## ✅ Core Deep Concepts to Master

### 1. ****Virtual DOM & Reconciliation****

* **Virtual DOM**: A lightweight JS representation of the real DOM. React updates this first and then **diffs** the new vs old VDOM (a process called **reconciliation**) to compute the minimal set of changes to the real DOM.
* **Misconception**: React doesn’t always only update the changed node—it **re-renders the entire component** if its props/state change. Only the **DOM update** is optimized.
* **Why whole form re-renders**:
  + If the Form component has internal state or props that depend on the input, changing an input can trigger a re-render of the form.
  + React does **component-level re-renders**, not element-level.
  + If you don’t isolate the Input with React.memo or split into child components, the parent Form re-renders.

**Interview Q:** If React uses virtual DOM, why do components still re-render so frequently?  
**Answer:** Because reconciliation determines what to update on the DOM, but rendering a component (i.e., running its function) is different. React still has to re-run components' render logic to compare virtual DOMs.

### 2. ****Component Re-renders & Optimization****

* **React.memo()**: Prevents re-rendering of a functional component unless its props change.
* **useCallback & useMemo**:
  + useCallback: Returns a memoized callback.
  + useMemo: Returns a memoized **value**.
  + Both help avoid unnecessary child re-renders.

js

CopyEdit

const memoizedFn = useCallback(() => doSomething(), [deps]);

**Interview Q:** When would you use useMemo and when React.memo?

### 3. ****Cleanup in**** useEffect

* **Purpose**: Prevent memory leaks or stale effects.
* Examples:
  + Clearing timeouts/intervals.
  + Unsubscribing from events or WebSockets.

js

CopyEdit

useEffect(() => {

const id = setInterval(logData, 1000);

return () => clearInterval(id); // cleanup

}, []);

**Interview Q:** What would happen if we forget to add a cleanup in a useEffect with an event listener?

### 4. ****State Batching and Event Loop****

* React batches state updates for performance (in synthetic events and lifecycle methods).
* **Non-synthetic async code (e.g., setTimeout)** may not be batched unless you're using **React 18+ automatic batching**.

js

CopyEdit

setTimeout(() => {

setCount(c => c + 1);

setFlag(f => !f);

}, 1000);

**Interview Q:** How does React’s event loop interact with state batching?

### 5. ****React 18 Features****

* **Automatic Batching**: setTimeout/fetch state changes are now batched.
* **Concurrent Rendering**: React can interrupt rendering and work on multiple tasks simultaneously.
* **useTransition**: For marking state updates as non-urgent.

### 6. ****Keys in Lists (And Why They're Important)****

* React uses key to identify which items have changed, added, or removed.
* Bad keys (like index) lead to bugs in dynamic lists (e.g., losing focus on input fields).

**Interview Q:** Why shouldn’t we use array index as a key?

### 7. ****Controlled vs Uncontrolled Components****

* Controlled: value is managed via React state.
* Uncontrolled: Use ref to get value from DOM directly.

js

CopyEdit

<input value={input} onChange={e => setInput(e.target.value)} />

**Interview Q:** Which is better for performance and why?

### 8. ****Context API & Prop Drilling****

* Context avoids prop drilling but may cause unnecessary re-renders.
* Use useContextSelector or memoization to optimize.

**Interview Q:** What are the downsides of the Context API for deeply nested trees?

### 9. ****Custom Hooks****

* Abstract repeated logic across components.

js

CopyEdit

function useLocalStorage(key, defaultValue) {

const [value, setValue] = useState(() => localStorage.getItem(key) || defaultValue);

useEffect(() => {

localStorage.setItem(key, value);

}, [key, value]);

return [value, setValue];

}

**Interview Q:** When would you write a custom hook instead of a utility function?

### 10. ****Error Boundaries****

* Used to catch JavaScript errors in the component tree.
* Only works with **class components** (you can wrap functional components).

