**Senior React + Redux Interview Questions**

**React (30 Questions)**

**1. useEffect vs useLayoutEffect – Give a scenario where useLayoutEffect prevents UI flicker.**

**useEffect vs useLayoutEffect**

|  |  |  |
| --- | --- | --- |
| **Hook** | **Runs...** | **Use Case** |
| useEffect | **After** the browser paints (async) | Side effects, data fetching |
| useLayoutEffect | **Before** the browser paints (sync) | DOM measurement, layout adjustment |

**Scenario: Preventing UI Flicker in Animation or Resize**

***Problem (with useEffect)***

Let’s say you want to measure the height of an element and apply a style (like collapsing a panel):

useEffect(() => {  
 const height = ref.current.offsetHeight;  
 if (height > 300) {  
 ref.current.style.height = '300px';  
 }  
}, []);

* This **runs after paint**, so:
* The **full height is rendered briefly** (visible flicker)
* Then height is set to 300px afterward (visual jump)

***Solution (with useLayoutEffect)***

useLayoutEffect(() => {  
 const height = ref.current.offsetHeight;  
 if (height > 300) {  
 ref.current.style.height = '300px';  
 }  
}, []);

* This runs **before the browser paints**, so:
* Height is immediately adjusted before any pixels hit the screen
* No flicker, no jump

**When to Use useLayoutEffect**

* Measuring DOM (getBoundingClientRect, offsetHeight)
* Synchronous layout changes before paint
* Scroll position adjustments
* Canvas or chart reflows
* Preventing layout shift in modals, tooltips

**2. What is the output of the following?**

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

useEffect(() => {

setCount(count + 1);

}, [count]);

* useEffect runs **after** every render.
* The dependency array [count] means this effect runs **whenever count changes**.
* Inside the effect, you're calling setCount(count + 1) which **changes count**.
* That causes a re-render → which triggers useEffect again → which updates count again → and so on...

Why does this cause an infinite loop? Fix it.

useEffect(() => {

setCount(prev => prev + 1);

}, []);

* This uses [] as the dependency array, so it runs only once.
* setCount(prev => prev + 1) ensures you use the **latest state**, not a stale count.

**3. Write a useCounter custom hook that returns a count and increment(), decrement() functions.**

import { useState, useCallback } from "react";

function useCounter(initialValue = 0) {

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

const increment = useCallback(() => {

setCount((prev) => prev + 1);

}, []);

const decrement = useCallback(() => {

setCount((prev) => prev - 1);

}, []);

return { count, increment, decrement };

}

export default useCounter;

**How to Use It in a Component**

import React from "react";

import useCounter from "./useCounter";

function CounterComponent() {

const { count, increment, decrement } = useCounter(10); // starts at 10

return (

<div>

<h2>Count: {count}</h2>

<button onClick={increment}>+ Increment</button>

<button onClick={decrement}>- Decrement</button>

</div>

);

}

**Features**

* Supports an optional initial value
* Uses useCallback to memoize actions (performance benefit in lists)
* Clean and readable

**4. Optimize the component:**

const handleClick = () => {

console.log(item.name);

};

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

It’s being passed to 100 items in a list. Prevent unnecessary re-renders.

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

**Problem:**

The function:

const handleClick = () => {  
 console.log(item.name);  
};

is **created fresh on every render**. If this is done inside a list of 100 items, each item gets a **new reference to a different handleClick**, even if item.name hasn't changed.

This breaks React’s **memoization** (e.g., React.memo) and can cause performance issues.

**Optimized Solution:**

Use useCallback with dependencies to **memoize** the function:

import React, { useCallback } from "react";  
  
const ListItem = ({ item }) => {  
 const handleClick = useCallback(() => {  
 console.log(item.name);  
 }, [item.name]);  
  
 return <button onClick={handleClick}>Click</button>;  
};  
  
export default React.memo(ListItem);

**Why This Works:**

* useCallback ensures handleClick has the **same reference** unless item.name changes.
* Wrapping ListItem in React.memo ensures it **doesn't re-render** unless its props change.
* This improves rendering performance significantly in lists.

