8:16

Accessibility Components

Interactive examples of accessible React Native components with code samples and best practices

Buttons & Touchables

Essential

Create accessible touch targets with proper sizing and feedback

Touch target sizing Haptic feedback

Form Controls

Complex

Implement accessible form inputs and controls

Error states Helper text

Media Content

Advanced

Make images and media content accessible

Aa Alt text Media controls

Modal Dialogs

Advanced

Screen Reader Support

Comprehensive guide for optimizing your app for VoiceOver and TalkBack

VoiceOver (iOS)

TalkBack (Android)

Implementation Guide

Semantic Structure

- Use proper heading hierarchy
- Implement meaningful landmarks
- Group related elements logically

View Code Examples →

Content Descriptions

- Provide clear accessibilityLabels
- Include meaningful hints
- Describe state changes

View Guidelines →

Interactive Elements

- Define proper roles
- Manage focus appropriately
- Handle custom actions

Screen Reader Support

Comprehensive guide for optimizing your app for VoiceOver and TalkBack

VoiceOver (iOS) | TalkBack (Android)

Essential Gestures

Single tap

Select an item

Double tap

Activate selected item

Three finger swipe up/down

Scroll content

Three finger tap

Speak current page

Two finger swipe up

Read from current position

Semantic Structure

- Use proper heading hierarchy
- Implement meaningful landmarks
- Group related elements logically

View Code Examples →

Aa Content Descriptions

- Provide clear accessibilityLabels
- Include meaningful hints
- Describe state changes

View Guidelines →

Interactive Elements

- Define proper roles
- Manage focus appropriately
- Handle custom actions

View Examples →

Testing Checklist

- ✓ Verify all elements have proper labels
- ✓ Test navigation flow with screen reader
- ✓ Confirm state changes are announced
- ✓ Validate custom actions work correctly

Semantic Structure

Building meaningful and well-organized content hierarchies

Content Hierarchy

Proper headings and landmarks help users quickly parse content. Avoid styling text as a heading without providing a semantic role. In React Native, use **accessibilityRole="heading"** for key titles.

```
// Example of multiple heading levels
            <View
accessibilityRole="header">
              <Text
accessibilityRole="heading" /* Level 1
equivalent */>
                Main Title (H1)
              </Text>
            </View>

            <View
accessibilityRole="main">
              <Text
accessibilityRole="heading" /* Level 2
equivalent */>
                Section Title (H2)
              </Text>
              <Text>
                Some descriptive
content here...
              </Text>
            </View>
```

Navigation & Skip Links

Logical tab order matching the visual layout improves navigation. Provide a skip link to let users jump past repetitive content.

```
// Example: "Skip to Main Content"
button
<TouchableOpacity
  onPress={() => {
    // Focus the main content or move
screen reader focus
  }}
  accessibilityRole="button"
  accessibilityLabel="Skip to Main
Content"
>
  <Text>Skip Navigation</Text>
</TouchableOpacity>

// Then your main content container
<View accessibilityRole="main">
  ...
</View>
```

Landmarks & ARIA Roles

Define distinct areas (e.g., navigation, complementary, contentinfo) to aid comprehension. In React Native, you can mimic these with **accessibilityRole** or custom logic.

- Use **accessibilityRole="navigation"** for top-level nav
- Provide **accessibilityRole="complementary"** for

```
  }}
  accessibilityRole="button"
  accessibilityLabel="Skip to Main
Content"
>
  <Text>Skip Navigation</Text>
</TouchableOpacity>

// Then your main content container
<View accessibilityRole="main">
  ...
</View>
```

Landmarks & ARIA Roles

Define distinct areas (e.g., navigation, complementary, contentinfo) to aid comprehension. In React Native, you can mimic these with **accessibilityRole** or custom logic.

- Use **accessibilityRole="navigation"** for top-level nav
- Provide **accessibilityRole="complementary"** for sidebars
- Mark footers with **accessibilityRole="contentinfo"**

Resources & Next Steps

Learn more about headings, landmarks, and ARIA roles:

- W3C WAI: https://www.w3.org/WAI/
- ARIA Roles: https://www.w3.org/TR/wai-aria-1.2/

VISUAL SETTINGS

Dark Mode
Enable dark theme for low light conditions

High Contrast Mode
Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text
Increase text size for improved readability

Reduce Motion
Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter
Apply basic grayscale filtering

