Top React Native Interview Questions with Answer

Anand Gaur

Follow

31 min read

·

Jun 27, 2025

90

Press enter or click to view image in full size

Whether you're a beginner or an experienced React Native developer, preparing for interviews can be tricky.

This blog covers **150+ frequently asked React Native interview questions** with detailed and beginner-friendly answers.

1. What is React Native?

React Native is an open-source framework developed by Meta that allows you to build mobile apps using JavaScript and React. It compiles to native code, which means the app will use real native components on both Android and iOS. The biggest advantage is code reusability between platforms.

2. How is React Native different from ReactJS?

ReactJS is used for building web apps, and it renders UI in the browser using HTML and CSS. React Native is used for mobile apps and renders native UI components. Though both use the

React library and JSX syntax, their rendering targets and components are different.

3. What are the core components in React Native?

Some important core components are:

- **View**: Like a `<div>`
- **Text**: Displays text
- **Image**: For images
- **ScrollView**: Scrollable container
- **TextInput**: Input field
- **TouchableOpacity**: For button-like interactions

4. What is JSX?

JSX stands for JavaScript XML. It’s a syntax extension that allows you to write HTML-like code inside JavaScript. In React

Native, it looks like HTML but actually represents native components like `<View>` and `<Text>` instead of `<div>` or `<p>`.

5. What is the use of state in React Native?

State is used to hold data that can change during the app's lifecycle. When state changes, the UI re-renders. For example, you can use state to store input values, toggle elements, or track counters.

6. What is the difference between state and props?

State is internal and managed within a component. Props are external and passed from a parent component to a child. Props are read-only, whereas state can be changed using `setState` or `useState`.

7. What are hooks in React Native?

Hooks are special functions that let you use state and other React features in functional components. Common hooks include `useState`, `useEffect`, `useRef`, and `useContext`.

8. What does useEffect do?

`useEffect` lets you perform side effects in a functional component. You can use it to fetch data, subscribe to events, or interact with timers. It runs after the component renders.

9. How do you navigate between screens in React Native?

React Native uses libraries like `react-navigation` for screen navigation. It provides different navigators like Stack, Tab, and Drawer. You can move between screens using `navigation.navigate('ScreenName')`.

10. What is the difference between ScrollView and FlatList?

`ScrollView` renders all items at once, which is fine for small lists. `FlatList` renders items lazily and efficiently, making it better for performance when dealing with large data sets.

11. What is Flexbox and how is it used in React Native?

Flexbox is a layout system used to align and distribute space among components. It helps in making the UI responsive. Key properties include `flexDirection`, `justifyContent`, and `alignItems`.

12. How do you handle API calls in React Native?

You can use JavaScript's `fetch` function or libraries like Axios to make API calls. Typically, you call APIs inside `useEffect` and store the result in state to update the UI.

13. What is AsyncStorage in React Native?

AsyncStorage is a simple, unencrypted key-value storage system for storing small pieces of data locally, like tokens or user preferences. It’s now replaced by community packages like `@react-native-async-storage/async-storage`.

14. What is TouchableOpacity used for?

`TouchableOpacity` is used to make elements clickable, like buttons. It changes opacity when pressed to give feedback to users. It wraps other components like `Text` or `Image`.

15. What is SafeAreaView?

`SafeAreaView` ensures that your app’s UI doesn’t overlap with device-specific areas like the notch or status bar. It’s mainly used for iPhones to avoid rendering content under the system UI.

16. What is FlatList in React Native?

`FlatList` is a performant component used to render long lists of data efficiently. It loads only the visible items on the screen and handles lazy loading, which improves performance for large datasets.

17. What is the difference between functional and class components?

- **Class Component**: Uses `class`, has lifecycle methods, state.
- **Functional Component**: Uses `function`, supports hooks (`useState`, `useEffect`).

18. How do you handle forms in React Native?

You can use `TextInput` for user input and manage the values using `useState`. For better form handling and validation, libraries like **Formik** and **Yup** are commonly used.

19. What is useRef used for?

`useRef` creates a reference that persists between renders. It's used to access DOM-like elements (e.g., a `TextInput`) or store mutable values that shouldn't trigger a re-render when changed.

20. What is the use of useContext?

`useContext` allows you to access values from a context without prop drilling. It's helpful for sharing global data like themes, user authentication, or language preferences across components.

21. How do you create reusable components in React Native?

To make reusable components, create a separate component file with props. Use props to make the component customizable. For example, a custom button can accept `title`, `onPress`, and `style` props.

22. What is a controlled component in React Native?

A controlled component is a form element whose value is controlled by React state. For example, a `TextInput` where the `value` and `onChangeText` are both tied to a state variable is controlled.

23. How do you handle conditional rendering in React Native?

You can use JavaScript conditional expressions like ternary (`condition ? A : B`) or logical `&&` to render different UI components based on conditions.

24. What are keys in a list and why are they important?

Keys are unique identifiers for list items. React uses them to track changes and efficiently update the UI. In `FlatList`, you can use `keyExtractor` to provide unique keys for each item.

25. What is the purpose of StyleSheet in React Native?

`StyleSheet.create` is used to define styles in a structured and optimized way. It improves performance by reducing object creation on every render.

26. How do you handle screen orientation changes?

You can use the `Dimensions` API or libraries like `react-native-orientation-locker` to detect orientation changes and apply conditional UI changes accordingly.

27. What are lifecycle methods in class components?

Lifecycle methods like

- `componentDidMount`
- `componentDidUpdate`

- `componentWillUnmount`

allow you to perform tasks at specific points in a component's life. For example, API calls can be placed in `componentDidMount`.

28. What is the React Native Navigation library?

React Navigation is a popular library for handling screen navigation. It supports Stack, Tab, Drawer, and more. It’s highly customizable and works across both Android and iOS.

29. How do you pass data between screens in React Native?

You can pass data via the `navigation.navigate` method like this:

```
navigation.navigate('Details', { userId: 1 });
```

And access it on the target screen using `route.params`.

