PART 2

React Interview Questions & Answers

Complete Guide for Technical Interviews

Table of Contents

4. Hooks (Q51-65)

51. How does the useContext hook work?

The `useContext` hook allows you to subscribe to React context without introducing nesting. It takes a context object (created by `React.createContext`) and returns the current context value for that context.

```
const value = useContext(MyContext);
```

When the context value changes, components using `useContext` will re-render with the latest value.

52. What is the difference between useCallback and useMemo?

useCallback returns a memoized callback function, while **useMemo** returns a memoized value.

- `useCallback(fn, deps)` is equivalent to `useMemo(() => fn, deps)`
- `useCallback` optimizes functions, preventing unnecessary re-creations
- `useMemo` optimizes expensive computations by caching their results

53. When should you use useCallback?

Use `useCallback` when:

- Passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders
- The function is a dependency of other hooks like `useEffect`
- The function is used in expensive computations

54. When should you use useMemo?

Use `useMemo` when:

- Performing expensive calculations that don't need to run on every render
- Referential equality matters for objects/arrays passed as props
- Stabilizing values that are dependencies of other hooks

55. What is the useRef hook used for? How is it different from useState?

`useRef` returns a mutable ref object that persists for the component's lifetime. Unlike `useState`:

- Changing `useRef` doesn't trigger re-renders
- `useRef` values can be mutated directly via `.current`
- Common uses: accessing DOM elements, storing mutable values that don't affect rendering, keeping previous values

56. Can you invoke a hook inside a regular JavaScript function? Why?

No, hooks can only be called inside React functional components or custom hooks. This is because:

- Hooks rely on the React call order to properly manage state
- React maintains an internal "memory cell" for each component
- Calling hooks conditionally or in regular functions breaks the rules of hooks

57. How can you create your own custom hook? What is its naming convention?

A custom hook is a JavaScript function whose name starts with "use" and that may call other hooks.

```
function useCustomHook(initialValue) {
  const [value, setValue] = useState(initialValue);

  // Custom logic here

  return [value, setValue];
}
```

Naming convention: Always prefix with "use" so React can automatically check for rules of hooks violations.

58. What is the purpose of the useImperativeHandle hook?

`useImperativeHandle` customizes the instance value that is exposed to parent components when using `ref`. It should be used with `forwardRef`.

```
const MyComponent = forwardRef((props, ref) => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current.focus(),
    clear: () => inputRef.current.value = ''
  }));

  return □;
});
```

59. How does the useDebugValue hook help?

`useDebugValue` can be used to display a label for custom hooks in React DevTools. It helps with debugging by providing additional information about the hook's state.

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

60. Can you explain the mental model behind the "Stale Closure" problem in Hooks?

The stale closure problem occurs when a function captures variables from an outdated render. In hooks, this happens when:

- Callbacks inside `useEffect` or event handlers reference state/props from previous renders
- The dependency array is missing or incorrect
- Functions aren't properly memoized with `useCallback`

Solution: Include all dependencies in dependency arrays or use the functional update form of `useState`.

61. How does the useEffect cleanup function help prevent memory leaks?

The cleanup function runs before the component unmounts and before re-running the effect due to dependencies change. It helps prevent memory leaks by:

- Cancelling network requests
- Clearing timeouts/intervals
- Removing event listeners
- Cleaning up subscriptions

```
useEffect(() => {
  const subscription = props.source.subscribe();

  return () => {
    subscription.unsubscribe(); // Cleanup
  };
}, [props.source]);
```

62. Why is it important to include functions in the useEffect dependency array?

Functions should be included in the dependency array because:

- Functions defined inside components are recreated on every render
- Without proper dependencies, effects might use stale values from previous renders
- It ensures the effect runs with the latest function definition

Solution: Either move the function inside the effect or memoize it with `useCallback`.

63. How can you avoid having a function as a dependency in useEffect?

You can avoid function dependencies by:

- Moving the function definition inside the `useEffect`
- Using `useCallback` to memoize the function with proper dependencies
- Using the functional update form of `useState` when only the setter is needed
- Using `useRef` for mutable values that don't trigger re-renders

64. What is the "lazy initial state" pattern with useState?

The lazy initial state pattern passes a function to `useState` that calculates the initial state. This is useful when the initial state is expensive to compute.

```
// Expensive computation runs on every render:
const [state, setState] = useState(expensiveComputation());

// Expensive computation runs only once:
const [state, setState] = useState(() => expensiveComputation());
```

The function version runs only during the initial render, improving performance.