jsx

CopyEdit

class ErrorBoundary extends React.Component {

state = { hasError: false };

static getDerivedStateFromError(error) { return { hasError: true }; }

componentDidCatch(error, info) { logError(error, info); }

render() { return this.state.hasError ? <Fallback /> : this.props.children; }

}

### 11. ****React Suspense & Lazy Loading****

* Load components or data **asynchronously**.

js

CopyEdit

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

**Interview Q:** How does Suspense help improve performance and user experience?

### 12. ****Refs and Forwarding Refs****

* useRef stores mutable values and DOM references.
* Forwarding refs allows parent to access child's DOM node.

js

CopyEdit

const MyInput = React.forwardRef((props, ref) => <input ref={ref} {...props} />);

## 🧠 Bonus Topics (Senior-level)

* **Render Props vs HOC vs Hooks**
* **React Fiber architecture** (scheduler for concurrent rendering)
* **React DevTools performance profiling**
* **Server-Side Rendering (Next.js)**
* **Hydration**
* **React Testing Library best practices**
* **How React handles re-renders at the scheduler level**

## 🔁 useCallback – Memoizes Functions

### ✅ Use when:

You want to **prevent a function from being recreated** on every render, especially when you pass it as a **prop** to child components.

### 💡 Without useCallback (function recreated every render):

jsx

CopyEdit

const MyComponent = () => {

const handleClick = () => {

console.log("Clicked");

};

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

};

✅ Works fine... but every re-render **creates a new** handleClick **function**, even if logic doesn't change.

### ✅ With useCallback (memoized function):

jsx

CopyEdit

import { useCallback } from 'react';

const MyComponent = () => {

const handleClick = useCallback(() => {

console.log("Clicked");

}, []); // only create once

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

};

### 🧠 Why?

If handleClick is passed to a child component, and the child is memoized, re-creating the function would still cause the child to re-render. useCallback avoids that.

## 🧠 useMemo – Memoizes Computations

### ✅ Use when:

You have **expensive calculations** that shouldn’t re-run unless **dependencies change**.

### 💡 Without useMemo (calculated on every render):

jsx

CopyEdit

const MyComponent = ({ number }) => {

const squared = number \* number; // recalculated every time

return <div>{squared}</div>;

};

### ✅ With useMemo:

jsx

CopyEdit

import { useMemo } from 'react';

const MyComponent = ({ number }) => {

const squared = useMemo(() => {

console.log("Calculating...");

return number \* number;

}, [number]);

return <div>{squared}</div>;

};

### 🧠 Why?

squared is only recalculated when number changes, saving computation and avoiding unnecessary re-renders.

## 🧱 React.memo – Memoizes Components

### ✅ Use when:

You want to **prevent re-rendering of a component** if its **props didn't change**.

### 💡 Without React.memo:

jsx

CopyEdit

const Child = ({ name }) => {

console.log("Rendering Child");

return <div>Hello {name}</div>;

};

const Parent = () => {

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

return (

<>

<Child name="Vishnu" />

<button onClick={() => setCount(count + 1)}>Increment</button>

</>

);

};

🔴 **Child will re-render** every time Parent re-renders, even though name never changes.

### ✅ With React.memo:

jsx

CopyEdit

const Child = React.memo(({ name }) => {

console.log("Rendering Child");

return <div>Hello {name}</div>;

});

✅ Now Child will only re-render **if** name **prop changes**.

## 🔁 Putting It All Together

Here’s a real-world pattern:

jsx

CopyEdit

const Child = React.memo(({ onClick }) => {

console.log("Child rendered");

return <button onClick={onClick}>Click</button>;

});

const Parent = () => {

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

// Without useCallback, this would recreate onClick every render

const onClick = useCallback(() => {

console.log("Clicked");

}, []);

return (

<>

<Child onClick={onClick} />

<button onClick={() => setCount(c => c + 1)}>Increment</button>

</>

);

};

✅ Now:

* Child is memoized with React.memo
* onClick is memoized with useCallback
* So, changing count doesn't re-render the child.

