

Accessibility tips for web developers



- [Are your UI components accessible?](#)
 - [Can your UI component be used with the keyboard alone?](#)
 - [Using tabindex](#)
 - [Using autofocus](#)
 - [Adding keyboard interaction](#)
 - [Can you use your UI component with a screen reader?](#)
 - [Do all components and images have meaningful text alternatives?](#)
 - [Do your components provide semantic information?](#)
 - [Can users understand everything without relying on color?](#)
 - [Is there sufficient contrast between text and images and their background?](#)
 - [Is the moving or flashing content in your components stoppable and safe?](#)
 - [Accessibility Tooling](#)
 - [Conclusions](#)
-

It's awesome to build sites that are inclusive and accessible to everyone. There are at least six key areas of disability you can optimize for: **visual**, **hearing**, **mobility**, **cognition**, **speech**, and **neural**. Many tools and resources can help here, even if you're totally new to web accessibility.

Over one billion people live with some form of disability. You might have been in a loud room at some point trying to hear the conversation around you or in a low-lighting condition trying to find something in the dark. Do you remember the frustration you felt with that circumstance? Now imagine if that temporary condition were permanent. How different would your experience on the web be?

To be accessible, sites need to work across multiple devices with varying screen sizes and different kinds of input, such as screen readers. Moreover, sites should be usable by the broadest group of users, including those with disabilities. Here's a sample of just a few disabilities your users may have:

Vision	Hearing	Mobility
<ul style="list-style-type: none"> • Low vision • Blindness • Color blindness • Cataracts • Sun glare 	<ul style="list-style-type: none"> • Hard of hearing • Deafness • Noise • Ear infection 	<ul style="list-style-type: none"> • Spinal cord injury • Limited dexterity • Hands full
Cognitive	Speech	Neural
<ul style="list-style-type: none"> • Learning disability • Sleepiness or distraction • Migraine • Autism • Seizure 	<ul style="list-style-type: none"> • Ambient noise • Sore throat • Speech impediment • Unable to speak 	<ul style="list-style-type: none"> • Depression • PTSD • Bipolar disorder • Anxiety

Visual issues range from an inability to distinguish colors to no vision at all.

- Ensure text content meets a minimum [contrast ratio threshold](#).
- Avoid communicating information [using solely color](#) and ensure that all text is [resizable](#).
- Ensure all user interface components can be used with assistive technologies such as screen readers, magnifiers, and braille displays. This entails ensuring that UI components are marked up such that accessibility APIs can programmatically determine the *role*, *state*, *value*, and *title* of any element.

Tip: The element inspection feature in the Chrome, Edge, and Firefox developer tools displays a tooltip of CSS properties that includes a quick check for color contrast ratio.

```
{% Img src="image/admin/avwZaOjEWT7pES1OqUly.jpg", alt="Screenshot of the Chrome DevTools Inspect Element tooltip.", width="800", height="370" %}
```

I personally live with low vision, and I often find myself zooming in on sites, their DevTools, and the terminal. While supporting zoom is almost never at the top of developers' to-do lists, it can make a world of difference to users like me.□□

Hearing issues mean a user may have issues hearing sound emitted from a page.

- Provide [text alternatives](#) for all content that is not strictly text.
- Test that your UI components are still functional [without sound](#).

```
{% Img src="image/admin/ZIbJHF5vyMxmKAN9juO1.jpg", alt="Screenshot of the ChromeVox screen reader reading a web page.", width="800", height="500" %}
```

Mobility issues can include the inability to operate a mouse, a keyboard, or a touch screen.

- Make the content of your UI components [functionally accessible from a keyboard](#) for any actions one would otherwise use a mouse for.
- Ensure pages are correctly marked up for assistive technologies—including screen readers, voice control software, and physical switch controls—which tend to use the same APIs.

Cognitive issues mean a user may require assistive technologies to help them with reading text, so it's important to ensure text alternatives exist.

- Be mindful when using animations. Avoid video and animation that [repeat](#) or flash, which can cause [issues](#) for some users.

The [prefers-reduced-motion](#) CSS media query allows you to limit animations and autoplaying videos for users who prefer reduced motion:

```
/*  
If the user expresses a preference for reduced motion, don't use animations on  
buttons.  
*/  
@media (prefers-reduced-motion: reduce) {  
  button {  
    animation: none;  
  }  
}
```

- Avoid interactions that are [timing-based](#).

This may seem like a lot of bases to cover, but we'll walk through the process for assessing and then improving the accessibility of your UI components.

