React Cheatsheet For 2025

Your Interview Handy Guide

Stefania Simon

Follow

Published in
JavaScript in Plain English

8 min read

Jan 13, 2025

Whether you're an experienced developer looking to validate your skills or a newcomer preparing for your first React interview, you may find this guide suitable for your needs, as it aims to equip you with the knowledge and confidence needed to tackle your upcoming challenges.

I've been working on a curated list of questions and verified resources, so let's dive into the world of React, a powerhouse in front-end development.

What does it mean that a React Component has been mounted?

- the term *mounted* refers to the lifecycle phase in which a component has been successfully rendered and added to the DOM (Document Object Model);
- when a component is *mounted*, it means that the component's `render` method has been called, and the resulting virtual DOM elements have been translated into actual HTML elements that are now part of the document;
- the *mounted* state is a suitable phase for performing tasks that require access to the DOM or interactions with external resources, as the

component is guaranteed to have a representation in the actual HTML document;

What are the key differences between Controlled and Uncontrolled Components?

- they are different approaches to managing and handling the state of form elements within a component;
- *controlled*: the state of the form elements (like input fields, checkboxes, etc.) is controlled by the React *component state* — which is *the single source of truth*, and any changes are handled through the React state updates; this approach is better for data validation;
- *uncontrolled*: the state of the form elements is directly managed by the DOM itself, and the values can be accessed using refs or DOM queries, but validation can be done only after submitting the form;

What triggers a Component to re-render?

- *state updates*: either via the setState function (in class components) or the useState hook (in function components);
- *props changes*: either in value (for primitive types) or reference (for reference types);
- *re-render* of a *parent component*: when a parent component re-renders, all its children re-render by default, unless they are memoized (React.memo);
- *reducer dispatch*: when the dispatch function from the `useReducer` hook is called, the component re-renders if the state returned by the reducer changes;
- *context value updates*: if a component consumes a React Context and the value provided by the Provider changes, a re-render will be triggered;

What are the differences between class components' lifecycle methods and functional components' hooks?

- *lifecycle methods* are predefined, overridden inside the *class*, and separated into mounting, updating, and unmounting phases;
- in *functional* components*, hooks* like `useEffect`, `useState`, `useRef`, and others manage state *and* lifecycle logic;
- *lifecycle methods* have a fixed structure; you must use them as React provides (`componentDidMount`, `componentDidUpdate`), and these two methods typically handle side effects;
- `useEffect` hook is more flexible and allows combining logic for multiple phases (mounting, updating, unmounting) in a single place; side effects are handled by variations of dependencies, allowing you to control when the effect runs;
- for class components, *cleanup* is usually placed in the `componentWillUnmount` method; for functional components, *cleanup* is defined in the return function of `useEffect`;
- for class components, logic is tightly coupled to the element, and sharing logic requires higher-order components (*HOCs*) or *render props*; in *functional* components, hooks allow reusable logic by creating *custom hooks*;

What is the reconciliation process?

- *reconciliation* is the way React updates the DOM to reflect changes in the component tree efficiently, it's a key part of React's performance optimization and is made possible by the *Virtual DOM;*
- React maintains a Virtual DOM (a lightweight in-memory representation of the actual DOM), and when a component's state or props change, React creates a new Virtual DOM tree to represent the updated UI;
- *Diffing Algorithm*: React compares the old Virtual DOM tree with the new one by using a highly optimized diffing algorithm to identify what has changed, then it calculates the *minimal* set of changes and updates only the parts of the real DOM that need to be updated;

What is the significance of the `key` prop in React lists, and how does it impact performance?

- the `key` prop is a special attribute used by React to identify each element in a list uniquely, and to determine which elements in the DOM need to be updated, added, or removed when the list changes;
- when correctly set, React only updates the DOM for elements that have changed, improving rendering performance, avoiding full re-renders, and retaining the states of components that have not changed;
- if incorrectly set, React may unnecessarily update all elements in the list, even if only one item has changed; if the `key` is based on the *array index* (which can shift as items are added or removed), React may discard the state of components if their position in the list changes;

What are some significant built-in hooks other than `useState` and `useEffect`?

- `useMemo`: optimizes performance by *memoizing* the *result* of an expensive computation, preventing unnecessary recalculations;
- `useCallback`: *memoizes* a *function* so that it does not get recreated on every render;
- `useReducer`: used to manage complex state logic using a *reducer function*, similar to *Redux*, especially for *state* that involves multiple sub-values or more complex state transitions;
- `useLayoutEffect`: similar to `useEffect`, but fires synchronously after all DOM mutations, which ensures that the effect runs before the browser paints, and it's useful for measuring DOM elements or making layout adjustments;

What are Higher-Order Components (HOCs) and how do they differ from Render Props?

