React - Practical

Basic Level Questions

1.Timer using `useEffect`

Goal: Increase a number every second (like a clock).

🔸 Concept:

- `useEffect` runs after component mounts.
- We use `setInterval` to update a timer every second.
- We return a cleanup function to stop it when the component unmounts.

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // cleanup
  }, []);

  return <h2>Timer: {seconds} seconds</h2>;
  }
```

2.Stopwatch using `useState` and `useEffect`

Goal: Start and stop a timer manually.

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

function Stopwatch() {
  const [time, setTime] = useState(0);
  const [isRunning, setIsRunning] = useState(false);

  useEffect(() => {
    let interval;
    if (isRunning) {
      interval = setInterval(() => {
        setTime(t => t + 1);
      }, 1000);
    } else {
      clearInterval(interval);
    }

    return () => clearInterval(interval);
  }, [isRunning]);

  return (
    <div>
      <h2>Stopwatch: {time}s</h2>
      <button onClick={() => setIsRunning(true)}>Start</button>
      <button onClick={() => setIsRunning(false)}>Stop</button>
      <button onClick={() => { setTime(0); setIsRunning(false); }}>Reset</button>
    </div>
  );
}
```

3.Change `document.title` using `useRef`

Goal: Update the page title when a button is clicked

```
import { useRef } from 'react';

function TitleChanger() {
  const inputRef = useRef();

  const changeTitle = () => {
    document.title = inputRef.current.value;
  };

  return (
    <div>
      <input ref={inputRef} placeholder="Enter new title" />
      <button onClick={changeTitle}>Change Title</button>
    </div>
  );
}
```

4. Change background color using `useRef`

Goal: Change the background color of a `div` when clicking a button.

```
import { useRef } from 'react';

function BackgroundChanger() {
  const boxRef = useRef();

  const changeColor = () => {
    boxRef.current.style.backgroundColor = "lightgreen";
  };

  return (
    <div>
      <div ref={boxRef} style={{ width: "200px", height: "100px", backgroundColor: 
"lightgray" }}></div>
      <button onClick={changeColor}>Change Background</button>
    </div>
  );
}
```

5.Change text color using `useRef`

Goal: Change the text color of a `p` tag using `useRef`.

```
import { useRef } from 'react';

function TextColorChanger() {
  const textRef = useRef();

  const changeTextColor = () => {
    textRef.current.style.color = "red";
  };

  return (
    <div>
      <p ref={textRef}>This is some text</p>
      <button onClick={changeTextColor}>Change Text Color</button>
    </div>
  );
}
```

6.Counter using `useRef` and `useState`

Goal: Create a counter that shows count and stores previous count using `useRef`.

```
import { useRef, useState } from 'react';

function RefStateCounter() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();

  const increment = () => {
    prevCountRef.current = count; // Store previous value
    setCount(count + 1);
  };

  return (
    <div>
      <h2>Current: {count}</h2>
      <h3>Previous: {prevCountRef.current}</h3>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

7.Passing state from **Parent to Child**

Goal: Send data from parent to child component via props.

```
function Parent() {
  const [name, setName] = useState("Bessy");
  return <Child username={name} />;
}

function Child({ username }) {
  return <h2>Hello {username}!</h2>;
}
```

8.Passing callback function from **Child to Parent**

Goal: Trigger a function defined in the parent when a button is clicked in the child.

```
function Parent() {
  const greet = (msg) => {
    alert("Child says: " + msg);
  };

  return <Child sendMessage={greet} />;
}

function Child({ sendMessage }) {
  return (
    <button onClick={() => sendMessage("Hi from Child!")}>
      Send to Parent
    </button>
  );
}
```

9.Child to Parent Communication using `useRef`

Goal: Access child's input value from parent using `useRef`

```
import { useRef } from 'react';

function Parent() {
  const childInputRef = useRef();

  const getChildValue = () => {
    alert("Child Input: " + childInputRef.current.value);
  };

  return (
    <div>
      <Child inputRef={childInputRef} />
      <button onClick={getChildValue}>Get Child Value</button>
    </div>
  );
}

function Child({ inputRef }) {
  return <input ref={inputRef} placeholder="Type something" />;
}
```

10.How to store data inside `useRef`

Goal: Store any data (like previous state, timer ID, etc.) inside `useRef`.

