Inclusive Components

Heydon Pickering

Accessible web interfaces, piece by piece

This black book is dedicated to my greyhound, Sweep, who is also black. A practically useless but improbably loveable creature, she is two parts bat and one part hairy dragon. I rescued her and I think she rescued me, since my other black dog, Depression, seemed to leave when she arrived. I'll miss her when she is gone."

Published 2019 by Smashing Media AG, Freiburg, Germany.
All rights reserved.
ISBN: 978-3-945749-82-1

Cover design: Heydon Pickering
Copyediting: Owen Gregory
Interior layout: Alex Clarke, Markus Seyfferth
Ebook production: Heydon Pickering

Inclusive Components was written by Heydon Pickering.

Please send errors to: errata@smashingmagazine.com

Contents

Introduction: A personal note

I am not a computer scientist. I have no idea how to grow a computer in a test tube, or how to convert the mysterious breast-enlarging substance ‘silicone’ into a semi-sentient logic machine. Or whatever it is computer scientists do.

That’s not to say I haven’t been around computers since, well, Lemmings. In fact, my Dad helped me build my first computer, because building computers was a thing back then. It just turns out I can *use* my computer without having to know the entire history of computing, or by remembering where each board and connector inside the beige metal box goes, or why. Some very clever people — mostly women — gave us computers. Good, thank you. Now let’s get to work.

It’s been a good decade since anyone assumed I would know how to fix their computer just because I bought my computer before they did theirs. Which leads me to think we’ve moved away from that era where everyone was clumsily divided into *computery* and *not computery*. But that makes it all the more astonishing that the world of professional web development is so fond of that binary.

Me in, I don’t know, 1988 probably?

The ascendant Full Stack Developer is someone who does *all* the code things. They are code’s gatekeepers. Considering the sheer scale of our project to digitize the entirety of human experience into multivarious simulacra, I think that’s rather a lot for any individual to take on.

You can do *all the code,* but only if you don’t do it all well. There’s just too much to learn to be an expert in everything. So when we hire generalist coders, we create terrible products and interfaces. The web isn’t inaccessible because web accessibility is especially hard to learn or implement. It’s inaccessible because it’s about the code where humans and computers meet, which is not a position most programmers care to be in, or are taught how to deal with. But they’re *the coders* so it’s their job, I guess.

Like I said, I'm not a computer scientist, but I learned to code because I started to work with the web. It was my *responsibility* to learn how to code, because code is what the web is made of. But the code of the web is not all the code of classical computer science, and should not be judged on the same terms. HTML is the code of writers, and CSS the code of graphic designers. Writers and designers are best positioned to write those kinds of code.

This book, an anthology of updated and expanded blog posts originally written for *inclusive-components.design*,[1] is designed to help you catch up on the kind of coding not taught in Java 101: the code of communication, interaction, and most of all *accommodation*. There's a lot of code in this book, but it's all code bent towards one specific goal: making interfaces more usable to more and different people. That's the only code I *really* know.

I dedicate this book to all the artists, designers, and humanities scholars who contribute code to the web. I also dedicate it to full stack developers, because you folks may have bitten off more than you can chew. And it's not your fault, it's the culture of expectation around you. Hopefully this will help to keep your heads above water, at least in terms of inclusive interface design.

1 http://inclusive-components.design

Thank you to all the people who have read and shared the articles from the blog, and especially to those who have helped fund its writing. Writing is my favorite thing, whether it’s natural language or code. I’m just lucky that English is my first language, because it takes me forever to learn the syntax of anything. If you wish to translate the book, please contact me using *heydon@heydonworks.com*, find me on Twitter as @*heydonworks*, or on Mastodon as @*heydon@mastodon.social*.

Yours — Heydon

Collapsible Sections

Collapsible sections are perhaps the most rudimentary of interactive design patterns on the web. All they do is let you toggle the visibility of content by clicking that content's label. Big whoop.

Although the interaction is simple, it's an interaction that does not have a consistent native implementation across browsers[59] — despite movement to standardize it. It is therefore a great "hello world" entry point into accessible interaction design using JavaScript and WAI-ARIA.

