

# Accessibility Handbook

Your one-stop guide for building accessible applications.



# What and why?

## What is Accessibility?

Accessibility (A11y) is about making digital experiences usable for everyone, including people with visual, auditory, motor, or cognitive disabilities. It's the practice of writing code, designing interfaces, and structuring content so that no user is left out, no matter how they interact with technology.

As developers, accessibility isn't just another checklist. It's part of building quality software. It means using semantic HTML, managing focus correctly, testing with real assistive tools, and ensuring that every component works for all users.




## Why it Matters?

Accessibility is inclusion in action. When we make products accessible, we're opening doors for millions of users who rely on assistive technologies like screen readers, keyboards, voice commands, and more.

Beyond empathy, accessibility also improves performance, SEO, usability, and code structure. It's good for users, good for business, and good for developers who care about clean, future-proof solutions.

Ultimately, accessibility isn't an optional feature; it's a mindset. It's how we ensure the web and apps we build are usable, flexible, and open to everyone.

## What You'll Find in This Handbook

-  **Overview** High-level introduction to what accessibility means in digital products.
-  **Web** Making Web Applications accessible
-  **Mobile** For Mobile Accessible apps

# Table of Contents

## Part 1

### Chapter 1 :Digital Accessibility

1.1 Understanding Disability	08
1.2 The Global Impact	09

### Chapter 2 :The POUR Principles

2.1 The Four Principles	10
2.2 Perceivable in Detail	11
2.3 Operable in Detail	11
2.4 Understandable in Detail	11
2.5 Robust in Detail	11

### Chapter 3: WCAG Conformance

3.1 The Three Levels	12
3.2 Level AA Requirements (Target)	12

### Chapter 4: Assistive Technologies

4.1 Screen Readers	13
4.2 Screen Magnification	13
4.3 Voice Control and Dictation	14
4.4 Alternative Input Devices	14

### Chapter 5: Testing Methodology

5.1 The Testing Pyramid	15
5.2 Automated Testing Tools	15
5.3 Manual Testing Checklist	15
5.4 User Testing with People with Disabilities	16

---

---

## Chapter 6: Semantic HTML - The Foundation

6.1 Why Semantic HTML Matters	18
6.2 Document Structure	18
6.3 Heading Hierarchy	19
6.4 Links and Buttons	20
6.5 Form Elements	21

## Chapter 7: ARIA - When and How

7.1 The Five Rules of ARIA	22
7.2 Common ARIA Roles	22
7.3 ARIA States and Properties	23
7.4 ARIA Live Regions	24

## Chapter 8: Keyboard Navigation

8.1 Focus Management	25
8.2 Tab Order and tabindex	25
8.3 Keyboard Event Handling	26
8.4 Modal Dialogs and Focus Traps	26

## Chapter 9: Forms and Input Validation

9.1 Form Labels and Association	27
9.2 Error Handling and Validation	27
9.3 Required Fields and Instructions	28

## Chapter 10: Color, Contrast, and Visual Design

10.1 Color Contrast Requirements	29
10.2 Don't Rely on Color Alone	29
10.3 Responsive and Zoom-Friendly Design	29

## Chapter 11: React Native Accessibility Fundamentals

11.1 Core Accessibility Props	31
11.2 Accessibility Roles	32
11.3 Accessibility State	33
11.4 Accessibility Value	33

## Chapter 12: Screen Readers on Mobile

12.1 VoiceOver on iOS	34
12.2 TalkBack on Android	35
12.3 Platform Differences	35

## Chapter 13: Touch Targets and Gestures

13.1 Minimum Touch Target Sizes	36
13.2 Using hitSlop and pressRetentionOffset	36
13.3 Gesture Conflicts with Screen Readers	37

## Chapter 14: Forms and Input in React Native

14.1 Accessible Text Inputs	38
14.2 Checkboxes and Radio Buttons	38
14.3 Form Validation and Error Messages	38

## Chapter 15: Focus Management and Navigation

15.1 Managing Focus Programmatically	40
15.2 Screen Reader Announcements	40

## Chapter 16: Lists, Navigation, and Complex Components

16.1 Accessible FlatList and SectionList	42
16.2 Tab Navigation	42
16.3 Accessible Modals and Bottom Sheets	43

## Chapter 17: Testing React Native Accessibility

17.1 Automated Testing	44
17.2 Manual Testing Checklist	44
17.3 Using Accessibility Inspector Tools	45

## Chapter 18: Best Practices and Common Patterns

Examples	47
Resources	51

## Closing Remarks



# PART 1

# Digital Accessibility

# Chapter 1: What is Digital Accessibility?

Digital accessibility ensures that digital products — websites, mobile apps, documents, and technologies — are designed so that people with disabilities can perceive, understand, navigate, and interact with them effectively.

## 1.1 Understanding Disability

Disability is diverse and affects people in different ways.  
Digital accessibility addresses multiple types of disabilities:

- **Visual:** Blindness, low vision, color blindness
- **Auditory:** Deafness, hard of hearing
- **Motor/Physical:** Limited dexterity, tremors, paralysis
- **Cognitive:** Learning disabilities, memory impairments, attention disorders
- **Neurological:** Seizure disorders, vestibular disorders
- **Speech:** Difficulty or inability to speak

**Temporary and Situational Disabilities:** Accessibility also benefits people with temporary impairments (broken arm, eye surgery) or situational limitations (bright sunlight, noisy environment, slow internet).





## 1.2 The Global Impact

According to the World Health Organization (WHO):

- Over 1.3 billion people (16% of global population) experience significant disability
- 253 million people are visually impaired
- 466 million people have disabling hearing loss
- These numbers are growing as populations age

# ACCESSIBILITY GLOBAL IMPACT



### THE BUSINESS CASE

- **Market Reach**  
People with disabilities represent a \$13 trillion market globally
- **Legal Compliance**  
Avoid lawsuits (ADA, Section 508, EN 501 549, AODA)
- **Better SEO**  
Accessible sites rank higher in search engines
- **Improved Usability**  
Benefits all users, not just those with disabilities
- **Brand Reputation**  
Demonstrates corporate social responsibility
- **Innovation**  
Accessibility drives innovation (voice controls, captions, dark mode)

### THE GLOBAL IMPACT

According to the World Health Organization (WHO):

- **Over 1.3 billion people** (16% of global population) experience significant disability
- **253 million people** are visually impaired
- **466 million people** have disabling hearing loss

**These numbers are growing as populations age**

# Chapter 2 :The POUR Principles

The Web Content Accessibility Guidelines (WCAG) are built on four fundamental principles, known by the acronym POUR. These principles apply to all digital platforms.

## 2.1 The Four Principles

WCAG 2.1 (and the newer WCAG 2.2) are organized around four principles:

**Perceivable:** Information and user interface components must be presentable to users in ways they can perceive

**Operable:** User interface components and navigation must be operable

**Understandable:** Information and the operation of the user interface must be understandable

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

### POUR Principles



## 2.2 Perceivable in Detail

Users must be able to perceive the information being presented. Key guidelines:

- **Text Alternatives:** Provide text alternatives for non-text content
- **Time-based Media:** Provide alternatives for audio and video
- **Adaptable:** Content can be presented in different ways without losing information
- **Distinguishable:** Make it easier to see and hear content (contrast, text size, color)

## 2.3 Operable in Detail

Users must be able to operate the interface. Key guidelines:

- **Keyboard Accessible:** All functionality available from keyboard
- **Enough Time:** Users have enough time to read and use content
- **Seizures:** Don't design content that causes seizures
- **Navigable:** Help users navigate, find content, and determine where they are
- **Input Modalities:** Make it easier to operate functionality through various inputs

## 2.4 Understandable in Detail

Information and operation of the user interface must be understandable.

Key guidelines:

- **Readable:** Make text readable and understandable
- **Predictable:** Web pages appear and operate in predictable ways
- **Input Assistance:** Help users avoid and correct mistakes

## 2.5 Robust in Detail

Content must be robust enough to work with current and future technologies.

Key guidelines:

- **Compatible:** Maximize compatibility with current and future user agents and assistive technologies
- **Valid Code:** Use valid, semantic code that assistive technologies can parse
- **Name, Role, Value:** All components have accessible names, roles, and values

# Chapter 3: WCAG Conformance Levels

WCAG defines three levels of conformance, each building on the previous level.



## 3.1 The Three Levels

Understanding conformance levels helps you set realistic accessibility goals:

- **Level A:** The most basic level of web accessibility. Failure to meet Level A creates significant barriers for users with disabilities. Essential foundation.
- **Level AA:** Deals with the biggest and most common barriers. This is the target level for most organizations and is often legally required. Recommended baseline.
- **Level AAA:** The highest level of accessibility. Not required for entire sites as some content cannot meet AAA. Target for specific content where appropriate.