**Even Better with ID Lookup (Bonus):**

If the button just needs item.id, you can do:

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

Pass only primitive props to avoid re-renders due to object identity.

**5. How do you prevent stale closures in useEffect when accessing latest props/state?**

In React, **stale closures** happen when a useEffect callback *closes over an outdated version* of props or state.

**Example of the Problem:**

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

useEffect(() => {

const timer = setInterval(() => {

console.log("Count is", count); // stale count

}, 1000);

return () => clearInterval(timer);

}, []);

This will always log Count is 0, because count is **closed over at the time of mounting**, and never updates inside setInterval.

**Solution 1: Use Functional Updates or Refs**

***Option A: Use useRef to always point to the latest value***

const countRef = useRef(count);

useEffect(() => {

countRef.current = count; // update ref whenever count changes

}, [count]);

useEffect(() => {

const timer = setInterval(() => {

console.log("Count is", countRef.current); // ✅ latest count

}, 1000);

return () => clearInterval(timer);

}, []);

**Solution 2: Include the dependency properly**

If your useEffect callback depends on a value, **always include it in the dependency array**:

useEffect(() => {

const timer = setInterval(() => {

console.log("Count is", count); // fresh count

}, 1000);

return () => clearInterval(timer);

}, [count]); // <- depends on count

**Solution 3: Use a Functional Update**

For timers or event handlers that rely on **previous state**, use a functional form:

setCount(prev => prev + 1);

**6. Implement a reusable useDebounce(value, delay) hook.**

**7. What’s wrong with this? Fix it:**

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

useEffect(() => {

fetch("/api/items").then(res => res.json()).then(setItems);

}, [items]);

**The Problem:**

* The useEffect has items in its dependency array.
* Inside the effect, you're calling setItems(...) → this **updates items**
* Which causes a **re-render**, which re-triggers the useEffect, which calls setItems again…

**Result: Infinite loop.**

**The Fix:**

If you want to **fetch items only once on mount**, use an **empty dependency array []**:

useEffect(() => {

fetch("/api/items")

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

.then(setItems);

}, []); // runs only once

**Want to refetch on demand later?**

You can pull it into a function:

const fetchItems = async () => {

const res = await fetch("/api/items");

const data = await res.json();

setItems(data);

};

useEffect(() => {

fetchItems();

}, []);

Now you can call fetchItems() manually from a "Refresh" button if needed.

**Rule of Thumb:**

* Never include a state in the dependency array **if you're setting it inside the effect** unless you’re **explicitly handling reactivity** (like polling or watching changes).

**8. Convert this class component to functional using hooks:**

componentDidMount() {

this.fetchData();

}

**Original Class Component**

class MyComponent extends React.Component {

componentDidMount() {

this.fetchData();

}

fetchData() {

// logic to fetch data

}

render() {

return <div>Data here</div>;

}

}

**Functional Version Using Hooks**

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

**function MyComponent() {**

**useEffect(() => {**

**fetchData();**

**}, []); // runs once on mount (like componentDidMount)**

**const fetchData = () => {**

**// logic to fetch data**

**};**

**return <div>Data here</div>;**

**}**

**Why This Works:**

* useEffect(..., []) runs once after the component mounts — exactly like componentDidMount.
* Putting fetchData() **inside** the component ensures it's accessible from the effect.