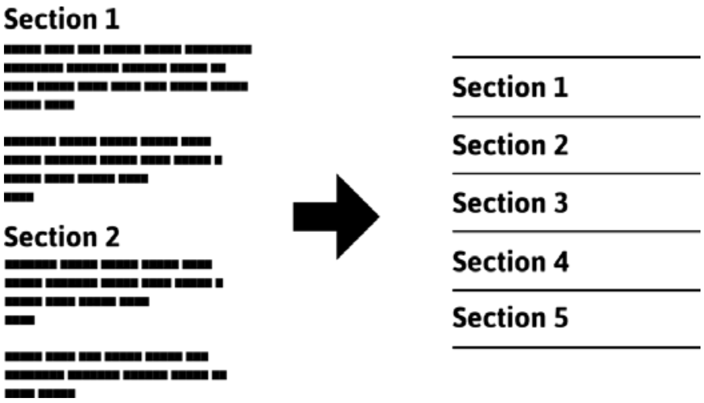
So why am I talking about it now, after covering more complex components? Because this article will focus on developer and author experience: we're going to make our collapsible regions web components, so they are easy to include as part of larger patterns and in content files.

As we did when approaching tab interfaces, it helps to consider what our component would be in the absence of JavaScript enhancement and why that enhancement actually makes things better. In this case, a collapsible section without JavaScript is simply a section. That is, a subheading introducing some content — prose, media, whatever.

59 https://smashed.by/caniusedetails

```
<h2>My section</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Cras efficitur laoreet massa. Nam eu porta
dolor. Vestibulum pulvinar lorem et nisl tempor
lacinia.</p>
<p>Cras mi #nisl, semper ut gravida sed, vulputate vel
mauris. In dignissim aliquet fermentum. Donec arcu
nunc, tempor sed nunc id, dapibus ornare dolor.</p>
```

One advantage of *collapsing* the content is that the headings become adjacent elements, giving the user an overview of the content available without having to scroll nearly so much. Expanding the content is choosing to see it.



Another advantage is that keyboard users do not have to step through all of the focusable elements on the page to get to where they want to go: hidden content is not focusable.

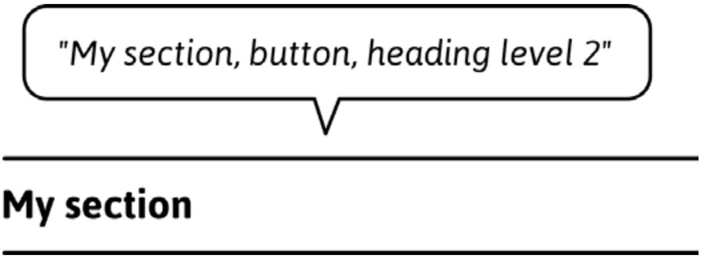
The adapted markup

Just attaching a click handler to the heading for the purposes of expanding the associated content is foolhardy, because it is not an interaction communicated to assistive software or achievable by keyboard. Instead, we need to adapt the markup by providing a standard button element.

```
<h2><button>My section</button></h2>
<div>
  <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Cras efficitur laoreet massa. Nam eu
porta dolor. Vestibulum pulvinar lorem et nisl tempor
lacinia.</p>
  <p>Cras mi nisl, semper ut gravida sed, vulputate
vel mauris. In dignissim aliquet fermentum. Donec arcu
nunc, tempor sed nunc id, dapibus ornare dolor.</p>
</div>
```

(**Note:** I have wrapped the content in a `<div>`, in preparation for showing and hiding it using the script to follow.)

The button is provided as a child of the heading. This means that, when a screen reader user focuses the `<button>`, the button itself is identified but also the presence of its parent: *“My section, button, heading level 2”* (or similar, depending on the screen reader).



Had we instead *converted* the heading into a button using ARIA’s `role="button"` we would be overriding the heading semantics. Screen reader users would lose the heading as a structural and navigational cue.