**Recommended Approach:** Most organizations should target Level AA conformance across their entire digital presence, with Level AAA for specific critical content or features.

## 3.2 Level AA Requirements (Target)

Key Level AA requirements you'll encounter frequently:

- **Color Contrast:** 4.5:1 for normal text, 3:1 for large text
- **Resize Text:** Text can be resized up to 200% without loss of content
- **Reflow:** Content reflows to single column without horizontal scrolling
- **Non-text Contrast:** 3:1 contrast for UI components
- **Text Spacing:** No loss of content when text spacing is adjusted
- **Focus Visible:** Keyboard focus indicator is visible
- **Consistent Navigation:** Navigation is consistent across pages

# Chapter 4: Assistive Technologies

Understanding assistive technologies helps developers create better accessible experiences. These technologies act as bridges between users with disabilities and digital content.

## 4.1 Screen Readers

Screen readers convert digital text into synthesized speech or refreshable Braille. They're used by blind and low-vision users.

Major Screen Readers:

- **JAWS:** Job Access With Speech (Windows, commercial)
- **NVDA:** NonVisual Desktop Access (Windows, free)
- **VoiceOver:** Built into macOS, iOS, iPadOS (free)
- **TalkBack:** Built into Android (free)
- **Narrator:** Built into Windows (free)

How Screen Readers Work?

- Parse HTML structure and semantics
- Announce headings, landmarks, links, form controls
- Allow navigation by element type (e.g., 'next heading', 'next link')
- Provide virtual cursor for exploring content
- Support various navigation modes (browse, forms, tables)

## 4.2 Screen Magnification

Screen magnifiers enlarge portions of the screen for users with low vision:

- **ZoomText:** Windows screen magnifier with speech
- **Magnifier:** Built into Windows and macOS
- **Browser Zoom:** Built-in zoom in all modern browsers

Design Considerations:

- Content must reflow when zoomed to 200%
- No horizontal scrolling at high zoom levels
- Maintain adequate spacing between interactive elements

## 4.3 Voice Control and Dictation

Voice control software allows users to navigate and interact using voice commands:

- **Dragon NaturallySpeaking:** Advanced voice recognition (Windows)
- **Voice Control:** Built into macOS and iOS
- **Voice Access:** Built into Android

Design Considerations:

- Use visible labels that match accessible names
- Provide unique names for similar elements
- Ensure all interactive elements are properly labeled

## 4.4 Alternative Input Devices

Users with motor disabilities may use alternative input devices:

- **Switch Controls:** One or two button input devices
- **Eye Tracking:** Control with eye movements
- **Head Mouse:** Control cursor with head movements
- **Mouth Stick:** Physical pointer held in mouth
- **Sip-and-Puff:** Air pressure controls

Design Considerations:

- Ensure full keyboard accessibility (most alternative devices emulate keyboard)
- Provide adequate target sizes (at least 44x44 pixels on mobile)
- Allow sufficient time for input
- Avoid time-based interactions or provide alternatives

# Chapter 5: Testing Methodology

## 5.1 The Testing Pyramid

A comprehensive testing strategy includes three layers:

- **Automated Testing (Foundation):** Catches 30-40% of issues, runs continuously
- **Manual Testing (Core):** Catches 50-60% of issues, requires expertise
- **User Testing (Peak):** Catches remaining issues, provides real-world validation

All three layers are necessary. Automated tools alone are insufficient.

## 5.2 Automated Testing Tools

Browser Extensions (Manual Audits):

- **axe DevTools:** Most accurate, detailed reports (Free tier available)
- **WAVE:** Visual feedback on page, good for learning
- **Lighthouse:** Built into Chrome DevTools, part of performance audits
- **IBM Equal Access Checker:** Free, comprehensive

CI/CD Integration (Automated):

- **Pa11y:** Command-line tool, integrates with CI/CD
- **axe-core:** JavaScript library for integration
- **jest-axe:** For React/Jest testing
- **Cypress-axe:** For Cypress E2E testing

## 5.3 Manual Testing Checklist

Essential Manual Tests:

- **Keyboard Navigation:** Navigate entire site using only Tab, Shift+Tab, Enter, Space, Arrow keys
- **Focus Visibility:** Ensure focus indicator is always visible and has 3:1 contrast
- **Screen Reader:** Test with NVDA (Windows) or VoiceOver (Mac)
- **Zoom to 200%:** Verify no content loss, no horizontal scrolling
- **Color Contrast:** Check all text and UI components meet minimum ratios
- **Color Independence:** Ensure color isn't the only way to convey information
- **Form Errors:** Verify errors are clearly announced and associated with fields
- **Alt Text:** Check all images have appropriate alternative text

- **Video Captions:** Verify all videos have accurate captions
- **Heading Structure:** Check logical heading hierarchy (no skipped levels)

Screen Reader Testing Commands:

- **NVDA:** Insert+Down Arrow (browse mode), Insert+Space (switch modes)
- **VoiceOver:** VO+A (read all), VO+Command+H (next heading), VO+Command+L (next link)
- **TalkBack:** Swipe right (next), swipe left (previous), double-tap (activate)

## 5.4 User Testing with People with Disabilities

Nothing replaces testing with actual users who have disabilities. They'll find issues that automated tools and expert reviews miss.

How to Conduct User Testing:

- Recruit users with diverse disabilities (vision, motor, cognitive, hearing)
- Provide scenarios that reflect real use cases
- Let users use their own assistive technologies and settings
- Observe without interrupting, ask questions afterward
- Compensate participants fairly for their time

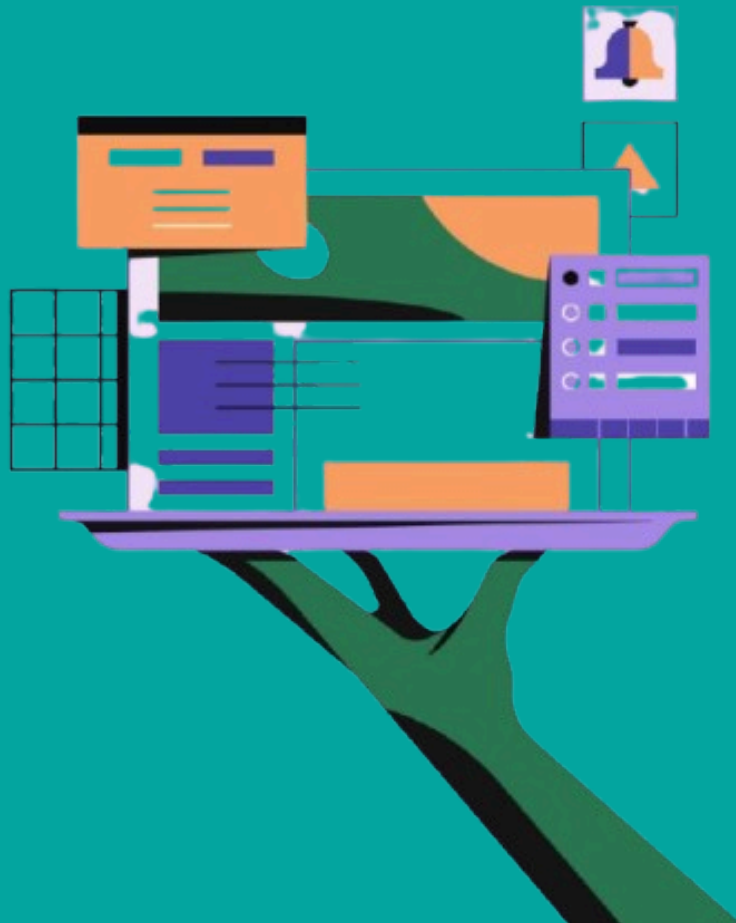
Where to Find Testers:

- **Fable (fable.io)** - Platform connecting to users with disabilities
- **Access Works** - User testing service
- Local disability organizations and advocacy groups
- Existing customers who use assistive technologies



# Part 2

# Web Accessibility



# Chapter 6: Semantic HTML - The Foundation

Semantic HTML provides meaning and structure to content, making it accessible to assistive technologies. It's the foundation of web accessibility.

