React Interview Questions & Answers

Author: PrepForHire
Level: Beginner to Advanced
Last Updated: January 2026

This PDF contains a curated collection of **React interview questions and answers** covering fundamentals, hooks, state management, Redux, routing, performance optimization, and advanced React patterns commonly asked in real-world interviews.

1. FUNDAMENTALS

Q1.1. How is styling applied to React components?

Styling in React components can be applied using external CSS files, inline styles defined as JavaScript objects, CSS Modules, or CSS-in-JS libraries. At a fundamental level, React supports traditional CSS via className and inline styles via the style prop, allowing developers to choose between static and dynamic styling approaches.

```
<div className="container" style={{ padding: "16px" }}>Content</div>
```

Why interviewers ask this:
Interviewers ask this to evaluate whether the candidate understands the basic styling mechanisms available in React and their appropriate usage.

Common mistakes:

- Assuming React introduces a completely new styling system instead of building on standard CSS.

Q1.2. Why does React use className instead of class for CSS classes?

React uses className instead of class because class is a reserved keyword in JavaScript. JSX maps className to the standard HTML class attribute at runtime while avoiding conflicts with JavaScript syntax.

```
<div className="card">Card Content</div>
```

Why interviewers ask this:
This checks foundational JSX knowledge and understanding of how JSX integrates JavaScript and HTML.

Common mistakes:

- Using class instead of className and relying on browser behavior to fix it.

Q1.3. How are inline styles written in React?

Inline styles in React are written as JavaScript objects rather than strings. CSS property names use camelCase instead of kebab-case, and values are usually strings or numbers. This allows styles to be computed dynamically using JavaScript logic.

```
<div style={{ backgroundColor: "blue", fontSize: 16 }}>Text</div>
```

Why interviewers ask this:
Interviewers use this to assess comfort with JSX syntax and JavaScript object usage.

Common mistakes:

- Writing inline styles as strings or using kebab-case property names.

Q1.4. How can styles be applied conditionally in React?

Conditional styling in React is typically achieved by dynamically changing class names or inline style values based on component state or props. This allows the UI to react visually to user interaction, application state, or business logic.

```
<div className={isActive ? "active" : "inactive"}>Status</div>
```

Why interviewers ask this:
This tests understanding of declarative UI updates and dynamic rendering.

Common mistakes:

- Embedding conditional logic directly inside CSS files instead of JSX.

Q1.5. What are the advantages and disadvantages of inline styles in React?

Inline styles provide dynamic styling capabilities and avoid CSS naming conflicts, but they lack support for pseudo-classes, media queries, and can reduce separation of concerns. They are best used for dynamic values rather than large style definitions.

Why interviewers ask this:
Interviewers want to see trade-off analysis rather than tool memorization.

Common mistakes:

- Using inline styles for complex layouts or hover effects.

Q1.6. How does React handle CSS specificity and conflicts?

React does not alter CSS specificity rules. Styles applied via className follow standard CSS cascading and specificity rules, while inline styles have higher priority than external styles. Understanding this is critical to predicting how styles are applied in complex applications.

Why interviewers ask this:
This reveals whether candidates understand that React does not abstract CSS behavior.

Common mistakes:

- Assuming React automatically resolves CSS conflicts.

Q1.7. Why is inline styling sometimes discouraged in large React applications?

In large applications, excessive inline styling can lead to poor maintainability, duplicated style logic, and reduced readability. It also prevents the use of advanced CSS features such as media queries and pseudo-selectors, making responsive design harder to manage.

Why interviewers ask this:
Interviewers ask this to assess architectural thinking and scalability concerns.

Common mistakes:

- Believing inline styles are inherently bad rather than context-dependent.

Q1.8. How can dynamic styling impact React performance?

Dynamic styling can trigger frequent re-renders if styles are recalculated on every render. When not memoized or optimized, this can lead to unnecessary updates. Using stable class names and memoization techniques helps mitigate performance issues.

Why interviewers ask this:
Tests performance awareness beyond visual correctness.

Common mistakes:

- Creating new style objects on every render without memoization.

Q1.9. How does React compare inline styles with traditional CSS during reconciliation?

Inline styles are treated as part of the element’s props. During reconciliation, React performs a shallow comparison of style objects and updates only the properties that change. Stable references help React minimize DOM mutations.

Why interviewers ask this:
This question targets understanding of reconciliation and rendering optimization.

Common mistakes:

- Assuming inline styles always cause full re-renders.

Q1.10. When should CSS files be preferred over inline styles in React?

CSS files should be preferred when styles are static, reused across components, or require advanced features like media queries, animations, or pseudo-classes. Inline styles are best reserved for dynamic values driven by state or props.

Why interviewers ask this:
Interviewers want practical decision-making, not absolute rules.

Common mistakes:

- Overusing inline styles due to convenience rather than design considerations.

Q1.11. How are events handled in React?

React uses a synthetic event system that wraps native browser events, providing a consistent interface across different browsers.

```
<button onClick={handleClick}>Click</button>
```

Why interviewers ask this:
Interviewers want to verify knowledge of React's event abstraction.

Common mistakes:

- Assuming React uses native DOM events directly.

Q1.12. Why do we pass functions instead of function calls to event handlers?

Passing a function reference prevents it from executing immediately during render. React calls the function only when the event occurs.

```
onClick={handleClick}
```

Why interviewers ask this:
Tests understanding of rendering vs event execution.

Common mistakes:

- Calling the function directly inside JSX.

Q1.13. How do you pass arguments to event handlers in React?

Arguments can be passed using arrow functions or by binding the function with parameters.

```
onClick={() => handleDelete(id)}
```

Why interviewers ask this:
Checks JSX and function binding knowledge.

Common mistakes:

- Using inline function calls incorrectly.

Q1.14. What is conditional rendering in React?

Conditional rendering allows components or elements to be rendered based on certain conditions using JavaScript expressions.

```
{isLoggedIn && <Dashboard />}
```

Why interviewers ask this:
Validates control flow understanding in JSX.

Common mistakes:

- Trying to use if statements directly inside JSX.

Q1.15. What are the common patterns used for conditional rendering?

Common patterns include ternary operators, logical AND (&&), switch statements outside JSX, and early returns.

```
{isLoading ? <Loader /> : <Content />}
```

Why interviewers ask this:
Tests awareness of readable rendering patterns.

Common mistakes:

- Overusing nested ternary expressions.

Q1.16. How does React handle event pooling?

In older versions of React, synthetic events were pooled and reused for performance. Modern React no longer pools events.
Why interviewers ask this:
Checks version awareness and internal knowledge.

Common mistakes:

- Accessing event properties asynchronously without persistence.

Q1.17. How can you prevent default browser behavior in React events?

Calling event.preventDefault() prevents the browser's default action, such as form submission or link navigation.

```
const handleSubmit = (e) => { e.preventDefault(); }
```

Why interviewers ask this:
Verifies handling of forms and links.

Common mistakes:

- Forgetting to pass the event object.

Q1.18. Why should event handlers be kept lightweight?

Heavy logic inside event handlers can cause performance issues and reduce readability. Business logic should be abstracted into functions or hooks.
Why interviewers ask this:
Assesses performance and architecture awareness.

Common mistakes:

- Putting API calls and complex logic directly in JSX.

Q1.19. How does conditional rendering affect component lifecycle?

Conditionally rendered components mount when the condition becomes true and unmount when it becomes false, triggering lifecycle hooks or effects.
Why interviewers ask this:
Tests understanding of component mount and unmount behavior.

Common mistakes:

- Assuming hidden components remain mounted.

Q1.20. What are common performance issues related to conditional rendering?

Unnecessary mounting and unmounting of components can impact performance. Memoization and conditional checks can help optimize rendering.

Why interviewers ask this:

Evaluates performance optimization thinking.

Common mistakes:

- Rendering large components conditionally without optimization.

Q1.21. What is a functional component in React?

A functional component is a JavaScript function that returns JSX and represents a reusable UI unit. It receives props as arguments and must be a pure function, meaning it returns the same UI for the same input.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Why interviewers ask this:

This question checks whether the candidate understands the modern and preferred way of building components in React.

Common mistakes:

- Confusing functional components with regular functions
- Believing class components are mandatory
- Not understanding props

Q1.22. What is a functional component in React?

A functional component is a JavaScript function that returns JSX and represents a reusable UI unit. It receives props as arguments and must be a pure function, meaning it returns the same UI for the same input.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Why interviewers ask this:

This question checks whether the candidate understands the modern and preferred way of building components in React.

Common mistakes:

- Confusing functional components with regular functions
- Believing class components are mandatory
- Not understanding props

Q1.23. How are props passed to a functional component?

Props are passed as arguments to a functional component. They are read-only and allow data to flow from parent components to child components.

```
function Welcome({ name }) {
  return <h1>Hello {name}</h1>;
}
```

Why interviewers ask this:

Interviewers use this to evaluate understanding of unidirectional data flow in React.

Common mistakes:

- Trying to modify props
- Confusing props with state
- Accessing props incorrectly

Q1.24. How are props passed to a functional component?

Props are passed as arguments to a functional component. They are read-only and allow data to flow from parent components to child components.

```
function Welcome({ name }) {
  return <h1>Hello {name}</h1>;
}
```

Why interviewers ask this:
Interviewers use this to evaluate understanding of unidirectional data flow in React.

Common mistakes:

- Trying to modify props
- Confusing props with state
- Accessing props incorrectly

Q1.25. What does it mean that functional components should be pure?

A pure functional component does not modify external variables or cause side effects during rendering. For the same props and state, it always returns the same JSX.

Why interviewers ask this:
This tests conceptual understanding of React's rendering philosophy and predictability.

Common mistakes:

- Performing API calls inside render
- Mutating global variables
- Depending on external mutable data

Q1.26. What does it mean that functional components should be pure?

A pure functional component does not modify external variables or cause side effects during rendering. For the same props and state, it always returns the same JSX.

Why interviewers ask this:
This tests conceptual understanding of React's rendering philosophy and predictability.

Common mistakes:

- Performing API calls inside render
- Mutating global variables
- Depending on external mutable data

Q1.27. How is state managed in functional components?

State in functional components is managed using Hooks such as useState. Hooks allow functional components to hold and manage internal state.

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

Why interviewers ask this:
This question verifies whether the candidate understands the evolution from class components to Hooks.

Common mistakes:

- Thinking state is only for class components
- Updating state directly
- Misunderstanding re-renders

Q1.28. How is state managed in functional components?

State in functional components is managed using Hooks such as useState. Hooks allow functional components to hold and manage internal state.

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

Why interviewers ask this:
This question verifies whether the candidate understands the evolution from class components to Hooks.

Common mistakes:

- Thinking state is only for class components
- Updating state directly
- Misunderstanding re-renders

Q1.29. What are Hooks and why are they used with functional components?

Hooks are functions that allow functional components to use React features such as state, lifecycle behavior, and context without using class components.

```
useEffect(() => {
  document.title = count;
}, [count]);
```

Why interviewers ask this:
Interviewers ask this to assess understanding of modern React patterns.

Common mistakes:

- Using hooks conditionally
- Calling hooks inside loops
- Assuming hooks replace all logic

Q1.30. What are Hooks and why are they used with functional components?

Hooks are functions that allow functional components to use React features such as state, lifecycle behavior, and context without using class components.

```
useEffect(() => {
  document.title = count;
}, [count]);
```

Why interviewers ask this:
Interviewers ask this to assess understanding of modern React patterns.

Common mistakes:

- Using hooks conditionally
- Calling hooks inside loops
- Assuming hooks replace all logic

Q1.31. Why can hooks only be called at the top level of a functional component?

Hooks rely on consistent call order between renders. Calling them conditionally or inside loops breaks this order and causes unpredictable behavior.

Why interviewers ask this:
This checks whether the candidate understands React's internal hook mechanism.