65. How would you implement a shouldComponentUpdate optimization in a functional component?

In functional components, you can optimize re-renders using:

- `React.memo` for props comparison (equivalent to PureComponent)
- `useMemo` for expensive computations
- `useCallback` for function props

```
const MyComponent = React.memo(function MyComponent(props) {
  // Component logic
}, (prevProps, nextProps) => {
  // Custom comparison function (like shouldComponentUpdate)
  return prevProps.id === nextProps.id;
});
```

5. Performance & Optimization (Q66-80)

66. What is the purpose of React.memo?

`React.memo` is a higher-order component that memoizes a functional component. It prevents unnecessary re-renders when props haven't changed, similar to `PureComponent` for class components.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

You can provide a custom comparison function as the second argument for more control over when to re-render.

67. How is React.memo different from useMemo?

React.memo memoizes an entire component based on its props, preventing re-renders when props don't change.

useMemo memoizes a computed value within a component, preventing expensive recalculations on every render.

```
// React.memo - component level
const ExpensiveComponent = React.memo(({ data }) => {
  return
{data}
;
});

// useMemo - value level
const Component = ({ items }) => {
  const expensiveValue = useMemo(() =>
    items.filter(item => item.isActive),
    [items]
  );

  return
{expensiveValue}
;
};
```

68. When should you not use React.memo?

Avoid `React.memo` when:

- The component frequently receives different props (memoization provides no benefit)
- The component is simple and cheap to render
- Props are complex objects that change frequently but with same values
- You're passing children as props (children are always new objects)

Memoization has a cost, so only use it when the performance benefit outweighs the cost.

69. What is code-splitting and how can you achieve it in React?

Code-splitting is a technique to split your bundle into smaller chunks that can be loaded on demand. In React, you can achieve it with:

- `React.lazy` for component-level splitting
- `import()` syntax for dynamic imports
- Route-based splitting with React Router

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

function MyComponent() {
  return (
    Loading...
}> ); }
```

70. What is the useCallback hook and how does it optimize performance?

`useCallback` returns a memoized version of a callback function that only changes if one of the dependencies has changed. It optimizes performance by:

- Preventing unnecessary re-renders of child components that depend on reference equality
- Reducing the need for garbage collection of function objects
- Stabilizing function references for dependency arrays

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b], // Recreate only if a or b changes
);
```

71. What is the useMemo hook and how does it optimize performance?

`useMemo` returns a memoized value that only recomputes when dependencies change. It optimizes performance by:

- Avoiding expensive calculations on every render
- Stabilizing object/array references to prevent unnecessary child re-renders
- Optimizing computations that depend on specific props or state

```
const expensiveValue = useMemo(() => {
  return expensiveComputation(a, b);
}, [a, b]); // Recompute only if a or b changes
```

72. What are the potential downsides of overusing useMemo and useCallback?

Overusing these hooks can lead to:

- **Memory overhead**: Storing memoized values/functions consumes memory
- **Complexity**: Code becomes harder to read and maintain
- **Premature optimization**: Optimizing before identifying actual bottlenecks
- **Dependency management issues**: Incorrect dependencies can cause bugs
- **Performance cost**: Memoization itself has computational cost

Use them only when you have measured performance issues.

73. How does lazy loading components with React.lazy and Suspense help performance?

`React.lazy` and `Suspense` help performance by:

- Reducing initial bundle size by splitting code
- Loading components only when they're needed
- Improving initial page load time
- Providing better user experience with loading states

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

function App() {
  return (


      Loading...
}>



); }
```

74. What is the "Virtual DOM" and how does it contribute to performance?

The **Virtual DOM** is a lightweight JavaScript representation of the actual DOM. It contributes to performance by:

- Batching multiple DOM updates into a single operation
- Minimizing direct DOM manipulation (which is expensive)
- Using efficient diffing algorithms to update only changed elements
- Providing a consistent programming model across browsers

When state changes, React creates a new Virtual DOM, compares it with the previous one, and efficiently updates the real DOM.

75. What are some common mistakes that lead to unnecessary re-renders?

Common mistakes include:

- Creating new objects/arrays in render without memoization
- Defining functions inside components without `useCallback`

- Not using keys properly in lists
- Mutating state directly instead of creating new references
- Incorrect dependency arrays in hooks
- Passing inline objects/functions as props

76. How can you identify performance bottlenecks in a React application?

You can identify bottlenecks using:

- **React DevTools Profiler**: Measures component rendering performance
- **Browser Performance tab**: Identifies JavaScript execution bottlenecks
- **Why did you render**: Library to detect unnecessary re-renders
- **Bundle analyzers**: Webpack Bundle Analyzer for bundle size issues
- **Lighthouse**: For overall performance audits

77. What is the "key" prop and how does it relate to performance?

The `key` prop helps React identify which items have changed, been added, or removed. It improves performance by:

- Enabling efficient reordering of list items
- Preventing unnecessary re-renders of unchanged items
- Maintaining component state correctly during list updates
- Reducing DOM operations during reconciliation

