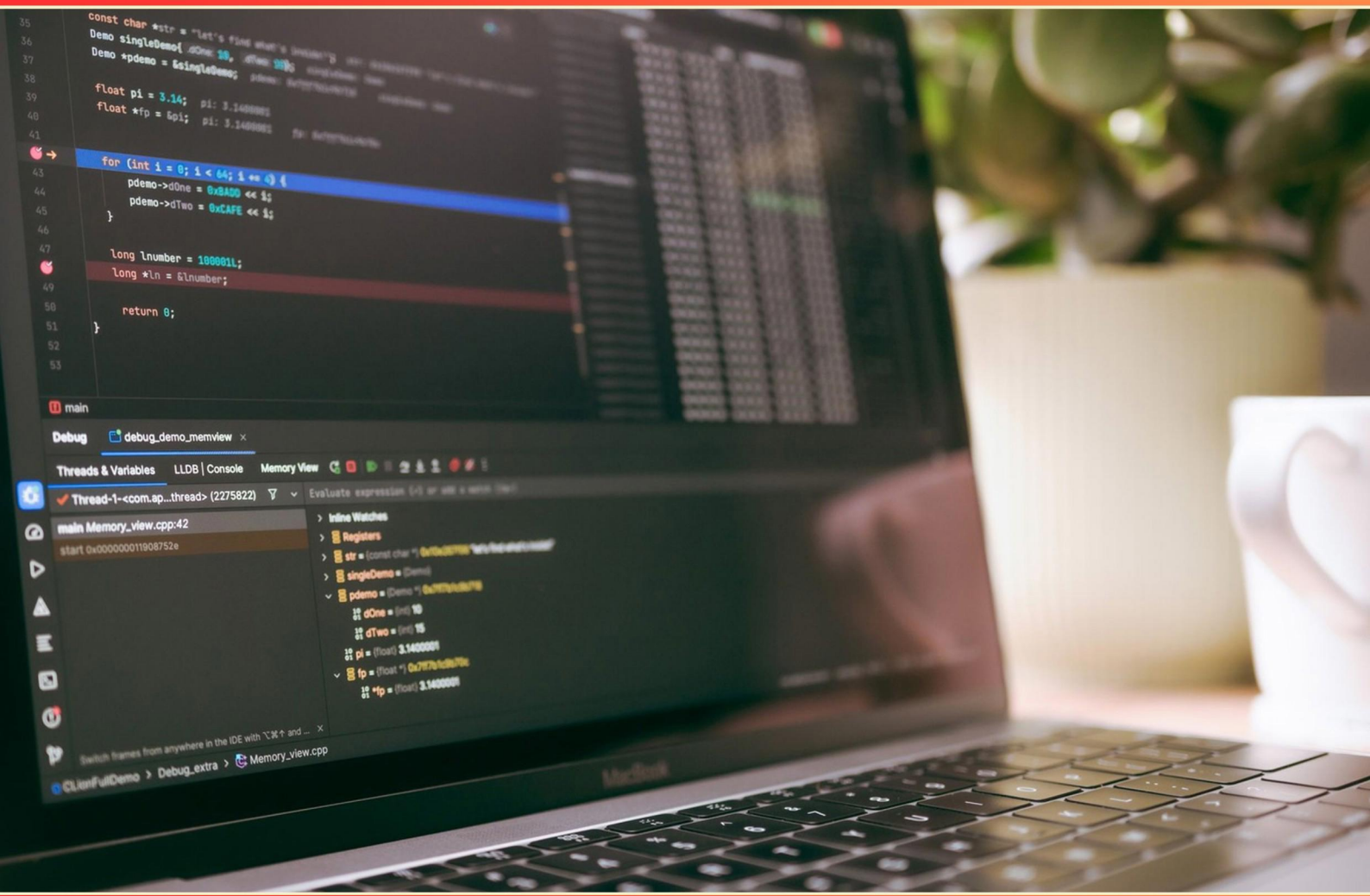
REACTJS PRACTICE TEST (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://reactjs.examzify.com

Table of Contents

Introduction

Preparing for a certification exam can feel overwhelming, but with the right tools, it becomes an opportunity to build confidence, sharpen your skills, and move one step closer to your goals. At Examzify, we believe that effective exam preparation isn't just about memorization, it's about understanding the material, identifying knowledge gaps, and building the test-taking strategies that lead to success.

This guide was designed to help you do exactly that.

Whether you're preparing for a licensing exam, professional certification, or entry-level qualification, this book offers structured practice to reinforce key concepts. You'll find a wide range of multiple-choice questions, each followed by clear explanations to help you understand not just the right answer, but why it's correct.