## 6.1 Why Semantic HTML Matters

Semantic HTML elements clearly describe their meaning to both browsers and assistive technologies. Using the correct HTML elements provides:

- Built-in Accessibility: Native keyboard support, roles, and behaviors
- Screen Reader Support: Proper announcement and navigation
- Better SEO: Search engines understand content structure
- Maintainability: Easier to understand and update
- Cross-device Compatibility: Works better on different devices

## 6.2 Document Structure

Proper document structure with semantic elements:



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Accessible Page</title>
</head>
<body>
  <header>
    <nav aria-label="Main navigation">
      <a href="/">Home</a> |
      <a href="/about">About</a> |
      <a href="/contact">Contact</a>
    </nav>
  </header>

  <main id="main-content">
    <h1>Page Title</h1>

    <article>
      <h2>Article Heading</h2>
      <p>Article content goes here...</p>
    </article>

    <aside aria-label="Related links">
      <h2>Related Articles</h2>
      <a href="/article1">Article 1</a>
    </aside>
  </main>

  <footer>
    <p>&copy; 2025 Company Name</p>
  </footer>
</body>
</html>
```

Key Semantic Elements:

- **<header>**: Contains introductory content or navigation
- **<nav>**: Contains navigation links
- **<main>**: Contains the main content (use only once per page)
- **<article>**: Self-contained content that could stand alone
- **<section>**: Thematic grouping of content with a heading
- **<aside>**: Content tangentially related to surrounding content
- **<footer>**: Contains footer information

## 6.3 Heading Hierarchy

Headings create a document outline that screen reader users can navigate. Proper heading structure is critical:

- Use one <h1> per page (the page title)
- Don't skip heading levels (h1 → h2 → h3, not h1 → h3)
- Use headings for structure, not styling
- Ensure headings accurately describes the content that follows

```

● ● ●
<!-- ✔ Good Heading Hierarchy -->
<section>
  <h1>Introduction to Web Accessibility</h1>

  <h2>What is Accessibility?</h2>
  <p>Content...</p>

  <h3>Types of Disabilities</h3>
  <p>Content...</p>

  <h3>Benefits of Accessibility</h3>
  <p>Content...</p>

  <h2>Getting Started</h2>
  <p>Content...</p>

  <h3>Essential Tools</h3>
  <p>Content...</p>
</section>
```

```

● ● ●
<!-- ✗ bad Heading Hierarchy: Skipped Levels -->
<section>
  <h1>Title</h1>
  <h4>Should be h2</h4>
</section>
```

## 6.4 Links and Buttons

Links and buttons are different and should be used appropriately:

- **Links (<a>):** Navigate to a different page or location
- **Buttons (<button>):** Trigger an action on the current page

```
<!-- ✅ Accessible icon button -->
<button aria-label="Close dialog">
  <span aria-hidden="true">&times;</span>
</button>

<!-- ✅ Toggle button -->
<button
  aria-pressed="false"
  onclick="this.setAttribute('aria-pressed',
    this.getAttribute('aria-pressed') === 'true' ? 'false' : 'true')"
>
  Bold
</button>

<!-- ✅ Expandable section -->
<button aria-expanded="false" aria-controls="details" onclick="toggleDetails()">
  Show Details
</button>
<div id="details" hidden>Details content...</div>

<!-- ✅ Live region for updates -->
<div aria-live="polite" aria-atomic="true">
  <p>Items in cart: <span id="cart-count">0</span></p>
</div>

<!-- ✅ Current page in navigation -->
<nav aria-label="Main">
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/products" aria-current="page">Products</a>
</nav>
```

### Link Best Practices:

- Use descriptive link text (not 'click here' or 'read more')
- Indicate if link opens in new window
- Distinguish visited from unvisited links
- Ensure adequate spacing between links

## 6.5 Form Elements

Forms must have proper labels and structure for accessibility:

```
<form>
  <!-- ✅ Text input with explicit label -->
  <label for="name">Full Name:</label>
  <input type="text" id="name" name="name" required>
  <br><br>

  <!-- ✅ Input with additional description -->
  <label for="email">Email Address:</label>
  <input
    type="email"
    id="email"
    name="email"
    aria-describedby="email-hint"
    required
  >
  <div id="email-hint">We'll never share your email.</div>
  <br><br>

  <!-- ✅ Radio buttons (grouped) -->
  <fieldset>
    <legend>Choose your plan:</legend>
    <label>
      <input type="radio" name="plan" value="basic"> Basic
    </label>
    <label>
      <input type="radio" name="plan" value="pro"> Pro
    </label>
  </fieldset>
  <br>

  <!-- ✅ Checkbox -->
  <label>
    <input type="checkbox" name="newsletter" value="yes">
    Subscribe to newsletter
  </label>
  <br><br>

  <!-- ✅ Select dropdown -->
  <label for="country">Country:</label>
  <select id="country" name="country" required>
    <option value="">Select a country</option>
    <option value="us">United States</option>
    <option value="ca">Canada</option>
  </select>
  <br><br>

  <!-- ✅ Submit button -->
  <button type="submit">Create Account</button>
</form>
```

# Chapter 7: ARIA - When and How

ARIA (Accessible Rich Internet Applications) fills gaps in HTML semantics. However, it must be used correctly — incorrect ARIA is worse than no ARIA.

## 7.1 The Five Rules of ARIA

Before using ARIA, understand these fundamental rules:

- **Rule 1:** If you can use a native HTML element or attribute with the semantics and behavior you require, instead of re-purposing an element and adding ARIA, then do so.
- **Rule 2:** Do not change native semantics, unless you really have to.
- **Rule 3:** All interactive ARIA controls must be usable with the keyboard.
- **Rule 4:** Do not use `role='presentation'` or `aria-hidden='true'` on a focusable element.
- **Rule 5:** All interactive elements must have an accessible name.

## 7.2 Common ARIA Roles

### What Are ARIA Roles (Web)

- ARIA (Accessible Rich Internet Applications) roles define the purpose or type of an element so that assistive technologies (like screen readers) can understand it.
- Roles are added via the `role` attribute in HTML, for example: `<div role="button">Click me</div>`.
- Many HTML elements have implicit roles, meaning the browser already knows what they are (e.g., `<button>` has role “button”).
- When you use a non-semantic element (like `<div>`), defining a role gives it meaning in the accessibility tree.

### Key Types of ARIA Roles

- Document Structure Roles
  - Describe the organization of content.
  - Examples: heading, paragraph, term, definition
  - Use semantic HTML when possible instead of ARIA for these.
- Widget Roles
  - Define interactive UI components.
  - Examples: button, checkbox, combobox, textbox, slider
  - Helps screen readers understand how to interact with custom widgets.
- Landmark Roles
  - Used for major page regions to help with navigation.

- Examples: navigation, main, complementary, banner, contentinfo
- Should be used sparingly — too many landmarks can overwhelm assistive tech.
- Live Region Roles
  - Inform screen readers about dynamically updated content.
  - Examples: alert, status
- Window Roles
  - For defining sub-windows or dialogs in a page.
  - Examples: dialog, alertdialog
- Abstract Roles
  - These are not meant for direct use in HTML — they help define the ARIA role hierarchy.
  - Examples: structure, range, widget (abstract)

## 7.3 ARIA States and Properties

ARIA attributes provide additional information about elements:

- **aria-label:** Provides an accessible name
- **aria-labelledby:** References element(s) that label this element
- **aria-describedby:** References element(s) that describe this element
- **aria-hidden:** Hides element from assistive technologies
- **aria-expanded:** Indicates if element is expanded or collapsed
- **aria-pressed:** Indicates pressed state of toggle button
- **aria-disabled:** Indicates element is disabled
- **aria-invalid:** Indicates validation error
- **aria-live:** Indicates region will be dynamically updated
- **aria-current:** Indicates current item in a set

```

<!-- ✅ Accessible icon button -->
<button aria-label="Close dialog">
  <span aria-hidden="true">&times;</span>
</button>

<!-- ✅ Toggle button -->
<button
  aria-pressed="false"
  onclick="this.setAttribute('aria-pressed',
    this.getAttribute('aria-pressed') === 'true' ? 'false' : 'true')"
>
  Bold
</button>

<!-- ✅ Expandable section -->
<button aria-expanded="false" aria-controls="details" onclick="toggleDetails()">
  Show Details
</button>
<div id="details" hidden>Details content...</div>

<!-- ✅ Live region for updates -->
<div aria-live="polite" aria-atomic="true">
  <p>Items in cart: <span id="cart-count">0</span></p>
</div>

<!-- ✅ Current page in navigation -->
<nav aria-label="Main">
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/products" aria-current="page">Products</a>
</nav>

```

## 7.4 ARIA Live Regions

Live regions announce dynamic content changes to screen readers:

- **aria-live='polite'**: Announces when user is idle
- **aria-live='assertive'**: Interrupts to announce immediately
- **aria-atomic**: Announces entire region vs. just changes
- **role='status'**: For status messages (implicitly polite)
- **role='alert'**: For important messages (implicitly assertive)

```
<!-- ✓ Status message -->
<div role="status" aria-live="polite">
  Changes saved successfully
</div>

<!-- ✓ Alert message -->
<div role="alert">
  Error: Your session has expired
</div>

