Inclusive Components

Heydon Pickering

Accessible web interfaces, piece by piece

Inclusive Components
Heydon Pickering

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 ‘silicon’ 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.

Me in, I don't know, 1988 probably?

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.

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 to 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, 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.

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

Toggle buttons

Some things are either on or off and, when those things aren't on (or off), they are invariably off (or on). The concept is so rudimentary that I've only complicated it by trying to explain it, yet on/off switches (or toggle buttons) are not all alike. Although their purpose is simple, their applications and forms vary greatly.

In this inaugural chapter, I'll be exploring what it takes to make toggle buttons inclusive. As with any component, there's no one way to go about this, especially when such controls are examined under different contexts. However, there's certainly plenty to forget to do or to otherwise screw up, so let's try to avoid any of that.

Changing state

If a web application did not change according to the instructions of its

user, the resulting experience would be altogether unsatisfactory. Nevertheless, the luxury of being able to make web documents augment themselves instantaneously, without recourse to a page refresh, has not always been present.

Unfortunately, somewhere along the way we decided that accessible web pages were only those where very little happened — static documents, designed purely to be read. Accordingly, we made little effort to make the richer, *stateful* experiences of web applications inclusive.

A popular misconception has been that screen readers *don't understand* JavaScript. Not only is this entirely untrue—all major screen readers react to changes in the DOM as they occur—but basic state changes, communicated both visually and to assistive technology software, do not necessarily depend on JavaScript to take place anyway.

Checkboxes and `radio` buttons

Form elements are the primitives of interactive web pages and, where we're not employing them directly, we should be paying close attention to how they behave. Their handling of state changes have established usability conventions we would be foolish to ignore.

Arguably, an out-of-the-box input of the `checkbox` type is a perfectly serviceable on/off switch all its own. Where labelled correctly, it has all the fundamental ingredients of an accessible control: it's screen reader and keyboard accessible between platforms and devices, and it communicates its change of state (*checked* to *unchecked* or vice versa) without needing to rebuild the entire document.

In the following example, a checkbox serves as the toggle for an email

notifications setting.

```
<input type="checkbox" id="notify" name="notify"
value="on">
<label for="notify">Notify by email</label>
```

Screen reader software is fairly uniform in its interpretation of this control. On focusing the control (moving to it using the Tab key) something similar to, *"Notify by email, checkbox, unchecked"* will be announced. That's the label, role, and state information all present.

On checking the checkbox, most screen reader software will announce the changed state, "checked" (sometimes repeating the label and role information too), immediately. Without JavaScript, we've handled state and screen reader software is able to feed back to the user.

Note

Screen readers are not just for the blind

Some operate screen readers to assist their understanding of an interface. Others may be visually dyslexic or have low literacy. There are even those who have little physical or cognitive trouble understanding an interface who simply prefer to have it read out to them sometimes.

Supporting screen reader software is supporting screen reader software, not blind people. Screen readers are a tool a lot of different people like to use. As with many so-called 'assistive technologies', screen readers are just a tool anyone can add to their tool set.

In this case, the on/off part of the switch is not communicated by the label but the state. Instead, the label is for identifying the thing that we are turning off or on. Should research show that users benefit from a more explicit on/off metaphor, a radio button group can be employed.

```
<fieldset>
  <legend>Notify by email</legend>
  <input type="radio" id="notify-on" name="notify"
value="on" checked>
  <label for="notify-on">on</label>
  <input type="radio" id="notify-off" name="notify"
value="off">
  <label for="notify-off">off</label>
</fieldset>
```

Group labels are a powerful tool. As their name suggests, they can provide a single label to related (grouped) items. In this case, the `<fieldset>` group element works together with the `<legend>` element to provide the group label “Notify by email” to the pair of radio buttons. These buttons are made a pair by sharing a `name` attribute value, which makes it possible to toggle between them using your arrow keys. HTML semantics don’t just add information but also affect behavior.

In the Windows screen readers JAWS and NVDA, when the user focuses the first control, the group label is prepended to that control’s individual label and the grouped radio buttons are enumerated. In NVDA, the term “grouping” is appended to make things more explicit. In the above example, focusing the first (checked by default) radio button elicits, *“Notify by email, grouping, on radio button, checked, one of two”*.

Now, even though the checked state (announced as “selected” in some screen readers) is still being toggled, what we’re really allowing the user to do it switch between “on” and “off”. Those are the two possible *lexical states*, if you will, for the composite control.

Note

Styling form elements

Form elements are notoriously hard to style, but there are well-supported CSS techniques for styling radio and checkbox controls, as I wrote in *Replacing Radio Buttons Without Replacing Radio Buttons*. For tips on how to style select elements and file inputs, consult *WTF Forms?* by Mark Otto.

This doesn't quite feel right

Both the checkbox and radio button implementations are tenable as on/off controls. They are, after all, accessible by mouse, touch, keyboard, and assistive technology software across different devices, browsers, and operating systems.

But accessibility is only a part of inclusive design. These controls also have to *make sense* to users; they have to play an unambiguous role within the interface.

The trouble with using form elements is their longstanding association with the collection of data. That is, checkboxes and radio buttons are established as controls for designating *values*. When a user checks a checkbox, they may just be switching a state, but they may *suspect* they are also choosing a value for submission.

Whether you're a sighted user looking at a checkbox, a screen reader user listening to its identity being announced, or both, its etymology is problematic. We expect toggle buttons to be buttons, but checkboxes and radio buttons are really inputs, as their `<input>` elements suggest.

A true toggle button

Sometimes we use `<button>` elements to submit forms. To be fully compliant and reliable these buttons should take the `type` value of `submit`.

```
<button type="submit">Send</button>
```

But these are only one variety of button, covering one use case. In truth, `<button>` elements can be used for all sorts of things, and not just in association with forms. They're just *buttons*. We remind ourselves of this by giving them the `type` value of `button`.

```
<button type="button">Send</button>
```

The generic button is your go-to element for changing anything within the interface (using JavaScript and without reloading the page) except one's location within and between documents, which is the purview of links.

Next to links, buttons should be the interactive element you use most prolifically in web applications. They come prepackaged with the "button" role and are keyboard and screen reader accessible by default. Unlike some form elements, they are also trivial to style.

So how do we make a `<button>` a toggle button? It's a case of using WAI-ARIA as a progressive enhancement. WAI-ARIA offers states that are not available in basic HTML, such as the *pressed* state. Imagine a

power switch for a computer. When it's pressed — or pushed in — that denotes the computer is in its "on" state. When it's unpressed — poking out — the computer must be off.

```
<button type="button" aria-pressed="true">
  Notify by email
</button>
```

WAI-ARIA state attributes like `aria-pressed` behave like booleans but, unlike standard HTML state attributes like `checked` they must have an explicit value of `true` or `false`. Just adding `aria-pressed` is not reliable. Also, the absence of the attribute would mean the *unpressed* state would not be communicated (a button without the attribute is just a generic button).

You can use this control inside a form, or outside, depending on your needs. But if you do use it inside a form, the `type="button"` part is important. If it's not there, some browsers will default to an implicit `type="submit"` and try to submit the form. You don't have to use `event.preventDefault()` on `type="button"` controls to suppress form submission.

Switching the state from `true` (on) to `false` (off) can be done via a simple click handler. Since we are using a `<button>`, this event type can be triggered with a mouse click, a press of either the **Space** or **Enter** keys, or by tapping the button through a touch screen. Being responsive to each of these actions is something built into `<button>` elements as standard. If you consult the HTMLButtonElement interface, you'll see that other properties, such as `disabled`, are also supported out-of-the-box. Where `<button>` elements are not used, these behaviors have to be emulated with bespoke scripting.

```
const toggle = document.querySelector('[aria-
pressed]');

toggle.addEventListener('click', () => {
  let pressed = toggle.getAttribute('aria-pressed') ===
'true';
  toggle.setAttribute('aria-pressed', !pressed);
});
```

Demo: View the toggle button demo, using `aria-pressed`

The toggle button in Vue.js

As already explained, ARIA states need explicit “true” and “false” values. Unfortunately, some frameworks like Vue.js will magically (?) remove attributes supplied with `false` values. To get around this, your Vue toggle button needs to stringify the value so that `false` becomes “false”.

```
<button :aria-pressed="this.pressed.toString()">Press
me</button>
```

A clearer state

An interesting thing happens when a button with the `aria-pressed` state is encountered by some screen readers: it is identified as a “toggle button” or, in some cases, “push button”. The presence of the state attribute changes the button’s identity.

When focusing the example button with `aria-pressed="true"` using NVDA, the screen reader announces, *"Notify by email, toggle button, pressed"*. The "pressed" state is more apt than "checked", plus we eschew the form data input connotations. When the button is clicked, immediate feedback is offered in the form of *"not pressed"*.

Styling

The HTML we construct is an important part of the design work we do and the things we create for the web. I'm a strong believer in doing HTML First Prototyping™, making sure there's a solid foundation for the styled and branded product.

In the case of our toggle button, this foundation includes the semantics and behavior to make the button interoperable with various input (e.g. voice activation software) and output (e.g. screen reader) devices. That's possible using HTML, but CSS is needed to make the control understandable visually.

Form should follow function, which is simple to achieve in CSS: everything in our HTML that makes our simple toggle button function can also be used to give it form.

- `<button>` → `button` element selector
- `aria-pressed="true"` → `[aria-pressed="true"]` attribute selector

In a consistent and, therefore, easy to understand interface, buttons should share a certain look. Buttons should all look like buttons. So, our basic toggle button styles should probably inherit from the `button` element block:

```
/* For example... */
button {
  color: white;
  background-color: #000;
  border-radius: 0.25rem;
  padding: 1em 1.5em;
}
```

There are a number of ways we could visually denote “pressed”. Interpreted literally, we might make the button look *pressed in* using some inset `box-shadow`. Let’s employ an attribute selector for this:

```
[aria-pressed='true'] {
  box-shadow: inset 0 0 0 0.15rem #000, inset 0.25em
0.25em 0 #fff;
}
```

To complete the pressed/unpressed metaphor, we can use some positioning and `box-shadow` to make the unpressed button “poke out”. This block should appear above the `[aria-pressed=“true”]` block in the cascade.

```
[aria-pressed] {
  position: relative;
  top: -0.25rem;
  left: -0.25rem;
  box-shadow: 0.125em 0.125em 0 #fff, 0.25em 0.25em
#000;
}
```

`aria-pressed="false"`

`aria-pressed="true"`

This styling method is offered just as one idea. You may find that something more explicit, like the use of “on”/“off” labels in an example to follow, is better understood by more users.

One issue with using `box-shadow` is that it is eliminated by Windows’ High Contrast themes. You could remedy this by detecting high contrast and inserting an explicit “✓” mark for buttons in their pressed state.

```
@media (-ms-high-contrast: active) {
  [aria-pressed="true"]::after {
    content: '\0020✓';
  }
}
```

Generally it’s better to choose techniques that work in High Contrast Mode as well, saving you from having to ‘fork’ the design.

Note

Don't rely on color alone

"On" is often denoted by a green color, and "off" by red. This is a well-established convention and there's no harm in incorporating it. However, be careful not to *only* use color to describe the button's two states. If you did, some color blind users would not be able to differentiate them.

`aria-pressed="false"` `aria-pressed="true"`

These versions of the control would fail WCAG 2.0 1.4.1 Use Of Color (Level A)

Focus styles

It's important buttons, along with *all* interactive components, have focus styles. Otherwise people navigating by keyboard cannot see which element is in their control and ready to be operated.

The best focus styles do not affect layout (the interface shouldn't jiggle around distractingly when moving between elements). Typically, one would use `outline`, but `outline` only ever draws a box in most browsers. To fit a focus style around the curved corners of our button, a `box-shadow` is better. Since we are using `box-shadow` already, we have to be a bit careful: note the two comma-separated `box-shadow` styles in the pressed-and-also-focused state.

```
/* Remove the default outline and
add the outset shadow */
[aria-pressed]:focus {
  outline: none;
  box-shadow: 0 0 0 0.25rem skyBlue;
}

[aria-pressed='true']:focus {
  box-shadow: 0 0 0 0.25rem skyBlue, inset 0 0 0
0.15rem #000,
    inset 0.25em 0.25em 0 #fff;
}
```

Note that `box-shadow`s tend to be eliminated when using Windows High Contrast Mode, so take that into consideration. One trick is to use a transparent outline, which will appear (become opaque) in place of the `box-shadow` styling when High Contrast Mode is switched on.

```
[aria-pressed]:focus {
  outline: 2px solid transparent; /* for WHCM */
  box-shadow: 0 0 0 0.25rem skyBlue;
  outline: 2px solid transparent;
}
```

Changing labels

The previous toggle button design has a self-contained, unique label and differentiates between its two states using a change in attribution which elicits a style. What if we wanted to create a button that changes its label from “on” to “off” or “play” to “pause”?

It's perfectly easy to do this in JavaScript, but there are a couple of things we need to be careful about.

1. If the *label* changes, what happens with the state?
2. If the label is just "on" or "off" ("play" or "pause"; "active" or "inactive") how do we know what the button actually controls?

In the previous toggle button example, the label described *what* would be on or off. Where the "what" part is not consistent, confusion quickly ensues: once "off"/unpressed has become "on"/pressed, I have to unpress the "on" button to turn the "off" button back on. What?

As a rule of thumb, you should never change pressed state and label together. If the label changes, the button has already changed state in a sense, just not via explicit WAI-ARIA state management.

In the following example, just the label changes.

```
const button = document.querySelector('button');

button.addEventListener('click', e => {
  let text = e.target.textContent === 'Play' ? 'Pause'
: 'Play';
  e.target.textContent = text;
});
```

The problem with this method is that the label change is not announced as it happens. That is, when you click the play button, feedback equivalent to "pressed" is absent. Instead, you have to unfocus and refocus the button manually to hear that it has changed. Not an issue for sighted users, but less ergonomic for blind screen

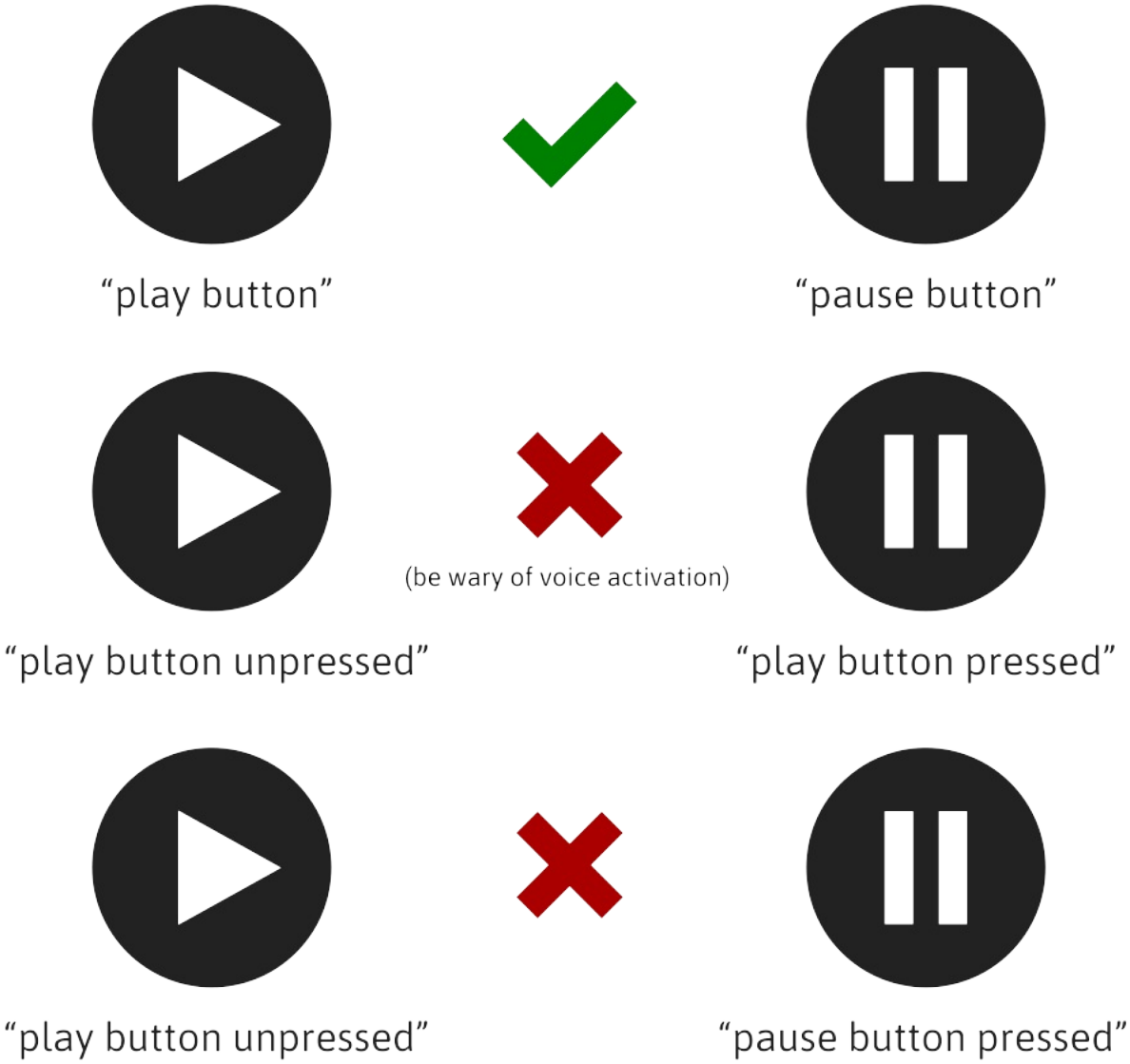
reader users.

Play/pause buttons usually switch between a play symbol (a triangle on its side) and a pause symbol (two vertical lines). We *could* do this while keeping a consistent non-visual label and changing state.

```
<!-- Paused state -->
<button type="button" aria-pressed="false" aria-
label="play">
  &#x25b6;
</button>

<!-- Playing state -->
<button type="button" aria-pressed="true" aria-
label="play">
  &#x23f8;
</button>
```

Because `aria-label` overrides the unicode symbol text node, the paused button is announced as something similar to, *"Play button, not pressed"* and the playing button as ,*"Play button, pressed"*.

This works pretty well, except for where voice recognition and activation is concerned. In voice recognition software, you typically need to identify buttons by vocalizing their labels. And if a user sees a pause symbol, their first inclination is to say "pause", not "play". For this reason, switching the label rather than the state is more robust here.



Never change label and state at the same time. In this example, that would result in a paused button in the pressed state. Since the video or audio would be playing at this point, the lexical "pause" state cannot also be considered "pressed" on "on".

Note the translation issues with `aria-label` meaning that, for an international audience — which is always the audience of the web — using a visually hidden `<span>` is preferable. Text nodes *are* reliably translated by Google's and Microsoft's translation services.

Where this is implemented, you have to manually hide the unicode points, using `aria-hidden="true"`:

```
<!-- Paused state -->
<button type="button" aria-pressed="false">
  <span aria-hidden="true">&#x25b6;</span>
  <span class="visually-hidden">play</span>
</button>

<!-- Playing state -->
<button type="button" aria-pressed="true">
  <span aria-hidden="true">&#x23f8;</span>
  <span class="visually-hidden">play</span>
</button>
```

The `visually-hidden` class is a special class that hides content visually without removing it from screen reader output. Note that `display: none` removes elements from the visual interface *and* screen reader output, so would not be appropriate here.

```
.visually-hidden {
  position: absolute !important;
  clip: rect(1px, 1px, 1px, 1px) !important;
  padding:0 !important;
  border:0 !important;
  height: 1px !important;
  width: 1px !important;
  overflow: hidden !important;
}
```

Auxiliary labeling

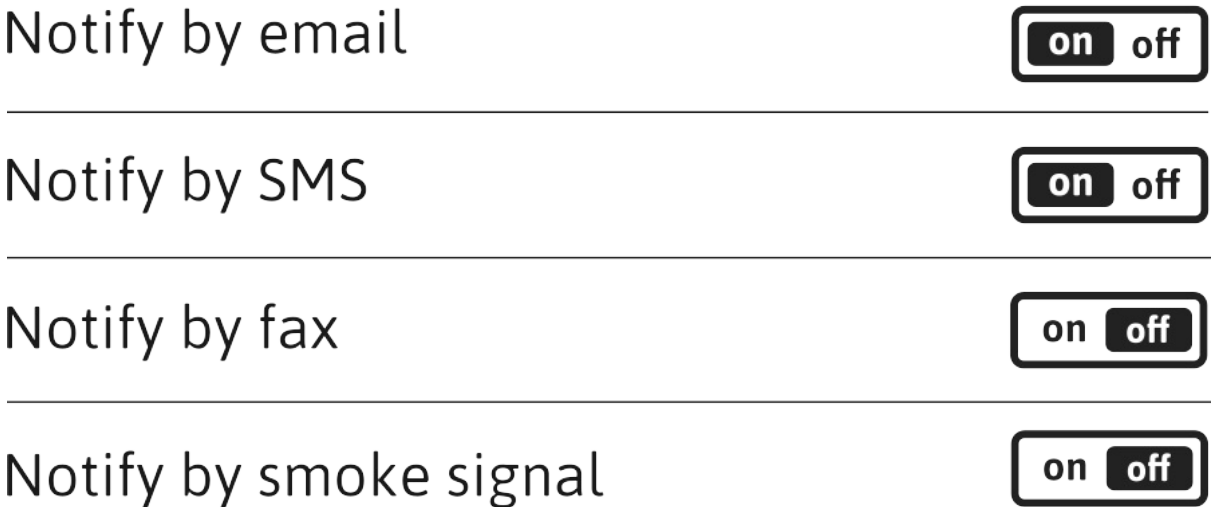
In some circumstances, we may want to provide on/off switches which actually read “on/off”. The trick with these is making sure there is a

clear association between each toggle switch and a respective, auxiliary label.

Imagine the email notification setting is grouped alongside other similar settings in a list. Each list item contains a description of the setting followed by an on/off switch. The on/off switch uses the terms “on” and “off” as part of its design. Some `<span>` elements are provided for styling.

```
<h2>Notifications</h2>
<ul>
  <li>
    Notify by email
    <button>
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <li>
    Notify by SMS
    <button>
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <!-- others -->
</ul>
```

Notifications



The virtue of lists is that, both visually and non-visually, they group items together, showing they are related. Not only does this help comprehension, but lists also provide navigational shortcuts in some screen readers. For example, JAWS provides the L (list) and I (list item) quick keys for moving between and through lists.

Each 'label' and button is associated by belonging to a common list item. However, not providing an explicit, unique label is dangerous territory — especially where voice recognition is concerned. Using `aria-labelledby`, we can associate each button with the list's text:

```
<h2>Notifications</h2>
<ul>
  <li>
    <span id="notify-email">Notify by email</span>
    <button aria-labelledby="notify-email">
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <li>
    <span id="notify-sms">Notify by SMS</span>
    <button aria-labelledby="notify-sms">
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <!-- others -->
</ul>
```

Each `aria-labelledby` value matches the appropriate span `id`, forming the association and giving the button its unique label. It works much like a `<label>` element's `for` attribute identifying a field's `id`.

The switch role

Importantly, the ARIA label overrides each button's textual content, meaning we can once again employ `aria-pressed` to communicate state. However, since these buttons are explicitly "on/off" switches, we can instead use the WAI-ARIA switch role, which communicates state via `aria-checked`.

```
<h2>Notifications</h2>
<ul>
  <li>
    <span id="notify-email">Notify by email</span>
    <button role="switch" aria-checked="true" aria-
labelledby="notify-email">
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <li>
    <span id="notify-sms">Notify by SMS</span>
    <button role="switch" aria-checked="true" aria-
labelledby="notify-sms">
      <span>on</span>
      <span>off</span>
    </button>
  </li>
  <!-- others -->
</ul>
```

How you would style the active state is quite up to you, but I'd personally save on writing class attributes to the `<span>`s with JavaScript. Instead, I'd write some CSS using pseudo classes to target the relevant span dependent on the state.

```
[role='switch'][aria-checked='true'] :first-child,
[role='switch'][aria-checked='false'] :last-child {
  background: #000;
  color: #fff;
}
```

Demo: Switch controls, with auxiliary labeling using `aria-labelledby`

Traversing the settings

Now let's talk about navigating through this settings section using two different strategies: by Tab key (jumping between focusable elements only) and browsing by screen reader (moving through each element).

Even when navigating by Tab key, it's not only the identity and state of the interactive elements you are focusing that will be announced in screen readers. For example, when you focus the first `<button>`, you'll hear that it is a *switch* with the label "Notify by email", in its *on* state. "Switch" is the role and `aria-checked="true"` is vocalized as "on" where this role is present.

Note

Switch role support

The `switch` role is not quite as well supported as `aria-pressed`. For example, it is not recognized by the ChromeVox screen reader extension for Chrome.

However, ChromeVox *does* support `aria-checked`. This means that, instead of *"Switch, Notify by email, on"* being announced, *"Button, Notify by email, checked"* is instead. This isn't as evocative, but it is adequate. More than likely, it will simply be mistaken for a checkbox input.

Curiously, NVDA regards a button with `role="switch"` and `aria-checked="true"` as a toggle button in its pressed state. Since on/off and pressed/unpressed are equivalent, this is acceptable (though slightly disappointing).

But in most screen readers you'll also be told you've entered a list of four items and that you're on the first item — useful contextual information that works a bit like the group labelling I covered earlier in this post.

Importantly, because we have used `aria-labelledby` to associate the adjacent text to the button as its label, this information is also available when navigating in this mode.

When browsing from item to item (for example, by pressing the down key when the NVDA screen reader is running), everything you encounter is announced, including the heading (*"Notifications, heading level two"*). Of course, browsing in this fashion, *"Notify by email"* is announced on its own as well as in association with the adjacent button. This is somewhat repetitive, but makes sense: "Here's the

setting name, and here's the on/off switch for the setting of this name."

How explicitly you need to associate controls to the things they control is a finer point of UX design and worth considering. In this case we've preserved our classic on/off switches for sighted users, without confusing or misleading either blind or sighted screen reader users no matter which keyboard interaction mode they are using. It's pretty robust.

Conclusion

How you design and implement your toggle buttons is quite up to you, but I hope you'll remember this chapter when it comes to adding this particular component to your pattern library. There's no reason why toggle buttons — or any interface component for that matter — should marginalize the number of people they often do.

You can take the basics explored here and add all sorts of design nuances, including animation. It's just important to lay a solid foundation first.

Checklist

- Use form elements such as checkboxes for on/off toggles if you are certain the user won't believe they are for submitting data.
- Use `<button>` elements, not links, with `aria-pressed` or `aria-checked`.
- Don't change label and state together.
- When using visual "on" and "off" text labels (or similar) you can override these with a unique label via `aria-labelledby`.
- Be careful to make sure the contrast level between the button's text and background color meets WCAG 2.0 requirements.