```
const TodoList = ({ todos }) => (


        {todos.map(todo => (
```

- `{todo.text}`

```
        ))}


);
```

78. What is the difference between a "production" and "development" build of React?

Development Build:

- Includes warnings, error messages, and dev tools
- Larger file size
- Slower performance
- Helpful for debugging

Production Build:

- Minified and optimized
- Smaller file size
- Faster performance
- Excludes development-only features

79. How can you optimize bundle size in a React application?

Bundle size optimization strategies:

- Code splitting with `React.lazy` and dynamic imports
- Tree shaking to remove unused code
- Using smaller alternative libraries
- Compression (gzip, Brotli)
- Bundle analysis to identify large dependencies
- Lazy loading non-critical components
- Using React's production build

80. What is "windowing" or "virtualization" and why is it used?

Windowing/virtualization is a technique that renders only the visible items in a large list, recycling DOM elements as the user scrolls. It's used to:

- Improve performance with large datasets
- Reduce DOM node count
- Prevent browser slowdowns
- Maintain smooth scrolling experience

> Popular libraries: `react-window`, `react-virtualized`.

6. Routing, State Management & Ecosystem (Q81-95)

81. What is the purpose of React Router?

React Router is a standard library for routing in React applications. It enables:

- Client-side routing without page refreshes
- Nested route configuration
- Programmatic navigation
- Route parameters and query strings
- Route-based code splitting

```
import { BrowserRouter, Route, Switch } from 'react-router-dom';

function App() {
  return (







  );
}
```

82. What is the difference between component and render prop in a React Router Route?

component prop: Passes a component directly, good for simple cases

render prop: Passes a function that returns JSX, good for inline rendering or passing additional props

```
 }
/>
```

children prop: Always renders, useful for animations or conditional rendering based on match

83. How do you handle "404 Not Found" pages in React Router?

You can handle 404 pages by:

- Adding a catch-all route at the end without a path
- Using the `Switch` component to render only the first matching route

```
function App() {
  return (





        {/* 404 Route - no path, matches everything */}



  );
}
```

84. What is the difference between client-side routing and server-side routing?

Client-side routing:

- Handled by JavaScript in the browser
- No full page reloads
- Faster navigation between views
- Better user experience
- Examples: React Router, Vue Router

Server-side routing:

- Browser requests a new page from server
- Full page reload on navigation
- Better for SEO (traditional approach)
- Slower navigation
- Examples: Express.js routes, Django URLs

85. What problem does a state management library like Redux solve?

Redux solves:

- **Prop drilling**: Passing props through multiple components
- **State synchronization**: Keeping multiple components in sync
- **Predictable state updates**: Enforcing unidirectional data flow
- **Debugging**: Time-travel debugging and state snapshots
- **Server-side rendering**: Consistent state initialization
- **Middleware**: Handling side effects consistently

86. What are the three principles of Redux?

1. Single Source of Truth: The global state is stored in a single store

2. State is Read-Only: State can only be changed by emitting actions

3. Changes are Made with Pure Functions: Reducers are pure functions that take previous state and action, return new state

87. What are "actions," "reducers," and "store" in Redux?

Actions: Plain JavaScript objects that describe what happened

```
{ type: 'ADD_TODO', payload: { text: 'Learn Redux' } }
```

Reducers: Pure functions that specify how state changes in response to actions

```
function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, action.payload];
    default:
      return state;
  }
}
```

Store: Object that brings actions and reducers together, holds application state

88. What is the difference between React's Context API and Redux?

Context API:

- Built into React
- Simpler setup

- Good for low-frequency updates
- No middleware support
- Less boilerplate

Redux:

- External library
- More powerful devtools
- Better for complex state logic
- Middleware ecosystem
- Time-travel debugging
- More boilerplate

89. When would you choose Context API over Redux?

Choose Context API when:

- Application is small to medium-sized
- State updates are infrequent
- You want to avoid external dependencies
- State structure is simple
- You don't need advanced features like middleware or time-travel
- You're passing down simple values or functions

90. What is "Thunk" in the context of Redux?

Redux Thunk is middleware that allows you to write action creators that return functions instead of actions. This enables:

- Async operations (API calls)
- Dispatching multiple actions
- Accessing current state
- Conditional dispatching