The content in this guide is based on real-world exam objectives and aligned with the types of questions and topics commonly found on official tests. It's ideal for learners who want to:

- Practice answering questions under realistic conditions,
- Improve accuracy and speed,
- Review explanations to strengthen weak areas, and
- Approach the exam with greater confidence.

We recommend using this book not as a stand-alone study tool, but alongside other resources like flashcards, textbooks, or hands-on training. For best results, we recommend working through each question, reflecting on the explanation provided, and revisiting the topics that challenge you most.

Remember: successful test preparation isn't about getting every question right the first time, it's about learning from your mistakes and improving over time. Stay focused, trust the process, and know that every page you turn brings you closer to success.

Let's begin.

How to Use This Guide

This guide is designed to help you study more effectively and approach your exam with confidence. Whether you're reviewing for the first time or doing a final refresh, here's how to get the most out of your Examzify study guide:

1. Start with a Diagnostic Review

Skim through the questions to get a sense of what you know and what you need to focus on. Your goal is to identify knowledge gaps early.

2. Study in Short, Focused Sessions

Break your study time into manageable blocks (e.g. 30 – 45 minutes). Review a handful of questions, reflect on the explanations.

3. Learn from the Explanations

After answering a question, always read the explanation, even if you got it right. It reinforces key points, corrects misunderstandings, and teaches subtle distinctions between similar answers.

4. Track Your Progress

Use bookmarks or notes (if reading digitally) to mark difficult questions. Revisit these regularly and track improvements over time.

5. Simulate the Real Exam

Once you're comfortable, try taking a full set of questions without pausing. Set a timer and simulate test-day conditions to build confidence and time management skills.

6. Repeat and Review

Don't just study once, repetition builds retention. Re-attempt questions after a few days and revisit explanations to reinforce learning. Pair this guide with other Examzify tools like flashcards, and digital practice tests to strengthen your preparation across formats.

There's no single right way to study, but consistent, thoughtful effort always wins. Use this guide flexibly, adapt the tips above to fit your pace and learning style. You've got this!

Questions

1. What is true about Error Boundaries in React?

A. Error Boundaries catch only errors in event handlers.

B. Error Boundaries are hooks-based functions.

C. Use an Error Boundary to catch render-time errors in child components and display fallback UI; they do not catch errors in event handlers.

D. Error Boundaries automatically fix errors in components.

2. Where must hooks be called?

A. Inside conditional blocks or loops.

B. After the render phase in lifecycle methods.

C. In class components only.

D. At the top level of a function component or a custom hook.

3. When should you use useCallback?

A. To memoize expensive calculations.

B. To manage state.

C. To memoize stable function references to prevent unnecessary re-renders.

D. To perform side effects.

4. What is the difference between server-side rendering and client-side rendering in React?

A. Server-side rendering renders HTML on the server for the initial page load; Client-side rendering renders markup in the browser after JavaScript runs, with hydration to reconcile server HTML with client code.

B. CSR renders markup in the browser after JavaScript runs; SSR renders HTML on the server for the initial load, with hydration to reconcile server HTML with client code.

C. SSR prints static HTML; CSR uses only CSS.

D. SSR uses Suspense; CSR uses portals.

5. A React component should use 'state' to store information that the component itself can change.

A. True

B. State is used to store static data from props

C. State should be mutated by external components

D. Props are used to manage internal changes

6. Props in a React component are intended to store information that can be changed by the component itself.

A. False

B. True

C. Depends on the Prop

D. Only With An Explicit Setter

7. Which statement about ReactDOM's purpose is accurate?

A. It is the library used to render React elements into the DOM.

B. It is the library used to manage component state.

C. It is used for server-side templating with React.

D. It provides a complete UI framework independent of the DOM.

8. What happens if you call a hook conditionally?

A. Calling a hook conditionally will not affect app behavior.

B. Hook order may change, leading to errors.

C. Conditional calls of hooks are allowed.

D. Calling a hook conditionally violates the Rules of Hooks and will lead to errors.

9. Can you access variables declared outside a JSX expression inside the JSX block?

A. No, you cannot access external variables.

B. Only variables defined inside the JSX block can be accessed.

C. External variables are ignored by JSX.

D. Yes, you can access variables declared outside the JSX expression.

10. Which statement best describes the public directory's purpose in a React app?

A. It contains source code that is transpiled.

B. It contains development-only scripts.

C. It contains static assets served directly without extra processing.

D. It stores test configurations.

Answers

1. D
2. D
3. C
4. B
5. A
6. A
7. A
8. D
9. D
10. C

Explanations

1. What is true about Error Boundaries in React?

A. Error Boundaries catch only errors in event handlers.

B. Error Boundaries are hooks-based functions.

C. Use an Error Boundary to catch render-time errors in child components and display fallback UI; they do not catch errors in event handlers.