A Todo List

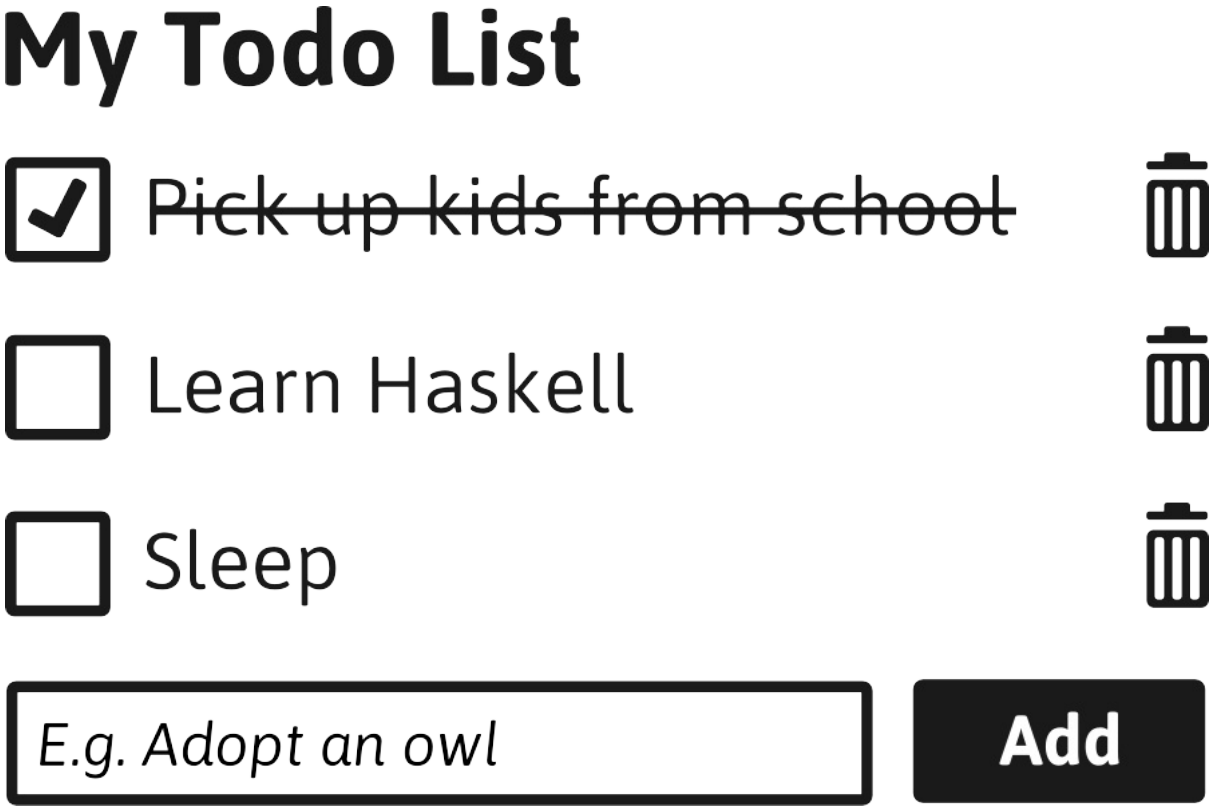
According to tradition, each new javascript framework is put through its paces in the implementation of a simple todo list app: an app for creating and deleting todo list entries. The first Angular.js example I ever read was a todo list. Adding and removing items from todo lists demonstrates the immediacy of the single-page application view/model relationship.

TodoMVC compares and contrasts todo app implementations of popular MV* frameworks including Vue.js, Angular.js, and Ember.js. As a developer researching technology for a new project, it enables you to find the most intuitive and ergonomic choice for your needs.

The inclusive design of a todo list interface is, however, framework agnostic. Your user doesn't care if it's made with Backbone or React; they just need the end product to be accessible and easy to use. Unfortunately, each of the identical implementations in TodoMVC have some shortcomings. Most notably, the delete functionality only appears on hover, making it an entirely inaccessible feature by keyboard.

In this article, I'll be building an integrated todo list component from the ground up. But what you learn doesn't have to apply *just* to todo lists — we're really exploring how to make the basic creation and deletion of content inclusive.

Unlike the simple, single element toggle buttons of the previous chapter, managed lists have a few moving parts. This is what we're going to make:



The heading

A great deal of usability is about labels. The `<label>` element provides labels to form fields, of course. But simple text nodes provided to

buttons and links are also labels: they tell you what those elements do when you press them.

Headings too are labels, giving names to the sections (regions, areas, modules) that make up an interface. Whether you are creating a static document, like a blog post, or an interactive single-page application, each major section in the content of that page should almost certainly be introduced by a heading. Our todo list's name, "My Todo List" in this case, should be marked up accordingly.

```
<h1>My Todo List</h1>
```

It's a very *on the nose* way of demarcating an interface, but on the nose is good. We don't want our users having to do any detective work to know what it is they're dealing with.

Heading level

Determining the correct level for the heading is often considered a question of importance, but it's actually a question of *belonging*. If our todo list is the sole content within the `<main>` content of the page, it should be level 1, as in the previous example. There's nothing surrounding it, so it's at the highest level in terms of depth.

If, instead, the todo list is provided as supplementary content, it should have a level which reflects that. For example, if the page is about planning a holiday, a "Things to pack" todo list may be provided as a supplementary tool.

- Plan for my trip (`<h1>`)

- Places to get drunk (<h2>)
  - Bars (<h3>)
  - Clubs (<h3>)
- Things to pack (todo list) (<h2>)

In the above example, both “Bars” and “Clubs” belong to “Places to get drunk”, which belongs to “Plan for my trip”. That’s three levels of belonging, hence the <h3>s.

Even if you feel that your packing todo list is less important than establishing which bars are good to visit, it’s still on the same *level* in terms of belonging, so it must have the same heading level.

As well as establishing a good visual hierarchy, the structure described by logically nested sections gives screen reader users a good feel for the page. Headings are also harnessed as navigational tools by screen readers. For example, in JAWS, the 2 key will take you to the next section labeled by an <h2> heading. The generic h key will take you to the next heading of any level.

Note

The <section> element

With all this talk of sections, surely we should be using <section> elements, right? Maybe. Here are a couple of things to consider:

1. Heading elements already describe sections. That is, the content that starts with a heading and ends just before a heading of the same level is a *de facto* section.
2. If you do use a <section> element, you still need to provide a heading to it, otherwise it is an unlabeled section.

In practice, the value added by `<section>` elements is limited, but still worth noting:

1. Some screen readers will announce the start and end of sections when their users are traversing the page element-to-element.
2. Some screen readers provide region navigation. For example, in JAWS, `<section>`s count as "regions" and can be moved between using r and Shift + r.
3. They can make code organization clearer by providing container elements for sections of the page.

To really make the most of `<section>`s you should label them recursively. That is, by connecting their headings to the `<section>` elements themselves using `aria-labelledby`:

```
<section aria-labelledby="todos-label">
  <h1 id="todos-label">My Todo List</h1>
  <!-- content -->
</section>
```

(Note that the `aria-labelledby` value must match the heading's `id` value.)

This effectively provides a group label to the section, meaning the label will be announced in some screen reader software upon entering the section by focus. Running the NVDA screen reader, when I enter the section and focus the first checkbox, I hear *"My Todo List region, list with three items, pick up kids from school checkbox, checked."* It's helpful to provide this contextual information to users navigating by focus rather than by region or heading.

The list

I talk about the virtues of lists in *Inclusive Design Patterns*. Alongside

headings, lists help to give pages structure. Without headings or lists, pages are featureless and monotonous, making them very difficult to unpick both visually and non-visually.

Not all lists need to be bullet lists, showing a `list-style`, but there should be some visual indication that the items within the list are similar or equivalent; that they belong together. Non-visually, using the `<ul>` or `<ol>` container means the list is identified when encountered and its items are enumerated. For our three-item todo list, screen readers should announce something like, *"list of three items"*.

A todo list is, as the name suggests, a list. Since our particular todo list component makes no assertions about priority, an unordered list is fine. Here's the structure for a static version of our todo list (the adding, deleting, and checking functionality has not yet been added):

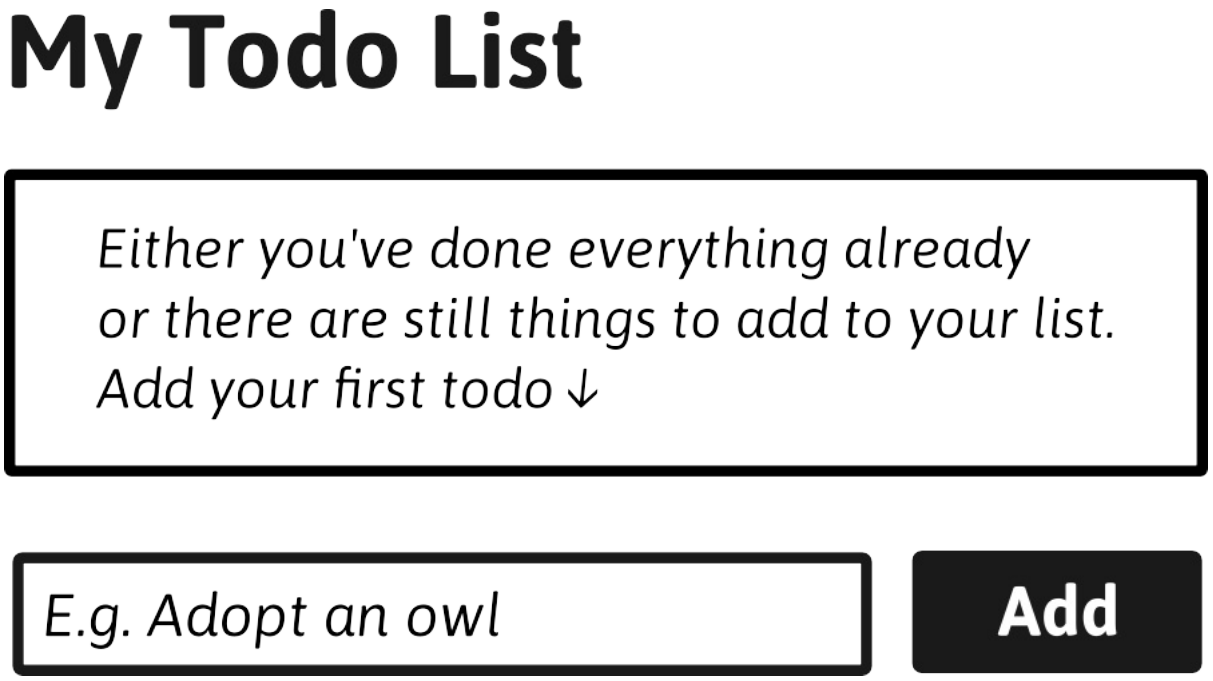
```
<section aria-labelledby="todos-label">
  <h1 id="todos-label">My Todo List</h1>
  <ul>
    <li>
      Pick up kids from school
    </li>
    <li>
      Learn Haskell
    </li>
    <li>
      Sleep
    </li>
  </ul>
</section>
```

Empty state

Empty states are an aspect of UI design which you neglect at your peril.

Inclusive design has to take user lifecycles into consideration, and some of your most vulnerable users are new ones. To them your interface is unfamiliar and, without carefully leading them by the hand, that unfamiliarity can be off-putting.

With our heading and “add” input present it may be obvious to some how to proceed, even without example todo items or instructions present. But your interface may be less familiar and more complex than this simple todo list, so let’s add an empty state anyway — for practice.



Revealing the empty state

It’s quite possible, of course, to use our data to determine whether the empty state should be present. In Vue.js, we might use a `v-if` block:

```
<div class="empty-state" v-if="!todos.length">
  <p>Either you've done everything already or there are
still things to add to your list. Add your first todo
&#x2193;</p>
</div>
```

But all the state information we need is actually already in the DOM, meaning all we need in order to switch between showing the list and showing the empty-state is CSS.

```
.empty-state, ul:empty {
  display: none;
}

ul:empty + .empty-state {
  display: block;
}
```

This is more efficient because we don't have to query the data or change the markup. It's also screen reader accessible: `display: none` makes sure the element in question is hidden both visually and from screen reader software.

All pseudo-classes pertain to implicit states. The `:empty` pseudo-class means the element is in an empty state; `:checked` means it's in a checked state; `:first-child` means it's positioned at the start of a set. The more you leverage these, the less DOM manipulation is required to add and change state with JavaScript.

Adding a todo item

We've come this far without discussing the adding of todos. Let's do that now. Beneath the list (or empty state if the list is empty) is a text input and "add" button:

Form or no form?

It's quite valid in HTML to provide an `<input>` control outside of a `<form>` element. The `<input>` will not succeed in providing data to the server without the help of JavaScript, but that's not a problem in an application using XHR.

But do `<form>` elements provide anything to users? When users of screen readers like JAWS or NVDA encounter a `<form>` element, they are automatically entered into a special interaction mode variously called "forms mode" or "application mode". In this mode, some keystrokes that would otherwise be used as special shortcuts are switched off, allowing the user to interact with the form fields fully.

Fortunately, most input types — including `type="text"` here — trigger forms mode themselves, on focus. For instance, if I were to type `h` in the pictured input, it would enter "h" into the field, rather than navigating me to the nearest heading element.

The real reason we need a `<form>` element is because we'll want to allow users to submit on `Enter`, and this only works reliably where a `<form>` contains the input upon which `Enter` is being pressed. The presence of the `<form>` is not just for code organization, or semantics,

but affects browser behavior.

```
<form>
  <input type="text" placeholder="E.g. Adopt an owl">
  <button type="submit">Add</button>
</form>
```

(**Note:** Léonie Watson reports that `range` inputs are non-functional in Firefox + JAWS unless a `<form>` is employed or forms mode is entered manually, by the user.)

Labeling

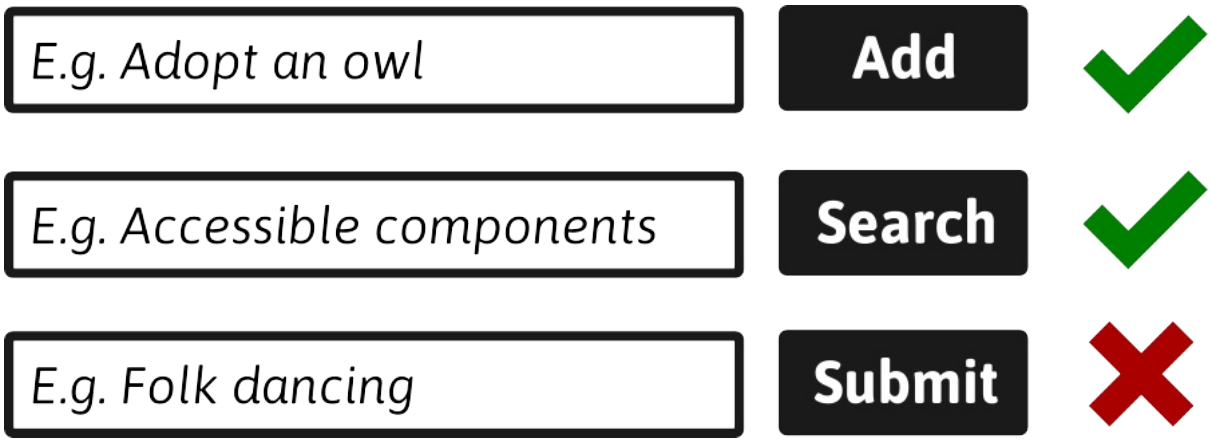
Can you spot the deliberate mistake in the above code snippet? The answer is: I haven't provided a label. Only a `placeholder` is provided and placeholders are intended for supplementary information only, such as the "adopt an owl" suggestion.

Placeholders are not reliable as labeling methods in assistive technologies, so another method must be provided. The question is: should that label be visible, or only accessible by screen reader?

In *almost* all cases, a visible label should be placed above or to the left of the input. Part of the reason for this is that placeholders disappear on focus and can be eradicated by autocomplete behavior, meaning sighted users lose their labels. Filling out information or correcting autocompleted information becomes guesswork.

However, ours is a bit of a special case because the “add” label for the adjacent button is quite sufficient. Those looking at the form know what the input does thanks to the button alone.

All inputs should have labels, because screen reader users don’t know to look ahead in the source to see if the submit button they’ve yet to reach gives them any clues about the form’s purpose. But simple input/submit button pairs like this and search regions can get away without *visible* labels. That is, so long as the submit button’s label is sufficiently descriptive.



In addition, make sure forms with multiple fields have visible labels for each field. Otherwise the user does not know which field is for what.

There are a number of ways to provide an invisible label to the input for screen reader users. One of the simpler and least verbose is `aria-label`. However, since `aria-label` values are not (as you’ll recall from the previous chapter) translatable strings, it’s better to use a `visually-hidden` `<label>` element. This is also better supported by older user agents.

```
<form>
  <label for="add-todo" class="visually-hidden">Add a
todo item</label>
  <input id="add-todo" type="text" placeholder="E.g.
Adopt an owl">
  <button type="submit">Add</button>
</form>
```

(**Note:** “E.g...” can be used as a standard prefix for placeholders, making it clear they do not represent a pre-filled but *possible* value. Otherwise the user may be in danger of believe no action needs to be taken.)

Note

Placeholder styling

Be aware that some user agents (browsers) provide very feint placeholder text, in grey, which can lead to a failure under WCAG 1.4.3 Contrast (Minimum).

I recommend you style placeholders to have a higher contrast cross-browser and use an additional style — such as italicization in the pictured example — to help differentiate it from user-entered text.

```
::-webkit-input-placeholder {
  color: #444;
  font-style: italic;
}
::-moz-placeholder {
  color: #444;
  font-style: italic;
}
:-ms-input-placeholder {
  color: #444;
  font-style: italic;
}
:-moz-placeholder {
  color: #444;
  font-style: italic;
}
```

The independent blocks are regrettably necessary because each browser has trouble parsing other browsers' proprietary selectors.

Submission behavior

One of the advantages of using a `<form>` with a button of the `submit`

`type` is that the user can submit by pressing the button directly, or by hitting Enter. Even users who do not rely exclusively on the keyboard to operate the interface may like to hit Enter because it's quicker. What makes interaction possible for some, makes it better for others. That's inclusion.

If the user tries to submit an invalid entry we need to stop them. By disabling the `<button>` until the input is valid, submission by click or by Enter is suppressed. In fact, the `type="submit"` button stops being focusable by keyboard. In addition to disabling the button, we provide `aria-invalid="true"` to the input. Screen readers will tell their users the input is invalid, letting them know they need to change it.

Be wary of disabling buttons in this way. Although the context and state make things clear in this case, making buttons unfocusable can often mean they are missed — especially since screen reader users will mist likely be navigating by Tab within a form. It's usually better to let users focus and press the button, then let them know if there are any errors.

```
<form>
  <input type="text" aria-invalid="true" aria-
label="Write a new todo item" placeholder="E.g. Adopt
an owl">
  <button type="submit" disabled>Add</button>
</form>
```

Feedback

The deal with human-computer interaction is that when one party does something, the other party should respond. It's only polite. For most users, the response on the part of the computer to adding an item is implicit: they simply see the item being added to the page. If it's

possible to *animate* the appearance of the new item, all the better: Some movement means its arrival is less likely to be missed.

For users who are not sighted or are not using the interface visually, nothing would seem to happen. They remain focused on the input, which offers nothing new to be announced in screen reader software. Silence.

Moving focus to another part of the page — the newly added todo, say — would cause that element to be announced. But we don't want to move the user's focus because they might want to forge ahead writing more todos. Instead we can use a live region.

The feedback live region

Live regions are elements that tell screen readers to announce their contents *whenever those contents change*. With a live region, we can make screen readers talk to their users without making those users perform any action (such as moving focus).

Basic live regions are defined by `role="status"` or the equivalent `aria-live="polite"`. To maximize compatibility with different screen readers, you should use both. It may feel redundant, but it increases your audience.

```
<div role="status" aria-live="polite">
  <!-- add content to hear it spoken -->
</div>
```

(**Note:** A more complete introduction to ARIA live regions is covered in the Notifications chapter)

On the submit event, I can simply append the feedback to the live

region and it will be immediately announced to the screen reader user. Here's a very basic function that might do that for us:

```
const liveRegion =
document.querySelector('[role="status"]');

function addedFeedback(todoName) {
  liveRegion.textContent = `${todoName} added.`;
}
```

One of the simplest ways to make your web application more accessible is to wrap your status messages in a live region. Then, when they appear visually, they are also announced to screen reader users.

It's conventional to color code status messages. This "success" message is green, for example. But it's important to not rely on color, lest you let down color blind users. Hence, a supplemental tick icon is provided.

Inclusion is all about different users getting an **equivalent experience**, not necessarily the same experience. Sometimes what works for one user is meaningless, redundant, or obstructive to another.

In this case, the status message is not really needed visually because the item can be seen joining the list. In fact, adding the item to the list and revealing a status message at the same time would be to pull the user's attention in two directions. In other words: the visible appending of the item and the announcement of "[item name] added" are already equivalent.

In which case, we can hide this particular messaging system from view, with a `visually-hidden` class.

```
<div role="status" aria-live="polite" class="visually-
hidden">
  <!-- add content to hear it spoken -->
</div>
```

As explored previously, this utility class uses some magic to make sure the element(s) in question are not visible or have layout, but are still detected and announced in screen readers. Here's what it looks like:

```
.visually-hidden {
  position: absolute !important;
  clip: rect(1px, 1px, 1px, 1px) !important;
  padding:0 !important;
  border:0 !important;
  height: 1px !important;
  width: 1px !important;
  overflow: hidden !important;
}
```

Checking off todo items

Unlike in the previous toggle button post, this time checkboxes feel like the semantically correct way to activate and deactivate. You don't *press* or *switch off* todo items; you *check them off*.

Luckily, checkboxes let us do that with ease — the behavior comes out-of-the-box. We just need to remember to label each instance. When

iterating over the checkbox data, we can write unique values to each `for`/`id` pairing using a for loop's current index and string interpolation. Here's how it can be done in Vue.js:

```
<ul>
  <li v-for="(todo, index) in todos">
    <input type="checkbox" :id="`todo-${index}`" v-
model="todo.done">
    <label :for="`todo-${index}`">{{todo.name}}</label>
  </li>
</ul>
```

(**Note:** In this example, we imagine that each todo has a `done` property, hence `v-model="todo.done"` which automatically checks the checkbox where it evaluates as true.)

The line-through style

Making robust and accessible components is easy when you use semantic elements as they were intended. In my version, I just add a minor enhancement: a `line-through` style for checked items. This is applied to the `<label>` via the `:checked` state using an adjacent sibling combinator.

```
:checked + label {
  text-decoration: line-through;
}
```

Once again, I'm leveraging implicit state to affect style. No need for adding and removing `class="crossed-out"` or similar.

(**Note:** If you want to style the checkbox controls themselves, WTF Forms gives guidance on doing so without having to create custom elements.)

Deleting todo items

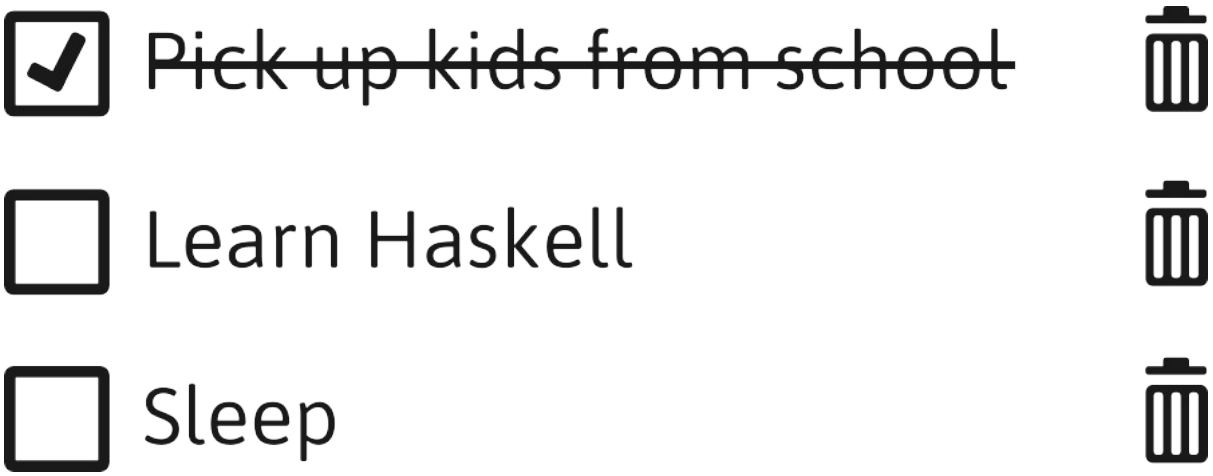
Checking off and deleting todo list items are distinct actions. Because sometimes you want to see which items you've done, and sometimes you add todo items to your list that you didn't mean to, or which become non-applicable.

The functionality to delete todos can be provided via a simple button. No need for any special state information — the label tells us everything we need to know.

In our case, a dustbin icon is provided using SVG. SVG is great because it's an image format that scales without degrading. Many kinds of users often feel the need to scale/zoom interfaces, including the short-sighted and those with motor impairments who are looking to create larger touch or click targets.

SVGs are not interpreted by screen readers unless they are explicitly given a label. We just need to provide a label alongside the SVG, again using a `visually-hidden` `<span>`.

```
<button>
  <svg>
    <use xlink:href="#bin-icon"></use>
  </svg>
  <span class="visually-hidden">delete {{todo.name}}
</span>
</button>
```



To reduce bloat when using multiple instances of the same inline SVG icon, we employ the `<use>` element, which refers to a canonical version of the SVG, defined as a `<symbol>` at the head of the document body:

```
<body>
  <svg style="display: none">
    <symbol id="bin-icon" viewBox="0 0 20 20">
      <path d="[path data here]">
    </symbol>
  </svg>
```

A bloated DOM can diminish the experience of many users since many operations will take longer. Assistive technology users especially may find their software unresponsive.

Note

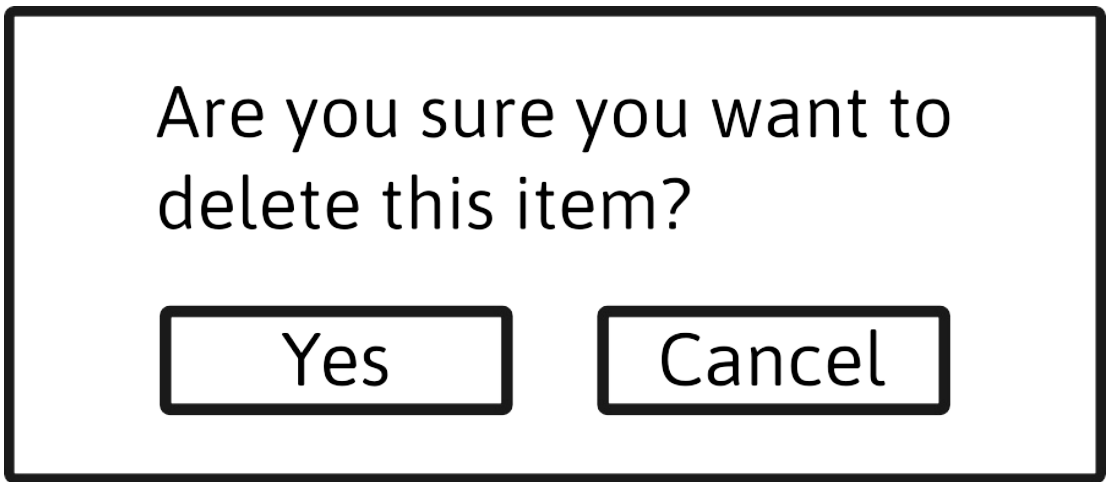
Inclusive icons

The enhanced usability offered to interfaces by icons is contested. Certainly,

in combination with text, they can help comprehension — especially for those who have low literacy or who are not reading the interface in their first language. However, any icons offered without supplementary text risk being misapprehended.

Does my dustbin icon really say *delete*? Could I make a better icon, which looks more like a dustbin? I could try something different, like using a cross icon (as in TodoMVC's implementation) but isn't that more associated with a *closing* action? There are no easy answers, so testing with real users is your best bet.

Fortunately, the accidental deletion of a todo item is not really a critical mistake, so users can safely find out what the icon means through trial and error. Where deletion *is* critical, a confirmation dialog should be provided, acting as both an explanation and a means to complete the action.