**Pro Tip for Async**

If fetchData is async, make sure you define it **inside** useEffect, or call it properly:

**useEffect(() => {**

**const fetchData = async () => {**

**const data = await fetch(...);**

**// set state here**

**};**

**fetchData();**

**}, []);**

**9. Refactor to use controlled component:**

<input />

**Uncontrolled Component**

<input />

* React does **not** manage the input’s value.
* You cannot easily read, set, or reset the input from React s

**Refactored Controlled Component**

import { useState } from "react";  
  
function ControlledInput() {  
 const [value, setValue] = useState("");  
  
 const handleChange = (e) => {  
 setValue(e.target.value);  
 };  
  
 return (  
 <input value={value} onChange={handleChange} />  
 );  
}

**Explanation**

|  |  |
| --- | --- |
| **Feature** | **Value** |
| value prop | Controlled by React state |
| onChange | Updates the state on every keystroke |
| useState | Holds the current input text |

**Now You Can...**

* Read input value at any time: console.log(value)
* Set/reset it programmatically: setValue("reset")
* Validate or format it (e.g. trim, uppercase, etc.)

**10. What is the difference between useMemo and useCallback? When would you use each?**

**11. Build a <List> component that accepts a renderItem prop and renders any array generically.**

**12. What is the output of this code? Explain why.**

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

const double = useMemo(() => count \* 2, []);

**Output:**

double will be 0 and **will never update**, even if count changes.

**Why?**

* useMemo(() => count \* 2, []) means:

Only run the memoized function **once on mount**, because the dependency array is empty.

* So the value of count at that time is 0 → double becomes 0
* Even if you later setCount(5), double will still be 0 because the memo never recalculates.

**Correct Version**

If you want double to reflect the **latest count**, you must include count in the dependency array:

const double = useMemo(() => count \* 2, [count]);

Now double will recompute **every time count changes**, and will always be up to date.

**Summary**

|  |  |
| --- | --- |
| **Dependency Array** | **Behavior** |
| [] | Run only once (on mount) |
| [count] | Re-run whenever count changes |

**13. Create a basic error boundary component using class components.**

**14. Explain why lifting state up is often necessary. Refactor a child-to-parent communication.**

**15. How can you lazy load a route in React Router v6?**

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

import React, { Suspense, lazy } from "react";

// Lazy import components

const Home = lazy(() => import("./pages/Home"));

const About = lazy(() => import("./pages/About"));

const Contact = lazy(() => import("./pages/Contact"));

function App() {

return (

<Router>

{/\* Fallback shown while lazy-loaded routes are loading \*/}

<Suspense fallback={<div>Loading...</div>}>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="/contact" element={<Contact />} />

</Routes>

</Suspense>

</Router>

);

}

export default App;

**16. What are Suspense and lazy? Show usage.**

lazy → tells React to load a component lazily (when needed).

Suspense → shows a fallback UI while the lazy component is being fetched.

Together, they enable code-splitting and better performance.

**17. Write a usePrevious custom hook to access the previous value of a prop.**

**usePrevious.js**

import { useEffect, useRef } from "react";

function usePrevious(value) {

const ref = useRef();

useEffect(() => {

ref.current = value; // update ref after each render

}, [value]);

return ref.current; // return previous value

}

export default usePrevious;

**counter.jsx**

import React, { useState } from "react";

import usePrevious from "./usePrevious";

function Counter() {

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

const prevCount = usePrevious(count);

return (

<div>

<h2>Current: {count}</h2>

<h3>Previous: {prevCount}</h3>

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

</div>

);

}

export default Counter;

**18. Implement a TextInput component that accepts only digits (0–9).**

/d in regex to filter

**19. Refactor a list rendering to use a key properly. What problems does using index cause?**

Always prefer a stable, unique key from your data (like id).

Avoid using index unless the list is static (never reordered/filtered/modified).

Keys help React reconcile efficiently and preserve component state correctly.

**20. What’s a good way to cancel a fetch inside useEffect to prevent memory leaks?**

**21. What is the difference between useEffect(() => ..., []) and useLayoutEffect(() => ..., [])? When would you use one over the other?**

*Rendering lifecycle and visual side effects.*

**22. Why is state considered asynchronous in React? How do you reliably update state based on the previous value?**

*Functional updates: setState(prev => prev + 1).*

**23. How would you avoid unnecessary re-renders in a component tree with heavy props and nested children?**

*Mention React.memo, useMemo, useCallback, and selectors in state management.*

**24. When would you use Context API over Redux, and when is Redux the better choice?**

*Shows architectural thinking — global static config (Context) vs dynamic state (Redux).*

