Web Dev Mastery

📘 React Hooks – Complete Guide

🔹 All React Hooks

1️⃣ State & Lifecycle Hooks

- `useState`
- `useEffect`
- `useLayoutEffect`

2️⃣ Context & State Management Hooks

- `useContext`
- `useReducer`

3️⃣ Performance Optimization Hooks

- `useMemo`
- `useCallback`
- `useTransition`

4️⃣ Ref & DOM Interaction Hooks

- `useRef`

5️⃣ Id & Debugging Hooks

- `useId`

6️⃣ Custom Hook

- `useFetch`

🔹 Code Examples, Use Cases, and Benefits

1️⃣ `useState` – State Management

📌 When to use?
Use when you need local component state, such as tracking user input, toggling elements, or storing values.

✅ Benefits:

- **Simplifies state management in functional components.**
- **Rerenders the component when the state changes.**

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

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

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

2️⃣ `useEffect` – Side Effects

📌 When to use?
Use when you need to perform side effects such as API calls, event listeners, or subscriptions.

✅ Benefits:

- **Runs after rendering to handle side effects.**
- **Cleanup logic prevents memory leaks.**

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

function Posts() {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const response = await
fetch("https://jsonplaceholder.typicode.com/posts");
        const data = await response.json();
        setPosts(data);
      } catch (error) {
        console.error("Error fetching posts:", error);
      } finally {
        setLoading(false);
      }
    };

    fetchPosts();
  }, []);

  if (loading) return <p>Loading posts...</p>;

  return (
    <div>
```

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

export default Posts;
```

3️⃣ `useLayoutEffect` – Sync Effects Before Paint

📌 When to use?
Use when you need to measure DOM size or manipulate the layout before the browser paints.

✅ Benefits:

- **Runs before browser paint, ensuring smoother UI updates.**

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

function ChangeBackground() {
  const [color, setColor] = useState("lightblue");

  useLayoutEffect(() => {
    document.body.style.backgroundColor = color;
  }, [color]); // Runs before paint when color changes

  return (
    <div style={{ textAlign: "center", padding: "20px" }}>
      <h2>Current Background: {color}</h2>
      <button onClick={() => setColor("lightcoral")}>Red</button>
      <button onClick={() => setColor("lightgreen")}>Green</button>
      <button onClick={() => setColor("lightblue")}>Blue</button>
    </div>
  );
}
```

```
export default ChangeBackground;
```

4️⃣ `useContext` – Context API Hook

📌 **When to use?**
Use when you need to access global state without prop drilling.

✅ **Benefits:**

- **Avoids passing props down multiple levels.**

Steps:

1️⃣ Create Context (`MoneyContext.jsx`)

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

// Creating Context

const MoneyContext = createContext();

export default MoneyContext;
```

2️⃣ Create Provider (`MoneyState.jsx`)

```
import MoneyContext from "./MoneyContext";

const MoneyState = (props) => {

  const money = 10000; // Context Value

  return (

    <MoneyContext.Provider value={{ money }}>

      {props.children}
```

```
    </MoneyContext.Provider>
  );
};
export default MoneyState;
```

3️⃣ Wrap the App with Provider (`App.jsx`)

```
import React from "react";
import People from "./People";
import MoneyState from "./MoneyState";
const App = () => {
  return (
    <MoneyState>
      <People />
    </MoneyState>
  );
};
export default App;
```

4️⃣ Use The Context in Child Component (`People.jsx`)

```
import { useContext } from "react";
import MoneyContext from "./MoneyContext";
const People = () => {
  const { money } = useContext(MoneyContext); // Using Context
  return <h1>Money = {money}</h1>;
```

```
};

export default People;
```

5️⃣ `useReducer` – Advanced State Management

📌 **When to use?**
Use when managing complex state logic, such as in form handling or state transitions.

✅ **Benefits:**

- **Works well for complex state logic.**
- **Predictable state updates using actions.**

📂 Folder Structure

```
/useReducerExample
  ├── src
  │   ├── components
  │   │   ├── Counter.jsx
  │   │   ├── counterReducer.js
  │   ├── App.jsx
  │   ├── main.jsx
```

1️⃣ Create Reducer File (counterReducer.js)

This file contains the **reducer function** that manages the counter state.

```
// counterReducer.js
const counterReducer = (state, action) => {
  switch (action.type) {
```

```
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    case "RESET":
      return { count: 0 };
    default:
      return state;
  }
};

export default counterReducer;
```

2️⃣ Create Counter Component (Counter.jsx)