Common mistakes:

- Calling hooks inside if blocks
- Calling hooks inside nested functions
- Ignoring hook rules

Q1.32. Why can hooks only be called at the top level of a functional component?

Hooks rely on consistent call order between renders. Calling them conditionally or inside loops breaks this order and causes unpredictable behavior.

Why interviewers ask this:

This checks whether the candidate understands React's internal hook mechanism.

Common mistakes:

- Calling hooks inside if blocks
- Calling hooks inside nested functions
- Ignoring hook rules

Q1.33. How do functional components differ from class components internally?

Function components rely on hooks and closures, while class components rely on instances and lifecycle methods. Function components are simpler and easier to optimize.

Why interviewers ask this:

Interviewers use this to separate surface-level React users from those who understand architectural differences.

Common mistakes:

- Thinking functional components are slower
- Believing lifecycle methods still exist in functional components

Q1.34. How do functional components differ from class components internally?

Function components rely on hooks and closures, while class components rely on instances and lifecycle methods. Function components are simpler and easier to optimize.

Why interviewers ask this:

Interviewers use this to separate surface-level React users from those who understand architectural differences.

Common mistakes:

- Thinking functional components are slower
- Believing lifecycle methods still exist in functional components

Q1.35. How does React handle re-rendering of functional components?

Function components re-render whenever their props or state change. React compares the previous and next virtual DOM trees and updates only the necessary parts of the real DOM.

Why interviewers ask this:

This question evaluates understanding of React's rendering and reconciliation behavior.

Common mistakes:

- Assuming full DOM re-render
- Not understanding dependency-driven renders
- Overusing memoization

Q1.36. How does React handle re-rendering of functional components?

Function components re-render whenever their props or state change. React compares the previous and next virtual DOM trees and updates only the necessary parts of the real DOM.

Why interviewers ask this:

This question evaluates understanding of React's rendering and reconciliation behavior.

Common mistakes:

- Assuming full DOM re-render
- Not understanding dependency-driven renders
- Overusing memoization

Q1.37. What is JSX in React?

JSX (JavaScript XML) is a syntax extension for JavaScript that allows writing HTML-like structures inside JavaScript. JSX is not HTML; it is transpiled into React.createElement() calls during build time, making UI code more readable and declarative.

Why interviewers ask this:

Interviewers use this question to verify whether the candidate understands that JSX is a syntax abstraction and not actual HTML. It tests conceptual clarity around how React elements are created and rendered.

Common mistakes:

- Assuming JSX is HTML
- Believing browsers understand JSX directly
- Not knowing JSX compiles to React.createElement

Q1.38. Is JSX mandatory in React?

No, JSX is not mandatory. React can be written using React.createElement() directly. JSX is recommended because it improves readability and developer experience.

Why interviewers ask this:

This question evaluates whether the candidate understands that JSX is optional and syntactic sugar. It also reveals familiarity with React's core API.

Common mistakes:

- Claiming JSX is required
- Not knowing React.createElement exists
- Thinking JSX affects performance

Q1.39. How does JSX get converted into JavaScript?

JSX is transpiled by Babel into React.createElement() calls. Browsers do not understand JSX directly; only the compiled JavaScript is executed.

```
React.createElement("h1", null, "Hello World");
```

Why interviewers ask this:

Interviewers ask this to test build-tool awareness and understanding of how JSX actually runs in the browser.

Common mistakes:

- Thinking JSX runs directly in the browser
- Confusing Babel with React
- Assuming JSX is a template engine

Q1.40. Why do we use className instead of class in JSX?

In JavaScript, class is a reserved keyword. JSX uses className to avoid conflicts while still mapping correctly to the HTML class attribute.

```
<div className="container">Content</div>
```

Why interviewers ask this:

This is a classic React syntax check used to see if candidates understand JSX's relationship with JavaScript.

Common mistakes:

- Using class instead of className
- Believing this is a React limitation rather than JavaScript

Q1.41. How can JavaScript expressions be embedded inside JSX?

JavaScript expressions can be embedded using curly braces {}. Only expressions are allowed, not statements like if or for.

```
const name = "React";
<h1>Hello {name}</h1>
```

Why interviewers ask this:
This question checks whether the candidate understands the boundary between JavaScript expressions and statements inside JSX.

Common mistakes:

- Trying to use if or for statements directly in JSX
- Overusing logic inside JSX instead of preprocessing

Q1.42. How does conditional rendering work in JSX?

JSX does not support if-else directly. Conditional rendering is achieved using ternary operators, logical AND (&&), or external logic.

```
{isLoggedIn ? <Dashboard /> : <Login />}
```

Why interviewers ask this:
Interviewers want to evaluate how candidates handle dynamic UI logic in a declarative framework.

Common mistakes:

- Using if statements inside JSX
- Creating deeply nested ternaries
- Ignoring readability

Q1.43. Why must JSX return a single parent element?

JSX must return a single root element because React.createElement() returns only one element. Multiple elements must be wrapped in a container or React.Fragment.

```
<>
  <Header />
  <Footer />
</>
```

Why interviewers ask this:
This tests understanding of how JSX maps to React element trees and why fragments exist.

Common mistakes:

- Wrapping everything in unnecessary divs
- Not knowing about React.Fragment or shorthand syntax

Q1.44. How are inline styles applied in JSX?

Inline styles in JSX are written as JavaScript objects with camelCase property names instead of kebab-case.

```
<div style={{ backgroundColor: "blue", fontSize: "16px" }}>Text</div>
```

Q1.45. How does JSX help prevent XSS attacks?

React automatically escapes values embedded in JSX before rendering, preventing malicious scripts from executing and protecting against XSS attacks.

Why interviewers ask this:
Security-focused interviews use this question to assess awareness of built-in XSS protections and safe rendering practices.

Common mistakes:

- Believing React is immune to XSS
- Not understanding value escaping
- Overusing dangerouslySetInnerHTML

Q1.46. What is dangerouslySetInnerHTML and why is it dangerous?

dangerouslySetInnerHTML allows inserting raw HTML into the DOM. It bypasses React’s built-in escaping and can expose the app to XSS vulnerabilities if misused.

```
<div dangerouslySetInnerHTML={{ __html: htmlContent }} />
```

Why interviewers ask this:
This question is used to test security maturity and whether candidates understand when and why React bypasses escaping.

Common mistakes:

- Using dangerouslySetInnerHTML unnecessarily
- Rendering user input directly
- Not sanitizing HTML

Q1.47. How does JSX differ from HTML at runtime?

JSX is converted into JavaScript objects representing React elements, not actual DOM nodes. React reconciles these objects and updates the real DOM efficiently.

Why interviewers ask this:
Interviewers ask this to differentiate between surface-level React users and those who understand React's rendering internals.

Common mistakes:

- Thinking JSX creates DOM nodes
- Confusing Virtual DOM with real DOM

Q1.48. How do comments work in JSX?

JSX comments must be written inside curly braces using JavaScript comment syntax. HTML comments do not work inside JSX.

```
{/* This is a JSX comment */}
```

Q1.49. What is list rendering in React and why is it commonly used?

List rendering in React refers to the process of displaying multiple similar UI elements by iterating over an array of data. It is commonly used when rendering collections such as menus, tables, cards, or dynamic datasets fetched from APIs. React leverages JavaScript array methods like map() to transform data into UI elements in a declarative and readable way.

```
{items.map(item => <li>{item}</li>)}
```

Why interviewers ask this:
Interviewers ask this to confirm that the candidate understands how React handles dynamic UI generation from data rather than hard-coded markup.

Common mistakes:

- Using loops like for or while inside JSX instead of array methods, or attempting to manually repeat JSX elements.

Q1.50. Why does React require a key prop when rendering lists?

React requires a key prop to uniquely identify each element in a list. Keys help React efficiently determine which items have changed, been added, or removed during reconciliation. This allows React to update only the necessary DOM elements instead of re-rendering the entire list.

```
<li key={item.id}>{item.name}</li>
```

Why interviewers ask this:
This question tests understanding of React's reconciliation algorithm and performance optimization strategies.

Common mistakes:

- Ignoring the key warning or assuming keys are only required to remove console errors.

Q1.51. What happens if you do not provide keys in a list?

If keys are not provided, React falls back to using array indexes internally, which can lead to inefficient updates and UI inconsistencies when items are reordered, added, or removed. React will also display a warning in the console to indicate improper list rendering.

Why interviewers ask this:

Interviewers want to evaluate awareness of subtle UI bugs caused by improper reconciliation.

Common mistakes:

- Assuming the UI will always behave correctly without keys.

Q1.52. Why is using array index as a key considered an anti-pattern?

Using array index as a key can cause incorrect UI updates when list items change order, are inserted, or removed. Since indexes change based on position rather than identity, React may reuse components incorrectly, leading to bugs such as incorrect state association or unexpected re-renders.

```
{items.map((item, index) => <Item key={index} />)}
```

Why interviewers ask this:

This question checks deeper understanding of identity vs position in React rendering.

Common mistakes:

- Using index as key for dynamic or mutable lists.

Q1.53. What makes a good key in React?

A good key is a stable, predictable, and unique identifier that does not change between renders. Typically, this comes from a database ID or a unique value intrinsic to the data. Stable keys ensure React can correctly track element identity across updates.

```
<Item key={user.id} user={user} />
```

Why interviewers ask this:

Interviewers look for best practices rather than workarounds.

Common mistakes:

- Using random values or Math.random() as keys.

Q1.54. How does React use keys during reconciliation?

During reconciliation, React compares the previous and next virtual DOM trees. Keys allow React to match elements between renders, identify moved or removed items, and minimize DOM mutations. Without proper keys, React may destroy and recreate elements unnecessarily.

Why interviewers ask this:

Tests understanding of React internals and performance behavior.

Common mistakes:

- Thinking keys are used only for rendering order.

Q1.55. Can keys be reused across different lists?

Keys must be unique only among siblings within the same list. The same key values can safely be reused in different lists or different component trees without causing conflicts.

Why interviewers ask this:

Checks scoping knowledge of keys.

Common mistakes:

- Assuming keys must be globally unique across the entire application.

Q1.56. How do improper keys affect component state?

When keys are incorrect or unstable, React may reuse component instances improperly. This can cause state from one list item to appear in another, leading to confusing bugs such as inputs retaining wrong values or toggles switching unexpectedly.

Why interviewers ask this:

This is a high-signal question that reveals real-world debugging experience.

Common mistakes:

- Blaming React state bugs without checking key usage.

Q1.57. How should keys be handled when rendering nested lists?

Each level of a nested list requires its own set of unique keys relative to its sibling elements. Keys should be applied at the level where the map() is executed to ensure proper reconciliation at every depth.

Why interviewers ask this:

Tests complex rendering scenarios and hierarchical thinking.

Common mistakes:

- Applying a single key at the top level only.

Q1.58. What are real-world scenarios where incorrect keys cause production bugs?

Incorrect keys commonly cause issues in sortable lists, drag-and-drop interfaces, editable tables, and animated lists. Symptoms include mismatched UI state, flickering components, and unexpected re-renders that are difficult to debug.

Why interviewers ask this:

Interviewers use this question to assess production-level experience with React.

Common mistakes:

- Failing to connect UI anomalies with key misconfiguration.

Q1.59. What are props in React?

Props (short for properties) are read-only inputs passed from a parent component to a child component. They allow components to receive data and configuration, making components reusable and dynamic.

```
function Greeting({ name }) {
  return <h1>Hello {name}</h1>;
}
```

Why interviewers ask this:

Interviewers ask this to verify understanding of React's data flow and component communication.

Common mistakes:

- Confusing props with state or trying to modify props inside a child component.

Q1.60. How do you pass data to a component using props?

Data is passed to components via attributes when rendering a component. These attributes are received as props inside the child component.

```
function Greeting(props) {
  return <h1>Hello {props.name}</h1>;
}
```

Why interviewers ask this:

Checks whether the candidate understands parent-to-child data flow.

Common mistakes:

- Forgetting to pass required props or mismatching prop names.

