Effect Cleanup

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the **useEffect** Hook.

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

 useEffect(() => {
  let timer = setTimeout(() => {
  setCount((count) => count + 1);
 }, 1000);

 return () => clearTimeout(timer)
 }, []);

 return <h1>I've rendered {count} times!</h1>;
}
```

React Context

React Context is a way to manage state globally.

It can be used together with the **useState** Hook to share state between deeply nested components more easily than with **useState** alone.

The Problem

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
function Component1() {
 const [user, setUser] = useState("Jesse Hall");
 return (
  <>
   <h1>{`Hello ${user}!`}</h1>
   <Component2 user={user} />
  </>
 );  }
function Component2({ user }) {
```

```
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
        <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

The Solution

To overcome the problem of prop drilling we make use **useContext**

Create Context

To create context, we must Import **createContext** and initialize it:

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

const UserContext = createContext()
```

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}
```

useContext Hook

In order to use the Context in a child component, we need to access it using the **useContext** Hook.

First, we need to include the **useContext** in the import statement:

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

Then we can access the user Context in all components:

```
function Component5() {
```

```
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

React useRef Hook

The **useRef** Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

```
function App() {
  const [inputValue, setInputValue] = useState("");
  const count = useRef(0);

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
```

```
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h1>Render Count: {count.current}</h1>
    </>
  );
}
```

useRef() only returns one item. It returns an Object called **current**.

Accessing DOM Elements

In React, we can add a **ref** attribute to an element to access it directly in the DOM.

```
function App() {
  const inputElement = useRef();

  const focusInput = () => {
    inputElement.current.focus();
  };

  return (
    <>
      <input type="text" ref={inputElement} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
```

Tracking State Changes

The **useRef** Hook can also be used to keep track of previous state values.

```
function App() {
  const [inputValue, setInputValue] = useState("");
  const previousInputValue = useRef("");

  useEffect(() => {
    previousInputValue.current = inputValue;
  }, [inputValue]);

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h2>Current Value: {inputValue}</h2>
      <h2>Previous Value: {previousInputValue.current}</h2>
    </>
```

```
); }
```

React useCallback Hook

The React **useCallback** Hook returns a memoized callback function.
The **useCallback** Hook only runs when one of its dependencies update.

One reason to use **useCallback** is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the **Todos** component will not re-render unless the **todos** change:

```
const App = () => {

 const [count, setCount] = useState(0);

 const [todos, setTodos] = useState([]);



 const increment = () => {

  setCount((c) => c + 1);

 };

 const addTodo = useCallback(() => {

  setTodos((t) => [...t, "New Todo"]);

 }, [todos]);



 return (

  <>

   <Todos todos={todos} addTodo={addTodo} />

   <hr />
```

```
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};
import { memo } from "react";

const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>
  );
};
export default memo(Todos);
```

React useMemo Hook

The React useMemo Hook returns a memoized value.
The **useMemo** Hook only runs when one of its dependencies updates.

We can use the **useMemo** Hook to memoize the **expensiveCalculation** function. This will cause the function to only run when needed.

We can wrap the expensive function call with **useMemo**.

The **useMemo** Hook accepts a second parameter to declare dependencies. The expensive function will only run when its dependencies have changed.

const memoizedValue = useMemo(() => computeValue(a, b), [a, b]);

```
import React, { useState, useMemo } from 'react';


function ExpensiveComponent({ number }) {
  const expensiveCalculation = (num) => {
    console.log("Calculating...");
    return num * 2;
  };

  const doubledNumber = useMemo(() => {
    return expensiveCalculation(number);
  }, [number]);

  return (
    <div>
      <p>Original: {number}</p>
      <p>Doubled (memoized): {doubledNumber}</p>
    </div>
  );
}
```

```
export default function App() {
  const [count, setCount] = useState(0);
  const [otherState, setOtherState] = useState(false);

  return (
    <div>
      <button onClick={() => setCount(c => c + 1)}>Increment Number</button>
      <button onClick={() => setOtherState(s => !s)}>Toggle</button>
      <ExpensiveComponent number={count} />
    </div>
  );
}
```

React useReducer

It is a **React Hook** that helps manage **complex state logic**

- When the state has multiple sub-values
- The next state depends on the previous one
- You want a cleaner alternative to multiple useState calls

const [state, dispatch] = useReducer(reducerFunction, initialState);

reducerFunction — a function that takes the current state and an action, and returns a new state

initialState — your starting state value

dispatch — a function used to send actions to the reducer

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <h2>Count: {state.count}</h2>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
    </div>
  );
};

export default Counter;
```

1. localStorage

- **What it is**: Stores key-value pairs in the browser.
- **Persists** even after the browser/tab is closed.
- Max size: ~5-10MB.
- **Good for**: Tokens, user preferences, app settings.

```
// Set item
localStorage.setItem('username', 'John');

// Get item
const user = localStorage.getItem('username');


// Remove item
localStorage.removeItem('username');

// Clear all
localStorage.clear();
```

2. sessionStorage

- Similar to localStorage, **BUT**
    - It only lasts until the **tab or window is closed**.
- **Good for**: Data you only need while the user is on the page/tab.

```
// Set item
sessionStorage.setItem('cart', JSON.stringify(['item1', 'item2']));
```

```
// Get item
sessionStorage.getItem('cart');

// Remove item
sessionStorage.removeItem('cart');
```

3. Cookies

- Can store small bits of data (usually < 4KB).
- **Can be sent automatically to the server with every HTTP request**.
- Useful for **authentication**, **tracking**, etc.
- You can control expiration, security, etc.

```
// Set cookie
document.cookie = "token=abc123; expires=Fri, 20 Apr 2025 12:00:00 UTC; path=/";

// Read cookie
console.log(document.cookie);

// Cookies usually managed with libraries in React, like js-cookie
```

Fetching API in React using `fetch`

Let's assume you're getting data from a placeholder API like https://jsonplaceholder.typicode.com/posts.

Example Code (Using useEffect and useState)

```
import React, { useEffect, useState } from 'react';
```

```
const Posts = () => {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then((data) => {
        setPosts(data);
        setLoading(false);
      })
      .catch((error) => {
        setError(error.message);
        setLoading(false);
      });
  }, []);
```

```
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <h2>Posts:</h2>
      <ul>
        {posts.slice(0, 5).map((post) => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default Posts;
```