This component **uses useReducer** to manage the counter state.

```
// Counter.jsx
import React, { useReducer } from "react";
import counterReducer from "./counterReducer";

const Counter = () => {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });

  return (
    <div style={{ textAlign: "center", padding: "20px" }}>
      <h2>Counter: {state.count}</h2>
      <button onClick={() => dispatch({ type: "INCREMENT"
})}>+</button>
      <button onClick={() => dispatch({ type: "DECREMENT"
})}>-</button>
      <button onClick={() => dispatch({ type: "RESET"
})}>Reset</button>
```

```
    </div>
  );
};

export default Counter;
```

3️⃣ Wrap It in App.jsx

The App.jsx imports the Counter component.

```
// App.jsx
import React from "react";
import Counter from "./components/Counter";

const App = () => {
  return (
    <div>
      <h1 style={{ textAlign: "center" }}>useReducer Example</h1>
      <Counter />
    </div>
  );
};

export default App;
```

6️⃣ useMemo – Optimize Performance

📌 When to use?
Use when performing expensive calculations to prevent unnecessary recomputation.

✅ Benefits:

- Caches the result, preventing unnecessary recalculations.

📂 File Structure

```
/useMemoExample
  ├── src
  │   ├── App.js  (Main Component)
  │   ├── index.js  (Entry Point)
  ├── package.json
  ├── README.md
```

📄 File: App.js (Without useMemo)

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

const App = () => {
  const [cart, setCart] = useState([
    { id: 1, name: "Laptop", price: 50000 },
    { id: 2, name: "Phone", price: 30000 },
    { id: 3, name: "Headphones", price: 2000 },
  ]);
  const [discount, setDiscount] = useState(0);

  // Calculate total price (Runs on every render)
  const totalPrice = cart.reduce((total, item) => {
    console.log("Calculating total price...");
    return total + item.price;
  }, 0);

  return (
    <div>
      <h2>Shopping Cart</h2>
      {cart.map((item) => (
        <p key={item.id}>{item.name}: ₹{item.price}</p>
      ))}
      <h3>Total Price: ₹{totalPrice}</h3>
      <button onClick={() => setDiscount(discount + 10)}>Increase
Discount</button>
    </div>
  );
};
```

```
export default App;
```

🛑 Problem Explanation:

- Every time **discount** changes, the component **re-renders**.
- The totalPrice **recalculates unnecessarily**, even though cart hasn't changed.
- This **wastes performance**.

🟢 Solution: Using useMemo (Optimized Code)

📄 File: App.js (Using useMemo)

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

const App = () => {
  const [cart, setCart] = useState([
    { id: 1, name: "Laptop", price: 50000 },
    { id: 2, name: "Phone", price: 30000 },
    { id: 3, name: "Headphones", price: 2000 },
  ]);
  const [discount, setDiscount] = useState(0);

  // Memoizing the total price calculation
  const totalPrice = useMemo(() => {
    console.log("Calculating total price...");
    return cart.reduce((total, item) => total + item.price, 0);
  }, [cart]); // Runs only when cart changes

  return (
    <div>
      <h2>Shopping Cart</h2>
```

```
      {cart.map((item) => (
        <p key={item.id}>{item.name}: ₹{item.price}</p>
      ))}
      <h3>Total Price: ₹{totalPrice}</h3>
      <button onClick={() => setDiscount(discount + 10)}>Increase
Discount</button>
    </div>
  );
};

export default App;
```

7️⃣ `useCallback` – Prevent Unnecessary Function Recreation

📌 **When to use?**
Use when passing functions as props to prevent unnecessary re-renders.

✅ **Benefits:**

- Prevents function recreation on each render.

📂 File Structure

```
/useCallbackExample
  ├── src
  │   ├── App.js  (Parent Component)
  │   ├── Button.js  (Child Component)
  │   ├── index.js  (Entry point)
  ├── package.json
  ├── README.md
```

🔴 Problem: Without useCallback (Unoptimized Code)

📄 File: Button.js (Child Component)

```
import React from "react";

const Button = ({ onClick }) => {
  console.log("Button re-rendered!");
  return <button onClick={onClick}>Click Me</button>;
};

export default Button;
```

📄 File: App.js (Parent Component)

```
import React, { useState } from "react";
import Button from "./Button";

const App = () => {
  const [count, setCount] = useState(0);
  const [darkMode, setDarkMode] = useState(false);

  // Function gets recreated on every render
  const handleClick = () => {
    setCount(count + 1);
  };

  return (
```

```
    <div style={{ background: darkMode ? "#333" : "#fff", color:
darkMode ? "#fff" : "#000", padding: "20px" }}>
      <h2>Count: {count}</h2>
      <Button onClick={handleClick} />
      <button onClick={() => setDarkMode(!darkMode)}>Toggle
Theme</button>
    </div>
  );
};

export default App;
```

🛑 Problem Explanation:

- The handleClick function is **re-created** every time App re-renders.
- Since handleClick **changes on each render**, the Button component **also re-renders unnecessarily**.
- This **wastes performance**, especially when there are multiple child components.

