ChatGPT

Sheet 1: React Native Interview Questions

1. **What is React Native?**
2. **How is React Native different from ReactJS?**
3. **What are the advantages of using React Native?**
4. **What is JSX in React Native?**
5. **What is the Metro Bundler in React Native?**
6. **How does React Native achieve near-native performance?**
7. **What are React Hooks and give examples (e.g. useState, useEffect)?**
8. **How do you handle navigation between screens in React Native?**
9. **What is the difference between controlled and uncontrolled components?**
10. **What is the difference between state and props in React Native?**
11. **What are keys in lists, and why are they important in React Native?**
12. **How can you optimize performance in a React Native app?**
13. **What is the difference between ScrollView and FlatList?**
14. **What is SafeAreaView and when would you use it?**
15. **What is AsyncStorage and when should you use it?**
16. **Why should you avoid placing API calls directly in the component body?**
17. **Why should you not mutate React state directly?**
18. **What is the difference between storing a token in state vs AsyncStorage?**
19. **Why might the UI look different on Android and iOS even with the same code?**
20. **Why is testing on a real device important before releasing an app?**
21. **If an AI suggests adding a heavy library for a small task, how would you respond?**
22. **How do you handle network failures when fetching data in React Native?**
23. **What is CodePush (over-the-air updates) in React Native?**
24. **What is the purpose of app permissions in mobile apps?**
25. **When should you use loading indicators in a mobile app?**
26. **How would you debug an issue where the release build crashes or shows a blank screen, but the debug build works?**
27. **What is a Higher-Order Component (HOC) in React Native?**
28. **What is the Fabric architecture in React Native?**
29. **What are TurboModules in React Native?**
30. **What is the Hermes JavaScript engine in React Native?**
31. **How do you integrate a native module (Java/Kotlin or Swift/Objective-C) in a React Native app?**
32. **How do you handle form inputs or controlled components in React Native?**
33. **How do you share state across multiple components (Context API vs Redux)?**
34. **What is useRef used for in React Native?**
35. **How and when would you use useMemo and useCallback hooks?**
36. **What is LayoutAnimation in React Native?**
37. **What is InteractionManager and why is it useful?**
38. **What are DeviceEventEmitter and NativeEventEmitter used for?**
39. **What is PanResponder used for in React Native?**
40. **What is the Animated library used for in React Native?**
41. **What are Error Boundaries and why are they important?**
42. **What is WebView and when might you use it?**
43. **What is NavigationContainer in React Navigation?**

44. **What is a VirtualizedList in React Native?**
45. **Why is reusing components important in mobile development?**
46. **How do you request and manage permissions (e.g. camera, location) in React Native?**
47. **What is the Platform module and how do you use it for conditional code or styling?**
48. **How do you implement smooth animated transitions between screens?**
49. **What is useLayoutEffect and how is it different from useEffect?**
50. **How would you implement background tasks or headless JS in a React Native app?**

Sheet 2: Questions and Answers

1. **What is React Native?**
   React Native is an open-source framework by Meta (Facebook) for building **cross-platform mobile apps** using JavaScript and React. It lets you write app code in JS and JSX that **compiles to native components**, giving you near-native performance on Android and iOS [1].

2. **How is React Native different from ReactJS?**
   ReactJS (React) is for building **web applications** (runs in a browser, uses HTML/DOM), whereas React Native is for **mobile apps** (uses native mobile UI components like `<View>`, `<Text>`, `<Image>` instead of HTML tags) [2]. Both use JSX and React concepts, but their rendering targets and built-in components differ.

3. **What are the advantages of using React Native?**
   React Native allows **cross-platform development** (one codebase for Android and iOS), **fast refresh (hot reloading)** for quick iteration, and **reusable components** across the app. It has strong community support and achieves near-native performance. According to one source, RN provides cross-platform support, hot reloading, reusable components, and "native-like performance" [3].

4. **What is JSX in React Native?**
   JSX stands for *JavaScript XML*. It is a syntax extension that lets you write UI code in a syntax similar to HTML (or XML) directly in your JavaScript. In React Native, JSX elements look like HTML (e.g. `<View>`, `<Text>`), but under the hood they compile to `React.createElement` calls or similar. In other words, JSX in React Native is syntactic sugar for calling `React.createElement` to create native components [4].