Note that the design of custom dialogs is not covered by this chapter. See Modal Dialogs for more information on both native and custom dialog

implementations.

Focus management

When a user clicks the delete button for a todo item, the todo item — including the checkbox, the label, and **the delete button itself** — will be removed from the DOM. This raises an interesting problem: what happens to focus when you delete the currently focused element?

Unless you're careful, the answer is *something very annoying* for keyboard users, including screen reader users.

The truth is, browsers don't know where to place focus when it has been destroyed in this way. Some maintain a sort of "ghost" focus where the item used to exist, while others jump to focus the next focusable element. Some flip out completely and default to focusing the outer document — meaning keyboard users have to crawl through the DOM back to where the removed element was.

For a consistent experience between users, we need to be deliberate and `focus()` an appropriate element, but which one?

One option is to focus the first checkbox of the list. Not only will this announce the checkbox's label and state, but also the total number of list items remaining: one fewer than a moment ago. All useful context.

```
document.querySelector('ul input').focus();
```

(**Note:** `querySelector` returns the *first* element that matches the selector. In our case: the first checkbox in the todo list.)

But what if we just deleted the last todo item in our list and had returned to the empty state? There's no checkbox we can focus. Let's try something else. Instead, I want to do two things:

1. Focus the region's "My Todo List" heading
2. Use the live region already instated to provide some feedback

You should never make non-interactive elements like headings focusable by users because the expectation is that, if they're focusable, they should actually *do* something. When I'm testing an interface and there are such elements, I would therefore fail it under WCAG 2.4.3 Focus Order.

However, sometimes you need to direct a user to a certain part of the page, via a script. In order to move a user to a heading and have it announced, you need to do two things:

1. Provide that heading with `tabindex="-1"`
2. Focus it using the `focus()` method in your script

```
<h1 tabindex="-1">My Todo List</h1>
```

The `-1` value's purpose is twofold: it makes elements unfocusable by users (including normally focusable elements) but makes them focusable by JavaScript. In practice, we can move a user to an inert element without it becoming a "tab stop" (an element that can be moved to via the Tab key) among focusable elements within the page.

In addition, focusing the heading will announce its text, role, level, and (in some screen readers) contextual information such as "region". At the very least, you should hear *"My Todo List, heading, level 2"*. Because it is in focus, pressing tab will step the user back inside the list and onto the first checkbox. In effect, we're saying, *"now that you've deleted that list item, here's the list again."*

I typically do not supply focus styles to elements which are focused programmatically in this way. Again, this is because the target element is not interactive and should not appear to be so.

```
[tabindex="-1"] { outline: none }
```

After the focused element (and with it its focus style) has been removed, the heading is focused. A keyboard user can then press Tab to find themselves on that first checkbox or — if there are no items remaining — the text input at the foot of the component.

The feedback

Arguably, we've provided enough information for the user and placed them in a perfect position to continue. But it's always better to be explicit. Since we already have a live region instated, why not use that to tell them the item has been successfully removed?

```
function deletedFeedback(todoName) {
  liveRegion.textContent = `${todoName} deleted.`;
}
```

I appreciate that you probably wouldn't be writing this in vanilla JavaScript, but this is basically how it would work.

Now, because we've used `role="status"` (`aria-live="polite"`), something neat happens in supporting screen readers: *"My Todo List, heading, level 2"* is read first, followed by *"[todo item name] deleted"*.

That's because *polite* live regions wait until the interface and the user have settled before making themselves known. Had I used `role="alert"` (`aria-live="assertive"`), the status message would override (or partially override) the focus-invoked heading announcement. Instead, the user knows both where they are, and that what they've tried to do has succeeded.

Working demo

I've created a demo to demonstrate the techniques in this post. It uses Vue.js, but could have been created with any JavaScript framework. It's offered for testing with different screen reader and browser combinations.

Demo: The Todo list, implemented in Vue.js

Conclusion

Counting semantic structure, labeling, iconography, focus management and feedback, there's quite a lot to consider when creating an inclusive todo list component. If that makes inclusive design seem dauntingly complex, consider the following:

1. This is new stuff. Don't worry, it'll become second nature soon enough.
2. Everything you've learned here is applicable to a wide range of content management components, and many other components.
3. You only need to build a rock solid component once. Then it can live in your pattern library and be reused indefinitely.

Checklist

- Give every major component, like this one, a well-written heading.
- Only provide "screen reader only" input labels if something else labels the input visually. Placeholders don't count.
- When you remove a focused element from the DOM, focus an appropriate nearby element with `focus()`.
- Consider the wording of empty states carefully. They introduce new users to your functionality.

Menus & Menu Buttons

Classification is hard. Take crabs, for example. Hermit crabs, porcelain crabs, and horseshoe crabs are not — taxonomically speaking — *true* crabs. But that doesn't stop us using the "crab" suffix. It gets more confusing when, over time and thanks to a process called *carcinisation*, untrue crabs evolve to resemble true crabs more closely. This is the case with king crabs, which are believed to have been hermit crabs in the past. Imagine the size of their shells!

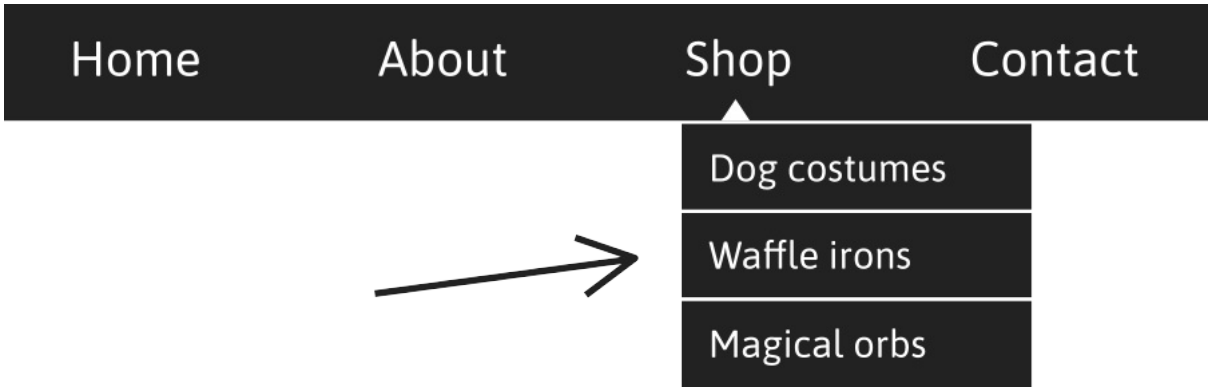
In design, we often make the same mistake of giving different things the same name. They *appear* similar, but appearances can be deceptive. This can have an unfortunate effect on the clarity of your component library. In terms of inclusion, it may also lead you to repurpose a semantically and behaviorally inappropriate component. Users will expect one thing and get another.

The term "dropdown" names a classic example. Lots of things "drop down" in interfaces, including the set of `<option>`s from a `<select>` element, and the JavaScript-revealed list of links that constitute a navigation submenu. Same name; quite different things. (Some people

call these “pulldowns”, of course, but let’s not get into that.)

Dropdowns which constitute a set of options are often called “menus”, and I want to talk about these here. We shall be devising a *true* menu, but there’s plenty to be said about not-really-true menus along the way.

Let’s start with a quiz. Is the box of links hanging down from the navigation bar in the illustration a menu?



The answer is no, not a true menu.

It’s a longstanding convention that navigation schemas are composed of lists of links. A convention nearly as longstanding dictates that sub-navigation should be provided as *nested* lists of links. If I were to remove the CSS for the component illustrated above, I should see something like the following, except colored blue and in Times New Roman.

- Home
- About
- Shop

    - Dog costumes
    - Waffle irons
    - Magical orbs
- Contact

Semantically speaking, nested lists of links are correct in this context. Navigation systems are really **tables of content** and this is how tables of content are structured. The only thing that really makes us think "menu" is the styling of the nested lists and the way they are revealed on hover or focus.

That's where some go wrong and start adding WAI-ARIA semantics: `aria-haspopup="true"`, `role="menu"`, `role="menuitem"` etc. There is a place for these, as we'll cover, but not here. Here are two reasons why:

1. ARIA menus are not designated for navigation but for application behavior. Imagine the menu system for a desktop application.
2. The top-level link should be usable *as a link*, meaning it does not behave like a menu button.

Regarding (2): When traversing a navigation region with submenus, one would expect each submenu to appear upon hovering or focusing the "top level" link ("Shop" in the illustration). This both reveals the submenu and places its own links in focus order. With a little help from JavaScript capturing focus and blur events to persist the appearance of the submenus while needed, someone using the keyboard should be able to tab through each link of each tier, in turn.

Menu buttons which take the `aria-haspopup="true"` property do not behave like this. They are activated on *click* and have no other purpose than to reveal a secreted menu.

`aria-expanded="false"` `aria-expanded="true"`

As pictured, whether that menu is open or closed should be communicated with `aria-expanded`. You should only change this state on click, not on focus. Users do not usually expect an explicit change of state on a mere focus event. In our navigation system, state doesn't really change; it's just a styling trick. Behaviorally, we can Tab through the navigation as if no such show/hide trickery were occurring.

The problem with navigation submenus

Navigation submenus (or "dropdowns" to some) work well with a mouse or by keyboard, but they're not so hot when it comes to touch. When you press the top-level "Shop" link in our example for the first time, you are telling it to both open the submenu and follow the link.

There are two possible resolutions here:

1. Prevent the default behavior of top-level links (`e.preventDefault()`) and script in full WAI-ARIA menu semantics and behavior.

2. Make sure each top-level destination page has a table of contents as an alternative to the submenu.

1. is unsatisfactory because, as I noted previously, these kinds of semantics and behaviors are not expected in this context, where links are the subject controls. Plus users could no longer navigate to a top-level page, if it exists.

Note

Which devices are touch devices?

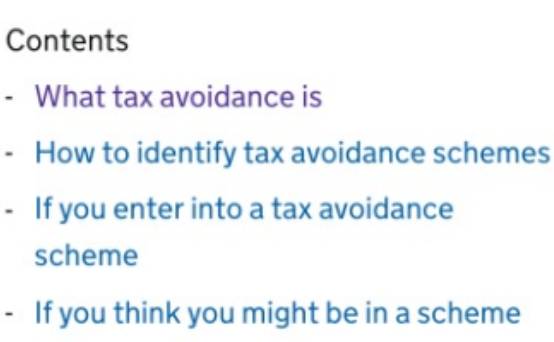
It's tempting to think, "this isn't a great solution, but I'll only add it for touch interfaces". The problem is: how does one detect if a device has a touch screen?

You certainly shouldn't equate "small screen" with "touch activated". Having worked in the same office as folks making touch displays for museums, I can assure you that some of the largest screens around are touch screens. Dual keyboard and touch input laptops are becoming increasingly prolific too.

By the same token, many but not all smaller devices are touch devices. In inclusive design, you cannot afford to make assumptions.

Resolution (2) is more inclusive and robust in that it provides a "fallback" for users of all inputs. But the scare quotes around the fallback term here are quite deliberate because I actually think in-page tables of content are a *superior* way of providing navigation.

The award winning Government Digital Services team would appear to agree. You may also have seen them on Wikipedia.



Tables of content

Tables of content are navigation for related pages or page sections and should be semantically similar to main site navigation regions, using a `<nav>` element, a list, and a group labeling mechanism.

```
<nav aria-labelledby="sections-heading">
  <h2 id="sections-heading">Products</h2>
  <ul>
    <li><a href="/products/dog-costumes">Dog
costumes</a></li>
    <li><a href="/products/waffle-irons">Waffle
irons</a></li>
    <li><a href="/products/magical-orbs">Magical
orbs</a></li>
  </ul>
</nav>
<!-- each section, in order, here -->
```

Notes

- In this example, we're imagining that each section is its own page, as it would have been in the dropdown submenu.

- It's important that each of these "Shop" pages has the same structure, with this "Products" table of content present in the same place. Consistency supports understanding.
- The list groups the items and enumerates them in assistive technology output, such as a screen reader's synthetic voice
- The `<nav>` is recursively labeled by the heading using `aria-labelledby`. This means "products navigation" will be announced in most screen readers upon entering the region by **Tab**. It also means that "products navigation" will be itemized in screen reader element interfaces, from which users can navigate to regions directly.

All on one page

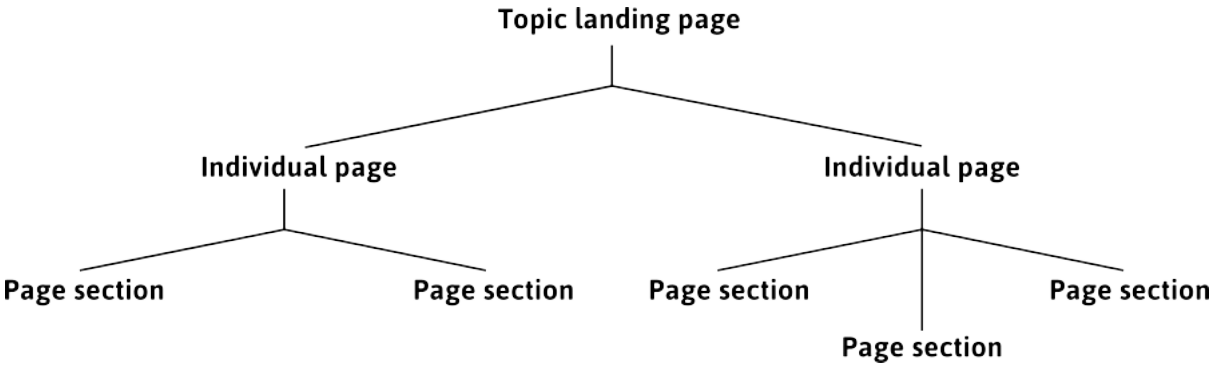
If you can fit all the sections onto one page without it becoming too long and arduous to scroll, even better. Just link to each section's hash identifier. For example, `href="#waffle-irons"` should point to `id="waffle-irons"`.

```
<nav aria-labelledby="sections-heading">
  <h2 id="sections-heading">Products</h2>
  <ul>
    <li><a href="#dog-costumes">Dog costumes</a></li>
    <li><a href="#waffle-irons">Waffle irons</a></li>
    <li><a href="#magical-orbs">Magical orbs</a></li>
  </ul>
</nav>
<!-- dog costumes section here -->
<section id="waffle-irons" tabindex="-1">
  <h2>Waffle Irons</h2>
</section>
<!-- magical orbs section here -->
```

(**Note:** Some browsers are poor at actually sending focus to linked page fragments. Placing `tabindex="-1"` on the target fragment fixes this. Although `tabindex="-1"` is more typically used with JavaScript and the `focus()` method, it is not needed in this case.)

Where a site has a lot of content, a carefully constructed information architecture, expressed through the liberal use of tables of content "menus" is infinitely preferable to a precarious and unwieldy dropdown system. Not only is it easier to make responsive, and requires less code to do so, but it makes things clearer: where dropdown systems hide structure away, tables of content lay it bare.

Some sites, including the Government Digital Service's gov.uk, include index (or "topic") pages that are *just* tables of content. It's such a powerful concept that the popular static site generator Hugo generates such pages by default.



Information architecture is a big part of inclusion. A badly organized site can be as technically compliant as you like, but will still alienate lots of users — especially those with cognitive impairments or those who are pressed for time.

Navigation menu buttons

While we're on the subject of faux navigation-related menus, it'd be remiss of me not to talk about navigation menu buttons. You've almost certainly seen these denoted by a three-line "hamburger" or "navicon" icon.

Even with a pared down information architecture and only one tier of navigation links, space on small screens is at a premium. Hiding navigation behind a button means there's more room for the main content in the viewport.

A navigation button is the closest thing we've studied so far to a *true* menu button. Since it has the purpose of toggling the availability of a menu on click, it should

1. Identify itself as a button, not a link;
2. Identify the expanded or collapsed state of its corresponding menu (which, in strict terms, is just a list of links).

Progressive enhancement

But let's not get ahead of ourselves. We ought to be mindful of progressive enhancement and consider how this would work without JavaScript.

In an unenhanced HTML document there's not a lot you can do with buttons (except submit buttons but that's not even closely related to what we want to achieve here). Instead, perhaps we should start with just a link which takes us to the navigation?

```
<a href="#navigation">navigation</a>
<!-- some content here perhaps -->
<nav id="navigation">
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/shop">Shop</a></li>
    <li><a href="/content">Content</a></li>
  </ul>
</nav>
```

There's not a lot of point in having the link unless there's a lot of content between the link and the navigation. Since site navigation should almost always appear near the top of the source order, there's no need. So, really, a navigation menu in the absence of JavaScript should just be... some navigation.

```
<nav id="navigation">
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/shop">Shop</a></li>
    <li><a href="/content">Content</a></li>
  </ul>
</nav>
```

You enhance this by adding the button, in its initial state, and hiding the navigation (using the `hidden` attribute):

```
<nav id="navigation">
  <button aria-expanded="false">Menu</button>
  <ul hidden>
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/shop">Shop</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

Some older browsers — you know which ones — don't support `hidden`, so remember to put the following in your CSS. It fixes the problem because `display: none` has the same effect of hiding the menu from assistive technologies and removing the links from focus order.

```
[hidden] {
  display: none;
}
```

Doing one's best to support older software is, of course, an act of inclusive design. Some are unable or unwilling to upgrade.

Placement

Where a lot of people go wrong is by placing the button *outside* the region. This would mean screen reader users who move to the `<nav>` using a shortcut would find it to be empty, which isn't very helpful. With the list hidden from screen readers, they'd just encounter this:

```
<nav id="navigation"></nav>
```

Here's how we might toggle state:

```
const navButton = document.querySelector('nav button');

navButton.addEventListener('click', function() {
  let expanded = this.getAttribute('aria-expanded') ===
'true';
  this.setAttribute('aria-expanded', !expanded);
  let menu = this.nextElementSibling;
  menu.hidden = !menu.hidden;
});
```

(**Note:** We are not using an arrow function here, because we don't want `this` to refer to the parent context.)

aria-controls

As I wrote in Aria-controls Is Poop, the `aria-controls` attribute, intended to help screen reader users navigate from a controlling element to a controlled element, is only supported in the JAWS screen reader. So you simply can't rely on it.

Without a good method for directing users between elements, you should instead make sure one of the following is true:

1. The expanded list's first link is next in focus order after the button (as in the previous code example).

2. The first link is focused programmatically upon revealing the list.

In this case, I would recommend (1). It's a lot simpler since you don't have to worry about moving focus back to the button and on which event(s) to do so. Also, there's currently nothing in place to warn users that their focus will be moved to somewhere different. In the true menus we'll be discussing shortly, this is the job of `aria-haspopup="true"`.

Employing `aria-controls` doesn't really do much harm, except that it makes readout in screen readers more verbose. However, some JAWS users may expect it. Here is how it would be applied, using the list's `id` as the cipher:

```
<nav id="navigation">
  <button aria-expanded="false" aria-controls="menu-
list">Menu</button>
  <ul id="menu-list" hidden>
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/shop">Shop</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

The menu and menuitem roles

A *true* menu (in the WAI-ARIA sense) should identify itself as such using the `menu` role (for the container) and, typically, `menuitem` children (other child roles may apply). These parent and child roles work together to provide information to assistive technologies. Here's how a list might be augmented to have menu semantics:

```
<ul role="menu">
  <li role="menuitem">Item 1</li>
  <li role="menuitem">Item 2</li>
  <li role="menuitem">Item 3</li>
</ul>
```

Since our navigation menu is beginning to behave somewhat like a "true" menu, should these not be present?

The short answer is: no. The long answer is: no, because our list items contain links and `menuitem` elements are not intended to have interactive descendants. That is, they *are* the controls in a menu.

We could, of course, suppress the list semantics of the `<li>`s using `role="presentation"` or `role="none"` (which are equivalent) and place the `menuitem` role on each link. However, this would suppress the implicit link role. In other words, the example to follow would be announced as "Home, menu item", *not* "Home, link" or "Home, menu item, link". ARIA roles simply override HTML roles.

```
<!-- will be read as "Home, menu item" -->
<li role="presentation">
  <a href="/" role="menuitem">Home</a>
</li>
```

We want the user to know that they are using a link and can expect link behavior, so this is no good. Like I said, true menus are for (JavaScript driven) application behavior.

What we're left with is a kind of hybrid component, which isn't quite a

true menu but at least tells users whether the list of links is open, thanks to the `aria-expanded` state. This is a perfectly satisfactory pattern for navigation menus.

The `<select>` element

If you've been involved in responsive design from the beginning, you may remember a pattern whereby navigation was condensed into a `<select>` element for narrow viewports.

As with the checkbox-based toggle buttons we discussed, using a native element that behaves somewhat as intended without additional scripting is a good choice for efficiency and — especially on mobile — performance. And `<select>` elements *are* menus of sorts, with similar semantics to the button-triggered menu we shall soon be constructing.

However, just as with the checkbox toggle button, we're using an element associated with entering input, not simply making a choice. This is likely to cause confusion for many users — especially since this pattern uses JavaScript to make the selected `<option>` behave like a link. The unexpected change of context this elicits is considered a failure according to WCAG's 3.2.2 On Input (Level A) criterion.

True menus

Now that we've had the discussion about false menus and quasi-menus, the time has arrived to create a *true* menu, as opened and closed by a true menu button. From here on in I will refer to the button and menu together as simply a "menu button".

But in what respects will our menu button be true? Well, it'll be a menu component intended for choosing options in the subject application, which implements all the expected semantics and corresponding behaviors to be considered conventional for such a tool.

As mentioned already, these conventions come from desktop application design. ARIA attribution and JavaScript governed focus management are needed to imitate them fully. Part of the purpose of ARIA is to help web developers create rich web experiences without breaking with usability conventions forged in the native world.

In this example, we'll imagine our application is some sort of game or quiz. Our menu button will let the user choose a difficulty level. With all the semantics in place, the menu looks like this:

```
<button aria-haspopup="true" aria-expanded="false">
  Difficulty
  <span aria-hidden="true">&#x25be;</span>
</button>
<div role="menu">
  <button role="menuitem">Easy</button>
  <button role="menuitem">Medium</button>
  <button role="menuitem">Incredibly Hard</button>
</div>
```

Notes

- The `aria-haspopup` property simply indicates that the button secretes a menu. It acts as warning that, when pressed, the user will be moved to the "popup" menu (we'll cover focus behavior shortly). Its value does not change — it remains as `true` at all times.
- The `<span>` inside the button contains the unicode point for a black down-pointing small triangle. This convention indicates visually what `aria-haspopup` does non-visually — that pressing the button will reveal something below it. The `aria-hidden="true"` attribution prevents screen readers from announcing *"down pointing triangle"* or similar. Thanks to `aria-haspopup`, it's not needed in the non-visual context.
- The `aria-haspopup` property is complemented by `aria-expanded`. This tells the user whether the menu is currently in an open (expanded) or closed (collapsed) state by toggling between `true` and `false` values.
- The menu itself takes the (aptly named) `menu` role. It takes descendants with the `menuitem` role. They do not need to be direct children of the `menu` element, but they are in this case — for simplicity.

Keyboard and focus behavior

When it comes to making interactive controls keyboard accessible, the best thing you can do is use the right elements. Because we're using `<button>` elements here, we can be assured that click events will fire on **Enter** and **Space** keystrokes, as specified in the HTMLButtonElement interface. It also means that we can disable the menu items using the button-associated `disabled` property.

There's a lot more to menu button keyboard interaction, though. Here's a summary of all the focus and keyboard behavior we're going to implement, based on WAI-ARIA Authoring Practices 1.1:

Enter, Space or ↓ on the menu button	Opens the menu
↓ on a menu item	Moves focus to the next menu item, or the first menu item if you're on the last one
↑ on a menu item	Moves focus to the previous menu item, or the last menu item if you're on the first one
↑ on the menu button	Closes the menu if open
Esc on a menu item	Closes the menu and focuses the menu button

The advantage of moving focus between menu items using the arrow keys is that Tab is preserved for moving out of the menu. In practice, this means users don't have to move through every menu item to exit the menu — a huge improvement for usability, especially where there are many menu items.

The application of `tabindex="-1"` makes the menu items unfocusable by Tab but preserves the ability to focus the elements programmatically, upon capturing key strokes on the arrow keys.

```
<button aria-haspopup="true" aria-expanded="false">
  Difficulty
  <span aria-hidden="true">&#x25be;</span>
</button>
<div role="menu">
  <button role="menuitem" tabindex="-1">Easy</button>
  <button role="menuitem" tabindex="-1">Medium</button>
  <button role="menuitem" tabindex="-1">Incredibly
Hard</button>
</div>
```

The open method

As part of a sound API design, we can construct methods for handling the various events.

For example, the `open` method needs to switch the `aria-expanded` value to “true”, change the menu’s hidden property to `false`, and focus the first `menuitem` in the menu that isn’t disabled:

```
MenuButton.prototype.open = function() {
  this.button.setAttribute('aria-expanded', true);
  this.menu.hidden = false;
  this.menu.querySelector(':not([disabled])').focus();

  return this;
};
```

We can execute this method where the user presses the down key on a focused menu button instance:

```
this.button.addEventListener(
  'keydown',
  function(e) {
    if (e.keyCode === 40) {
      this.open();
    }
  }.bind(this)
);
```

In addition, a developer using this script will now be able to open the menu programmatically:

```
exampleMenuButton = new
MenuButton(document.querySelector('[aria-haspopup]'));

exampleMenuButton.open();
```

Note

The checkbox hack

As much as possible, it's better not to use JavaScript unless you need to. Involving a third technology on top of HTML and CSS is necessarily an increase in systemic complexity and fragility. However, not all components can be satisfactorily built without JavaScript in the mix.

In the case of menu buttons, an enthusiasm for making them "work without JavaScript" has led to something called the checkbox hack. This is where the checked (or unchecked) state of a hidden checkbox is used to toggle the visibility of a menu element using CSS.

```
/* menu closed */
[type="checkbox"] + [role="menu"] {
  display: none;
}

/* menu open */
[type="checkbox"]:checked + [role="menu"] {
  display: block;
}
```

To screen reader users, the checkbox role and checked state are nonsensical in this context. This can be partly overcome by adding `role="button"` to the checkbox.

```
<input type="checkbox" role="button" aria-haspopup="true"
id="toggle">
```

Unfortunately, this suppresses the implicit checked state communication, depriving us of JavaScript-free state feedback (poor though it would have been as "checked" in this context).

But it *is* possible to spoof `aria-expanded`. We just need to supply our label with two spans as below.

```
<input type="checkbox" role="button" aria-haspopup="true"
id="toggle" class="visually-hidden">
<label for="toggle" data-opens-menu>
  Difficulty
  <span class="visually-hidden expanded-
text">expanded</span>
  <span class="visually-hidden collapsed-
text">collapsed</span>
  <span aria-hidden="true">&#x25be;</span>
</label>
```

These are both visually hidden using the `visually-hidden` class, but —

depending on which state we're in — only one is hidden to screen readers as well. That is, only one has `display: none`, and this is determined by the extant (but not communicated) checked state:

```
/* class to hide spans visually */
.visually-hidden {
  position: absolute;
  clip: rect(1px, 1px, 1px, 1px);
  clip-path: inset(100%);
  padding: 0;
  border: 0;
  height: 1px;
  width: 1px;
  overflow: hidden;
}

/* reveal the correct state wording to screen readers
based on state */
[type='checkbox']:checked + label .expanded-text {
  display: inline;
}

[type='checkbox']:checked + label .collapsed-text {
  display: none;
}

[type='checkbox']:not(:checked) + label .expanded-text {
  display: none;
}

[type='checkbox']:not(:checked) + label .collapsed-text {
  display: inline;
}
```

This is clever and all, but our menu button is still incomplete since the expected focus behaviors we've been discussing simply cannot be implemented without JavaScript.

