How To Do an Accessibility Review

By Rob Dodson
Rob is a contributor to Web**Fundamentals**

Determining if your web site or application is accessible can seem like an overwhelming task. If you are approaching accessibility for the first time, the sheer breadth of the topic can leave you wondering where to start - after all, working to accommodate a diverse range of abilities means there are a correspondingly diverse range of issues to consider. In this post, I'm going to break down these issues into a logical, step by step process for reviewing an existing site for accessibility.

Start with the keyboard

For users who either cannot or choose not to use a mouse, keyboard navigation is their primary means of reaching everything on screen. This audience includes users with motor impairments, such as Repetitive Stress Injury (RSI) or paralysis, as well as screen reader users. For a good keyboarding experience aim to have a logical tab order and easily discernable focus styles.

Key points

- Start by tabbing through your site. The order in which elements are focused should aim to follow the DOM order. If you're unsure which elements should receive focus, see Focus Fundamentals for a refresher. The general rule of thumb is that any control a user can interact with or provide input to should aim to be focusable and display a focus indicator (e.g., a focus ring). It's common practice to disable focus styles without providing an alternative by using `outline: none` in CSS, but this is an anti-pattern. If a keyboard user can't see what's focused, they have no way of interacting with the page. If you need to differentiate between mouse and keyboard focus for styling, consider adding a library like what-input.
- Custom interactive controls should aim to be focusable. If you use JavaScript to turn a `<div>` into a fancy dropdown, it will not automatically be inserted into the tab order. To make a custom control focusable, give it a `tabindex="0"`.
- Avoid controls with a `tabindex` > 0. These controls will jump ahead of everything else in the tab order, regardless of their position in the DOM. This can be confusing for screen reader users as they tend to navigate the DOM in a linear fashion.
- Non-interactive content (e.g., headings) should avoid being focusable. Sometimes developers add a `tabindex` to headings because they think they're important. This is also an anti-pattern because it makes keyboard users who can see the screen less efficient. For screen reader users, the screen reader will already announce these headings, so there's no need to make them focusable.
- If new content is added to the page, try to make sure that the user's focus is directed to that content so they can take action on it. See Managing Focus at the Page Level for examples.
- Beware of completely trapping focus at any point. Watch out for autocomplete widgets, where keyboard focus may get stuck. Focus can be temporarily trapped in specific situations, such as displaying a modal, when you don't want the user interacting with the rest of the page - but you should aim to provide a keyboard-accessible method of escaping the modal as well. See the guide on Modals and Keyboard Traps for an example.

Just because something is focusable doesn't mean it's usable

If you've built a custom control, then aim for a user to be able to reach *all* of its functionality using only the keyboard. See Managing Focus In Components for techniques on improving keyboard access.

Don't forget offscreen content

Many sites have offscreen content that is present in the DOM but not visible, for example, links inside a responsive drawer menu or a button inside a modal window that has yet to be displayed. Leaving these elements in the DOM can lead to a confusing keyboarding experience, especially for screen readers which will announce the offscreen content as if it's part of the page. See Handling Offscreen Content for tips on how to deal with these elements.

Try it with a screen reader

Improving general keyboard support lays some groundwork for the next step, which is to check the page for proper labeling and semantics and any obstructions to screen reader navigation. If you're unfamiliar with how semantic markup gets interpreted by assistive technology, see the Introduction to Semantics for a refresher.

Key points

- Check all images for proper `alt` text. The exception to this practice is when images are primarily for presentation purposes and are not essential pieces of content. To signify that an image should be skipped by a screen reader, set the value of the `alt` attribute to an empty string, e.g., `alt=""`.
- Check all controls for a label. For custom controls this may require the use of `aria-label` or `aria-labelledby`. See ARIA Labels and Relationships for examples.
- Check all custom controls for an appropriate `role` and any required ARIA attributes that confer their state. For example, a custom checkbox will need a `role="checkbox"` and `aria-checked="true|false"` to properly convey its state. See the Introduction to ARIA for a general overview of how ARIA can provide missing semantics for custom controls.
- The flow of information should make sense. Because screen readers navigate the page in DOM order, if you've used CSS to visually reposition elements, they may be announced in a nonsensical sequence. If you need something to appear earlier in the page, try to physically move it earlier in the DOM.

- Aim to support a screen reader's navigation to all content on the page. Avoid letting any sections of the site be permanently hidden or blocked from screen reader access.
- If content *should* be hidden from a screen reader, for instance, if it's offscreen or just presentational, make sure that content is set to `aria-hidden="true"`. Take a look at the guide on Hiding content for further explanation.

Familiarity with even one screen reader goes a long way

Though it might seem daunting to learn a screen reader, they're actually pretty easy to pick up. In general, most developers can get by with just a few simple key commands.