Large Touch Targets

Enable dark theme for low light conditions

High Contrast Mode

Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text

Increase text size for improved readability

Reduce Motion

Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter

Apply basic grayscale filtering

Large Touch Targets

Increase the tappable area of interactive elements

8:27

VISUAL SETTINGS

Dark Mode
Enable dark theme for low light conditions

High Contrast Mode
Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text
Increase text size for improved readability

Reduce Motion
Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter
Apply basic grayscale filtering

Large Touch Targets

8:28

VISUAL SETTINGS

Dark Mode
Enable dark theme for low light conditions

High Contrast Mode
Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text
Increase text size for improved readability

Reduce Motion
Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter
Apply basic grayscale filtering

Large Touch Targets

VISUAL SETTINGS

Dark Mode
Enable dark theme for low light conditions

High Contrast Mode
Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text
Increase text size for improved readability

Reduce Motion
Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter
Apply basic grayscale filtering

VISUAL SETTINGS

Dark Mode
Enable dark theme for low light conditions

High Contrast Mode
Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text
Increase text size for improved readability

Reduce Motion
Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter
Apply basic grayscale filtering

Enable dark theme for low light conditions

High Contrast Mode

Increase contrast for improved readability

READABILITY ENHANCEMENTS

Large Text

Increase text size for improved readability

Reduce Motion

Minimize animations for improved comfort

COLOR & TOUCH SETTINGS

Color Filter

Apply basic grayscale filtering

Large Touch Targets

Large Touch Targets enabled

Testing Tools

Essential tools for testing accessibility in your mobile applications

Screen Readers

TalkBack (Android) Built-in

Android's built-in screen reader. Essential gestures:

- Single tap: Select item
- Double tap: Activate selected item
- Swipe right/left: Next/previous item

VoiceOver (iOS) Built-in

iOS's integrated screen reader. Key gestures:

- Single tap: Select and speak
- Double tap: Activate item
- Three finger swipe: Scroll

Development Tools

Accessibility Inspector

Built-in tool to inspect accessibility properties:

Accessibility Inspector

Built-in tool to inspect accessibility properties:

- Verify accessibility labels and hints
- Check navigation order
- Test screen reader announcements

Contrast Analyzer

Verify color contrast ratios for WCAG guidelines:

- Check text contrast ratios
- Verify UI component contrast
- Support for WCAG 2.2 standards

Testing Checklist

Automated Testing

Essential checks for accessibility testing:

- Run accessibility linter
- Verify accessibility props
- Check navigation order
- Test color contrast

Demonstrate skip links and consistent navigation order.

Skip to Main Content

Why Focus Order Matters

Proper focus order helps screen reader and keyboard users navigate without confusion. A skip link allows bypassing repetitive blocks

Button Pressed

You pressed the first button!

OK

Main Content

Below are interactive elements in a logical sequence.

Focusable Button 1

aaaa

Submit Feedback

Form Controls - Interactive Example

Build accessible, validated forms with proper labels, roles, hints, and date/time pickers.

Form Demo

Name

Email

Gender

Male Female

Preferred Contact Time

Morning Afternoon Evening

Birth Date

Tap to select date

Appointment Time (Optional)

Tap to select time

Email

aaa@a.com

Gender

Male Female

Preferred Contact Time

Morning Afternoon Evening

Success!

Your form has been submitted successfully.

Communication Preferences

Email

Phone

SMS

Agree to terms and conditions

Submit

Code Implementation

JSX ✓ Copied!

```
<View accessibilityRole="form">
  {/* Input Field */}
  <Text style={styles.label}>Name</
Text>
  <TextInput
    value={formData.name}
    accessibilityLabel="Enter your
name"
    accessibilityHint="Type your
full name"
    style={styles.input}
  />

  {/* Radio Group */}
  <View
accessibilityRole="radiogroup">
    {['Option 1', 'Option
2'].map((option) => (
      <TouchableOpacity
        accessibilityRole="radio"

accessibilityState={{ checked:
selectedOption === option }}
        accessibilityLabel={`Select
${option}`}
      >
        <View
style={styles.radioButton} />
        <Text>{option}</Text>
      </TouchableOpacity>
```

```
    accessibilityLabel="Submit Form"
    accessibilityState={{ disabled: !
isValid }}
    style={styles.submitButton}
  >
    <Text>Submit</Text>
  </TouchableOpacity>
</View>
```

Accessibility Features

Input Labels

Clear, descriptive labels that properly associate with form controls

Semantic Roles

Proper role assignments for form controls (radio, checkbox, button)