30. What is a Splash Screen and how do you implement it?

A splash screen is the initial screen shown when the app is launched. It’s typically used to show branding or loading status. You can implement it using native code or libraries like `react-native-splash-screen`.

31. What are the different types of navigators in React Navigation?

React Navigation provides several navigators to handle different types of app navigation:

- **Stack Navigator**: Navigates between screens like a stack (push/pop).
- **Tab Navigator**: Displays screens in tabs (like Instagram).
- **Drawer Navigator**: Shows a side menu (drawer) for navigation.

Each can be used standalone or combined for complex apps.

32. What is the difference between TouchableOpacity and Pressable?

`TouchableOpacity` reduces the opacity of the component when pressed, providing visual feedback.

`Pressable` is a newer, more flexible alternative that supports more events like `onPressIn`, `onPressOut`, and `onLongPress`.

33. What is a Modal in React Native?

A Modal is a UI component that appears above the current screen. It's used for dialogs, popups, or alerts. The `Modal` component is built-in and allows properties like `transparent` and `animationType`.

34. How do you create animations in React Native?

You can use the built-in `Animated` API for smooth transitions and effects. For simpler animations, use `LayoutAnimation`. For complex animations, libraries like `Reanimated` or `Lottie` are recommended.

35. What is useMemo and when do you use it?

`useMemo` is a React hook that memoizes a value. It helps avoid expensive calculations on every render by only recalculating when dependencies change. It's useful for performance optimization.

36. What is useCallback and why is it used?

`useCallback` returns a memoized version of a callback function. It helps prevent unnecessary re-renders of child components when a function is passed as a prop.

37. What is a VirtualizedList?

`VirtualizedList` is the base component behind `FlatList` and `SectionList`. It efficiently renders only a small subset of items in a long list, improving performance on large datasets.

38. How do you handle app permissions in React Native?

Use the `react-native-permissions` library to request and check permissions like camera, location, or storage. On Android, also declare required permissions in the `AndroidManifest.xml`.

39. What is the purpose of useLayoutEffect?

`useLayoutEffect` is like `useEffect`, but it runs synchronously after all DOM mutations and before the screen is painted. It’s useful for measuring layout or updating UI before the user sees it.

40. What is the purpose of Platform module in React Native?

The `Platform` module helps you write platform-specific code. You can use `Platform.OS` to check whether the app is running on Android or iOS and change styles or logic accordingly.

41. What is SafeAreaView and why is it important?

`SafeAreaView` ensures that your content is not hidden behind device notches or status bars. It's especially important for iPhones with notches or rounded corners.

42. What is the difference between useEffect and useLayoutEffect?

- `useEffect` runs **after** the component is rendered on the screen.
- `useLayoutEffect` runs **before** the paint, allowing layout changes to happen without visual flicker.

43. How do you debug a React Native app?

You can debug using:

- **Chrome Debugger**
- **React Native Debugger**
- **Flipper (Meta’s official tool)**
- `console.log()` for simple checks

These help track state, network calls, performance, and UI issues.

44. What is Flipper in React Native?

Flipper is a desktop debugging platform by Meta. It supports React Native plugins like network inspector, Redux devtools, layout viewer, and crash logs, making debugging more powerful and visual.

45. How do you secure sensitive data in React Native apps?

For sensitive data like login tokens or user credentials, avoid `AsyncStorage`. Instead, use secure storage libraries like:

- `react-native-keychain`
- `expo-secure-store`

  These provide encrypted storage and are safer for handling private data.

46. How Do You Install and Create a React Native Application?

To create a React Native app, first install the **React Native CLI** or use **Expo CLI** for faster setup.

For React Native CLI:

```
npx react-native init MyApp

cd MyApp

npx react-native run-android

npx react-native run-ios
```

For Expo (easier for beginners):

```
npm install -g expo-cli
```

```
expo init MyApp
```

```
cd MyApp
```

```
expo start
```

Expo runs on Android/iOS simulators or physical devices via the Expo Go app.

47. What Is Redux and When Should You Use It?

Redux is a state management library used for managing global state in complex apps. You use it when state needs to be shared across many components or screens (e.g., user login state, theme, cart items). Redux uses a central **store**, **actions**, and **reducers** to manage state updates in a predictable way.

48. Describe How to Rerender a FlatList.

`FlatList` rerenders automatically when its `data` prop changes. If it doesn’t, you may need to use `extraData` to force an update.

Example:

```
<FlatList data={items} extraData={selectedItem} renderItem={...} />
```

Changing the value of `extraData` triggers a re-render.

49. What Happens When You Call setState()?

Calling `setState()` updates the component's state and triggers a re-render. React batches state updates for performance and may delay the update slightly. `setState` is asynchronous, so the new state isn’t available immediately after calling it.

50. What Are Higher-Order Components (HOC) And How Do You Use Them?

A Higher-Order Component (HOC) is a function that takes a component and returns a new component with added behavior or data. It’s often used for code reuse like authentication, logging, or theming.

Example:

```
const withLogger = (WrappedComponent) => (props) => {
 console.log('Props:', props);
 return <WrappedComponent {...props} />;
};
```

51. How Do You Style a Component in React Native?

Use the `StyleSheet.create()` method to define styles or use inline styles.

```
const styles = StyleSheet.create({
 title: { fontSize: 20, color: 'blue' }
});
```

```
<Text style={styles.title}>Hello</Text>
```

52. How Do You Call a Web API in React Native?

Use `fetch()` or libraries like `axios` inside `useEffect`.

```
useEffect(() => {

 fetch('https://api.example.com/data')

    .then(res => res.json())

    .then(data => setData(data));

}, []);
```

53. Describe How the Virtual DOM Works.

The Virtual DOM is a lightweight copy of the real DOM. React uses it to calculate differences (called **diffing**) between current and previous UI states. Only the changed parts are updated in the real DOM, improving performance.