<!-- ✓ Loading state -->
<div aria-live="polite" aria-busy="true" aria-label="Loading content">
  Loading...
</div>

<!-- ✓ Search results counter -->
<div role="status" aria-live="polite" aria-atomic="true">
  Found <span id="result-count">0</span> results
</div>
```



# Chapter 8: Keyboard Navigation

Keyboard accessibility ensures that users with motor disabilities, blind users, and even power users can fully operate your website without a mouse. Every interactive element must be reachable, operable, and understandable using only the keyboard.

Good keyboard UX depends on three pillars:

1. Everything must be focusable
2. Focus must be visible
3. Focus must move in a logical, predictable order

## 8.1 Focus Management

Proper focus management ensures users always know where they are and what will happen next.

Key Principles

- All interactive elements must be keyboard focusable.
- Focus indicator must be visible with at least 3:1 contrast.
- Focus order should follow the visual layout.
- Never “trap” focus — except inside modals.
- When content changes (e.g., opening a dialog), move focus intentionally to the new element.
- 

## 8.2 Tab Order and tabindex

The `tabindex` attribute controls keyboard focus:

- **`tabindex='0'`**: Makes element focusable in natural tab order
- **`tabindex='-1'`**: Makes element programmatically focusable only
- **`tabindex='1+'`**: Creates explicit tab order (avoid unless necessary)

**Best Practices:**

- Use native interactive elements when possible (built-in tab order)
- Only use positive `tabindex` values when absolutely necessary
- Test tab order with keyboard navigation
- Implement skip links for long navigation menus

## 8.3 Keyboard Event Handling

Custom interactive elements must respond to keyboard events:

Common Keyboard Patterns:

- **Tab/Shift+Tab:** Navigate between elements
- **Enter/Space:** Activate buttons and links
- **Arrow Keys:** Navigate within components (menus, tabs, radio groups)
- **Escape:** Close dialogs and menus
- **Home/End:** Jump to first/last item in lists

## 8.4 Modal Dialogs and Focus Traps

A modal dialog must:


- Trap focus inside the dialog
- Return focus to the trigger element when closed
- Prevent background content from receiving focus

# Chapter 9: Forms and Input Validation

Accessible forms enable all users to complete tasks successfully. Clear labels, helpful error messages, and proper validation are essential.

## 9.1 Form Labels and Association

Every `<input>` must have a properly associated `<label>` using `for` + `id`. This ensures screen readers announce the label when the user focuses on the field. Visual proximity alone is not enough for accessibility.

```

<!-- ✅ Correct (Accessible) Example -->
<label for="email">Email Address</label>
<input id="email" type="email" placeholder="Enter your email" />

<!-- ❌ Incorrect Example -->
<p>Email Address</p>
<input type="email" placeholder="Enter your email" />
```

## 9.2 Error Handling and Validation

Error messages must be:

- Clear: describe exactly what is wrong
- Specific: not “Invalid input”
- Announced using `aria-live` so screen readers detect the update
- Use `aria-invalid="true"` to mark invalid fields.

```

<label for="email">Email Address</label>
<input
  id="email"
  type="email"
  aria-invalid="true"
  aria-describedby="email-error"
/>

<p id="email-error" role="alert" aria-live="assertive">
  Please enter a valid email address.
</p>
```

## 9.3 Required Fields and Instructions

Clearly indicate required fields and provide instructions before the form:

```
<!-- Form instructions -->
<form aria-labelledby="form-title" aria-describedby="form-instructions">
  <h2 id="form-title">Create Account</h2>
  <p id="form-instructions">
    Fields marked with <span aria-label="required">*</span> are required.
  </p>

  <!-- Required field -->
  <label for="name">
    Full Name <span aria-label="required">*</span>
  </label>
  <input
    type="text"
    id="name"
    name="name"
    required
    aria-required="true"
  >

  <!-- Optional field -->
  <label for="phone">
    Phone Number <span class="optional">(optional)</span>
  </label>
  <input type="tel" id="phone" name="phone">

  <button type="submit">Create Account</button>
</form>
```

# Chapter 10: Color, Contrast, and Visual Design

Visual design must be perceivable by users with various visual abilities. Proper contrast, color usage, and responsive design are essential.

## 10.1 Color Contrast Requirements

WCAG defines minimum contrast ratios for text and UI components:

**Normal text (Level AA):** 4.5:1 contrast ratio

**Large text (Level AA):** 3:1 contrast ratio (18pt+ or 14pt+ bold)

**Normal text (Level AAA):** 7:1 contrast ratio

**Large text (Level AAA):** 4.5:1 contrast ratio

**UI components (Level AA):** 3:1 contrast ratio

**Focus indicators:** 3:1 contrast ratio against background

Tools for Checking Contrast:

- WebAIM Contrast Checker ([webaim.org/resources/contrastchecker/](https://webaim.org/resources/contrastchecker/))
- Chrome DevTools (Inspect element → Contrast ratio)
- Figma Contrast Plugin
- Stark (design tool plugin)

## 10.2 Don't Rely on Color Alone

Never use color as the only way to convey information. Users with color blindness need alternative indicators:

## 10.3 Responsive and Zoom-Friendly Design

Users must be able to zoom content to 200% without loss of functionality or content:

- Use relative units (rem, em, %) instead of fixed pixels
- Avoid horizontal scrolling at 200% zoom
- Use responsive design techniques
- Test at different zoom levels

# PART 3

## Mobile Accessibility (react native)



# Chapter 11: React Native Accessibility Fundamentals

React Native accessibility focuses on making mobile applications accessible on iOS and Android. This section covers platform-specific APIs, screen readers, and mobile interaction patterns.

## Overview

Mobile accessibility has unique challenges and opportunities. React Native provides cross-platform accessibility APIs, but understanding platform differences (iOS VoiceOver vs Android TalkBack) is crucial for creating truly accessible mobile apps.

React Native provides accessibility props that work across iOS and Android. Understanding these props is the foundation of mobile accessibility.

## 11.1 Core Accessibility Props

React Native provides several accessibility props that map to platform-specific APIs:

**accessible:** Groups children into a single accessibility element (default: true for touchable elements)

**accessibilityLabel:** Text read by screen readers (like alt text)

**accessibilityHint:** Additional context about what happens when activated

**accessibilityRole:** Describes the element type (button, link, header, etc.)

**accessibilityState:** Describes current state (selected, disabled, checked, expanded)

**accessibilityValue:** Current value of adjustable elements (sliders, progress bars)

**accessibilityActions:** Custom actions available to the user

```

import React from 'react';
import { TouchableOpacity, Text, View, Alert } from 'react-native';

// Basic accessible button
function AccessibleButton() {
  const handleAddToCart = () => {
    Alert.alert('Item added to cart!');
  };

  return (
    <View style={{ alignItems: 'center', justifyContent: 'center', flex: 1 }}>
      <TouchableOpacity
        accessible={true}
        accessibilityLabel="Add to cart"
        accessibilityHint="Adds this item to your shopping cart"
        accessibilityRole="button"
        onPress={handleAddToCart}
        style={{
          backgroundColor: '#007AFF',
          paddingVertical: 10,
          paddingHorizontal: 20,
          borderRadius: 8,
        }}
      >
        <Text style={{ color: 'white', fontSize: 16 }}>Add to Cart</Text>
      </TouchableOpacity>
    </View>
  );
}

export default AccessibleButton;

```

## 11.2 Accessibility Roles

The `accessibilityRole` prop describes what an element is or does:

- **button:** Interactive element that triggers an action
- **link:** Navigates to another screen or opens a URL
- **header:** Section heading (like h1-h6 in HTML)
- **text:** Static text that should not be interactive
- **image:** Image element
- **imagebutton:** Button containing an image
- **search:** Search input field
- **checkbox:** Toggle with checked/unchecked states
- **radio:** Radio button in a group
- **switch:** On/off toggle switch
- **adjustable:** Element with adjustable value (slider)
- **tab:** Tab in a tab bar
- **menu:** Menu component
- **menubar:** Container for menu items
- **menuitem:** Item within a menu



## 11.3 Accessibility State

The `accessibilityState` prop describes the current state of an element:

```
{/* Checkbox */}
<TouchableOpacity
  accessibilityRole="checkbox"
  accessibilityState={{ checked: isChecked }}
  onPress={() => setIsChecked(!isChecked)}
  style={styles.checkboxContainer}
>
  <View style={[styles.checkbox, isChecked && styles.checkedBox]}>
    {isChecked && <Text style={styles.checkmark}></Text>}
  </View>
  <Text style={styles.checkboxLabel}>Accept terms and conditions</Text>
</TouchableOpacity>
```

## 11.4 Accessibility Value

The `accessibilityValue` prop describes the current value of adjustable elements:

```
import React, { useState } from 'react';
import { View, Text, Button, AccessibilityInfo } from 'react-native';

export default function AccessibilityValueExample() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(prev => prev + 1);
    // Optional: Announce change to screen readers
    AccessibilityInfo.announceForAccessibility(`Counter value is ${count + 1}`);
  };

  return (
    <View
      style={{
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
      }}
      accessible={true}
      accessibilityLabel="Counter example"
      accessibilityHint="Double tap to increase the counter"
      accessibilityRole="adjustable"
      accessibilityValue={{ min: 0, max: 10, now: count, text: `${count}` }}
    >
      <Text style={{ fontSize: 24, marginBottom: 10 }}>Count: {count}</Text>
      <Button title="Increase" onPress={increment} />
    </View>
  );
}
```

# Chapter 12: Screen Readers on Mobile

VoiceOver (iOS) and TalkBack (Android) are the primary screen readers on mobile. Understanding how they work is essential for testing and optimizing accessibility.

## 12.1 VoiceOver on iOS

VoiceOver is Apple's screen reader, built into iOS, iPadOS, and macOS.

VoiceOver Gestures:

- **Swipe Right:** Move to next element
- **Swipe Left:** Move to previous element
- **Double Tap:** Activate selected element
- **Triple Tap:** Double-click selected element
- **Two-Finger Tap:** Pause/resume speech
- **Two-Finger Swipe Up:** Read all from top
- **Two-Finger Swipe Down:** Read all from current position
- **Rotor:** Two fingers, rotate to change navigation mode
- **Magic Tap:** Two-finger double tap (performs main action)

**VoiceOver Rotor: The rotor allows users to navigate by:**

- Headings
- Links
- Form controls
- Buttons
- Text fields
- Adjustable elements
- Images
- Containers

**Enabling VoiceOver:**

1. Settings → Accessibility → VoiceOver → Toggle On
2. Triple-click home button (if configured)
3. Ask Siri: 'Turn on VoiceOver'
4. Accessibility shortcut (triple-click side button)

## 12.2 TalkBack on Android

TalkBack is Google's screen reader for Android devices.

TalkBack Gestures:

- **Swipe Right:** Move to next element
- **Swipe Left:** Move to previous element
- **Double Tap:** Activate selected element
- **Swipe Down then Up:** Read from top
- **Swipe Up then Down:** Read from current position
- **Swipe Right then Left:** Go back
- **Swipe Up then Right:** Open TalkBack menu
- **Swipe Down then Left:** Open global context menu
- **L-Shaped Gesture:** Various shortcuts

TalkBack Reading Controls:

- **Swipe up/down with two fingers:** Scroll
- **Swipe left/right with two fingers:** Change reading granularity
- **Reading granularity:** Characters → Words → Lines → Paragraphs → Pages

Enabling TalkBack:

1. Settings → Accessibility → TalkBack → Toggle On
2. Volume up + Volume down (hold for 3 seconds)
3. Say 'Hey Google, turn on TalkBack'

## 12.3 Platform Differences

While React Native provides cross-platform APIs, VoiceOver and TalkBack have different behaviors:

**Announcement Style:** VoiceOver is more verbose by default, TalkBack is more concise

**Role Announcement:** Different pronunciation ('button' vs 'btn')

**Gesture Sets:** Different gestures for same actions

**Navigation Modes:** Rotor (iOS) vs Reading Controls (Android)

**Focus Order:** May differ slightly between platforms

**Custom Actions:** Accessed differently on each platform

Testing on Both Platforms is Essential: Always test your app with both VoiceOver and TalkBack to ensure consistent experience.

# Chapter 13: Touch Targets and Gestures

Mobile interaction requires careful consideration of touch target sizes and gesture patterns to accommodate users with motor disabilities.

## 13.1 Minimum Touch Target Sizes

Touch targets must be large enough for users with limited dexterity:

- **WCAG 2.1:** Minimum 44x44 points (iOS) or 48x48 dp (Android)
- **Apple HIG:** Recommends 44x44 points minimum
- **Material Design:** Recommends 48x48 dp minimum
- **Best Practice:** Use 48x48 dp minimum for all interactive elements

## 13.2 Using `hitSlop` and `pressRetentionOffset`

Interactive elements in React Native — especially icons, small buttons, or close icons — can be hard to tap. To make them easier and more accessible, React Native provides two important props: `hitSlop` and `pressRetentionOffset`.

### **`hitSlop`**

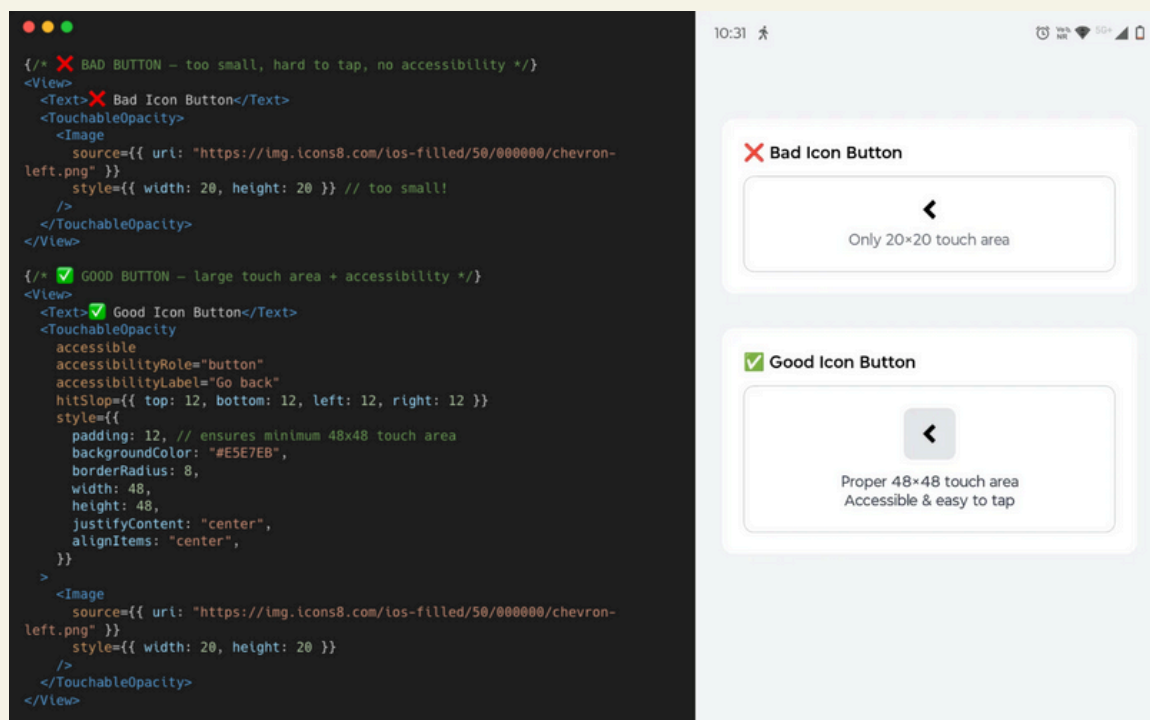
`hitSlop` increases the touchable area around a component without changing its visual size. This is extremely useful for small icons, like back buttons, delete icons, or close (“X”) buttons.

### **`pressRetentionOffset`**

`pressRetentionOffset` defines how far the user’s finger can drift outside the button before the press is cancelled.

This ensures the touch remains active even if the user slightly moves their finger — important for small controls. for small icons, like back buttons, delete icons, or close (“X”) buttons.

# Example



## 13.3 Gesture Conflicts with Screen Readers

Custom gestures can conflict with screen reader gestures. Provide alternative ways to access functionality:

- **Swipe Gestures:** Conflict with screen reader navigation
- **Pinch/Zoom:** May be used by screen magnification
- **Multi-Touch:** Screen reader users use different gestures
- **Solution:** Provide alternative controls (buttons, menus)

# Chapter 14: Forms and Input in React Native

Forms are a core part of most apps. To be accessible, they must include proper labels, clear roles, accurate states, and meaningful feedback — especially for screen reader users.

## 14.1 Accessible Text Inputs

Text inputs must always:

- Have clear, descriptive labels
- Use `accessibilityLabel` if visual labels are not programmatically connected
- Provide hints for what the user should enter
- Announce errors and success states
- Ensure the focus moves to the input when needed

## 14.2 Checkboxes and Radio Buttons

React Native does not provide built-in checkbox or radio components, so custom implementations must expose the correct roles, states, and labels for assistive technologies. Without these, screen readers cannot interpret whether an option is selected or not.

Key Requirements

- Use `accessibilityRole="checkbox"` or `"radio"`.
- Provide `accessibilityState={{ checked: true/false }}` for checkboxes or `accessibilityState={{ selected: true/false }}` for radio buttons
- Wrap each option in a touch target of at least  $48 \times 48$
- Ensure the label clearly describes the choice
- For radio buttons: Group them inside a container with `accessibilityRole="radiogroup"`

## 14.3 Form Validation and Error Messages

Form validation must be accessible so every user — including screen reader users — can understand what went wrong and how to fix it. Errors should be clear, announced, and linked to the relevant fields.

## Key Requirements

- Display error messages close to the input
- Give errors an alert role so screen readers announce them
- Move focus to the first invalid field
- Announce validation results such as “Email required” or “Invalid password”
- Avoid showing only color-based errors (e.g., red border)

