# Top 60 React.js Interview Questions for Front-End Developer Roles in 2025

## Table of Contents

*(Note: The UI will automatically generate the table of contents.)*

## Introduction: The State of React.js Interviews in 2025

In 2025, **React.js remains the primary front-end framework** in demand among tech companies, including industry giants and rising startups. The React ecosystem now features powerful additions such as React 19, Concurrent Mode, React Server Components, and an increasing adoption of TypeScript and advanced CSS-in-JS solutions. Front-end developer interviews have evolved accordingly, demanding not only **solid foundational knowledge** but also a deep understanding of hooks, state management, performance optimization, testing, integration with modern tools, TypeScript, accessibility, and security best practices1.

The following sections organize the top 60 React.js interview questions by topic, providing the *expected answer*, *explanation*, and *code snippets* where appropriate. This guide prepares candidates for interviews at leading companies by covering foundational, intermediate, and advanced concepts encountered in contemporary React recruitment processes.

## I. React Components & JSX

### 1. **What is a React component? What types of components exist?**

**Expected Answer:**  
A React component is a reusable, self-contained piece of UI. There are two main types: **Functional components** (the modern standard, using hooks for state and effects) and **Class components** (legacy, use ES6 classes for state and lifecycle).

**Explanation:**  
Functional components are plain JavaScript functions that return JSX. Class components use ES6 classes and must implement a render() method. Since React 16.8, functional components can manage state and use lifecycle features via hooks.

**Code Snippet:**

|  |
| --- |
| // Functional function Welcome({ name }) {  return <h1>Hello, {name}!</h1>; }  // Class class Welcome extends React.Component {  render() {  return <h1>Hello, {this.props.name}!</h1>;  } } |

### 2. **What is JSX and why is it used in React?**

**Expected Answer:**  
JSX stands for JavaScript XML. It allows developers to write HTML-like code within JavaScript, making UI code more readable and expressive.

**Explanation:**  
JSX is syntactic sugar for React.createElement() calls. It enables easy integration of JavaScript logic into visual UI structures, and is transpiled by Babel before browsers can interpret it1.

**Code Snippet:**

|  |
| --- |
| const element = <h1>Hello, world!</h1>; // Equivalent to: const element = React.createElement('h1', null, 'Hello, world!'); |

### 3. **How does conditional rendering work in React?**

**Expected Answer:**  
You can conditionally render elements using JavaScript operators like if/else, ternary, logical AND (&&), or switch within JSX.

**Code Snippet:**

|  |
| --- |
| function Greeting({ isLoggedIn }) {  return (  <div>  {isLoggedIn ? <p>Welcome!</p> : <p>Please log in.</p>}  </div>  ); } |

**Explanation:**  
React allows any valid JavaScript expression within JSX using curly braces {}. This enables flexible UI responses to state.

### 4. **What are props, and how are they passed between components?**

**Expected Answer:**  
Props (short for properties) are read-only inputs passed from a parent to a child component to make them dynamic and reusable.

**Explanation:**  
Props enable one-way data flow. They are immutable in the child and often used for configuration and data sharing.

**Code Snippet:**

|  |
| --- |
| function Child({ message }) {  return <p>{message}</p>; } <Child message="Hello" /> |

### 5. **What’s the difference between state and props in React?**

**Expected Answer:**

* **Props** are immutable inputs from a parent.
* **State** is mutable, local, and managed inside the component.

|  |  |  |
| --- | --- | --- |
|  | Props | State |
| Origin | Parent (external) | Component itself (internal) |
| Mutability | Read-only | Mutable |
| Usage | Functional & Class components | Class (via this.state), Function (via hooks) |

### 6. **How do you handle events in React?**

**Expected Answer:**  
Events are handled using camelCase event names (e.g., onClick), assigned to functions.

**Code Snippet:**

|  |
| --- |
| function Button() {  function handleClick() {  alert('Clicked!');  }  return <button onClick={handleClick}>Click Me</button>; } |

**Explanation:**  
React’s event system is built on **SyntheticEvent**, which normalizes events across browsers.

### 7. **What are controlled and uncontrolled components?**

**Expected Answer:**

* **Controlled components** have their value and changes managed by React state.
* **Uncontrolled components** rely on the DOM for value, accessed via refs.

**Code Snippet:**

|  |
| --- |
| // Controlled <input value={value} onChange={e => setValue(e.target.value)} />  // Uncontrolled <input ref={inputRef} /> |

**Explanation:**  
Controlled components allow for validation and state-driven UI. Uncontrolled components can simplify scenarios with minimal state management.