Error States

Clear error messages and validation feedback for screen readers

Touch Targets

Adequate sizing for interactive elements (minimum 44x44 points)

State Management

Proper announcements for selection controls and submit button

Date/Time Pickers

Integration with native pickers, with announced changes for screen readers

Framework Comparison

Compare key features and capabilities of popular mobile development frameworks

React Native | Flutter | Ionic

Overview | Accessibility | Perfor

React Native

by Meta (Facebook)

Version 0.73

A framework for building native applications using React

Language
JavaScript/ TypeScript

Learning Curve
Moderate

Hot Reload
Yes

Framework Comparison

Compare key features and capabilities of popular mobile development frameworks

React Native | Flutter | Ionic

Overview | Accessibility | Perfo

Screen Reader Support

Full VoiceOver support with native bridge

Complete TalkBack integration

4.5

Semantic Support

Extensive semantic property support

- accessibilityLabel
- accessibilityHint
- accessibilityRole
- accessibilityState
- accessibilityValue

4.5

Framework Comparison

Compare key features and capabilities of popular mobile development frameworks

React Native | Flutter | Ionic

Overview | Accessibility | Perfo

Startup Time
1.2s

Memory Usage
Medium

Bundle Size
7-12MB

Overall Performance Rating

4.0

Gestures Tutorial

Practice tap gestures: single tap, double tap, and long press.

Single Tap

Tap me!

For screen readers, double tap activates the item.

Double Tap

Double Tap me!

Tap twice quickly (if using a screen reader, double tap will activate).

Long Press

Long Press me!

Press and hold the button. Note: In screen readers, long press might not be available. Instead, select the item and simulate the press with double tapping.

Gestures Tutorial

Practice tap gestures: single tap, double tap, and long press.

Single Tap

Tap me!

For screen readers, double tap activates the item.

Double Tap

Double Tap me!

Double tap successful!

Tap twice quickly (if using a screen reader, double tap will activate).

Long Press

Long Press me!

Press and hold the button. Note: In screen readers, long press might not be available. Instead, select the item and simulate the press with double tapping.

WCAG 2.2 Guidelines

Essential principles for building accessible mobile apps

Perceivable

Information and user interface components must be presentable to users in ways they can perceive.

- Provide text alternatives for non-text content
- Provide captions and other alternatives for multimedia
- Create content that can be presented in different ways without losing meaning
- Make it easier for users to see and hear content

Operable

User interface components and navigation must be operable.

- Make all functionality available from a keyboard

- Make all functionality available from a keyboard
- Give users enough time to read and use content
- Do not use content that causes seizures or physical reactions
- Help users navigate and find content

Understandable

Information and the operation of user interface must be understandable.

- Make text readable and understandable
- Make content appear and operate in predictable ways
- Help users avoid and correct mistakes

Robust

Content must be robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies.

- Maximize compatibility with current and future user tools

8:15

The ultimate accessibility-driven toolkit for developers

A comprehensive resource for building inclusive React Native applications with verified accessibility standards – explore for more!

18	76%	88%
Components	**WCAG 2.2**	**Screen Reader**
Ready to Use	Level AA	Test Coverage

