**React.js interview Questions**

### **ES6+ JavaScript Features**

**1. Rest vs. Spread Operators**

**Q:** What’s the difference between the rest and spread operators?  
 **A:**

* **Spread (...)** expands an iterable into individual elements.
* **Rest (...)** collects remaining elements into an array or object.  
   Example:

const obj = { a: 1, b: 2, c: 3 };  
const { a, ...rest } = obj; // rest = { b: 2, c: 3 }  
  
const arr1 = [1, 2];  
const arr2 = [...arr1, 3]; // [1, 2, 3]

In React, we often use spread in props:

<MyComponent {...props} />

**2. Arrow Functions & this**

**Q:** How does this behave differently in arrow functions?  
 **A:** Arrow functions **don’t bind their own this** — they inherit it from their lexical context.  
 In class components:

class MyComponent extends React.Component {  
 handleClick = () => {  
 console.log(this); // always refers to the component instance  
 }  
}

This avoids the need for .bind(this) in constructors.

**3. Promises vs. async/await**

**Q:** Refactor using async/await.

fetch('/api/user')

.then(res => res.json())

.then(data => console.log(data));

const fetchData = async () => {  
 try {  
 const res = await fetch('/api/user');  
 const data = await res.json();  
 console.log(data);  
 } catch (err) {  
 console.error(err);  
 }  
};

**Pros:** cleaner syntax, easier to read.  
 **Cons:** harder to run parallel async tasks unless using Promise.all.

**4. Map/Reduce/Filter**

**Q:** Transform an array of users into a role-count object:

"Can you walk me through how you'd use .map() and .reduce() to transform this array of objects into a summary object for a dashboard?"

const users = [  
 { name: 'Alice', role: 'admin' },  
 { name: 'Bob', role: 'editor' },  
 { name: 'Jane', role: 'admin' }  
];  
  
const result = users.reduce((acc, user) => {  
 acc[user.role] = (acc[user.role] || 0) + 1;  
 return acc;  
}, {});  
// result: { admin: 2, editor: 1 }

Great for building summaries on dashboards.

### **React (Hooks, State, Rendering)**

**5. useEffect vs Lifecycle Methods**

**Q:** How does useEffect differ?

How does useEffect differ from lifecycle methods in class components? When does it run? How do you avoid infinite loops with it?"  
 **A:** useEffect runs *after* the component renders.

* Equivalent to componentDidMount, componentDidUpdate, and componentWillUnmount depending on dependency array.

useEffect(() => {  
 console.log("Mounted or deps changed");  
 return () => console.log("Cleanup");  
}, [dep1, dep2]); // only re-runs when deps change

Avoid infinite loops by properly setting the dependency array.

**6. React Optimization: memo, useMemo, useCallback**

**Q:** When to use them?  
 **A:**

* React.memo: prevents re-rendering of components unless props change.
* useCallback: memoizes function references.
* useMemo: memoizes expensive calculations.

const memoizedValue = useMemo(() => computeExpensive(data), [data]);  
const handleClick = useCallback(() => doSomething(), []);

Use them when unnecessary re-renders or recalculations are hurting performance.

**7. State Management (Complex State)**

**Q:** Manage deeply nested state in forms?  
 **A:**

* If the form is large or nested, use useReducer instead of multiple useState calls.
* Otherwise, external state management like **Zustand**, **Redux**, or **React Hook Form**.

const reducer = (state, action) => {  
 switch (action.type) {  
 case 'UPDATE\_FIELD':  
 return { ...state, [action.field]: action.value };  
 }  
};

**8. Rendering Behavior & Optimization**

**Q:** Why might a component re-render unnecessarily?  
 **A:**

* Parent re-render
* Function/prop reference changes
* State set to same value (with a different reference)

**Prevent it with:**

* React.memo
* useCallback
* Lifting state carefully

Use tools like **React DevTools Profiler** to track render causes.

### **Next.js**

**9. getStaticProps vs getServerSideProps vs getInitialProps**

|  |  |  |
| --- | --- | --- |
| **Method** | **When it Runs** | **Use Case** |
| getStaticProps | Build time | Static content (e.g. marketing pages) |
| getServerSideProps | On each request | Real-time data (e.g. dashboards) |
| getInitialProps | Legacy, SSR | Used in older Next.js versions |

Prefer getStaticProps when data doesn’t change frequently.