54. Describe Flexbox Along With Its Most Used Properties.

Flexbox helps in building responsive UIs. Key properties include:

- `flexDirection`: row or column layout
- `justifyContent`: alignment along the main axis
- `alignItems`: alignment along the cross axis
- `flex`: how much a component should grow/shrink

  It simplifies layout for different screen sizes.

55. How Can You Fetch Data From a Local JSON File in React Native?

Import the JSON file directly if it’s part of your project:

```
import data from './data.json';
```

```
console.log(data);
```

Or use `require()` for static assets. No need for fetch calls in this case.

56. List Some Ways You Can Optimize an Application.

- Use `FlatList` instead of `ScrollView` for large lists
- Use `React.memo`, `useMemo`, `useCallback` to avoid re-renders
- Optimize images and assets
- Avoid unnecessary state updates
- Use lazy loading and code-splitting

57. How Do You Create a StackNavigator in React Native?

Install React Navigation stack package:

```
npm install @react-navigation/native @react-navigation/stack
react-native-screens
```

Then use it like this:

```
const Stack = createStackNavigator();

<NavigationContainer>

 <Stack.Navigator>

   <Stack.Screen name="Home" component={HomeScreen} />

   <Stack.Screen name="Profile" component={ProfileScreen} />

 </Stack.Navigator>

</NavigationContainer>
```

58. What Are Some Causes of Memory Leaks and How Can You Detect Them (iOS & Android)?

Common causes:

- Unmounted components still listening (e.g., setInterval, subscriptions)
- Holding references to large objects in global variables
- Not clearing listeners or timers

Detection tools:

- **Android**: Android Profiler, LeakCanary
- **iOS**: Instruments (Allocations, Leaks), Xcode memory graph

Fix by:

- Cleaning up in `useEffect`'s return function or `componentWillUnmount`
- Avoiding global states unless needed

59. What is the difference between React Native and React?

React is a JavaScript library for building web interfaces, primarily for browsers. React Native, on the other hand, allows you to build real mobile applications using React. While both use JSX and the component-based architecture, React Native uses native components (`View`, `Text`) instead of web components (`div`, `span`). Also, React targets the browser DOM, while React Native renders to native platforms.

60. How does hot reloading differ from fast refresh?

Hot Reloading (older method) allowed injecting updated code into a running app without a full reload, but it sometimes missed state updates. Fast Refresh (current method) is more reliable and resets only the file where the change occurred. It maintains the component state whenever possible and gives a better developer experience.

61. How would you store sensitive data on-device?

Use secure storage libraries like `react-native-keychain`, `expo-secure-store`, or `react-native-sensitive-info` to store data like tokens or credentials. These use platform-specific encrypted storage mechanisms (e.g., Android Keystore, iOS Keychain) to keep data safe.

62. How do you handle different screen sizes in React Native?

Use Flexbox for responsive layouts, and the `Dimensions` API to get screen width and height. You can also use

`react-native-responsive-dimensions`,

`react-native-size-matters`, or

`react-native-responsive-fontsize` for better control across devices.

63. How do you handle push notifications?

Use services like Firebase Cloud Messaging (FCM) with `react-native-push-notification` or `@notifee/react-native` for local notifications. On iOS, configure APNs properly. Don't forget to handle permissions and background handling using `@react-native-firebase/messaging` and `react-native-permissions`.

64. How do you handle feature toggling in production?

Use remote config tools like **Firebase Remote Config**, **LaunchDarkly**, or build your own toggle service. Store feature flags on the server or local config, and conditionally render features using those values.

65. How do you integrate React Native with a native SDK that has no JS bindings?

You need to create a native module. For Android, use Java/Kotlin; for iOS, use Swift/Objective-C. Define a bridge between native code and JavaScript using the `NativeModules` API in React Native.

66. How do you implement geo-fencing and background tasks?

Use native modules or third-party libraries like `react-native-background-geolocation`. Background tasks on Android need `WorkManager` or `Headless JS`; on iOS, use `BackgroundFetch` and ensure background modes are enabled.

67. What is your approach to testing native integration boundaries?

Use unit tests for JS logic, and integration or end-to-end testing for native interactions. Tools like **Jest**, **Detox**, or **Appium** help in simulating real device behavior and ensure the bridge between JS and native is working.

68. How do you handle real-time updates in React Native apps?

Use WebSockets (via `Socket.IO`), Firebase Realtime Database, or services like Pusher/Ably. Update app state when a real-time message or event is received to reflect changes in UI instantly.

69. How do you handle complex animations?

Use `react-native-reanimated` or `react-native-animatable` for smoother performance and control. `Animated` API works well for simple animations, but for gesture-based and complex sequences, `Reanimated` is preferred.

70. How do you manage deep linking and universal links?

Use `react-navigation` with `Linking` API. For universal links on iOS and app links on Android, configure the app's native files (like `AndroidManifest.xml` and `Info.plist`) and test using links like `myapp://screen/path`.

71. What are the trade-offs between Expo and bare React Native?

Expo is easier to start with, handles a lot of config, and is great for small to mid apps. However, it may limit access to native modules. **Bare React Native** gives full control over native code, which is better for apps with native SDKs or custom modules.

72. How do you manage state in React Native?

You can manage state using:

- `useState` and `useReducer` (local component state)
- `Context API` (global light-weight state)

- External libraries like Redux, MobX, Zustand, Recoil depending on app complexity and size.

73. What is Rendering Prop Pattern in React Native?

The rendering prop pattern involves passing a function as a prop to a component, which that component uses to determine what to render. It promotes code reuse and flexibility. Example:

```
<MyComponent render={(data) => <Text>{data.name}</Text>} />
```

This lets the parent decide how to render the child component's internal data.

74. How Do We Manage State in a Big & Scalable React Native Application?

For large apps, local state is not enough. Use centralized state managers like Redux, MobX, or Zustand. Modularize slices of state (e.g., user, theme, cart) and use selectors and middleware for better scalability and maintainability.