```
import { useRef, useState } from 'react';

function StoreInRef() {
  const inputRef = useRef();          // For DOM element
  const dataRef = useRef("Initial"); // For storing value

  const updateData = () => {
    dataRef.current = inputRef.current.value;
    alert("Stored in useRef: " + dataRef.current);
  };

  return (
    <div>
      <input ref={inputRef} placeholder="Enter value" />
      <button onClick={updateData}>Store in Ref</button>
    </div>
  );
}
```

11.Change `div` color when clicking a button

Goal: Click a button to change the color of a `div`.

```
import { useRef } from 'react';

function ChangeDivColor() {
  const divRef = useRef();

  const changeColor = () => {
    divRef.current.style.backgroundColor = "skyblue";
  };

  return (
    <div>
      <div ref={divRef} style={{ width: "200px", height: "100px", backgroundColor:
"lightgray" }} />
      <button onClick={changeColor}>Change Div Color</button>
    </div>
  );
}
```

12.Check whether text in 2 input fields match

Goal: Validate if two inputs (like password fields) are equal.

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

function MatchInputs() {
  const [text1, setText1] = useState('');
  const [text2, setText2] = useState('');

  return (
    <div>
      <input onChange={(e) => setText1(e.target.value)} placeholder="Enter Text 1"
/>
      <input onChange={(e) => setText2(e.target.value)} placeholder="Enter Text 2"
/>
      <p>{text1 === text2 ? "Matching" : "Not Matching"}</p>
    </div>
  );
}
```

13.Create input field to show entered text in `h1`

Goal: Show live user input inside an `h1`

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

function LiveInputDisplay() {
  const [text, setText] = useState('');

  return (
    <div>
      <input onChange={(e) => setText(e.target.value)} placeholder="Type
something..." />
      <h1>{text}</h1>
    </div>
  );
}
```

14.Lifecycle events using `useEffect`

Goal: Show mount, update, and unmount behaviors using `useEffect`.

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

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

  useEffect(() => {
    console.log("Mounted or Updated. Count:", count);
    return () => {
      console.log("Component Unmount or Before Update");
    };
  }, [count]);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(c => c + 1)}>Increase</button>
    </div>
  );
}
```

15.Show alert

Goal: When clicked, it shows a browser alert with a message.

```
import React from 'react';

function ShowAlert() {
  const handleClick = () => {
    alert("Hello Bessy! This is an alert.");
  };

  return (
    <div>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}

export default ShowAlert;
```

Intermediate Level Questions

16.Counter using `useContext`

Goal: Share counter state across multiple components using Context API.

```
// CounterContext.js
import { createContext, useState } from 'react';
export const CounterContext = createContext();

export function CounterProvider({ children }) {
  const [count, setCount] = useState(0);
  return (
    <CounterContext.Provider value={{ count, setCount }}>
      {children}
    </CounterContext.Provider>
  );
}

// App.js
import { useContext } from 'react';
import { CounterContext, CounterProvider } from './CounterContext';

function CounterDisplay() {
  const { count } = useContext(CounterContext);
  return <h2>Count: {count}</h2>;
}

function CounterControls() {
  const { setCount } = useContext(CounterContext);
  return <button onClick={() => setCount(c => c + 1)}>Increment</button>;
}

function App() {
  return (
    <CounterProvider>
      <CounterDisplay />
      <CounterControls />
    </CounterProvider>
  );
}

export default App;
```

17.Methods to store and update data using Context API

With Context, we,

1. **Create Context** → `createContext()`

2. **Wrap app with Provider** → `<MyContext.Provider value={data}>`
3. **Consume in children** → `useContext(MyContext)`
4. **Update** by using `setState()` (or `useReducer()` for complex updates)