In addition, we would have to custom-code all of the browser behaviors `<button>` gives us for free, such as focus (see `tabindex` in the example below) and key bindings to actually activate our custom control.

```
<!-- DON'T do this -->
<h2 role="button" tabindex="0">My section</h2>
```

STATE

Our component can be in one of two mutually exclusive states: collapsed or expanded. This state can be suggested visually, but also needs to be communicated non-visually. We can do this by applying `aria-expanded` to the button, initially in the `false` (collapsed) state. Accordingly, we need to hide the associated `<div>` — in this case, with `hidden`.

```
<h2><button aria-expanded="false">My section</
button></h2>
<div hidden>
  <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Cras efficitur laoreet massa. Nam eu
porta dolor. Vestibulum pulvinar lorem et nisl tempor
lacinia.</p>
  <p>Cras mi nisl, semper ut gravida sed, vulputate
vel mauris. In dignissim aliquet fermentum. Donec arcu
nunc, tempor sed nunc id, dapibus ornare dolor.</p>
</div>
```

Some make the mistake of placing `aria-expanded` on the *target* element rather than the control itself. This is understandable since it is the content that actually switches state. But, if you think about it, this wouldn't be any good: the user would have to find the expanded content — which is only possible if it's actually expanded! — and then look around for an element that might control it. State is, therefore, communicated through the control that one uses to switch it.

IS THAT ALL THE BUTTON ARIA?

Why yes. We don't need to add `role="button"` because the `<button>` element implicitly has that role (the ARIA role is just for imitating the native role). And unlike menu buttons, we are not instigating an immediate change of context by moving focus. Therefore, `aria-haspopup` is not applicable.

Some folks add `aria-controls` and point it to the content container's `id`. Be warned that `aria-controls` only works in JAWS[60] at the time of writing. So long as the section's content follows the heading/button in the source order, it isn't needed. The user will (immediately) encounter the expanded content as they move down the page.

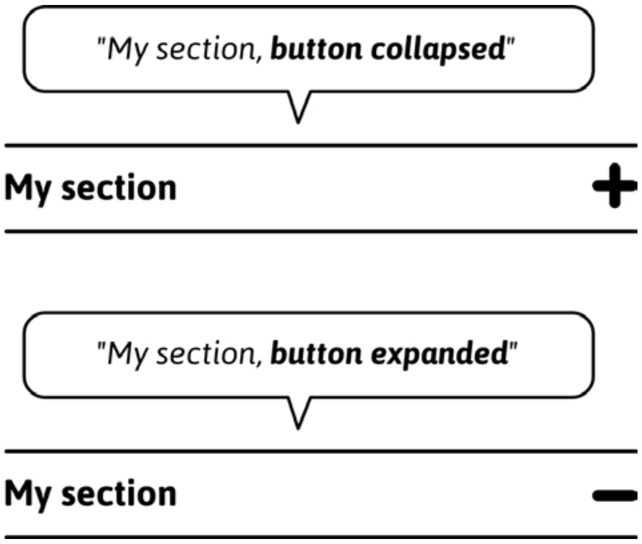
STYLING THE BUTTON

We've created a situation wherein we've employed a button, but a button that should look like an enhanced version of the heading it populates. The most efficient way to do this is to eradicate any user agent and author styles for buttons, forcing this button to inherit from its heading parent.

```
h2 button {
  all: inherit;
}
```

60 https://smashed.by/ariacontrols

Great, but now the button has no affordance.[61] It doesn't look like it can be activated. This is where, conventionally, a plus/minus symbol is incorporated. Plus indicates that the section can be expanded, and minus that it may be collapsed.


"My section, **button collapsed**"
My section
"My section, **button expanded**"
My section

The text label and/or icon for a button should always show what pressing that button will *do, hence the minus sign in the expanded state indicating that the button will take the section content away.*

61 https://smashed.by/affordances

The question is: how do we render the icon? The answer: as efficiently and accessibly as possible. Simple shapes such as rectangles (`<rect>`) are a highly efficient way to create icons with SVG, so let's do that.

```
<svg viewBox="0 0 10 10">
  <rect height="8" width="2" y="1" x="4"/>
  <rect height="2" width="8" y="4" x="1"/>
</svg>
```

There, that's small enough to fit in a tweet. Since the parent button is the control, we don't need this graphic to be interactive. In which case, we need to add the `focusable="false"` attribute, which prevents Internet Explorer and early versions of Edge from putting the SVG in focus order.

```
<button aria-expanded="false">
  My section
  <svg viewBox="0 0 10 10" focusable="false">
    <rect class="vert" height="8" width="2" y="1"
x="4"
 />
    <rect height="2" width="8" y="4" x="1" />
  </svg>
</button>
```