These behaviors are conventional and expected, making the button more usable. However, if you really need to implement a menu button without

JavaScript, this is about as close as you can get. Considering the cut-down navigation menu button I covered previously offers menu content that is *not* JavaScript dependent itself (i.e. links), this approach may be a suitable option.

Demo: Just for fun, here's a demo of a hamburger menu that works without JavaScript

(**Note:** Only Space opens the menu.)

The "choose" event

Executing some methods should emit events so that we can set up listeners. For example, we can emit a `choose` event when a user clicks a menu item. We can set this up using `CustomEvent`, which lets us pass an argument to the event's `detail` property. In this case, the argument ("choice") would be the chosen menu item's DOM node.

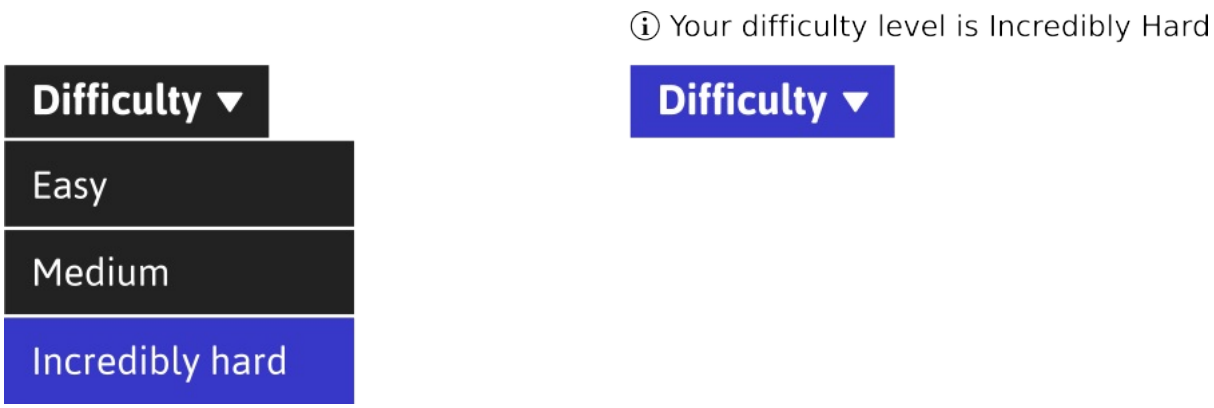
```
MenuButton.prototype.choose = function(choice) {
  // Define the 'choose' event
  var chooseEvent = new CustomEvent('choose', {
    detail: {
      choice: choice
    }
  });
  // Dispatch the event
  this.button.dispatchEvent(chooseEvent);

  return this;
};
```

There are all sorts of things we can do with this mechanism. Perhaps we have a live region set up with an `id` of `menuFeedback`:

```
<div role="alert" id="menuFeedback"></div>
```

Now we can set up a listener and populate the live region with the information secreted inside the event:

```
// Save a reference to the live region
const liveRegion =
document.getElementById('menuFeedback');

exampleMenuButton.addEventListener('choose', e => {
  // Get the node's text content (label)
  let choiceLabel = e.details.choice.textContent;

  // Populate the live region
  liveRegion.textContent = `Your difficulty level is
${choiceLabel}`;
}):
```



When a user chooses an option, the menu closes and focus is returned to the menu button. It's important users are returned to the triggering element after the menu is closed.

When a menu item is selected, the screen reader user will hear, "*You chose [menu item's label]*". A live region (defined here with the `role="alert"` attribution) announces its content in screen readers whenever that content changes. The live region isn't mandatory, but it is an example of what might happen in the interface as a response to the user making a menu choice.

Persisting choices

Not all menu items are for choosing persistent settings. Many just act like standard buttons which make something in the interface happen when pressed. However, in the case of our difficulty menu button, we'd like to indicate which is the current difficulty setting — the one chosen last.

The `aria-checked="true"` attribute works for items that, instead of `menuitem`, take the `menuitemradio` role. The enhanced markup, with the second item checked (*set*) looks like this:

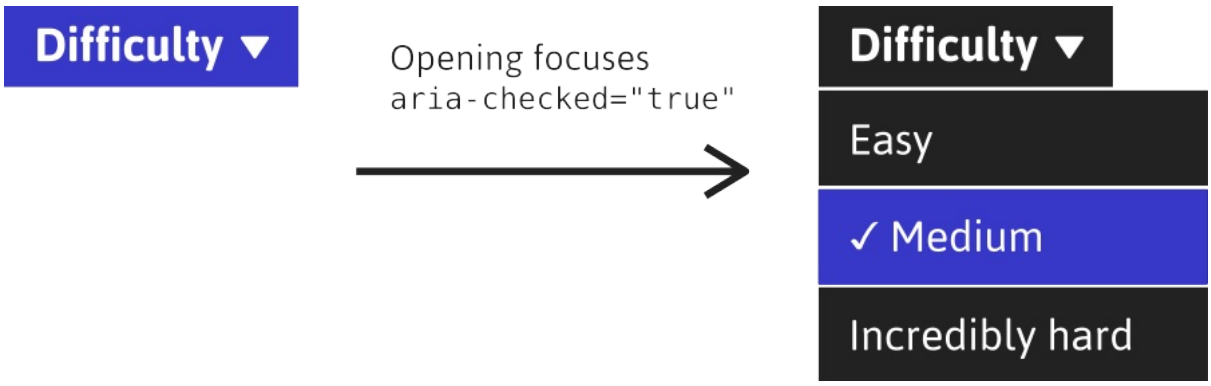
```
<button aria-haspopup="true" aria-expanded="false">
  Difficulty
  <span aria-hidden="true">&#x25be;</span>
</button>
<div role="menu">
  <button role="menuitemradio"
tabindex="-1">Easy</button>
  <button role="menuitemradio" aria-checked="true"
tabindex="-1">Medium</button>
  <button role="menuitemradio" tabindex="-1">Incredibly
Hard</button>
</div>
```

Native menus on many platforms indicate chosen items using check marks. We can do that with no trouble using a little extra CSS:

```
[role='menuitem'][aria-checked='true']::before {
  content: '\2713\0020';
}
```

While traversing the menu with a screen reader running, focusing this checked item will prompt an announcement like *"check mark, Medium menu item, checked"*.

The behavior on opening a menu with a checked `menuitemradio` differs slightly. Instead of focusing the first (enabled) item in the menu, the *checked* item is focused instead.



What's the benefit of this behavior? The user (any user) is reminded of their previously selected option. In menus with numerous incremental options (for example, a set of zoom levels), people operating by keyboard are placed in the optimal position to make their adjustment.

Using the menu button with a screen reader

Video: I'll show you what it's like to use the menu button with the Voiceover screen reader and Chrome. The example uses items with `menuitemradio`, `aria-checked` and the focus behavior discussed. Similar experiences can be expected across the gamut of popular screen reader software.

Inclusive Menu Button on Github

Hugo Giraudel and I have worked together on creating a menu button component with the API features I have described, and more. You have Hugo to thank for many of these features, since they were based on the

work they did on a11y-dialog — an accessible modal dialog. It is available on Github and NPM.

```
npm i inclusive-menu-button --save
```

In addition, Hugo has created a React version for your delectation.

Checklist

- Don’t use ARIA menu semantics in navigation menu systems.
- On content heavy sites, don’t hide structure away in nested dropdown-powered navigation menus.
- Use `aria-expanded` to indicate the open/closed state of a button-activated navigation menu.
- Make sure said navigation menu is next in focus order after the button that opens/closes it.
- Never sacrifice usability in the pursuit of JavaScript-free solutions. It’s vanity.

Tooltips & Toggletips

Tooltips — affectionately misnomered as “tootlips” by my friend Steve — are a precariously longstanding interface pattern. Literally “tips for tools”, they are little bubbles of information that clarify the purpose of otherwise ambiguous controls/tools. A common example is a control that is only represented by a cryptic icon, the meaning of which the user has yet to learn.

When and how these bubbles transpire is apparently up for debate, since I’ve encountered many a tooltip and they all seem to behave slightly differently. I’ve come to the conclusion that these numerous implementations fall into two distinct groups: true tooltips, and a pattern I’m hesitantly calling the “toggletip”, coined by the aforementioned Steve in some research and experimentation he did not long ago.

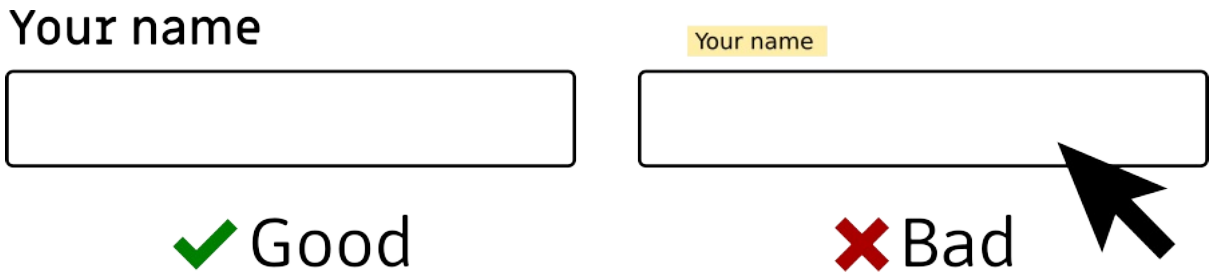
Inclusive design is often about providing the user with the right tool for the job, and the right kind of tooltip to go with that tool. In this article, I’ll be looking at situations which might call for a tooltip or else a toggletip, and formulating inclusive implementations for each.

The `title` attribute

We can't talk about tooltips without bringing up the `title` attribute: the HTML standard for providing contextual information bubbles. The Paciello Group blog pulls no punches in describing the `title` attribute's contribution to web interfaces:

"If you want to hide content from mobile and tablet users as well as assistive tech users and keyboard only users, use the `title` attribute."
— *The Paciello Group blog*

That's *pretty bad* in terms of inclusion. In fact, the only place I can think of where the `title` attribute works reliably in screen reader software is on form elements like `<input>`s. Even then, touch and keyboard users won't get to see the `title` message appear. In short: just provide a clearly worded, permanently visible label.



I'm a big supporter of using standard HTML elements and attributes wherever they're available. It's the most efficient and performant way to build usable web interfaces. But, despite being a specification mainstay, the `title` attribute really isn't fit for purpose.

Then again, we've yet to define that purpose. What *should* tooltips be used for? And even if we can design them to be usable by the many, do we need them at all?

A case for tooltips

As we already established, tooltips are for clarification; they are for providing *missing* information. But why should that information be missing in the first place? As I wrote in Inclusive Design Patterns, icons can accelerate the understanding of an interface, and help to internationalize it. But icons provided in isolation are always in danger of completely confounding a user — because they don't *spell anything out*. To users who don't recognize or can't decipher the icon, information is missing.

Most of the time, you should simply be providing text alongside icons. Like the perma-visible field labels I just mentioned, textual labels are the most straightforward way of labeling things and they're automatically screen reader accessible if provided as real text (not images of text).

The usual excuse for *not* providing textual labels is, "there's no space". And there likely isn't, if you set out not to include textual labels in the first place. If you treat them as important from the beginning, you will find a way.

There's always room for text if you make it, but some configurations leave more space for text than others.

Tooltips are a last resort, where space really is at a premium — perhaps due to the sheer number of controls, like in the toolbar for a WYSIWYG editor. So, how would we go about designing them to be as inclusive as possible?

Inclusive tooltips

The first thing to get right is making the text in the tooltip accessible to assistive technologies. There are a couple of different methods for associating the tooltip to the focused control, and we choose between them based on the specific role of that tooltip: Is the tooltip there to provide a primary label or an auxiliary clarification?

A notifications control with a tooltip reading “Notifications” treats the tooltip as a primary label. Alternatively, a tooltip that reads “View notifications and manage settings” is supplementary.

Tooltip as primary label

To associate one element with another as its primary label, you can use `aria-labelledby`. The relationship is forged by the `aria-labelledby` and `id` attributes sharing the same value.

```
<button class="notifications" aria-
labelledby="notifications-label">
  <svg><use xlink:href="#notifications-icon"></use>
</svg>
</button>
<div role="tooltip" id="notifications-
label">Notifications</div>
```

- Note the use of the `tooltip` role. In practical terms, all this role offers is an assurance that `aria-describedby` works reliably where supported. As Léonie Watson writes, ARIA labels and descriptions sometimes don't work with all elements unless you incorporate an appropriate role. In this case, the most important role is the implicit button role of the subject `<button>` element, but `role="tooltip"` may extend support for this labeling method in some software.
- Whatever text content is in the linked SVG, it won't be read out. The `aria-labelledby` association *supersedes* the text content of the button as the label.

To a screen reader — and its user — the above is now functionally similar to using a simple text label, like this:

```
<button class="notifications">Notifications</button>
```

Remember that text nodes are translatable by Google's and Microsoft's translation services, so using `aria-labelledby` to connect a text node to an element is better than using `aria-label` as an attribute with its own value.

The tooltip text is available on focus, just as it would be on hover for sighted users. In fact, if all text appeared only on hover, a sighted mouse user's experience of an interface would be somewhat analogous to that of a blind screen reader user.

Note

Redundant tooltips

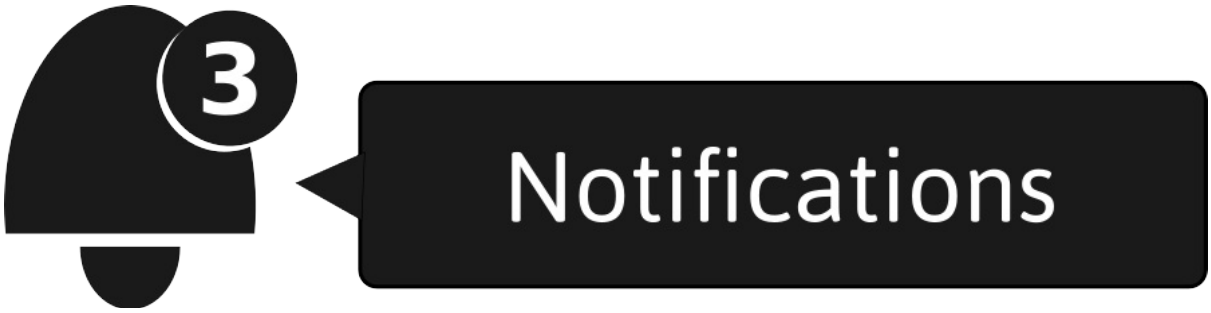
All the time as an interface design consultant, I see people providing `title` attributes to links with identical text nodes.

```
<a href="/some/path" title="Heydon's special
page">Heydon's special page</a>
```

Since the text node is already perfectly visible, this is completely redundant. It doesn't even add anything for screen readers except — in some cases — repetition.

Including notification count

What if the notifications button showed a count of unread notifications, as these things often do (I'm thinking, of course, of Twitter)?



Fortunately, `aria-labelledby` can accept multiple, space separated `id`s.

```
<button class="notifications" aria-
labelledby="notifications-count notifications-label">
  <svg><use xlink:href="#notifications-icon"></use>
</svg>
  <span id="notifications-count">3</span>
</button>
<div role="tooltip" id="notifications-
label">Notifications</div>
```

Despite the `#notifications-count` element appearing inside the `<button>`, it does not form the label by itself: It forms the first part of the label as the first `id` listed in the `aria-labelledby` value. It is placed where it is so that the designer can take advantage of relative and absolute positioning to arrange the element visually.

To screen reader users, the label is now “3 notifications”. This succinctly acts as both a current count of notifications and a reminder that this is the notifications control.

Tooltip as auxiliary description

Now let’s try setting up the tooltip as a supplementary description, which is the archetypal form. Like `<input>` placeholders, tooltips should be for added information and clarification.

Some interactive elements may have accessible descriptions, but all interactive elements need accessible labels. If we’re using `aria-describedby` to connect the tooltip text, we’ll need another method for providing the “Notifications” label. Instead of `aria-labelledby` we can add a visually hidden span to the button’s text node, alongside the

existing “3” counter.

```
<button class="notifications" aria-
describedby="notifications-desc">
  <svg><use xlink:href="#notifications-icon"></use>
</svg>
  <span id="notifications-count">3</span>
  <span class="visually-hidden">Notifications</span>
</button>
<div role="tooltip" id="notifications-desc">View and
manage notifications settings</div>
```

The `visually-hidden` class corresponds to some special CSS we’ve discussed before in this book. It hides the `<span>` visually without stopping it from being read out in screen readers.

```
.visually-hidden {
  clip-path: inset(100%);
  clip: rect(1px, 1px, 1px, 1px);
  height: 1px;
  overflow: hidden;
  position: absolute;
  white-space: nowrap;
  width: 1px;
}
```

The prescribed behavior of `aria-describedby` is to be announced as the last piece of information for the control, after the label and role. In this case: *“Notifications button… View and manage notifications settings”* (most screen readers will leave a pause before the description).

Interaction

To improve upon the notoriously awful `title` attribute, our custom tooltips should appear on focus as well as hover. By supplying the tooltip in an element adjacent to the button, we can do this with just CSS:

```
[role='tooltip'] {
  display: none;
}

button:hover + [role='tooltip'],
button:focus + [role='tooltip'] {
  display: block;
}
```

However, we may need to wrap the button and tooltip in a container element for positioning purposes:

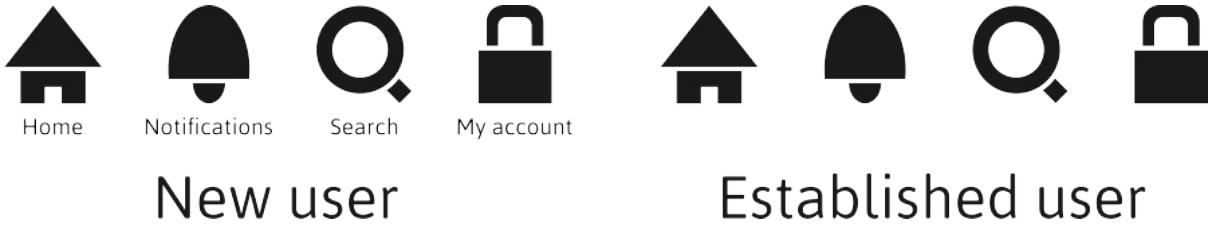
```
.button-and-tooltip {
  position: relative;
}

[role='tooltip'] {
  position: absolute;
  /* left/top/right/bottom values as required */
}
```

Touch interaction

So far this simply doesn't work so well for touch screen users because the focus and active states happen simultaneously. In practice, this means you'll see the tooltip, but only as the button is being pressed.

How *much* of a problem this is depends on the nature of the app to which the control belongs. How bad is it if the user presses the control without really knowing what it does the first time? How easily can they recover?

There are other things you could try, of course. One might be to suppress the button's action on the first press so it *just* shows the tooltip that time around. You could take this "tutorial mode" idea further still and show the tooltips as inline text for newcomers, but streamline the interface to just show icons for established users. By then, they should have learned what the icons represent.



In either case, the landing screen for each of the options should have a clear (`<h1>`) heading with the same wording as the labels. Then at least the user knows where the icon took them upon arrival.

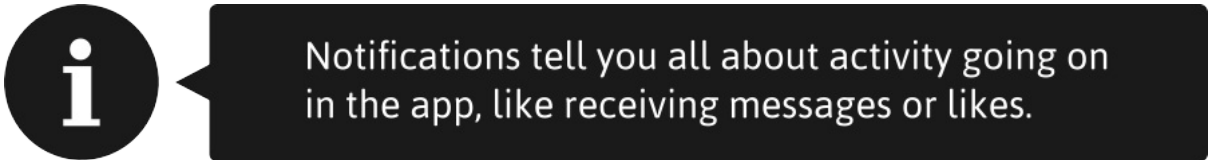
This would be to do away with tooltips altogether, which is probably for the best anyway. However, the tooltip's sister component — the toggletip — can work for mouse, keyboard *and* touch users.

Inclusive toggletips

Toggletips are like tooltips in the sense that they can provide

supplementary or clarifying information. However, they differ by making the control itself supplementary: toggletips exist to reveal information balloons, and serve no other purpose.

Often they take the form of little “i” icons:



To work by touch just as well as by mouse or keyboard, toggletips are revealed by click rather than by hover and focus. Crucially, this means the `aria-describedby` association is no longer appropriate. Why? Because a screen reader user would have access to the information before pressing the button, so pressing it would appear not to do anything. Technically, they have *access* to the information, making the control “accessible” — but the control simply wouldn’t make sense. In other words, it’s a user experience issue more than a strict accessibility one, but it’s important.

Toggletips with live regions

The trick is to make screen readers announce the information after the click event. This is a perfect use case for a live region. We can supply an empty live region, and populate it with the toggletip “bubble” when it is invoked. This will both make the bubble appear visually and cause the live region to announce the tooltip’s information.

To follow is the markup with the live region unpopulated. Note the `.tooltip-container` element, which is provided to help positioning. This element would have `position: relative`, allowing for absolutely positioning the generated `.toggletip-bubble` element nearby.

```
<span class="tooltip-container">
  <button type="button" data-toggletip-content="This
clarifies whatever needs clarifying">
    <span aria-hidden="true">i</span>
    <span class="visually-hidden">More info</span>
  </button>
  <span role="status"></span>
</span>
```

As in previous examples, I am silencing the “i” in screen readers with `aria-hidden`, and including a visually hidden span as a more descriptive label for non-visual users.

Note the `type="button"` attribution which is to stop some browsers mistaking the button as a submit button when placed inside forms. Here is the markup with the live region populated (after the toggletip button is clicked):

```
<span class="tooltip-container">
  <button type="button" data-toggletip-content="This
clarifies whatever needs clarifying">
    <span aria-hidden="true">i</span>
    <span class="visually-hidden">More info</span>
  </button>
  <span role="status">
    <span class="toggletip-bubble">This clarifies
whatever needs clarifying</span>
  </span>
</span>
```

The accompanying script and demo, with notes to follow:

```
(function() {
  // Get all the toggletip buttons
  const toggletips = document.querySelectorAll('[data-
toggletip-content]');

  // Iterate over them
  Array.prototype.forEach.call(toggletips, toggletip =>
{
    // Get the message from the data-content element
    var message = toggletip.getAttribute('data-
toggletip-content');

    // Get the live region element
    var liveRegion = toggletip.nextElementSibling;

    // Toggle the message
    toggletip.addEventListener('click', () => {
      liveRegion.innerHTML = '';
      window.setTimeout(function() {
        liveRegion.innerHTML =
          '<span class="toggletip-bubble">' + message +
'</span>';
      }, 100);
    });

    // Close on outside click
    document.addEventListener('click', e => {
      if (toggletip !== e.target) liveRegion.innerHTML
= '';
    });

    // Close on blur
    toggletip.addEventListener('blur', e => {
      liveRegion.innerHTML = '';
    });

    // Remove toggletip on ESC
    toggletip.addEventListener('keydown', e => {
```

```
      if ((e.keyCode || e.which) === 27)
liveRegion.innerHTML = '';
    });
  });
})();
```

Demo: Basic toggletip with live region

Notes

- Our button is not a toggle button — at least, not in the usual sense. Instead of clicking the button showing or hiding the bubble, it only shows it. Hiding the bubble is achieved by unfocusing the button, mouse clicking away from the button or pressing escape.
- When the button is clicked for a second (or third, fourth etc.) time, the live region is repopulated after a discreet interval, re-announcing the content in screen readers. This is simpler and more intuitive than implementing toggle states (a “message in on” state makes little sense, especially once it has already been read out).
- The “discreet interval” (see last item) is implemented using `setTimeout`. Without it, some setups are not able to register the repopulation of the live region and do not re-announce the contents.
- The `role="tooltip"` attribution is not applicable since we are using `role="status"` for the live region.

Progressively enhancing `title`

As discussed, the `title` attribute is *really* flaky. But it *does* at least provide an accessible label to some assistive technologies, available

when the button is focused. We could provide the bubble content via `title` and use this to build the `data-toggletip-content` attribute on page load. Our script's initial hook now becomes the boolean `data-toggletip`:

```
<button data-toggletip title="This clarifies whatever
needs clarifying">
  <span aria-hidden="true">i</span>
  <span class="visually-hidden">More info</span>
</button>
```

In the script, we need to take the value from `title` to build `data-tooltip-content`, then destroy `title` because we don't need it and it might still appear / get announced if left festering.

```
var toggletips = document.querySelectorAll('[data-
toggletip][title]');

Array.prototype.forEach(toggletips, toggletip => {
  var message = toggletip.getAttribute('title');
  toggletip.setAttribute('data-tooltip-content',
message);
  toggletip.removeAttribute('title');
});
```

Better progressive enhancement

A button that doesn't do anything and happens to have a title attribute isn't really a very good baseline. Instead, I would recommend displaying the toggletip's content inline and then enhancing by creating the toggletip button dynamically.

Demo: Toggletip with progressive enhancement

Tests and error messages

Something I haven't talked about in this book is writing tests, so let's do a little of that here. Don't worry, I don't mean unit tests.

If our toggletip component is to belong to a design system, it may be borrowed and used by lots of different people. By writing tests and including warnings, we can try to ensure it isn't being used improperly.

A toggletip button that isn't a `<button>` provides a deceptive role to assistive technologies and is not focusable by keyboard (unless it's another, inappropriate focusable element like a hyperlink). In our script, we can detect the element `nodeName` and return an error message if it is not `BUTTON`. We use `return` to stop the remainder of the IIFE (Immediately Invoked Function Expression) from executing.

```
if (toggletip.nodeName !== 'BUTTON') {
  console.error('Toggletip buttons need to be <button>
elements.');
  return;
}
```

```
▶Toggletip buttons need to be button elements.        VM1280 console_runner-079c09a….js:1
```

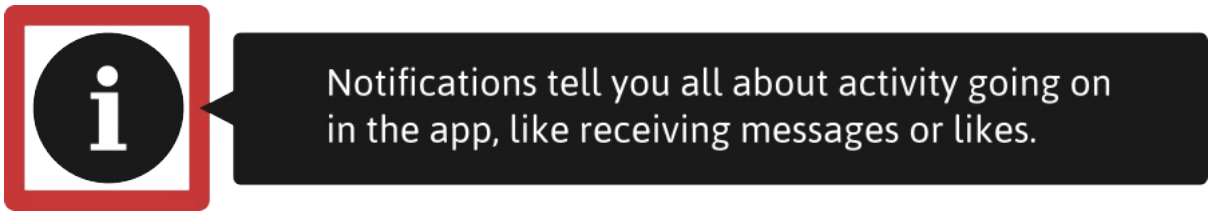
CSS tests and error messages

In Inclusive Design Patterns I write about creating deliberate visual regressions to highlight code errors, and providing error messages in the developer tools CSS inspector.

The error we caught with JavaScript earlier can be caught using the CSS selector `[data-tooltip]:not(button)`. We can highlight the erroneous element with a red outline, and provide an error message using the made-up `ERROR` property:

```
[data-tooltip]:not(button) {
  outline: red solid 0.5em;
  error: Toggletip buttons need to be <button>
elements.;
}
```

Despite being an invalid property, the `ERROR` will appear in dev tools when the element is inspected.



The clear red outline shows there is an error present and guides the developer's DOM inspector cursor.

Conclusion

Most of the time, tooltips shouldn't be needed if you provide clear textual labeling and familiar iconography. Most of the time toggletips are a complex way of providing information that could just be part of

the document's prose content. But I see each of these components all the time in auditing websites, so I wanted to provide some guidance on how to make the most of them.