Tip: The GOV.UK accessibility team has made some great [accessibility dos and don'ts digital posters](#) to spread awareness of best practices in your team:

```
{% Img src="image/admin/lmcS2dTmBHUo3j0ERCp.jpg", alt="Digital posters showing  
accessibility dos and don'ts.", width="800", height="721" %}
```

Are your UI components accessible?

Summary

When auditing your page's UI components for accessibility, ask yourself:

- **Can you use your UI component with the keyboard only?**

Does the component manage focus and avoid focus traps? Can it respond to the appropriate keyboard events?

- **Can you use your UI component with a screen reader?**

Have you provided text alternatives for any information presented visually? Have you added semantic information using ARIA?

- **Can your UI component work without sound?**

Turn off your speakers and go through your use cases.

- **Can your UI component work without color?**

Ensure your UI component can be used by someone who cannot see colors. A helpful tool for simulating color blindness is a Chrome extension called [SEE](#). (Try all four forms of color blindness simulation available.) You may also be interested in the [Daltonize](#) extension, which is similarly useful.

- **Can your UI component work with high-contrast mode enabled?**

All modern operating systems support a high contrast mode. [High Contrast](#) is a Chrome extension that can help here.

Standardized controls (such as `<button>` and `<select>`) have accessibility built in by the browser. They are focusable using the `Tab` key; they respond to keyboard events (like the `Enter`, `Space`, and arrow keys); and they have semantic roles, states, and properties used by accessibility tools. Their default styling should also meet the accessibility requirements listed above.

Custom UI components (with the exception of components that extend standard elements like `<button>`) do not have any built-in functionality, including accessibility, so you need to provide it. A good place to start when implementing accessibility is to compare your component to an analogous standard element (or a combination of several standard elements, depending on how complex your component is).

Tip: Most browser developer tools support inspecting the accessibility tree of a page. In Chrome DevTools, this is available via the **Accessibility** tab in the **Elements** panel:

```
{% Img src="image/admin/FcpD3oNA1C39KbQuNinf.png", alt="Screenshot of the accessibility tree view in Chrome DevTools.", width="800", height="486" %}
```

Firefox also has an **Accessibility** panel:

```
{% Img src="image/admin/DRqDC9mtfcejAljLlmdR.jpg", alt="Screenshot of the accessibility tree view in FireFox DevTools.", width="800", height="500" %}
```

Safari exposes accessibility information in the **Elements** panel's **Node** tab.

The following is a list of questions you can ask yourself when attempting to make your UI components more accessible.

Can your UI component be used with the keyboard alone?

Ideally, ensure that all functionality in your UI component can be accessed via keyboard. When designing your user experience, think about how you would use your element with the keyboard alone and figure out a consistent set of keyboard interactions.

First, ensure that you have a sensible focus target for each component. For example, a complex component like a menu may be one focus target within a page but should then manage focus within itself so that the active menu item always takes focus.

```
{% Img src="image/admin/8zpjMaaY0DuAHdugbSr3.png", alt="A screenshot of a menu and submenu that requires focus management.", width="800", height="377" %}
```

Managing focus within a complex element.

Using tabindex

The `tabindex` attribute allows elements and UI components to be focused using the keyboard. Keyboard-only and assistive technology users both need to be able to place keyboard focus on elements to interact with them. Built-in interactive elements (like `<button>`) are implicitly focusable, so they don't need a `tabindex` attribute unless you wish to change their position in the tab order.

There are three types of `tabindex` values:

- `tabindex="0"` is the most common and places the element in the natural tab order (defined by the DOM order).
- a `tabindex` value greater than 0 places the element in a manual tab order—all elements in the page with a positive `tabindex` value are visited in numerical order before elements in the natural tab order.
- a `tabindex` value equal to -1 causes the element to be programmatically focusable, but not in the tab order.

For custom UI components, always use `tabindex` values of 0 or -1, as you won't be able to determine the order of elements on a given page ahead of time—and even if you did, the order may change. A `tabindex` value of -1 is particularly useful for managing focus within complex components as described above.

Also ensure that focus is always visible, whether by allowing the default focus ring style or by applying a discernible custom focus style. Remember not to trap keyboard users—they should be able to move focus away from an element using only the keyboard.

{% Aside %} You may also be interested in the roving `tabindex` or `aria-activedescendant` approaches, [covered over on MDN](#). {% endAside %}

Using autofocus

The HTML autofocus attribute allows an author to specify that a particular element should automatically take focus when the page is loaded. `autofocus` is already supported on [all web form controls](#), including buttons. To autofocus elements in your own custom UI components, call the `focus()` method, supported on all HTML elements that can be focused (for example, `document.querySelector('myButton').focus()`).