Note the class of “vert” for the rectangle that represents the vertical strut. We’re going to target this with CSS to show and hide it depending on the state, transforming the icon between a plus and minus shape.

```
[aria-expanded="true"] .vert {
  display: none;
}
```

Tying state and its visual representation together is *a very good thing*. It ensures that state changes are communicated interoperably. Do not listen to those who advocate the absolute separation of HTML semantics and CSS styles. Form should follow function, and *directly* is most reliable. It’s also more efficient, because there’s one less attribute to augment.

```
button.setAttribute('aria-expanded', !expanded);
// Not needed ↓
button.classList.toggle('expanded');
```

Note that the default focus style was removed with `inherit: all`. We can delegate a focus style to the SVG with the following:

```
h2 button:focus svg {
  outline: 2px solid;
}
```

High contrast themes

One more thing: we can ensure the `<rect>` elements respect high contrast themes. By applying a `fill` of `currentColor` to the `<rect>` elements, they change color with the surrounding text when it is affected by the theme change.

```
[aria-expanded] rect {
  fill: currentColor;
}
```

To test high contrast themes against your design on Windows, search for **High contrast settings** and apply a theme from **Choose a theme**. Many high contrast themes invert colors to reduce light intensity. This helps folks who suffer migraines or photophobia, as well as making elements clearer to those with vision impairments.

Why not use `<use>`?

If we had many collapsible regions on the page, reusing the same SVG `<pattern>` definition via `<use>` elements[62] and `xlink:href` would reduce redundancy.

```
<button aria-expanded="false">
  My section
  <svg viewBox="0 0 10 10 aria-hidden="true"
focusable="false">
    <use xlink:href="#plusminus" />
  </svg>
</button>
```

Unfortunately, this would mean we could no longer target the specific `.vert` rectangle to show and hide it. By using little code to define each identical SVG, bloat is not a big problem in our case.

62 https://smashed.by/uselement

A SMALL SCRIPT

Given the simplicity of the interaction and all the elements and semantics being in place, we need only write a very terse script:

```
(function() {
  const headings = document.querySelectorAll('h2');

  Array.prototype.forEach.call(headings, h => {
    let btn = h.querySelector('button');
       btn.onclick = () => {
      let expanded = btn.getAttribute('aria-expanded')
 === 'true';

      btn.setAttribute('aria-expanded', !expanded);
      target.hidden = expanded;
    }
  })
})()
```

Demo: Basic collapsible sections[63]

63 https://smashed.by/collapsiblesectionsdemo

PROGRESSIVE ENHANCEMENT

The trouble with the script above is that it requires the HTML to be adapted manually for the collapsible sections to work. Implemented by an engineer as a component via a template or JSX, this is expected. However, for largely static sites like blogs there are two avoidable issues:

- If JavaScript is unavailable, there are interactive elements in the DOM that don't do anything, with semantics that therefore make no sense.
- The onus is on the author/editor to construct the complex HTML.

Instead, we can take basic prose input (say, written in Markdown or in a WYSIWYG) and enhance it *after the fact* with the script. This is quite trivial in jQuery given the `nextUntil` and `wrapAll` methods, but in plain JavaScript we need to do some iteration. Here's another demo that automatically adds the toggle button and groups the section content for toggling. It targets all `<h2>`s found in the `<main>` part of the page.

Demo: Progressive collapsible sections[64]

64 https://smashed.by/progcollapsiblesections

Why write it in plain JavaScript? Because modern browsers support Web API methods very consistently now, and because small interactions should not depend on large libraries.

A progressive web component

The last example meant we didn't have to think about our collapsible sections during editorial; they'd just appear automatically. But what we gained in convenience, we lost in control. Instead, what if there was a compromise wherein there was very little markup to write, but what we *did* write let us choose which sections should be collapsible and what state they should be in on page load?

Web components could be the answer. Consider the following:

```
<toggle-section open="false">
  <h2>My section</h2>
  <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Cras efficitur laoreet massa. Nam eu
porta dolor. Vestibulum pulvinar lorem et nisl tempor
lacinia.</p>
  <p>Cras mi nisl, semper ut gravida sed, vulputate
vel mauris. In dignissim aliquet fermentum. Donec arcu
nunc, tempor sed nunc id, dapibus ornare dolor.</p>
</toggle-section>
```

The custom element name is easy to remember, and the `open` attribute has obvious implications. Better still, where Java-Script is unavailable, this outer element is treated like a mere `<div>` and the collapsible section remains a simple section. No real harm done.