### 8. **What are keys in React lists, and why are they important?**

**Expected Answer:**  
A key is a special string attribute used to uniquely identify elements in lists. Keys help React identify which items have changed, been added, or removed for efficient updating.

**Code Snippet:**

|  |
| --- |
| {items.map(item => <li key={item.id}>{item.text}</li>)} |

**Explanation:**  
Never use index as key if the list can change order. Proper keys reduce unnecessary re-renders and bugs.

### 9. **What are React Fragments and why use them?**

**Expected Answer:**  
Fragments let you return multiple elements from a component without adding extra DOM nodes.

**Code Snippet:**

|  |
| --- |
| <>  <ChildA />  <ChildB /> </> |

**Explanation:**  
Fragments improve DOM structure and performance, especially in lists and layouts.

## II. Lifecycle Methods, Functional vs. Class Components

### 10. **What is the component lifecycle in React?**

**Expected Answer:**  
The lifecycle consists of *Mounting*, *Updating*, and *Unmounting* phases. Each phase triggers specific methods in class components.

|  |  |  |
| --- | --- | --- |
| Phase | Class Component Methods | Hook Alternative |
| Mounting | constructor, componentDidMount | useEffect(..., []) |
| Updating | shouldComponentUpdate, componentDidUpdate | useEffect([...deps]) |
| Unmounting | componentWillUnmount | useEffect cleanup |

**Explanation:**  
Hooks replicate lifecycle methods, making functional components fully featured.

### 11. **Describe the**

**Expected Answer:**

* **componentDidMount**: Runs once after initial render; used for fetching data, subscriptions.
* **componentDidUpdate**: Runs after every update (state/props); used for responding to changes.

**Code Snippet:**

|  |
| --- |
| // componentDidMount useEffect(() => {  // Runs once }, []);  // componentDidUpdate useEffect(() => {  // Runs any time [count] changes }, [count]); |

### 12. **How does the Virtual DOM work in React?**

**Expected Answer:**  
React creates a lightweight copy called the Virtual DOM. On state/props changes, it diffs the Virtual DOMs and updates the real DOM minimally.

**Explanation:**  
This **reconciliation** process drastically improves performance and UI consistency2.

## III. React Hooks

### 13. **What are React Hooks and why were they introduced?**

**Expected Answer:**  
Hooks are functions that allow functional components to use state, effects, refs, context, and more, introduced in React 16.8 to eliminate reliance on class components.

**Explanation:**  
Hooks promote code reuse, simpler component trees, and unlock advanced patterns without classes3.

### 14. **How does useState work? Provide an example.**

**Expected Answer:**  
useState returns a stateful value and a setter function.

**Code Snippet:**

|  |
| --- |
| const [count, setCount] = useState(0); // To update: setCount(count + 1); |

**Explanation:**  
State persistence and updates are tied to the render cycle.

### 15. **Explain useEffect with examples.**

**Expected Answer:**  
useEffect enables side effects (like data fetching, subscriptions, timers) after rendering. The second argument is the dependency array.

**Code Snippet:**

|  |
| --- |
| // Runs only on mount useEffect(() => {  document.title = 'Mounted!'; }, []);  // Runs on every change of count useEffect(() => {  document.title = `Count: ${count}`; }, [count]); |

### 16. **How does useContext work in React?**

**Expected Answer:**  
useContext allows a function component to access context values from the nearest provider above in the tree.

**Code Snippet:**

|  |
| --- |
| const ThemeContext = React.createContext('light'); function ThemedButton() {  const theme = useContext(ThemeContext);  return <button className={theme}>Click me</button>; } |

### 17. **When would you use useReducer instead of useState?**

**Expected Answer:**  
useReducer is suited for complex state logic involving multiple sub-values or when state transitions depend on the previous state. It works like a Redux reducer.

**Code Snippet:**

|  |
| --- |
| function reducer(state, action) {  switch (action.type) {  case 'increment': return { count: state.count + 1 };  default: throw new Error();  } } const [state, dispatch] = useReducer(reducer, { count: 0 }); |

### 18. **What is useCallback and when do you use it?**

**Expected Answer:**  
useCallback memoizes a function to prevent it from being recreated on every render, which is vital when passing callbacks to children to avoid unnecessary renders.