Checklist

- If you have space, don't use tooltips or toggletips. Just provide clear labels and sufficient body text.
- If it's a tooltip you are looking to use, decide whether the tip's content should be provided as the label or description and choose ARIA properties accordingly.
- Don't rely on `title` attributes. They are not keyboard accessible and are not supported in many screen reader setups.
- Don't describe toggletips with `aria-describedby`. It makes the subject button's action redundant to screen reader users.
- Don't put interactive content such as close and confirm buttons or links in tooltips or toggletips. This is the job of more complex menu and dialog components.

A Theme Switcher

My mantra for building web interfaces is, “if it can’t be done efficiently, don’t do it at all.” In fact, I’ve preached about writing less damned code around the UK, Europe, and China. If a feature can only be achieved by taking a significant performance hit, the net effect is negative and the feature should be abandoned. That’s how critical performance is on the web.

Offering users choices over the display of your interface is friendly, so long as it isn’t intrusive. It helps to satisfy the Offer choice inclusive design principle. *However*, choices such as theme options are nice-to-haves and should only be implemented if it’s possible to do so efficiently.

Typically, alternative themes are offered as separate stylesheets that can be switched between using JavaScript. In some cases they represent a performance issue (because an override theme requires loading a lot of additional CSS) and in most cases they represent a maintenance issue (because separate stylesheets have to be kept up to date as the site is further developed).

One of the few types of alternative theme that adds real value to users is a low light intensity “night mode” theme. Not only is it easier on the eyes when reading in the dark, but it also reduces the likelihood of migraine and the irritation of other light sensitivity disorders. As a migraine sufferer, I’m interested!

In this article, I’ll be covering how to make an efficient and portable React component that allows users to switch a default light theme into “dark mode” and persist this setting using the `localStorage` API.

Given a light theme (predominantly dark text on light backgrounds) the most efficient course of action is not to provide a completely alternative stylesheet, but to augment the existing styles directly, as tersely as possible. Fortunately, CSS provides the `filter` property, which allows you to invert colors. Although this property is often associated with image elements, it can be used on any elements, including the root `<html>` element:

```
:root {
  filter: invert(100%);
}
```

(**Note:** Some browsers support `invert()` as a shorthand, but not all, so write out `100%` for better support.)

The only trouble is that `filter` can only invert *stated* colors. Therefore, if the element has no background color, the text will invert but the implicit (white) background will remain the same. The result? Light text on a light background.

This is easily fixed by stating a light `background-color`.

```
:root {
  background-color: #fefefe;
  filter: invert(100%);
}
```

But we may still run into problems with child elements that also have no stated background color. This is where CSS's `inherit` keyword comes in handy.

```
:root {
  background-color: #fefefe;
  filter: invert(100%);
}

* {
  background-color: inherit;
}
```

On first impression, this may seems like a lot of power we're wielding, but never fear: the `*` selector is very low specificity, meaning it only provides a `background-color` to elements for which one isn't already stated. In practice, `#fefefe` is just a fallback.

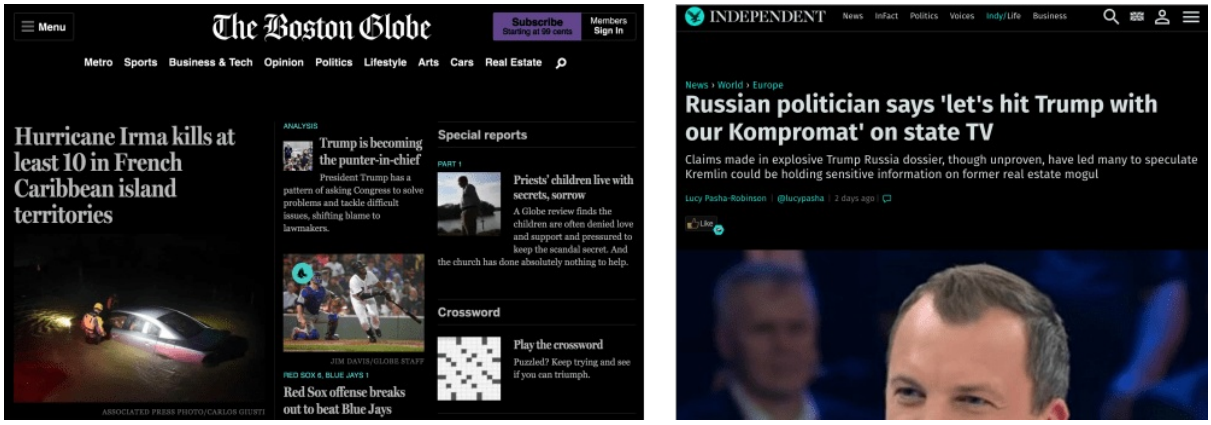
Preserving raster images

While we are intent on inverting the theme, we're probably not going

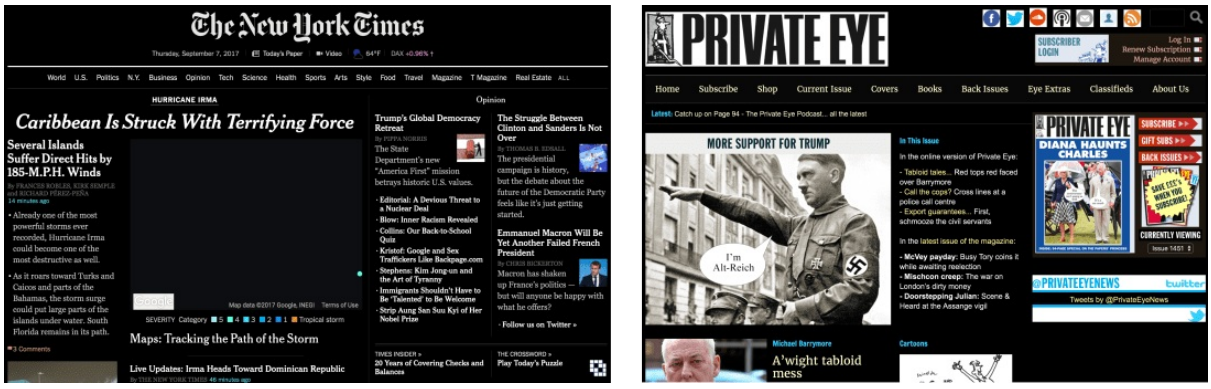
to want to invert raster images or videos, otherwise the design will become filled with spooky looking negatives. The trick here is to double-invert `<img/>` tags. The selector I'm using excludes SVG images, because — typically presented as flat color diagrams — they should invert successfully and pleasantly.

```
:root {
  background-color: #fefefe;
  filter: invert(100%);
}

* {
  background-color: inherit;
}

img:not([src*='.png']),
video {
  filter: invert(100%);
}
```

Clocking in at 153 bytes uncompressed, that's dark theme support pretty much taken care of. If you're not convinced, here's the CSS applied to some popular news sites:



The Boston Globe and The Independent



The New York Times and Private Eye

The theme switch component

Since the switch between light (default) and dark (inverted) themes is just an on/off, we can use something simple like the toggle buttons we explored in an earlier article. However, this time we'll implement the toggle button as part of a React component. There are a few reasons for this:

- Maximum reusability between the React-based projects many of you are used to working in.
- Ability to take advantage of React's `props` and `defaultProps`.

- Some people think frameworks like React and Angular preclude you from writing accessible HTML somehow, and that falsehood needs to die.

We're also going to incorporate some progressive enhancement, only showing the component if the browser supports `filter: invert(100%)`.

Setting up

If you don't already have a setup for React development, you can create one easily using `create-react-app`.

```
npm i -g create-react-app
create-react-app theme-switch
cd theme-switch
npm start
```

The boilerplate app will now be running at `localhost:3000`. In the new `theme-switch` project, our component will be called **ThemeSwitch** and will be included in `App.js`'s render function as `<ThemeSwitch/>`.

```
class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo"
alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code>
and save to reload.
        </p>
        <ThemeSwitch />
      </div>
    );
  }
}
```

(**Note:** I'm being lazy and leaving the boilerplate in. To really test the theme switcher, include it alongside a bunch of styled content pulled in from another project. You can include CSS in `App.css`.)

Don't forget to import the **ThemeSwitch** component at the top of this `App.js` file:

```
import ThemeSwitch from './components/ThemeSwitch';
```

The skeleton component file

As implied by the path in that last import line, we'll be working on a file called `ThemeSwitch.js`, placed in a new "components" folder, so

you'll need to create both the folder and the file. The skeleton for **ThemeSwitch** looks like this:

```
import React, { Component } from 'react';

class ThemeSwitch extends Component {
  render() {
    // The component's markup, in JSX
  }
}

export default ThemeSwitch;
```

The rendered markup for the switch, imagined in a default/inactive state, would look like this (notes to follow):

```
<div>
   <button aria-pressed="false">
      dark theme:
      <span aria-hidden="true">off</span>
   </button>
   <style media="none">
      html { filter: invert(100%); background: #fefefe
}
      * { background-color: inherit }
      img:not([src*=".png"]), video { filter:
invert(100%) }
   </style>
</div>
```

- Not all toggle buttons are created the same. In this case, we're using `aria-pressed` to toggle accessible state and an explicit

"on"/"off" for sighted users. So that the "on" or "off" part is not read out to contradict the state, it is suppressed from assistive technologies with `aria-hidden`. Screen reader users will hear "dark theme toggle button, not pressed" or "dark theme toggle button, pressed" or similar.
- The CSS is so terse, we're going to provide it as an embedded stylesheet. This is set to `media="none"` — or `media="screen"` when the dark theme is activated

This markup will get very messy shortly, as we convert it to JSX.

Switching state

Our component will be stateful, allowing the user to toggle the dark theme between inactive and active. First we initialize the state on the component's constructor:

```
constructor(props) {
   super(props);

   this.state = {
      active: 'false'
   };
}
```

To bring things to life, a `toggle()` function that actually toggles the state:

```
toggle() {
  this.setState({
    active: !this.state.active
  });
}
```

In the render function for the component, we can use `this.state.active` to determine the `aria-pressed` value, the button text, and the value of the stylesheet's `media` attribute:

```
return (
  <div>
    <button aria-pressed={this.state.active} onClick=
{this.toggle}>
      inverted theme:{' '}
      <span aria-hidden="true">{this.state.active ?
'on' : 'off'}</span>
    </button>
    <style media={this.state.active ? 'screen' :
'none'}>
      {this.state.active ? this.css.trim() : this.css}
    </style>
  </div>
);
```

`aria-pressed="false"` `aria-pressed="true"`

Of course, when the dark theme is on, the button itself is also inverted.

Note the `{this.css}` part. JSX doesn't support embedding CSS directly, so we have to save it to a variable and enter it here dynamically. In the constructor:

```
this.css = `
html { filter: invert(100%); background: #fefefe; }
* { background-color: inherit }
img:not([src*=".png"]), video { filter: invert(100%)
}`;
```

Overcoming browser issues

Unfortunately, just switching between `media="none"` and `media="screen"` does not apply the styles to the page in all browsers. To force a repaint, it turns out we have to rewrite the text content of the `<style>` tag. The easiest way I found of doing this was to incorporate the `trim()` method. Curiously, this only seemed to be needed in Chrome.

```
<style media={this.state.active ? 'screen' : 'none'}>
  {this.state.active ? this.css.trim() : this.css}
</style>
```

Persisting the theme preference

To persist the user's choice of theme, we can use `localStorage` and React lifecycle methods. First, we'll set a store key, falling back to 'ThemeSwitch':

```
this.storeKey = this.props.storeKey || 'ThemeSwitch';
```

Using the `componentDidMount` method, I can fetch and apply the saved setting after the component mounts to the page. The expression defaults the value to 'false' if the storage item is yet to be created.

```
componentDidMount() {
  this.setState({
    active: localStorage.getItem(this.storeKey) ||
false
  });
}
```

Because state is managed asynchronously in React, it's not reliable to simply save a changed state after it has been augmented. Instead, I need to use the `componentDidUpdate` method:

```
componentDidUpdate() {
  localStorage.setItem(this.storeKey,
this.state.active);
}
```

Progressive enhancement in React?

Some browsers are yet to support `filter: invert(100%)`. For those browsers, we will hide our theme switch altogether. It's better that it is not available than it is available and doesn't work. With a special `isDeclarationSupported` function, we can query support to set a

`supported` state.

If you've ever used Modernizr you might have used a similar CSS property/value test. However, we don't want to use Modernizr because we don't want our component to rely on any dependencies unless completely necessary.

```
isDeclarationSupported(property, value) {
  var prop = property + ':',
    el = document.createElement('test'),
    mStyle = el.style;
  el.style.cssText = prop + value;
  return mStyle[property];
}

componentDidMount() {
  if (this.store) {
    this.setState({
      supported: this.isDeclarationSupported('filter',
'invert(100%)'),
      active: this.store.getItem('ThemeSwitch') ||
false
    });
  }
}
```

The most efficient thing to do if the crticial feature here isn't supported is tonot render the component at all. React expects `null` as an early `return` value for this:

```
render() {
  if (!this.supported) {
    return null;
  }

  return (
    <div>
      <button aria-pressed={this.state.active} onClick=
{this.toggle}>
        inverted theme: <span aria-hidden="true">
{this.state.active ? 'on' : 'off'}</span>
      </button>
      <style media={this.state.active ? 'screen' :
'none'}>
        {this.state.active ? this.css.trim() :
this.css}
      </style>
    </div>
  );
}
```

Note

Windows High Contrast Mode

Windows users are offered a number of high contrast themes at the operating system level — some light-on-dark like our inverted theme. In addition to supplying our theme switcher feature, it's important to make sure WHCM is supported as well as possible. Here are some tips:

- Do not use background images as content. Not only will this invert the images in our inverted dark theme, but they'll be eliminated entirely in most Windows high contrast themes. Provide salient, non-decorative images in `<img/>` tags with descriptive `alt` text values
- For inline SVG icons, use the `currentColor` value for fill and stroke. This way, the icon color will change along with the surrounding text color when the high contrast theme is activated.
- Buttons may lose their `background-color` styles in high contrast mode, and end up appearing like plain text. To give them shape, one trick is to add a `transparent` border in the CSS. This will appear only when WHCM is switched on.
- If you need to detect WHCM to make special amendments, you can use the following media query:

```
@media (-ms-high-contrast: active) {
  /* WHCM-specific code here */
}
```

The `preserveRasters` prop

Props (component properties) are the standard way to make components configurable. A configurable component can be used in a greater variety of situations and projects and is therefore more

inclusive.

In our case, why don't we make it so that the implementor has a choice over whether raster images are indeed preserved, or if they're inverted with everything else. I'll create a Boolean `preserveRasters` prop:

```
<ThemeSwitch preserveRasters />
```

I can query this prop in the formulation of the CSS string, and only re-invert images if its value is "true":

```
this.css = `
  html { filter: invert(100%); background: #fefefe; }
  * { background-color: inherit }
`;

if (this.props.preserveRasters) {
  this.css +=
    'img:not([src*=".png"]), video, [style*="url("] {
filter: invert(100%) }';
}
```

Installing the component

A version of this component is available on NPM:

```
npm i --save react-theme-switch
```

Plain JavaScript version

Demo: A standalone and framework independent version of the theme switcher.

(Just for experimentation; `localStorage` persistence not implemented here.)

Placement

The only thing left to do is decide where you're going to put the component in the document. As a rule of thumb, utilities like theme options should be found in a landmark region — just not the `<main>` region, because the screen reader user expects this content to change between pages. The `<header>` (`role="banner"`) or `<footer>` (`role="contentinfo"`) are both acceptable.

The switch should appear in the same place on all pages so that, once the user has located it once, they can easily find it again. Take note of the Be consistent inclusive design principle, which applies here.

Checklist

- Only implement nice-to-have features if the performance hit is minimal and the resulting interface does not increase significantly in complexity
- Only provide interfaces for supported features. Use feature detection.
- Use semantic HTML in your React components — they'll still work!
- Use props to make your components more configurable and reusable

Tabbed Interfaces

When you think about it, most of your basic interactions are showing or hiding something *somehow*. I've already covered popup menu buttons and the simpler and less assuming tooltips and toggletips. You can add simple disclosure widgets, compound "accordions", and their sister component the **tabbed interface** to that list. It's also worth noting that routed single-page applications emulate the showing and hiding of entire web pages using JavaScript.

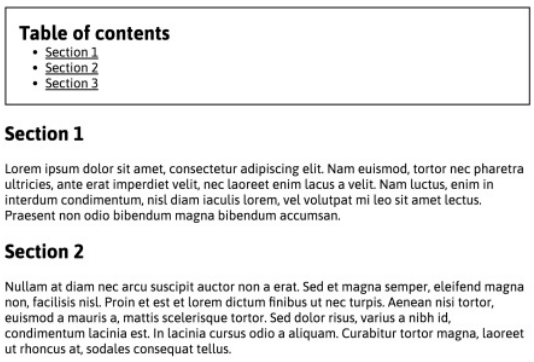
As we shall explore, the conceptual boundaries between tab panels, application views, and simple document fragments are not as clearly defined as we might like to pretend. Nonetheless, we need to assert with confidence precisely what kind of interface we are providing the user, otherwise they won't find the confidence to successfully operate it.

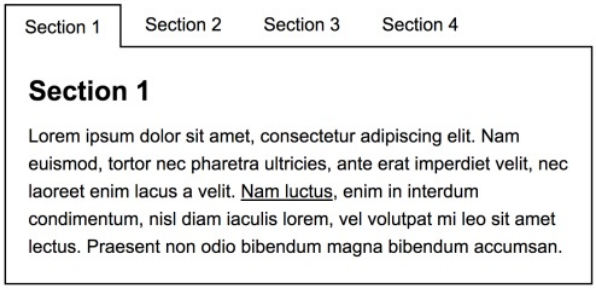
Proponents of progressive enhancement conceive interfaces in terms

of structured static content before contemplating how those interfaces may become enhanced with additional JavaScript-enabled interactivity. Even if you are happy to concede a JavaScript dependency early in an interface's design, it's beneficial to build a robust foundation using semantic HTML, capitalizing on standard browser behaviors. Sometimes you may even find JavaScript enhancement is a step you needn't take.

For my money, an embryonic tabbed interface is just a table of content with same-page links pointing at different sections of the page. Both a table of content and a list of tabs allow the user to choose between distinct sections of content.

- Table of contents → tab list
- Same-page links → tabs
- Sections → tab panels





Enhancement

What if I used some CSS to make just the chosen section from my table of contents visible? This is certainly possible using the `:target` pseudo-class.

```
section:not(:target) {
  display: none;
}
```

By placing the disclosure of content under the user's control, our CSS-enhanced TOC interface moves towards being a tabbed interface in one critical regard. Since `display: none` hides content from assistive technologies, this enhancement affects screen reader users as much as anyone else.

But this modestly enhanced version of the interface already has an issue. If you link to a subsection *inside* a section, the parent section isn't the target and won't become unhidden.

To fix this, we would need some JavaScript. In fact, it's quite involved. We need:

- A way to extract and interpret hash fragments
- A way to determine if a hash fragment either corresponds to a section or a subsection within a section
- To focus sections and subsections with JavaScript, because same-page linking becomes broken when there's hidden content
- `DOMContentLoaded` and `hashchange` events to read and act on URLs at appropriate intervals

Demo: Table of contents with hash tracking

This demo is provided for curiosity only. A table of contents does not

conventionally control the visibility of individual sections, and should be avoided. Only in a true tabbed interface is this kind of behavior expected. It's important that the behavior of your components is consistent with their appearance.

True tabbed interfaces

The advantage of using lists of same-page links and the standard browser behaviors they invoke is that they are simple and easy to understand — especially since the behavior of links is peculiar to the web.

Tabbed interfaces, on the other hand, are a paradigm imported from desktop applications. If they are understood by users in the context of web pages *at all* it is only through very careful and deliberate exposition of visual design and ARIA semantics.

What makes a tabbed interface a tabbed interface is in the ergonomics of its keyboard behavior. Really the only reason the ARIA semantics need be present is to alert screen reader users to the keyboard behaviors they should expect. Here is the basic semantic structure with notes to follow:

```
<ul role="tablist">
  <li role="presentation">
    <a role="tab" href="#section1" id="tab1" aria-
selected="true">Section 1</a>
  </li>
  <li role="presentation">
    <a role="tab" href="#section2" id="tab2">Section
2</a>
  </li>
  <li role="presentation">
     <a role="tab" href="#section3" id="tab3">Section
3</a>
  </li>
</ul>
<section role="tabpanel" id="section1" aria-
labelledby="tab1">
  ...
</section>
<section role="tabpanel" id="section2" aria-
labelledby="tab2" hidden>
  ...
</section>
<section role="tabpanel" id="section3" aria-
labelledby="tab3" hidden>
  ...
</section>
```

- This tabbed interface is progressively enhanced from a table of content and corresponding document sections. In some cases this means adding (`aria-selected`) or overriding (`role="tab"`) semantics. In others (`role="presentation"`) it means removing semantics that are no longer applicable or helpful. You don't want a set of tabs to also be announced as a plain list.
- `role="tablist"` does not delegate semantics to its children automatically. It must be used in conjunction with individual `tab`

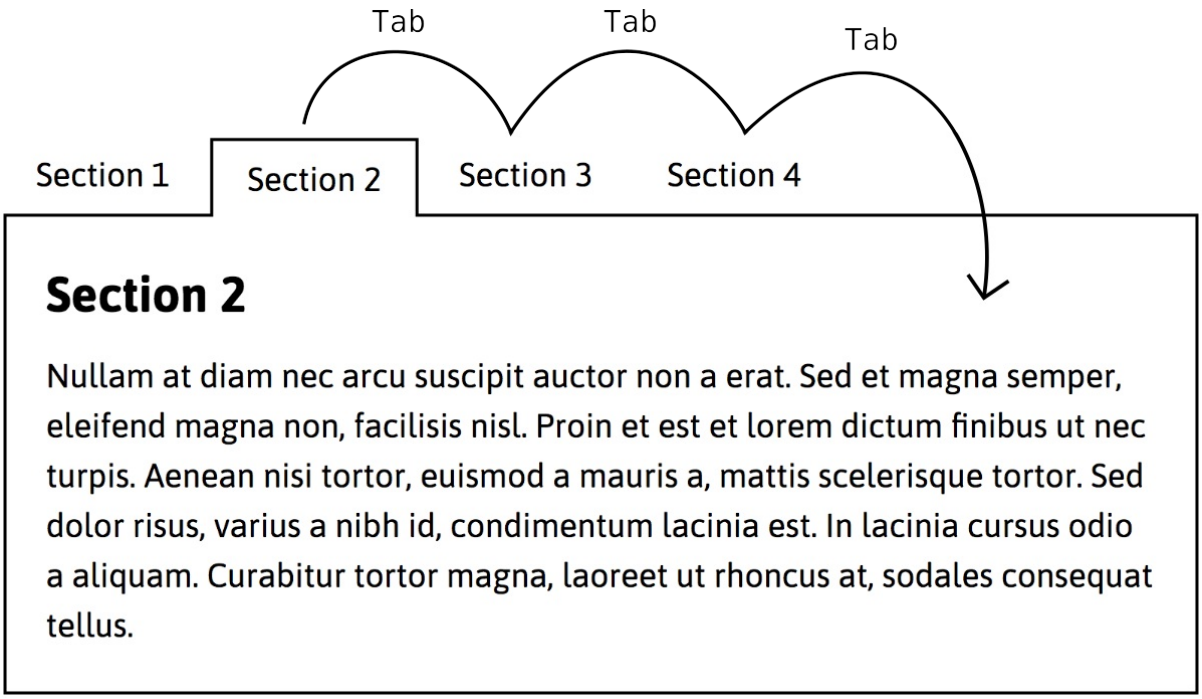
roles in order for tabs to be identified and enumerated in assistive technologies.

- The `tabpanel` elements which do not correspond to the selected tab are hidden using the `hidden` attribute/property.
- Users who enter a tabpanel should be assured of its identity. Hence, `aria-labelledby` is used to label the panel via the tab name. In practice, this means a screen reader user entering a panel and focusing a link will hear something like *"Section 1 tab panel, [link label] link"*.

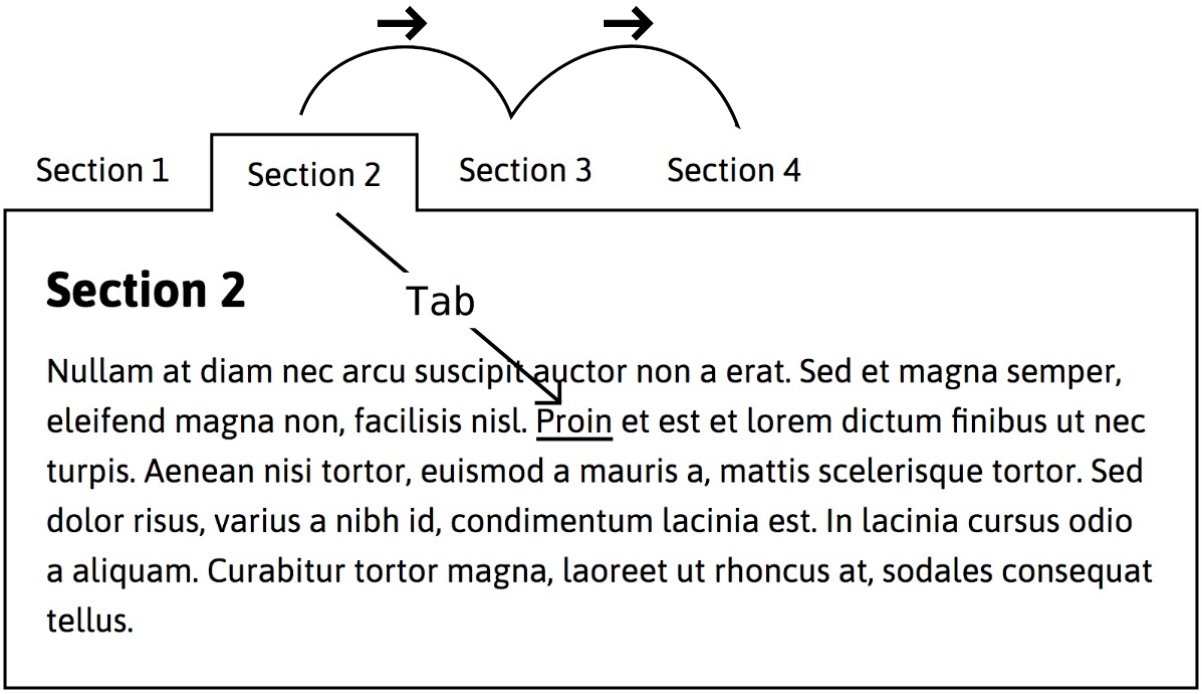
Keyboard behavior

Unlike a same-page link, a tab does not move the user to the associated section/panel of content. It just reveals the content visually. This is advantageous to sighted users (including sighted screen reader users) who wish to flit between different sections without having to wade back up the page each time they want to choose a new one.

This comes with an unfortunate side effect: If the user wishes to move *to* a section by keyboard and interact with its internal content, they have to step through any tabs to the right of the current tab, which are in focus order.



This problem is solved by delegating tab selection to arrow keys. The user is able to select and activate tabs using the arrow keys, while the `Tab` key is preserved for focusing contents within and below the active tab panel. To put it another way: `Tab` is not for tabs, which I concede is a bit confusing. I wish the key and the control had different names, but alas.