75. How Do we Use Axios In React Native?

Install it with `npm install axios`. Import and use it like:

```
axios.get('https://api.example.com/data')

 .then(response => console.log(response.data))

 .catch(error => console.error(error));
```

You can also create a custom Axios instance with headers and base URLs.

76. What are Controlled and Uncontrolled Components in React Native?

- **Controlled components** have their state managed by React (via `useState` and `value` props).

- **Uncontrolled components** store their own state internally and use `ref` to access values. Controlled components are preferred for predictable behavior.

77. How do We Create an SQLite database in React Native?

Use `react-native-sqlite-storage` or `react-native-sqlite-2`. Install the library and create/open a DB:

```
SQLite.openDatabase('mydb.db', '1.0', '', 1);
```

Use SQL queries to create tables and store data locally.

78. What is Fabric in React Native?

Fabric is React Native's new rendering engine. It improves performance, UI responsiveness, and concurrency. It enables direct communication between JavaScript and the native layer without relying on the traditional bridge.

79. How Does React Native Handle Different Screen Sizes?

React Native uses Flexbox for layout, and you can use `Dimensions`, `PixelRatio`, or responsive UI libraries. Components like `SafeAreaView`, percentage-based widths, and scalable fonts also help.

80. How to Write Platform-Specific Code in React Native?

Use:

- `Platform.OS` to conditionally apply logic/styling.
- `.ios.js` and `.android.js` file extensions to create separate files.
- Platform-specific imports using `Platform.select()`.

81. What is Watchman in React Native?

Watchman is a file-watching service developed by Meta. It monitors changes in the file system and improves rebuild and reload speeds in development. It’s installed automatically with the React Native CLI.

82. What is Yoga in React Native?

Yoga is a layout engine used in React Native to calculate the position and size of components. It implements Flexbox layout rules and ensures consistent layout across platforms.

83. How do you integrate native Android/iOS modules in React Native?

Write a native module in Java/Kotlin (Android) or Swift/Obj-C (iOS), expose the methods, and register them using the `ReactPackage` interface or module registry. Then use `NativeModules` to access them in JavaScript.

84. How do you handle complex navigation patterns?

Use `react-navigation`'s nested navigators (stack inside drawer/tab). For complex flows, combine navigators and share context/state between them. Deep linking and conditional routes also help control flows.

85. How do you manage over-the-air updates securely?

Use services like CodePush or Expo Updates. To secure them:

- Sign update bundles.
- Verify app versions.
- Use SSL and update over HTTPS only.
- Avoid exposing sensitive logic in JS.

86. What's your strategy for handling app state during background/foreground transitions?

Use the `AppState` API to detect transitions. Save necessary state/data to local storage or state management before going into the background. Restore state on foreground based on `AppState.currentState`.

87. How do you ensure accessibility (a11y) in React Native?

- Use `accessible={true}` on UI elements.
- Add `accessibilityLabel`, `accessibilityHint`, and roles.
- Test with screen readers.
- Avoid absolute positioning that breaks logical flow.

88. What are common React Native performance pitfalls you've seen?

- Unoptimized FlatLists
- Heavy operations on the UI thread
- Re-rendering due to props/state changes
- Large images

- Memory leaks via unmounted timers/listeners

89. What are the biggest trade-offs of React Native in enterprise?

- Native performance can be harder to match
- Some libraries may lag behind platform updates
- Requires team expertise in both native and JS
- Dependency management across native platforms can be challenging

90. What is the role of the bridge in React Native?

The bridge connects JavaScript and native code. It serializes and sends messages asynchronously. While it works well, it can be a bottleneck, which is why the new architecture (Fabric + TurboModules) aims to reduce its use.

91. What is useEffect used for?

`useEffect` is used to run side effects like:

- API calls
- Subscriptions
- Timers

Example:

```
useEffect(() => {

 fetchData();

}, []);
```

92. What is a Stack Navigator?

A stack navigator allows navigation in a stack-based manner

(push/pop screens like a deck of cards).

93. How do you manage state globally?

Use:

- **Context API**
- **Redux**
- **Recoil**
- **MobX**

94. What are props in React Native?

Props are like parameters passed from one component to another to configure or display content dynamically.

94. What testing tools are used in React Native?

- Jest (unit tests)
- Detox (end-to-end tests)
- React Test Renderer

95. How to improve React Native performance?

- Use `FlatList` over `ScrollView`
- Avoid anonymous functions in render

- Use memoization (`React.memo`, `useCallback`)
- Optimize images
- Avoid unnecessary re-renders

96. What is the use of React.memo()?

Prevents unnecessary re-renders by memoizing a component unless props change.

97. What are the types of navigators in React Navigation?

- Stack Navigator
- Bottom Tab Navigator
- Drawer Navigator
- Material Top Tabs

98. Difference between Animated and LayoutAnimation?

- `Animated`: For custom animations

- `LayoutAnimation`: For layout changes (adding/removing items)

99. How do you handle hardware back button in Android?

Use `BackHandler` API.

100. How do you handle environment variables?

Use libraries like `react-native-dotenv` to load variables from `.env`.

101. What is deep linking in React Native?

A way to open the app via a URL (`myapp://profile/2`) and navigate to a specific screen.

102. How do you secure sensitive data?

Use secure storage options:

- `react-native-keychain`

- `SecureStore`
- Avoid storing tokens in AsyncStorage

103. How to create utility types in TypeScript?

```
type PartialUser = Partial<User>;
```

```
type ReadonlyUser = Readonly<User>;
```

Common built-in utility types:

- `Partial<T>`
- `Readonly<T>`
- `Pick<T, K>`
- `Omit<T, K>`
- `Record<K, T>`

104. What is hoisting in JavaScript?

Hoisting is JavaScript's behavior of **moving declarations** (but not initializations) to the top of the current scope.

```
console.log(a); // undefined
```

```
var a = 5;
```

Function declarations are hoisted entirely, but `let` and `const` are **not hoisted in the same way** — they remain in the **Temporal Dead Zone (TDZ)** until declaration.