## 🔍 Summary Table

| Hook / Method | What it memoizes | When to use |
| --- | --- | --- |
| useCallback | Function references | Passing functions to child components |
| useMemo | Computation result | Expensive calculations on props/state |
| React.memo | Entire component | Prevent re-render if props don’t change |

## 🔍 What is React Virtual DOM (vDOM)?

### 🤓 Definition:

The **Virtual DOM** is a lightweight **copy** of the actual browser DOM that React uses to:

1. Track changes in the UI.
2. Batch and apply only **minimal changes** to the real DOM.

## ✅ Step-by-Step: How Virtual DOM Works

### 1. ****Initial Render:****

React creates a **virtual DOM tree** (JavaScript objects representing HTML elements).

jsx

CopyEdit

<div>

<input value="test" />

<p>Hello</p>

</div>

React stores this in memory, like:

js

CopyEdit

{

type: 'div',

props: {

children: [

{ type: 'input', props: { value: 'test' } },

{ type: 'p', props: { children: 'Hello' } }

]

}

}

### 2. ****State or Props Change:****

User types something → setState() is called → React creates a **new virtual DOM**.

### 3. ****Diffing:****

React **compares (diffs)** old vDOM with new vDOM.

Only the **changed parts** are identified (e.g., input value).

### 4. ****Reconciliation:****

React finds the **minimal number of actual DOM updates** and patches the real DOM.

So if only the input value changed, **only that input’s value is updated** — not the whole DOM.

### 🚨 Then why does "the whole form reload" on input change?

This is key: **React doesn't reload the entire page**. But you might be triggering a full re-render of your component or app unnecessarily.

Let’s explain with **why this happens in practice**:

## ⚠️ Common Reasons Why “Everything Seems to Re-render”

### 🔁 1. ****No**** key ****in List Rendering****

Improper keys in dynamic lists cause full remounting.

jsx

CopyEdit

{items.map(item => <ItemComponent item={item} />)} // ❌ wrong

Should be:

jsx

CopyEdit

{items.map(item => <ItemComponent key={item.id} item={item} />)} // ✅

### 🔁 2. ****State at the top level forces re-render****

If your entire form is in **one big parent component**, any state update inside (like input) causes the **whole form to re-render** — even if the virtual DOM minimizes real DOM changes.

jsx

CopyEdit

const Form = () => {

const [name, setName] = useState('');

const [email, setEmail] = useState('');

return (

<>

<input value={name} onChange={e => setName(e.target.value)} />

<input value={email} onChange={e => setEmail(e.target.value)} />

</>

);

};

➡️ In this example, **any input change causes** Form **to re-render**, and so both inputs are rebuilt in the virtual DOM.

### 🧱 Solution: Split into components + memo

jsx

CopyEdit

const NameInput = React.memo(({ value, onChange }) => {

return <input value={value} onChange={onChange} />;

});

const EmailInput = React.memo(({ value, onChange }) => {

return <input value={value} onChange={onChange} />;

});

const Form = () => {

const [name, setName] = useState('');

const [email, setEmail] = useState('');

return (

<>

<NameInput value={name} onChange={e => setName(e.target.value)} />

<EmailInput value={email} onChange={e => setEmail(e.target.value)} />

</>

);

};

✅ Now only the changed input will **re-render** — thanks to React.memo.

## 🧠 Summary Table

| Concept | React Behavior |
| --- | --- |
| Virtual DOM | JS object copy of real DOM |
| Diffing | Compares old vs new virtual DOM |
| Reconciliation | Only changed nodes are patched to real DOM |
| Full component render | Happens on state/prop change |
| Real DOM update | Only changes minimal elements |
| Optimization | Use React.memo, split components, use stable props/functions |

## 🧪 Visual Example (Simplified)

You type in <input /> → React:

* Creates new vDOM: { input: { value: "A" } }
* Diffs with old one: { input: { value: "" } }
* Says: only value changed → updates just input.value in real DOM

But **your component function still runs fully** unless you optimize it!

Let me know if you want:

* A visual diagram 🖼️
* A CodeSandbox example 🔗
* Or a mock interview question based on virtual DOM 🎯