**25. What are the pitfalls of using index as a key in a list? What problems can it cause?**

*Answer should mention reordering issues and stale component state.*

**26. Explain how you’d design a reusable Modal or Dropdown component in React.**

*Component isolation, props, children, onClose, portals.*

**27. How do you manage side-effects like data fetching in React? Can you walk through a pattern you use for loading/error/success states?**

*Use of useEffect, loading flags, and error handling.*

**28. What are controlled vs uncontrolled components? Which one is preferred and why?**

*Controlled: React manages the state. Preferred for predictability.*

**29. Describe the Virtual DOM and reconciliation. Why is React performant despite re-renders?**

*Understanding of React’s rendering engine.*

**30. If a parent component re-renders, do all child components re-render too? How can this be prevented?**

*Look for React.memo, prop comparison, and lifting state .*

**Redux / Redux Toolkit (30 Questions)**

**1. Write a slice using createSlice to manage a list of todos.**

**2. Add async fetch to your slice using createAsyncThunk. Handle loading, success, error.**

**3. How do you access and update state in a React component using useSelector and useDispatch?**

**4. What is a selector? How would you memoize it using createSelector?**

**5. Given this state, write a selector to get total currentValue of selected assets:**

{

portfolio: {

items: [{ id, name, selected, currentValue }],

filter: "Stock"

}

}

**6. Normalize this array into a dictionary format in Redux slice:**

[

{ id: "1", name: "AAPL" },

{ id: "2", name: "TSLA" }

]

**7. What problem does createEntityAdapter solve? How is it different from manual normalization?**

**8. What middleware would you use to log actions and state after every dispatch?**

**9. How do you structure Redux slices for a multi-feature app (e.g., auth, dashboard, settings)?**

**10. Write a middleware that blocks any action of type "auth/LOGIN" if already logged in.**

**11. What happens if two reducers try to update the same slice of state? How would you manage conflicts?**

**12. Write a test for a Redux reducer using Jest. Mock initial state and assert updates.**

**13. How do you avoid re-rendering all components when one slice of Redux state changes?**

**14. What is the difference between dispatching plain actions vs thunks?**

**15. Explain Redux DevTools and how they help debugging. How do you integrate them?**

**16. What’s the difference between global Redux store and local component state? When to use which?**

**Global Redux Store vs Local Component State**

|  |  |  |
| --- | --- | --- |
| **🔍 Feature** | **⚛️ Local State (useState)** | **🌐 Global State (Redux Store)** |
| **Scope** | Component-level | App-wide (shared across components) |
| **Persistence** | Lost on unmount | Persisted across components & routes |
| **Access** | Only within the component | Accessible via useSelector() |
| **Use for** | UI state, form fields, toggles | Auth state, user data, theme, cart |
| **Setup** | Simple (just useState) | Needs store, slice, provider setup |
| **Debuggable** | Harder to track | Redux DevTools provides history |
| **Sharing across components** | Requires prop drilling or lifting | No prop drilling — accessed globally |

**When to Use Local Component State**

Use useState or useReducer **within components** when:

* State is **only needed in one component**
* Example:
* Input fields in a form
* Modal open/close toggle
* UI hover/focus states
* Tab selection within a single page

js

CopyEdit

const [isOpen, setIsOpen] = useState(false);

**✅ When to Use Global Redux State**

Use Redux when:

* State is **shared across multiple components or routes**
* You need **global access**, such as:
* Authentication/user info
* Shopping cart
* Feature flags or theme mode
* Real-time data cache (e.g., prices, portfolios)
* You want **centralized async logic and action logging**

js

CopyEdit

const user = useSelector((state) => state.auth.user);

**🧠 Rule of Thumb:**

🔹 Use **local state by default**.  
 🔹 **Promote to Redux** only when:

* Multiple components need access
* You want persistent or tracked state
* You need middleware, async handling, or debugging tools

**17. Implement undo/redo functionality for a Redux-powered canvas app.**

**18. How would you persist Redux state across refreshes? What libraries can help?**

To **persist Redux state across page refreshes**, you need to **save the state to a persistent storage (like localStorage)** and **rehydrate it** when the app loads.

