Accessibility

Both Android and iOS provide APIs for integrating apps with assistive technologies like the bundled screen readers VoiceOver (iOS) and TalkBack (Android). React Native has complementary APIs that let your app accommodate all users.

> **INFO**
>
> Android and iOS differ slightly in their approaches, and thus the React Native implementations may vary by platform.

Accessibility properties

`accessible`

When `true`, indicates that the view is an accessibility element. When a view is an accessibility element, it groups its children into a single selectable component. By default, all touchable elements are accessible.

On Android, `accessible={true}` property for a react-native View will be translated into native `focusable={true}`.

```
<View accessible={true}>
  <Text>text one</Text>
  <Text>text two</Text>
</View>
```

In the above example, we can't get accessibility focus separately on 'text one' and 'text two'. Instead we get focus on a parent view with 'accessible' property.

`accessibilityLabel`

When a view is marked as accessible, it is a good practice to set an accessibilityLabel on the view, so that people who use VoiceOver know what element they have selected. VoiceOver will read this string when a user selects the associated element.

To use, set the `accessibilityLabel` property to a custom string on your View, Text or Touchable:

```
<TouchableOpacity
  accessible={true}
  accessibilityLabel="Tap me!"
  onPress={onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Press me!</Text>
  </View>
</TouchableOpacity>
```

In the above example, the `accessibilityLabel` on the TouchableOpacity element would default to "Press me!". The label is constructed by concatenating all Text node children separated by spaces.

`accessibilityLabelledBy` Android

A reference to another element nativeID used to build complex forms. The value of `accessibilityLabelledBy` should match the `nativeID` of the related element:

```
<View>
  <Text nativeID="formLabel">Label for Input Field</Text>
  <TextInput
    accessibilityLabel="input"
    accessibilityLabelledBy="formLabel"
  />
</View>
```

In the above example, the screenreader announces `Input, Edit Box for Label for Input Field` when focusing on the TextInput.

`accessibilityHint`

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not clear from the accessibility label.

To use, set the `accessibilityHint` property to a custom string on your View, Text or Touchable:

```
<TouchableOpacity
  accessible={true}
  accessibilityLabel="Go back"
  accessibilityHint="Navigates to the previous screen"
  onPress={onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Back</Text>
  </View>
</TouchableOpacity>
```

iOS

In the above example, VoiceOver will read the hint after the label, if the user has hints enabled in the device's VoiceOver settings. Read more about guidelines for `accessibilityHint` in the iOS Developer Docs

Android

In the above example, TalkBack will read the hint after the label. At this time, hints cannot be turned off on Android.

`accessibilityLanguage` iOS

By using the `accessibilityLanguage` property, the screen reader will understand which language to use while reading the element's **label**, **value** and **hint**. The provided string value must follow the BCP 47 specification.

```
<View
  accessible={true}
  accessibilityLabel="Pizza"
  accessibilityLanguage="it-IT">
```

```
    <Text>🍕</Text>
  </View>
```

`accessibilityIgnoresInvertColors` iOS

Inverting screen colors is an Accessibility feature that makes the iPhone and iPad easier on the eyes for some people with a sensitivity to brightness, easier to distinguish for some people with color blindness, and easier to make out for some people with low vision. However, sometimes you have views such as photos that you don't want to be inverted. In this case, you can set this property to be `true` so that these specific views won't have their colors inverted.

`accessibilityLiveRegion` Android

When components dynamically change, we want TalkBack to alert the end user. This is made possible by the `accessibilityLiveRegion` property. It can be set to `none`, `polite` and `assertive`:

- **none** Accessibility services should not announce changes to this view.
- **polite** Accessibility services should announce changes to this view.
- **assertive** Accessibility services should interrupt ongoing speech to immediately announce changes to this view.

```
<TouchableWithoutFeedback onPress={addOne}>
  <View style={styles.embedded}>
    <Text>Click me</Text>
  </View>
</TouchableWithoutFeedback>
<Text accessibilityLiveRegion="polite">
  Clicked {count} times
</Text>
```

In the above example method `addOne` changes the state variable `count`. As soon as an end user clicks the TouchableWithoutFeedback, TalkBack reads text in the Text view because of its `accessibilityLiveRegion="polite"` property.