Quick Start

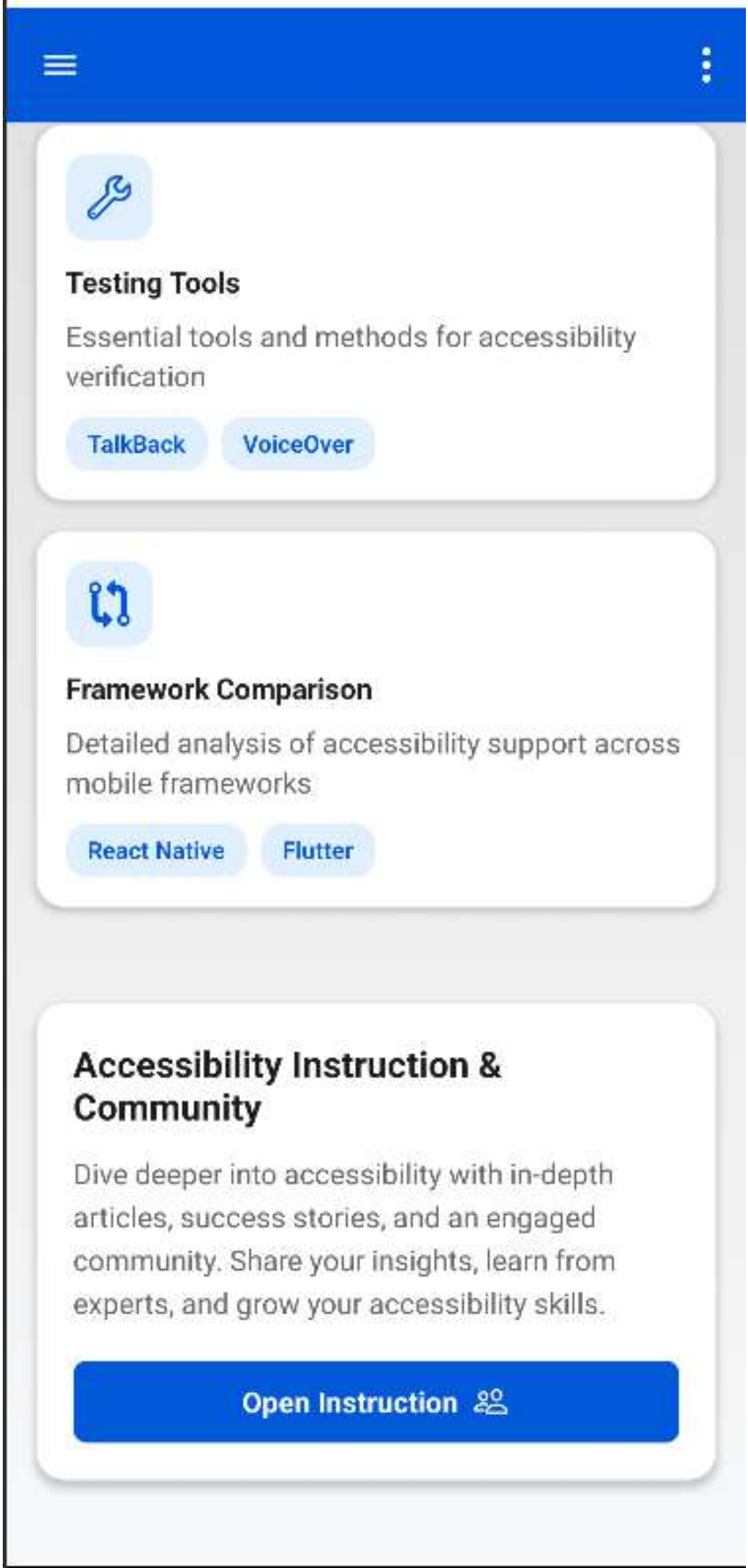
Explore accessible component examples

Development Resources

Best Practices

Comprehensive WCAG 2.2 implementation guidelines for React Native

WCAG 2.2 Guidelines


Testing Tools
Essential tools and methods for accessibility verification
TalkBack
VoiceOver
Framework Comparison
Detailed analysis of accessibility support across mobile frameworks
React Native
Flutter
Accessibility Instruction & Community
Dive deeper into accessibility with in-depth articles, success stories, and an engaged community. Share your insights, learn from experts, and grow your accessibility skills.
Open Instruction

Media Content - Interactive Example

View images with detailed alternative text and roles. Use the controls below to navigate.

Media Demo

Show Alt Text

Code Implementation

JSX Copy

```jsx
<Image
  source={require('./path/to/
image.png')}
  accessibilityLabel="Detailed
description of the image content"
```

Media Content - Interactive Example

View images with detailed alternative text and roles. Use the controls below to navigate.

Media Demo

Hide Alt Text

Alternative Text:

A placeholder image (first example)

Role: Interface example

Code Implementation

JSX Copy

Code Implementation

JSX Copy

```jsx
<Image
  source={require('./path/to/
image.png')}
  accessibilityLabel="Detailed
description of the image content"
  accessible={true}
  accessibilityRole="image"
  style={{
    width: 300,
    height: 200,
    borderRadius: 8,
  }}
/>
```

Accessibility Features

Alternative Text

Descriptive text that conveys the content and function of the image

Role Announcement

Screen readers announce the element as an image

Touch Target

Interactive images should have adequate touch targets

Implement accessible form inputs and controls

Error states Helper text

Media Content

Advanced

Make images and media content accessible

Aa Alt text Media controls

Modal Dialogs

Advanced

Implement accessible modal dialogs with proper focus management and screen reader support

Focus trapping Screen reader alerts

Loading & Navigation

Beta

Explore advanced patterns such as Tabs/ Carousels, Progress Indicators, Alerts/ Toasts, and Sliders.

