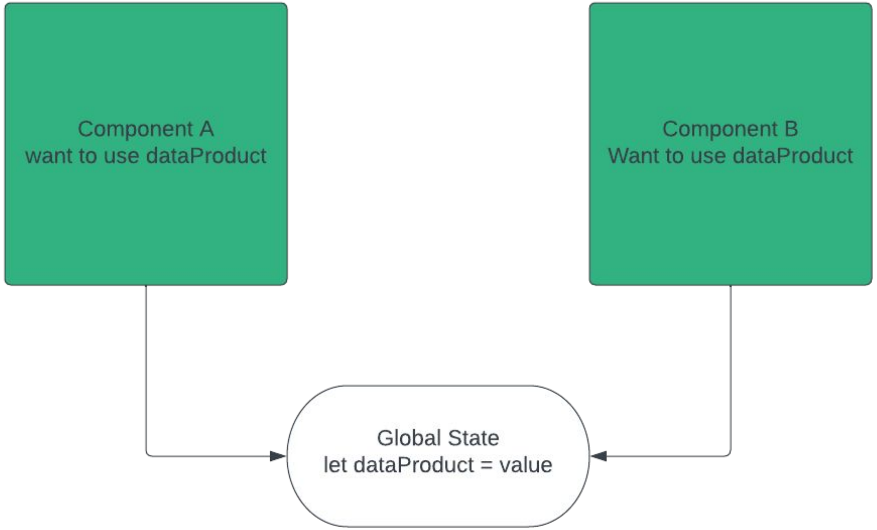
Global state is shared across multiple components.

When Do You Need Global State?

- Authentication status
- User profile
- Shopping cart
- Theme settings

Local state → Single component

Global state → Many components need the same data

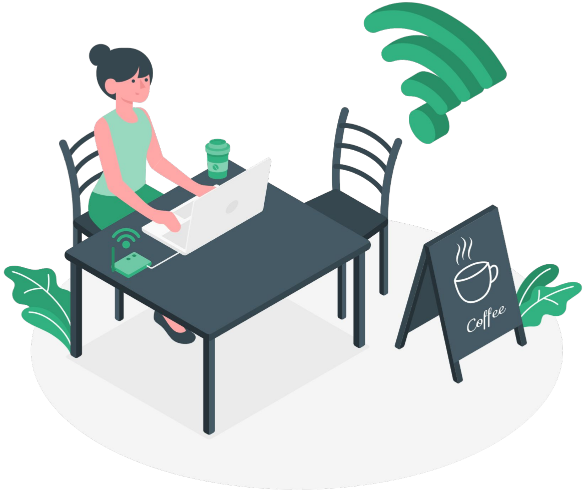


How to Handle Global State in React

To manage **global state** (data that’s shared between different parts of your app), there are a few ways you can do it:

- **Browser Features** – like **Local Storage, Session Storage, or Cookies.**
  These let you save data directly in the browser so it can be used across different pages, or even after refreshing the app.
- **React Hooks** – using **useContext** (sometimes with useState or useReducer).
  This is a simple built-in way to share data between components without needing extra libraries.
- **State Management Libraries** – such as **Zustand** or **Redux**.
  These tools help manage larger or more complex state in big applications, making your code more organized.

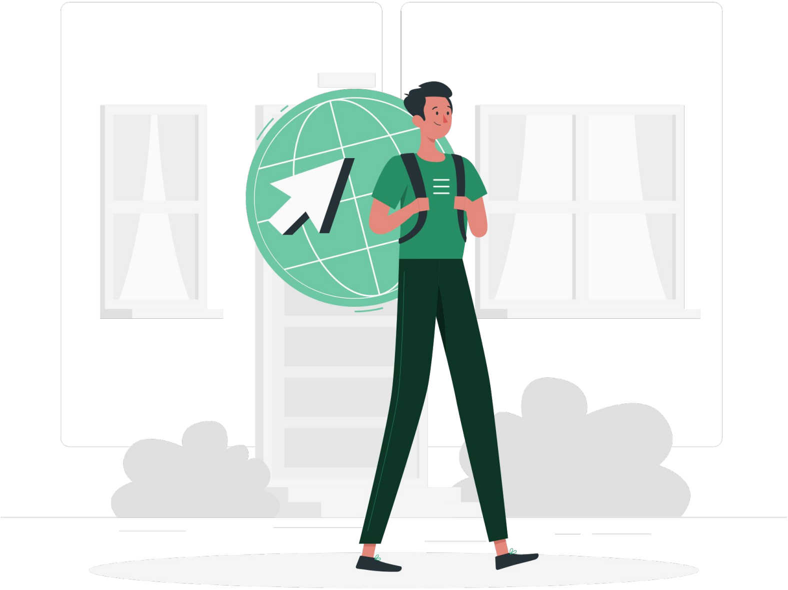
Let’s see how each method works and when to use them!