`accessibilityRole`

`accessibilityRole` communicates the purpose of a component to the user of an assistive technology.

`accessibilityRole` can be one of the following:

- **adjustable** Used when an element can be "adjusted" (e.g. a slider).
- **alert** Used when an element contains important text to be presented to the user.
- **button** Used when the element should be treated as a button.
- **checkbox** Used when an element represents a checkbox which can be checked, unchecked, or have mixed checked state.
- **combobox** Used when an element represents a combo box, which allows the user to select among several choices.
- **header** Used when an element acts as a header for a content section (e.g. the title of a navigation bar).
- **image** Used when the element should be treated as an image. Can be combined with button or link, for example.
- **imagebutton** Used when the element should be treated as a button and is also an image.
- **keyboardkey** Used when the element acts as a keyboard key.
- **link** Used when the element should be treated as a link.
- **menu** Used when the component is a menu of choices.
- **menubar** Used when a component is a container of multiple menus.
- **menuitem** Used to represent an item within a menu.
- **none** Used when the element has no role.
- **progressbar** Used to represent a component which indicates progress of a task.
- **radio** Used to represent a radio button.
- **radiogroup** Used to represent a group of radio buttons.
- **scrollbar** Used to represent a scroll bar.
- **search** Used when the text field element should also be treated as a search field.
- **spinbutton** Used to represent a button which opens a list of choices.

- **summary** Used when an element can be used to provide a quick summary of current conditions in the app when the app first launches.
- **switch** Used to represent a switch which can be turned on and off.
- **tab** Used to represent a tab.
- **tablist** Used to represent a list of tabs.
- **text** Used when the element should be treated as static text that cannot change.
- **timer** Used to represent a timer.
- **togglebutton** Used to represent a toggle button. Should be used with accessibilityState checked to indicate if the button is toggled on or off.
- **toolbar** Used to represent a tool bar (a container of action buttons or components).
- **grid** Used with ScrollView, VirtualizedList, FlatList, or SectionList to represent a grid. Adds the in/out of grid announcements to the android GridView.

`accessibilityState`

Describes the current state of a component to the user of an assistive technology.

`accessibilityState` is an object. It contains the following fields:

NAME	DESCRIPTION	TYPE	REQUIRED
disabled	Indicates whether the element is disabled or not.	boolean	No
selected	Indicates whether a selectable element is currently selected or not.	boolean	No
checked	Indicates the state of a checkable element. This field can either take a boolean or the "mixed" string to represent mixed checkboxes.	boolean or 'mixed'	No
busy	Indicates whether an element is currently busy or not.	boolean	No
expanded	Indicates whether an expandable element is currently expanded or collapsed.	boolean	No

To use, set the `accessibilityState` to an object with a specific definition.

`accessibilityValue`

Represents the current value of a component. It can be a textual description of a component's value, or for range-based components, such as sliders and progress bars, it contains range information (minimum, current, and maximum).

`accessibilityValue` is an object. It contains the following fields:

NAME	DESCRIPTION	TYPE	REQUIRED
min	The minimum value of this component's range.	integer	Required if `now` is set.
max	The maximum value of this component's range.	integer	Required if `now` is set.
now	The current value of this component's range.	integer	No
text	A textual description of this component's value. Will override `min`, `now`, and `max` if set.	string	No

`accessibilityViewIsModal` iOS

A Boolean value indicating whether VoiceOver should ignore the elements within views that are siblings of the receiver.