🟢 Solution: Using useCallback (Optimized Code)

📄 File: Button.js (Child Component with React.memo)

```
import React from "react";

const Button = React.memo(({ onClick }) => {
  console.log("Button re-rendered!");
  return <button onClick={onClick}>Click Me</button>;
});
```

```
export default Button;
```

📄 File: App.js (Parent Component with useCallback)

```
import React, { useState, useCallback } from "react";
import Button from "./Button";

const App = () => {
  const [count, setCount] = useState(0);
  const [darkMode, setDarkMode] = useState(false);

  // Memoizing handleClick function using useCallback
  const handleClick = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  return (
    <div style={{ background: darkMode ? "#333" : "#fff", color:
darkMode ? "#fff" : "#000", padding: "20px" }}>
      <h2>Count: {count}</h2>
      <Button onClick={handleClick} />
      <button onClick={() => setDarkMode(!darkMode)}>Toggle
Theme</button>
    </div>
  );
};

export default App;

  }, []);

  return <Child onClick={handleClick} />;
}
```

8️⃣ `useRef` – Reference DOM Elements

📌 **When to use?**
Use when you need to persist values without causing re-renders or reference DOM elements.

✅ **Benefits:**

- Stores a mutable value without causing re-renders.

File: `App.js` (Using `useRef` to Change Image on Button Click)

```
import React, { useRef, useState } from "react";

const App = () => {
  const imageRef = useRef(null);
  const [isFirstImage, setIsFirstImage] = useState(true);

  const handleImageChange = () => {
    if (imageRef.current) {
      imageRef.current.src = isFirstImage
        ?
"https://via.placeholder.com/300/FF5733/FFFFFF?text=Second+Image"
        :
"https://via.placeholder.com/300/3498db/FFFFFF?text=First+Image";
      setIsFirstImage(!isFirstImage);
    }
  };

  return (
    <div>
      <h2>Change Image Using useRef</h2>
      <img
        ref={imageRef}

src="https://via.placeholder.com/300/3498db/FFFFFF?text=First+Image"
        alt="Placeholder"
        width="300"
      />
      <br />
```

```
      <button onClick={handleImageChange}>Change Image</button>
    </div>
  );
};

export default App;
```

9️⃣ `useId` – Unique IDs for Accessibility

📌 **When to use?**
Use when generating unique IDs for accessibility or form elements.

✅ **Benefits:**

- Avoids ID conflicts across multiple renders.

```
import { useId } from "react";

function Form() {
  const id = useId();
  return <label htmlFor={id}>Name: <input id={id} /></label>;
}
```

🔟 `useTransition` – Deferred Rendering for Performance

📌 **When to use?**
Use when transitioning between UI states without blocking interactions.

✅ **Benefits:**

- Prioritizes rendering, improving performance in complex UIs.

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

function App() {
```

```
  const [search, setSearch] = useState("");
  const [isPending, startTransition] = useTransition();

  const handleSearch = (e) => {
    startTransition(() => {
      setSearch(e.target.value);
    });
  };

  return <input onChange={handleSearch} value={search} />;
}
```

11️⃣ **useFetch** – Custom Hook for Fetching Data

📂 File Name: `useFetch.js`

Custom Hook: `useFetch`

`useFetch` is a reusable React hook that fetches data from an API, handling loading, errors, and responses efficiently.

🔹 Code - `useFetch.js`

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

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      try {
        const response = await fetch(url);
        if (!response.ok) throw new Error("Failed to fetch data");
        const result = await response.json();
        setData(result);
```

```
        setError(null);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
};

export default useFetch;
```

🔹 How to Use in a Component

📂 **File Name:** `UserList.js`

```
import React from "react";
import useFetch from "./useFetch";

const UserList = () => {
  const { data, loading, error } =
useFetch("https://jsonplaceholder.typicode.com/users");

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

  return (
    <ul>
      {data.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
```

```
};

export default UserList;
```

🔹 Other Hooks

`useImperativeHandle` – Customizes behavior of a component's `ref`.

`useDebugValue` – Provides a custom label for React DevTools.

`useSyncExternalStore` – Allows subscribing to external state sources (React 18+).

`useInsertionEffect` – Runs before DOM mutations, useful for injecting styles.

📝 Summary of Hook Uses & Benefits

Hook	When to Use	Benefits
`useState`	Manage local component state	Simple and fast
`useEffect`	Handle side effects (API, subscriptions)	Clean and efficient
`useContext`	Access global state	Avoids prop drilling
`useReducer`	Complex state logic	Predictable updates
`useMemo`	Expensive calculations	Optimized performance
`useCallback`	Memoize functions	Prevents unnecessary renders
`useRef`	Access DOM elements	No re-renders
`useTransition`	Deferred rendering	Improved responsiveness
