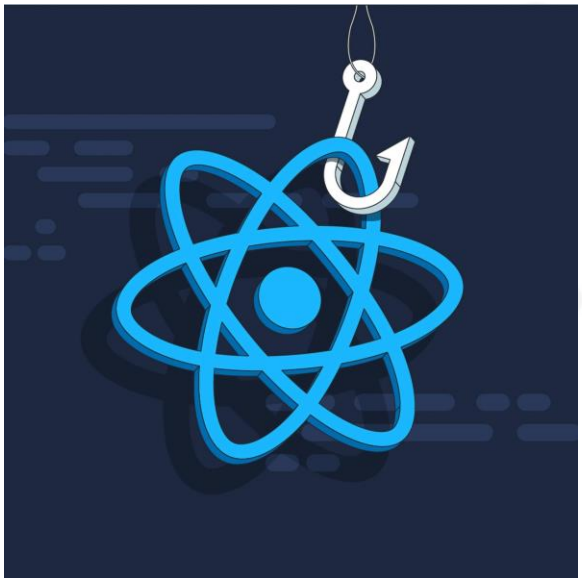
Hooks

Hooks are special functions that let you “hook into” React features from function components. They were introduced in **React 16.8** to let developers use **state and lifecycle methods** without writing class components.



Note:

- You can use hooks only within React functional components.
- Hooks must be invoked at the top level of a React function, not inside loops or conditionals.
- You should not call hooks conditionally; they must run in the same order on every render.
- React class components do not support hooks.

Most Common React Hooks

Hook	Purpose
useState	Adds state to a functional component
useEffect	Performs side effects (like fetching data, timers)
useContext	Accesses values from React Context
useRef	Keeps a mutable reference that doesn't re-render the component
useMemo	Optimizes performance by memoizing expensive calculations
useCallback	Memoizes callback functions to prevent unnecessary re-renders
useReducer	Alternative to useState for complex state logic
useId	Generates unique IDs for accessibility or list rendering

useEffect

useEffect is a **hook** in React that lets you run **side effects** in function components. Side effects include things like:

- Fetching data from an API
- Subscribing to events
- Directly manipulating the DOM
- Setting up timers

```
useEffect(() => {
        // Code to run on mount or update
        return () => {
                // Cleanup code (optional)
        };
}, [dependencies]);
```

useEffect

Key Concepts of useEffect:

Behavior	Example
Runs on every render	useEffect(() => { ... })
Runs only on the first render	useEffect(() => { ... }, [])
Runs when a variable (props or state) changes	useEffect(() => { ... }, [count])
Cleanup (like componentWillUnmount)	Use return () => { ... } inside useEffect

useRef

Think of **useRef** like a **box** that can store a value.

- The value stays the same between re-renders.
- Updating it **does not** cause the component to re-render.
- Most commonly used to **access DOM elements** directly.

```
const refContainer = useRef(initialValue);
```

Why do we need it?

Sometimes, you need to:

- Focus an input field
- Store some value between renders without causing a re-render (like a counter)

Why use useRef instead of useState?

If we attempt to track the number of times our app renders using the **useState** Hook, it will trigger an infinite loop because updating state itself causes a re-render. To prevent this, we can use the **useRef** Hook instead.

useReducer

useReducer is just like useState, but with **a brain (reducer function)** that decides how to update the state.
It's perfect when your state logic is **complex** or **depends on the previous state**.

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

state → Current value (like from useState)
dispatch → Function you call to update the state
reducerFunction → A function (state, action) => newState that decides how to change the state

Why use useReducer instead of useState?

When to use useState → Simple state (e.g., count, text)
When to use useReducer → Complex state logic or multiple related values
Example: managing a form, todo list, shopping cart.

useMemo

useMemo is a React Hook that **caches the result of a calculation** so React doesn't re-calculate it unnecessarily on every render.
It is mainly used for **performance optimization**.

```
const memoizedValue = useMemo(() => {
            // some expensive calculation
            return result;
}, [dependencies]);
```

() => { ... } → A function returning the value you want to cache.
[dependencies] → Only when these values change, the function will re-run.

✅ **When to use useMemo**
You have **expensive calculations** (sorting, filtering, computing).
You want to avoid unnecessary recalculations.

❌ **Don't use everywhere:**
For small, cheap calculations → adds unnecessary complexity.

useCallback

useCallback is a React Hook that **returns a memoized version of a function**. It ensures that the **same function instance** is reused **between renders** (unless dependencies change).
It is mainly used to **prevent unnecessary re-renders of child components**.

```
const memoizedCallback = useCallback(() => {
            // function body
}, [dependencies]);
```

Problem Without useCallback

- Imagine we have a parent component that passes a function (addTodo) to a child component.
- Every time the parent re-renders, a **new function is created**, causing the child to re-render too (even if props didn't change).

useCallback with Dependency

```
const memoizedFn = useCallback(() => {
// function logic
}, [dependency1, dependency2]);
```

The function will only be **re-created** if one of the dependencies changes.
If dependencies remain the same → React **reuses the old function** (avoiding re-renders in memorized children).

Why dependencies matter

- If you leave [] (empty array), the function is created **once** and never changes → but it won't capture updated state.
- If you include state/props in [dependencies], the function updates **whenever those values change**.

useId

```
const id = useId();
```

useId introduced in React 18.
It helps generate **unique IDs** that stay consistent across server and client rendering.

Why do we need useId?

- Normally, when you create form elements like <label> and <input>, they should be connected using the for (or htmlFor) attribute. That requires **unique IDs**.
- If you hardcode IDs, they might **clash** when you have multiple components.
  If you generate IDs randomly, they might **mismatch between server and client** in SSR (React hydration issue).
- useId solves this by giving **unique, stable, deterministic IDs**.