**Code Snippet:**

|  |
| --- |
| const memoizedHandler = useCallback(() => {  doSomething(); }, [dependency]); |

### 19. **What is useMemo and how does it differ from useCallback?**

**Expected Answer:**

* useMemo memoizes the **value** of the result of a function.
* useCallback memoizes the **function itself**.

**Code Snippet:**

|  |
| --- |
| const expensiveValue = useMemo(() => computeExpensiveValue(a, b), [a, b]); const memoizedCallback = useCallback(() => fn(a, b), [a, b]); |

### 20. **How do you create a custom hook? Give an example.**

**Expected Answer:**  
A custom hook is a function prefixed with use that may call other hooks, encapsulating reusable logic.

**Code Snippet:**

|  |
| --- |
| function useDocumentTitle(title) {  useEffect(() => {  document.title = title;  }, [title]); } // Usage useDocumentTitle('My Page'); |

### 21. **What are the rules of hooks?**

**Expected Answer:**

1. Call hooks only at the top level (not in loops, conditions, or nested functions).
2. Only call hooks from React function components or custom hooks.

**Explanation:**  
This ensures the order of hook calls remains consistent between renders.

### 22. **What’s the difference between useEffect and useLayoutEffect?**

**Expected Answer:**

* useEffect: Runs asynchronously after the browser has painted.
* useLayoutEffect: Runs synchronously after DOM mutations but **before** the browser paint. Use for layout reads/writes.

### 23. **How do you test components that use hooks?**

**Expected Answer:**  
Use @testing-library/react and Jest to render components and assert behavior. For hooks, use the @testing-library/react-hooks or wrap them in a test component.

**Code Snippet:**

|  |
| --- |
| import { render, fireEvent } from '@testing-library/react'; test('counter increments', () => {  const { getByText } = render(<Counter />);  fireEvent.click(getByText('Increment'));  expect(getByText('Count: 1')).toBeInTheDocument(); }); |

## IV. State Management

### 24. **What is the Context API, and when would you use it?**

**Expected Answer:**  
The Context API lets you pass data through the component tree without manually passing props at every level. It’s suited for global data like user info, theme, or locale.

**Code Snippet:**

|  |
| --- |
| const UserContext = React.createContext(null); function App() {  const [user, setUser] = useState('Alice');  return (  <UserContext.Provider value={{ user, setUser }}>  <Profile />  </UserContext.Provider>  ); } |

### 25. **What is Redux? Describe its key concepts.**

**Expected Answer:**  
Redux is a state management library for predictable, centralized app state. Key concepts:

* **Store**: Holds global state.
* **Actions**: Describe state changes.
* **Reducers**: Pure functions updating state based on actions.
* **Middleware**: Handles async/side effects (e.g., Redux Thunk).

**Code Snippet:**

|  |
| --- |
| const reducer = (state = 0, action) => action.type === 'INCREMENT' ? state + 1 : state; const store = createStore(reducer); store.dispatch({ type: 'INCREMENT' }); |

### 26. **When should you use Redux over Context API?**

**Expected Answer:**  
Use Redux for:

* Large applications with complex, frequently updated global state.
* When middleware, debugging, or scaling becomes essential.

Use Context API for:

* Small-/medium-sized apps.
* Simple global data sharing.

### 27. **What is prop drilling and how do you avoid it?**

**Expected Answer:**  
Prop drilling is passing data through multiple intermediary components. Avoid with the Context API or state management libraries (Redux, Zustand, MobX).

### 28. **What is lifting state up? Provide an example.**

**Expected Answer:**  
Lifting state up means moving shared state to the nearest common ancestor of components that both rely on the data.

**Code Snippet:**

|  |
| --- |
| function Parent() {  const [text, setText] = useState('');  return <>  <Input onChange={setText} />  <Display value={text} />  </>; } |

### 29. **How do you combine multiple reducers in Redux?**

**Expected Answer:**  
Use combineReducers to compose several reducers, each managing part of the state.

**Code Snippet:**

|  |
| --- |
| import { combineReducers } from 'redux'; const rootReducer = combineReducers({  user: userReducer,  posts: postReducer }); |

## V. Performance Optimization

### 30. **How do you optimize performance in a React app? List techniques.**

**Expected Answer:**

* Use React.memo for function components.
* Use useMemo and useCallback for expensive calculations and stable function references.
* Lazy load components with React.lazy and Suspense.
* Code-splitting for large apps.
* Properly use keys in lists.
* Virtualize long lists with libraries like react-window.
* Keep render trees shallow.