Tabs & Carousels

Progress Indicators Alerts & Toasts

Sliders & Range

Advanced Accessible Components

Demonstrating Tabs/Carousels, Progress Indicators, Alerts/Toasts, and Sliders in one screen

Tabs & Carousels

Tab One | Tab Two | Tab Three

Current tab: Tab One

JSX Copy

```jsx
// Minimal Tabs
const [selectedTab,
setSelectedTab] = useState(0);
const tabs = ['Tab One', 'Tab
Two', 'Tab Three'];

<View style={{ flexDirection:
'row' }}>
  {tabs.map((tab, idx) => (
    <TouchableOpacity
      key={idx}
      accessibilityRole="tab"
      accessibilityLabel={`Select
${tab}`}

accessibilityState={{ selected:
selectedTab === idx }}
      onPress={() =>
```

8:21

Progress Indicators

Current progress: 75%

0% 25% 50% 75% 100%

JSX Copy

```jsx
// Basic progress bar
const [progress, setProgress] =
useState(0);

const progressAnimated = new
Animated.Value(progress);

useEffect(() => {

Animated.timing(progressAnimated,
{
    toValue: progress,
    duration: 300,
    useNativeDriver: false,
  }).start();
}, [progress]);

<Animated.View
  style={{
    height: 10,
    backgroundColor: 'blue',
    width:
progressAnimated.interpolate({
      inputRange: [0, 100],
      outputRange: ['0%', '100%'],
    }),
  }}
/>
```

```
      inputRange: [0, 100],
      outputRange: ['0%', '100%'],
    }),
  }}
/>
```

Alerts & Toasts

Trigger Alert

Something happened!

JSX Copy

```jsx
// Minimal toast/alert
const [showToast, setShowToast] =
useState(false);

function showToastMessage() {
  setShowToast(true);
  AccessibilityInfo.announceFor
Accessibility('Alert: Something
happened');
  setTimeout(() =>
setShowToast(false), 2000);
}

{showToast && (
  <View style={{ ... }}>
    <Text>Something happened!</
Text>
  </View>
)}
```

Sliders & Range Inputs

Current slider value: 30.46875

JSX Copy

```jsx
// Minimal slider example using
@react-native-community/slider
import Slider from '@react-native
-community/slider';

<Slider
  minimumValue={0}
  maximumValue={100}
  value={sliderValue}
  onSlidingComplete={(val) => {
    setSliderValue(val);
    AccessibilityInfo.announceForA
ccessibility(`Slider value set to
${Math.round(val)}`);
  }}
  style={{ width: '100%' }}
  minimumTrackTintColor={colors.pr
imary}
  maximumTrackTintColor="#ccc"
/>
<Text>Current slider value:
{sliderValue}</Text>
```

Accessibility Features

Tab Navigation

Proper role and state management for tab

```
    AccessibilityInfo.announceForA
ccessibility(`Slider value set to
${Math.round(val)}`);
  }}
  style={{ width: '100%' }}
  minimumTrackTintColor={colors.pr
imary}
  maximumTrackTintColor="#ccc"
/>
<Text>Current slider value:
{sliderValue}</Text>
```

Accessibility Features

Tab Navigation

Proper role and state management for tab controls

Progress Updates

Live announcements of progress changes

Alert Notifications

Immediate feedback for important events

Slider Controls

Accessible range inputs with value announcements

Mobile Accessibility Best Practices

Essential guidelines for creating accessible React Native applications

WCAG Guidelines

Documentation

Understanding and implementing WCAG 2.2 guidelines in mobile apps

Success Criteria Examples

Semantic Structure

Code Examples

Creating meaningful and well-organized content hierarchies

Hierarchy Implementation

Gesture Tutorial

Interactive Guide

Learn and test common accessibility gestures

Gesture Patterns Interactive Demo

Screen Reader Support

Guidelines

2.2 guidelines in mobile apps

Success Criteria Examples

Semantic Structure

Code Examples

Creating meaningful and well-organized content hierarchies

Hierarchy Implementation

Gesture Tutorial

Interactive Guide

Learn and test common accessibility gestures

Gesture Patterns Interactive Demo

Screen Reader Support

Guidelines

Optimizing your app for VoiceOver and TalkBack

Platform-specific Gestures

Logical Focus Order

Interactive Guide

Managing focus and keyboard navigation effectively

Focus Flow Interactive Demo

Accessible Button

Learn how to implement an accessible, properly labeled button with minimal touch target and role/hint.