5. **What is the Metro Bundler in React Native?**
   Metro is the JavaScript bundler that ships with React Native. It scans your JavaScript and asset files, resolves all imports, and compiles them into a single bundle that the app loads at runtime. Essentially, Metro takes your code and resources and packages them into one optimized JS bundle for the device to execute [5].

6. **How does React Native achieve near-native performance?**
   React Native uses a *bridge* to communicate between JavaScript and the platform's native UI. All UI elements are rendered with the device's **native APIs** rather than a WebView. In other words, RN maps React components to native widgets. For example, a `<Button>` in RN becomes a `UIButton` on iOS or an `android.widget.Button` on Android. This direct use of native components ensures that the app's performance is close to a fully native app [6].

7. **What are React Hooks and give examples (e.g. useState, useEffect)?**
   Hooks are special functions introduced in React 16.8 that let you use React state and lifecycle

features in *functional* components. For example, `useState` lets you add state to a functional component, and `useEffect` lets you run side-effects (like data fetching) after a render. Other common hooks include `useRef`, `useMemo`, and `useContext`. They allow you to use state, refs, context, and other React features without writing a class component [7].

8. **How do you handle navigation between screens in React Native?**
   Most developers use a navigation library. The most popular is **React Navigation**, which provides stack, tab, and drawer navigators. Another option is **React Native Navigation** by Wix. Using React Navigation, you set up navigators (e.g. a `StackNavigator`) and call `navigation.navigate('ScreenName')` to switch screens. The navigation library manages the screen stack, transitions, and parameters passed between screens [8] [9].

9. **What is the difference between controlled and uncontrolled components?**
   A **controlled component** is one where form inputs (like `<TextInput>`) have their value bound to React state, and any change updates the state. An **uncontrolled component** holds its own state internally (like a plain HTML input without React state) and you access the value via a ref. In React Native (and React), controlled inputs are managed by state variables and callbacks, whereas uncontrolled ones bypass React's state for simplicity. For example, a controlled `<TextInput>` would have `value={someState}` and `onChangeText={setSomeState}`, whereas an uncontrolled input might use a ref to read its value [10] [11].

10. **What is the difference between state and props in React Native?**
    In React Native (and React in general), **state** is managed internally by a component; it holds data that may change over time (with `setState` or `useState`). **Props** are data passed *into* a component by its parent and are read-only within the child. In other words, state is mutable (the component can update it) and is local to that component, whereas props are immutable values given from outside. The devlog explains it well: *"props are passed down from parent to child, state is managed internally by the component. Props are immutable, state can change via setState"* [12].

11. **What are keys in lists, and why are they important?**
    In React Native lists (e.g. using `FlatList` or mapping over arrays), each item should have a unique **key**. Keys help React identify which items have changed, been added, or removed. They optimize rendering by allowing React's diffing algorithm to match items between renders. Without keys (or with non-unique keys), you can get duplicate UI or unnecessary re-renders. For example, using a stable ID as `key` ensures React can reorder or remove list items efficiently [13].

12. **How can you optimize performance in a React Native app?**
    Some common strategies are: use **FlatList** (or `VirtualizedList`) instead of rendering large lists with a plain `ScrollView` (FlatList lazily renders only visible items) [14], optimize images (e.g. resize or use appropriate formats), and **avoid unnecessary re-renders** (e.g. use `React.memo`, `useMemo`, `useCallback`). Other tips include minimizing expensive work on the JavaScript bridge, offloading work to native modules if needed, and reducing component depth. In short, render only what's needed and use memoization hooks when appropriate [14].

13. **What is the difference between ScrollView and FlatList?**
    A `ScrollView` renders all of its child components at once, which is fine for a **small** list. However, for long lists, this can be very slow and memory-intensive. A `FlatList` (or `SectionList`) only renders items that are currently visible on screen and a few buffer items,

which is much more performant for large datasets. In short, use `ScrollView` for short lists or static content, and use `FlatList` when you have large or dynamically loaded data [15].

14. **What is SafeAreaView and when would you use it?**
    `SafeAreaView` is a component that ensures your content is rendered within the "safe area" boundaries of a device (e.g. below the status bar or notch, and above the home indicator on iPhones). On devices with notches or rounded corners, it prevents your UI from overlapping those areas. In practice, you wrap the top-level view in `SafeAreaView` (or use `SafeAreaProvider`) so that on iOS especially, things like the status bar or notch don't cover your content [16].