### 31. **What is React.memo and when should you use it?**

**Expected Answer:**  
React.memo is a higher-order component that memoizes a functional component, preventing unnecessary re-renders if props haven’t changed.

**Code Snippet:**

|  |
| --- |
| const MyComponent = React.memo(function MyComponent(props) {  // Will only rerender if props change }); |

### 32. **How can you prevent unnecessary re-renders?**

**Expected Answer:**

* Use React.memo for components.
* Use useMemo and useCallback.
* Pass stable prop references.
* Use selectors for computed props in Redux.

### 33. **What is code splitting and lazy loading? How are they implemented in React?**

**Expected Answer:**  
Code splitting divides your code into smaller bundles loaded on demand. Lazy loading defers component loading until needed.

**Code Snippet:**

|  |
| --- |
| const LazyComponent = React.lazy(() => import('./LazyComponent')); // Usage: <Suspense fallback={<p>Loading...</p>}>  <LazyComponent /> </Suspense> |

### 34. **How do you optimize rendering of large lists?**

**Expected Answer:**  
Use virtualization libraries (e.g., react-window, react-virtualized) to render only visible items.

**Code Snippet:**

|  |
| --- |
| import { FixedSizeList as List } from 'react-window'; <List height={400} itemCount={1000} itemSize={35} width={300}>  {({ index, style }) => <div style={style}>Row {index}</div>} </List> |

## VI. Testing React Applications

### 35. **How do you test React components?**

**Expected Answer:**

* Use Jest or Mocha as a test runner.
* Use @testing-library/react to render and interact with components.
* Use snapshot testing with react-test-renderer.

**Code Snippet:**

|  |
| --- |
| import { render, fireEvent } from '@testing-library/react'; test('increments counter', () => {  const { getByText } = render(<Counter />);  fireEvent.click(getByText('Increment'));  expect(getByText('Count: 1')).toBeInTheDocument(); }); |

### 36. **How do you test asynchronous operations in React?**

**Expected Answer:**  
Use async/await and waitFor utilities from the testing library.

**Code Snippet:**

|  |
| --- |
| import { render, waitFor } from '@testing-library/react'; test('fetches and displays data', async () => {  const { getByText } = render(<FetchComponent />);  await waitFor(() => getByText('Data Loaded')); }); |

### 37. **What is snapshot testing and why is it used?**

**Expected Answer:**  
Snapshot testing captures the rendered output of a component. Future changes are compared to this snapshot to detect unintended changes.

**Code Snippet:**

|  |
| --- |
| import renderer from 'react-test-renderer'; test('matches snapshot', () => {  const tree = renderer.create(<MyComponent />).toJSON();  expect(tree).toMatchSnapshot(); }); |

### 38. **How do you mock modules or APIs in Jest?**

**Expected Answer:**  
Use jest.mock() to create mock versions of modules or functions, such as network requests.

**Code Snippet:**

|  |
| --- |
| jest.mock('./api'); import { getData } from './api'; getData.mockResolvedValue({ foo: 'bar' }); |

### 39. **How do you test Redux actions and reducers with Jest?**

**Expected Answer:**  
Dispatch actions and assert the new state, and verify action creators return correct action objects.

**Code Snippet:**

|  |
| --- |
| expect(reducer({ count: 0 }, { type: 'INCREMENT' })).toEqual({ count: 1 }); expect(incrementAction()).toEqual({ type: 'INCREMENT' }); |

## VII. Integration with Other Technologies

### 40. **Explain CORS and how it affects React development.**

**Expected Answer:**  
CORS (Cross-Origin Resource Sharing) controls HTTP requests across different domains, affecting how React apps interact with APIs on other servers. Use the fetch API or axios, and set backend headers properly4.

### 41. **How do you use axios in a React app?**

**Expected Answer:**  
Axios is a promise-based HTTP client. Install via npm i axios. Use it for network requests in useEffect or event handlers.

**Code Snippet:**

|  |
| --- |
| import axios from 'axios'; useEffect(() => {  axios.get('/api/data').then(response => setData(response.data)); }, []); |

### 42. **What is React Material UI?**

**Expected Answer:**  
Material UI is a React component framework implementing Google's Material Design, with reusable, customizable UI components.