For example, in a window that contains sibling views `A` and `B`, setting `accessibilityViewIsModal` to `true` on view `B` causes VoiceOver to ignore the elements in the view `A`. On the other hand, if view `B` contains a child view `C` and you set `accessibilityViewIsModal` to `true` on view `C`, VoiceOver does not ignore the elements in view `A`.

`accessibilityElementsHidden` iOS

A Boolean value indicating whether the accessibility elements contained within this accessibility element are hidden.

For example, in a window that contains sibling views `A` and `B`, setting `accessibilityElementsHidden` to `true` on view `B` causes VoiceOver to ignore the elements

in the view `B`. This is similar to the Android property `importantForAccessibility="no-hide-descendants"`.

`aria-valuemax`

Represents the maximum value for range-based components, such as sliders and progress bars.

`aria-valuemin`

Represents the minimum value for range-based components, such as sliders and progress bars.

`aria-valuenow`

Represents the current value for range-based components, such as sliders and progress bars.

`aria-valuetext`

Represents the textual description of the component.

`aria-busy`

Indicates an element is being modified and that assistive technologies may want to wait until the changes are complete before informing the user about the update.

TYPE	DEFAULT
boolean	false

`aria-checked`

Indicates the state of a checkable element. This field can either take a boolean or the "mixed" string to represent mixed checkboxes.

TYPE	DEFAULT
boolean, 'mixed'	false

aria-disabled

Indicates that the element is perceivable but disabled, so it is not editable or otherwise operable.

TYPE	DEFAULT
boolean	false

aria-expanded

Indicates whether an expandable element is currently expanded or collapsed.

TYPE	DEFAULT
boolean	false

aria-hidden

Indicates whether the accessibility elements contained within this accessibility element are hidden.

For example, in a window that contains sibling views `A` and `B`, setting `aria-hidden` to `true` on view `B` causes VoiceOver to ignore the elements in the view `B`.

TYPE	DEFAULT
boolean	false

aria-label

Defines a string value that labels an interactive element.

TYPE
string

aria-labelledby Android

Identifies the element that labels the element it is applied to. The value of `aria-labelledby` should match the `nativeID` of the related element:

```
<View>
  <Text nativeID="formLabel">Label for Input Field</Text>
  <TextInput aria-label="input" aria-labelledby="formLabel" />
</View>
```

TYPE
string

aria-live Android

Indicates that an element will be updated, and describes the types of updates the user agents, assistive technologies, and user can expect from the live region.

- **off** Accessibility services should not announce changes to this view.
- **polite** Accessibility services should announce changes to this view.
- **assertive** Accessibility services should interrupt ongoing speech to immediately announce changes to this view.

TYPE	DEFAULT
enum('assertive', 'off', 'polite')	'off'

aria-modal iOS

Boolean value indicating whether VoiceOver should ignore the elements within views that are siblings of the receiver.

TYPE	DEFAULT
boolean	false

`aria-selected`

Indicates whether a selectable element is currently selected or not.

TYPE
boolean

`importantForAccessibility` Android

In the case of two overlapping UI components with the same parent, default accessibility focus can have unpredictable behavior. The `importantForAccessibility` property will resolve this by controlling if a view fires accessibility events and if it is reported to accessibility services. It can be set to `auto`, `yes`, `no` and `no-hide-descendants` (the last value will force accessibility services to ignore the component and all of its children).

```
<View style={styles.container}>
  <View
    style={[styles.layout, {backgroundColor: 'green'}]}
    importantForAccessibility="yes">
    <Text>First layout</Text>
  </View>
  <View
    style={[styles.layout, {backgroundColor: 'yellow'}]}
    importantForAccessibility="no-hide-descendants">
    <Text>Second layout</Text>
  </View>
</View>
```

In the above example, the `yellow` layout and its descendants are completely invisible to TalkBack and all other accessibility services. So we can use overlapping views with the

same parent without confusing TalkBack.