**10. Dynamic Routing & File Structure**

**Q:** Explain dynamic routing for /blog/[slug]

"Explain how dynamic routing works in Next.js. Can you describe the file/folder structure needed for a blog with slug-based URLs like /blog/my-article?"  
 **A:** In pages/blog/[slug].js, use:

export async function getStaticPaths() {  
 const paths = posts.map(post => ({  
 params: { slug: post.slug },  
 }));  
 return { paths, fallback: false };  
}  
  
export async function getStaticProps({ params }) {  
 const post = getPostBySlug(params.slug);  
 return { props: { post } };  
}

**Folder structure:**

pages/  
 blog/  
 [slug].js

**Short React code snippets**

### **1. useEffect Pitfall**

useEffect(() => {  
 fetchData();  
}, []);

**Q:** What happens if fetchData is defined inside the component?

**A:**  
 If fetchData is defined inside the component, it will be recreated on every render. Since it's not in the dependency array, it may capture **stale state/props** or skip updates.  
 **Fix:** Add it to the dependencies, or define it with useCallback.

### **🔹 2. State Update Loop**

const [count, setCount] = useState(0);  
  
useEffect(() => {  
 setCount(count + 1);  
}, [count]);

**Q:** Why does this cause an infinite loop?

**A:**  
 Every call to setCount triggers a re-render, which changes count, so useEffect runs again — a loop.  
 **Fix:** Use a condition, or better yet:

setCount(prev => prev + 1);

and make sure it’s not inside a dependency-triggered effect.

### **🔹 3. useMemo Misuse**

const result = useMemo(() => expensiveComputation(), []);

**Q:** Why might this optimization be ineffective?

**A:**  
 If expensiveComputation() doesn't depend on props or state, useMemo adds unnecessary complexity. Also, if the result isn’t used or it's cheap to recompute, memoization is a waste.

### **🔹 4. React.memo and Prop Equality**

const data = { name: 'John' };  
return <Child data={data} />;

**Q:** Why does Child re-render even if the data looks the same?

**A:**  
 A **new object reference** is created on every render. Shallow comparison in React.memo sees it as different.  
 **Fix:** Memoize data with useMemo:

const data = useMemo(() => ({ name: 'John' }), []);

### **🔹 5. Next.js getStaticProps**

export async function getStaticProps() { ... }

**Q:** How often is this data fetched?

**A:**  
 Only at **build time**. The page is static.  
 To regenerate every hour:

return {  
 props: { ... },  
 revalidate: 3600, // ISR (Incremental Static Regeneration)  
};

### **🔹 6. useCallback Trap**

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

**Q:** Will this log the updated count?

**A:**  
 No. count is not in the dependency array, so it captures the initial value (likely 0).  
 **Fix:** Add count to the dependency array:

useCallback(() => console.log(count), [count]);

### **🔹 7. Conditional Rendering Quirk**

{isLoggedIn && <Dashboard />}

**Q:** What if isLoggedIn is 0 or ''?

**A:**  
 React treats 0, '', and false as falsy — so it won’t render <Dashboard />, even if 0 is a valid value.  
 **Fix:** Be explicit:

{isLoggedIn === true && <Dashboard />}

### **🔹 8. Component Re-render Debugging**

<MyComponent onClick={() => console.log("clicked")} />

**Q:** How to prevent re-renders if onClick is recreated every time?

**A:**  
 Wrap the function in useCallback or define it outside the render body:

const handleClick = useCallback(() => console.log("clicked"), []);  
<MyComponent onClick={handleClick} />

Or use React.memo on the child component.

### **🔹 9. State Update Race Condition**

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

**Q:** Is this safe with rapid clicks?

**A:**  
 Not always. If React batches updates, the count variable might be stale.  
 **Fix:** Use updater function:

setCount(prev => prev + 1);

### **🔹 10. Lazy Initial State**

const [items, setItems] = useState(() => computeInitialItems());

**Q:** Why is this better than calling computeInitialItems() directly?

**A:**  
 Using a function delays execution until **first render only**. If you call it directly, it runs on **every render**, even if state isn't used.

***11. State & useEffect: Execution Order***

function Demo() {

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

useEffect(() => {

console.log("Effect ran:", count);

}, [count]);

console.log("Rendered:", count);

return (

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

);

}

#### **Question:**

What will be logged to the console when the button is clicked?

#### **Expected Answer:**

* On first render: Rendered: 0 → Effect ran: 0
* On click: Rendered: 1 → Effect ran: 1
* So: Render happens **before** the useEffect.

***12. Props and Re-renders***

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

console.log("Child render");

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

});

function Parent() {

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

return (

<>

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

<Child value="static" />

</>

);

}

#### **Question:**

Will the Child component re-render on every click?

#### **Expected Answer:**

* No. React.memo avoids re-render since prop (value) does not change.