Q1.61. Are props mutable or immutable?

Props are immutable. A component must never modify its own props. Any change to props must come from the parent component.

Why interviewers ask this:

Tests understanding of React's unidirectional data flow.

Common mistakes:

- Attempting to update props directly inside a component.

Q1.62. What is props destructuring and why is it useful?

Props destructuring extracts values directly from the props object, making code cleaner and more readable.

```
function User({ name, age }) {
  return <p>{name} is {age} years old</p>;
}
```

Why interviewers ask this:

Interviewers look for clean coding practices and modern JavaScript usage.

Common mistakes:

- Over-destructuring or forgetting default values.

Q1.63. How can you pass functions as props?

Functions can be passed as props to allow child components to communicate events back to the parent component.

```
function Parent() {
  const handleClick = () => alert("Clicked");
  return <Child onClick={handleClick} />;
}
```

Why interviewers ask this:

Checks understanding of event handling and child-to-parent communication.

Common mistakes:

- Calling the function immediately instead of passing a reference.

Q1.64. What are default props and how are they used?

Default props provide fallback values for props when no value is passed from the parent component.

```
function Button({ label = "Click" }) {
  return <button>{label}</button>;
}
```

Why interviewers ask this:

Tests defensive programming and component robustness.

Common mistakes:

- Relying too heavily on defaults instead of validating inputs.

Q1.65. How do props differ from state?

Props are passed from parent to child and are immutable, while state is managed within a component and can change over time.

Why interviewers ask this:

This is a classic React comparison question.

Common mistakes:

- Using state where props are sufficient or vice versa.

Q1.66. What is prop drilling and why is it a problem?

Prop drilling occurs when props are passed through multiple levels of components that do not need them, making code harder to maintain.

Why interviewers ask this:

Interviewers want to assess scalability and architecture awareness.

Common mistakes:

- Ignoring prop drilling instead of using Context or state management.

Q1.67. How can you avoid prop drilling in React?

Prop drilling can be avoided using React Context, component composition, or state management libraries like Redux.

Why interviewers ask this:

Tests knowledge of advanced data-sharing patterns.

Common mistakes:

- Overusing Context for simple cases.

Q1.68. Can props be used to pass JSX or components?

Yes, props can pass JSX elements or even entire components, enabling flexible and reusable component composition.

```
function Layout({ header }) {
  return <div>{header}</div>;
}
```

Why interviewers ask this:

Assesses understanding of React's compositional model.

Common mistakes:

- Overcomplicating components by passing too much JSX as props.

Q1.69. What is state in React?

State is a built-in mechanism that allows components to store and manage data that can change over time and trigger re-rendering of the UI when updated.

Why interviewers ask this:

Interviewers ask this to ensure the candidate understands React's reactive data model.

Common mistakes:

- Confusing state with regular variables or props.

Q1.70. What is the useState hook and what does it return?

useState is a React Hook that lets functional components manage state. It returns an array containing the current state value and a function to update it.

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

Why interviewers ask this:

Checks understanding of Hooks API fundamentals.

Common mistakes:

- Misunderstanding the array destructuring pattern.

Q1.71. How do you update state using useState?

State is updated by calling the setter function returned by useState. Updating state causes React to re-render the component.

```
setCount(count + 1);
```

Why interviewers ask this:
Validates knowledge of state updates and re-render behavior.

Common mistakes:

- Updating state directly instead of using the setter.

Q1.72. Why should you not update state directly?

Directly mutating state does not notify React about the change, which can lead to inconsistent UI and bugs. State must be updated using the setter function.

Why interviewers ask this:
Interviewers look for understanding of immutability and React internals.

Common mistakes:

- Mutating objects or arrays stored in state.

Q1.73. What is the functional form of setState in useState?

The functional form accepts a function that receives the previous state and returns the next state. It is useful when the new state depends on the previous state.

```
setCount(prevCount => prevCount + 1);
```

Why interviewers ask this:
Tests understanding of asynchronous state updates.

Common mistakes:

- Using stale state values in successive updates.

Q1.74. How does React batch state updates?

React may batch multiple state updates into a single re-render for performance reasons, especially within event handlers.

Why interviewers ask this:
Checks performance awareness and rendering behavior.

Common mistakes:

- Assuming state updates are applied immediately.

Q1.75. How do you update objects or arrays in state?

Objects and arrays should be updated immutably using spread operators or helper methods to create new references.

```
setUser({ ...user, age: 30 });
```

Why interviewers ask this:
Interviewers want to see immutability best practices.

Common mistakes:

- Mutating arrays with push or objects by direct assignment.

Q1.76. Why is state update asynchronous in React?

React schedules state updates to optimize rendering and improve performance. This allows React to batch updates and minimize unnecessary re-renders.

Why interviewers ask this:

Assesses deeper understanding of React's rendering model.

Common mistakes:

- Writing logic that assumes immediate state updates.

Q1.77. When should you use multiple useState hooks instead of one?

Using multiple useState hooks is preferred when state values are unrelated, improving readability and reducing unnecessary updates.

Why interviewers ask this:

Tests component design and maintainability.

Common mistakes:

- Grouping unrelated state into a single object.

Q1.78. What happens when state updates cause a re-render?

When state changes, React re-executes the component function, compares the new virtual DOM with the previous one, and updates only the necessary parts of the real DOM.

Why interviewers ask this:

Checks understanding of reconciliation and rendering.

Common mistakes:

- Thinking React re-renders the entire DOM.

2. HOOKS

Q2.1. What are custom hooks and why are they useful?

Custom hooks are functions that encapsulate reusable stateful logic using existing hooks. They improve code reuse, separation of concerns, and testability without altering component structure.

```
function useFetch(url) { /* logic */ }
```

Why interviewers ask this:

Interviewers assess abstraction and reuse skills.

Common mistakes:

- Trying to use custom hooks for UI rendering.

Q2.2. What rules must custom hooks follow?

Custom hooks must follow the Rules of Hooks: they must be called at the top level and only from React components or other hooks. This ensures React can track hook order correctly.

Why interviewers ask this:

Checks understanding of hook invariants.

Common mistakes:

- Calling hooks conditionally inside custom hooks.

Q2.3. What is the difference between useLayoutEffect and useEffect in React?

`useEffect` runs after the component renders and the browser has painted, making it suitable for side effects like data fetching, subscriptions, or logging. `useLayoutEffect` runs synchronously after all DOM mutations but before the browser paints, allowing

updates to layout, measurements, or DOM manipulations that need to happen before the user sees the UI. Using `useLayoutEffect` incorrectly can block the paint and cause jank.

```
// Example:
useEffect(() => {
  fetch("/api/data").then(setData);
}, []);
```

Why interviewers ask this:
Tests subtle understanding of React rendering phases, layout timing, and proper side-effect management.

Common mistakes:

- Confusing the timing of execution; using useLayoutEffect for async operations unnecessarily, or trying to measure DOM before it exists.

Q2.4. When should you use useReducer instead of useState in React?

`useReducer` is useful for managing complex state logic where state transitions depend on previous state or involve multiple sub-values. It provides a predictable state update pattern using a reducer function, similar to Redux but at component level. This is particularly valuable in forms, multi-step wizards, or when replacing Redux-like logic locally.

```
// Usage:
dispatch({ type: "increment" });
```

Why interviewers ask this:
Assesses the ability to design maintainable and scalable state management in React components.

Common mistakes:

- Overcomplicating simple state by using useReducer unnecessarily, or mismanaging the reducer function structure.

Q2.5. What problem does useCallback solve?

useCallback memoizes function references so they are not recreated on every render. This is important when passing callbacks to memoized child components to prevent unnecessary re-renders.

```
useCallback(() => handleClick(id), [id]);
```

Why interviewers ask this:
Tests understanding of referential equality.

Common mistakes:

- Using useCallback without memoized children.

Q2.6. What problem does useContext solve?

useContext allows components to consume shared data without prop drilling. It is commonly used for global concerns like themes, authentication, or localization where passing props through many layers becomes cumbersome.

```
const theme = useContext(ThemeContext);
```

Why interviewers ask this:
Interviewers check architectural thinking and state distribution.

Common mistakes:

- Using context for frequently changing data leading to unnecessary re-renders.

Q2.7. How does useContext impact performance?

When context value changes, all consuming components re-render. Without memoization or context splitting, this can cause performance issues. Proper context design minimizes unnecessary updates.

Why interviewers ask this:
Tests performance awareness.

Common mistakes:

- Using a single large context for unrelated data.

Q2.8. What is useEffect and when does it run?

useEffect is used to perform side effects in functional components such as data fetching, subscriptions, or manual DOM manipulation. By default, it runs after the component renders, ensuring the UI is committed before side effects execute.

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

Why interviewers ask this:
Interviewers test understanding of side effects vs rendering.

Common mistakes:

- Performing side effects directly inside render logic.

Q2.9. How does the dependency array control useEffect execution?

The dependency array tells React when to re-run the effect by tracking value changes via reference equality. If a dependency changes, React cleans up the previous effect and runs it again, ensuring state and effects stay in sync.

Why interviewers ask this:
This reveals deep understanding of closures, identity, and synchronization.

Common mistakes:

- Omitting dependencies or disabling lint rules instead of fixing logic.

Q2.10. When should useMemo be used?

useMemo memoizes expensive calculations to avoid recomputation on every render. It should be used when a calculation is costly and its dependencies change infrequently.

```
useMemo(() => expensiveCalc(data), [data]);
```

Why interviewers ask this:
Interviewers want to avoid premature optimization.

Common mistakes:

- Using useMemo everywhere without performance justification.

Q2.11. What is useRef commonly used for?

useRef provides a mutable container that persists across renders without causing re-renders. It is commonly used to access DOM elements or store mutable values like timers or previous state.

```
const inputRef = useRef(null);
```

Why interviewers ask this:
Checks understanding of refs vs state.

Common mistakes:

- Using useRef to trigger UI updates.

Q2.12. How does useRef differ from useState?

useRef does not trigger re-renders when updated, whereas useState does. This makes refs ideal for storing non-UI mutable values, while state should be used for data that affects rendering.

Why interviewers ask this:
Tests mental model clarity.

Common mistakes:

- Replacing state with refs for convenience.

Q2.13. What problem does the useState hook solve in React?

useState allows functional components to manage internal state. Before hooks, state was only available in class components. useState enables stateful logic inside functional components, making components simpler, more reusable, and easier to reason about.

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

Why interviewers ask this:
Interviewers ask this to ensure the candidate understands why hooks exist, not just how to use them.

Common mistakes:

- Thinking useState is only a replacement for setState without understanding functional updates.

Q2.14. Why is state updating via useState asynchronous?

State updates in React are batched for performance. When setState is called, React schedules an update rather than applying it immediately. This ensures fewer re-renders and predictable reconciliation.

Why interviewers ask this:
Tests understanding of rendering lifecycle and batching.

Common mistakes:

- Expecting state to update immediately after calling the setter.

3. STATE MANAGEMENT

Q3.1. What problem does the Context API solve in state management?

The Context API solves prop drilling by allowing data to be shared across component trees without manually passing props through each level. It is best suited for low-frequency, broadly shared data.

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

Why interviewers ask this:
Interviewers assess understanding of built-in React tools.

Common mistakes:

- Using Context as a replacement for all state management.

Q3.2. Why is Context API not a full state management solution?

Context does not provide built-in mechanisms for state normalization, caching, middleware, or advanced debugging. Frequent updates to context can cause widespread re-renders, making it unsuitable for high-frequency state changes.

Why interviewers ask this:
This checks whether candidates understand limitations.

Common mistakes:

- Using Context for rapidly changing UI state.

Q3.3. What is the difference between local state and global state in React?

Local state is owned and managed by a single component and affects only its internal UI. Global state is shared across multiple components and represents application-wide concerns such as user authentication, theme, or cached server data.

Why interviewers ask this:

Interviewers want to evaluate understanding of state ownership and scope.