### 43. **How do you implement routing in React?**

**Expected Answer:**  
Use React Router’s <BrowserRouter>, <Route>, <Switch>, and link components.

**Code Snippet:**

|  |
| --- |
| import { BrowserRouter, Route, Link } from "react-router-dom"; <BrowserRouter>  <nav>  <Link to="/about">About</Link>  </nav>  <Route path="/about" component={AboutPage} /> </BrowserRouter> |

### 44. **How do you integrate TypeScript into a React project?**

**Expected Answer:**  
Use Create React App with TypeScript template or add TypeScript via npm i -D typescript @types/react. Use .tsx files for components.

**Code Snippet:**

|  |
| --- |
| type MyButtonProps = { label: string }; const MyButton: React.FC<MyButtonProps> = ({ label }) => <button>{label}</button>; |

## VIII. Advanced Patterns and Architectures

### 45. **What are Higher Order Components (HOC)? Give an example.**

**Expected Answer:**  
An HOC is a function that takes a component and returns a new component with extra props or logic, used for code reuse or features.

**Code Snippet:**

|  |
| --- |
| function withLogger(Wrapped) {  return function(props) {  console.log(props);  return <Wrapped {...props} />;  } } |

### 46. **What is the render props pattern in React?**

**Expected Answer:**  
A render prop is a function prop that returns JSX. It enables sharing logic by letting a component delegate rendering.

**Code Snippet:**

|  |
| --- |
| <MyComponent render={data => <div>{data}</div>} /> |

### 47. **What are portals and what are their use cases?**

**Expected Answer:**  
Portals let you render children into a DOM node outside the parent hierarchy. Used for modals, tooltips, etc.

**Code Snippet:**

|  |
| --- |
| ReactDOM.createPortal(child, document.getElementById('modal-root')); |

### 48. **What is the React Fiber architecture, and why is it important?**

**Expected Answer:**  
Fiber, React's core reconciliation engine since v16, enables efficient, incremental rendering, prioritization of updates, and concurrency.

### 49. **What are Pure Components?**

**Expected Answer:**  
Pure Components only re-render if their props or state change, using shallow comparison. For function components, use React.memo.

## IX. Server-Side Rendering, SSG, and Modern React

### 50. **What is server-side rendering (SSR) in React, and what are its benefits?**

**Expected Answer:**  
SSR is rendering components on the server, sending fully rendered HTML to the client, improving SEO and initial load time. Libraries like Next.js simplify SSR.

### 51. **Compare static site generation (SSG) and SSR.**

**Expected Answer:**

|  |  |  |
| --- | --- | --- |
|  | SSR | SSG |
| Timing | On each request | At build time (pre-generated) |
| Performance | Slower per request | Fast (static HTML) |
| Use-case | Dynamic content | Static content |

### 52. **How do you use getStaticProps and getServerSideProps in Next.js?**

**Expected Answer:**

* getStaticProps: Generate static pages at build time.
* getServerSideProps: Fetch and render data on each server request.

**Code Snippet:**

|  |
| --- |
| export async function getServerSideProps() {  // Fetch data  return { props: { data } }; } |

## X. TypeScript with React

### 53. **How do you define prop types for a React component using TypeScript?**

**Expected Answer:**  
Use interfaces or type aliases, and React.FC<Props>.

**Code Snippet:**

|  |
| --- |
| interface ButtonProps {  label: string;  onClick: () => void; } const Button: React.FC<ButtonProps> = ({ label, onClick }) => (  <button onClick={onClick}>{label}</button> ); |

### 54. **How can you use generics in React components with TypeScript?**

**Expected Answer:**  
Generics allow you to create components that accept multiple types of props or children.

**Code Snippet:**

|  |
| --- |
| type ListProps<T> = { items: T[]; render: (item: T) => React.ReactNode; }; function List<T>({ items, render }: ListProps<T>) {  return <ul>{items.map(render)}</ul>; } |

### 55. **How does TypeScript improve the scalability and maintainability of React apps?**

**Expected Answer:**  
TypeScript statically checks prop types, prevents runtime errors, improves refactoring, enables advanced typing (e.g., union, intersection types), and supports tooling for autocomplete and documentation.

## XI. Styling in React

### 56. **What are the main ways to style React components?**

**Expected Answer:**

* **Inline styles**: { color: 'red' }
* **CSS classes/stylesheets**: import './App.css'
* **CSS Modules**: Scoped, imported as objects.
* **Styled-components** or Emotion: CSS-in-JS libraries supporting dynamic, scoped, and theme-based styles.

### 57. **How does styled-components work, and what are its advantages?**

**Expected Answer:**  
Styled-components lets you define styled elements using template literals inside JS. It scopes CSS, supports theming, dynamic props, and SSR.