Local Storage

This read-only interface property provides access to the Document's local storage object, the stored data is stored across browser sessions.

The data stored in LocalStorage is specific to a protocol in the document. If the site is loaded over HTTP (e.g., http://example.com), localStorage returns a different object than if it is loaded over HTTPS (e.g., https://abc.com).



Local Storage

Local storage has 4 methods:

- **setItem() Method –** This method takes two parameters one is key and another one is value. It is used to store the value in a particular location with the name of the key. Syntax: localStorage.setItem(key, value)
- **getItem() Method –** This method takes one parameter that is key which is used to get the value stored with a particular key name. Syntax: localStorage.getItem(key)
- **removeItem() Method –** This is method is used to remove the value stored in the memory in reference to key. Syntax: localStorage.removeItem(key)
- **clear() Method –** This method is used to clear all the values stored in localstorage. Syntax: localStorage.clear()

Session Storage

Session Storage objects can be accessed using the sessionStorage read-only property. The difference between sessionStorage and localStorage is that localStorage data does not expire, whereas sessionStorage data is cleared when the page session ends.

Session Storage

Session Storage has 4 methods:

- **setItem() Method –** This method takes two parameters one is key and another one is value. It is used to store the value in a particular location with the name of the key. Syntax: sessionStorage.setItem(key, value)
- **getItem() Method –** This method takes one parameter that is key which is used to get the value stored with a particular key name. Syntax: sessionStorage.getItem(key)
- **removeItem() Method –** This is method is used to remove the value stored in the memory in reference to key. Syntax: sessionStorage.removeItem(key)
- **clear() Method –** This method is used to clear all the values stored in the session storage. Syntax: sessionStorage.clear()

Cookies

A **cookie** is a small piece of text data that a website saves in your browser. It helps the website **remember who you are** and **store your preferences** — for example, keeping you logged in, saving your theme choice (dark or light mode), or remembering items in your cart.

When you visit a website, your browser sends a **request** to the server asking for a web page. Normally, each request is treated as new — the server doesn't "remember" you.
But with cookies, the website can recognize your browser and respond with personalized content, because it has stored small bits of information locally.

Cookies let websites remember information about you to improve your experience.

You can manage cookies in React using these popular libraries:

- **js-cookie** – simple and lightweight, perfect for basic cookie operations (get, set, remove).
- **react-cookie** – integrates cookies with React components and hooks, great for stateful apps.

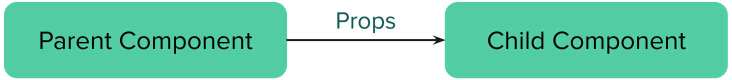
Difference Between Local Storage, Session Storage, And Cookies

Local Storage	Session Storage	Cookies
The storage capacity of local storage is 5MB/10MB	The storage capacity of session storage is 5MB	The storage capacity of Cookies is 4KB
As it is not session-based, it must be deleted via javascript or manually	It’s session-based and works per window or tab. This means that data is stored only for the duration of a session, i.e., until the browser (or tab) is closed	Cookies expire based on the setting and working per tab and window
The client can only read local storage	The client can only read session storage	Both clients and servers can read and write the cookies

useContext

Why do we use useContext?

- Normally, to send data from a **Parent Component** to **Child Components**, we use props.