Button Demo

Submit

Code Implementation

JSX Copy

```jsx
<TouchableOpacity
  accessibilityRole="button"
  accessibilityLabel="Submit form"
  accessibilityHint="Activates form
submission"
  style={{
    minHeight: 44,
    paddingHorizontal: 16,
    backgroundColor: colors.primary,
    borderRadius: 8,
    justifyContent: 'center',
    alignItems: 'center',
  }}
>
  <Text style={{ color: isDarkMode ?
colors.surface : colors.background }}>
```

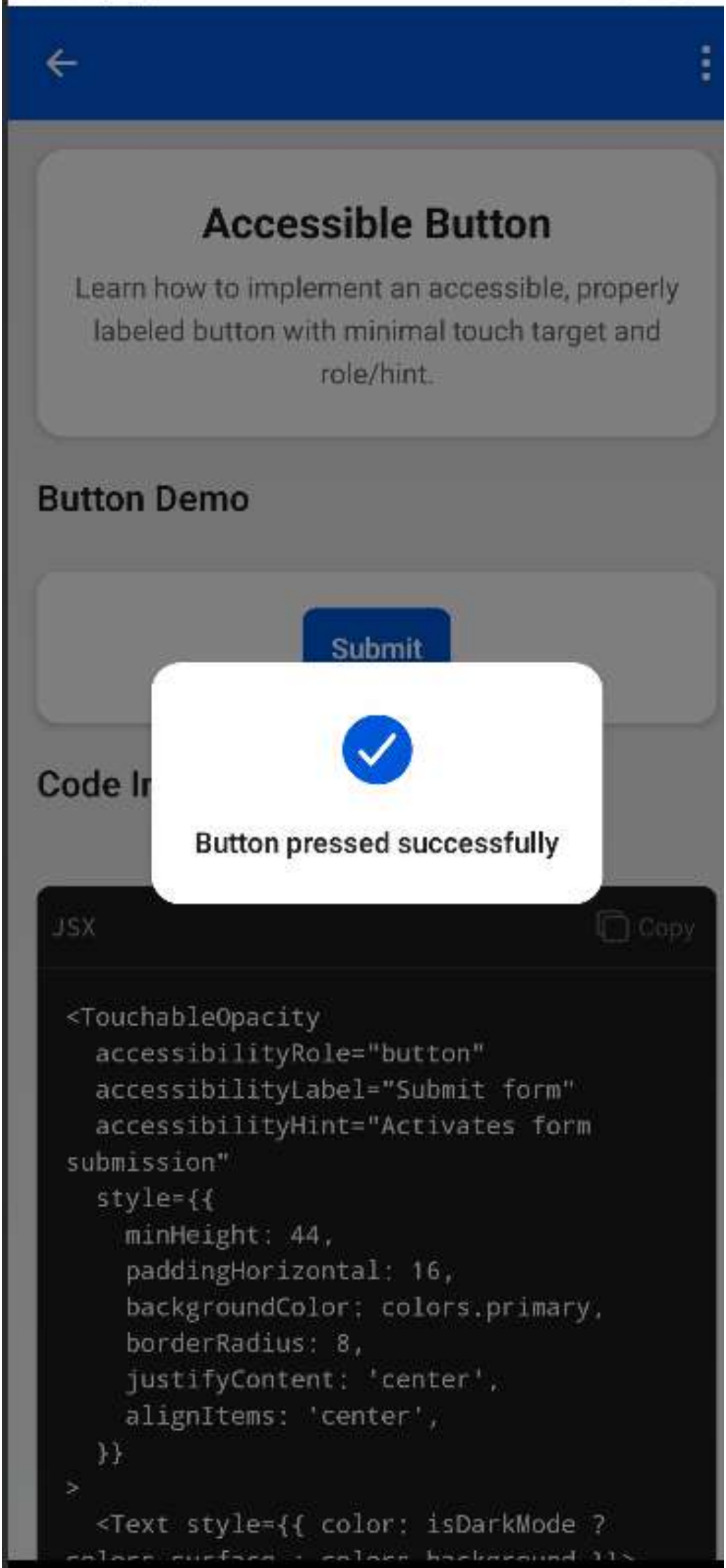
```
<TouchableOpacity
  accessibilityRole="button"
  accessibilityLabel="Submit form"
  accessibilityHint="Activates form
submission"
  style={{
    minHeight: 44,
    paddingHorizontal: 16,
    backgroundColor: colors.primary,
    borderRadius: 8,
    justifyContent: 'center',
    alignItems: 'center',
  }}
>
  <Text style={{ color: isDarkMode ?
colors.surface : colors.background }}>
    Submit
  </Text>
</TouchableOpacity>
```

