Control focus with tabindex

- Check if your controls are keyboard accessible
- Insert an element into the tab order
- Remove an element from the tab order
- Avoid tabindex > 0
- Create accessible components with "roving tabindex"
- Keyboard access recipes

Standard HTML elements such as `<button>` or `<input>` have keyboard accessibility built in for free. If you're building *custom* interactive components, however, use the `tabindex` attribute to ensure that they're keyboard accessible.

Whenever possible, use a built-in HTML element rather than building your own custom version. `<button>`, for example, is very easy to style and already has full keyboard support. This will save you from needing to manage `tabindex` or add semantics with ARIA.

Check if your controls are keyboard accessible

A tool like Lighthouse is great at detecting certain accessibility issues, but some things can only be tested by a human.

Try pressing the `Tab` key to navigate through your site. Are you able to reach all the interactive controls on the page? If not, you may need to use `tabindex` to improve the focusability of those controls.

If you don't see a focus indicator at all, it may be hidden by your CSS. Check for any styles that mention `:focus { outline: none; }`. You can learn how to fix this in our guide on styling focus.

Insert an element into the tab order

Insert an element into the natural tab order using `tabindex="0"`. For example:

```
<div tabindex="0">Focus me with the TAB key</div>
```

To focus an element, press the `Tab` key or call the element's `focus()` method.

https://codepen.io/web-dot-dev/pen/XWzzMqp

Remove an element from the tab order

Remove an element using `tabindex="-1"`. For example:

```
<button tabindex="-1">Can't reach me with the TAB key!</button>
```

This removes an element from the natural tab order, but the element can still be focused by calling its `focus()` method.

https://codepen.io/web-dot-dev/pen/BammWGg

Note that applying `tabindex="-1"` to an element doesn't affect its children; if they're in the tab order naturally or because of a `tabindex` value, they'll remain in the tab order. To remove an element and all its children from the tab order, consider using the WICG's `inert` polyfill. The polyfill emulates the behavior of a proposed `inert` attribute, which prevents elements from being selected or read by assistive technologies.

The `inert` polyfill is experimental and may not work as expected in all cases. Test carefully before using in production.

Avoid `tabindex > 0`

Any `tabindex` greater than 0 jumps the element to the front of the natural tab order. If there are multiple elements with a `tabindex` greater than 0, the tab order starts from the lowest value

greater than zero and works its way up.

Using a `tabindex` greater than 0 is considered an **anti-pattern** because screen readers navigate the page in DOM order, not tab order. If you need an element to come sooner in the tab order, it should be moved to an earlier spot in the DOM.

Lighthouse makes it easy to identify elements with a `tabindex` > 0. Run the Accessibility Audit (Lighthouse > Options > Accessibility) and look for the results of the "No element has a [tabindex] value greater than 0" audit.

Create accessible components with "roving `tabindex`"

If you're building a complex component, you may need to add additional keyboard support beyond focus. Consider the built-in `select` element. It is focusable and you can use the arrow keys to expose additional functionality (the selectable options).

To implement similar functionality in your own components, use a technique known as "roving `tabindex`". Roving tabindex works by setting `tabindex` to -1 for all children except the currently-active one. The component then uses a keyboard event listener to determine which key the user has pressed.

When this happens, the component sets the previously focused child's `tabindex` to -1, sets the to-be-focused child's `tabindex` to 0, and calls the `focus()` method on it.

Before

```
<div role="toolbar">
  <button tabindex="-1">Undo</button>
  <button tabindex="0">Redo</button>
  <button tabindex="-1">Cut</button>
</div>
```

After

```
<div role="toolbar">
  <button tabindex="-1">Undo</button>
  <button tabindex="-1">Redo</button>
  <button tabindex="0">Cut</button>
</div>
```

https://codepen.io/web-dot-dev/pen/LYOOWam

Curious what those `role=""` attributes are for? They let you change the semantics of an element so it will be announced properly by a screen reader. You can learn more about them in our guide on screen reader basics.

Keyboard access recipes

If you're unsure what level of keyboard support your custom components might need, you can refer to the ARIA Authoring Practices 1.1. This handy guide lists common UI patterns and identifies which keys your components should support.

□□□ #3
□ □□□ □□□ 2026-10-31 21:33:03 UTC
□ cocobook-sync □□□ 2026-11-04 17:36:22 UTC