18. `useRef` set timer with start and stop

Goal: Use `useRef` to hold timer ID so we can start/stop it easily.

```
import { useState, useRef } from 'react';

function TimerWithRef() {
  const [count, setCount] = useState(0);
  const timerRef = useRef(null);

  const startTimer = () => {
    timerRef.current = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);
  };

  const stopTimer = () => {
    clearInterval(timerRef.current);
  };

  return (
    <div>
      <h2>{count}</h2>
      <button onClick={startTimer}>Start</button>
      <button onClick={stopTimer}>Stop</button>
    </div>
  );
}
```

19. Render component once when page is resized

Goal: Run a side-effect when the window is resized using `useEffect`.

```
import { useEffect } from 'react';

function ResizeComponent() {
  useEffect(() => {
    const handleResize = () => {
      console.log("Window Resized:", window.innerWidth);
    };

    window.addEventListener('resize', handleResize);

    return () => {
      window.removeEventListener('resize', handleResize); // Cleanup
    };
  }, []);

  return <h2>Resize the window and check console.</h2>;
}
```

20.Implement Higher Order Component (HOC)

Goal : Reuse it

```
function withGreeting(WrappedComponent) {
  return function EnhancedComponent(props) {
    return (
      <div>
        <h2>Hello from HOC!</h2>
        <WrappedComponent {...props} />
      </div>
    );
  };
}

function SimpleComponent() {
  return <p>I’m a normal component</p>;
}

const Enhanced = withGreeting(SimpleComponent);

function App() {
  return <Enhanced />;
}
```

21.Create Custom Hook

Goal: Make reusable logic using custom hook

```
// useCounter.js
import { useState } from 'react';

export function useCounter(initial = 0) {
  const [count, setCount] = useState(initial);
  const increment = () => setCount(c => c + 1);
  const decrement = () => setCount(c => c - 1);
  return { count, increment, decrement };
}


// App.js
import { useCounter } from './useCounter';

function CounterComponent() {
  const { count, increment, decrement } = useCounter();

  return (
    <div>
      <h2>{count}</h2>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  );
}
```

22. `useMemo` Implementation

Goal: Optimize expensive calculations.

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

function ExpensiveCalcComponent() {
  const [count, setCount] = useState(0);
  const [toggle, setToggle] = useState(true);

  const expensiveResult = useMemo(() => {
    console.log("Calculating...");
    return count * 100;
  }, [count]);

  return (
    <div>
      <h2>Expensive: {expensiveResult}</h2>
      <button onClick={() => setCount(c => c + 1)}>Increment Count</button>
      <button onClick={() => setToggle(!toggle)}>Toggle</button>
    </div>
  );
}
```

23.Counter using `useState` and `useReducer`

With `useState`:

```
const [count, setCount] = useState(0);
<button onClick={() => setCount(c => c + 1)}>+</button>
```

✅ With `useReducer`:

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

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

function ReducerCounter() {
  const [count, dispatch] = useReducer(reducer, 0);

  return (
    <div>
      <h2>{count}</h2>
      <button onClick={() => dispatch({ type: 'inc' })}>+</button>
      <button onClick={() => dispatch({ type: 'dec' })}>-</button>
    </div>
  );
}
```

24.To-do List using `useReducer

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

function todoReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [ ... state, { text: action.payload }];
    case 'remove':
      return state.filter((_, i) => i !== action.index);
    default:
      return state;
  }
}

function TodoApp() {
  const [todos, dispatch] = useReducer(todoReducer, []);
  const [text, setText] = useState('');

  return (
    <div>
      <input onChange={(e) => setText(e.target.value)} />
      <button onClick={() => dispatch({ type: 'add', payload: text })}>
        Add Todo
      </button>
      <ul>
        {todos.map((todo, i) => (
          <li key={i}>
            {todo.text}
            <button onClick={() => dispatch({ type: 'remove', index: i })}>
              X
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