105. What is the difference between `null` and `undefined`?

- `undefined`: a variable has been declared but not assigned a value.
- `null`: a deliberate assignment representing "no value".

```
let a;         // undefined
let b = null; // null
```

106. How does JavaScript handle `this` keyword?

`this` refers to the object that is executing the current function.

- In a method: refers to the object.
- In a function: `undefined` in strict mode, `window` otherwise.
- In arrow functions: inherits `this` from the parent lexical scope.

```
const obj = {
 name: 'JS',
 log: function() {
   console.log(this.name);
 }
}
```

107. What is a closure and how can it lead to memory leaks in React Native?

Closures allow inner functions to access outer scope variables even after the outer function has returned. In React Native, if closures capture large variables (like DOM nodes, timers, or subscriptions) and are not cleaned up, they can cause memory leaks — especially in components that unmount.

108. How does the JavaScript event loop affect React Native performance?

React Native runs JS code on a single thread. If long-running operations (loops, parsing large JSON, etc.) block the event loop, animations and gestures become unresponsive. Always delegate such tasks to native modules or background threads using libraries like `react-native-reanimated` or `worker_threads`.

109. What's the difference between `null`, `undefined`, and `NaN` in a React Native app context?

- `undefined`: variable declared but not assigned.
- `null`: assigned as empty.
- `NaN`: result of an invalid number operation.

  In React Native, unhandled `undefined` or `null` props can lead to crashes, blank screens, or misrendered components — especially in FlatLists or navigation params.

110. What are arrow functions and how do they handle `this` differently?

Arrow functions do **not bind their own `this`**. In React Native, this is useful when passing callbacks in components — avoids `.bind(this)`:

```
onPress={() => this.handleClick()} // arrow maintains lexical `this`
```

But avoid overuse in `render()` for performance reasons.

111. Explain the concept of `debounce` and `throttle`. How are they useful in React Native?

- **Debounce**: delays a function call until user stops performing an action (e.g., text input search).
- **Throttle**: limits function execution to once every x milliseconds (e.g., scroll event).

In React Native, use them for scroll handlers, search input, or GPS updates to prevent performance hits.

112. What is the use of `Object.freeze()` in React Native state management?

Used to make objects immutable. Helpful when managing Redux or Context state, ensuring components re-render properly and accidental mutations are prevented.

113. How does `setTimeout()` or `setInterval()` behave in React Native when the app is backgrounded?

On iOS, timers are paused or delayed when app goes to background. On Android, behavior can vary depending on battery optimizations. Consider using `react-native-background-timer` or native modules for critical background timers.

114. What are pure functions and how are they related to React.memo or PureComponent?

Pure functions don't cause side effects and return the same output for the same input. React Native uses `React.memo()` and `PureComponent` to prevent unnecessary re-renders if the props haven't changed — improving performance for list items or child components.

115. What are default parameters and rest/spread operators, and how are they used in RN apps?

```
function greet(name = 'Guest') {
 return `Hello, ${name}`;
```

```
}
```

```
const newObj = { ...oldObj }; // spread operator
```

In React Native:

- Spread props into child components.
- Collect unknown props using rest `...rest`.

116. What are Promises, async/await, and how are they chained in React Native?

Use Promises to handle async tasks (e.g., network requests):

```
fetchData()
```

```
 .then(res => res.json())
 .catch(err => console.error(err));
```

```
async function getData() {
 try {
   const res = await fetch(url);
   const json = await res.json();
 } catch (err) {
   // handle error
 }
}
```

Used extensively in API calls, async storage, camera/file handling.

117. How does TypeScript improve development in React Native apps?

- Catch errors at compile time
- IntelliSense & auto-complete
- Better documentation through types
- Safer code when dealing with props, APIs, and state

118. How do you define a component’s props and state using TypeScript in React Native?

```
interface Props {

 title: string;

 onPress: () => void;

}



const MyButton: React.FC<Props> = ({ title, onPress }) => (
 <Button title={title} onPress={onPress} />
```

```
);
```

Helps prevent bugs from passing incorrect props or missing callbacks.

119. What is the difference between `type` and `interface` in TS, and when would you use each?

- Use `interface` for component props and object shapes (can extend easily).
- Use `type` for unions, primitives, and function signatures.

```
type Status = 'pending' | 'completed';
```

```
interface User { name: string; }
```

120. How do you handle optional and readonly props in TypeScript for React Native components?

```
interface Props {

 readonly id: number;

 label?: string;

}
```

`label?` is optional, `id` can’t be changed once passed in.

121. What are utility types like `Partial`, `Pick`, `Omit`, and how are they useful in RN?

```
type User = { id: number; name: string; age: number };

type UserPreview = Pick<User, 'id' | 'name'>;

type PartialUser = Partial<User>;
```

Used when only part of the object is needed, especially in forms or API request bodies.

;

122. How do you use Generics in TypeScript for reusable functions or components?

```
function useList<T>(items: T[]): T[] {

 return items;

}
```

Generics let you reuse logic without losing type safety — useful for form handlers, lists, custom hooks.

123. How would you enforce types for navigation parameters using TS in React Navigation?

```
type RootStackParamList = {

 Home: undefined;

 Details: { id: number };

};
```

```
type Props = StackScreenProps<RootStackParamList, 'Details'>;
```

Ensures you don’t miss or mis-type navigation parameters.

124. What is type narrowing and how does it work in React Native apps?

```
function display(value: string | number) {

 if (typeof value === 'string') {

   console.log(value.toUpperCase());

 }

}
```

Used when handling props or API responses with multiple possible types.

125. How do you type `useState` and `useRef` correctly in TypeScript?

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

```
const inputRef = useRef<TextInput>(null);
```

Prevents issues when accessing ref methods or initializing state.