```
import React, { useState, useRef, useEffect } from "react";
import {
  View, Text, TextInput, TouchableOpacity,
  AccessibilityInfo, findNodeHandle
} from "react-native";

export default function AccessibleForm() {
  const [name, setName] = useState("");
  const [gender, setGender] = useState("");
  const [terms, setTerms] = useState(false);
  const [error, setError] = useState("");
  const errorRef = useRef(null);

  const handleSubmit = () => {
    if (!name) return setError("Name is required");
    if (!gender) return setError("Select a gender");
    if (!terms) return setError("Accept terms to continue");
    setError("");
    AccessibilityInfo.announceForAccessibility("Form submitted");
  };

  useEffect(() => {
    if (error && errorRef.current) {
      const tag = findNodeHandle(errorRef.current);
      AccessibilityInfo.setAccessibilityFocus(tag);
    }
  }, [error]);

  return (
    <View style={{ padding: 20, gap: 12 }}>
      <Text Input */>
      <TextInput
        value={name}
        onChangeText={setName}
        accessibilityLabel="Name"
        accessibilityHint="Enter your name"
        style={{ borderWidth: 1, padding: 10 }}
      />

      <Radio Group */>
      <View accessibilityRole="radiogroup">
        [{"Male", "Female"].map(option) => (
          <TouchableOpacity
            key={option}
            accessibilityRole="radio"
            accessibilityState={{ selected: gender === option }}
            onPress={() => setGender(option)}
          >
            <Text>{gender === option ? "•" : "○"} {option}</Text>
          </TouchableOpacity>
        )]}
      </View>

      <Checkbox */>
      <TouchableOpacity
        accessibilityRole="checkbox"
        accessibilityState={{ checked: terms }}
        onPress={() => setTerms(!terms)}
      >
        <Text>{terms ? "☑" : "☐"} Accept terms</Text>
      </TouchableOpacity>

      <Error */>
      {error && (
        <Text
          ref={errorRef}
          accessible
          accessibilityRole="alert"
          style={{ color: "red" }}
        >
          {error}
        </Text>
      )}

      <TouchableOpacity
        accessibilityRole="button"
        onPress={handleSubmit}
        style={{ padding: 12, backgroundColor: "#2563EB" }}
      >
        <Text style={{ color: "white" }}>Submit</Text>
      </TouchableOpacity>
    </View>
  );
}
```

# Chapter 15: Focus Management and Navigation

Proper focus management ensures screen reader users can navigate your app efficiently and understand where they are.

## 15.1 Managing Focus Programmatically

In dynamic interfaces, content can appear, update, or move based on user actions. Screen reader users may lose their place if focus is not handled properly. Managing focus programmatically ensures that assistive technology knows where the user should land next.

```
import React, { useRef, useState, useEffect } from 'react';
import { View, Text, TextInput, Button } from 'react-native';

export default function ManageFocusExample() {
  const nameRef = useRef(null);
  const emailRef = useRef(null);
  const [submitted, setSubmitted] = useState(false);

  useEffect(() => {
    // After submission, focus on the next important element
    if (submitted) {
      emailRef.current?.focus();
    }
  }, [submitted]);

  return (
    <View style={{ padding: 16 }}>
      <Text>Name</Text>
      <TextInput
        ref={nameRef}
        placeholder="Enter your name"
        accessibilityLabel="Name input"
        style={{ borderWidth: 1, marginBottom: 10, padding: 8 }}
      />
      <Text>Email</Text>
      <TextInput
        ref={emailRef}
        placeholder="Enter your email"
        accessibilityLabel="Email input"
        style={{ borderWidth: 1, marginBottom: 10, padding: 8 }}
      />
      <Button title="Next" onPress={() => setSubmitted(true)} />
    </View>
  );
}
```

## 15.2 Screen Reader Announcements

When your app updates content dynamically — loading new data, showing an error, submitting a form, or changing a section — screen reader users may not know something changed unless it's announced.



Use announcements for:

- Status updates (loading, success, failure)
- Form errors
- Filter or search results
- Background events (e.g., “Download completed”)
- Navigation changes that don’t move focus automatically

Announcements should be:

- Short
- Clear
- Relevant
- Triggered only when necessary (to avoid noise)

Use AccessibilityInfo API to make dynamic announcements:

```
import React, { useState } from 'react';
import { View, Text, Button, AccessibilityInfo } from 'react-native';

export default function ScreenReaderAnnouncement() {
  const [count, setCount] = useState(0);

  const increment = () => {
    const newCount = count + 1;
    setCount(newCount);
    AccessibilityInfo.announceForAccessibility(`Count updated to ${newCount}`);
  };

  return (
    <View style={{ padding: 16 }}>
      <Text accessibilityLiveRegion="polite">Count: {count}</Text>
      <Button title="Increase Count" onPress={increment} />
    </View>
  );
}
```

# Chapter 16: Lists, Navigation, and Complex Components

Lists, tab bars, and other interactive UI patterns require special accessibility considerations in React Native. These components often contain multiple focusable elements, dynamically updated content, or groupings that a screen reader must interpret correctly.

## 16.1 Accessible FlatList and SectionList

Lists need:

- Proper announcements when new items load, refresh, or change.
- Clear focus order so users move through items predictably.
- Accessibility roles to let screen readers identify list items.
- Fallback labels for images or icons inside list items.
- Keyboard navigation support (where applicable).



```
<FlatList
  data={items}
  keyExtractor={({item}) => item.id}
  accessibilityLabel="Product list"
  accessibilityRole="list"
  onEndReached={() => {
    AccessibilityInfo.announceForAccessibility("Loading more items");
  }}
  renderItem={({ item }) => (
    <View
      accessible
      accessibilityRole="button"
      accessibilityLabel={` ${item.name}, price ${item.price}`}
    >
      <Text>{item.name}</Text>
      <Text>{item.price}</Text>
    </View>
  )}
/>
```

## 16.2 Tab Navigation

Tab bars require:

- Tab roles for each tab.
- State announcements such as “selected” or “not selected.”
- Clear focus movement when a tab is selected.
- No decorative-only elements receiving focus.

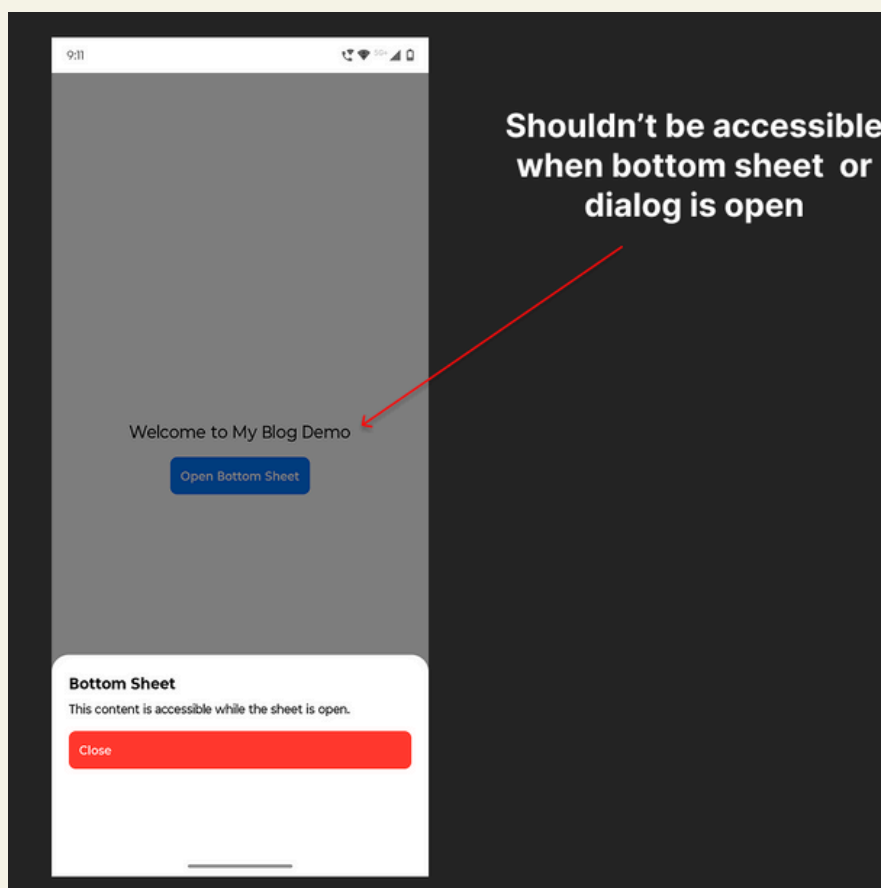
```