15. **What is AsyncStorage and when should you use it?**
    `AsyncStorage` is a simple, asynchronous key-value storage system for React Native (similar to localStorage on web). It's used to persist small pieces of data on the device, such as user preferences or auth tokens. For example: `await AsyncStorage.setItem('user', JSON.stringify(userData))`. It persists across app restarts. (Note: `AsyncStorage` is now moved to the community-maintained `@react-native-async-storage/async-storage` package.) For sensitive data (like tokens), consider secure storage like Keychain/EncryptedStorage instead of plain AsyncStorage [17].

16. **Why should you avoid placing API calls directly in the component body?**
    Placing an API call directly in the component's render/body means it will run on every render (which could be infinite loops) and you have no control over it. Instead, you should perform side-effects like data fetching in a `useEffect` hook or in a lifecycle method (like `componentDidMount`). This ensures the call happens at the right time (for example, once on mount) and you can manage loading and error states. It also keeps the component code pure and easier to reason about.

17. **Why should you not mutate React state directly?**
    React relies on immutability to detect changes. If you modify an object or array in state directly (e.g. pushing into an array), React may not detect the change and may not re-render. This breaks the predictable state model and can cause bugs. Always use state setters (`setState` or the updater from `useState`) and return a new object/array instead of mutating the old one. Immutability also makes undo/redo and change tracking easier.

18. **What is the difference between storing a token in state vs AsyncStorage?**
    State (memory) is reset whenever the app is closed or refreshed, so it cannot persist a token across sessions. `AsyncStorage` (or secure storage) can persist data even if the user closes the app or the OS kills it. Typically, we retrieve the token from AsyncStorage on app startup to stay logged in. Also note: plain AsyncStorage is not encrypted, so for sensitive tokens consider Keychain (iOS) or encrypted storage libraries.

19. **Why might the UI look different on Android and iOS even with the same code?**
    Android and iOS have different default styles and system UI. For example, default fonts, button styles, and layout behaviors can differ. Also, pixel densities and status bar heights vary. Some components have platform-specific props (like `Picker` or `DatePicker`). To manage this, you can use the `Platform` module or conditional styling. In practice, you often tweak styles (margins, paddings) to make the app look consistent on both platforms.

20. **Why is testing on a real device important before releasing an app?**
    Emulators and simulators cannot capture all real-world conditions: hardware performance, real sensors (GPS, camera, accelerometer), native permission dialogs, background/foreground transitions, battery conditions, and different OS versions. Testing on real devices catches issues like touch sensitivity, animations lag, and permission flows. It ensures the app actually works in the environments end users will use, leading to better quality and user satisfaction.

21. **If an AI suggests adding a heavy library for a small task, how would you respond?**
    Always critically review code suggestions. For a small task, adding a large dependency can bloat the app and degrade performance. It's often better to implement a simple function yourself or find a lightweight library. In interviews, you might answer: *"I would avoid unnecessary heavy dependencies; it's better to write a small custom solution or use a smaller library to keep the app performant."* This shows awareness of bundle size and performance.

22. **How do you handle network failures when fetching data in React Native?**
    You should implement error handling and retries for network requests. Using an HTTP library like Axios, you can set up an interceptor to catch errors (e.g. no connection) and respond accordingly (retry, show a toast, etc.). You can also detect connectivity with `NetInfo` and show offline UIs. In general, you'd show a loading indicator, catch any errors, and update the UI to show an error message or fallback content if the network call fails. For example, wrap fetches in try/catch, or use Axios's `.catch()` to handle failures gracefully.

23. **What is CodePush and how does it relate to over-the-air updates?**
    CodePush (now part of Microsoft App Center) allows React Native apps to receive JavaScript code updates over-the-air without going through the app store. You can push bug fixes or small feature changes to users' devices directly. It's useful for non-critical updates. Under the hood, it downloads a new JS bundle to the device and applies it on next app start or restart. It's important to use it carefully, as it cannot update native code or major app framework changes.

24. **What is the purpose of app permissions in mobile apps?**
    App permissions (like camera, location, microphone) are enforced by the OS to protect user privacy and security. Before an app can access sensitive data or hardware, the user must grant permission. This prevents malicious apps from secretly using features like contacts, camera, or GPS. In React Native, you must declare required permissions (especially on Android via `AndroidManifest.xml`) and usually prompt the user at runtime (e.g. with the `react-native-permissions` library) to obtain consent.