126. How do you use enums or union types to limit prop values in RN?

```
type ButtonType = 'primary' | 'secondary';
```

```
interface ButtonProps {

 type: ButtonType;

}
```

Useful for styling, component variants, or user roles.

127. How would you architect a React Native app for both mobile and web using a single codebase?

To achieve this:

- Use **React Native Web** to render native components on the web.
- Create a **universal design system** with platform-agnostic components.
- Use conditional imports (`Platform.OS === 'web'`) or `*.native.js`, `*.web.js` file suffixes.
- Manage navigation using `react-navigation` (mobile) and `react-router` (web), or unify via `react-native-url-router`.
- Use tools like **Expo Router** for multi-platform routing with web support.

- Test all shared components on both mobile and web platforms for consistent behavior.

128. Explain React Native’s new architecture and how it improves performance.

React Native’s new architecture introduces:

- **Fabric Renderer**: Uses concurrent React rendering and enables direct native tree rendering without JSON bridge overhead.
- **TurboModules**: Lazy-loads native modules only when needed, reducing memory usage and improving startup time.
- **JSI (JavaScript Interface)**: Replaces the old bridge by allowing synchronous and direct calls between JS and native.
- **Codegen**: Generates native bindings automatically from JS types (via Flow or TypeScript).

This architecture improves performance, memory usage, startup time, and simplifies cross-platform logic.

129. How do you handle thread management and heavy computations in React Native?

React Native runs JS on a single thread. To avoid UI blocking:

- Offload heavy computations to a **native thread** or **worker thread** (using `react-native-workers` or `JSThread` via JSI).
- Use libraries like **reanimated** for animations, which execute on the native UI thread.
- Use **Web Workers (in web)** or delegate to server-side for data-heavy tasks.
- Profile using Flipper's Performance Monitor or Chrome DevTools.

130. How would you implement offline-first functionality in a complex React Native app?

- Use a local database like **Realm**, **WatermelonDB**, or **SQLite** for persistent local storage.
- Sync with server when network is available using **queueing strategies** (e.g., Redux Offline).
- Use libraries like **NetInfo** to detect network changes.
- Maintain a **sync status flag** for records.
- Implement **conflict resolution** strategies on sync (last write wins, server-first, or merge).

131. You're building a modular enterprise app used by multiple teams. How would you manage code sharing and separation?

- Set up a **monorepo** using Nx, Turborepo, or Lerna.
- Create **shared packages**:
- `@company/ui` – shared components
- `@company/utils` – utility functions

- `@company/hooks` – reusable logic
- `@company/api` – services
- Enforce **coding standards and lint rules** across packages.
- Enable **CI/CD pipeline** to test and build only affected packages.

132. How does React Native manage memory and garbage collection, and how do you detect memory leaks?

- Memory is managed by JS (V8 or Hermes) GC and native layer (Java/Obj-C).
- Common leak sources: listeners not removed, large images, unmounted components.

Use tools like:

- **Android Profiler**

- **Xcode Instruments**
- **Flipper Memory Plugin**
- Monitor heap usage and retained object graphs.
- Always clean up:

```
useEffect(() => {

 const sub = something.addListener(...);

 return () => sub.remove(); // cleanup

}, []);
```

133. How would you design an OTA (Over The Air) update mechanism in a secure enterprise React Native app?

- Use **Microsoft CodePush** or **Expo Updates**.
- Ensure **bundle signing** before publishing.
- Verify version compatibility with native code (native APIs must not change without App Store update).

- Maintain rollback support via versioned bundles.
- Avoid storing sensitive logic in JS — keep critical logic native.
- Ensure OTA does not override builds with breaking native dependencies.

134. Explain how the React Native bridge works and the limitations that led to the new architecture.

- The old bridge serializes messages into JSON and passes them asynchronously between JS and native layers.
- It’s **asynchronous**, **batch-based**, and doesn’t support direct synchronous calls.

Limitations:

- **Latency** in high-frequency data (e.g., gestures, animations).
- **Complex debugging** due to decoupling.
- Poor startup performance.

New architecture (JSI + TurboModules + Fabric) allows:

- Direct calls (sync/async)
- Better memory management
- Faster communication and rendering

135. What is the role of Hermes in React Native? When would you disable it?

Hermes is a JS engine optimized for React Native (mainly Android):

- Smaller bundle size
- Faster cold starts

- Better memory usage

Disabling Hermes:

- Might be needed for specific libraries that rely on V8 or aren't Hermes-compatible.
- On iOS, Hermes is optional and can be toggled in `Podfile`.

136. How do you set up CI/CD for a React Native project with support for both Android and iOS?

- Use tools like **GitHub Actions**, **Bitrise**, **Fastlane**, or **EAS Build**.
- Set up:
- Lint → Test → Build → Deploy pipeline.
- Android: Gradle build with signed keystore.
- iOS: Xcode build with provisioning profiles.
- Automate versioning and changelogs.

- Integrate **Detox or Jest tests** into CI before build.
- Use secrets management (e.g., GitHub Secrets, Bitrise Vault).

137. How would you structure a React Native project that supports dynamic feature modules (code splitting)?

React Native doesn't natively support Web-style code splitting, but you can emulate dynamic features by:

- Dividing the app into **feature-based folders or packages**.
- Lazy-load screens/components using `React.lazy` + `Suspense` (experimental in RN).
- For large-scale apps, manage features via **monorepo packages**, and load only what's needed.
- Use **navigation-based lazy loading** with `getComponent()` for dynamic imports:

```
Screen: {

 getComponent: () => require('./FeatureX').default

}
```

138. How would you handle flaky network conditions in a React Native app, especially for offline-first design?

- Use `@react-native-community/netinfo` to detect connectivity status.
- Queue failed API requests (Redux Offline / custom queue).
- Use **optimistic updates** with rollback support.
- Maintain local cache (AsyncStorage, Realm, SQLite) and sync when online.
- Show UI feedback like “You’re offline” banners and sync states.