<View accessibilityRole="tablist">
  {tabs.map((tab, index) => (
    <TouchableOpacity
      key={tab.key}
      accessibilityRole="tab"
      accessibilityState={{ selected: index === activeTab }}
      accessibilityLabel={tab.label}
      onPress={() => setActiveTab(index)}
    >
      <Text>{tab.label}</Text>
    </TouchableOpacity>
  ))}
</View>

```

## 16.3 Accessible Modals and Bottom Sheets

On mobile, modals and bottom sheets should clearly announce themselves to screen readers and temporarily take over the accessibility focus. Users should not be able to navigate to elements behind them while they are open. All controls inside the modal or sheet must be properly labeled and reachable through swipe navigation. When the modal or sheet is dismissed, focus should return to a logical point, typically the element or action that opened it, so the experience feels smooth and predictable for assistive technology users.



# Chapter 17: Testing React Native Accessibility

Testing is crucial for ensuring your React Native app is accessible. Use a combination of automated tools, manual testing with screen readers, and real user testing.

## 17.1 Automated Testing

Integrate automated accessibility testing into your development workflow:

CI/CD Integration: Run accessibility tests automatically on every commit:

## 17.2 Manual Testing Checklist

### iOS VoiceOver Testing:

- Turn on VoiceOver (Settings → Accessibility → VoiceOver)
- Navigate through entire app using swipe gestures
- Verify all interactive elements are announced correctly
- Test all user flows (login, checkout, etc.)
- Check that images have appropriate labels
- Verify form labels and error messages are announced
- Test modals and dialogs trap focus correctly
- Verify rotor navigation works (headings, links, buttons)
- Test landscape and portrait orientations
- Verify announcements for dynamic content

### Android TalkBack Testing:

- Turn on TalkBack (Settings → Accessibility → TalkBack)
- Navigate using swipe gestures
- Test reading controls (swipe up/down to change granularity)
- Verify custom actions work correctly
- Test with different Android versions (behavior varies)
- Check that all content is accessible
- Verify proper heading hierarchy
- Test form validation and error announcements

**Additional Manual Tests:**

- Test with large text sizes (Settings → Display → Text Size)
- Test with reduced motion enabled
- Test with color blindness simulators
- Verify touch targets are at least 48x48 dp
- Test with switch control (iOS) / switch access (Android)
- Verify color contrast meets WCAG AA standards

## 17.3 Using Accessibility Inspector Tools

**iOS Accessibility Inspector:**

- Built into Xcode (Xcode → Open Developer Tool → Accessibility Inspector)
- Inspect accessibility properties of UI elements
- Run audit to find common issues
- Test VoiceOver without enabling it system-wide
- Simulate different text sizes

**Android Accessibility Scanner:**

- Download from Google Play Store
- Scans screens for accessibility issues
- Provides suggestions for improvements
- Checks touch target sizes, contrast, labels

# Chapter 18: Best Practices and Common Patterns

This chapter covers best practices, common patterns, and real-world examples for React Native accessibility.

## Accessibility Best Practices Check

Essential Best Practices:

- ✓ Use appropriate `accessibilityRole` for all interactive elements
- ✓ Provide descriptive `accessibilityLabel` for all buttons and images
- ✓ Add `accessibilityHint` when action result isn't obvious
- ✓ Use `accessibilityState` for dynamic states (checked, selected, disabled)
- ✓ Ensure minimum 48x48 dp touch targets
- ✓ Use `accessible={true}` to group related elements
- ✓ Set `accessible={false}` for decorative elements
- ✓ Manage focus when content changes (modals, navigation)
- ✓ Announce important changes with `AccessibilityInfo.announceForAccessibility()`
- ✓ Test with both VoiceOver and TalkBack
- ✓ Use `accessibilityViewIsModal` for modal dialogs
- ✓ Provide custom actions for swipeable components
- ✓ Support text scaling (don't use fixed pixel sizes)
- ✓ Maintain proper heading hierarchy
- ✓ Hide decorative images from screen readers

# Few Descriptive examples

## Decorative vs Informative vs Incorrect

These are UI elements that convey no meaning and are not necessary for a screen reader user.

1. Background images
2. Icons used only for visual styling
3. Separator lines
4. Purely aesthetic illustrations

```
<View style={styles.container}>
  <Text style={styles.heading}>Decorative vs Informative vs Incorrect</Text>

  { /* ✅ Decorative: Icon with Text */ }
  <View style={styles.section}>
    <Text style={styles.subheading}>👉 Decorative Icon (with text)</Text>

    <TouchableOpacity
      style={styles.backButton}
      accessibilityRole="button"
      accessibilityLabel="Go back"
      onPress={() => handlePress("Go Back")}
    >
      { /* Icon hidden from screen readers */ }
      <Icons
        name="arrow-back"
        size={24}
        color="white"
        accessible={false}
        importantForAccessibility="no"
      />
      <Text style={styles.buttonText}>Go Back</Text>
    </TouchableOpacity>

    <Text style={styles.note}>
      ✅ Screen reader announces only: "Go back, button."
      The arrow icon is decorative and skipped.
    </Text>
  </View>

  { /* ✅ Informative: Icon Alone (properly labeled) */ }
  <View style={styles.section}>
    <Text style={styles.subheading}>👉 Informative Icon (standalone)</Text>

    <TouchableOpacity
      style={styles.iconButton}
      accessibilityRole="button"
      accessibilityLabel="Go back"
      onPress={() => handlePress("Go Back")}
    >
      <Icons name="arrow-back" size={24} color="white" />
    </TouchableOpacity>

    <Text style={styles.note}>
      ✅ Screen reader announces: "Go back, button."
      Icon has a label because there's no text.
    </Text>
  </View>

  { /* ❌ Incorrect: Icon Alone (no label) */ }
  <View style={styles.section}>
    <Text style={styles.subheading}>👉 Incorrect (missing label)</Text>

    <TouchableOpacity
      style={styles.iconButton, { backgroundColor: "#E53935" }}
      onPress={() => handlePress("No Label")}
    >
      <Icons name="arrow-back" size={24} color="white" />
    </TouchableOpacity>