In fact, if we detect support for the `<template>` element and `attachShadow` within our script, the same fallback will be presented to browsers not supporting these features.

```
if ('content' in document.createElement('template')) {
  // Define the <template> for the web component

  if (document.head.attachShadow) {

    // Define the web component using the v1 syntax
  }
}
```

Frameworks or web components?

The promise of web components is that you should be able to create components like you would in React or Vue, but in native code. Fewer dependencies, and faster to run.

However, as noted in “The Case for React-like Web Components,”[65] web components are limited when in comes to data binding and state.

Nonetheless, there’s a good case for writing at least your *functional* components as web components. The more of your design system that’s written in native code, the more interoperable, reusable, and future-proof it is.

THE TEMPLATE

We could place a template element in the markup and reference it, or create one on the fly. I’m going to do the latter.

```
tmpl.innerHTML = `
  <h2>
    <button aria-expanded="false">
      <svg aria-hidden="true" focusable="false"
viewBox="0 0 10 10">
        <rect class="vert" height="8" width="2" y="1"
x="4"/>
        <rect height="2" width="8" y="4" x="1"/>
      </svg>
    </button>
  </h2>
  <div class="content" hidden>
    <slot></slot>
  </div>
```

65 https://smashed.by/reactwebcomponents

```
<style>
  h2 {
    margin: 0;
  }

  h2 button {
    all: inherit;
    box-sizing: border-box;
    display: flex;
    justify-content: space-between;
    width: 100%;
    padding: 0.5em 0;
   }
 button svg {
    height: 1em;
    margin-left: 0.5em;
  }

  [aria-expanded="true"] .vert {
    display: none;
  }

  [aria-expanded] rect {
    fill: currentColor;
  }
</style>
```

This template content will become the Shadow DOM subtree for the component.

By styling the collapsible section from *within* its own Shadow DOM, the styles do not affect elements in Light DOM (the standard, outer DOM). Not only that, but they are not applied unless the browser supports `<template>` and custom elements.

DEFINING THE COMPONENT

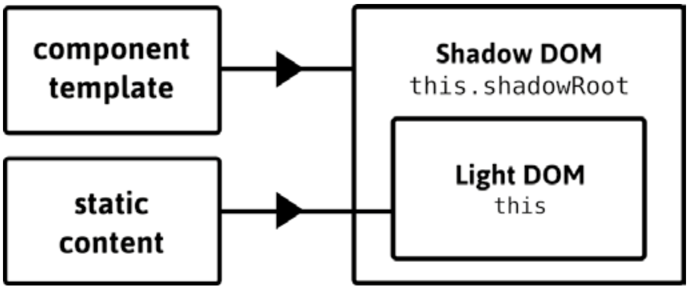
Note the `<slot>` element in the template HTML, which is a window to our Light DOM. This makes it much easier to wrap the content provided by the author than in the previous progressive enhancement demo.[66]

Inside the component definition, `this.innerHTML` refers to this Light DOM content. We shall create a `shadowRoot` and populate it with the template's content. The Shadow DOM markup is instead found with `this.shadowRoot.innerHTML`.

```
class ToggleSection extends HTMLElement {
  constructor() {
    super()

    this.attachShadow({ mode: 'open' })
    this.shadowRoot.appendChild(tmpl.content.
cloneNode(true))
  }
}
```

66 https://smashed.by/progcollapsiblesections



With these references, we can move Light DOM to Shadow DOM. Which means we can repurpose the Light DOM `<h2>`'s label and eliminate the now superfluous element. It probably seems dirty doing this DOM manipulation — especially when you're used to simple, declarative (React) components. But it's what makes the web component progressive.

```
this.btn = this.shadowRoot.querySelector('h2 button');
var oldHeading = this.querySelector('h2');
var label = oldHeading.textContent;
this.btn.innerHTML = label + this.btn.innerHTML;
oldHeading.parentNode.removeChild(oldHeading);
```

Actually, we can do one better and support different introductory heading levels. Instead of targeting headings at all, we can just get the first element in the Light DOM. Making sure the first element is a heading would be a matter for editorial guidance. However, if it's not a heading, we can make good of any element — as I shall demonstrate.