139. How do you test native modules or components in React Native?

For JavaScript:

- Use **Jest** to test logic with mocks.

For Native:

- Use **Android Instrumentation tests** or **XCTest** for native iOS code.
- Use **Detox** for end-to-end testing, including native interactions.

Example: If you've created a custom camera native module, write a bridge-level test in Jest and behavior tests in Detox.

140. What are TurboModules and how do they differ from traditional Native Modules?

TurboModules are part of the new React Native architecture that:

- Are lazily initialized, reducing startup time.
- Use **JSI** to invoke native methods directly, bypassing the old bridge.
- Support synchronous calls.
  They replace the old bridge-based native modules which were initialized eagerly and only worked asynchronously via JSON.

141. How would you handle platform-specific third-party libraries that only support Android or iOS?

- Use conditional imports with `Platform.OS` or `require()` at runtime.
- Wrap native features with **platform guards**:

```
if (Platform.OS === 'android') {

 import NativeAndroidOnly from './NativeAndroidOnly';

}
```

- Use `.ios.js` and `.android.js` file extensions for platform-specific files.
- For shared logic, abstract it behind a single JS API.

142. How do you enforce code security in a React Native app distributed via OTA updates?

- Minify and obfuscate JS bundles using **Metro configs**.
- Never store or expose secrets in JS — use native Secure Storage or remote fetch.
- Use **Hermes** to compile and reduce reverse engineering risks.
- Secure OTA updates using **signed update bundles** (e.g., Expo Updates or CodePush with signing).

- Implement **runtime integrity checks** on native layers.

143. How would you debug a memory leak in a production React Native app?

- Use **Flipper memory plugin** for Android or **Instruments** for iOS.

Track:

- Unmounted components not cleaned up.
- Long-living subscriptions (e.g., event listeners, timers).
- Large image/data objects held in state.
- Profile heap snapshots and retain cycles.
- Use `useEffect` cleanups and avoid unnecessary refs.

144. How would you architect an app with shared logic across multiple apps (e.g., white-labeled apps)?

Use a **monorepo** with:

- `@shared/ui` → buttons, modals
- `@shared/hooks` → reusable hooks
- `@shared/themes` → styling
- `@shared/api` → service layer
- Each app imports shared modules and overrides branding via config files.
- Create branded theme tokens and use dynamic assets (logos/colors) based on env.

145. How do you optimize the app bundle size in React Native?

- Enable **Hermes** (smaller JS engine size).
- Use `react-native-exclude` to remove unused assets.

- Tree-shake and remove dead code using Babel + Metro config.
- Lazy-load features and use `dynamic import()` where possible.
- Compress assets and avoid large image bundles.
- Use Proguard (Android) and Bitcode (iOS) for native binary optimization.

146. How do you handle UI testing in React Native, especially for accessibility and layout correctness?

- Use **Detox** for E2E testing — simulate gestures, focus, scrolls, etc.
- Use **`accessibilityLabel`**, **`accessibilityHint`**, **`testID`** for targeting.
- For layout consistency:
- Snapshot testing via **Jest** + **react-test-renderer**

- Use **Storybook** for component-driven testing and isolation.
- Manually test with screen readers to validate a11y compliance.

147. Scenario: Your app crashes only in the Android release build, not in debug. What steps do you take?

This typically points to Proguard or missing native configurations. Here’s how I’d debug it:

- **Check `proguard-rules.pro`**: If you're using libraries like Firebase, Realm, or Retrofit, they may need specific rules.
- **Enable crash logging** using `adb logcat` for release builds.
- **Use tools like Firebase Crashlytics or Sentry** to capture release crashes.

- Disable minify temporarily to isolate the issue:

```
minifyEnabled false
```

```
shrinkResources false
```

148. Scenario: You need to support a 5-step onboarding flow with conditional navigation based on user input. How would you structure it?

- Use **`react-navigation`** and create a stack navigator for the onboarding screens.
- Manage conditional flow using **Context or local state**.
- Use conditional logic in `navigation.navigate()` to decide the next screen based on form input.
- If onboarding is needed only once, store a flag like `hasOnboarded` in AsyncStorage or secure storage.

149. Scenario: You notice memory leaks when modals are opened and closed repeatedly. What's your approach?

- Use the **Flipper memory plugin** or **Xcode Instruments/Android Profiler** to track memory.
- Check if any **event listeners or intervals** inside modals aren't cleaned up on close.
- Use `useEffect` cleanup:

```
useEffect(() => {
 const interval = setInterval(...);
 return () => clearInterval(interval); // clean up
}, []);
```

- Avoid unnecessary re-renders by memoizing large components or using `React.memo`.

150. Scenario: You have to store tokens and user data securely. How do you handle it?

- Use `react-native-keychain` or `expo-secure-store` for storing sensitive data like tokens.
- Avoid AsyncStorage for confidential data.
- On logout, always clear secure storage and AsyncStorage.
- Implement **token rotation** with refresh tokens and check expiry before each API call.

151. Scenario: You need real-time updates in a chat app. What's your stack and how do you handle it?

- Use **WebSockets** with libraries like `socket.io-client` or Firebase's real-time DB.
- Connect WebSocket in a `useEffect` inside a global context or message screen.
- Emit and listen to events like `message:new` or `user:typing`.

- Debounce typing indicators and ensure cleanup on component unmount.

152. Scenario: Your feed screen with videos/images drops frames while scrolling. How do you fix it?

- Use `FlatList` with `windowSize`, `removeClippedSubviews`, and `initialNumToRender` props for optimization.
- Avoid inline styles and functions in `renderItem`.
- Use `react-native-fast-image` for optimized image loading.
- Avoid auto-playing multiple videos; only play one using `onViewableItemsChanged`.

153. Scenario: You need to support dark mode and runtime theme switching across 100+ components. What's your approach?

- Use **Context API or Zustand** to manage theme state.