Common mistakes:

- Promoting local state to global prematurely.

Q3.4. How do you decide whether state should be local or global?

State should be local if it is used by a single component or tightly coupled to its UI. It should be global only when multiple distant components require access or synchronization. Elevating state too early increases complexity and cognitive load.

Why interviewers ask this:

This tests architectural judgment.

Common mistakes:

- Assuming global state management is inherently better.

Q3.5. When should Redux NOT be used?

Redux should not be used for local UI state, simple forms, or applications with minimal shared state. Introducing Redux unnecessarily increases cognitive load, code complexity, and maintenance overhead.

Why interviewers ask this:

This is a high-signal question that reveals seniority.

Common mistakes:

- Adding Redux as a default project setup.

Q3.6. What problems does Redux Toolkit solve compared to classic Redux?

Redux Toolkit reduces boilerplate by providing utilities like createSlice and createAsyncThunk. It enforces best practices by default, simplifies immutable updates, and improves developer experience.

```
const slice = createSlice({ name: "counter", initialState, reducers: {} });
```

Why interviewers ask this:

Interviewers want to ensure modern Redux knowledge.

Common mistakes:

- Using legacy Redux patterns with Redux Toolkit.

Q3.7. When is Redux a good choice for state management?

Redux is suitable for large applications with complex shared state, predictable data flow requirements, and the need for advanced debugging, caching, or middleware. It excels in scenarios where state changes must be traceable and deterministic.

Why interviewers ask this:

This tests real-world experience.

Common mistakes:

- Introducing Redux for small or simple applications.

Q3.8. How do you design a state management strategy for a large React application?

A robust strategy uses a combination of local state, Context for low-frequency shared data, and external state libraries for complex global concerns. Senior engineers prioritize simplicity, scalability, and team clarity over tooling preferences.

Why interviewers ask this:

This distinguishes senior engineers from framework users.

Common mistakes:

- Relying on a single tool for all state problems.

Q3.9. Why might a team choose Zustand or Recoil over Redux?

Modern alternatives like Zustand or Recoil offer simpler APIs, reduced boilerplate, and more granular subscriptions. They are often easier to adopt while still supporting global state patterns.

Why interviewers ask this:

Interviewers want awareness of evolving ecosystem.

Common mistakes:

- Choosing a library based on popularity rather than fit.

4. ARCHITECTURE

Q4.1. How do you decide which component pattern to use in a large React application?

The choice depends on data ownership, frequency of updates, reuse requirements, and team conventions. Senior engineers balance simplicity, performance, and maintainability rather than following rigid patterns.

Why interviewers ask this:

This question distinguishes senior engineers from mid-level developers.

Common mistakes:

- Applying patterns dogmatically without considering context.

Q4.2. What is the compound components pattern in React?

The compound components pattern allows multiple components to work together while sharing implicit state via context. It provides a flexible and expressive API while keeping related logic encapsulated.

Why interviewers ask this:

This reveals advanced component API design skills.

Common mistakes:

- Overusing compound patterns for simple components.

Q4.3. What is the difference between controlled and uncontrolled components in React?

Controlled components derive their value from React state and update through event handlers, making the React component the single source of truth. Uncontrolled components rely on the DOM to manage state internally, typically accessed via refs.

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

Why interviewers ask this:

Interviewers use this to assess understanding of state ownership and predictability.

Common mistakes:

- Using uncontrolled components for complex forms where validation and synchronization are required.

Q4.4. When would you choose an uncontrolled component over a controlled one?

Uncontrolled components are suitable for simple forms, performance-sensitive inputs, or when integrating with non-React libraries. They reduce re-rendering overhead but sacrifice fine-grained control.

```
const inputRef = useRef();
```

Why interviewers ask this:
This tests practical decision-making rather than rigid rules.

Common mistakes:

- Believing controlled components are always superior.

Q4.5. What does lifting state up mean in React?

Lifting state up refers to moving shared state to the closest common ancestor so that multiple components can stay synchronized. This establishes a clear data flow and prevents duplicated or conflicting state.

Why interviewers ask this:
Interviewers want to see understanding of data flow design.

Common mistakes:

- Duplicating state across sibling components.

Q4.6. What are the downsides of excessive state lifting?

Excessive lifting can lead to bloated parent components, unnecessary re-renders, and prop drilling. At scale, it can harm readability and maintainability, indicating a need for better state distribution strategies.

Why interviewers ask this:
Tests architectural maturity.

Common mistakes:

- Using lifting as the only state management strategy.

Q4.7. What is the difference between presentational and container components?

Presentational components focus on how things look and receive data via props, while container components handle logic, data fetching, and state management. This separation improves reusability and testability.

Why interviewers ask this:
Interviewers assess separation of concerns.

Common mistakes:

- Mixing heavy logic into UI-only components.

Q4.8. What is prop drilling and when does it become a problem?

Prop drilling occurs when data is passed through many layers of components that do not directly use it. It becomes problematic when it increases coupling, reduces readability, and makes refactoring difficult.

Why interviewers ask this:
Tests awareness of maintainability issues.

Common mistakes:

- Using context immediately without evaluating component structure.

Q4.9. When should Context API be preferred over prop drilling?

Context should be used when data is truly global or widely shared, such as themes or authentication state. Overusing context can harm performance and obscure data flow.

Why interviewers ask this:
This question exposes architectural judgment.

Common mistakes:

- Replacing all prop passing with context.

Q4.10. What makes a React component truly reusable?

A reusable component is flexible, predictable, and decoupled from specific business logic. It exposes configuration through props, avoids hard-coded dependencies, and focuses on a single responsibility.

Why interviewers ask this:

Interviewers want insight into scalable UI design.

Common mistakes:

- Overgeneralizing components and making APIs too complex.

5. ROUTING

Q5.1. Why and how do you lazy load routes in React?

Lazy loading routes reduces initial bundle size by loading route components only when needed. React.lazy combined with Suspense enables code splitting at the route level.

```
const Page = React.lazy(() => import("./Page"));
```

Why interviewers ask this:

Performance optimization is a strong senior signal.

Common mistakes:

- Not providing a fallback UI.

Q5.2. What are nested routes and why are they useful?

Nested routes allow rendering child routes within parent layouts. They help organize UI structure and share common layout components like headers or sidebars.

```
<Route path="dashboard" element={<Layout />} />
```

Why interviewers ask this:

Checks understanding of layout-driven routing.

Common mistakes:

- Forgetting to use <Outlet>.

Q5.3. How do you implement protected routes in React?

Protected routes are implemented by conditionally rendering route elements based on authentication state. If unauthorized, users are redirected using <Navigate>.

```
return isAuth ? <Outlet /> : <Navigate to="/login" />;
```

Why interviewers ask this:

This tests real-world auth handling.

Common mistakes:

- Checking auth inside every component.

Q5.4. What problem does React Router solve?

React Router maps URLs to React components, enabling navigation and deep linking in Single Page Applications. It keeps the UI in sync with the browser URL.

Why interviewers ask this:

This checks foundational routing knowledge.

Common mistakes:

- Manually handling routing logic with window.location.

Q5.5. Explain the difference between <BrowserRouter> and <HashRouter>.

BrowserRouter uses the HTML5 history API for clean URLs, while HashRouter uses URL hashes (#) and works without server-side configuration. BrowserRouter is preferred for production when server support is available.

```
<BrowserRouter><App /></BrowserRouter>
```

Why interviewers ask this:
Interviewers assess deployment awareness.

Common mistakes:

- Using BrowserRouter without server fallback configuration.

Q5.6. What is the difference between <Link> and <a> tags?

Link performs client-side navigation without reloading the page, preserving application state. Anchor tags cause a full page reload.

```
<Link to="/home">Home</Link>
```

Why interviewers ask this:
Interviewers want SPA behavior clarity.

Common mistakes:

- Using <a> for internal navigation.

Q5.7. How can query parameters be handled in React Router?

Query parameters can be accessed and manipulated using the useSearchParams hook. They are commonly used for filters, pagination, and sorting.

```
const [params] = useSearchParams();
```

Why interviewers ask this:
Tests URL-driven state understanding.

Common mistakes:

- Storing filter state only in local state.

Q5.8. How do route parameters work in React Router?

Route parameters allow dynamic values in the URL, accessed using the useParams hook. They enable reusable routes such as user profiles or product pages.

```
const { id } = useParams();
```

Why interviewers ask this:
Tests practical routing usage.

Common mistakes:

- Trying to access params via props in v6.

Q5.9. What are common routing anti-patterns in React applications?

Common anti-patterns include deeply nested routes, mixing routing logic inside components, and overusing redirects. Clean routing architecture improves maintainability and readability.

Why interviewers ask this:
This evaluates architectural maturity.

Common mistakes:

- Treating routing as an afterthought.

Q5.10. What is client-side routing in a Single Page Application?

Client-side routing allows navigation between different views without reloading the page. Instead of requesting new HTML from the server, the browser updates the URL and React renders different components based on the route, resulting in faster transitions and better user experience.

Why interviewers ask this:
Interviewers want to ensure understanding of SPA fundamentals.

Common mistakes:

- Confusing client-side routing with traditional server-side routing.

6. FORMS

Q6.1. What is a controlled form component in React?

A controlled form component is one where form input values are managed by React state. Every change triggers a state update, making React the single source of truth for the form data.

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

Why interviewers ask this:
Interviewers want to confirm understanding of React's data flow.

Common mistakes:

- Mixing controlled and uncontrolled inputs unintentionally.

Q6.2. What are the advantages and disadvantages of controlled forms?

Controlled forms offer better validation, predictable state, and easier debugging. However, they can introduce performance overhead for large forms due to frequent re-renders.

Why interviewers ask this:
Tests trade-off analysis.

Common mistakes:

- Using controlled inputs everywhere without considering scale.

Q6.3. When would you prefer uncontrolled components over controlled ones?

Uncontrolled components are useful for simple forms or when integrating with non-React libraries. They rely on refs instead of state, reducing re-renders.

```
const inputRef = useRef();
```

Why interviewers ask this:
Interviewers assess performance awareness.

Common mistakes:

- Using refs for complex validation logic.

Q6.4. How should form validation errors be handled and displayed?

Validation errors should be displayed near relevant fields with clear messages. Centralized error handling improves accessibility and user experience.

Why interviewers ask this:

UX-focused evaluation.

Common mistakes:

- Displaying generic error messages.

Q6.5. How do you manage loading and submission states in forms?

Submission state is managed using flags such as isSubmitting or loading indicators. Disabling buttons and showing progress feedback prevents duplicate submissions.

```
setIsSubmitting(true);
```

Why interviewers ask this:

Tests real-world UX awareness.

Common mistakes:

- Allowing multiple form submissions.

Q6.6. Why are form libraries like React Hook Form or Formik used?

Form libraries reduce boilerplate, improve performance, and provide structured validation and error handling. React Hook Form is especially popular due to its minimal re-renders.

Why interviewers ask this:

Tests familiarity with industry tools.

Common mistakes:

- Using custom form logic when libraries would simplify.

Q6.7. How does React Hook Form improve performance compared to controlled forms?

React Hook Form uses uncontrolled inputs and refs internally, minimizing re-renders and improving performance for large or complex forms.

```
const { register } = useForm();
```

Why interviewers ask this:

High-signal performance-related question.

Common mistakes:

- Treating React Hook Form as controlled.

Q6.8. What are common form-related anti-patterns in React applications?

Common anti-patterns include deeply nested form state, duplicating validation logic, ignoring accessibility, and overusing local state instead of form libraries.

Why interviewers ask this:

Separates senior engineers from beginners.

Common mistakes:

- Ignoring accessibility and UX.

Q6.9. How is schema-based validation handled in React?

Schema-based validation uses libraries like Yup or Zod to define form validation rules declaratively. This improves maintainability and consistency across forms.

```
const schema = z.object({ email: z.string().email() });
```

Why interviewers ask this:
Interviewers look for scalable validation strategies.