Adding keyboard interaction

Once your UI component is focusable, provide a good keyboard interaction story when a component is focused by handling appropriate keyboard events. For example, allow the user to use arrow keys to select menu options and `Space` or `Enter` to activate buttons. The ARIA [design patterns guide](#) provides some guidance here.

Finally, ensure that your keyboard shortcuts are discoverable. A common practice is to have a keyboard shortcut legend (on-screen text) to inform the user that shortcuts exist. For example, "Press ? for keyboard shortcuts." Alternatively, a hint such a tooltip can be used to inform the user about a shortcut.

The importance of managing focus cannot be overstated. One example is a navigation drawer. If you add a UI component to the page, you need to direct focus to an element inside of it; otherwise, users may have to tab through the entire page to get there. This can be a frustrating experience, so be sure to test focus for all keyboard navigable components in your page.

Tip: You can use [Puppeteer](#) to automate running keyboard accessibility tests for toggling UI states. [WalkMe Engineering](#) has a great guide on this I recommend reading.

```
{% Img src="image/admin/3lZosaL1YXafLn4ZRINl.gif", alt="WalkMe state toggle test.", width="441", height="400" %}
```

```
// Example for expanding and collapsing a category with the Space key
const category = await page.$(`.category`);

// verify tabIndex, role and focus
expect(await page.evaluate(elem => elem.getAttribute(`role`), category)).toEqual(`button`);
expect(await page.evaluate(elem => elem.getAttribute(`tabindex`), category)).toEqual(`0`);
```

```
expect(await page.evaluate(elem => window.document.activeElement === elem,
category)).toEqual(true);

// verify aria-expanded = false
expect(await page.evaluate(elem => elem.getAttribute(`aria-expanded`),
category)).toEqual(`false`);

// toggle category by pressing Space
await page.keyboard.press('Space');

// verify aria-expanded = true
expect(await page.evaluate(elem => elem.getAttribute(`aria-expanded`),
category)).toEqual(`true`);
```

Can you use your UI component with a screen reader?

Around 1-2% of people use a screen reader. Can you understand all important information and interact with the component using the screen reader and keyboard alone?

The following questions should help you address screen reader accessibility.

Do all components and images have meaningful text alternatives?

Wherever information about the *name* or *purpose* of an interactive component is conveyed visually, provide an accessible text alternative.

For example, if your `<fancy-menu>` UI component only displays a gear icon to indicate that it's a settings menu, it needs an accessible text alternative, such as "settings," that conveys the same information. Depending on context, you can provide a text alternative using an `alt` attribute, an `aria-label` attribute, an `aria-labelledby` attribute, or plain text in the Shadow DOM. You can find general technical tips in [WebAIM Quick Reference](#).

Any UI component that displays an image should provide a mechanism for providing alternative text for that image, analogous to the `alt` attribute.

Do your components provide semantic information?

Assistive technology conveys semantic information that is otherwise expressed to sighted users via visual cues such as formatting, cursor style, or position. Standardized elements have this semantic information built-in by the browser, but for custom components you need to use [ARIA](#) to add the information.

As a rule of thumb, any component that listens to a mouse click or hover event should not only have some kind of keyboard event listener, but also an ARIA role and potentially ARIA states and attributes.

For example, a custom `<fancy-slider>` UI component might take an ARIA role of slider, which has some related ARIA attributes: `aria-valuenow`, `aria-valuemin` and `aria-valuemax`. By binding these attributes to the relevant properties on your custom component, you can allow users of assistive technology to interact with the element, change its value, and even cause the element's visual presentation to change accordingly.

```
{% Img src="image/admin/BD2bS6XEEUIORVOGHwhe.png", alt="A screenshot of a slider.", width="159", height="56" %}
```

A range slider component.

```
<fancy-slider role="slider" aria-valuemin="1" aria-valuemax="5" aria-valuenow="2.5"></fancy-slider>
```

Can users understand everything without relying on color?

Color shouldn't be used as the only means of conveying information, such as indicating a status, prompting the user for a response, or visualizing data. For example, if you have a pie chart, provide labels and values for each slice so users who have visual impairments can understand the information even if they can't tell where the slices begin and end:

```
{% Img src="image/admin/4DIbNfCX7AYYrPSpQg4L.png", alt="A pie chart with labels and values to ensure accessibility.", width="500", height="442" %}
```

An accessible pie chart. (From the [W3C Web Accessibility Initiative](#).)