***13. Callback with Stale State***

***function Counter() {***

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

***const handleClick = useCallback(() => {***

***setCount(count + 1);***

***}, []);***

***return <button onClick={handleClick}>Count: {count}</button>;***

***}***

#### **Question:**

Will setCount(count + 1) work correctly on every click?

#### **Answer:**

* No, count is stale (always 0).
* Fix: Use functional update: setCount(c => c + 1);

14. Dependency Misuse

useEffect(() => {

const timer = setInterval(() => {

console.log("Tick:", count);

}, 1000);

return () => clearInterval(timer);

}, []);

#### **Question:**

Will this log the updated count?

#### **Answer:**

* No. It captures the count at first render (closure issue).
* Fix: Add count to dependency array or use ref.

5. **Re-render Trigger**

**function App() {**

**const data = { name: "John" };**

**return <Child data={data} />;**

**}**

#### **Question:**

Will the child component re-render on every parent render?

#### **Answer:**

* Yes, because data is a **new object** each time.
* To prevent this, use useMemo in parent:

const data = useMemo(() => ({ name: "John" }), []);

====================================================================

## **Counter with Step**

### **Problem Statement Recap:**

Modify the app to include a button that increases a counter by 2 on each click. Show the current count.

### **Solution Code (App.js)**

import "./styles.css";

import { useState } from "react";

export default function App() {

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

const handleClick = () => {

setCount(count + 2);

};

return (

<div className="App">

<h1>Counter with Step</h1>

<p>Count: {count}</p>

<button onClick={handleClick}>Increase by 2</button>

</div>

);

}

### **What This Tests:**

* Basic use of useState
* Event handling (onClick)
* JSX and component re-renders

For a **functional update** (to avoid stale state in async contexts), use:

**setCount(prev => prev + 2);**

## **2. Input Reset on Submit**

Add an input box and a Submit button. When submitted, show the value below and clear the input.

// Add:

// [input box] [Submit]

// On submit: show the text and clear input

import "./styles.css";

import { useState } from "react";

export default function App() {

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

const [submitted, setSubmitted] = useState("");

const handleSubmit = () => {

if (text.trim() === "") return;

setSubmitted(text);

setText("");

};

return (

<div className="App">

<h1>Input Reset</h1>

<input

type="text"

value={text}

onChange={e => setText(e.target.value)}

placeholder="Type something"

/>

<button onClick={handleSubmit}>Submit</button>

{submitted && <p>You submitted: {submitted}</p>}

</div>

);

}

#### **What It Tests:**

* Controlled components
* State updates
* Form handling

## **Toggle Text**

Toggle between "Hello" and "Goodbye" on button click

import { useState } from "react";

export default function App() {

const [isHello, setIsHello] = useState(true);

return (

<div>

<h1>{isHello ? "Hello" : "Goodbye"}</h1>

<button onClick={() => setIsHello(prev => !prev)}>Toggle</button>

</div>

);

}

## **Checkbox Controlled**

Show "Subscribed!" when checkbox is checked

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

***export default function App() {***

***const [isChecked, setIsChecked] = useState(false);***

***return (***

***<div>***

***<label>***

***<input***

***type="checkbox"***

***checked={isChecked}***

***onChange={e => setIsChecked(e.target.checked)}***

***/>***

***Subscribe***

***</label>***

***{isChecked && <p>Subscribed!</p>}***

***</div>***

***);***

***}***

## **Even/Odd Checker**

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

**export default function App() {**

**const [num, setNum] = useState("");**

**const isEven = num !== "" && parseInt(num, 10) % 2 === 0;**

**return (**

**<div>**

**<input**

**type="number"**

**value={num}**

**onChange={e => setNum(e.target.value)}**

**placeholder="Enter number"**

**/>**

**{num !== "" && <p>{isEven ? "Even" : "Odd"}</p>}**

**</div>**

**);**

**}**

## **Simple Calculator**

Two inputs + button = show sum

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

**export default function App() {**

**const [a, setA] = useState("");**

**const [b, setB] = useState("");**

**const [sum, setSum] = useState(null);**

**const calculate = () => {**

**setSum(Number(a) + Number(b));**

**};**

**return (**

**<div>**

**<input**

**type="number"**

**value={a}**

**onChange={e => setA(e.target.value)}**

**placeholder="First number"**

**/>**

**<input**

**type="number"**

**value={b}**

**onChange={e => setB(e.target.value)}**

**placeholder="Second number"**

**/>**

**<button onClick={calculate}>Sum</button>**

**{sum !== null && <p>Result: {sum}</p>}**

**</div>**

**);**

**}**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*