It's equally important that pressing **Shift** + **Tab** returns the user to the selected tab. This is all possible by giving each tab but the selected tab `tabindex="-1"`, which removes the inactive tabs from focus order but allows focus via a script. In the following example, the second tab is the selected tab, as denoted by the `aria-selected` state being set to true.

```
<ul role="tablist">
  <li role="presentation">
    <a role="tab" tabindex="-1"
href="#section1">Section 1</a>
  </li>
  <li role="presentation">
    <a role="tab" href="#section2" aria-
selected="true">Section 2</a>
  </li>
  <li role="presentation">
     <a role="tab" tabindex="-1"
href="#section2">Section 3</a>
  </li>
</ul>
```

With `tabindex="-1"` in place for the remaining tabs, I can capture the `keydown` event for the left or right arrow keys to make the desired inactive tab active.

```
tab.addEventListener('keydown', e => {
  // Get the index of the current tab in the tabs node
list
  let index = Array.prototype.indexOf.call(tabs,
e.currentTarget);

  // Determine key pressed
  var dir = e.which === 37 ? index - 1 : e.which === 39
? index + 1 : null;

  // Switch to the new tab if it exists
  if (dir !== null) {
    e.preventDefault();

    // Find correct tab to focus
    let newIndex;
    if (tabs[dir]) {
      newIndex = dir;
    } else {
      // Loop around if adjacent tab doesn't exist
      newIndex = dir === index - 1 ? tabs.length - 1 :
0;
    }
    switchTab(e.currentTarget, tabs[newIndex]);
    tabs[newIndex].focus();
  }
});
```

Each time a user selects a new tab, the corresponding tab panel is revealed. When the second of four tabs is selected, any screen reader running will say something similar to *"[tab label], selected, tab, 2 of 4"*. Plentiful information.

Note the 'loop around' feature which is outlined in the WAI-ARIA authoring practices section on keyboard behavior:

Left arrow (← **)**	Moves focus to the previous tab. If focus is on the first tab, moves focus to the last tab.
Right arrow (→ **)**	Moves focus to the next tab. If focus is on the last tab, moves focus to the first tab.

Demo: A true tabbed interface

Problems

Note that much of the behavior described so far is only the *prescribed* behavior for tabbed interfaces. The interaction is only ergonomic if the user is already familiar with the interaction paradigm, and the shortcut keys.

Testing cited in Danger! ARIA tabs suggests that users are not necessarily aware of the arrow key navigation, and may expect to be able to focus tabs (like any pretty much any other interactive element but radio buttons!) with the Tab key. A testing session held in a workshop ran by Hidde de Vries suggests that even screen reader power users can be baffled by the pattern:

We tested the arrow pattern with a blind, tech savvy user at my accessible components workshop, and he had no clue how to navigate to the next tab, I've avoided the pattern since.

— *Twitter, 4 Nov 2018*

Another problem regards the ability of users to move between the tab and corresponding tab panel. Currently, you can only move *into* the tab panel by focus if it contains interactive content. For sighted keyboard users this might not be a problem, because they can see and read the content anyway. But screen reader users need to browse their way through each of the inactive tabs (using the down arrow key in Windows screen readers like JAWS and NVDA).

One solution is to add `tabindex="0"` to the tab panels. This way, users should be able to focus the tab panel itself, and move into the tab panel from this position.

Unfortunately, Voiceover on macOS has trouble with focusable elements *inside* other focusable elements. Instead of the next **Tab** stop being the first interactive element inside the panel, it will be the first one outside and *past* the panel. In addition, the general rule is not to allow the focus of non-interactive elements by the user because the expectation is that focusable elements will each actually *do* something. Accordingly, a focusable tab panel would technically fail WCAG's **2.4.3 Focus Order** success criterion. It offers an unusable element to be used.

The `aria-controls` attribute is the only other option I can think of. By pointing the tab to a corresponding panel `id` with `aria-controls`, one exposes a keyboard shortcut to 'skip' to the panel directly in JAWS. Big problem: `aria-controls` *only* works in JAWS.

I have wrestled with these problems for a while and have begun suggesting the following to clients:

1. Do not use a tabbed interface at all, if there are more than four tabs. Use something else like an accordion.
2. Let the tabs receive focus naturally, by **Tab** key

3. Make the tabs behave like the same-page links they are enhanced from: focus the corresponding tab panel when they are clicked

It's perfectly fine to go against standardized authoring advice if research provides compelling evidence that the patterns they define fail users. Rules beget conventions, which is good for keeping the web consistent. But some conventions are stronger than others. Tabbing between controls is something most every keyboard and assistive technology user understands.

Responsive design

Responsive design is inclusive design. Not only is a responsive design compatible with a maximal number of devices, but it's also sensitive to a user's zoom settings. Full-page zoom triggers `@media` breakpoints just as narrowing the viewport does.

A tabbed interface needs a breakpoint where there is insufficient room to lay out all the tabs horizontally. The quickest way to deal with this is to reconfigure the content into a single column.

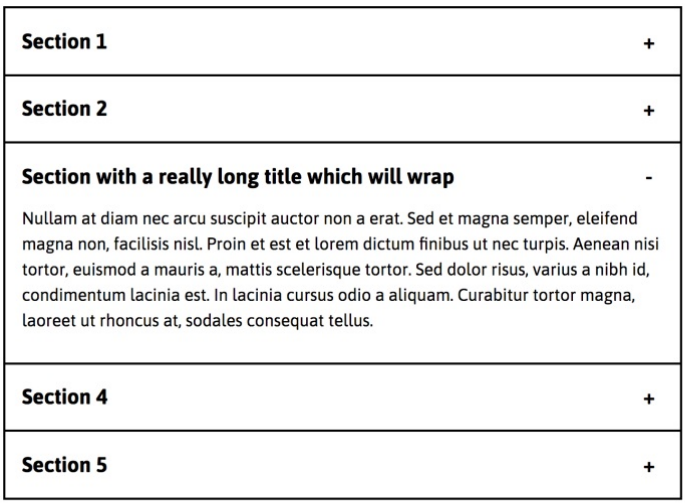
This can no longer be considered a "tabbed interface" visually because the tabs no longer look like *tabs*. This is not necessarily a problem so long as the selected tab (well, "option") is clearly marked. Non-visually, via screen readers, it presents and behaves the same.

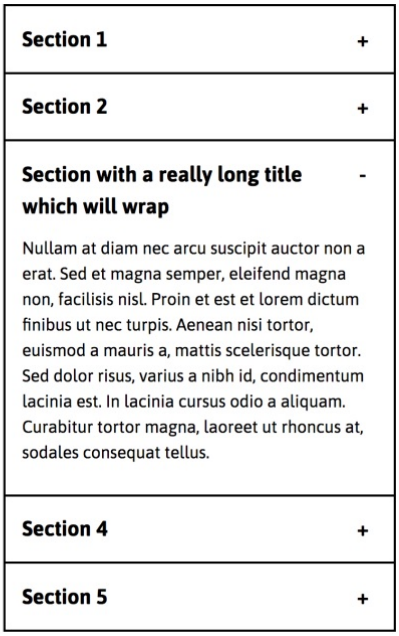
Accordions for small viewports?

Some have made noble attempts to reconfigure tabbed interfaces into accordion interfaces for small viewports. Given that accordions are structured, attributed, and operated completely differently to tabs, I would recommend against this.

Accordions do have the advantage of pairing each heading/button with its content in the source, which is arguably a better browsing experience in a one-column configuration. But the sheer complexity of a responsive tabbed interface/accordion hybrid is just not worth it in performance terms.

Where there are very many tabs or the number of tabs are an unknown quantity, an accordion at *all* screen widths is a safe bet. Single-column layouts are responsive regardless of content quantity.





My preference for narrower viewports is not to enhance into an accordion *or* a tabbed interface. Since we are using progressive enhancement anyway, we can simply refuse to run the script and leave the interface as a table of contents linking to sections. Still a perfectly servicable may to consume the content, and saves on CPU.

Using `matchMedia`, fork the logic. But we first need to detect if `matchMedia` is supported. In the following version, we only run the enhancement script above `400px`.

```
if (typeof window.matchMedia !== 'undefined') {
  if (window.matchMedia('(min-width: 400px)').matches)
{
    // Enhance into tabbed interface
  }
}
```

You have to think very carefully about whether the enhancement is really an enhancement. Do your users' really need a tabbed interface at any viewport width?

When panels are views

You'll recall my note from earlier that making the set of links in site navigation appear like a set of tabs is deceptive: A user should expect the keyboard behaviors of a tabbed interface, as well as focus remaining on a tab in the current page. A link pointing to a different page will load that page and move focus to its document (`body`) element.

What about the "views" in single-page applications: the different screens found at different routes? Technically, they are closer to the panels of our tabbed interface than whole web pages. But that's not to say they should be *communicated* as tab panels, because that's not what a user is likely to expect.

Single-page application views are typically intended to seem like distinct web pages or regions in a web page, so that is the story that should be told. Here are some provisions to make:

Use links!

Make sure the links that allow users to choose between views are indeed links — whether or not those links `return false` and use JavaScript to switch to the new view. Since these controls will navigate the user (by changing their focus location; see below) the link role is the most appropriate for the behavior. Link elements do not need the `link` ARIA role attribute; they are communicated as "link" by default.

In Xiao, a progressive enhancement-based router system for single-page applications, standard hash fragments are used to denote views. The links to these fragments will be communicated as *"same page links"* in most assistive software. By capitalizing on standard browser behavior, the user will be made aware they are being redirected to a new, distinct part of the page/application.

```
<a href="#some-route">Some route</a>
```

Manage focus

Just replacing some content on the page does not automatically move the user to that content or (in the case of blind assistive technology users) alert them to its existence. As covered under **The focus of non-interactive elements** above, you can focus the principle heading of the new route view, or the outer view element. If you are focusing the outer view element, it is recommended it is labeled either directly using `aria-label` or by the principle heading using `aria-labelledby`. The `aria-labelledby` method is preferred because it reduces redundancy — and the danger of things going out of sync — while also ensuring the label, as a text node, is translatable.

```
<div aria-labelledby="heading" role="region"
tabindex="-1">
  <h1 id="heading">Home</h1>
  ...
</div>
```

When used in conjunction with the `region` role (as in the above code snippet), when the element is focused the contextual information *"Home, region"* will be announced in screen readers.

Using Xiao, no region is focused on initial page load. This means focus defaults to the body/document element and the `<title>` is announced in screen readers (see below).

Update the `<title>`

The application name should be appended by the label for the specific view. This conforms to the recommended pattern for static sites where the site name is appended by the page name.

```
<title>[Application name]: [View name]</title>
```

You can load a Xiao-routed application at any route by simply including the route's hash fragment in the URL. On the load event, the `<title>` takes that route's label and screen readers identify the application and the specific route within it.

In React

The can achieve the same ends in React on a per-component basis, by tapping into each route's component's `componentDidMount()` method. The best way is probably to create a `ref` for the target element.

```
class Home extends React.Component {
  constructor(props) {
    super(props);
    this.focusTarget = React.createRef();
  }

  componentDidMount() {
    // Change the <title>
    document.title = 'My App: Home';
    // Focus the view
    this.focusTarget.focus();
  }

  render() {
    return (
      <div aria-labelledby="heading" role="region"
tabindex="-1" ref={this.focusTarget}>
        <h1 id="heading">Home</h1>
        // Content here
      </div>
    )
  }
}
```

Having to include this all for each route component is a pain, so looking to manage focus centrally, from the router itself, is a better solution. It's possible to achieve this in React Router 4 by wrapping the app in the `withRouter` HOC (Higher Order Component) and listening to history changes.

```
class App extends Component {

  componentWillMount() {
    this.unlisten =
this.props.history.listen((location, action) => {
      // Focus and <title> change here
    });
  }
  componentWillUnmount() {
      this.unlisten();
  }
  render() {
     return (
         <div>{this.props.children}</div>
      );
  }
}
export default withRouter(App);
```

Alternatively, Reach Router uses some heuristics to manage focus for you automatically, and comes highly recommended.

Conclusion

JavaScript can show and hide or create and destroy content with ease, but these DOM events can have different purposes and meanings depending on the context. In this article, we facilitated the basic show/hide ability of JavaScript to create two quite different interfaces: a tabbed interface and single-page application navigation.

There's really no right or wrong in inclusive design. It's just about trying your hardest to provide a valuable experience to as many people as

you can. A large part of this is pairing your presentation and behavior in ways that users — no matter how they are operating or reading your interface — would expect for the task in hand.

Checklist

- Don't provide tabbed interfaces unless they are suited to the use case and are likely to be understood and appreciated by the user. Just because you can doesn't mean you should.
- Tables of content and same-page links are a simpler and more robust approach to many of the ostensible use cases for tabbed interfaces. If you want to show/hide content, accordions are more responsive and simpler to implement. The next chapter will look at the collapsible sections that make up accordions.
- Make sure interfaces that appear as tabbed interfaces have the semantics and behaviors expected of them.
- Single-page applications should not present or behave as tabbed interfaces, despite their shared use of JavaScript to switch between and/or populate content panes.

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 — despite movement to standardize it. It is therefore a great "hello world" entry point into accessible interaction design using JavaScript and WAI-ARIA.

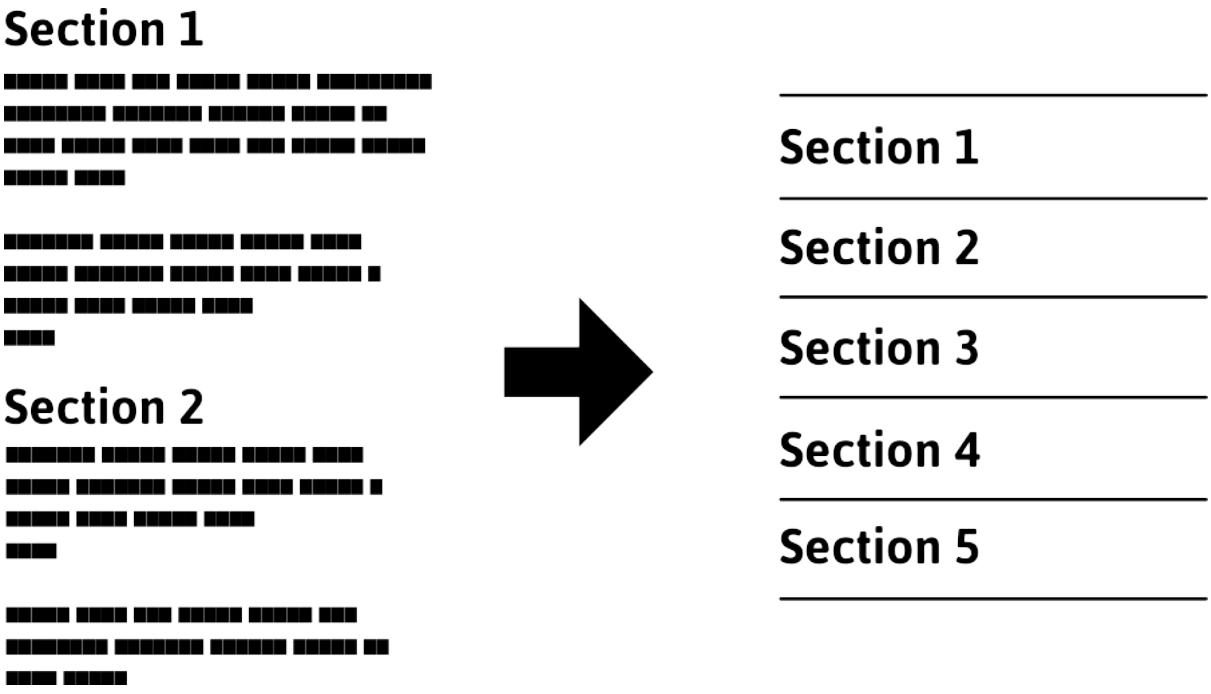
So why am I talking about it now, after covering more complex components? Because this article will focus on developer and author experience: we're going to make our collapsible regions web components, so they are easy to include as part of larger patterns and in content files.

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.

```
<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.

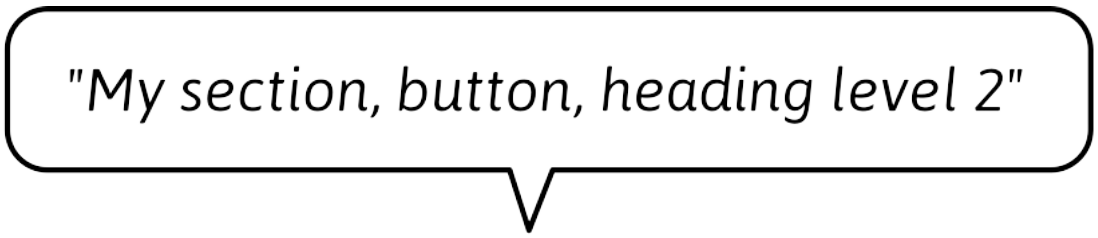
The adapted markup

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).



My section

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.

Note

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 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.

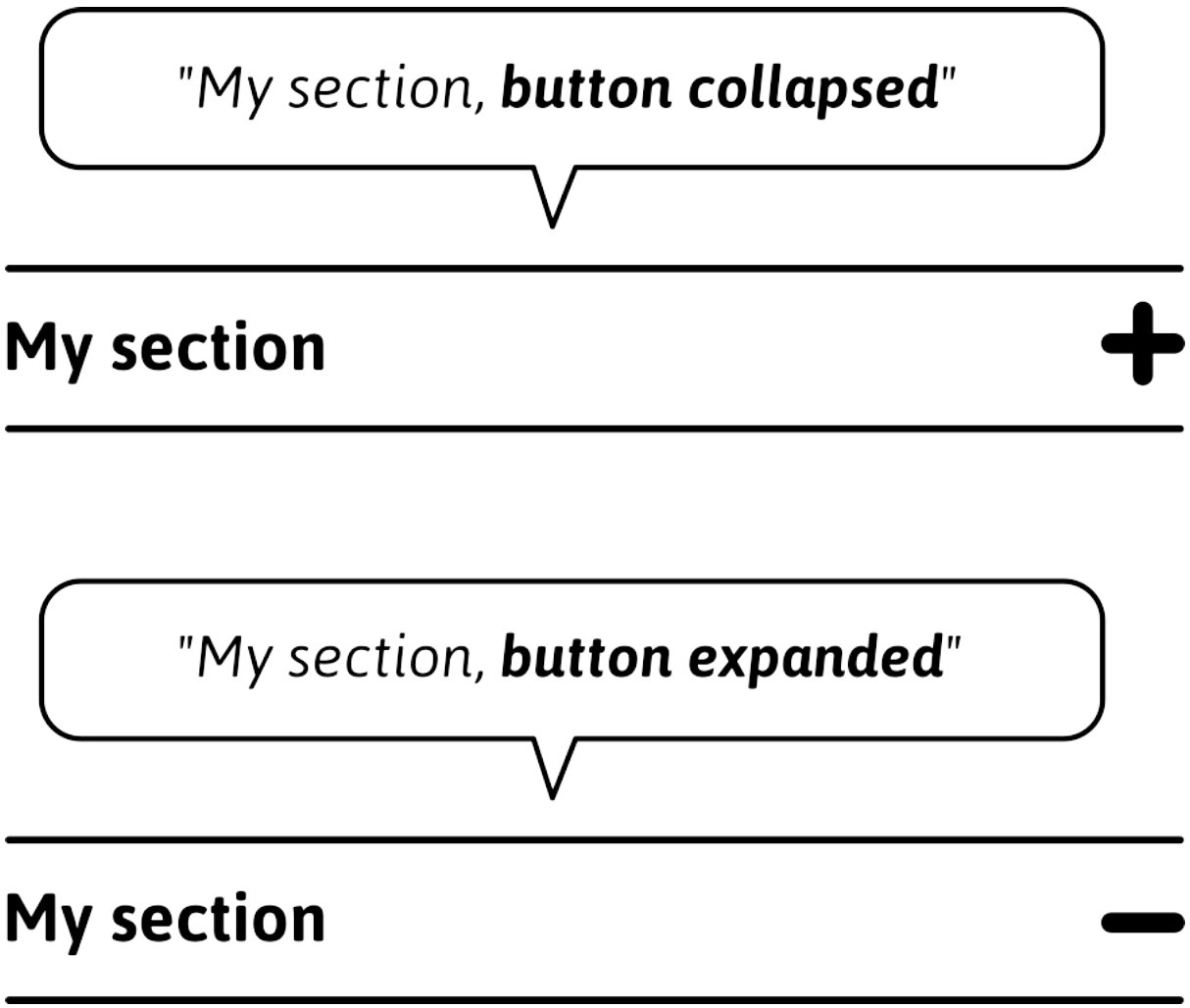
Styling the button

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;
}
```

Great, but now the button has no affordance. 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.



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.

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.

Note

Why not use `<use>`?

If we had many collapsible regions on the page, reusing the same SVG `<pattern>` definition via `<use>` elements 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, this is not a big problem in our case.

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');
    let target = h.nextElementSibling;

    btn.onclick = () => {
      let expanded = btn.getAttribute('aria-expanded')
=== 'true';

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

Demo: Basic collapsible sections

Progressive enhancement

The trouble with the above script 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 / 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

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 JavaScript 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
  }
}
```

Note

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 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 later.

```
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>
```

```
    <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

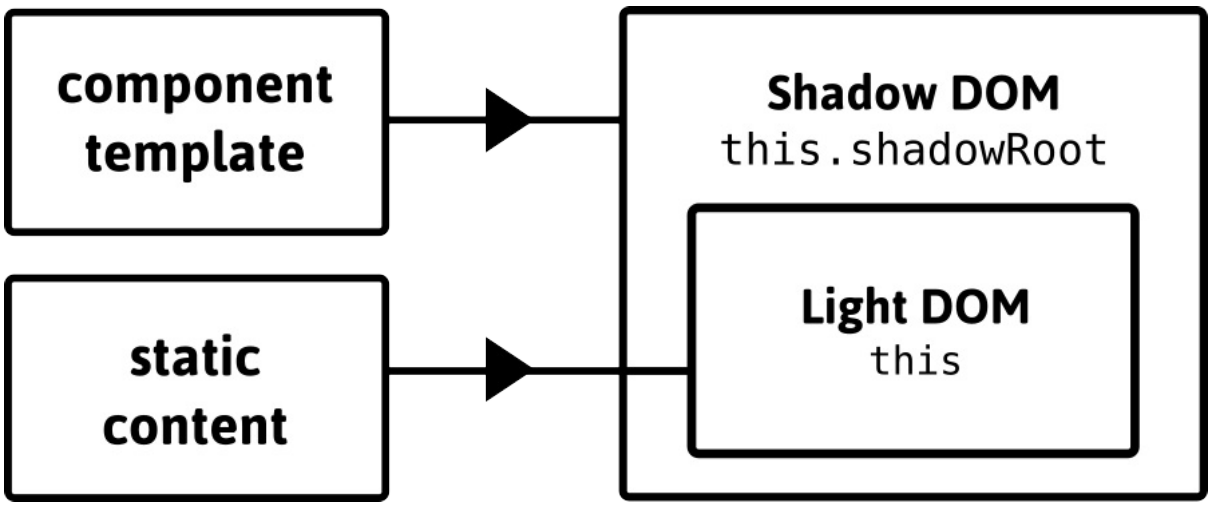
`<template>` and custom elements.

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.

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))
  }
}
```



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.

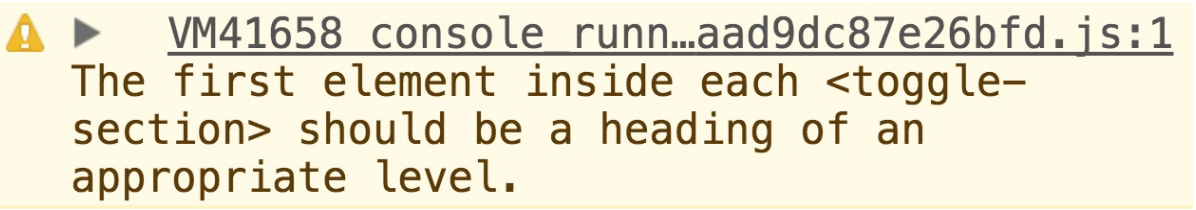
```
var oldHeading = this.querySelector(':first-child')
```

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.');
}
```



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 **A Todo 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 but many screen readers *do* provide region shortcuts. Adding `role="region"` gives us quite a bit:

- 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` value changes between `true` and `false` 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:

```
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 changes the `open` value, from outside the component. For *users* to change the state, we can just flip `open` inside a click function:

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

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.getAttribute('open') === 'true';
  this.btn.setAttribute('aria-expanded', expanded);
  this.shadowRoot.querySelector('.content').hidden =
!expanded;
}
```

Demo: Web component with additional expand/collapse all functionalitys

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.

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: the lists of navigation links discussed in **Menus & Menu Buttons**. 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()` lifecycle 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)

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

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.
- 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!

A Content Slider

Carousels (or ‘content sliders’) are like men. They are not *literally* all bad — some are even helpful and considerate. But I don’t trust anyone unwilling to acknowledge a glaring pattern of awfulness. Also like men, I appreciate that many of you would rather just avoid dealing with carousels, but often don’t have the choice. Hence this article.

Carousels don’t have to be bad, but we have a culture of making them bad. It is usually the features of carousels, rather than the underlying concept that is at fault. As with many things *inclusive*, the right solution is often not what you do but what you don’t do in the composition of the component.

Here, we shall be creating something that fulfills the basic *purpose* of a carousel — to allow the traversal of content along a horizontal axis — without being too reverential about the characteristics of past implementations.

Control

In the broadest terms, any inclusive component should be:

- Clear and easy to use
- Interoperable with different inputs and outputs
- Responsive and device agnostic
- Performant
- **Under the user's control**

That last point is one I have been considering a lot lately, and it's why I added *"Do not include third parties that compromise user privacy"* to the inclusive web design checklist. As well as nefarious activities, users should also be protected from unexpected or unsolicited ones. This is why WCAG prescribes the **2.2.2 Pause, Stop, Hide** criterion, mandating the ability to cease unwanted animations. In carousel terms, we're talking about the ability to cease the automatic cycling of content 'slides' by providing a pause or stop button.

It's something, but I don't think it's good enough. You're not truly giving control, you're taking it away then handing it back later. For people with vestibular disorders for whom animations can cause nausea, by the time the pause button is located, the damage will have been done.

For this reason, I believe a truly inclusive carousel is one that never moves without the user's say-so. This is why I prefer the term 'content slider' — accepting that the operative slider is the user, not a script. Content sliders start and stop moving as the user sees fit.

Our slider will not slide except when *slid*. But how is sliding instigated?

Multimodal interaction

'Multimodal' means *"can be operated in different ways"*. Supporting different modes of operation may sound like a lot of work, but browsers are multimodal by default. Unless you screw up, all interactive content can be operated by mouse, keyboard, and (where the device supports it) touch.