25.Toggle Button ON/OFF

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

function ToggleButton() {
  const [isOn, setIsOn] = useState(false);

  return (
    <button onClick={() => setIsOn(!isOn)}>
      {isOn ? "ON" : "OFF"}
    </button>
  );
}
```

26.List UI using `<li>`

```
function ListComponent() {
  const items = ["React", "Django", "PostgreSQL"];

  return (
    <ul>
      {items.map((item, i) => <li key={i}>{item}</li>)}
    </ul>
  );
}
```

Advanced Level Questions

27.Store items to `localStorage`

Goal: Store data in browser storage that stays even after refresh

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

function LocalStorageExample() {
  const [name, setName] = useState("");

  useEffect(() => {
    const stored = localStorage.getItem("username");
    if (stored) {
      setName(stored);
    }
  }, []);

  const handleChange = (e) => {
    setName(e.target.value);
    localStorage.setItem("username", e.target.value);
  };

  return (
    <div>
      <input value={name} onChange={handleChange} placeholder="Enter Name" />
      <p>Hello, {name}</p>
    </div>
  );
}
```

Explanation:

- `localStorage.setItem("key", value)` saves data.
- `localStorage.getItem("key")` fetches data.
- `useEffect` loads data on component mount

28. Example of `useCallback`

Goal: Memoize a function to avoid re-creating it on each render.

```
import { useCallback, useState } from 'react';

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

  const handleClick = useCallback(() => {
    console.log("Clicked");
  }, []);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(c => c + 1)}>+</button>
      <button onClick={handleClick}>Log</button>
    </div>
  );
}
```

Explanation:

- `useCallback(fn, deps)` returns memoized version of function.
- Prevents unnecessary re-creations unless dependencies change.

29.Lazy Loading with Example

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <h1>Main App</h1>
      <Suspense fallback={<p>Loading...</p>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```

Explanation:

- `lazy()` tells React to load `MyComponent` only when needed.
- `Suspense` shows fallback (like "Loading...") while loading.

30.Create a simple `useEffect` with `fetch` (Async Callback)

Goal: Fetch data from API using async in `useEffect`

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

function FetchData() {
  const [user, setUser] = useState([]);

  useEffect(() => {
    async function fetchData() {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      const data = await res.json();
      setUser(data);
    }
    fetchData();
  }, []);

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

Explanation:

- We define `async fetchData()` inside `useEffect`.
- It fetches user data and updates the state

31.Input field entered text show in `<h1>` tag

Goal: Show whatever the user types in a heading.

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

function InputToH1() {
  const [text, setText] = useState("");

  return (
    <div>
      <input onChange={(e) => setText(e.target.value)} />
      <h1>{text}</h1>
    </div>
  );
}
```

32.Check if two input fields match (like passwords)

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

function MatchInputs() {
  const [first, setFirst] = useState('');
  const [second, setSecond] = useState('');

  const match = first === second;

  return (
    <div>
      <input placeholder="Enter" onChange={(e) => setFirst(e.target.value)} />
      <input placeholder="Confirm" onChange={(e) => setSecond(e.target.value)} />
      <p>{match ? "Matched ✅" : "Not Matching ❌"}</p>
    </div>
  );
}
```

33.Change a `<div>`'s background color on button

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

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

  return (
    <div>
      <div style={{ height: 100, backgroundColor: color }}>Color Box</div>
      <button onClick={() => setColor("lightgreen")}>Change Color</button>
    </div>
  );
}
```

34.Store data inside `useRef`

Goal: Hold values without causing re-renders.

```
import { useRef } from 'react';

function RefStorage() {
  const valueRef = useRef("");

  const handleChange = (e) => {
    valueRef.current = e.target.value;
    console.log("Stored value:", valueRef.current);
  };

  return (
    <div>
      <input onChange={handleChange} />
      <p>Check console for stored value</p>
    </div>
  );
}
```