- Define global themes (light/dark/custom) with tokens like `colors.primary`, `text`, etc.
- Use `useContext(ThemeContext)` in components to apply styles dynamically.
- Persist theme preference in AsyncStorage and sync on app start.

154. Scenario: You need to scale your React Native app in a monorepo for multiple teams. How would you set it up?

- Use **Nx**, **Turborepo**, or **Lerna** for monorepo management.
- Organize the codebase into packages:
- `@app/mobile`
- `@shared/ui`
- `@shared/hooks`
- Use TypeScript project references and aliases for import paths.

- Apply CI workflows to lint/test only affected packages.

155. Scenario: Push notifications work on Android but silently fail on iOS. How do you debug this?

- Confirm APNs key/cert setup on Apple Developer Portal.
- Use a test tool like **Pusher** or **Firebase console** to send a notification directly.
- Check iOS-specific settings:
- Capabilities → Push Notification
- Background Modes → Remote notifications
- Log device tokens and errors during registration.
- Ensure notification payload has the `aps` key and correct structure.

156. Scenario: You need to support both deep links and universal links. What are the steps?

- Use `react-navigation`'s linking config and `Linking.addEventListener`.
- On **Android**, configure intent filters in `AndroidManifest.xml`.
- On **iOS**, configure Associated Domains in Xcode and apple-app-site-association file on your server.
- Parse the path and navigate accordingly:

```
const linking = {
  prefixes: ['myapp://', 'https://myapp.com'],
  config: {
    screens: {
      Home: 'home',
      Profile: 'user/:id'
    }

  }

};
```

157. Scenario: You’re asked to migrate an existing app from React Native CLI to Expo. What considerations and challenges do you foresee?

Migrating from React Native CLI to Expo can simplify development but has trade-offs:

- **Check for unsupported native modules**. Expo has limits unless you use the “bare workflow.”
- Evaluate **OTA update strategy** — Expo makes it easier using `expo-updates`.
- Review app.json setup and **migrate config files** (`AndroidManifest.xml`, `Info.plist` → `app.json`).
- Expo uses its own build service (EAS), so CI/CD needs updates.

- Some platform-specific customizations may require ejecting again — a risk to assess.

158. Scenario: Your Android app crashes on certain devices with lower RAM during image-heavy flows. How do you approach it?

- Use **`react-native-fast-image`** for caching and optimized rendering.
- Downscale large images before loading (server-side or locally).
- Monitor memory usage using Android Profiler.
- Use `Image.getSize()` to load optimized dimensions dynamically.
- Use **lazy loading** techniques — avoid rendering all images at once (e.g., in `FlatList`).

159. Scenario: You need to support biometric authentication (FaceID/Fingerprint) in the app. What is your implementation plan?

- Use `react-native-keychain` or `expo-local-authentication`.
- Detect availability using APIs like:

```
LocalAuthentication.hasHardwareAsync()
```

- Use biometric data to unlock stored credentials or secure content.
- Add fallbacks (PIN/password) for unsupported devices.
- Handle biometric permission rejection gracefully.

160. Scenario: A third-party SDK is not working with React Native. It only supports native platforms. What do you do?

- Wrap the native SDK using a **Native Module**.
- For Android: write a `MySDKModule.java` class.
- For iOS: write a `MySDKModule.swift` or Obj-C class.

- Register the module and expose methods to JavaScript via `NativeModules`.
- Write tests for the bridge to ensure accurate behavior.
- If SDKs include UI, expose them via `requireNativeComponent`.

161. Scenario: Your app has an AsyncStorage migration issue after an update. Some users report data loss. How do you handle it?

- Create a **versioning mechanism** for storage. Save `storageVersion` in AsyncStorage.
- On app launch, check version and apply migration logic if outdated.
- Back up important data before overwriting.
- Use `@react-native-async-storage/async-storage` and avoid key collisions.

162. Scenario: You’re implementing deep links with custom domains. Links work on Android but not on iOS. What might be wrong?

Likely issues:

- `apple-app-site-association` file is missing or misconfigured.
- App ID doesn’t match the one in Apple Developer Portal.
- Associated Domains entitlements aren’t added correctly in Xcode.
- Use Apple’s validation tool or Safari debug logs to trace issues.
- Ensure HTTPS is enforced; iOS requires secure universal links.

163. Scenario: You must write tests for a component using camera access. How do you do it?

- Use **mocking** for camera modules (`react-native-camera`, `expo-camera`).
- Write **unit tests** for logic, not the actual camera behavior.
- Use `jest.mock()` to avoid native errors during tests.
- For end-to-end (E2E), use **Detox** or **Appium** with mocked camera access or pre-recorded video feed.

164. Scenario: You need to support RTL (Right-to-Left) languages like Arabic or Hebrew. What changes are needed?

- Use `I18nManager` to enable RTL:

```
I18nManager.forceRTL(true);
```

- Update styles to use `flexDirection: 'row-reverse'` or `start/end` instead of `left/right`.
- Wrap strings with an internationalization library like `i18n-js` or `react-i18next`.
- Ensure `flip` options are respected in custom components.

165. Scenario: You notice your Redux store is bloated with too much temporary UI state. What’s your plan to refactor?

- Move UI-specific state (modals, spinners, form errors) to component-level state (`useState`) or Context.
- Keep Redux for shared/business-critical data (auth, user, cart, theme).
- Split reducers into domains and use selectors to avoid tight coupling.
- Possibly use **Zustand** or **Recoil** for isolated state slices.

166. Scenario: A client wants the app to support code push (OTA updates). What are the challenges and how do you implement it?

- Use **Microsoft CodePush** or **Expo Updates**.
- Ensure no sensitive logic or API keys are exposed in JS bundle.
- Prepare for version conflicts — native changes require App Store/Play Store updates.
- Plan for rollback in case of bad deployment.
- Track app version and update logs to ensure visibility.

Thank you for reading. 🙌🙏✌️.