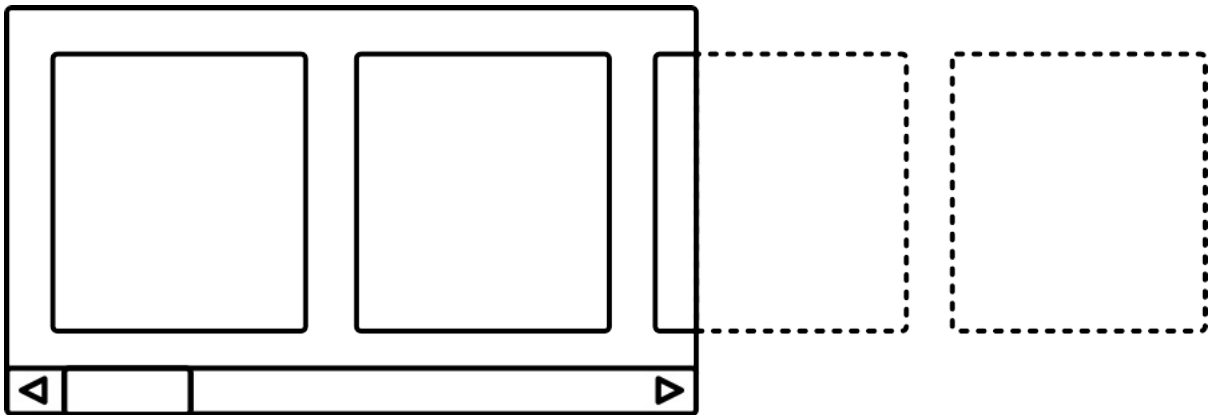
By deferring to standard browser behavior, we can support multimodality in our content slider with very little effort.

Horizontal scrolling

The simplest conceivable content slider is a region containing unwrapped content laid out on a horizontal axis, traversable by scrolling the region horizontally. The declaration `overflow-x: scroll` does the heavy lifting.

```
.slider {
  overflow-x: scroll;
}

.slider li {
  display: inline-block;
  white-space: nowrap;
}
```

Save for some margins and borders to improve the appearance of the slider, this is a serviceable MVP (Minimum Viable Product) for mouse users. They can scroll by pulling at a visible scrollbar, or by hovering over the slider and using trackpad gestures. And that animation is *smooth* too, because it's the browser doing it, not a JavaScript function fired every five milliseconds.



(Where no scrollbar is visible, affordance is not so obvious. Don't worry, I'll deal with that shortly.)

Keyboard support

For mouse users on most platforms, hovering their cursor over the

slider is enough to enable scrolling of the hovered element. For touch users, simply swiping left and right does the trick. This is the kind of effortless multimodality that makes the web great.

For those using the keyboard, only when the slider is focused can it be interacted with.

Under normal circumstances, most elements do not receive focus my default — only designated interactive elements such as links and `<button>`s. Elements that are *not* interactive should not be directly focusable by the user and, if they are, it is a violation of **WCAG 2.4.3 Focus Order**. The reason being that focus should precede activation, and if activation is not possible then why put the element in the user's hand?

To make our slider element focusable by the user, we need to add `tabindex="0"`. Since the (focused) element will now be announced in screen readers, we ought to give it a role and label, identifying it. In the demos to follow, we'll be using the slider to show artworks, so "gallery" seems apt.

```
<div role="region" aria-label="gallery" tabindex="0">
   <!-- list of gallery pictures -->
</div>
```

The `region` role is fairly generic, but is suitable for sizable areas of content and its presence ensures that `aria-label` is supported correctly and announced. You can't just go putting `aria-label` on any inert `<div>` or `<span>`. Note that there are translation issues with `aria-label`, so a visually hidden span is preferable in many cases. However, since this is effectively a *group label* we can't just secrete a `<span>`. We would have to connect a `<span>` to the element with `aria-labelledby`

like so:

```
<div role="region" aria-labelledby="gallery-label"
tabindex="0">
   <span id="gallery-label" aria-hidden="true"
class="visually-hidden">Gallery</span>
   <!-- list of gallery pictures -->
</div>
```

For brevity the examples to follow use `aria-label`, but always use text nodes where you can.

Now that focus is attainable, the standard behavior of being able to scroll the element using the left and right arrow keys is possible. We just need a focus style to show sighted users that the slider is actionable:

```
[aria-label="gallery"]:focus {
  outline: 4px solid skyBlue;
}
```

Demo: A basic content slider (no JavaScript)

Affordance

There are already a couple of things that tell the user this is a slidable region: the focus style, and the fact that the right-most image is usually

cut off, suggesting there is more to see.

Depending on how critical it is for users to see the hidden content, you may deem this enough — plus it keeps things terse code-wise.

The scrollbar

For elements that are scrollable, some operating systems and user agents provide a visible scrollbar. Others tidy the scroll bar away. Since a visible scroll bar gives added affordance, it would be better to show it wherever possible.

For webkit browsers, we can create a custom scrollbar and handle. This both reveals the scrollbar, and gives us an opportunity to style it as we wish.

```
[aria-label="gallery"]::-webkit-scrollbar {
  height: 0.75rem;
}

[aria-label="gallery"]::-webkit-scrollbar-track {
  background-color: #eee;
}

[aria-label="gallery"]::-webkit-scrollbar-thumb {
  background-color: #000;
}
```

It's possible to mimic a scrollbar using custom elements and JavaScript, but it's a notoriously hacky and heavyweight affair. By simply enhancing the standard browser scrollbar, the solution is more efficient and reliable. Non-webkit users will have a different experience, but not a broken one. That's the beauty of progressive

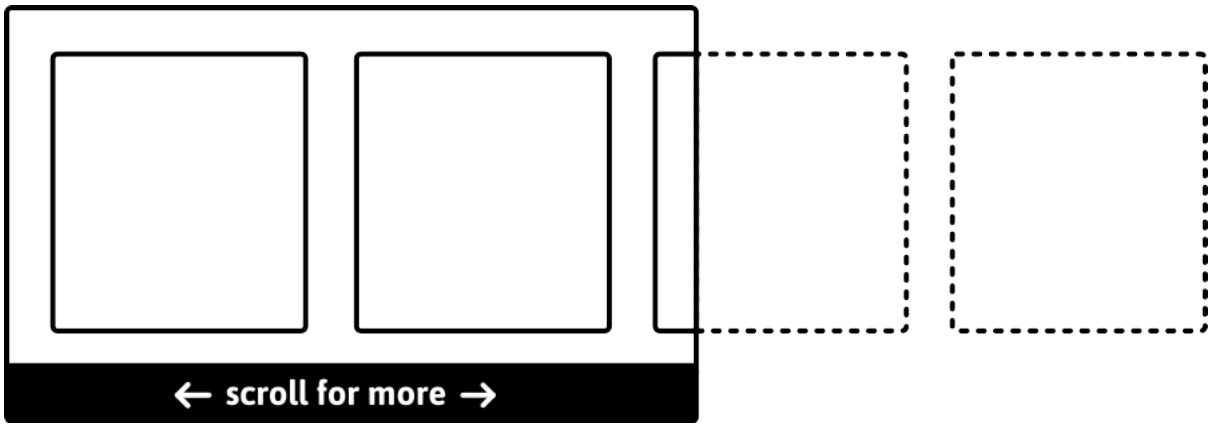
enhancement.

Instructions

We can take things one step further and literally spell out how the gallery content slider can be used. Inclusive design mantra: *If in doubt, spell it out*.

We can do this by creating an 'instructions' element after the slider to reveal messages depending on the state of the slider itself. For instance, we could reveal a `:hover` specific message of "*scroll for more*". The adjacent sibling combinator (+) transcribes the `:hover` style to the `.instructions` element.

```
#hover {
  display: none;
}

[aria-label="gallery"]:hover + .instructions #hover {
  display: block;
}
```



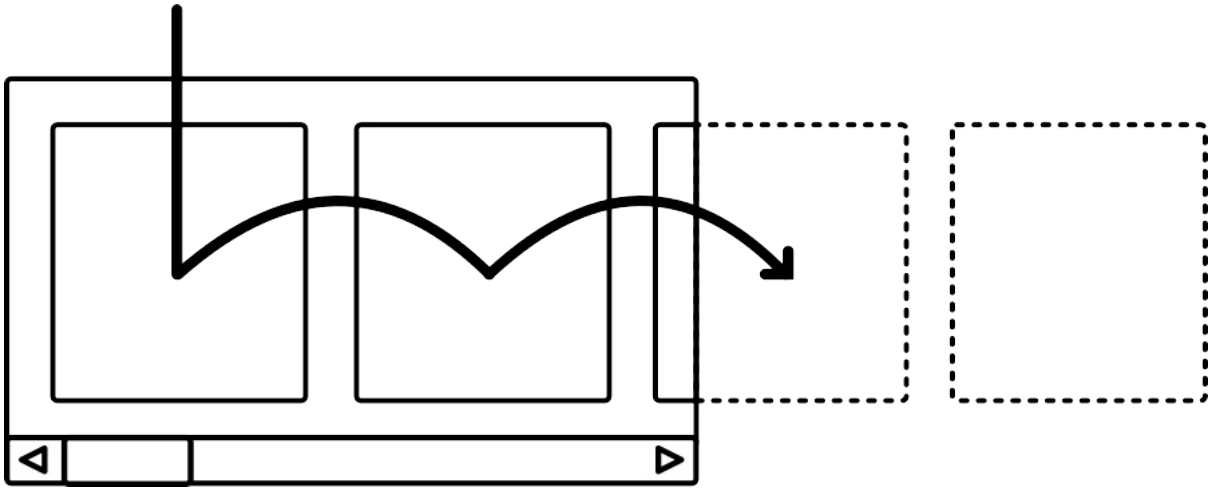
The `:focus` message can be done in much the same way. But we'll also

want to associate this message to the slider region for people running screen readers. Whether or not the region is of interest to any one screen reader user is irrelevant. It gives more context as to what it is for *should* it be appealing to them. And they know better what they're avoiding if not.

For this we can use our faithful `aria-describedby` property. We point it at the focus message element using its `id` as the value:

```
<div role="region" aria-label="gallery" tabindex="0"
aria-describedby="focus">
  <!-- list of gallery pictures -->
</div>
<div class="instructions">
  <p id="hover">scroll for more</p>
  <p id="focus">use your arrow keys for more</p>
</div>
```

Now, when focusing the gallery slider, screen readers will announce something similar to *"gallery, region, use your arrow keys for more."* As a further note on multimodality, be assured that screen reader users in "browse mode" (stepping through each element) will simply enter the region and traverse through each image in turn. In other words, the slider is multimodal even for screen reader users.



The path of a screen reader user in browse mode is much the same as a keyboard user's path given linked/interactive slides. In either case, the browser/reader will slide the container to bring the focused items into view.

Hover and focus?

It's interesting sometimes what you find in testing. In my case, I noticed that when I both hovered *and* focused the slider, both messages appeared. Of course.

As a refinement, I discovered I could concatenate the states (`:hover:focus`) and reveal a message that addresses both use cases at once.

```
[aria-label="gallery"]:hover:focus + .instructions
#hover-and-focus {
  display: block;
}
```

Using the general sibling combinator (~) I was able to make sure the other two messages were hidden (otherwise I'd see all three!):

```
[aria-label="gallery"]:hover:focus + .instructions
#hover-and-focus ~ * {
  display: none;
}
```

This is all very clever, but do we really need special messages for each interaction mode? After all, *scrolling* is the end, not the means, so "scroll for more" is probably adequate for both hover and focus. See it implemented in this demo:

Demo: A basic content slider with instructions

Handling the touch case

So far the touch experience is poor: No instructions are provided by default and there's no way to elicit the instructions without first operating the slider. Handling the touch interaction case means first detecting if the user is operating by touch.

Critically, we don't want to detect *touch support* at a device level, because so many devices support touch alongside other input methods. Instead, we just want to know if the *user* happens to be interacting by touch. This is possible by detecting a single `touchstart` event. Here's a tiny script (all the best scripts are!):

```
window.addEventListener('touchstart', function
touched() {
  document.body.classList.add('touch');
  window.removeEventListener('touchstart', touched,
false);
}, false)
```

All the script does is detect an initial `touchstart` event, use it to add a `class` to the `<body>` element, and remove the listener. With the `class` in place, we can make our “scroll for more” message a permanent fixture.

```
[aria-label="gallery"]:hover + #instructions,
[aria-label="gallery"]:focus + #instructions,
.touch #instructions {
  display: block;
}
```

Slides

Depending on your use case and content, you could just stop and call the slider good here, satisfied that we have something interoperable and multimodal that only uses about 100 bytes of JavaScript. That’s the advantage of choosing to make something simple, from scratch, rather than depending on a one-size-fits-all library.

But so far our slider doesn’t really do “slides”, which typically take up the full width of their container. If we handle this responsively, folks can admire each artwork in isolation, across different viewports. It

would also be nice to be able to add some captions, so we're going to use `<figure>` and `<figcaption>` from now on.

```
<li>
  <figure>
    <img src="[url]" alt="[description]">
    <figcaption>[Title of artwork]</figcaption>
  </figure>
</li>
```

Let's switch to Flexbox for layout.

```
[aria-label="gallery"] ul {
  display: flex;
}

[aria-label="gallery"] li {
  list-style: none;
  flex: 0 0 100%;
}
```

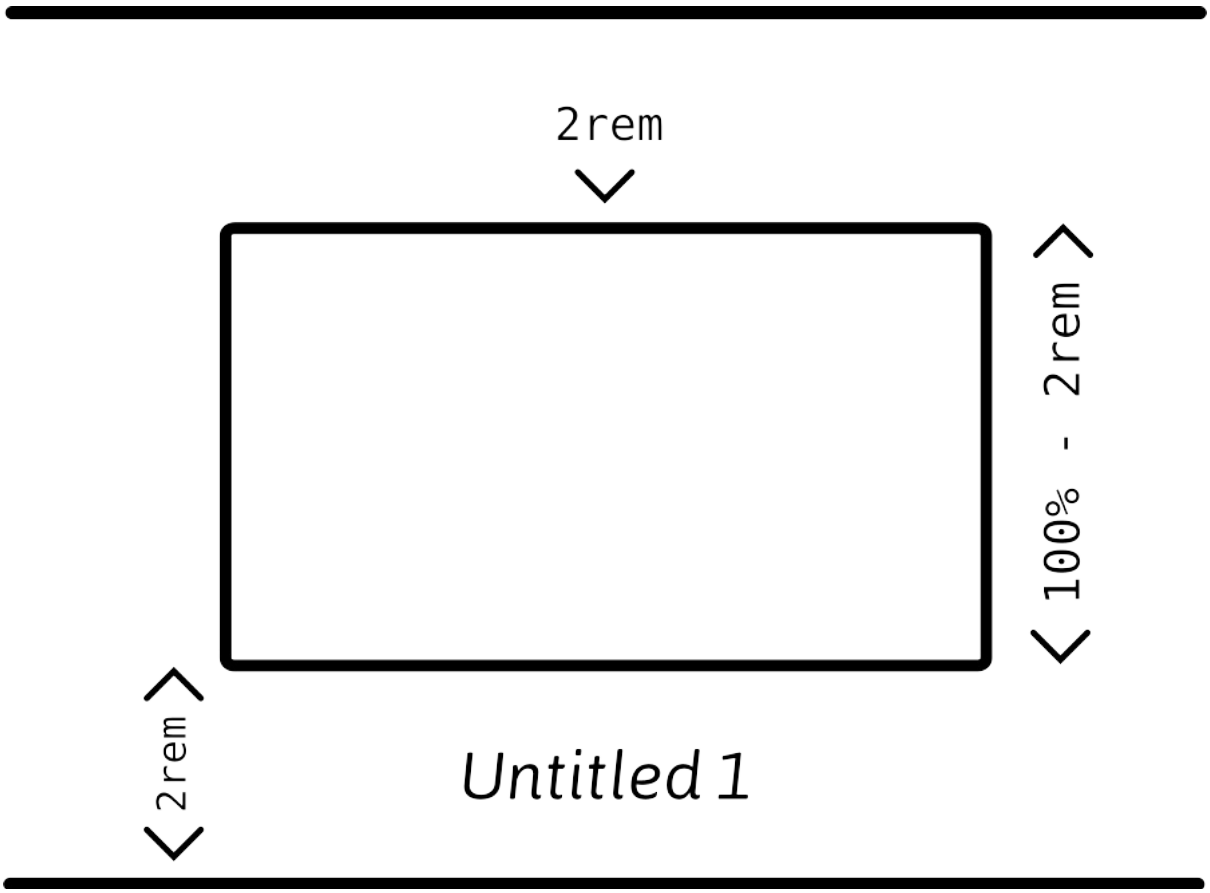
- Just `display: flex` is all we need on the container because `flex-wrap` defaults to `nowrap`
- The `100%` in the `flex` shorthand is the `flex-basis`, making each item take up 100% of the `<ul>` container.

I'm making the `<figure>` a flex context too, so that I can center each figure's contents along both the vertical and horizontal axes.

```
[aria-label="gallery"] figure {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  width: 100%;
  height: 50vh;
}
```

That `50vh` value is the only fixed(ish) dimension I am using. It's to make sure the slider has a reasonable height, but fits within the viewport. For the image and `<figcaption>` to always fit within the container, we make the image scale proportionately, but compensate for the predictable height of the `<figcaption>`. For this we can use `calc`:

```
[aria-label="gallery"] figcaption {
  height: 2rem;
  line-height: 2rem;
}

[aria-label="gallery"] img {
  display: block;
  margin: 2rem auto 0;
  max-width: 100%;
  max-height: calc(100% - 2rem);
}
```



The `<figcaption>` is set to a `2rem` height. This is removed from the flexible image's height using `calc`. A `margin-top` of `2rem` then re-centers the image.

Demo: Full width slides with captioned artworks

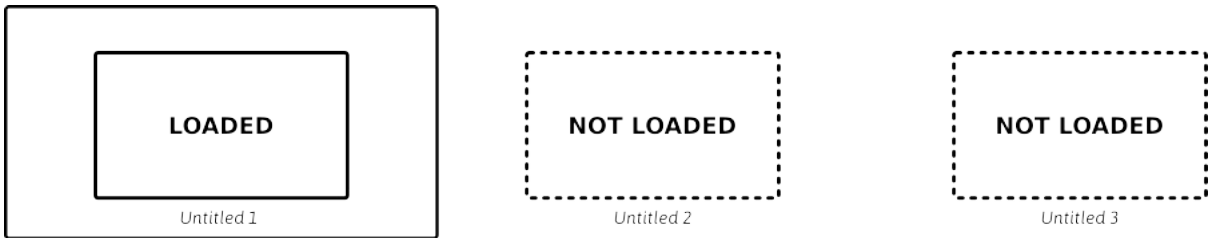
Performance and lazy loading

One of the most striking observations noted in the classic shouldiuseacarousel.com is that, of carousels that contain linked content, *"1% clicked a feature. Of those, 89% were the first position."*

Even for auto-rotating carousels, the research shows that the number of clicks on slides proceeding the initial slide drops off dramatically.

It is entirely likely that the first image in our content slider is the only one that most readers will ever see. In which case, we should treat it as the only image, and load subsequent images *if* the user chooses to view them.

We can use `IntersectionObserver`, where supported, to load each image as each slide begins to scroll into view.



Here's the script, with notes to follow:

```
const observerSettings = {
  root: document.querySelector('[aria-
label="gallery"]')
}

if ('IntersectionObserver' in window) {
  Array.prototype.forEach.call(slides, function (slide)
{
    let img = slide.querySelector('figure > img');
    img.classList.add('dots');
  });

  const callback = (slides, observer) => {
    Array.prototype.forEach.call(slides, function
(entry) {
      if (!entry.isIntersecting) {
        return;
      }
      let img = entry.target.querySelector('img');
      img.onload = () => img.classList.remove('dots');
      img.setAttribute('src', img.dataset.src);
      observer.unobserve(entry.target);
    })
  }

  const observer = new IntersectionObserver(callback,
observerSettings);
  Array.prototype.forEach.call(slides, t =>
observer.observe(t));
} else {
  Array.prototype.forEach.call(slides, function (s) {
    let img = s.querySelector('img');
    img.setAttribute('src', img.getAttribute('data-
src'));
  })
}
```

- In `observerSettings` we define the outer gallery element as the root. When `<li>` elements become visible within it, that's when we take action.
- We feature detect with `'IntersectionObserver' in window` and just load the images straight away if not. Sorry, old browser users, but that's the best we can offer here — at least you get the content.
- For each slide that intersects, we set its `src` from the dummy `data-src` attribute in typical lazy loading fashion.

In this implementation, the placeholder `src` is a loading indicator. I add a flashing animation style to this indicator via the `.dots` class and remove this class on the image's `onload` event. This is the only way to ensure the animation styling only affects the indicator and not the artwork that replaces it. In other words, it gets around the race condition that changing the class takes less time that loading the image.

```
img.onload = () => img.classList.remove('dots');
```

Note that, to begin with, the class is added dynamically — only where `IntersectionObserver` is supported. Otherwise, the images inside the `else` block will retain the class and the artworks themselves will flash perpetually. Not nice.

The indicator is provided as a data URL, meaning it is not itself a resource that needs to be waited upon. I use UTF encoding to describe an SVG:

```
src='data:image/svg+xml;utf8,<svg
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 6 2"
stroke="currentColor" stroke-dasharray="1,0.5"><path
d="M1,1 5,1" /></svg>'
```

The inline `stroke` styles are necessary because I can't target the SVG's DOM from the parent page.

Demo: Content slider with lazy loading images

(To see the effect, try throttling the network in Chrome's developer tools by setting to Mid-tier or Low-end mobile).

(**Note:** It's not necessary to know the gallery image's dimensions ahead of time to stop the page 'jumping' as they load in our case, because the gallery has a set height anyway.)

No JavaScript

Currently, users with no JavaScript running are bereft of images because the `data-src`/`src` switching cannot occur. The simplest solution seems to be to provide `<noscript>` tags containing the images with their true `src` values already in place.

```
<noscript>
  <img src="[url]" alt="[description]">
</noscript>
```

In addition, we need to hide the loading indicator image. I placed a `no-js` class on the gallery container and add the following CSS. There's some other wrangling for `<noscript>` styling which I'll leave for you to discover in the demo.

```
.no-js .dots {
  display: none;
}
```

Since our slider is operable without JavaScript, we're pretty good. However, this only handles the 'no JavaScript' case — which is rare — and not the 'broken/failed JavaScript' case which is distressingly common. Rik Schennink has solved this problem by placing a `mutationObserver` in the *head* of the document. A demo is available for this technique, which initially swaps `src` to `data-src` and, in testing, fairly reliably prevents the fetching of the image resources on first run.

Previous and next buttons

Typical sliders have buttons on either side of them for moving backwards or forwards through the slides. This is a convention that might be worth embracing for two reasons:

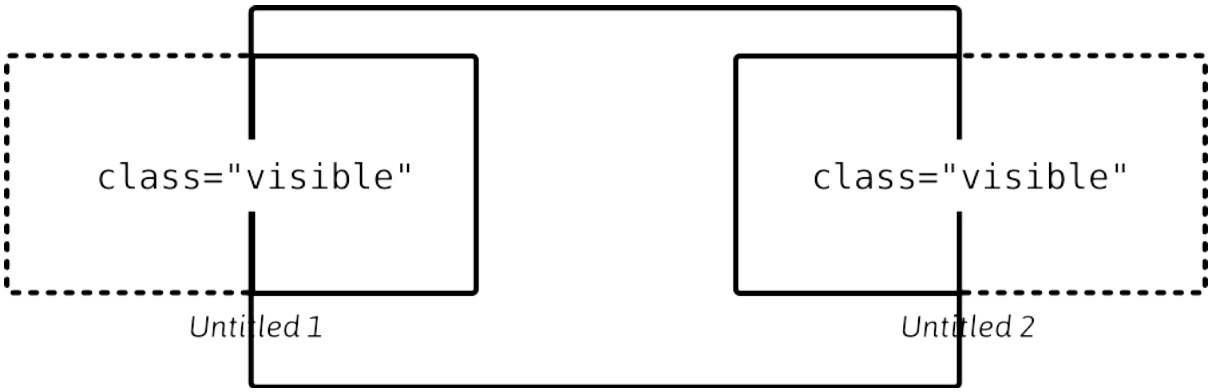
- The mere presence of the buttons makes the slider more slider-like, increasing its affordance.
- The buttons allow the user to 'snap' slides into place. No more scrolling back and forth to get the desired slide centered exactly.

The trick is in building upon the functionality we've already designed, rather than replacing it. Our buttons should be aware of and able to respond to scrolling and swiping actions that may already have taken place.

By adapting our `IntersectionObserver` script, we can add and remove a `.visible` class to our slides:

```
slides.forEach(entry => {
  entry.target.classList.remove('visible')
  if (!entry.isIntersecting) {
    return;
  }
  let img = entry.target.querySelector('img');
  if (img.dataset.src)  {
    img.setAttribute('src', img.dataset.src);
    img.removeAttribute('data-src');
  }
  entry.target.classList.add('visible');
})
```

Not only does this mean we'll find `class="visible"` on any slide that's 100% in view (such as the initial slide), but in the case that the user has scrolled to a position between two slides, they'll both carry that class.



To move the correct slide fully into view when the user presses one of the buttons, we need to know just three things:

1. How wide the container is
2. How many slides there are
3. Which direction the user wants to go

If two slides are partially visible and the user presses ‘next’, we identify the requested slide as the second of the `.visible` node list. We then change the container’s `scrollLeft` value based on the following formula:

requested slide index (container width / number of slides)

Note the size of the previous and next buttons in the following demo. Optimized for easy touch interaction without hindering the desktop experience.

The button group

By placing the two buttons in a list, they are treated as grouped items and enumerated. Since `<ul>` implicitly supports `aria-label` we can provide a helpful group label of “gallery controls” to further identify the purpose of the buttons.

```
<ul aria-label="gallery controls">
  <li>
    <button class="previous" aria-label="previous
artwork">
      <svg aria-hidden="true" focusable="false"><use
xlink:href="#arrow-left"></use></svg>
    </button>
  </li>
  <li>
    <button class="next" aria-label="next artwork">
      <svg aria-hidden="true" focusable="false"><use
xlink:href="#arrow-right"/></svg>
    </button>
  </li>
</ul>
```

Each button, of course, must have an independent label, administered with `aria-label` for brevity in this case, but be aware of the translation issues stated earlier in the chapter and throughout the book. When a screen reader user encounters the first button, they will hear something similar to *"previous button, list, gallery controls, two items"*.

We only provide the controls if the browser supports `IntesectionObserver`. For browsers that don't support it, the content slider still renders and is still mouse, keyboard, and touch accessible.

```
gallery.parentNode.insertBefore(controls, gallery);
```

Scrolling enhancements

A couple of final enhancements to improve the scrolling experience.

The first is to add `scroll-behavior: smooth` to the scrollable element. Although not supported everywhere, this is a highly efficient way to animate the button-activated scrolling. Without it, artworks seem to just appear and disappear, meaning users may not be aware there is a linear continuum.

The second is to support 'snap points'. Some browsers — Safari and Firefox included — support a simple CSS method of snapping slides into place as you scroll or use your arrow keys. Since Safari doesn't support `IntersectionObserver`, this is one way to improve the UX for users of that browser. The following mess of proprietary and standard properties is what worked in this case.

```
[aria-label="gallery"] {
  -webkit-overflow-scrolling: touch;
  -webkit-scroll-snap-type: mandatory;
  -ms-scroll-snap-type: mandatory;
  scroll-snap-type: mandatory;
  -webkit-scroll-snap-points-x: repeat(100%);
  -ms-scroll-snap-points-x: repeat(100%);
  scroll-snap-points-x: repeat(100%);
}
```

Tip: the `repeat(100%)` part refers to the 100% width of each slide.

Demo: Content slider with buttons, `scroll-behavior: smooth` and snap points

With the buttons now in place, it's tempting to remove the ability to

scroll the region directly. In fact, if your preferred method is using the buttons, then the `tabindex="0"` on the container could be considered an obstructuve extra tab stop. However, different people like to interact with things in different ways — hence erring on the side of multimodality.