Parent Component

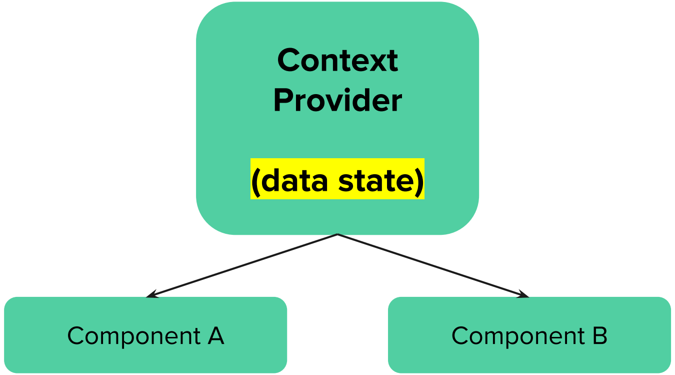
Props

Child Component

- When a component tree becomes deeper, props must be passed through multiple components even if those components don't use the data.
  ➜ This is called props drilling.

Think of it like this:

- Props = handing data manually from one component to another.
- Context = placing data in a shared storage so any component can take it when needed.



Context Provider

(data state)

Component A

Component B

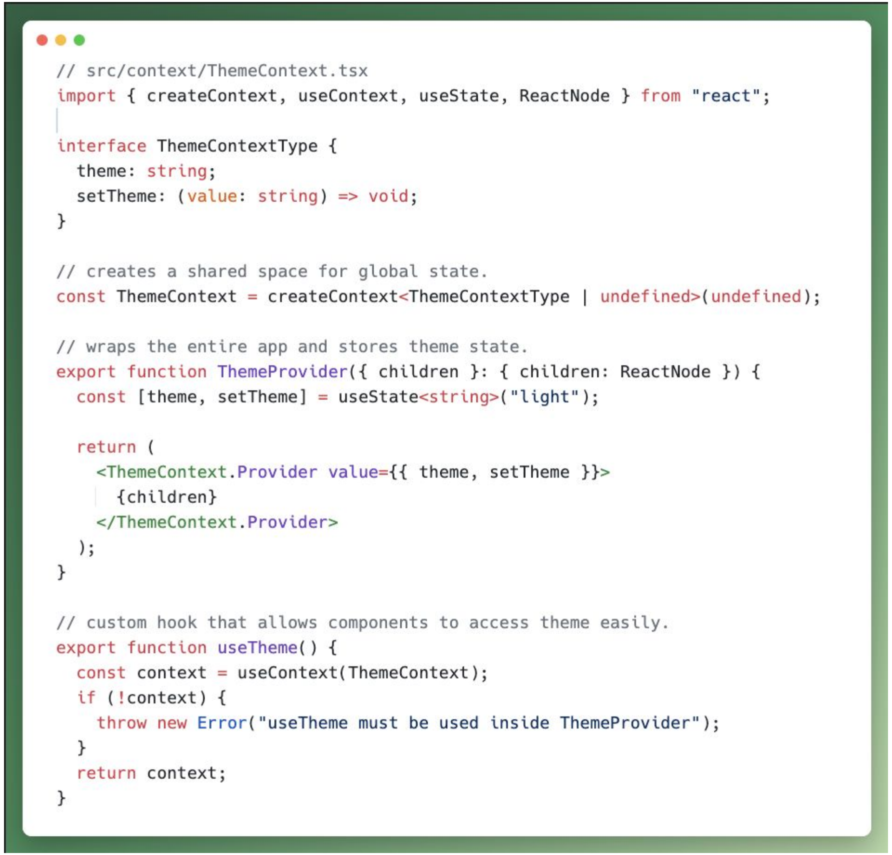
useContext solves that problem

Instead of passing data through props, we store the data in a Context Provider, and any component can access it directly using useContext.

useContext Implementation

We need preparation to implement useContext. Create file context to write configuration:

```
src/
│
├── context/
│   └── ThemeContext.tsx      <-- create this
│
├── pages/
│   ├── Home.tsx
│   └── About.tsx
│
├── App.tsx
└── main.tsx
```