`onAccessibilityEscape` iOS

Assign this property to a custom function which will be called when someone performs the "escape" gesture, which is a two finger Z shaped gesture. An escape function should move back hierarchically in the user interface. This can mean moving up or back in a navigation hierarchy or dismissing a modal user interface. If the selected element does not have an `onAccessibilityEscape` function, the system will attempt to traverse up the view hierarchy until it finds a view that does or bonk to indicate it was unable to find one.

`onAccessibilityTap`

Use this property to assign a custom function to be called when someone activates an accessible element by double tapping on it while it's selected.

`onMagicTap` iOS

Assign this property to a custom function which will be called when someone performs the "magic tap" gesture, which is a double-tap with two fingers. A magic tap function should perform the most relevant action a user could take on a component. In the Phone app on iPhone, a magic tap answers a phone call, or ends the current one. If the selected element does not have an `onMagicTap` function, the system will traverse up the view hierarchy until it finds a view that does.

`role`

`role` communicates the purpose of a component to the user of an assistive technology. Has precedence over the `accessibilityRole` prop.

`role` can be one of the following:

- **alert** Used when an element contains important text to be presented to the user.
- **button** Used when the element should be treated as a button.

- **checkbox** Used when an element represents a checkbox which can be checked, unchecked, or have mixed checked state.
- **combobox** Used when an element represents a combo box, which allows the user to select among several choices.
- **grid** Used with ScrollView, VirtualizedList, FlatList, or SectionList to represent a grid. Adds the in/out of grid announcements to the android GridView.
- **heading** Used when an element acts as a header for a content section (e.g. the title of a navigation bar).
- **img** Used when the element should be treated as an image. Can be combined with button or link, for example.
- **link** Used when the element should be treated as a link.
- **list** Used to identify a list of items.
- **menu** Used when the component is a menu of choices.
- **menubar** Used when a component is a container of multiple menus.
- **menuitem** Used to represent an item within a menu.
- **none** Used when the element has no role.
- **presentation** Used when the element has no role.
- **progressbar** Used to represent a component which indicates progress of a task.
- **radio** Used to represent a radio button.
- **radiogroup** Used to represent a group of radio buttons.
- **scrollbar** Used to represent a scroll bar.
- **searchbox** Used when the text field element should also be treated as a search field.
- **slider** Used when an element can be "adjusted" (e.g. a slider).
- **spinbutton** Used to represent a button which opens a list of choices.
- **summary** Used when an element can be used to provide a quick summary of current conditions in the app when the app first launches.
- **switch** Used to represent a switch which can be turned on and off.
- **tab** Used to represent a tab.
- **tablist** Used to represent a list of tabs.
- **timer** Used to represent a timer.
- **toolbar** Used to represent a tool bar (a container of action buttons or components).

Accessibility Actions

Accessibility actions allow an assistive technology to programmatically invoke the actions of a component. In order to support accessibility actions, a component must do two things:

- Define the list of actions it supports via the `accessibilityActions` property.
- Implement an `onAccessibilityAction` function to handle action requests.

The `accessibilityActions` property should contain a list of action objects. Each action object should contain the following fields:

NAME	TYPE	REQUIRED
name	string	Yes
label	string	No

Actions either represent standard actions, such as clicking a button or adjusting a slider, or custom actions specific to a given component such as deleting an email message. The `name` field is required for both standard and custom actions, but `label` is optional for standard actions.

When adding support for standard actions, `name` must be one of the following:

- `'magicTap'` - iOS only - While VoiceOver focus is on or inside the component, the user double tapped with two fingers.
- `'escape'` - iOS only - While VoiceOver focus is on or inside the component, the user performed a two finger scrub gesture (left, right, left).
- `'activate'` - Activate the component. Typically this should perform the same action as when the user touches or clicks the component when not using an assistive technology. This is generated when a screen reader user double taps the component.
- `'increment'` - Increment an adjustable component. On iOS, VoiceOver generates this action when the component has a role of `'adjustable'` and the user places focus on it and swipes upward. On Android, TalkBack generates this action when the user places accessibility focus on the component and presses the volume up button.