D. Error Boundaries automatically fix errors in components.

Error boundaries focus on protecting the UI from render-time errors by catching exceptions that occur while a part of the component tree renders, in the lifecycle methods, or in constructors of child components. When such an error happens, the boundary renders a fallback UI instead of crashing the whole app, so users see something graceful while the rest of the interface remains usable. They do not catch errors in event handlers, so errors thrown inside click or submit handlers aren't handled by the boundary—you'd handle those inside the event handler itself or with local try/catch. Error boundaries are implemented as class components that define componentDidCatch and, optionally, getDerivedStateFromError; they are not hooks-based. They do not fix errors automatically; they merely provide a safe fallback UI when an error occurs.

2. Where must hooks be called?

A. Inside conditional blocks or loops.

B. After the render phase in lifecycle methods.

C. In class components only.

D. At the top level of a function component or a custom hook.

Hooks must be called at the top level of a function component or a custom hook. This ensures the order of every hook call stays the same across renders, which React relies on to correctly attach state and effects to each hook. Calling a hook inside a conditional, loop, or nested function would change that order from render to render. That breaks React's internal bookkeeping and can lead to errors or unpredictable behavior. By placing all hook calls at the top level, you guarantee a stable sequence, and you use the hook results later in render logic or effects as needed. Within a component, you should call the hooks unconditionally in the function body, and then use conditional logic to decide what to render or how to react based on the hook values. Similarly, a custom hook follows the same rule: its hook calls must be at the top level of the function, not inside conditionals. Other options miss this requirement because hooks aren't tied to lifecycle methods or class components, and they aren't meant to be invoked after rendering. The correct approach is to call them at the top level of a function component or a custom hook.

3. When should you use useCallback?

A. To memoize expensive calculations.

B. To manage state.

C. To memoize stable function references to prevent unnecessary re-renders.

D. To perform side effects.

useCallback is about keeping function references stable across renders. In React, a new function instance is created on every render, and if you pass that function down to a child component that relies on referential equality (for example a child wrapped with React.memo), the child can re-render unnecessarily just because the function prop changed. Wrapping the function with useCallback returns the same function instance unless its dependencies change, which helps prevent those extra re-renders and keeps performance where it matters. This doesn't memoize a computed value—useMemo does that. It also isn't about managing state or performing side effects; those are handled by useState and useEffect, respectively. So the correct idea is that useCallback memoizes stable function references to prevent unnecessary re-renders.

4. What is the difference between server-side rendering and client-side rendering in React?

A. Server-side rendering renders HTML on the server for the initial page load; Client-side rendering renders markup in the browser after JavaScript runs, with hydration to reconcile server HTML with client code.

B. CSR renders markup in the browser after JavaScript runs; SSR renders HTML on the server for the initial load, with hydration to reconcile server HTML with client code.

C. SSR prints static HTML; CSR uses only CSS.

D. SSR uses Suspense; CSR uses portals.

Rendering strategy in React is about where the initial HTML comes from. In client-side rendering, the browser runs the JavaScript bundle to build the UI, producing the markup in the DOM after the JS executes. In server-side rendering, the server generates the HTML for the initial page request and sends that HTML to the browser, so the user sees content right away without waiting for JavaScript to run. When the client loads the server-rendered page, React runs and hydrates that markup—reconciling the server HTML with the client-side code so event handlers and interactivity work correctly without replacing the entire DOM. Hydration is the key bridging step: it attaches the React components to the server-generated HTML, enabling interactivity while preserving the fast, content-first initial load that SSR provides. This explains why SSR often helps with perceived performance and SEO, while CSR can offer quicker interactivity after the bundle loads but may delay the initial visible content. The correct description matches this idea: the browser renders after JavaScript runs, while the server renders the initial HTML and the client hydrates it to become fully interactive. The other options either misstate what SSR or CSR do (for example, claiming CSS-only rendering, or tying SSR/CSR to features like Suspense or portals that aren't their defining difference).