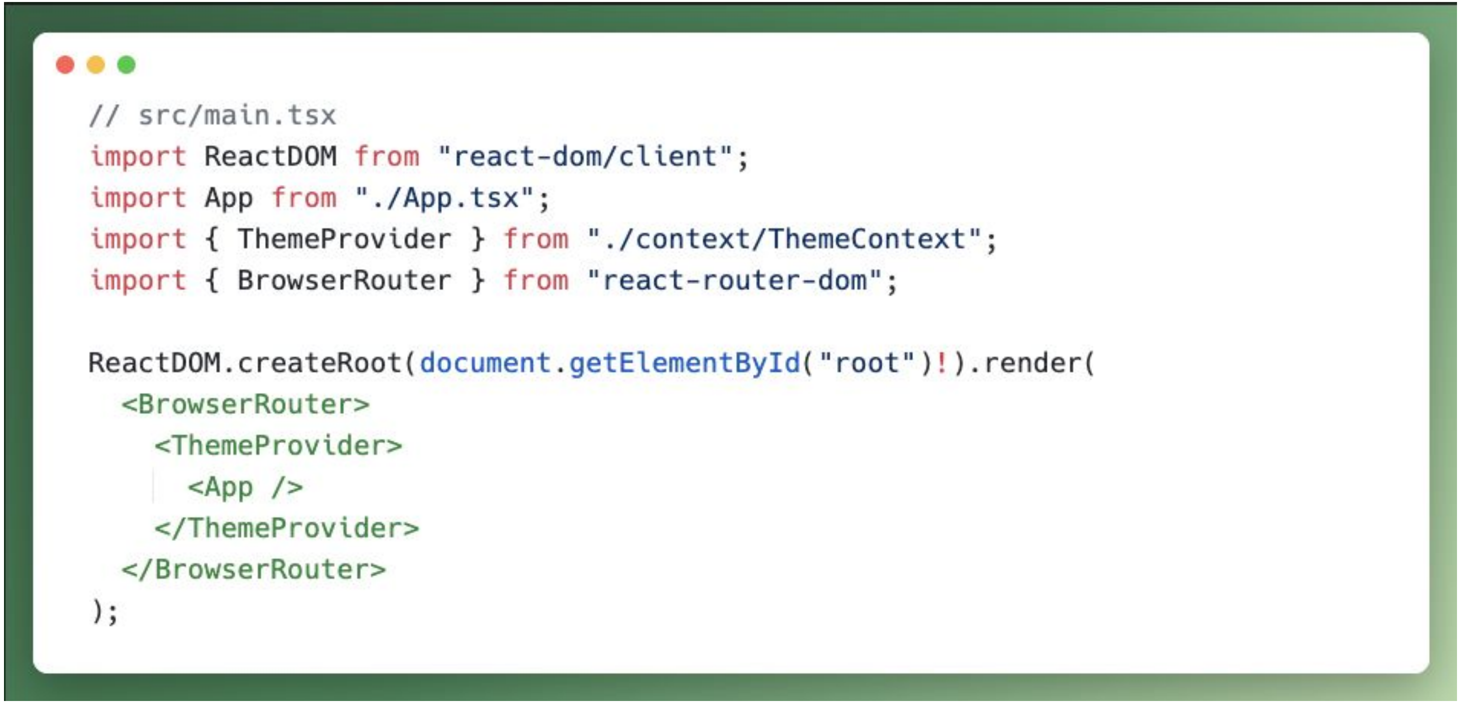


```tsx
// src/context/ThemeContext.tsx
import { createContext, useContext, useState, ReactNode } from "react";

interface ThemeContextType {
  theme: string;
  setTheme: (value: string) => void;
}

// creates a shared space for global state.
const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

// wraps the entire app and stores theme state.
export function ThemeProvider({ children }: { children: ReactNode }) {
  const [theme, setTheme] = useState<string>("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// custom hook that allows components to access theme easily.
export function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error("useTheme must be used inside ThemeProvider");
  }
  return context;
}
```

useContext Implementation

And after that, we can wrap entire application inside **ThemeProvider** and enable router.

- **ThemeProvider** supplies global theme to every component.
- **BrowserRouter** enables page routing.