- `'decrement'` - Decrement an adjustable component. On iOS, VoiceOver generates this action when the component has a role of `'adjustable'` and the user places focus on it and swipes downward. On Android, TalkBack generates this action when the user places accessibility focus on the component and presses the volume down button.
- `'longpress'` - Android only - This action is generated when the user places accessibility focus on the component and double tap and holds one finger on the screen. Typically, this should perform the same action as when the user holds down one finger on the component while not using an assistive technology.

The `label` field is optional for standard actions, and is often unused by assistive technologies. For custom actions, it is a localized string containing a description of the action to be presented to the user.

To handle action requests, a component must implement an `onAccessibilityAction` function. The only argument to this function is an event containing the name of the action to perform. The below example from RNTester shows how to create a component which defines and handles several custom actions.

```
<View
  accessible={true}
  accessibilityActions={[
    {name: 'cut', label: 'cut'},
    {name: 'copy', label: 'copy'},
    {name: 'paste', label: 'paste'},
  ]}
  onAccessibilityAction={event => {
    switch (event.nativeEvent.actionName) {
      case 'cut':
        Alert.alert('Alert', 'cut action success');
        break;
      case 'copy':
        Alert.alert('Alert', 'copy action success');
        break;
      case 'paste':
        Alert.alert('Alert', 'paste action success');
        break;
    }
```

```
  }}
/>
```

Checking if a Screen Reader is Enabled

The `AccessibilityInfo` API allows you to determine whether or not a screen reader is currently active. See the AccessibilityInfo documentation for details.

Sending Accessibility Events Android

Sometimes it is useful to trigger an accessibility event on a UI component (i.e. when a custom view appears on a screen or set accessibility focus to a view). Native UIManager module exposes a method 'sendAccessibilityEvent' for this purpose. It takes two arguments: view tag and a type of an event. The supported event types are `typeWindowStateChanged`, `typeViewFocused` and `typeViewClicked`.

```
import {Platform, UIManager, findNodeHandle} from 'react-native';

if (Platform.OS === 'android') {
  UIManager.sendAccessibilityEvent(
    findNodeHandle(this),
    UIManager.AccessibilityEventTypes.typeViewFocused,
  );
}
```

Testing TalkBack Support Android

To enable TalkBack, go to the Settings app on your Android device or emulator. Tap Accessibility, then TalkBack. Toggle the "Use service" switch to enable or disable it.

Android emulators don't have TalkBack installed by default. You can install TalkBack on your emulator via the Google Play Store. Make sure to choose an emulator with the Google Play store installed. These are available in Android Studio.

You can use the volume key shortcut to toggle TalkBack. To turn on the volume key shortcut, go to the Settings app, then Accessibility. At the top, turn on Volume key shortcut.

To use the volume key shortcut, press both volume keys for 3 seconds to start an accessibility tool.

Additionally, if you prefer, you can toggle TalkBack via command line with:

```
# disable
adb shell settings put secure enabled_accessibility_services
com.android.talkback/com.google.android.marvin.talkback.TalkBackService

# enable
adb shell settings put secure enabled_accessibility_services
com.google.android.marvin.talkback/com.google.android.marvin.talkback.TalkBackService
```

Testing VoiceOver Support iOS

To enable VoiceOver, go to the Settings app on your iOS device (it's not available for simulator). Tap General, then Accessibility. There you will find many tools that people use to make their devices more usable, such as bolder text, increased contrast, and VoiceOver.

To enable VoiceOver, tap on VoiceOver under "Vision" and toggle the switch that appears at the top.

At the very bottom of the Accessibility settings, there is an "Accessibility Shortcut". You can use this to toggle VoiceOver by triple clicking the Home button.

Additional Resources

- Making React Native Apps Accessible

Is this page useful? 

Edit this page

*Last updated on **Jun 21, 2023***