Common mistakes:

- Hardcoding validation logic inside components.

Q6.10. What are the benefits of Zod over Yup?

Zod is TypeScript-first and provides static type inference from schemas, reducing runtime errors and duplication between validation and types.

Why interviewers ask this:
Checks modern TypeScript awareness.

Common mistakes:

- Using Yup in strongly typed projects without type safety.

7. API

Q7.1. What are common anti-patterns when fetching data in React?

Anti-patterns include: fetching data directly inside render, failing to cancel subscriptions on unmount, over-fetching the same data, and poor caching strategies. Avoiding these ensures maintainable, performant applications.

Why interviewers ask this:
Tests architectural maturity.

Common mistakes:

- Triggering multiple identical requests due to missing dependency arrays.

Q7.2. How does caching and revalidation work in React Query?

React Query caches fetched data to avoid unnecessary requests. Revalidation ensures data freshness by refetching in the background when the component mounts or at specified intervals. It helps maintain a balance between performance and data accuracy.

```
queryClient.invalidateQueries("todos");
```

Why interviewers ask this:
Interviewers check understanding of optimized server-state management.

Common mistakes:

- Not invalidating stale data or manually refreshing unnecessarily.

Q7.3. How do you handle API errors in React applications effectively?

API errors should be caught and displayed to the user. Retry mechanisms, toast notifications, and fallback UI improve UX. Using libraries like Axios interceptors or React Query error handling simplifies global error handling.

```
try { await axios.get("/api"); } catch(err) { setError(err.message); }
```

Why interviewers ask this:
Assesses ability to handle real-world failure scenarios.

Common mistakes:

- Ignoring errors or crashing the app.

Q7.4. How do you fetch data from an API in React using fetch or Axios?

Data can be fetched using the browser fetch API or Axios library. Typically, the request is triggered inside useEffect to ensure it runs after component mount. Axios provides additional features like automatic JSON parsing and request cancellation.

```
useEffect(() => { fetch("/api/data").then(res => res.json()).then(setData); }, []);
```

Why interviewers ask this:
Interviewers want to ensure candidates understand async data flow.

Common mistakes:

- Making requests directly in the component body (causing infinite loops).

Q7.5. How should loading, error, and success states be managed when fetching data?

These states should be managed in local state (or React Query) to provide clear user feedback. For example: isLoading, isError, and data. Proper handling prevents displaying stale or partial data and improves UX.

```
const [data, setData] = useState(null);
const [isLoading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

Why interviewers ask this:
Evaluates practical data fetching design.

Common mistakes:

- Not handling errors or showing blank screens.

Q7.6. How do you implement pagination or infinite scroll in React?

Pagination is implemented by requesting data in chunks using page numbers or cursors. Infinite scroll detects when the user reaches the bottom of the list and triggers additional API calls. This improves performance and reduces initial load time.

```
window.addEventListener("scroll", handleScroll);
```

Why interviewers ask this:
Tests performance optimization and real-world UI handling.

Common mistakes:

- Loading all data at once causing slow rendering.

Q7.7. What is React Query and why use it over traditional fetching?

React Query (TanStack Query) provides caching, background updates, query invalidation, pagination, and request deduplication. It abstracts away boilerplate and ensures consistent server state management.

```
const { data, isLoading, error } = useQuery("todos", fetchTodos);
```

Why interviewers ask this:
Tests knowledge of modern React data fetching best practices.

Common mistakes:

- Manually managing cache and loading states for every request.

Q7.8. Why do we often use useEffect for API calls in React?

useEffect allows you to perform side effects in functional components. Placing API calls inside useEffect ensures they are executed after the component has mounted, avoiding repeated calls on each render.

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

Why interviewers ask this:
Tests understanding of React's rendering lifecycle.

Common mistakes:

- Triggering API calls inside the render function.

8. STYLING

Q8.1. What are the differences between CSS and SCSS in React projects?

CSS is a standard styling language; SCSS extends CSS with variables, nesting, and mixins. SCSS improves maintainability, reduces repetition, and allows modular organization of styles in React apps.

```
@import "variables.scss";
.container { color: $primary-color; }
```

Why interviewers ask this:
Checks knowledge of standard styling tools and maintainability practices.

Common mistakes:

- Using flat CSS for large apps causing repetition and poor maintainability.

Q8.2. What are the risks of using global CSS in React apps?

Global CSS can cause style collisions, specificity wars, and unintended overrides across components. Large applications become hard to maintain without proper modularization.

```
body { font-family: Arial; }
```

Why interviewers ask this:
Assesses understanding of potential pitfalls in styling React apps.

Common mistakes:

- Using global CSS indiscriminately leading to bugs and overrides.

Q8.3. How do CSS Modules prevent style conflicts in React?

CSS Modules automatically scope class names locally per component. Each class name is hashed at build time, preventing collisions and ensuring component-level style isolation.

```
import styles from "./Button.module.css";
<button className={styles.primary}>Click</button>;
```

Why interviewers ask this:
Tests knowledge of modular and maintainable styling patterns.

Common mistakes:

- Importing module incorrectly or forgetting to use generated classNames.

Q8.4. What are the main differences between CSS Modules and Styled Components?

CSS Modules keep CSS in separate files, scoped to components. Styled Components allow writing CSS inside JS files with dynamic props, theming, and runtime style evaluation. Choice depends on team preference, dynamic styling needs, and theming

requirements.

```
// CSS Modules: styles.button
// Styled Components: const Button = styled.button``;
```

Why interviewers ask this:
Checks ability to compare modern styling strategies.

Common mistakes:

- Assuming one is always better; not considering performance and dynamic styling.

Q8.5. When should you use inline styles instead of CSS Modules or Styled Components?

Inline styles are suitable for dynamic values, small style adjustments, or conditional styling. For large static styles, CSS Modules or Styled Components are preferred for maintainability and readability.

```
<div style={{ color: isActive ? "red" : "blue" }}>Text</div>
```

Why interviewers ask this:
Tests practical judgment in choosing styling approaches.

Common mistakes:

- Using inline styles for all components, leading to poor maintainability.

Q8.6. How do you create a styled component and pass dynamic props?

Styled Components allows defining component-level styles. Props can be used for dynamic styling, e.g., color, padding. This reduces the need for multiple CSS classes and improves readability.

```
const Button = styled.button`background: ${props => props.primary ? "blue" : "gray"}; color: white;`;
<Button primary />
```

Why interviewers ask this:
Tests understanding of dynamic styling in JS-driven styles.

Common mistakes:

- Hardcoding values instead of using props leading to repetitive components.

Q8.7. How can theming be implemented with Styled Components?

Styled Components provides a ThemeProvider. Define a theme object (colors, fonts) and pass it down to components. Components access theme via props or helper functions. This allows consistent theming across the app.

```
<ThemeProvider theme={theme}><Button>Click</Button></ThemeProvider>
```

Why interviewers ask this:
Tests advanced styling patterns and maintainable UI architecture.

Common mistakes:

- Hardcoding colors and fonts instead of using a theme object.

Q8.8. What are common performance pitfalls in styling React components?

Re-rendering components with inline style objects, frequent dynamic className changes, or heavy CSS-in-JS computations can hurt performance. Memoizing styles or using CSS Modules/Styled Components efficiently mitigates these issues.

```
const style = useMemo(() => ({ color: dynamicColor }), [dynamicColor]);
```

Why interviewers ask this:
Checks ability to optimize component rendering and styling.

Common mistakes:

- Not memoizing dynamic styles, causing unnecessary re-renders.

Q8.9. How do you use Tailwind CSS in React for utility-first styling?

Tailwind CSS uses utility classes directly in className. This enables rapid UI development with a consistent design system. Classes control padding, margin, color, font, etc., without writing separate CSS.

```
<div className="bg-blue-500 text-white p-4 rounded">Hello</div>
```

Why interviewers ask this:
Tests familiarity with modern utility-first frameworks.

Common mistakes:

- Overusing arbitrary utility classes causing unreadable JSX.

Q8.10. How does Tailwind handle responsive design in React?

Tailwind provides responsive prefixes (sm:, md:, lg:, xl:) to apply classes at specific breakpoints. Combining them allows building fully responsive components declaratively in JSX.

```
<div className="p-2 sm:p-4 md:p-6">Content</div>
```

Why interviewers ask this:
Evaluates ability to handle mobile-first and responsive UI.

Common mistakes:

- Ignoring mobile breakpoints or duplicating classes unnecessarily.

Q8.11. What are the advantages of using UI libraries like MUI, AntD, or Chakra?

UI libraries provide pre-built, accessible, and responsive components, consistent theming, and reduce development time. They also often include dark mode support and ready-to-use design patterns.

```
import { Button } from "@mui/material";
<Button variant="contained">Click Me</Button>;
```

Why interviewers ask this:
Assesses practical production experience with component libraries.

Common mistakes:

- Over-customizing components or ignoring the library's theming system.

Q8.12. How do you customize components in UI libraries while maintaining theming?

Use provided theming APIs (ThemeProvider, styled overrides) instead of overriding CSS. This ensures consistency, maintainability, and avoids breaking library updates.

```
const theme = createTheme({ palette: { primary: { main: "#1976d2" }}});
```

Why interviewers ask this:
Tests advanced knowledge of theming and maintainable UI design.

Common mistakes:

- Directly overriding CSS classes causing inconsistencies.

9. PERFORMANCE

Q9.1. How can you prevent unnecessary re-renders in React components?

Strategies include: using React.memo for functional components, splitting large components into smaller ones, memoizing expensive calculations with useMemo, memoizing callbacks with useCallback, and ensuring proper key usage in lists.

Why interviewers ask this:

Tests practical knowledge of render performance tuning.

Common mistakes:

- Not splitting large components, causing all children to re-render frequently.

Q9.2. How do code splitting and lazy loading improve React app performance?

Code splitting allows you to break the bundle into smaller chunks, loading only what is needed. React.lazy and Suspense enable lazy loading of components, which reduces initial load time and improves perceived performance.

```
const LazyComponent = React.lazy(() => import("./MyComponent"));
<Suspense fallback={<Loading />}><LazyComponent /></Suspense>;
```

Why interviewers ask this:

Tests knowledge of bundle optimization and modern React patterns.

Common mistakes:

- Not using Suspense fallback properly or over-splitting causing too many network requests.

Q9.3. Why is the key prop important when rendering lists in React?

The key prop helps React identify which items have changed, are added, or removed. Using a stable key improves reconciliation performance and prevents unnecessary re-renders of list items.

```
<ul>{items.map(item => <li key={item.id}>{item.name}</li>)}</ul>
```

Why interviewers ask this:

Checks understanding of React's reconciliation and virtual DOM.

Common mistakes:

- Using index as key in dynamic lists causing incorrect component re-renders.

Q9.4. What is React.memo and how does it improve performance?

React.memo is a higher-order component that memoizes a functional component. It prevents unnecessary re-renders by shallowly comparing props, so the component only re-renders when its props change.

```
export default React.memo(MyComponent);
```

Why interviewers ask this:

Interviewers want to test understanding of re-render optimizations in functional components.

Common mistakes:

- Using React.memo indiscriminately on all components without considering prop complexity or functions inside props.

Q9.5. How does useCallback differ from useMemo and when should it be used?

useCallback memoizes a function reference so that it does not change between renders unless its dependencies change. This is particularly useful when passing callbacks to child components wrapped with React.memo to avoid unnecessary re-renders.

```
const handleClick = useCallback(() => { doSomething(); }, [dependency]);
```

Why interviewers ask this:
Evaluates functional optimization skills and memoization understanding.

Common mistakes:

- Wrapping all functions in useCallback unnecessarily or ignoring the dependency array.

Q9.6. When and why would you use useMemo in a component?

useMemo memoizes the result of a computation between renders. It is useful for expensive calculations that do not need to be recalculated unless dependencies change, preventing unnecessary performance overhead.