**Code Snippet:**

|  |
| --- |
| import styled from 'styled-components'; const Button = styled.button`  color: white;  background: ${({ primary }) => (primary ? 'blue' : 'gray')}; `; |

### 58. **How do you apply conditional styling in React?**

**Expected Answer:**  
Use ternaries in className or style assignments, dynamic styled-component props, or the classnames utility.

**Code Snippet:**

|  |
| --- |
| <button className={isActive ? 'active' : ''}>Click</button> |

### 59. **How does CSS-in-JS differ from CSS modules?**

**Expected Answer:**  
CSS-in-JS (styled-components, Emotion) colocates style logic in JS, enables dynamic styling, and themes. CSS modules keep CSS in .module.css files, ensuring local scoping and static extraction.

## XII. Accessibility in React

### 60. **What are key accessibility practices in React applications?**

**Expected Answer:**

* Use semantic HTML (e.g., <button>, <nav>)
* Provide alt text for images
* Ensure keyboard navigability (tab index, focus states)
* Use ARIA attributes where necessary
* Test with screen readers and accessibility tools (axe, Lighthouse)
* Maintain color contrast ratios

**Code Snippet Example:**

|  |
| --- |
| <button aria-label="Close" onClick={closeModal}>×</button> <img src="/logo.png" alt="Company Logo" /> <label htmlFor="email">Email:</label> <input id="email" /> |

**Explanation:**  
Accessibility not only improves user experience for users with disabilities but also broadens your product's reach and complies with legal requirements.

## XIII. Security Best Practices

### Extra Security Question: **How do you prevent XSS in React?**

**Expected Answer:**

* React escapes content by default.
* Never use dangerouslySetInnerHTML with unsanitized user input.
* Sanitize all user data before rendering.
* Set strong Content Security Policy (CSP) headers.

**Example:**

|  |
| --- |
| <div>{userInput}</div> // Safe // Dangerous:  <div dangerouslySetInnerHTML={{ \_\_html: userInput }} /> // Unsafe if unsanitized! |

**Security Headers Example:**

|  |
| --- |
| <meta http-equiv="Content-Security-Policy" content="script-src 'self';" /> |

**Explanation:**  
React encodes HTML entities by default, protecting against most XSS vectors unless you bypass with dangerouslySetInnerHTML5.

# Conclusion

Mastering the above 60 questions, their explanations, and code patterns will prepare candidates for **front-end developer roles specializing in React.js** at top companies in 2025. This coverage spans **component patterns, hooks, state management, performance, testing, integration, TypeScript, styling, accessibility**, and **security**-the knowledge domains most often probed during modern React interviews367.

Candidates should supplement theoretical knowledge by building real-world projects, contributing to open-source React tools, and staying current with the React documentation, as the landscape continues to evolve swiftly.

**Note:**  
For topic-specific deep-dives-such as Next.js SSR, advanced TypeScript typing, or cutting-edge state libraries-reference official React documentation or leading interview question repositories for up-to-date examples and best practices.897

# References (14)

1. *Top 55 React JS Interview Questions and Answers (2025)*. <https://tutedude.com/blogs/react-js-interview-questions/>

2. *ReactJS: Top 10 Virtual DOM in ReactJS interview questions*. <https://interviewer.live/reactjs/web-development-reactjs/reactjs-virtual-dom-interview-questions/>

3. *React Hooks Interview Questions & Answers - 2025 - GeeksforGeeks*. <https://www.geeksforgeeks.org/reactjs/top-react-hooks-interview-questions-answers/>

4. *React Interview Questions and Answers - GeeksforGeeks*. <https://www.geeksforgeeks.org/reactjs/react-interview-questions/>

5. *XSS Prevention Guide: Complete Tutorial - Techoral*. <https://techoral.com/security/xss-prevention.html>

7. *Using CSP to reinforce your React application against XSS attacks*. <https://stackoverflow.com/collectives/articles/75138763/using-csp-to-reinforce-your-react-application-against-xss-attacks>

6. *Essential React JS Accessibility Checklist for Developers*. <https://moldstud.com/articles/p-essential-react-js-accessibility-checklist-for-developers>

8. *Top 50 TypeScript Interview Questions and Answers in 2025 - Edureka*. <https://www.edureka.co/blog/interview-questions/typescript-interview-questions/>

9. *20 Styled Components Interview Questions and Answers - CLIMB*. <https://climbtheladder.com/styled-components-interview-questions/>