```
// Thunk action creator
const fetchUser = (userId) => {
  return (dispatch, getState) => {
    dispatch({ type: 'USER_FETCH_START' });

    fetch(`/api/users/${userId}`)
```

```
      .then(response => response.json())
      .then(user => {
        dispatch({ type: 'USER_FETCH_SUCCESS', payload: user });
      })
      .catch(error => {
        dispatch({ type: 'USER_FETCH_ERROR', payload: error });
      });
  };
};
```

91. What is the connect function in React-Redux?

`connect` is a higher-order component that connects a React component to the Redux store. It provides:

- Access to store state as props
- Ability to dispatch actions as props
- Automatic re-renders when relevant state changes

```
import { connect } from 'react-redux';

const TodoList = ({ todos, addTodo }) => (
  // Component JSX
);

const mapStateToProps = (state) => ({
  todos: state.todos
});

const mapDispatchToProps = {
  addTodo
};

export default connect(mapStateToProps, mapDispatchToProps)(TodoList);
```

92. What are the useSelector and useDispatch hooks in React-Redux?

These hooks provide a simpler alternative to `connect`:

useSelector: Extracts data from the Redux store state

```
const todos = useSelector(state => state.todos);
```

useDispatch: Returns a reference to the dispatch function

```
const dispatch = useDispatch();

const handleAddTodo = (text) => {
  dispatch(addTodo(text));
};
```

93. What is the purpose of the Provider component in React-Redux?

The `Provider` component makes the Redux store available to any nested components that need to access it. It uses React Context under the hood.

```
import { Provider } from 'react-redux';
import store from './store';

function App() {
  return (




  );
}
```

Any component inside `Provider` can connect to the store using `connect` or the Redux hooks.

94. What is "Immutability" and why is it important in Redux?

Immutability means not modifying existing objects/arrays, but creating new ones with changes. It's important in Redux because:

- Enables efficient change detection (reference comparison)
- Makes state predictable and debuggable
- Supports time-travel debugging
- Works well with React's rendering optimization

```
// MUTATION (bad)
state.todos.push(newTodo);

// IMMUTABLE UPDATE (good)
return {
  ...state,
  todos: [...state.todos, newTodo]
};
```

95. How can you perform asynchronous actions in Redux?

You can handle async actions using:

- **Redux Thunk**: Most popular, allows functions as actions

- **Redux Saga**: Uses generators for complex async flows
- **Redux Observable**: Uses RxJS observables
- **RTK Query**: Built-in data fetching in Redux Toolkit

```
// Using Redux Thunk
const fetchPosts = () => async (dispatch) => {
  try {
    dispatch({ type: 'POSTS_LOADING' });
    const response = await fetch('/api/posts');
    const posts = await response.json();
    dispatch({ type: 'POSTS_SUCCESS', payload: posts });
  } catch (error) {
    dispatch({ type: 'POSTS_ERROR', payload: error });
  }
};
```

7. Advanced Patterns & Miscellaneous (Q96-100)

96. What are Higher-Order Components (HOCs)?

A Higher-Order Component is a function that takes a component and returns a new component with additional props or functionality.

```
const withAuth = (WrappedComponent) => {
  return (props) => {
    const [isAuthenticated, setIsAuthenticated] = useState(false);

    // Authentication logic

    return isAuthenticated ?
       :
      ;
  };
};

const ProtectedComponent = withAuth(MyComponent);
```

HOCs are used for code reuse, logic abstraction, and props manipulation.

97. What is the "Render Props" pattern?

The render props pattern involves passing a function as a prop that returns React elements. This allows components to share code and state.

```
class MouseTracker extends React.Component {
  state = { x: 0, y: 0 };
  handleMouseMove = (event) => {
    this.setState({ x: event.clientX, y: event.clientY });
  };

  render() {
    return (


        {this.props.render(this.state)}


    );
  }
}

// Usage
 (
```

The mouse position is ({x}, {y})

```
)} />
```

This pattern is being replaced by custom hooks in many cases.

98. What are Error Boundaries in React?

Error Boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of crashing.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

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

  componentDidCatch(error, errorInfo) {
    console.log('Error caught by boundary:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return this.props.fallback ||
```

Something went wrong.

```
;
    }
    return this.props.children;
  }
}

// Usage
}>
```

99. Can error boundaries catch errors inside event handlers?

No, error boundaries do not catch errors in:

- Event handlers
- Asynchronous code (setTimeout, requestAnimationFrame callbacks)
- Server-side rendering
- Errors thrown in the error boundary itself

Error boundaries only catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

100. What is the StrictMode component used for?

`StrictMode` is a tool for highlighting potential problems in an application. It helps by:

- Identifying components with unsafe lifecycles
- Warning about legacy string ref API usage
- Detecting unexpected side effects
- Detecting legacy context API
- Ensuring reusable state (in React 18+)

```
function App() {
  return (



  );
}
```

StrictMode only runs in development and doesn't render any visible UI.