```
const memoizedValue = useMemo(() => expensiveCalculation(a, b), [a, b]);
```

Why interviewers ask this:
Tests ability to optimize expensive computations.

Common mistakes:

- Overusing useMemo on cheap calculations or forgetting dependency arrays.

10. REDUX

Q10.1. How does Redux handle asynchronous logic?

Redux itself is synchronous. Asynchronous logic is handled via middleware such as redux-thunk, which allows dispatching functions that perform async operations and then dispatch synchronous actions based on results.

Why interviewers ask this:
Tests understanding of Redux extensibility.

Common mistakes:

- Trying to perform async logic inside reducers.

Q10.2. What is createAsyncThunk and why is it preferred?

createAsyncThunk standardizes async logic by automatically generating pending, fulfilled, and rejected action types. It reduces boilerplate and enforces consistent async handling patterns.

```
export const fetchData = createAsyncThunk("data/fetch", async () => {});
```

Why interviewers ask this:
This is a very high-signal modern Redux question.

Common mistakes:

- Manually managing loading and error states.

Q10.3. What role does middleware play in Redux?

Middleware sits between dispatching an action and the reducer. It enables logging, async logic, error handling, and side effects without breaking Redux’s core principles.

```
const logger = store => next => action => next(action);
```

Why interviewers ask this:
Interviewers assess system extensibility understanding.

Common mistakes:

- Using middleware for state mutation.

Q10.4. What are the three core principles of Redux?

Redux is built on three principles: Single source of truth (the entire application state lives in one store), state is read-only (state can only be changed via actions), and changes are made with pure functions (reducers). These principles ensure predictability, traceability, and easier debugging.

Why interviewers ask this:

Interviewers check whether candidates understand why Redux exists, not just how to use it.

Common mistakes:

- Reciting principles without understanding their practical impact.

Q10.5. Explain the Redux data flow step by step.

Redux follows a unidirectional data flow: a UI event dispatches an action, the reducer processes the action and returns a new state, the store updates, and subscribed UI components re-render based on the updated state. This strict flow ensures state changes are predictable and traceable.

```
dispatch(action) ? reducer(state, action) ? newState ? UI re-render
```

Why interviewers ask this:

This tests architectural clarity and mental models.

Common mistakes:

- Thinking reducers directly update the UI or mutate state.

Q10.6. How do you decide if Redux is the right choice for a project?

Redux is appropriate when the application has complex shared state, needs predictable updates, benefits from debugging tools, or requires advanced async handling. Senior engineers evaluate cost vs benefit before introducing Redux.

Why interviewers ask this:

This tests architectural decision-making.

Common mistakes:

- Choosing Redux based on popularity.

Q10.7. What are common Redux anti-patterns?

Common anti-patterns include storing derived state, duplicating server state unnecessarily, overusing Redux for local UI concerns, and creating overly complex store structures.

Why interviewers ask this:

This separates senior engineers from tool users.

Common mistakes:

- Using Redux as a default for all state.

Q10.8. What problems does Redux Toolkit solve?

Redux Toolkit reduces boilerplate by providing utilities like createSlice, configureStore, and built-in Immer support. It enforces best practices by default and eliminates common Redux pain points such as manual action types and immutable update complexity.

```
const slice = createSlice({ name: "counter", initialState, reducers: {} });
```

Why interviewers ask this:

Interviewers want to confirm modern Redux knowledge.

Common mistakes:

- Using legacy Redux patterns with Redux Toolkit.

Q10.9. Why is Immer important in Redux Toolkit?

Immer allows developers to write code that looks mutable while maintaining immutability under the hood. This improves readability, reduces bugs, and makes reducer logic easier to reason about.

```
state.count += 1;
```

Why interviewers ask this:
Checks understanding of immutability and developer ergonomics.

Common mistakes:

- Assuming state is actually mutated.

Q10.10. What does configureStore provide over createStore?

configureStore automatically sets up Redux DevTools, applies recommended middleware, enables better defaults, and simplifies store configuration. It represents the modern standard for creating Redux stores.

```
const store = configureStore({ reducer });
```

Why interviewers ask this:
Interviewers want best-practice alignment.

Common mistakes:

- Manually configuring middleware and DevTools.

11. TESTING

Q11.1. How do you test asynchronous behavior in React components?

Use async/await with RTL utilities like waitFor or findBy*. Ensure promises resolve and DOM updates are asserted correctly. Avoid using setTimeout in tests directly.

```
await waitFor(() => expect(screen.getByText("Loaded")).toBeInTheDocument());
```

Why interviewers ask this:
Tests understanding of handling asynchronous UI behavior in tests.

Common mistakes:

- Not waiting for async updates causing flaky tests.

Q11.2. What are common mistakes in testing React applications?

Common mistakes include over-testing implementation details, not testing user interactions, using real APIs, and ignoring async updates. Avoid brittle tests by focusing on behavior and output rather than internal state.

```
// Conceptual best-practice guidance, no code needed
```

Why interviewers ask this:
Checks awareness of best practices and potential traps.

Common mistakes:

- Writing tests that break on any small internal refactor or DOM change.

Q11.3. How do you test components that rely on React Context?

Wrap the component under test in the context provider and provide mock values. This ensures the component behavior can be tested without relying on actual app context.

```
<MyContext.Provider value={{ user: { name: "Test" }}}>
  <Profile />
</MyContext.Provider>
```

Why interviewers ask this:
Assesses skill in isolating context-dependent components for testing.

Common mistakes:

- Not providing mock context leading to undefined errors.

Q11.4. How do you test user events in React components?

React Testing Library provides fireEvent and userEvent to simulate clicks, typing, and other interactions. Tests should assert the result of user actions, such as DOM updates or function calls.

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
const handleClick = jest.fn();
render(<Button onClick={handleClick}>Click</Button>);
userEvent.click(screen.getByText("Click"));
expect(handleClick).toHaveBeenCalled();
```

Why interviewers ask this:
Tests ability to verify real user interactions.

Common mistakes:

- Testing implementation functions instead of visible outcomes.

Q11.5. How do you test controlled forms and validation in React?

Controlled form inputs can be tested by changing their value using fireEvent or userEvent, then asserting validation messages or form state updates. Libraries like React Hook Form and Formik provide testing utilities.

```
userEvent.type(screen.getByLabelText("Email"), "test@example.com");
expect(screen.getByText("Valid email")).toBeInTheDocument();
```

Why interviewers ask this:
Assesses ability to test complex form logic.

Common mistakes:

- Not simulating real user input, only testing state directly.

Q11.6. What is Jest and why is it commonly used with React?

Jest is a JavaScript testing framework developed by Facebook. It provides zero-configuration setup, snapshot testing, mocks, timers, and code coverage. It integrates seamlessly with React for unit and integration tests.

```
// Example Jest test
import { sum } from "./utils";
test("adds 1 + 2 to equal 3", () => {
  expect(sum(1,2)).toBe(3);
});
```

Why interviewers ask this:
Tests familiarity with standard React testing tools.

Common mistakes:

- Confusing Jest with other testing frameworks, or not using its mocking capabilities.

Q11.7. What is snapshot testing and when should it be used?

Snapshot testing captures a rendered component's output and compares it to a stored snapshot. Useful for detecting unexpected UI changes. Should be used for stable, presentational components, not dynamic data-heavy ones.

```
import { render } from "@testing-library/react";
import renderer from "react-test-renderer";
const tree = renderer.create(<Button>Click</Button>).toJSON();
expect(tree).toMatchSnapshot();
```

Why interviewers ask this:
Tests knowledge of Jest snapshot feature and appropriate usage.

Common mistakes:

- Overusing snapshots for frequently changing components causing unnecessary failures.

Q11.8. What is React Testing Library and how is it different from Enzyme?

React Testing Library (RTL) focuses on testing components from a user perspective rather than implementation details. Unlike Enzyme, which exposes internal component state, RTL encourages testing DOM interactions, accessibility, and behavior.

```
import { render, screen } from "@testing-library/react";
render(<Button>Click</Button>);
screen.getByText("Click");
```

Why interviewers ask this:
Tests understanding of modern testing best practices in React.

Common mistakes:

- Testing internal state or implementation details instead of user interactions.

Q11.9. Why is testing important in React applications?

Testing ensures code correctness, prevents regressions, and allows refactoring with confidence. It also improves maintainability, reliability, and developer productivity in production applications.

```
// Conceptual question, no code required
```

Why interviewers ask this:
Evaluates understanding of testing philosophy and best practices.

Common mistakes:

- Skipping tests or relying only on manual QA, leading to undetected bugs.

Q11.10. How can custom hooks be tested in React?

Custom hooks can be tested using RTL's renderHook utility from @testing-library/react-hooks or wrapping the hook inside a test component. Tests focus on state changes, side effects, and return values.

```
import { renderHook, act } from "@testing-library/react-hooks";
const { result } = renderHook(() => useCounter());
act(() => result.current.increment());
expect(result.current.count).toBe(1);
```

Why interviewers ask this:
Assesses knowledge of advanced testing techniques for hooks.

Common mistakes:

- Testing hooks outside of React context or ignoring side effects.

Q11.11. How do you mock API calls in component or hook tests?

API calls can be mocked using Jest's jest.mock(), mock service worker (MSW), or libraries like axios-mock-adapter. This isolates components/hooks from network and allows predictable test outcomes.

```
jest.mock("axios");
import axios from "axios";
axios.get.mockResolvedValue({ data: [] });
```

Why interviewers ask this:
Tests practical skills in isolating components for testing.

Common mistakes:

- Calling real APIs in tests leading to flaky or slow tests.

Q11.12. What is the difference between unit tests and integration tests in React?

Unit tests verify individual functions or components in isolation. Integration tests verify that multiple components work together correctly, including API calls, context, and child components. Both are essential for reliable applications.

```
// Unit test: test Button renders
// Integration test: test Button inside Form with context
```

Why interviewers ask this:
Evaluates understanding of testing scope and strategy.

Common mistakes:

- Writing only unit tests without integration coverage, missing interactions.

12. DEBUGGING

Q12.1. What is React Strict Mode and why should it be used?

React Strict Mode is a tool for highlighting potential problems in an application during development. It does not render any visible UI but activates additional checks and warnings for deprecated APIs, unsafe lifecycles, and side effects. Using Strict Mode helps identify issues early and promotes better coding practices.

```
// Wrap your app
<React.StrictMode>
  <App />
</React.StrictMode>
```

Why interviewers ask this:
Tests understanding of proactive debugging and code quality improvement in React.

Common mistakes:

- Confusing Strict Mode with production behavior; expecting it to affect runtime in production (it only affects development).

Q12.2. Why does React double-invoke certain functions and lifecycle methods in Strict Mode during development?

Strict Mode intentionally double-invokes functions such as component constructors, render, and effects (useEffect, useState setters) in development. This helps detect side-effects that may be unsafe or impure, ensuring components are resilient and side-effect-free.

```
// Example: useEffect will run twice in development
useEffect(() => {
```

```
  console.log("Effect runs");
}, []);
```

Why interviewers ask this:
Assesses deeper understanding of React internals and lifecycle behavior.

Common mistakes:

- Assuming double invocation is a bug, leading to unnecessary debugging.

Q12.3. How does Strict Mode affect component lifecycle and behavior?

Strict Mode does not alter production behavior but affects development lifecycle: constructors, render, and useEffect are called twice; legacy lifecycle methods may trigger warnings. This encourages pure functions, safe side-effects, and helps detect accidental mutations.

Why interviewers ask this:
Tests ability to reason about lifecycle impacts and debugging implications.

Common mistakes:

- Ignoring warnings or side-effects detected by Strict Mode can lead to bugs in production.

Q12.4. What is React DevTools Profiler and how is it used for performance debugging?

React DevTools Profiler allows developers to record performance of component renders. It shows the time spent in rendering each component, highlighting expensive renders and unnecessary re-renders. By analyzing the profiler data, developers can optimize rendering, memoize components, or refactor state management.

```
// In browser, open React DevTools > Profiler > Record interactions to measure component render time.
```

Why interviewers ask this:
Tests practical performance debugging skills and optimization knowledge.

