

Q1. What is React Native and what UI problem does it address?

A: React Native is a component-based library or framework for building interactive user interfaces. It provides a reactive programming model where UI updates automatically when underlying state changes, reducing imperative DOM manipulation.

Q2. How do components communicate in React Native?

A: Parent-to-child communication uses props (one-way data flow). Child-to-parent uses events or callbacks. Siblings share state via a common parent or a global store (Redux, Pinia, Signals). Avoid prop-drilling by using context or state management.

Q3. How does React Native manage state for complex applications?

A: Local component state handles UI-only concerns. Shared state (user auth, cart) lives in a global store with a predictable update pattern (actions, reducers, or mutations). Server state (API data) is managed by libraries like React Query or SWR.

Q4. What rendering strategies does React Native support?

A: React Native supports CSR (browser renders), SSR (server renders HTML on each request), SSG (pages generated at build time), and ISR (pages regenerated on a schedule or on-demand). Choose based on SEO requirements and data freshness needs.

Q5. How does React Native handle client-side routing?

A: React Native's router (built-in or a library like React Router) maps URL paths to components without page reloads. Dynamic segments (/users/:id) are parsed and passed as params. Guards or middleware can protect routes requiring authentication.

Q6. How do you fetch data and handle loading/error states in React Native?

A: Use a data fetching hook or library that returns { data, isLoading, error }. Show a skeleton or spinner while loading, an error message on failure, and the data on success. Avoid fetching in render to prevent waterfalls.

Q7. What performance techniques apply to React Native applications?

A: Code splitting loads only the code needed for the current route. Memoization (useMemo, React.memo) prevents unnecessary re-renders. Virtualization renders only visible list items. Images use lazy loading and next-gen formats (WebP, AVIF).

Q8. How do you handle forms and validation in React Native?

A: React Native manages form state with controlled inputs or a form library (React Hook Form, VeeValidate). Validation runs on blur or submit, displaying field-level error messages. Schema libraries (Zod, Yup) define validation rules declaratively.

Q9. How do you test React Native components?

A: Unit tests check individual component rendering and interactions using a component testing library. Integration tests verify multi-component flows. End-to-end tests (Playwright, Cypress) simulate real user journeys in a browser.

Q10. What accessibility (a11y) practices are essential in React Native?

A: Use semantic HTML (button, nav, main) instead of divs with click handlers. Ensure keyboard navigability (tab order, focus management). Add ARIA roles and labels for dynamic content. Test with a screen reader and run axe or Lighthouse audits.