25. **When should you use loading indicators in a mobile app?**
    You should show a loading indicator or spinner whenever the app is performing a time-consuming task (like fetching data, processing images, or any operation that makes the user wait). For example, when the app launches and is retrieving user data, or when loading a new screen's content. Indicators communicate to the user that work is in progress, preventing confusion about unresponsive UI. Typically, you show it while `isLoading` state is true and hide it once the data has arrived.

26. **How would you debug an issue where the release build crashes or shows a blank screen, but the debug build works?**
    Release builds often enable optimizations (minification, proguard/R8) and disable dev tools. Common steps: check device logs (Xcode's Console or `adb logcat`) for crash traces, enable source maps (for JS errors), and test on a real device. Sometimes disabling minification or extra

optimizers helps isolate the problem. Ensure all native modules are correctly linked and environment variables or build configs are set for release. Tools like Sentry or Flipper can capture errors in production. In short, gather logs and compare configurations between debug and release to find discrepancies.

27. **What is a Higher-Order Component (HOC) in React Native?**
    A **Higher-Order Component** is a function that takes a component and returns a new, enhanced component. It's a pattern used to add reusable functionality (like logging, theming, or data fetching) to many components without repeating code. For example, a `withAuth` HOC might wrap a component and inject authentication props. According to one source: *"A HOC takes a component as input and returns an enhanced version of it"* [18]. HOCs are a way to reuse component logic.

28. **What is the Fabric architecture in React Native?**
    Fabric is the new React Native rendering system (part of the new architecture). It replaces the old asynchronous JSON bridge with a faster, more direct communication layer using the JavaScript Interface (JSI). Fabric aims to improve UI performance and reduce latency (no JSON serialization). It provides a more synchronous, layout-driven rendering pipeline and supports features like concurrent rendering. In summary, Fabric's goal is smoother animations and more efficient updates [19].

29. **What are TurboModules in React Native?**
    TurboModules are the new way of writing native modules in the upcoming RN architecture. They allow native modules to be initialized lazily and be called directly from JavaScript via JSI, reducing bridge overhead. They are "improved native modules for faster access" as RN's docs say. In practice, TurboModules make it quicker to call native APIs from JS. (The DevLog notes *"Improved TurboModules for faster native module access"* in newer RN versions [20].)

30. **What is the Hermes JavaScript engine in React Native?**
    Hermes is an open-source JavaScript engine (by Meta) optimized for React Native. It compiles JS to bytecode ahead-of-time, leading to faster app startup and lower memory usage. As of recent RN versions, Hermes is often enabled by default to improve performance. For example, one reference notes: *"Hermes engine is now the default for all builds"* in the 2025 React Native updates [20].

31. **How do you integrate a native module (Java/Kotlin or Swift/Obj-C) in a React Native app?**
    To integrate native code, you write a native module on each platform and expose its methods to JavaScript. For Android, you implement Java/Kotlin classes and register them (extend `ReactContextBaseJavaModule`); for iOS, you write an Objective-C/Swift class and export it to React Native. Then you can call those methods from JS. In general, *"You write native code in Java or Swift and expose it using the Native Modules API"* [21]. Tools like the React Native CLI or modules (e.g. `react-native-create-bridge`) can scaffold this.

32. **How do you handle form inputs (controlled components) in React Native?**
    In React Native, `<TextInput>` is the core input component. A controlled form input means its value is driven by state. You typically keep each input's value in a state variable (`useState` or Redux) and update it via `onChangeText`. For example: `<TextInput value={username} onChangeText={setUsername} />`. Libraries like Formik or React Hook Form can simplify handling form state and validation. Uncontrolled inputs (using refs) are possible but less common in React.

33. **How do you share state across multiple components (Context API vs Redux)?**
    For global state, one can use React's **Context API** or a state management library like Redux/MobX. Context allows you to create a provider that holds state (e.g. user, theme) and then any nested component can consume it with `useContext`, avoiding prop drilling. Redux centralizes state in a store with reducers. Context is simpler for smaller apps; Redux scales better for very large state. The key idea: lift shared state up or use a global store so different screens/components can access the same data without passing props around.