Common mistakes:

- Relying on console.log for performance measurement or ignoring expensive renders.

Q12.5. How can you inspect the render count of a component and why is it important?

Render count helps identify unnecessary re-renders which may affect performance. Using React DevTools or console tracking inside the component helps measure renders. High render counts may indicate improper state management, lack of memoization, or unoptimized props.

```
// Example: counting renders
function MyComponent() {
  const renderCount = useRef(0);
  renderCount.current++;
  console.log("Render count:", renderCount.current);
  return <div>Hello</div>;
}
```

Why interviewers ask this:
Assesses awareness of React performance pitfalls and optimization techniques.

Common mistakes:

- Ignoring high render counts leads to slower apps; misunderstanding frequent renders as a bug instead of normal reconciliation behavior.

Q12.6. How do you profile render cost and optimize expensive components in React?

Profiling render cost involves using React DevTools Profiler to measure the time each component takes to render. After identifying expensive components, optimizations include React.memo, useMemo, useCallback, splitting components, or lazy-loading. Profiling ensures smooth UI and prevents unnecessary re-renders.

```
// Example: wrapping component in React.memo to avoid unnecessary renders
const OptimizedComponent = React.memo(function Component({ data }) { return <div>{data}</div>; });
```

Why interviewers ask this:
Tests ability to perform performance analysis and apply optimization patterns.

Common mistakes:

- Optimizing without profiling, or prematurely memoizing components without identifying true bottlenecks.

13. ADVANCED

Q13.1. What is concurrent rendering in React and why is it important?

Concurrent rendering is a feature that allows React to interrupt long-running rendering tasks to keep the UI responsive. It enables React to prioritize urgent updates (like user input) over less critical ones. This leads to smoother user experiences, especially in large applications.

```
// Conceptual, but usage example: enable Concurrent Mode in React 18
import { createRoot } from "react-dom/client";
const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

Why interviewers ask this:
Evaluates understanding of modern performance and responsiveness techniques.

Common mistakes:

- Assuming all renders are synchronous, leading to poor responsiveness in complex apps.

Q13.2. How does concurrent mode differ from traditional React rendering?

Traditional rendering is synchronous: React processes updates one at a time, blocking the main thread. Concurrent Mode splits rendering into multiple units and can pause, resume, or prioritize updates without blocking the UI, improving responsiveness.

```
// Conceptual explanation with concurrent root
const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

Why interviewers ask this:
Tests deeper understanding of React 18+ performance improvements.

Common mistakes:

- Not understanding the implications of interleaved rendering on component behavior.

Q13.3. What are Error Boundaries in React and how do you use them?

Error Boundaries are React components that catch JavaScript errors in their child component tree during rendering, lifecycle methods, and constructors. They prevent the whole app from crashing and allow showing fallback UI.

```
class ErrorBoundary extends React.Component {
  constructor(props) { super(props); this.state = { hasError: false }; }
  static getDerivedStateFromError() { return { hasError: true }; }
  render() { return this.state.hasError ? <h1>Something went wrong.</h1> : this.props.children; }
}
```

Why interviewers ask this:
Checks understanding of robust error handling in React.

Common mistakes:

- Trying to catch errors in functional components without using Error Boundary class components.

Q13.4. What are best practices for error handling with Error Boundaries?

Always place Error Boundaries around high-level UI blocks, provide fallback UI, log errors to monitoring services, and avoid swallowing errors silently. Consider combining with logging tools for production readiness.

```
<ErrorBoundary fallback={<ErrorUI />}><Dashboard /></ErrorBoundary>
```

Why interviewers ask this:
Assesses readiness for production-level robust apps.

Common mistakes:

- Using Error Boundaries only in one place or ignoring logging.

Q13.5. Can Error Boundaries catch errors in Suspense fallbacks?

Error Boundaries can catch errors during rendering but not errors in event handlers or asynchronous code. When using Suspense, the fallback renders while waiting; errors inside lazy components can still be caught by Error Boundaries.

```
// Wrap lazy component with Error Boundary
<ErrorBoundary>
  <Suspense fallback={<Loading />}>
    <LazyComponent />
  </Suspense>
</ErrorBoundary>
```

Why interviewers ask this:
Tests ability to combine advanced patterns safely.

Common mistakes:

- Assuming Error Boundaries catch all async errors automatically.

Q13.6. What are Server Components in React (Next.js) and how do they differ from client components?

Server Components are rendered on the server and sent as HTML to the client. They reduce bundle size and improve performance because heavy logic does not need to run in the browser. Client components handle interactivity, while server components handle static rendering and data fetching.

```
// Example in Next.js 13+
export default function ServerComponent() {
  const data = await fetch("/api/data").then(res => res.json());
  return <div>{data.title}</div>;
}
```

Why interviewers ask this:
Assesses knowledge of modern SSR strategies and performance optimization.

Common mistakes:

- Confusing server and client components leading to unnecessary client-side rendering.

Q13.7. What is streaming and partial hydration in React?

Streaming allows the server to send parts of the HTML as they are rendered, reducing Time to First Byte (TTFB) and showing content faster. Partial hydration hydrates only interactive parts of the page, saving CPU and improving perceived performance.

```
// Conceptual example with Next.js 13
export default function Page() {
  return (
    <Suspense fallback={<Loading />}>
      <ServerComponent />
```

```
    </Suspense>
  );
}
```

Why interviewers ask this:
Evaluates understanding of advanced server-client rendering optimizations.

Common mistakes:

- Hydrating the whole page unnecessarily, leading to slow performance.

Q13.8. Why is streaming important for large React apps?

Streaming reduces time to first meaningful paint, allows the browser to progressively render content, and improves user perception of speed, especially for large pages with heavy data.

```
// Conceptual example
export default function Page() {
  return <ServerComponent />;
}
```

Why interviewers ask this:
Checks understanding of performance optimizations in real-world apps.

Common mistakes:

- Sending the whole HTML at once causing slower perceived load.

Q13.9. How is partial hydration applied in modern React apps?

Partial hydration hydrates only interactive components on the client, leaving static content untouched. This reduces JS execution, improves performance, and works well with server-rendered apps like Next.js.

```
// Example: hydrate only interactive parts
<InteractiveWidget />
```

Why interviewers ask this:
Tests knowledge of practical optimizations and server-client rendering balance.

Common mistakes:

- Hydrating static content unnecessarily, wasting CPU.

Q13.10. What is React Suspense and how does it help with data fetching?

Suspense allows components to "wait" for asynchronous operations (like API calls or lazy-loaded components) and display fallback content in the meantime. It simplifies handling loading states and improves user experience by coordinating UI readiness.

```
const Profile = React.lazy(() => import("./Profile"));
<Suspense fallback={<Spinner />}>
  <Profile />
</Suspense>;
```

Why interviewers ask this:
Tests knowledge of modern declarative asynchronous patterns.

Common mistakes:

- Using manual loading state management instead of declarative Suspense.

Q13.11. How does Suspense help with code splitting?

Suspense allows lazy-loaded components to display fallback content while the chunk is being fetched. This enables splitting the application bundle into smaller pieces and loading only what is needed.

```
const LazyComponent = React.lazy(() => import("./LazyComponent"));
<Suspense fallback={<Spinner />}><LazyComponent /></Suspense>;
```

Why interviewers ask this:
Assesses understanding of modern optimization patterns.

Common mistakes:

- Not using Suspense fallback, causing blank screens during lazy loading.

Q13.12. How do Suspense and Server Components work together in Next.js?

Suspense handles async rendering of server components. Server components fetch data on the server, and Suspense shows fallback UI while waiting. This combination improves performance, reduces client bundle size, and ensures smooth UX.

```
<Suspense fallback={<Loading />}>
  <ServerComponent />
</Suspense>
```

Why interviewers ask this:
Tests understanding of advanced React 18+ and Next.js patterns.

Common mistakes:

- Confusing client-side lazy loading with server component streaming.

14. ADVANCED PATTERNS

Q14.1. What are Render Props in React and how do they help in component reuse?

Render Props is a pattern where a component takes a function as a prop to determine what to render. It allows sharing behavior between components without repeating code. The component using a render prop can pass any data or state to the function, making it highly reusable.

```
// Component definition
function DataProvider({ render }) {
  const data = ["item1","item2"];
  return render(data);
}
```

Why interviewers ask this:
Tests knowledge of advanced composition patterns and component reusability.

Common mistakes:

- Confusing Render Props with HOCs or not properly passing props to the render function.

Q14.2. What are Higher-Order Components (HOCs) and why are they used?

HOCs are functions that take a component and return a new component with additional props or behavior. They enable code reuse, cross-cutting concerns like logging, error boundaries, or injecting global state without modifying the original component.

```
function withLogger(WrappedComponent) {
  return function(props) {
    console.log("Rendering", WrappedComponent.name);
    return <WrappedComponent {...props} />;
  }
}
```

Why interviewers ask this:
Assesses understanding of reusable abstraction patterns and component enhancement.

Common mistakes:

- Mutating the original component or forgetting to forward refs/props leading to bugs.

Q14.3. What is React.forwardRef and when should it be used?

React.forwardRef is a technique for passing a ref through a component to one of its child DOM elements. This is useful for exposing DOM nodes to parent components, handling focus, or integrating with third-party libraries that require refs.

```
// Parent usage
const ref = useRef(null);
<InputWithRef ref={ref} />;
```

Why interviewers ask this:
Tests understanding of refs, component encapsulation, and integration with imperative code.

Common mistakes:

- Attempting to attach a ref to a functional component without using forwardRef (which would fail).

Q14.4. What are React Portals and how are they used?

Portals provide a way to render children into a DOM node outside the parent component hierarchy. They are often used for modals, tooltips, or popovers where UI elements must visually break out of container boundaries while keeping React state and events intact.

```
// The component remains controlled by React
function Modal() { return <div className="modal">Hello</div>; }
```

Why interviewers ask this:
Tests knowledge of advanced DOM rendering strategies and practical UI patterns.

Common mistakes:

- Attempting to manually manipulate the DOM outside React or ignoring event bubbling across portals.

15. ECOSYSTEM

Q15.1. How do you build and deploy a React or Next.js application?

For CRA: run npm run build to generate static files in the build folder and serve using any static hosting. For Next.js: run npm run build and npm start for production SSR or deploy to platforms like Vercel for automatic SSR/SSG handling.

```
// CRA build
npm run build
// Next.js build
npm run build && npm start
```

Why interviewers ask this:
Evaluates understanding of production readiness and deployment strategies.

Common mistakes:

- Skipping build optimization or using development server in production.

Q15.2. What are best practices for continuous deployment of React apps?

Use CI/CD pipelines, automated tests, environment-specific builds, code reviews, and version control. Platforms like Vercel, Netlify, or GitHub Actions simplify deployment and rollback in case of errors.

```
// Example CI workflow
name: Deploy
on: push
jobs:
```

```
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm run build
      - run: npm run deploy
```

Why interviewers ask this:
Assesses readiness for professional software engineering workflows.

Common mistakes:

- Deploying manually without tests or version control, leading to downtime or errors.

Q15.3. How do you use environment variables in React applications?

Environment variables allow you to store sensitive or environment-specific information (API keys, endpoints). In CRA, variables must start with REACT_APP_. In Next.js, you can define variables in .env.local or .env.production, and use process.env.VARIABLE_NAME.

```
// CRA example
const apiUrl = process.env.REACT_APP_API_URL;
// Next.js example
const apiKey = process.env.API_KEY;
```

Why interviewers ask this:
Tests understanding of configuration management and security practices.

Common mistakes:

- Exposing sensitive keys directly in client-side code or forgetting REACT_APP_ prefix in CRA.

Q15.4. How does a production environment differ from development in React?