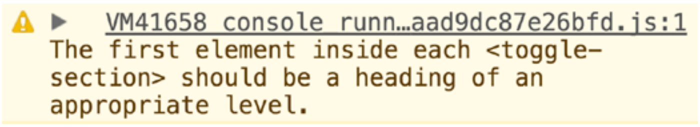
```
var oldHeading = this.querySelector(':first-child')
```

Now we just need to make sure the *level* for the Shadow DOM heading is faithful to the Light DOM original. I can query the `tagName` of the Light DOM heading and augment the Shadow DOM level with `aria-level` accordingly.

```
let level = parseInt(oldHeading.tagName.substr(1));
this.heading = this.shadowRoot.querySelector('h2');
if (level && level !== 2) {
  this.heading.setAttribute('aria-level', level);
}
```

The second character of `tagName` is parsed as an integer. If this is a true integer (`NaN` is falsey) and isn't the `2` offered implicitly by `<h2>`, `aria-level` is applied. As a fallback, a non-heading element still gives up its `textContent` as the label for the extant Shadow DOM `<h2>`. This can be accompanied by a polite `console.warn`, advising developers to use a heading element as a preference.

```
if (!level) {
  console.warn('The first element inside each <toggle-
section> should be a heading of an appropriate
level.');
}
```



```
VM41658 console runn…aad9dc87e26bfd.js:1
The first element inside each <toggle-
section> should be a heading of an
appropriate level.
```

One advantage of using `aria-level` is that, in our case, it is not being used as a styling hook — so the appearance of the heading/button remains unchanged.

```
<h2 aria-level="3">
  <button aria-expanded="false">
    <svg aria-hidden="true" focusable="false"
viewBox="0 0 10 10">
      <rect class="vert" height="8" width="2" y="1"
x="4"/>
      <rect height="2" width="8" y="4" x="1"/>
    </svg>
  </button>
</h2>
```

If you wanted your collapsible section headings to reflect their level, you could include something like the following in your CSS:

```
toggle-section [aria-level="2"] {
  font-size: 2rem;
}

toggle-section [aria-level="3"] {
  font-size: 1.5rem;
}

/* etc */
```

The region role

Any content that is introduced by a heading is a *de facto* (sub)section within the page. But, as I covered in chapter 2, "A To-do List", you can create explicit sectional container elements in the form of `<section>`. You get the same effect by applying `role="region"` to an element, such as our custom `<toggle-section>` (which otherwise offers no such accessible semantics).

```
<toggle-section role="region">
  ...
</toggle-section>
```

Screen reader users are more likely to traverse a document by heading than region[67] but many screen readers *do* provide region shortcuts. Adding `role="region"` gives us quite a bit:

67 https://smashed.by/screenreadersurvey

- It provides a fallback navigation cue for screen reader users where the Light DOM does not include a heading.
- It elicits the announcement of "region" when the screen reader user enters that section, which acts as a structural cue.
- It gives us a styling hook in the form `toggle-button[role="region"]`. This lets us add styles we only want to see if the script has run and web components are supported.

TETHERING OPEN AND ARIA-EXPANDED

When the component's `open` attribute (a Boolean) is added or removed, we want the appearance of the content to toggle. By harnessing `observedAttributes()` and `attributeChangedCallback()` we can do this *directly*. We place this code after the component's constructor:

```
get open() {
  return this.hasAttribute('open');
}

set open(val) {
  if (val) {
    this.setAttribute('open', '');
  } else {
    this.removeAttribute('open');
  }
}
static get observedAttributes() {
  return ['open']
}

attributeChangedCallback(name) {
  if (name === 'open') {
    this.switchState();
  }
}
```

- `observedAttributes()` takes an array of all the attributes on the parent `<toggle-section>` that we wish to watch
- `attributeChangedCallback(name)` lets us execute our `switchState()` function in the event of a change to `open`

The advantage here is that we can toggle state using a script that simply adds or removes `open`, from outside the component. For *users* to change the state, we can just flip `open` inside a click function:

```
this.btn.onclick = () => {
  this.toggleAttribute('open');
}
```