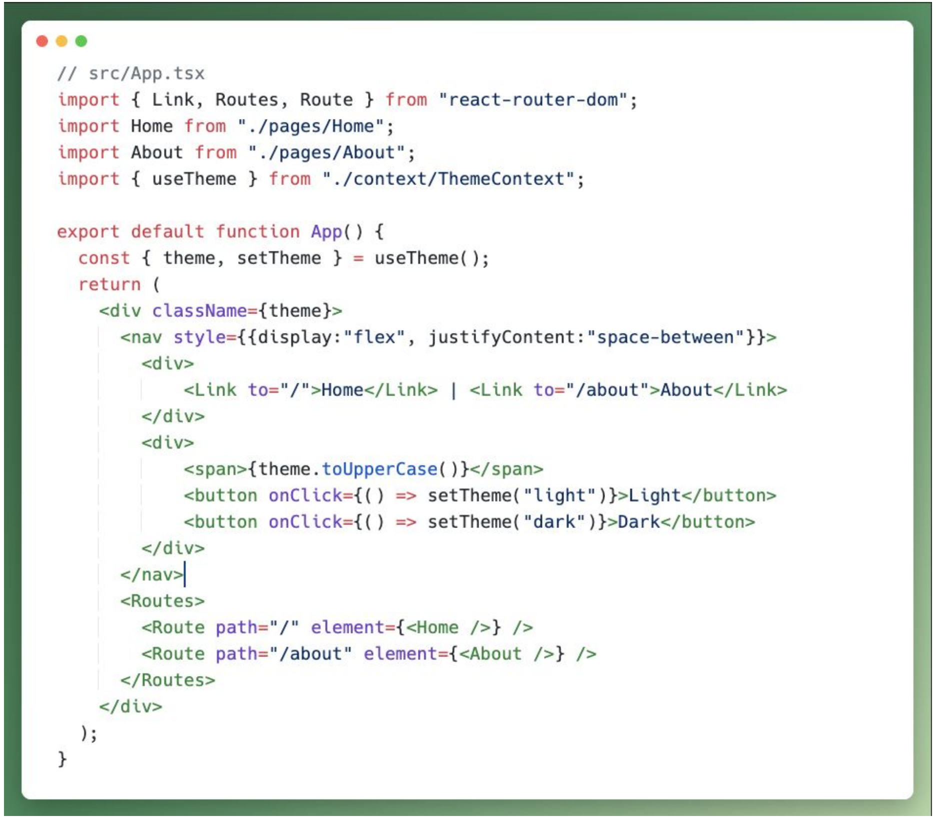


```
// src/main.tsx
import ReactDOM from "react-dom/client";
import App from "./App.tsx";
import { ThemeProvider } from "./context/ThemeContext";
import { BrowserRouter } from "react-router-dom";

ReactDOM.createRoot(document.getElementById("root")!).render(
  <BrowserRouter>
    <ThemeProvider>
      <App />
    </ThemeProvider>
  </BrowserRouter>
);
```

useContext Implementation

Uses the global theme and renders routes.

- **useTheme() ➝** consumes theme and update function.
- **theme** automatically changes CSS **class on <div>**.
- Buttons switch theme globally (without props).



```
// src/App.tsx
import { Link, Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";
import { useTheme } from "./context/ThemeContext";

export default function App() {
  const { theme, setTheme } = useTheme();
  return (
    <div className={theme}>
      <nav style={{display:"flex", justifyContent:"space-between"}}>
        <div>
            <Link to="/">Home</Link> | <Link to="/about">About</Link>
        </div>
        <div>
            <span>{theme.toUpperCase()}</span>
            <button onClick={() => setTheme("light")}>Light</button>
            <button onClick={() => setTheme("dark")}>Dark</button>
        </div>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  );
}
```

What is Zustand?

- A small, fast, and scalable state-management tool for React.
- No boilerplate, no context provider required.
- Ideal for global state in React / Next.js applications.

Creating a Global Store with Zustand

Import create from Zustand
We import the create function — this is the core API from Zustand that allows us to build a simple, lightweight state management store.

Define the Store Type (CounterState)
The **CounterState** type defines what the store looks like.

