JavaScript

- Trigger events
- Page titles
- Dynamic content
- Focus management
  - Component level
  - Page level
- State management
  - Component level
  - Page level

JavaScript plays a major role in almost everything we create—from smaller dynamic components to full products running on a JavaScript framework, such as React or Angular.

This use (or overuse) of JavaScript has brought forward many alarming trends, such as long load times due to large amounts of code, use of non-semantic HTML elements, and injection of HTML and CSS through JavaScript. And you may be unsure of how accessibility fits into each of these pieces.

JavaScript can have a huge impact on the accessibility of your site. In this module, we'll share some general patterns for accessibility that are enhanced by JavaScript, as well as solutions for accessibility issues that arise from using JavaScript frameworks.

Trigger events

JavaScript events allow users to interact with web content and perform a specific action. Many people, such as screen reader users, people with fine-motor skill disabilities, people without a mouse or trackpad, and others, rely on keyboard support to interact with the web. It's critical that you add keyboard support to your JavaScript actions, as it affects all of these users.

Let's look at a click event. If an `onClick()` event is used on a semantic HTML element such as a `<button>` or `<a>`, it naturally includes both mouse and keyboard functionality. However, keyboard functionality is not automatically applied when an `onClick()` event is added to a non-semantic

element, such as a generic `<div>`.

```
<div role="button" tabindex="0" onclick="doAction()">Click me!</div>
```

```
<button onclick="doAction()">Click me!</button>
```

Preview this comparison on CodePen.

If a non-semantic element is used for a trigger event, a keydown/keyup event must be added to detect the enter or space key press. Adding trigger events to non-semantic elements is often forgotten. Unfortunately, when it's forgotten, the result is a component that's only accessible via a mouse. Keyboard-only users are left without access to the associated actions.

Page titles

As we learned in the Document module, the page title is essential for screen reader users. It tells users what page they are on and whether they have navigated to a new page.

If you use a JavaScript framework, you need to consider how you handle page titles. This is especially important for single-page apps (SPAs) that load from a singular `index.html` file, as transitions or routes (page changes) will not involve a page reload. Each time a user loads a new page in an SPA, the title won't change by default.

For SPAs, the document.title value can be added manually or with a helper package (depending on the JavaScript framework). Announcing the updated page titles to a screen reader user may take some additional work, but the good news is you've got options, such as dynamic content.

Dynamic content

One of the most powerful JavaScript functionalities is the ability to add HTML and CSS to any element on the page. Developers can create dynamic applications based on the actions or behaviors of the users.

Let's say you need to send a message to users when they log in to your website or app. You want the message to stand out from the white background and relay the message: "You are now logged in."

You can use the element `innerHTML` to set the content:

```
document.querySelector("#banner").innerHTML = '<p>You are now logged in</p>';
```

You can apply CSS in a similar way, with `setAttribute`:

```
document.querySelector("#banner").setAttribute("style", "border-color:#0000ff;");
```

With great power comes great responsibility. Unfortunately, JavaScript injection of HTML and CSS has historically been misused to create inaccessible content. Some common misuses are listed here:

Possible misuse	Correct use
Render large chunks of non-semantic HTML	Render smaller pieces of semantic HTML
Not allowing time for dynamic content to be recognized by assistive technology	Using a `setTimeout()` time delay to allow users to hear the full message
Applying style attributes for `onFocus()` dynamically	Use `:focus` for the related elements in your CSS stylesheet
Applying inline styles may cause user stylesheets to not be read properly	Keep your styles in CSS files to keep the consistency of the theme
Creating very large JavaScript files that slow down overall site performance	Use less JavaScript. You may be able to perform similar functions in CSS (such as animations or sticky navigation), which parse faster and are more performant

For CSS, toggle CSS classes instead adding inline styles, as this allows for reusability and simplicity. Use hidden content on the page and toggle classes to hide and show content for dynamic HTML. If you need to use JavaScript to dynamically add content to your page, ensure it's simple and concise, and of course, accessible.

Focus management

In the Keyboard focus module, we covered focus order and indicator styles. Focus management is knowing when and where to trap the focus and when it shouldn't be trapped.

Focus management is critical for keyboard-only users.

Component level

You can create keyboard traps when a component's focus is not properly managed. A keyboard trap occurs when a keyboard-only user gets stuck in a component, or the focus is not maintained when it should be.

One of the most common patterns where users experience focus management issues is in a modal component. When a keyboard-only user encounters a modal, the user should be able to tab between the actionable elements of the modal, but they should never be allowed outside of the modal without explicitly dismissing it. JavaScript is essential to properly trapping this focus.

https://codepen.io/web-dev-codepen-external/pen/ZERVJMz

https://codepen.io/web-dev-codepen-external/pen/zYaydmv

Page level

Focus must also be maintained when a user navigates from page-to-page. This is especially true in SPAs, where there is no browser refresh, and all content dynamically changes. Anytime a user clicks on a link to go to another page within your application, the focus is either kept in the same place or potentially placed somewhere else entirely.

When transitioning between pages (or routing), the development team must decide where the focus goes when the page loads.

There are multiple techniques to achieve this:

- Place focus on the main container with an `aria-live` announcement.
- Put the focus back to a link to skip to the main content.
- Move the focus to the top-level heading of the new page.

Where you decide to put the focus will depend on the framework you are using and the content you want to serve up to your users. It may be context- or action-dependent.

State management

Another area where JavaScript is critical to accessibility is state management, or when a component or page's current visual state is relayed to a low-vision, blind, or deafblind assistive technology user.

Often, the state of a component or page is managed through ARIA attributes, as introduced in the ARIA and HTML module. Let's review a few of the most common types of ARIA attributes used to help manage the state of an element.

Component level

Depending on your page content and what information your users need, there are many ARIA states to consider when relaying information about a component to the user.

For example, you may use an `aria-expanded` attribute to tell the user whether a drop-down menu or list is expanded or collapsed.

https://codepen.io/web-dev-codepen-external/pen/oNyJeQZ

Or you might use `aria-pressed` to indicate that a button has been pressed.

https://codepen.io/web-dev-codepen-external/pen/jOKXLyv

It's important to be selective when applying ARIA attributes. Think through the user flow to understand what critical information should be conveyed to the user.

Page level

Developers often use a visually hidden area called the ARIA live region to announce changes on the screen and alert messages to assistive technology (AT) users. This area can be paired with JavaScript to notify users of dynamic changes to the page without requiring the entire page to reload.

https://codepen.io/web-dev-codepen-external/pen/Yzvdxdr

Historically, JavaScript has struggled to announce content in `aria-live` and alert regions due to its dynamic nature. Asynchronously adding content into the DOM makes it hard for AT to pick up the region and announce it. For the content to be properly read, the live or alert region must be in the DOM on load, then the text can dynamically be swapped out.

If you use a JavaScript framework, the good news is almost all of them have a "live announcer" package that does all the work for you and is fully accessible. There is no need to worry about creating a live region and dealing with the issues described in the previous section.

Here are some live packages for common JavaScript frameworks:

- React: react-aria-live and react-a11y-announcer
- Angular: `LiveAnnouncer`
- Vue: vue-a11y-utils

Modern JavaScript is a powerful language that allows web developers to create robust web applications. This sometimes leads to over-engineering and, by extension, inaccessible patterns. By following the JavaScript patterns and tips in this module, you can make your apps more accessible to all users.

Special thanks to Mark Steadman for providing additional support on this module. Read more of his accessibility articles.

□□□ #3
□ □□□ □□□ 2026-10-31 22:44:19 UTC
□ cocobook-sync □□□ 2026-11-04 17:39:05 UTC