Accessibility Features

Minimum Touch Target
44x44 points ensures the button is easy to tap

Screen Reader Label
Clear description announces the button's purpose

Action Hint
Additional context about what happens on activation


Accessible Button
Learn how to implement an accessible, properly labeled button with minimal touch target and role/hint.
Button Demo
Submit
Button pressed successfully
Code In
JSX
Copy
<TouchableOpacity
accessibilityRole="button"
accessibilityLabel="Submit form"
accessibilityHint="Activates form submission"
style={{
minHeight: 44,
paddingHorizontal: 16,
backgroundColor: colors.primary,
borderRadius: 8,
justifyContent: 'center',
alignItems: 'center',
}}
>
<Text style={{ color: isDarkMode ?

8:26

Accessibility Instruction & Community

Explore in-depth articles, best practices, and community success stories to master accessibility in mobile development.

Understanding WCAG Guidelines

Essential

Learn the fundamentals of WCAG and why they are critical for accessible apps. Discover common pitfalls and success strategies.

Read More >

Designing Accessible Interfaces

Best Practices

Get practical tips on designing interfaces that cater to users with disabilities. See real-world examples and guidelines.

Read More >

Focus Management

Interactive

Designing Accessible Interfaces

Best Practices

Get practical tips on designing interfaces that cater to users with disabilities. See real-world examples and guidelines.

Read More >

Focus Management Techniques

Interactive

Understand how to manage focus effectively in your applications to improve navigation and usability.

Read More >

Community Success Stories

Community

Read inspiring case studies and share your own accessibility achievements with our community.

Read More >

Modal Dialogs - Interactive Example

Build dialogs with focus trapping, screen reader support, and proper roles.

Dialog Demo

Open Dialog

Code Implementation

JSX Copy

```jsx
// Accessible Dialog Implementation
const AccessibleDialog = ({ visible,
onClose, title, children }) => {
  const closeRef = useRef(null);
  const contentRef = useRef(null);

  useEffect(() => {
    if (visible) {
      // Focus first element when
dialog opens
      contentRef.current?.focus();
    }
  }, [visible]);

  return (
    <Modal
      visible={visible}
```

8:20

JSX Copy

```jsx
// Accessible Dialog Implementation
const AccessibleDialog = ({ visible,
onClose, title, children }) => {
  const closeRef = useRef(null);
  const contentRef = useRef(null);

  useEffect(() => {
    if (visible) {
      // Focus first element when
dialog opens
      contentRef.current?.focus();
    }
  }, [visible]);

  return (
    <Modal
      visible={visible}
      transparent
```

Accessibility Features

Focus Management

Proper focus trapping and restoration when the dialog opens and closes

Keyboard Navigation

Full keyboard support including escape key to close the dialog

Screen Reader Support

Proper ARIA roles and live region announcements

8:20

Modal Dialogs - Interactive Example

Build dialogs with focus trapping, screen reader support, and proper roles.

Dialog Demo

Example Dialog

This is an example of an accessible dialog with proper focus management, keyboard interactions, and screen reader announcements.

Cancel Confirm

```
onClose, title, children }) => {
  const closeRef = useRef(null);
  const contentRef = useRef(null);

  useEffect(() => {
    if (visible) {
      // Focus first element when
dialog opens
      contentRef.current?.focus();
    }
  }, [visible]);

  return (
    <Modal
      visible={visible}
```

Modal Dialogs - Interactive Example

Build dialogs with focus trapping, screen reader support, and proper roles.