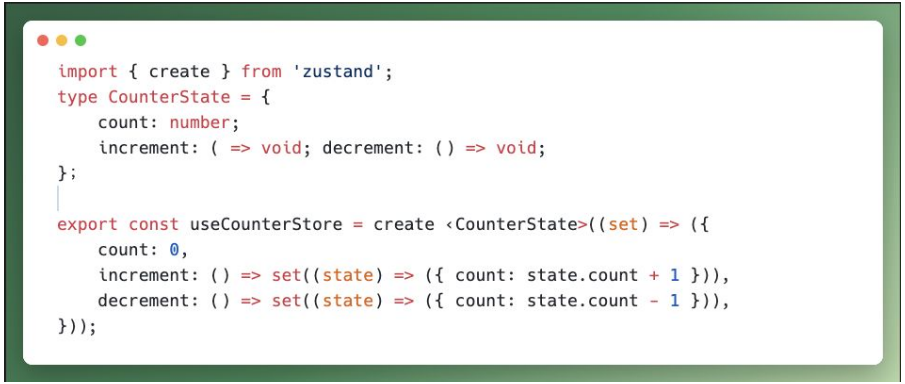
- **count**: a numeric value representing the current counter state.
- **increment and decrement**: functions that modify the state.

Create the Store (useCounterStore)
We call **create<CounterState>()** to build a store following our defined type. The store returns a **custom React hook** that we can use in any component.

set Function in Action
Zustand provides a set function that directly updates the state.

- **increment**: increases count by 1
- **decrement**: decreases count by 1

Result
With useCounterStore, we can access and update global state in any component without prop drilling or context setup — just like using useState, but globally.



```
import { create } from 'zustand';
type CounterState = {
    count: number;
    increment: ( => void; decrement: () => void;
};

export const useCounterStore = create ‹CounterState>((set) => ({
    count: 0,
    increment: () => set((state) => ({ count: state.count + 1 })),
    decrement: () => set((state) => ({ count: state.count - 1 })),
}));
```

Using the Store in a Component

Import the Store Hook

We import useCounterStore from our Zustand store file.
This hook gives us access to the global state and its actions.

Select State and Actions

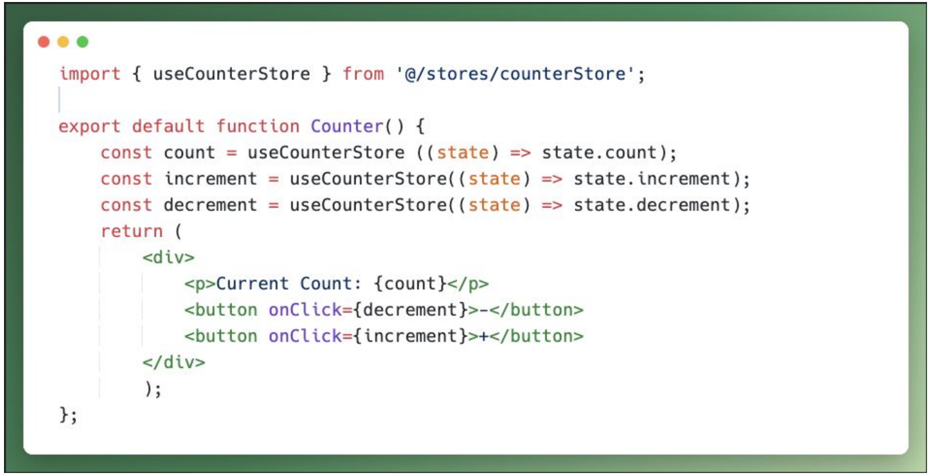
- **count** ➝ Reads the current value of the counter.
- **increment** ➝ Function to increase the counter.
- **decrement** ➝ Function to decrease the counter.
  Each is accessed by calling useCounterStore((state) => state.property).

Render the UI

- The **<p>** tag displays the current counter value.
- Two buttons **(+ and -)** call increment and decrement to update the state.

How it Works

When you click a button, Zustand automatically updates the global state.
The component re-renders only when the selected part of the state changes — making it **efficient and reactive.**



```
import { useCounterStore } from '@/stores/counterStore';

export default function Counter() {
    const count = useCounterStore ((state) => state.count);
    const increment = useCounterStore((state) => state.increment);
    const decrement = useCounterStore((state) => state.decrement);
    return (
        <div>
            <p>Current Count: {count}</p>
            <button onClick={decrement}>-</button>
            <button onClick={increment}>+</button>
        </div>
        );
};
```

Exercise

- Create Sign In Page at todo list app.
- Try to store input data from Sign In Page at global store zustand
- Access data email at todo app navbar

Thank you