If you're on a Mac check out this video on using VoiceOver, the screen reader that comes with Mac OS. If you're on a PC check out this video on using NVDA, a donation supported, open source screen reader for Windows.

`aria-hidden` does not prevent keyboard focus

It's important to understand that ARIA can only affect the *semantics* of an element; it has no effect on the *behavior* of the element. While you can make an element hidden to screen readers with `aria-hidden="true"`, that does not change the focus behavior for that element. For offscreen interactive content, you will often need to combine `aria-hidden="true"` and `tabindex="-1"` to make sure it's truly removed from the keyboard flow. The proposed inert attribute aims to make this easier by combining the behavior of both attributes.

Interactive elements like links and buttons should indicate their purpose and state

Providing visual hints about what a control will do helps people operate and navigate your site. These hints are called affordances. Providing affordances makes it possible for people to use your site on a wide variety of devices.

Key points

- Interactive elements, like links and buttons, should be distinguishable from non-interactive elements. It is difficult for users to navigate a site or app when they cannot tell if an element is clickable. There are many valid methods to accomplish this goal. One common practice is underlying links to differentiate them from their surrounding text.

- Similar to the focus requirement, interactive elements like links and buttons require a hover state for mouse users so they know if they are hovering over something clickable. However, the interactive element still must be distinguishable on its own. Relying on a hover state alone to indicate clickable elements does not help touch screen devices.

Take advantage of headings and landmarks

Headings and landmark elements imbue your page with semantic structure, and greatly increase the navigating efficiency of assistive technology users. Many screen reader users report that, when they first land on an unfamiliar page, they typically try to navigate by headings. Similarly, screen readers also offer the ability to jump to important landmarks like `<main>` and `<nav>`. For these reasons it's important to consider how the structure of your page can be used to guide the user's experience.

Key points

- Make proper use of `h1-h6` hierarchy. Think of headings as tools to create an outline for your page. Don't rely on the built-in styling of headings; instead, consider all headings as if they were the same size and use the semantically appropriate level for primary, secondary, and tertiary content. Then use CSS to make the headings match your design.
- Use landmark elements and roles so users can bypass repetitive content. Many assistive technologies provide shortcuts to jump to specific parts of the page, such as those defined by `<main>` or `<nav>` elements. These elements have implicit landmark roles. You can also use the ARIA `role` attribute to explicitly define regions on the page, e.g., `<div role=”search”>`. See the guide on headings and landmarks for more examples.
- Avoid `role=”application”` unless you have prior experience working with it. The `application` landmark role will tell assistive technology to disable its shortcuts and pass through all key presses to the page. This means that the keys screen reader users

typically use to move around the page will no longer work, and you will need to implement *all* keyboard handling yourself.

Quickly review headings and landmarks with a screen reader

Screen readers like VoiceOver and NVDA provide a context menu for skipping to important regions on the page. If you're doing an accessibility check, you can use these menus to get a quick overview of the page and determine if heading levels are appropriate and which landmarks are in use. To learn more check out these instructional videos on the basics of VoiceOver and NVDA.

Automate the process

Manually testing a site for accessibility can be tedious and error prone. Eventually you'll want to automate the process as much as possible. This can be done through the use of browser extensions, and command line accessibility test suites.

Key points

- Does the page pass all the tests from either the aXe or WAVE browser extensions? These extensions are just two available options and can be a useful addition to any manual test process as they can quickly pick up on subtle issues like failing contrast ratios and missing ARIA attributes. If you prefer to do things from the command line, axe-cli provides the same functionality as the aXe browser extension, but can be easily run from your terminal.
- To avoid regressions, especially in a continuous integration environment, incorporate a library like axe-core into your automated test suite. axe-core is the same engine that powers the aXe chrome extension, but in an easy-to-run command line utility.
- If you're using a framework or library, does it provide its own accessibility tools? Some examples include protractor-accessibility-plugin for Angular, and a11ysuite for

Polymer and Web Components. Take advantage of available tools whenever possible to avoid reinventing the wheel.

If you're building a Progressive Web App consider giving Lighthouse a shot

Lighthouse is a tool to help measure the performance of your progressive web app, but it also uses the axe-core library to power a set of accessibility tests. If you're already using Lighthouse, keep an eye out for failing accessibility tests in your report. Fixing these will help improve the overall user experience of your site.

Wrapping Up

Making accessibility reviews a regular part of your team process, and doing these checks early and often, can help improve the overall experience of using your site. Remember, good accessibility equals good UX!

Additional Resources

- Web Accessibility by Google
- Accessibility Fundamentals
- A11ycasts

Last updated July 20, 2018.