Dialog Demo

Open Dialog

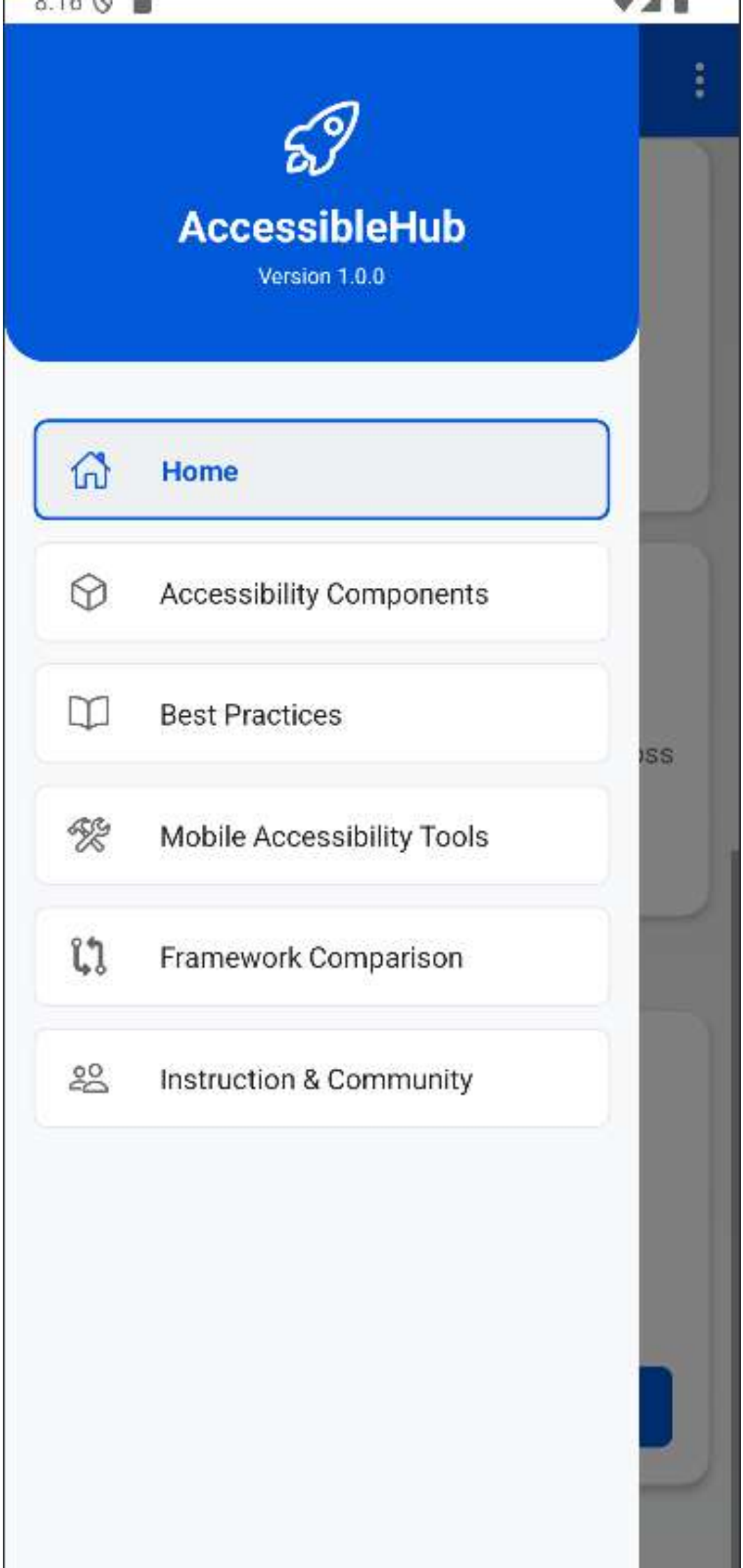
Success!

Your action has been confirmed.

```
// Accessible Dialog Implementation
const AccessibleDialog = ({ visible,
onClose, title, children }) => {
  const closeRef = useRef(null);
  const contentRef = useRef(null);

  useEffect(() => {
    if (visible) {
      // Focus first element when
dialog opens
      contentRef.current?.focus();
    }
  }, [visible]);

  return (
    <Modal
      visible={visible}
```


AccessibleHub
Version 1.0.0
Home
Accessibility Components
Best Practices
Mobile Accessibility Tools
Framework Comparison
Instruction & Community

8:24

Logical Focus Order

Demonstrate skip links and consistent navigation order.

Skip to Main Content

Why Focus Order Matters

Proper focus order helps screen reader and keyboard users navigate without confusion. A skip link allows bypassing repetitive blocks, ensuring more efficient access to primary content.

Main Content

Below are interactive elements in a logical sequence.

Focusable Button 1

Enter feedback

Submit Feedback

Skip to Main Content

Why Focus Order Matters

Proper focus order helps screen reader and keyboard users navigate without confusion.

Feedback Submitted

Feedback: aaaa

OK

Main Content

Below are interactive elements in a logical sequence.

Focusable Button 1

aaaa

Submit Feedback