    <Text style={styles.note, { color: "#C62828" }}>
      ❌ Screen reader says nothing or just "button" —
      It has no label, so users won't know its purpose.
    </Text>
  </View>
</View>
```

9:27

### Decorative vs Informative vs Incorrect

👉 Decorative Icon (with text)

← Go Back

✅ Screen reader announces only: "Go back, button." The arrow icon is decorative and skipped.

👉 Informative Icon (standalone)

←

✅ Screen reader announces: "Go back, button." Icon has a label because there's no text.

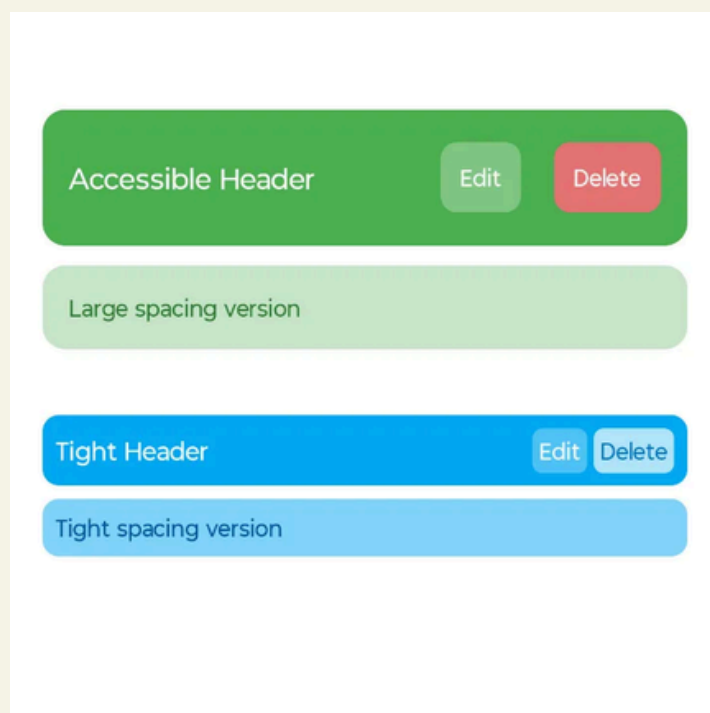
❌ Incorrect (missing label)

←

❌ Screen reader says nothing or just "button" — It has no label, so users won't know its purpose.

# Accessible Interaction Design: Clear vs Confined Touch Areas

- Interactive elements should provide comfortable touch targets with enough separation between controls.
- When spacing is reduced, users with motor impairments, larger fingers, or small screens face difficulty interacting reliably.



## Ensuring Clear Visibility Through Proper Color Contrast

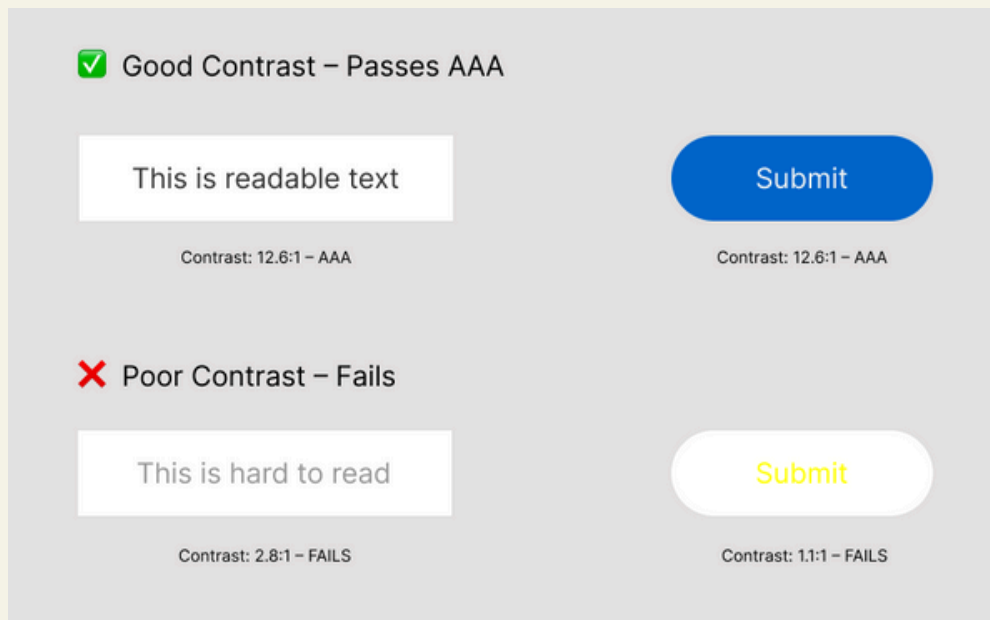
Color contrast is essential for making interactive elements easy to see, understand, and use.

High-contrast combinations ensure text remains legible and controls stand out across different lighting conditions.

- According to WCAG guidelines, normal text should meet at least a **4.5:1** contrast ratio, while large text and bold text require a minimum of **3:1**.
- For interactive components like buttons or icons, maintaining a contrast ratio of **3:1** against their background helps users clearly identify actions.



- Poor contrast blurs visual boundaries, increases eye strain, and creates barriers for users with low vision or color perception differences.
- Choosing strong, accessible color pairs ensures that interactions remain confident, visible, and inclusive for all users.



## Grouping Content Into a Single Accessible Element

Sometimes a UI component contains multiple child elements (image, text, icons), but exposing every piece individually to a screen reader can create noise, cause double announcements, or break the intended meaning.

In these cases, you should treat the entire card as one accessible unit and hide all of its children from screen readers.

This improves clarity, reduces verbosity, and makes the UI easier to navigate for screen reader users.

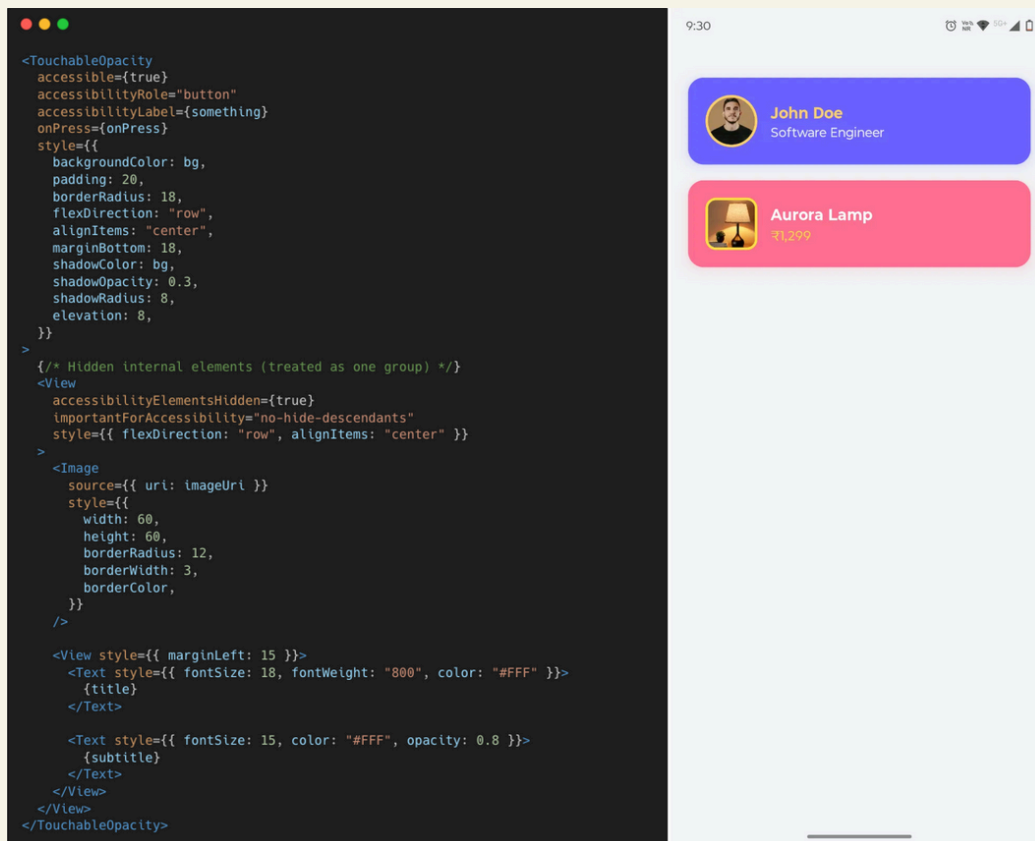
### When to Group?

Use grouping when:

- The card represents one actionable item
- The text, image, and icons together form one meaning
- Child items are not useful on their own

## Examples:

- Product card
- Profile card
- Notification card
- List item with image + title + subtitle



To treat an entire container as one accessible element, you make the parent view accessible and hide all child elements from the accessibility tree. In React Native, this is done by setting **accessible={true}** on the parent and adding **accessibilityElementsHidden={true}** with **importantForAccessibility="no-hide-descendants"** on the inner wrapper.

The screen reader will ignore individual icons, images, and text inside the card and announce only the parent's accessibilityLabel and accessibilityHint. This ensures the whole card behaves like a single, clean, well-described component instead of a clutter of multiple focusable items.

# Accessibility Resources

Resources for Continued Learning:

- **React Native Docs:** <https://reactnative.dev/docs/accessibility>
- **Apple Accessibility:** <https://developer.apple.com/accessibility/>
- **Android Accessibility:** <https://developer.android.com/guide/topics/ui/accessibility>
- **WCAG Mobile:** <https://www.w3.org/TR/mobile-accessibility-mapping/>
- **A11y Project:** <https://www.a11yproject.com/>



# Thank You

We are truly fortunate to have the access, tools, and resources that enable us to build great experiences.

If we can use that privilege to make these same experiences accessible to people with diverse abilities, we move one step closer to a world that is not only more inclusive, but genuinely equal. Thank you for taking the time to learn, improve, and contribute to creating a better, more accessible future for everyone.

We would greatly appreciate your thoughts — please feel free to share your feedback!

---

## Contact Information

**Name** Mohammed Abdullah Khan  
**Email** [mohammedabdullahkhan26523@gmail.com](mailto:mohammedabdullahkhan26523@gmail.com)  
**LinkedIn** <https://www.linkedin.com/in/mohammed-abdullah-khan-7b82a31a5/>  
**Website** <https://www.mohammedabdullahkhan.com/>

**Name** Monika Hamand  
**Email** [monikahamand237@gmail.com](mailto:monikahamand237@gmail.com)  
**LinkedIn** <https://www.linkedin.com/in/monikahamand/>