Accessibility

Edit on GitHub

Why Accessibility?

Web accessibility (also referred to as **a11y**) is the design and creation of websites that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages.

React fully supports building accessible websites, often by using standard HTML techniques.

Standards and Guidelines

WCAG

The Web Content Accessibility Guidelines provides guidelines for creating accessible web sites.

The following WCAG checklists provide an overview:

- WCAG checklist from Wuhcag
- WCAG checklist from WebAIM
- Checklist from The A11Y Project

WAI-ARIA

The Web Accessibility Initiative - Accessible Rich Internet Applications document contains techniques for building fully accessible JavaScript widgets.

Note that all `aria-*` HTML attributes are fully supported in JSX. Whereas most DOM properties and attributes in React are camelCased, these attributes should be lowercased:

Code

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

Accessible Forms

Labeling

Every HTML form control, such as `<input>` and `<textarea>`, needs to be labeled accessibly. We need to provide descriptive labels that are also exposed to screen readers.

The following resources show us how to do this:

- The W3C shows us how to label elements
- WebAIM shows us how to label elements
- The Paciello Group explains accessible names

Although these standard HTML practices can be directly used in React, note that the `for` attribute is written as `htmlFor` in JSX:

Code

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

Notifying the user of errors

Error situations need to be understood by all users. The following link shows us how to expose error texts to screen readers as well:

- The W3C demonstrates user notifications
- WebAIM looks at form validation

Focus Control

Ensure that your web application can be fully operated with the keyboard only:

- WebAIM talks about keyboard accessibility

Keyboard focus and focus outline

Keyboard focus refers to the current element in the DOM that is selected to accept input from the keyboard. We see it everywhere as a focus outline similar to that shown in the following image:

Only ever use CSS that removes this outline, for example by setting `outline: 0`, if you are replacing it with another focus outline implementation.

Mechanisms to skip to desired content

Provide a mechanism to allow users to skip past navigation sections in your application as this assists and speeds up keyboard navigation.

Skiplinks or Skip Navigation Links are hidden navigation links that only become visible when keyboard users interact with the page. They are very easy to implement with internal page anchors and some styling:

- WebAIM - Skip Navigation Links

Also use landmark elements and roles, such as `<main>` and `<aside>`, to demarcate page regions as assistive technology allow the user to quickly navigate to these sections.

Read more about the use of these elements to enhance accessibility here:

- Deque University - HTML 5 and ARIA Landmarks

Programmatically managing focus

Our React applications continuously modify the HTML DOM during runtime, sometimes leading to keyboard focus being lost or set to an unexpected element. In order to repair this, we need to programmatically nudge the keyboard focus in the right direction. For example, by resetting keyboard focus to a button that opened a modal window after that modal window is closed.

The Mozilla Developer Network takes a look at this and describes how we can build keyboard-navigable JavaScript widgets.

To set focus in React, we can use Refs to Components.

Using this, we first create a ref to an element in the JSX of a component class:

Code

```
render() {
  // Use the `ref` callback to store a reference to the text input DOM
  // element in an instance field (for example, this.textInput).
  return (
    <input
      type="text"
      ref={(input) => { this.textInput = input; }} />
  );
}
```

Then we can focus it elsewhere in our component when needed:

Code

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  this.textInput.focus();
}
```

A great focus management example is the react-aria-modal. This is a relatively rare example of a fully accessible modal window. Not only does it set initial focus on the cancel button (preventing the

keyboard user from accidentally activating the success action) and trap keyboard focus inside the modal, it also resets focus back to the element that initially triggered the modal.

> **Note:**
>
> While this is a very important accessibility feature, it is also a technique that should be used judiciously. Use it to repair the keyboard focus flow when it is disturbed, not to try and anticipate how users want to use applications.

More Complex Widgets

A more complex user experience should not mean a less accessible one. Whereas accessibility is most easily achieved by coding as close to HTML as possible, even the most complex widget can be coded accessibly.

Here we require knowledge of ARIA Roles as well as ARIA States and Properties. These are toolboxes filled with HTML attributes that are fully supported in JSX and enable us to construct fully accessible, highly functional React components.

Each type of widget has a specific design pattern and is expected to function in a certain way by users and user agents alike:

- WAI-ARIA Authoring Practices - Design Patterns and Widgets
- Heydon Pickering - ARIA Examples
- Inclusive Components

Other Points for Consideration

Setting the language

Indicate the human language of page texts as screen reader software uses this to select the correct voice settings:

- WebAIM - Document Language

Setting the document title

Set the document `<title>` to correctly describe the current page content as this ensures that the user remains aware of the current page context:

- WCAG - Understanding the Document Title Requirement

We can set this in React using the React Document Title Component.

Color contrast

Ensure that all readable text on your website has sufficient color contrast to remain maximally readable by users with low vision:

- WCAG - Understanding the Color Contrast Requirement
- Everything About Color Contrast And Why You Should Rethink It
- A11yProject - What is Color Contrast

It can be tedious to manually calculate the proper color combinations for all cases in your website so instead, you can calculate an entire accessible color palette with Colorable.

Both the aXe and WAVE tools mentioned below also include color contrast tests and will report on contrast errors.

If you want to extend your contrast testing abilities you can use these tools:

- WebAIM - Color Contrast Checker
- The Paciello Group - Color Contrast Analyzer

Development and Testing Tools

There are a number of tools we can use to assist in the creation of accessible web applications.

The keyboard

By far the easiest and also one of the most important checks is to test if your entire website can be reached and used with the keyboard alone. Do this by:

1. Plugging out your mouse.
2. Using `Tab` and `Shift+Tab` to browse.
3. Using `Enter` to activate elements.
4. Where required, using your keyboard arrow keys to interact with some elements, such as menus and dropdowns.

Development assistance

We can check some accessibility features directly in our JSX code. Often intellisense checks are already provided in JSX aware IDE's for the ARIA roles, states and properties. We also have access to the following tool:

eslint-plugin-jsx-a11y

The eslint-plugin-jsx-a11y plugin for ESLint provides AST linting feedback regarding accessibility issues in your JSX. Many IDE's allow you to integrate these findings directly into code analysis and source code windows.

Create React App has this plugin with a subset of rules activated. If you want to enable even more accessibility rules, you can create an `.eslintrc` file in the root of your project with this content:

Code

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

Testing accessibility in the browser

A number of tools exist that can run accessibility audits on web pages in your browser. Please use them in combination with other accessibility checks mentioned here as they can only test the technical accessibility of your HTML.

aXe, aXe-core and react-axe

Deque Systems offers aXe-core for automated and end-to-end accessibility tests of your applications. This module includes integrations for Selenium.

The Accessibility Engine or aXe, is an accessibility inspector browser extension built on `aXe-core`.

You can also use the react-axe module to report these accessibility findings directly to the console while developing and debugging.

WebAIM WAVE

The Web Accessibility Evaluation Tool is another accessibility browser extension.

Accessibility inspectors and the Accessibility Tree

The Accessibility Tree is a subset of the DOM tree that contains accessible objects for every DOM element that should be exposed to assistive technology, such as screen readers.

In some browsers we can easily view the accessibility information for each element in the accessibility tree:

- Activate the Accessibility Inspector in Chrome
- Using the Accessibility Inspector in OS X Safari

Screen readers

Testing with a screen reader should form part of your accessibility tests.

Please note that browser / screen reader combinations matter. It is recommended that you test your application in the browser best suited to your screen reader of choice.

NVDA in FireFox

NonVisual Desktop Access or NVDA is an open source Windows screen reader that is widely used.

Refer to the following guides on how to best use NVDA:

- WebAIM - Using NVDA to Evaluate Web Accessibility
- Deque - NVDA Keyboard Shortcuts

VoiceOver in Safari

VoiceOver is an integrated screen reader on Apple devices.

Refer to the following guides on how activate and use VoiceOver:

- WebAIM - Using VoiceOver to Evaluate Web Accessibility

- Deque - VoiceOver for OSX Keyboard Shortcuts
- Deque - VoiceOver for iOS Shortcuts

JAWS in Internet Explorer

Job Access With Speech or JAWS, is a prolifically used screen reader on Windows.

Refer to the following guides on how to best use JAWS:

- WebAIM - Using JAWS to Evaluate Web Accessibility
- Deque - JAWS Keyboard Shortcuts