Is there sufficient contrast between text and images and their background?

Any text content displayed in your component should meet the [minimum WCAG AA-level contrast threshold](#). Consider providing a high-contrast theme that meets the [higher AAA threshold](#), and ensure that user agent style sheets can be applied if users require custom contrast or different colors. You can use this [Color Contrast Checker](#) as an aid when designing your component.

Is the moving or flashing content in your components stoppable and safe?

Content that moves, scrolls, or blinks for more than five seconds should be able to be paused, stopped or hidden. In general, avoid flashing content. If something must flash, make sure it flashes no more than three times per second.

Accessibility Tooling

A number of tools are available that can assist with debugging the accessibility of your visual components.

- [Axe](#) provides automated accessibility testing for your framework or browser of choice. [Axe Puppeteer](#) can be used for writing automated accessibility tests.
- The [Lighthouse](#) Accessibility audits provide helpful insights for discovering common accessibility issues. The accessibility score is a weighted average of all accessibility audits based on [Axe user impact assessments](#). For monitoring accessibility via continuous integration, see [Lighthouse CI](#).

```
{% Img src="image/admin/Bft9DABqWge4ZRpF6wzK.png", alt="Screenshot of the Lighthouse accessibility audit.", width="800", height="649" %}
```

- [Tenon.io](#) is useful for testing common accessibility problems. Tenon has strong integration support across build tools, browsers (via extensions), and even text editors.
- There are many library- and framework-specific tools for highlighting accessibility issues with components. For example, [web.dev](#) explains how to use [eslint-plugin-jsx-a11y](#) to highlight accessibility issues for React components in your editor:

```
{% Img src="image/admin/9bAOYq7Cr11vzf4P04d2.png", alt="Screenshot of a code editor with an accessibility issue flagged by eslint-plugin-jsx-a11y.", width="800", height="500" %}
```

If you use Angular, [codelyzer](#) provides in-editor accessibility audits too:

```
{% Img src="image/admin/Ca3rCQlJ53C7lLx3hwot.png", alt="Screenshot of a code editor with an accessibility issue flagged by codelyzer.", width="800", height="433" %}
```

- You can examine the way that assistive technologies see web content by using [Accessibility Inspector](#) (Mac) or [Windows Automation API Testing Tools](#) and [AccProbe](#) (Windows). You can also see the full accessibility tree that Chrome creates by navigating to `about://accessibility`.
- The best way to test for screen reader support on a Mac is using the VoiceOver utility. Use `⌘F5` to enable or disable it, `Ctrl+Option ↔` to move through the page, and `Ctrl+Shift+Option + ↑↓` to move up and down the accessibility tree. For more detailed instructions, see the [full list of VoiceOver commands](#) and the [list of VoiceOver Web commands](#).
- [totally](#) is a useful visualizer for assistive technology issues built by Khan Academy. It's a script that adds a button to your document that triggers several plugins for annotating things like insufficient contrast ratio and other a11y violations.
- On Windows, [NVDA](#) is a free, open source screen reader that's fully featured and rapidly gaining in popularity. However, note that it has a much steeper learning curve for sighted users than VoiceOver.
- [ChromeLens](#) helps develop for the visually impaired. It also has great support for visualizing keyboard navigation paths.
{% Img src="image/admin/RcDZ95ALFvNeDdpdv2Mo.jpg", alt="Screenshot of ChromeLens.", width="800", height="514" %}
- [ChromeVox](#) is a screen reader available as a Chrome extension and built into ChromeOS devices.

Conclusions

We still have a long way to go to improve accessibility on the web. Per the [Web Almanac](#):

- 4 out of every 5 sites have text that easily blends into the background, making it unreadable.
- 49.91% of pages still fail to provide `alt` attributes for some of their images.
- Only 24% of pages that use buttons or links include textual labels for them.
- Only 22.33% of pages provide labels for all their form inputs.

To learn more about accessibility fundamentals and help improve those statistics, I recommend the [Accessible to all](#) docs on web.dev. There's much we can do to build experiences that are more

accessible for everyone.

{% Img src="image/admin/iMaXm08LsbVKpkNCZEwA.jpg", alt="Screenshot of the web.dev 'Accessible to all' collection page.", width="800", height="513" %}

📄 #2	
📄 📄	📄 2026-10-31 21:35:30 UTC
📄 📄	📄 2026-10-31 21:35:32 UTC