34. **What is useRef used for in React Native?**
    The `useRef` hook returns a mutable ref object that persists across renders. It's commonly used to access underlying native UI components or to hold mutable values without causing re-renders. For example, you can attach a ref to a `<TextInput ref={inputRef} />` to imperatively focus it (`inputRef.current.focus()`). Or you can use `useRef` to store a mutable count or timeout ID that doesn't trigger a state change. Essentially, `useRef` gives you a container to store a value that survives re-renders [22].

35. **How and when would you use useMemo and useCallback hooks?**
    `useMemo` and `useCallback` are React hooks for performance optimization. **useMemo** memoizes (caches) the result of an expensive function call so it only recomputes when its dependencies change. **useCallback** memoizes a callback function itself so that its identity doesn't change on every render unless dependencies change. Use them to prevent unnecessary re-calculations or re-renders. For example, wrap an expensive computation in `useMemo` or use `useCallback` when passing a function prop to a child component. According to one source, both hooks *"help improve performance by memoizing values and functions"* [23].

36. **What is LayoutAnimation in React Native?**
    `LayoutAnimation` is a built-in API for animating layout changes (like height/width) automatically. When you change the layout of views (for example, toggling the visibility of an item or adjusting margins), you can call `LayoutAnimation.configureNext(...)` and the framework will smoothly animate the change. It's simpler than manual animated values, as it animates the transition of layout properties. LayoutAnimation supports animations like scaling, fading, and sliding without writing explicit `Animated` code [24].

37. **What is InteractionManager and why is it useful?**
    `InteractionManager` lets you schedule work to run *after* animations or interactions are complete. It provides methods like `runAfterInteractions`. It's useful when you have tasks (e.g. heavy computations, data fetching) that you want to defer so they don't interfere with user interactions and UI responsiveness. By using `InteractionManager`, you ensure smoother animations or scrolling because the expensive work will wait until after touches and animations have settled [25].

38. **What are DeviceEventEmitter and NativeEventEmitter?**
    These are React Native modules for handling events from the native side. `DeviceEventEmitter` (part of core RN) lets native code emit events that JS can listen for (e.g. a sensor update). `NativeEventEmitter` is a wrapper that provides a consistent API for these events. In practice, you use them to communicate asynchronous data or callbacks from native modules to JavaScript. For example, if a native SDK posts an event, you can subscribe in JS with `DeviceEventEmitter.addListener(...)` [26].

39. **What is PanResponder used for in React Native?**
`PanResponder` is a low-level gesture handler. It lets you handle complex touch gestures (like dragging, swiping, pinch-to-zoom) by providing callbacks for touch start/move/end events. You create a PanResponder that defines `onPanResponderMove`, `onPanResponderGrant`, etc., and attach it to a view. It's essential when you need to implement custom gestures beyond simple taps (e.g., drag-and-drop or swiping items in a list) [27].

40. **What is the Animated library used for?**
React Native's `Animated` library provides a powerful, declarative API for creating smooth and complex animations. It allows you to animate properties like position, opacity, scale, and more, typically using `Animated.Value` and `Animated.timing`/`spring`. You can sequence or parallelize animations, and they run on the native UI thread for performance. In short, use `Animated` for performance-critical or custom animations (like moving a view or fading content) [28].

41. **What are Error Boundaries and why are they important?**
Error Boundaries are React components that catch JavaScript errors in their child component tree during rendering, lifecycle methods, and constructors. They prevent those errors from crashing the entire app. If an error occurs, the Error Boundary can show a fallback UI (like an error message). This improves app stability. As one explanation states, Error Boundaries *"catch and handle errors occurring within their child components … preventing crashes from propagating up the component tree"* [29].

42. **What is WebView and when might you use it?**
A `WebView` is a component that displays web content (HTML, CSS, JS) inside a React Native app. You would use it if you need to embed a web page or use web-based content in your app. For example, displaying a help page or a legacy web form. However, WebViews add overhead (memory, slower performance), so use them sparingly. They are essentially a mini browser in your app, and you should consider security and responsiveness when using one [30].

43. **What is NavigationContainer in React Navigation?**
`NavigationContainer` is a core component in React Navigation (v5+). It manages the app's navigation state and must wrap your top-level navigators (stack, tab, etc.). It acts as a context provider for navigation, ensuring all navigators share the same state and linking configuration. In practice, you wrap your app in `<NavigationContainer>` to enable navigation functionality (deeplinks, state persistence, theme, etc.) throughout the app [31].