- a *HOC* is a function that takes a component (as a parameter) and returns a new enhanced component, allowing the reuse of logic across multiple components by wrapping them with a common *enhancing function*;

- *Render Props* is a technique where a component accepts a function as a prop and uses it to render something; the function provides the necessary logic or data to the children and allows control over what to render;
- HOCs can be used when a component needs to be *enhanced* in a reusable way, fine-grained control is not needed, and the component's structure doesn't need to be modified;
- Render Props are more *flexible* and allow control over rendering, used for highly dynamic UI patterns or logic that changes frequently;

What are Portals and when should they be used?

- `createPortal` is a function in React that allows rendering a child component into a DOM node that exists outside the DOM hierarchy of the parent component;
- they are ideal for rendering UI components that need to break free from the parent layout constraints, and common use cases include *modals*, *tooltips*, *dropdowns*, and *notifications*;
- some benefits include escaping overflow or positioning issues, improved accessibility and usability;

What are `refs` in React and when are they used?

- a `ref` is an *object* with a single property called `current` which can be read or set, it usually retains information between re-renders of a React component — its value can be mutated without triggering a re-render;
- `(!)forwardRef` has been used for *passing* a ref from a parent component to a child component, giving the parent direct access to the child's DOM element or custom instance;
- `(!)forwardRef` is *deprecated* starting with React 19, `ref` can now directly be accessed prop for function components;
- the `useImperativeHandle` hook can be used with refs to expose only some specific functionality of a component, rather than the entire DOM node:

it allows customizing the instance value exposed to the parent via the ref;

What are the differences between the Context API and more traditional state management libraries?

- *Context API* is a feature that allows sharing state or data globally across components without having to pass props manually through every level of the component tree (*prop drilling*);
- `useContext` is the hook that enables reading and subscribing to *context* from the component;
- used for managing *global state* in small to medium-sized projects, unlike more robust state management libraries (*Redux/Mobx*), which are designed for complex state management across large apps;
- *Context API* only provides a single value or object, but any number of stores can be defined in one application, and it uses a *Provider/Consumer* pattern;
- *Redux* provides advanced features like reducers, stores, and actions, and works with a centralized store as the single source of truth, while *Mobx* can also have any number of stores and uses the *Observable-reactions* pattern;

What is server-side rendering (SSR) in React, and how can it benefit an application?

- *SSR* is a technique where React components are rendered on the server as HTML and sent to the browser — unlike *Client-Side Rendering (CSR),* where the browser renders the React application after JavaScript is downloaded and executed;
- when a user requests a page, the server processes the request, executes the React components, generates an HTML string with the content, and sends the pre-rendered HTML to the browser;
- the last step is *hydration*: React “hydrates” the static HTML on the client side by attaching event listeners and making the page interactive;

- as *benefits*, it's better for *SEO* (Search Engine Optimization) — as search engines can crawl pages much faster — and faster *Time-to-First-Byte* (TTFB), better performance for the initial load, and improved accessibility;
- as *drawbacks*, the server load increases because it has to render the HTML for each request, has slower interactivity — the browser may take longer to hydrate pages, difficult to manage async data fetching and cache dynamic data;
- *Next.js* is the most popular framework built on top of React that uses SSR;

What is lazy loading in React, and how does it contribute to code splitting and improved performance?

- *lazy loading* is a design pattern in React that delays the loading of a component or resource until it is required;
- the default behavior involves *bundling* all components and their dependencies together, which can result in large bundle sizes, slowing down the initial load time;
- *lazy loading* involves splitting components into smaller bundles, and only the required parts are loaded when needed, improving performance and reducing the time-to-first-byte;
- *lazy loading* enables code splitting by dynamically importing components — using `React.lazy()`, while `React.Suspense` provides a fallback UI until the lazy-loaded component is ready;

What are React's Concurrent features?

- introduced in *React 18*, *concurrent features* enable React to work on multiple tasks simultaneously by making rendering interruptible, which allows prioritizing urgent tasks over less critical ones, improving responsiveness and user experience;
- *automatic batching*: previously, React only batched updates inside event handlers; now updates are batched automatically across all

asynchronous operations which reduces unnecessary renders and improves performance;

- `startTransition`: marks updates as *non-urgent* or *low-priority*, so that urgent updates (like clicks or typing) are prioritized over transitions;
- `useDeferredValue` hook: it allows deferring updates to certain values until more urgent updates have been processed, for optimizing performance when rendering expensive components;

What are some essential new features introduced in React 19?

- *actions*: simplify state updates by allowing the use of asynchronous functions within transitions (`startTransition`); this automates the handling of pending states, errors, forms, and optimistic updates, reducing the need for manual state management;
- *Server Components*: enable building applications that seamlessly integrate *client and server rendering*, which offers improved performance and user experience by offloading certain components to the server;
- `useActionState` hook: manages state related to *form actions*, simplifying the handling of pending and error states;
- `useFormStatus` hook: provides status information of the last form submission, aiding in form validation and user feedback;
- `useOptimistic` hook: facilitates optimistic UI updates by allowing immediate UI changes while awaiting server confirmation;
- document *metadata* support: allows managing document metadata (`title`, `meta`) directly within React components, streamlining SEO management and improving integration with various tools and platforms;
- *custom elements* support: enables better interoperability and allows incorporating web components seamlessly into React applications;