Since the `switchState()` function augments the `aria-expanded` value, we have tethered `open` to `aria-expanded`, making sure the state change is accessible.

```
this.switchState = () => {
  let expanded = this.hasAttribute('open');
  this.btn.setAttribute('aria-expanded', expanded);
  this.shadowRoot.querySelector('.content').hidden =
!expanded;
}
```

Demo: Web component with additional expand/collapse all functionalities[68]

EXPAND/COLLAPSE ALL

Since we toggle `<toggle-section>` elements via their `open` attribute, it's trivial to afford users an 'expand/collapse all' behavior. One advantage of such a provision is that users who have opened multiple sections independently can reset to an initial, compact state for a better overview of the content. By the same token, users who find fiddling with interactive elements distracting or tiresome can revert to scrolling through open sections.

68 https://smashed.by/expandcollapseall

It’s tempting to implement ‘expand/collapse all’ as a single toggle button. But we don’t know how many sections will initially be in either state. Nor do we know, at any given time, how many sections the user has opened or closed manually.

Instead, we should group two alternative controls.

```
<ul class="controls">
  <li><button id="expand">expand all</button></li>
  <li><button id="collapse">collapse all</button></li>
</ul>
```

It’s important to group related controls together, and lists are the standard markup for doing so. (See also chapter 3 “Menus and Menu Buttons” on page 70.) Lists and list items tell screen reader users when they are interacting with related elements and how many of these elements there are.

Some compound ARIA widgets have their own grouping mechanisms, like `role="menu"` grouping `role="menuitem"` elements, or `role="tablist"` grouping `role="tab"` elements. Our use case does not suit either of these paradigms, and a simple list suffices.

Arguably, a group label should be provided to the controls as well. I don’t believe it’s necessary here because the individual labels are sufficiently descriptive. It is possible, to use `aria-label` and `aria-labelledby` with `<ul>` elements, however.

TRACKING THE URL

One final refinement.

Conventionally, and in the absence of JavaScript enhancement, users are able to follow and share links to specific page sections by their `hash`. This is expected, and part of the generic UX of the web.

Most parsers add `id` attributes for this purpose to heading elements. As the heading element for a target section in our enhanced interface may be inside a collapsed/unfocusable section, we need to open that to reveal the content and move focus to it. The `connectedCallback()` life cycle lets us do this when the component is ready. It's like `DOMContentLoaded` but for web components.

```
connectedCallback() {
  if (window.location.hash.substr(1) === this.heading.
id) {
    this.setAttribute('open', 'true');
    this.btn.focus();
  }
}
```

Note that we focus the button inside the component's heading. This takes keyboard users to the pertinent component ready for interaction. In screen readers, the parent heading level will be announced along with the button label.

Further to this, we should be updating the hash each time the user opens successive sections. Then they can share the specific URL without needing to dig into dev tools (if they know how!) to copy/paste the heading's `id`. Let's use `pushState` to dynamically change the URL without reloading the page:

```
this.btn.onclick = () => {
  let open = this.getAttribute('open') === 'true';
  this.setAttribute('open', open ? 'false' : 'true');

  if (this.heading.id && !open) {
    history.pushState(null, null, '#' + this.heading.
id);
  }
}
```

Demo: Final version, with history (hash tracking)[69]

(Note that the presence of the open property will mean the section is open, regardless of whether it matches the URL #)

69 https://smashed.by/withhistory

Conclusion

Your role as an interface designer and developer (yes, you can be both at the same time) is to serve the needs of the people receiving your content and using your functionality. These needs encompass both those of end users and fellow contributors. The product should, of course, be accessible and performant, but maintaining and expanding the product should be possible without esoteric technical knowledge.

Whether implemented through web components or not, progressive enhancement not only ensures the interface is well structured and robust. As we've seen here, it can also simplify the editorial process. This makes developing the application and its content more inclusive.

CHECKLIST

- Don't depend on large libraries for small interactions, unless the library in question is likely to be used for multiple other interactive enhancements.
- Do not override important element roles. See the second rule of ARIA use.[70]
- Support high contrast themes in your SVG icons with `currentColor`.
- If the content is already otherwise static, there is a good case for basing your web component on progressive enhancement.
- Do please come up with more descriptive labels for your sections than "Section 1", "Section 2" etc. Those are just placeholders!

70 https://smashed.by/2ndrule

The world is a miracle. So are you.

Thanks for being smashing.