44. **What is a VirtualizedList in React Native?**
`VirtualizedList` is the underlying component used by `FlatList` and `SectionList`. It optimizes performance by **only rendering the list items currently visible on screen** (plus some buffer). As the user scrolls, items are recycled and off-screen items are not rendered. This "virtualization" drastically improves performance for long lists by reducing memory and processing overhead [32].

45. **Why is reusing components important in mobile development?**
Reusing components (and code) saves development time and reduces bugs. Instead of duplicating similar UI or logic, you create a single component that can be parameterized. This follows the DRY principle and keeps the codebase maintainable. Reusable components improve consistency across the app and allow easier updates. In general, code reuse *"allows developers to*

focus on new features instead of reinventing the wheel" and reduces the chance of inconsistencies [33].

46. **How do you request and manage permissions (e.g. camera, location) in React Native?**
On iOS and Android, you must both declare required permissions and request them at runtime. In React Native, you can use the `react-native-permissions` library (or platform-specific APIs) to request and check permissions. For Android, you also need to add permissions in `AndroidManifest.xml` (and for iOS in the Info.plist). The process typically involves checking if permission is granted, requesting it if not, and handling the user's response before accessing the feature (e.g. camera). Properly handling permissions ensures your app doesn't crash and follows platform guidelines [34].

47. **What is the Platform module and how do you use it for conditional code or styling?**
The `Platform` module in React Native provides information about the OS. For example, `Platform.OS` returns `"ios"` or `"android"`. You can use this to write platform-specific code or styles. Example:

```
const styles = {
  padding: Platform.OS === 'ios' ? 20 : 10
};
```

Or conditionally render components like `<MapView />` differently per OS. This is useful to account for differences between platforms without duplicating code. As one source notes, it helps you "write platform-specific code" by checking `Platform.OS` [35].

48. **How do you implement smooth animated transitions between screens?**
For custom screen transitions, you usually use the navigation library's built-in options. With React Navigation, you can apply *TransitionPresets* or define a custom `screenInterpolator`/`TransitionSpec` in a stack navigator to specify how one screen transitions to another (e.g. fade, slide). You can also use the `Animated` API for advanced transitions. The idea is to configure the navigator's animation settings so that the old screen animates out and the new screen animates in, creating a smooth effect [36].

49. **What is useLayoutEffect and how is it different from useEffect?**
`useLayoutEffect` is like `useEffect`, but it fires *synchronously* after all DOM (layout) mutations and before the browser has painted. This means any changes you make in `useLayoutEffect` will happen before the user sees the UI, which can avoid flickering. In contrast, `useEffect` runs after the component has rendered on screen. For example, if you need to measure layout or make changes that affect the layout without showing an intermediate state, use `useLayoutEffect`. In summary: "useLayoutEffect runs before paint, allowing layout changes without visual flicker" [37].

50. **How would you implement background tasks or headless JS in a React Native app?**
For tasks that run in the background (even when the app is closed), you can use libraries like `react-native-background-fetch` or native modules like **Headless JS** on Android. These allow scheduling periodic jobs (e.g. fetch new content) or reacting to events (like push notifications). On Android, you might use `JobScheduler` or `HeadlessJS`; on iOS, you can use `BackgroundFetch` or silent push notifications. When implementing background tasks, be

mindful of battery impact and OS limitations. The goal is to offload non-UI work so that the app can update data or perform actions even in the background.

Sources: Answers above are based on React Native documentation and community interview guides [1] [12] [38] [28] [26] [33], among others, to ensure coverage of common real-world interview questions and concepts in React Native.

---

[1] [2] [3] [4] [5] [6] [7] [8] [10] [12] [13] [14] [17] [18] [19] [20] [21] [38] Latest React Native and JavaScript Interview Prep Guide for 2025
https://www.devlogjournal.blog/2025/10/react-native-interview-questions-and.html

[9] [11] [15] [16] [22] [34] [35] [37] Top React Native Interview Questions with Answer | by Anand Gaur | Medium
https://medium.com/@anandgaur2207/top-react-native-interview-questions-with-answer-c5b59ccdd9f0

[23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [36] Top 100 React Native Interview Questions and Answers
https://www.turing.com/interview-questions/react-native

[33] What is code reuse and why is it important?
https://www.opslevel.com/resources/what-is-code-reuse-and-why-is-it-important