HOOKS IN REACT

1. use State

- A hook that allows you to add state (data) to functional components.
- It returns a state variable and a function to update it.
- When the state changes, the component re-renders.

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

function Example() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

2. use Effect

- A hook that lets you perform side effects in a component.
- Examples: fetching data, setting timers, or subscribing to events.
- Runs after rendering and can include a cleanup function.

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

function Example() {
  const [time, setTime] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => setTime(t => t + 1), 1000);
    return () => clearInterval(timer); // cleanup
  }, []);

  return <h3>Time: {time}</h3>;
}
```

3. use Context

- A hook that allows you to use values from a React Context.
- It helps avoid **prop drilling** by providing data directly to components.

```jsx
import React, { useContext, createContext } from "react";

const UserContext = createContext("Guest");

function Example() {
  const user = useContext(UserContext);
  return <h2>Hello, {user}</h2>;
}
```

4. useReducer

- An alternative to useState for managing complex state logic.
- Works with a reducer function that takes the current state and an action, and returns a new state.

```jsx
import React, { useReducer } from "react";

function reducer(state, action) {
  if (action.type === "inc") return state + 1;
  return state;
}

function Example() {
  const [count, dispatch] = useReducer(reducer, 0);
  return <button onClick={() => dispatch({ type: "inc" })}>{count}</button>;
}
```

5. useCallback

- A hook that memoizes a function.
- It returns the same function instance unless its dependencies change.
- Useful to prevent unnecessary re-renders of child components.

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

function Example() {
  const [count, setCount] = useState(0);
  const increment = useCallback(() => setCount(c => c + 1), []);
  return <button onClick={increment}>Count: {count}</button>;
}
```

6. useMemo

- A hook that memoizes the result of a calculation.
- Only re-computes when its dependencies change.
- Helps optimize performance for expensive computations.

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

function Example() {
  const [num, setNum] = useState(2);
  const double = useMemo(() => num * 2, [num]);
  return <h2>Double: {double}</h2>;
}
```

7. useRef

- A hook that gives you a mutable object (ref).
- You can store a value or access DOM elements with it.
- Changing a ref does not cause re-render.

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

function Example() {
  const inputRef = useRef();
  return (
    <>
      <input ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>Focus</button>
    </>
  );
}
```

8. useImperativeHandle

- Used with forwardRef to customize the value exposed to parent components via refs.
- Allows you to define custom methods that parent components can call.

```jsx
import React, { useImperativeHandle, useRef, forwardRef } from "react";

const MyInput = forwardRef((props, ref) => {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current.focus(),
  }));
  return <input ref={inputRef} />;
});

function Example() {
  const ref = useRef();
  return <button onClick={() => ref.current.focus()}>Focus Input</button>;
}
```

9. useLayoutEffect

- Similar to useEffect, but it runs **synchronously** after all DOM updates.
- Useful when you need to measure or mutate the DOM before the browser paints.

```jsx
import React, { useLayoutEffect, useRef } from "react";

function Example() {
  const divRef = useRef();
  useLayoutEffect(() => {
    divRef.current.style.color = "red";
  }, []);
  return <div ref={divRef}>Hello</div>;
}
```

10. useDebugValue

- Used inside custom hooks to display a label in React DevTools.
- Helpful for debugging custom hooks.

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

function useCounter() {
  const [count, setCount] = useState(0);
  useDebugValue(count > 5 ? "High" : "Low");
  return [count, setCount];
}
```

11. useId (React 18)

- Generates a unique, stable ID for elements.
- Useful for accessibility attributes like id and htmlFor.

```jsx
import React, { useId } from "react";
function InputField() {
  const id = useId();
  return <>
    <label htmlFor={id}>Name</label>
    <input id={id} />
  </>;
}
```

12. useTransition (React 18)

- Allows you to mark certain state updates as **non-urgent**.
- Improves UI responsiveness by deferring expensive updates.

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

function Example() {
  const [isPending, startTransition] = useTransition();
  const [text, setText] = useState("");

  const handleChange = (e) => {
    startTransition(() => setText(e.target.value));
  };

  return (
    <>
      <input onChange={handleChange} />
      {isPending ? "Loading..." : <p>{text}</p>}
    </>
  );
}
```

13. useDeferredValue (React 18)

- Defers updating a value until less urgent rendering work is finished.
- Helps keep the UI responsive when rendering large lists or heavy computations.

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

function Example() {
  const [text, setText] = useState("");
  const deferredText = useDeferredValue(text);

  return (
    <>
      <input onChange={(e) => setText(e.target.value)} />
      <p>{deferredText}</p>
    </>
  );
}
```

14. useSyncExternalStore (React 18)

- Used to subscribe to external data sources (like Redux or other stores).
- Ensures the component stays in sync with external state changes.

```jsx
import React, { useSyncExternalStore } from "react";

function subscribe(callback) {
  window.addEventListener("resize", callback);
  return () => window.removeEventListener("resize", callback);
}

function Example() {
  const size = useSyncExternalStore(
    subscribe,
    () => window.innerWidth
  );
  return <h2>Width: {size}</h2>;
}
```

15. useInsertionEffect (React 18)

- Runs synchronously before DOM mutations are applied.
- Mainly used for injecting styles dynamically.

```jsx
import React, { useInsertionEffect } from "react";

function Example() {
  useInsertionEffect(() => {
    const style = document.createElement("style");
    style.innerHTML = "body { background: lightblue; }";
    document.head.appendChild(style);
  }, []);
  return <h2>Background Applied</h2>;
}
```

Hook	Definition	Use-case (When to Use)
useState	Adds state to a functional component.	For counters, toggles, form inputs.
useEffect	Runs side effects after render.	API calls, timers, event listeners.
useContext	Access values from Context.	Avoid prop drilling (theme, auth, language).
useReducer	Manages complex state with reducer function.	Todo app, forms, multiple state updates.
useCallback	Memoizes a function.	Prevents re-render of child components.
useMemo	Memoizes result of a calculation.	Expensive calculations, filtering, sorting.
useRef	Stores mutable value or DOM reference.	Access input focus, store previous values.
useImperativeHandle	Customizes value exposed via `ref`.	Expose methods (like `.focus()`) to parent.
useLayoutEffect	Runs before paint, after DOM update.	DOM measurement, animations, sync layout.
useDebugValue	Shows custom hook value in DevTools.	Debugging custom hooks.
useId (React 18)	Generates unique stable IDs.	For form elements (`id`, `htmlFor`).
useTransition (React 18)	Marks state updates as non-urgent.	Smooth UI updates (search box typing).
useDeferredValue (React 18)	Defers non-urgent value updates.	Large list rendering without blocking input.
useSyncExternalStore (React 18)	Syncs with external store.	Redux, Zustand, window events.
useInsertionEffect (React 18)	Runs before DOM mutations.	Inject styles dynamically (CSS-in-JS).