**✅ Solution: Use redux-persist**

**🔧 What It Does:**

* Automatically saves your Redux store to localStorage, sessionStorage, or AsyncStorage
* Rehydrates the store when the app starts
* Supports blacklisting/whitelisting specific slices

**🛠️ Steps to Persist Redux State**

**1. 📦 Install redux-persist**

bash

CopyEdit

npm install redux-persist

**2. 🧱 Wrap Your Redux Store**

js

CopyEdit

// store.js  
import { configureStore } from '@reduxjs/toolkit';  
import storage from 'redux-persist/lib/storage'; // defaults to localStorage  
import { persistReducer, persistStore } from 'redux-persist';  
import rootReducer from './rootReducer';  
  
const persistConfig = {  
 key: 'root',  
 storage,  
 whitelist: ['auth', 'cart'] // Only persist these slices  
};  
  
const persistedReducer = persistReducer(persistConfig, rootReducer);  
  
export const store = configureStore({  
 reducer: persistedReducer  
});  
  
export const persistor = persistStore(store);

**3. 🧩 Wrap <Provider> with <PersistGate>**

js

CopyEdit

import React from 'react';  
import ReactDOM from 'react-dom';  
import { Provider } from 'react-redux';  
import { PersistGate } from 'redux-persist/integration/react';  
import { store, persistor } from './store';  
import App from './App';  
  
ReactDOM.render(  
 <Provider store={store}>  
 <PersistGate loading={null} persistor={persistor}>  
 <App />  
 </PersistGate>  
 </Provider>,  
 document.getElementById('root')  
);

**🧠 When to Use It**

Use Redux persistence when:

* You want login/auth state to survive refresh
* You want cart/session data to stay available
* You don’t want to re-fetch data every reload

**📦 Other Libraries**

|  |  |
| --- | --- |
| **Library** | **Description** |
| redux-persist | Most popular, full-featured |
| redux-localstorage | Lightweight alternative |
| localForage + custom middleware | Async storage (IndexedDB) |

**19. Convert the below legacy Redux code to Redux Toolkit:**

**function counterReducer(state = 0, action) {**

**switch (action.type) {**

**case "INC": return state + 1;**

**default: return state;**

**}**

**}**

**20. Can you implement optimistic updates in Redux? Describe the pattern.**

**21. Explain the core concepts of Redux: Store, Action, Reducer. How does data flow through the system?**

*Classic question — answer should explain unidirectional flow clearly.*

**22. What problems does Redux Toolkit solve over traditional Redux?**

*Mention boilerplate reduction, createSlice, immer, built-in middleware.*

**23. What is createAsyncThunk? How does it work and when would you use it?**

*Understand async flow handling and integration with slices.*

**24. How do you manage loading, error, and success states using Redux Toolkit when performing async operations?**

*Describe pending, fulfilled, rejected action types and slice state structure.*

**25. How would you normalize data in Redux? Why is it important in large apps?**

*Describe createEntityAdapter or manual normalization with IDs.*

**26. What are some common performance issues in Redux and how can you optimize them?**

*Answer should include: selectors, memoization, avoiding state bloat, limiting re-renders.*

**27. What are selectors and why are they important in Redux? How do you create a memoized selector?**

*Describe use of reselect or createSelector.*

**28. What’s the difference between useSelector and connect()? When would you use one over the other?**

*Function components use useSelector; class components use connect().*

**29. How would you structure Redux in a large app with multiple domains (auth, dashboard, settings, etc.)?**

*Describe feature-based folders, modular slices, lazy loading reducers.*

**30. What is a Redux middleware? Can you give an example of a custom middleware you’ve written or used?**

*Describe logging, analytics, async handling, token injection, etc.*

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