5. A React component should use 'state' to store information that the component itself can change.

A. True

B. State is used to store static data from props

C. State should be mutated by external components

D. Props are used to manage internal changes

State is the place where a React component stores data it can change over time. When you update that data, React re-renders the component so the UI stays in sync with the current values. This is why the statement in the question matches how React components manage their own dynamic information: the component uses state to hold data it can alter, such as user input, toggles, counters, or form fields. Props, on the other hand, come from outside the component and are read-only inside it. They're useful for configuring a component, but the component itself shouldn't try to mutate them. If you need to reflect changes from outside, you'd typically lift state up or have the parent pass a new prop value, not mutate the prop directly. The idea that state should be mutated by external components is also not how React state works—the component owns its own state, and updates to it should go through the provided update mechanism (like setState or a setter from useState). Similarly, props are not meant to manage internal changes; they're the inputs the component receives from its parent. In short, state is the right tool for internal, changeable data, which is why the statement is correct.

6. Props in a React component are intended to store information that can be changed by the component itself.

A. False

B. True

C. Depends on the Prop

D. Only With An Explicit Setter

Props are inputs from a parent to a child and are meant to be read by the child, not changed by it. In React, data flows one way: from parent to child, and props are treated as immutable inside the component. If you need to reflect changes, store the value in the component's own state or have the parent update the prop by calling a callback (lifting state up). Mutating a prop (even if it's an object or array) is an anti-pattern because it can lead to bugs and won't reliably trigger re-renders; instead, create new values when updating and pass them down. So the statement is false.

7. Which statement about ReactDOM's purpose is accurate?

A. It is the library used to render React elements into the DOM.

B. It is the library used to manage component state.

C. It is used for server-side templating with React.

D. It provides a complete UI framework independent of the DOM.

ReactDOM's job is to render React elements into the browser's DOM and keep that DOM in sync as your components update. It acts as the bridge between React's component model and the actual page, mounting your React tree into a DOM node and applying efficient updates whenever state or props change. State management happens inside the React components themselves (with useState, useReducer, or class state), not in ReactDOM. Server-side rendering uses a different tool (ReactDOMServer), and ReactDOM is specifically for rendering to the DOM in web environments, rather than being a DOM-independent UI framework.

8. What happens if you call a hook conditionally?

A. Calling a hook conditionally will not affect app behavior.

B. Hook order may change, leading to errors.

C. Conditional calls of hooks are allowed.

D. Calling a hook conditionally violates the Rules of Hooks and will lead to errors.

Hooks must be called at the top level of a function component, in the same order on every render. Calling a hook inside a conditional means the number and order of hook invocations can differ from one render to the next. React relies on the sequence of hook calls to associate state and effects with the right places in your component; changing that sequence breaks this mapping and leads to errors. To use conditional behavior correctly, declare the hook unconditionally and place the conditional logic inside its effect or in the rendering logic after the hook has been called, or render different components conditionally. This ensures hook calls stay stable across renders, preventing runtime errors.

9. Can you access variables declared outside a JSX expression inside the JSX block?

A. No, you cannot access external variables.

B. Only variables defined inside the JSX block can be accessed.

C. External variables are ignored by JSX.

D. Yes, you can access variables declared outside the JSX expression.

Variables declared outside a JSX expression are available inside JSX because JSX is standard JavaScript and follows normal scope rules. To show their values, wrap the expression in curly braces within the JSX. For example, const userName = 'Alex'; return Welcome, {userName}! ; Here, userName comes from outer scope and is inserted into the rendered output. The same idea applies to props and state inside a component: return {props.label} or return {state.count} . If a variable isn't in scope, you'll encounter an error when rendering, which highlights that only in-scope variables can be accessed. So yes, you can access variables declared outside the JSX expression.

10. Which statement best describes the public directory's purpose in a React app?

A. It contains source code that is transpiled.

B. It contains development-only scripts.

C. It contains static assets served directly without extra processing.

D. It stores test configurations.

The public directory holds static assets that the app serves directly without going through the build or bundling process. Files placed here are not transformed by Webpack or Babel; they're copied to the final build as-is and can be accessed by absolute URLs like /logo.png. The main HTML file that bootstraps the app also lives here, and during the build these assets are made available at the root of the served site. This is why it's described as containing static assets served directly without extra processing. Other options point to things that belong elsewhere—source code lives in the src directory and is transpiled, development scripts live in the project configuration, and test configurations are stored separately—so they don't fit the public directory's role.

Next Steps

Congratulations on reaching the final section of this guide. You've taken a meaningful step toward passing your certification exam and advancing your career.

As you continue preparing, remember that consistent practice, review, and self-reflection are key to success. Make time to revisit difficult topics, simulate exam conditions, and track your progress along the way.

If you need help, have suggestions, or want to share feedback, we'd love to hear from you. Reach out to our team at hello@examzify.com.

Or visit your dedicated course page for more study tools and resources:

https://reactjs.examzify.com

We wish you the very best on your exam journey. You've got this!