In production, React disables development warnings, minifies JS, and optimizes performance. Environment variables differ (.env.production vs .env.development), and build artifacts are static files or server-optimized bundles for deployment.

```
// Access production environment variables
const apiUrl = process.env.REACT_APP_API_URL;
```

Why interviewers ask this:
Tests understanding of environment awareness in production readiness.

Common mistakes:

- Using development variables or unoptimized builds in production.

Q15.5. Why are ESLint and Prettier important in React projects?

ESLint enforces coding standards and catches errors early, while Prettier formats code consistently. Together, they ensure maintainable, readable, and bug-free code, which is critical for team collaboration and code quality.

```
// ESLint config example
{
  "extends": ["react-app", "eslint:recommended"]
}
// Prettier config example
{
  "semi": true,
  "singleQuote": true
}
```

Why interviewers ask this:
Tests knowledge of tooling that improves developer productivity and code quality.

Common mistakes:

- Using only one without the other, leading to either poor formatting or missing error checks.

Q15.6. What is the difference between SSR (Server-Side Rendering) and SSG (Static Site Generation) in Next.js?

SSR renders pages on each request at runtime, ensuring fresh data on every page load. SSG pre-renders pages at build time, which improves performance but serves static content until next build. SSR is ideal for dynamic content, while SSG is ideal for static blogs or marketing pages.

```
// SSR example
export async function getServerSideProps(context) {
  const data = await fetchAPI();
  return { props: { data } };
}

// SSG example
export async function getStaticProps() {
  const data = await fetchAPI();
  return { props: { data } };
}
```

Why interviewers ask this:
Tests understanding of performance and rendering strategies in modern React frameworks.

Common mistakes:

- Confusing SSR with client-side rendering, leading to poor SEO or stale content.

Q15.7. How are dynamic routes handled in Next.js?

Next.js allows dynamic routes using bracket notation in the filename (e.g., [id].js). This enables pages to handle variable URL segments and fetch data accordingly, improving scalability and SEO.

```
// pages/posts/[id].js
export default function Post({ data }) { return <div>{data.title}</div>; }
export async function getStaticPaths() { return { paths: [], fallback: true }; }
```

Why interviewers ask this:
Assesses practical knowledge of routing in production apps.

Common mistakes:

- Using query parameters incorrectly instead of file-based dynamic routes.

Q15.8. How does Next.js optimize images for performance?

Next.js Image component automatically optimizes images by resizing, lazy loading, and serving in modern formats (WebP). This improves performance, reduces bandwidth, and ensures faster load times for end users.

```
<Image src="/avatar.png" width={200} height={200} alt="Avatar" />
```

Why interviewers ask this:
Tests knowledge of framework-specific performance optimizations.

Common mistakes:

- Using standard img tag for large images, causing slower load times.

Q15.9. What are Next.js API routes and why are they useful?

Next.js API routes allow you to build backend endpoints directly in your Next.js app without needing a separate server. Useful for serverless functions, handling form submissions, or simple backend logic alongside frontend.

```
// pages/api/hello.js
export default function handler(req, res) {
  res.status(200).json({ message: "Hello World" });
}
```

Why interviewers ask this:
Assesses full-stack React understanding.

Common mistakes:

- Creating a separate backend unnecessarily or mismanaging serverless routes.

Q15.10. What are the main differences between Vite and Create React App (CRA)?

Vite is a modern build tool using ES modules with extremely fast cold start and HMR (Hot Module Replacement). CRA relies on Webpack and can be slower for large apps. Vite improves development experience and build speed significantly.

```
// Vite project initialization
npm create vite@latest my-app --template react
// CRA initialization
npx create-react-app my-app
```

Why interviewers ask this:
Evaluates understanding of modern tooling and dev efficiency.

Common mistakes:

- Assuming CRA and Vite are interchangeable without considering performance and modern JS features.

Q15.11. What is fast refresh in Vite and why is it important?

Fast refresh allows real-time updates of React components without losing state, improving development speed and experience. Unlike CRA, Vite achieves this with ES module-based HMR, making updates near-instant.

```
// Vite automatically handles HMR on component save
```

Why interviewers ask this:
Tests understanding of developer tooling efficiency.

Common mistakes:

- Confusing fast refresh with full page reload, losing state frequently.

Q15.12. When would you choose Next.js over CRA for a large React application?

Next.js provides SSR, SSG, API routes, image optimization, and file-based routing. CRA is suitable for SPA only. For SEO-sensitive, data-heavy, or server-rendered apps, Next.js is preferred.

```
// Decision: use Next.js for SSR/SSG
// CRA for SPA projects without SSR needs
```

Why interviewers ask this:
Evaluates architectural decision-making for large-scale React apps.

Common mistakes:

- Using CRA for SEO-heavy or dynamic content projects, causing limitations.

16. REACT INTERNALS

Q16.1. What is the difference between Virtual DOM and Real DOM in React?

The Real DOM is the actual browser DOM that manipulates the UI elements and is costly to update frequently. The Virtual DOM is an in-memory lightweight representation of the Real DOM. React maintains the Virtual DOM and applies a diffing algorithm to calculate the minimal changes required, which are then efficiently applied to the Real DOM.

```
// Conceptual example
const vdom = React.createElement("div", null, "Hello");
// React compares this with previous VDOM to update the real DOM efficiently
```

Why interviewers ask this:
Tests understanding of core React rendering optimization and performance principles.

Common mistakes:

- Confusing Virtual DOM as a separate rendering engine; assuming it replaces the real DOM completely.

Q16.2. What is React Reconciliation and how does the diffing algorithm work?

Reconciliation is the process React uses to update the Real DOM efficiently. When state or props change, React generates a new Virtual DOM tree and compares it with the previous tree using its diffing algorithm. It then calculates the minimal set of changes (additions, deletions, updates) needed to sync the Real DOM. Key strategies include comparing element types, keys for lists, and subtree replacements.

```
// Example of key usage for reconciliation
<ul>
  {items.map(item => <li key={item.id}>{item.name}</li>)}
</ul>
```

Why interviewers ask this:
Assesses deep understanding of React's optimization and internal mechanisms.

Common mistakes:

- Failing to use keys in lists, leading to unnecessary re-renders or incorrect reconciliation.

Q16.3. What is React Fiber architecture and why was it introduced?

React Fiber is a reimplementation of the React core algorithm introduced to improve rendering responsiveness and concurrency. It allows breaking rendering work into units, prioritizing updates, and pausing or resuming work to ensure high-priority updates (like user input) are handled promptly. Fiber enables features like Concurrent Mode, Suspense, and time-slicing.

```
// Conceptual: Fiber allows React to pause, resume, and prioritize rendering work
const element = <App />;
ReactDOM.createRoot(root).render(element);
```

Why interviewers ask this:
Tests understanding of advanced React rendering internals and performance engineering.

Common mistakes:

- Assuming Fiber changes the programming model for components; not understanding it only affects internal scheduling.

17. REACT 19 FEATURES

Q17.1. What are the key new features introduced in React 19?

React 19 introduces a set of features focused on simplifying async workflows, improving form handling, and enabling better user experience during transitions. Major additions include Actions for handling async mutations, new hooks like useFormStatus, useFormState, and useOptimistic, improved Suspense integration, automatic batching enhancements, and better support for Server Components. These features reduce boilerplate around loading, error, and optimistic UI states while encouraging more declarative data mutations.

Why interviewers ask this:
Interviewers ask this to check whether the candidate is up to date with the React ecosystem and understands why React is evolving toward async-first and server-driven UI patterns.

Common mistakes:

- Candidates often list features without explaining their purpose, or confuse React 19 features with earlier additions like useTransition or Suspense from previous versions.

Q17.2. What are Actions in React 19 and why were they introduced?

Actions in React 19 provide a built-in pattern for handling asynchronous mutations such as form submissions or data updates. An Action is an async function that React can track automatically, enabling seamless handling of loading, success, and error states. This removes the need for manual useState flags and reduces complex side-effect logic in components.

```
async function submitForm(formData) {
  await saveUser(formData);
}
```

Why interviewers ask this:
This question evaluates whether a candidate understands modern React's move away from imperative state handling toward declarative async flows.

Common mistakes:

- Many candidates think Actions are just another name for event handlers or confuse them with Redux actions, missing their async lifecycle integration.

Q17.3. How does useFormStatus work and what problem does it solve?

useFormStatus allows components to read the current state of a form submission, such as pending or completed, without prop drilling. It integrates tightly with Actions and enables fine-grained UI updates like disabling buttons or showing loaders while a form is submitting.

```
const { pending } = useFormStatus();
<button disabled={pending}>Submit</button>
```

Why interviewers ask this:
Interviewers use this to test knowledge of React 19 form handling improvements and understanding of context-driven state propagation.

Common mistakes:

- A common mistake is trying to manage loading state manually with useState instead of relying on form status.

Q17.4. Explain useOptimistic and its role in improving user experience.

useOptimistic enables optimistic UI updates by temporarily assuming a successful result before an async operation completes. This allows the UI to feel faster and more responsive, especially in network-bound interactions. If the action fails, React can revert to the previous state.

```
const [optimisticState, addOptimistic] = useOptimistic(state, updater);
```

Why interviewers ask this:
This question checks whether the candidate understands perceived performance and modern UX strategies in React.

Common mistakes:

- Candidates often implement optimistic updates manually and overlook built-in hooks, or forget to handle rollback scenarios.

Q17.5. How does React 19 improve form handling compared to earlier versions?

React 19 introduces native form Actions, useFormState, and useFormStatus, eliminating much of the boilerplate previously required for managing form submission state, validation feedback, and loading indicators. This results in simpler, more maintainable form logic.

Why interviewers ask this:
Interviewers want to see if candidates understand how React is reducing reliance on external form libraries for common use cases.

Common mistakes:

- Many candidates still default to heavy form libraries without recognizing that React now covers many built-in scenarios.

Q17.6. What is the relationship between React 19 and Server Components?

React 19 strengthens integration with Server Components by enabling better async boundaries, streaming, and mutation handling through Actions. Server Components allow rendering logic to run on the server, reducing bundle size and improving initial load performance.

Why interviewers ask this:
This helps interviewers assess architectural understanding of modern full-stack React applications.

Common mistakes:

- Candidates often assume Server Components replace client components entirely or misunderstand where hooks can be used.

Q17.7. How does React 19 improve handling of loading and error states?

With Actions and enhanced Suspense support, React 19 automatically tracks async states and propagates them through the component tree. This reduces manual error handling and makes loading states more consistent and predictable.

Why interviewers ask this:
Interviewers ask this to see if candidates still rely on manual flags or embrace declarative async patterns.

Common mistakes:

- A common mistake is mixing old loading patterns with new Action-based flows, causing redundant state.

Q17.8. How does React 19 change the way developers think about side effects?

React 19 encourages moving side-effect-heavy logic into Actions and server boundaries, reducing reliance on useEffect for data mutations. This leads to clearer separation between rendering and side effects.

Why interviewers ask this:
This question distinguishes candidates who understand React’s long-term design philosophy.

Common mistakes:

- Candidates often continue using useEffect for mutations instead of embracing Action-driven flows.

Q17.9. Can React 19 reduce the need for external state management libraries?

Yes, for many applications. With Actions, useOptimistic, and Server Components, React 19 can handle common async and shared state patterns that previously required Redux or similar libraries. However, complex global state may still benefit from dedicated solutions.

Why interviewers ask this:
Interviewers want to see balanced judgment rather than extreme opinions.

Common mistakes:

- Claiming Redux is obsolete without understanding trade-offs is a common red flag.

Q17.10. What kind of applications benefit the most from React 19?

Applications with heavy async interactions, form-driven workflows, and server-rendered content benefit most. React 19 is particularly powerful for modern SaaS dashboards, content platforms, and data-intensive applications.

Why interviewers ask this:
This question tests the ability to apply technology choices to real-world scenarios.

Common mistakes:

- Candidates often answer generically without mapping features to actual use cases.