Disabling buttons

If the scroll position of the gallery element is right at the start or all the way to the end, the previous or next button isn't going to do anything. You may want to consider disabling the redundant button under these circumstances. But there are a few things to consider:

- Disabled buttons (buttons with the `disabled` attribute/property) are not focusable. When a button the user is currently operating becomes disabled, things may therefore get confusing. The user may tab away, then tab back only to find the button isn't there anymore. Screen reader users can still reach it by moving their virtual cursor to the element, but they wouldn't know to do this in the context, since `Tab` worked perfectly well before.
- Disabled styles can be problematic. By default, disabled buttons are just 'greyed out' which doesn't necessarily say 'disabled' to some people — especially if they are color blind.
- To keep things consistent, we would need to disable buttons in response the clicking the buttons *and* scrolling the gallery directly. This means listening to the `scroll` event, which has an inherent performance impact.

The performance issue can be overcome using debouncing. Many authors reach for a library like Lodash for this kind of thing, but we can implement a simple debounce in a couple of lines.

```
var debounced;
gallery.addEventListener('scroll', function () {
  window.clearTimeout(debounced);
  debounced = setTimeout(disable, 200);
});
```

We simply assign the `debounced` variable, then use it to create and clear a `setTimeout`. The upshot is that the function only fires if the user's scrolling has been idle for over 200 milliseconds. Operation becomes much less janky.

The `disable` function just tests to see if the scroll position is at the start or end and sets disabled where applicable:

```
function disable() {
  prev.disabled = gallery.scrollLeft < 1;
  next.disabled = gallery.scrollLeft ===
list.scrollWidth - list.offsetWidth;
}
```

In terms of styling, we'd rather not rely on color. In this case, the clearest and simplest interpretation to my mind is the removal of the SVG icon. The 'button' then becomes an inert border, with little affordance.

```
.gallery-controls button[disabled] svg {
  display: none;
}
```

Previous button disabled

Next button disabled

Finally, we have to disable the previous button on page load:

```
prev.disabled = true;
```

Demo: Content slider with buttons that disable, via a debouncing function

Whether disabling the buttons explicitly like this is a good idea or not is difficult to know. On the one hand, having a button that doesn't do anything in focus order is redundant. On the other hand, bringing buttons in and out of focus order may be disorienting. The best way to know if it's a good idea is to test the component *in context* and with *real content*. There are any number of contextual factors regarding the user's and the application's overall state that may exaserpate issues on either side.

One thing to consider from a technical standpoint is that `tabindex="0"` will not reinstate the ability to focus buttons with the `disabled` property. Which is a shame, because being able to focus a disabled button (and hear "dimmed" or "disabled") may be instructive to the user. Instead, to support keyboard and screen reader users of all kinds,

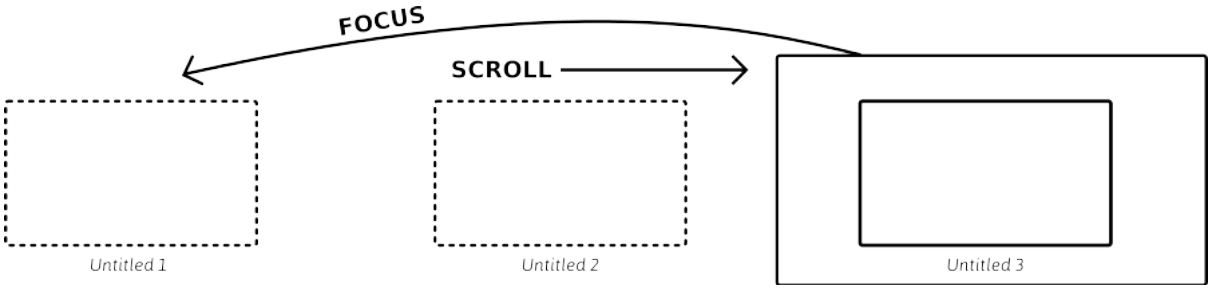
you would have to use a combination of `tabindex` and ARIA. In the following example, `tabindex="-1"` removes the button from focus order.

```
<!-- enabled -->
<button
  class="previous"
  aria-label="previous artwork">
</button>

<!-- disabled -->
<button
  class="previous"
  aria-label="previous artwork"
  aria-disabled="true"
  tabindex="-1">
</button>
```

Handling linked content

Focus order is currently very simple in our slider: The button controls receive focus first, followed by the scrollable region.

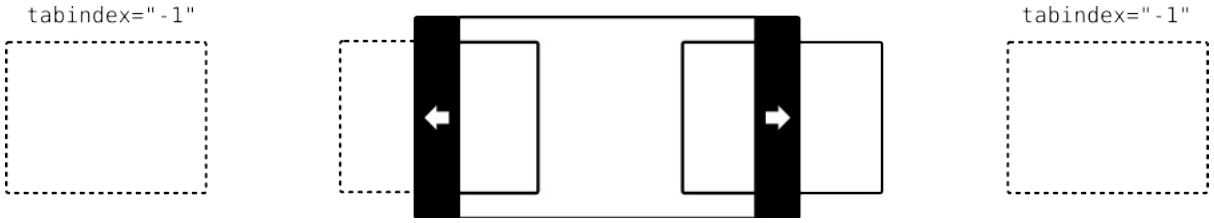
But what if the content of each slide were linked? After you focused the button controls, the first slide would take focus no matter whether it is currently visible or not. That is, if the user has scrolled the region to view the third item, they would expect that item to be the one that receives focus next. Instead, the first item takes focus and the slider is slung back to the start, bringing that first item into view.



This is no disaster. In fact, items receiving focus being automatically brought into view, without JavaScript, stands us in good stead. Invisible content should never become focusable.

But where `IntersectionObserver` is supported and our button controls have been rendered, having only the currently visible item(s) in the focus order makes for a good enhancement. We can amend our script so that links in items that are not intersecting take `tabindex="-1"`, making them unfocusable. See the lines commented (1) and (2) in the following.

```
Array.prototype.forEach.call(slides, entry => {
  entry.target.classList.remove('visible');
  let a = entry.target.querySelector('a');
  a.setAttribute('tabindex', '-1'); // (1)
  if (!entry.intersectionRatio > 0) {
    return;
  }
  let img = entry.target.querySelector('img');
  img.onload = () => img.classList.remove('dots');
  img.setAttribute('src', img.dataset.src);
  entry.target.classList.add('visible');
  a.removeAttribute('tabindex', '-1'); // (2)
})
```



Arguably, slides that are not visible should not be perceivable to screen readers either. This is eminently achievable by adding and removing the `aria-hidden` state. However, in the interest of multimodality, it's better to let screen reader users traverse the list directly using their 'virtual cursor'.

The virtual cursor lets screen reader users navigate around web pages without having to rely on interactive elements and focus. It lets them move from element to element. Most screen readers superimpose their own quasi-focus ring to show sighted operators where there virtual cursor is situated.

In the case of our conent slider, moving the virtual cursor to a currently invisible slide scrolls it into view and (therefore) removes `tabindex="-1"` from its link. Note that, with slide `<li>`s taking up 100% of the container width, applying `aria-hidden="true"` to each invisible one would mean screen reader users always encounter a list of just one (or two) list items. This would be confusing since numerous items would be revealed through interaction.

The complete script for this content slider is less than `2KB` minified. The first result when searching for 'carousel plugin' using Google search is 41.9KB minified and uses incorrect WAI-ARIA attribution, in some cases hiding focusable content from screen reader software using `aria-hidden`. Beware the fourth rule of ARIA use. In this final demo, a `flexbox` image scaling bug was suppressed by using `min-width: 1px` and `min-height: 1px` on the images.

Demo: Content slider with focus management for links

Safari does not support `IntersectionObserver` yet, but a polyfill is available for about 6KB gzipped. The slider works okay in Safari and other non-supporting browsers without it.

Conclusion

Inclusive design is not about giving everyone the same experience. It's about give as many people as possible a decent experience. Our slider isn't the fanciest implementation out there, but that's just as well: it's the content that should be wowing people, not the interface. I hope you enjoyed my generative artworks used in the demonstration. You can generate your own at mutable.gallery.

In my conference talk Writing Less Damned Code, I introduce the concept of *unprogressive non-enhancement* — the idea that the flow content from which we construct tab interfaces, carousels and similar, should often be left unreconstructed. No enhancement can be better than 'enhancement'. But, when used judiciously and with care, augmented presentations of content such as content sliders can be quite compelling ways of consuming information. There just better be a good, well-researched reason to take that leap.

Checklist

- Use list markup to group the slides together. Then screen reader users in 'browse' mode can use list navigation shortcuts to traverse them.

- Provide a reasonable experience in HTML with CSS, then feature detect when enhancing with JavaScript.
- Don’t preload content users are not likely to see. Defer until they perform an action to see it.
- Provide generous touch targets for touch users on mobile / small screens.
- If in doubt of a control’s (or widget’s) affordance, spell it out with instructions
- If you are a man and got past the first paragraph without being personally offended: Congratulations! You do not see men and women as competing teams.

Notifications

The key difference between a website and a web app is… highly contested. On the whole, I'd say the more links there are, the more site-like, and the more buttons, the more app-like. If it includes a page with a form, it's probably a kind of site. If it essentially *is* a form, you might call it an app. In any case, your web 'product' is really just interactive content, consumed and transmitted by an app we call a 'browser'.

One thing that certainly makes a web page *feel* more like a desktop app is statefulness. Web pages that undergo changes as you are operating them are something quite unlike web pages that just load and unload as you click hyperlinks.

Sometimes the user might instigate a change in state. Sometimes another user might affect the app remotely, in real time. Occasionally, the app might be subject to environmental and time-based events independent of user interaction. In each case, it's important users are kept abreast of changing state, which is a question of notifying them.

In this article, I'll be looking at notification components and how they

can increase confidence in the use of web applications, in an inclusive way.

Drawing attention

One of the biggest challenges in creating usable interfaces is knowing when to draw attention to something. Over-sharing may be considered a nuisance, but under-sharing might make the user feel they are missing critical information. This makes some hesitant, even where there is really nothing they "need to know" at the time.

Then there's the *how*. Broadly speaking, there are two kinds of messages which need two different approaches to be accessible:

1. Messages asking users to take action
2. *Just FYI* messages

Typically, a message asking a user to do something would form the content of a dialog window (or inline disclosure), and be accompanied by a choice of action buttons. Because the keyboard operator will need to access those buttons, focus must be moved into the dialog.

For the purpose of this article, what I mean by "notification" is a message that just lets you know what's going on. This may be so you can choose to take action later, or it may be to assure you of an event having taken place already.

In screen reader and keyboard accessibility terms, it's important that focus is *not* moved to such messages. If there is nothing to be done

with the tool, you don't put the tool in the person's hand. Despite this, moving focus has endured as a 'best practice'. Why? Because focusing an element has, traditionally, been the most reliable way to get that element and its contents announced in screen readers.

"Okay great. But where am I? What do I do now?"

Fortunately, we have live regions to help us break this habit.

Live regions 101

We've used live regions before on Inclusive Components, but I'm going to take the time to give you a broad overview here.

A live region is just a container element that sets a perimeter around 'live' content: content that will be announced — by screen reader software — without user interaction, under certain conditions. By default, a live region will announce anything that is added or changed inside it.

Somewhat perplexingly, there are two equivalent APIs for live regions: the `aria-live` attribute and live region ARIA roles. In most cases, you will want to use one of `role="status"` or `aria-live="polite"`. Using both simultaneously maximizes compatibility with different browser

and assistive technology pairings:

```
<div role="status" aria-live="polite">
</div>
```

Adding “Take a short break!” to this live region (as illustrated below) will trigger announcement immediately after the text node is inserted. It doesn’t have to be a text node; it can be any markup.

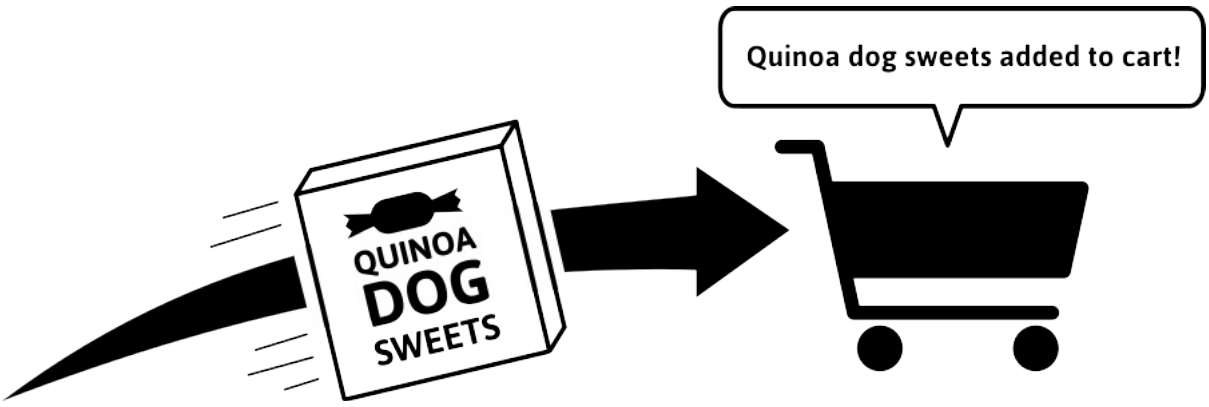
```
<div role="status" aria-live="polite">
  Take a short break!
</div>
```

Now the “Take a short break!” message’s arrival in the interface can be seen and heard simultaneously, creating a parity between the visual and (screen reader assisted) aural experience. It is not the *same* experience, but it is a *comparable* one: it serves the same purpose.

And everyone should take a periodic screen break.

Invisible live regions

Sometimes, to create an overall comparable experience, a little extra aural information may be needed as a supplement. For example, when a user clicks an ‘add to cart’ button, the interface’s response may be to animate the product moving into the cart. A direct translation of this may be a whooshing and clunking sound, but I suspect a visually hidden live region stating “product added successfully” (or similar) would be a lot clearer.



Adding a live region to a page already containing the content you wish to be announced is not reliable. There should be at least some time between the live region being appended to the DOM and the content being appended to the live region.

For simply making screen readers "say things" alongside events in your scripts, I have created a small module. Here's a hypothetical instantiation, using default settings:

```
const liveRegion = new OnDemandLiveRegion();

liveRegion.say('Take a short break!');
```

Since the script creates hidden ARIA live regions and populates them on the fly, it makes communicating to screen readers procedurally trivial. However, in most cases — and in the case of status messages especially — we want to be communicating to *users*. Not users running screen readers or users *not* running screen readers; just users. Live regions make it easy to communicate through visual and aural channels simultaneously.

A chat application

In a chat application (something like Slack, say, where most everything happens in real time) there are a number of opportunities for status messages. For example:

- Users coming online
- Users being added to channels
- Users replying to your messages
- Users ‘reacting’ to your messages

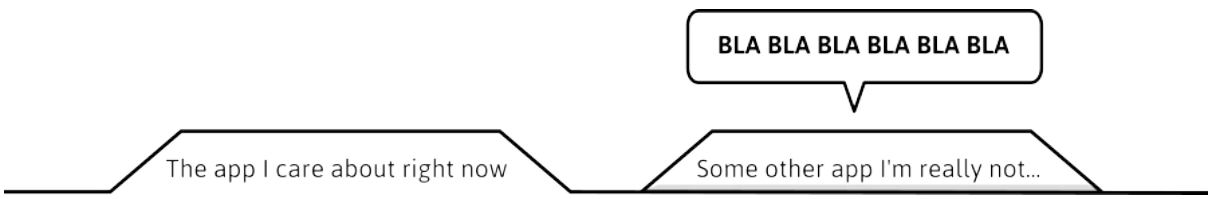
All of these types of messages coming in all the time is going to quickly become distracting and irritating, especially in their aural form. You can avert your eyes, but not your ears.

We would need to do a couple of things to make the experience more tolerable:

- Restrict messages only to suitable contexts and situations
- Give the user control over messaging verbosity

Restricting messages to contexts

Something I noticed recently while running a screen reader on one browser tab was that I could hear live regions rattling off updates from another open tab, not visible to me. The only solution was to close down the hidden tab. Not ideal, because I would have liked to switch back and forth between them.



For a sighted user, unseen is unknown. It doesn't matter if the messages keep getting displayed. But, for screen reader users (blind or otherwise), we need to silence output for hidden tabs. We can do this by querying `document.hidden` within the `visibilitychange` event from the Page Visibility API and switching the live region between active and inactive. Inactive live regions take `role="none"` and/or `aria-live="off"`.

Here's how that would work:

```
const notifications =
document.getElementById('notifications');

document.addEventListener('visibilitychange', () => {
  let setting = document.hidden ? ['none', 'off'] :
['status', 'polite'];

  notification.setAttribute('role', setting[0]);
  notification.setAttribute('aria-live', setting[1]);
});
```

Note

Your setup may vary

It's worth noting that some combinations of screen reader software and browser automatically silence at least some types of live region for hidden or unfocused tabs and windows. However, you can't rely on all your users having these setups and — where they don't — the experience is very off-putting.

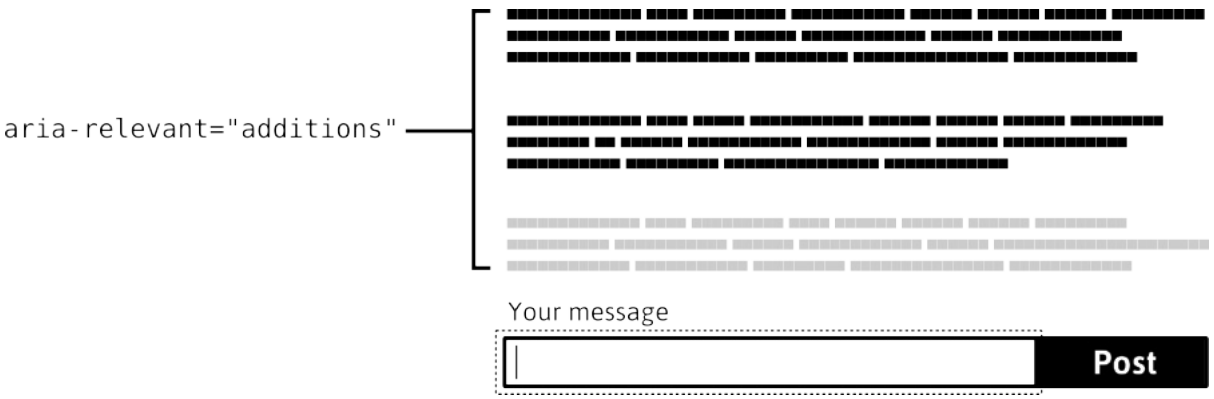
Conversations

Even when inside the open tab for the chat application, you won't want to be inundated by a flurry of any and all notifications. Visually, it could get irritating; aurally it almost *certainly* will.

Knowing when to notify the user is a question of determining what activity they are currently engaged in. For example, users probably aren't interested in the messages of users not posting in the current thread, or the arrival online of users they have no history of engaging with in the past.

On the other hand, if the user is focused on the text input for a thread and a new message pops in, they're probably going to want to know about it. In this case, the message would just appear if you're a sighted user. For a blind screen reader user, you make the new message its own notification with a live region.

The `aria-relevant` attribute controls which kinds of changes to the live region are considered worthy of readout. In this case, only newly added messages are really of interest so we set `aria-relevant="additions"` on the parent element for the message stream.



When the new mesage, in grey, appears, only its contents — and not the contents of the other messages — are announced in screen readers.

Removed or edited messages would not be re-announced, but edited messages *should* remain discoverable. Hence, the markup for messages should be well-formed and semantically clear, using a list (`<ul>`) structure to group them together.

```
<h1>Self care chat</h1>
<div role="status" aria-live="polite" aria-
relevant="additions">
  <ul class="messages">
    <li>
      <h2>Heydon, <small>22 minutes ago</small>:</h2>
      <p>Take a screen break. It's been 15 hours.</p>
    </li>
    <li>
      <h2>Heydon:</h2>
      <p>Oh, I guess you are already.</p>
    </li>
  </ul>
<div>
<form>
   <label for="message">Your message</label>
   <textarea id="message"></textarea>
   <button type="submit">Post</button>
</form>
```

When that last item is appended to the master list, screen reader users hear “Heydon: Oh, I guess you are already.” Arguably, you should append each message with the word ‘message’ to differentiate messages from other notifications. We’ll come to those shortly.

One refinement might be that, if the screen reader user is *not* focused on the text input they should only hear a new message being announced if it addresses them directly — using an “@”, say. We’re not depriving these users; we’re just not interrupting them during a

different task, unless it's a specific 'hey I need you'.

Here's some rough pseudo-code for how that logic might work:

```
if (message.includesMention()) {
  message.alert = true;
}
```

Flash messages

Flash messages — little colored strips of text that appear above the 'action' of the page — are often employed to keep users abreast of changing state. A single ARIA live region will suffice for these non-actionable notifications.

I'll come to how these should be designed shortly, but first we need to make sure they can be switched off. The first thing I do when I install an app like Skype is switch off the notification sounds, and for good reason: I find them very distracting.

The settings screen

Designing a page that houses application settings does not require a feat of engineering. It's just headings, subheadings, and form controls. But, through lack of care, you can botch the information architecture and terminology.

"General" and "Content" don't really mean anything as category names, for example. And hiding what you subjectively consider 'advanced' settings behind a tiny, hard to locate link doesn't help either.

Word everything descriptively, and structure everything logically. Use

standard form controls such as checkboxes, radio buttons, and sliders. The settings screen gives users control over how they use the application; don’t make it an afterthought.

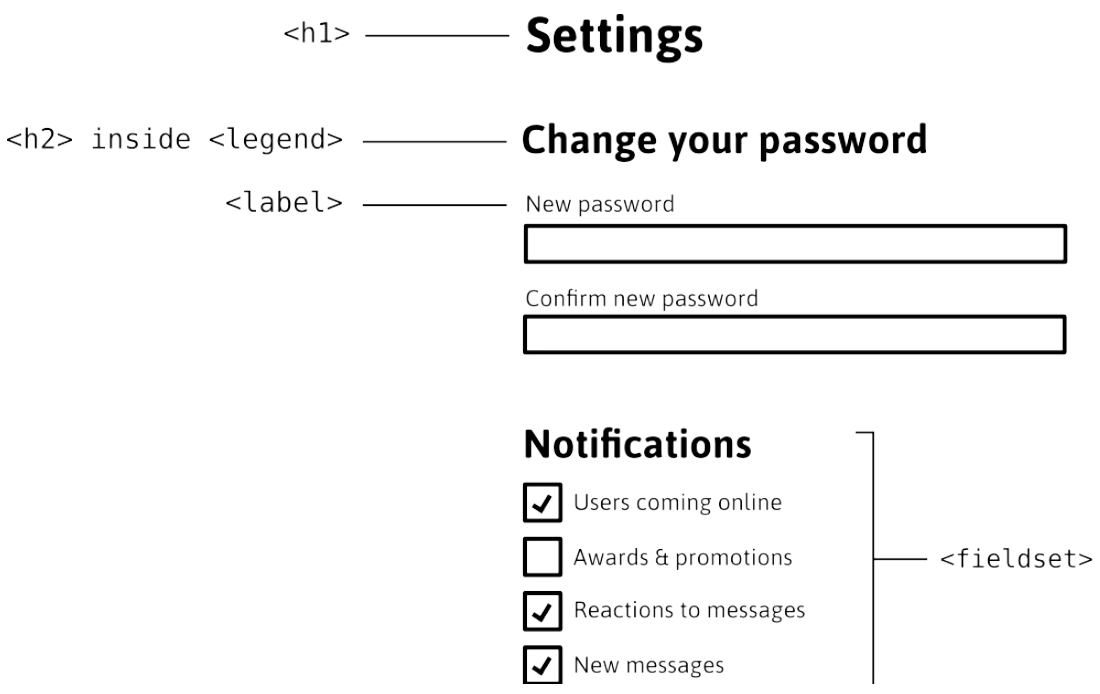
Note

Headings inside legends

When structuring (long) forms, it often helps to group related controls together inside `<fieldset>` elements. Then `<legend>` elements can be employed to provide 'group labels'. These are announced when screen reader users enter the fieldset and focus the first control. They give contextual information.

`<legend>`s tend to supplant headings, because otherwise you'd be labeling sections of the form twice. The trouble is, headings have their own advantages for screen reader navigation.

Fortunately, a recent change to the HTML spec now allows you to author pages with headings inside your `<legend>`s: the best of both worlds. Here's the sort of structure, we should be going for:



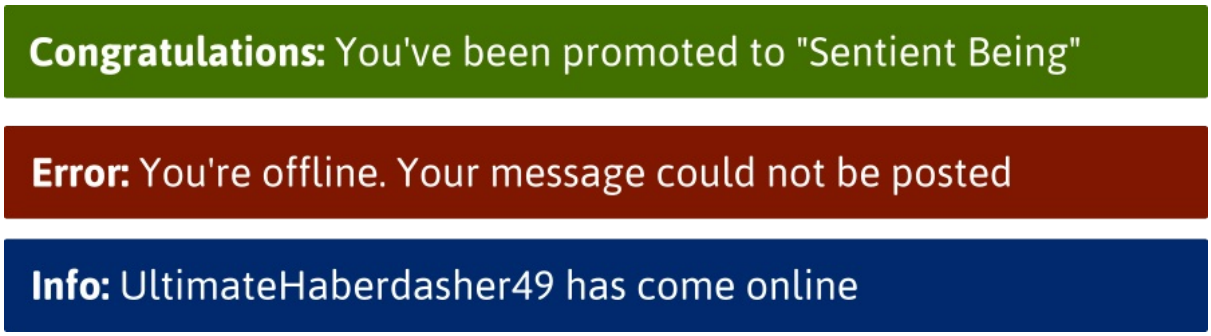
Note that turning off a notification type would mean it no longer occurs visually or aurally (in screen reader output). It's likely that

certain notifications would be much less desirable to many screen reader users, and they're more likely to turn them off. But everyone has the *same* control and can make decisions for themselves. We're not making assumptions for them.

Differentiating message types

Our singular live region may play host to a variety of notification types. Basic information will probably be most common, but there may be warnings, errors, and messages of congratulation — perhaps the user can earn awards for being a helpful member of the community.

The general rule is that **any part of an interface differentiated only by style and not content will be inaccessible**. Things like shape, color, position are just not enough on their own to define something inclusively. In this case, the MVP for differentiating messages is therefore to preface with terms like "Error:", "Info:", "Congratulations:" or whatever is suitable. A bold style is typical.



Should you wish to supplant the text with icons you'll have to be careful they are visually comprehensible, include alternative text for screen reader users, and are still visible where Windows High Contrast Mode is running.

Try an optimized, inline SVG with a `fill` set to `currentColor` to honor

high contrast mode. For alternative text, `aria-label` is not recommended because it is not picked up by translation services like Google's. The same, unfortunately, applies to any text (`<title>` or `<text>`, say) *inside* SVGs. The best we can do is insert some visually hidden text just for assistive software.

```
<div role="status" aria-live="polite">
  <div class="message award">
    <p>
      <strong>
        <svg viewBox="0 0 20 20" focusable="false">
          <use xlink:href="#star"></use>
        </svg>
        <span class="visually-hidden">Congratulations!
</span>
      </strong>
      You've been awarded 6 fake internet points
    </p>
  </div>
</div>
```

The hidden span would of course be completely invisible. The outline is shown here just to indicate its whereabouts.

Dismissing notifications

Working as a design consultant, I often see notification messages include little "✖" buttons to dismiss them.

While I always want to applaud efforts to put users in control of the interface, I'm not so sure in this case. I just don't think the ability to manually dismiss notifications is important enough to *bother* users with; it's not something worth encountering or having to think about. (There's also the issue of managing focus when the close button is removed from the DOM after being pressed, as covered in A Todo List).

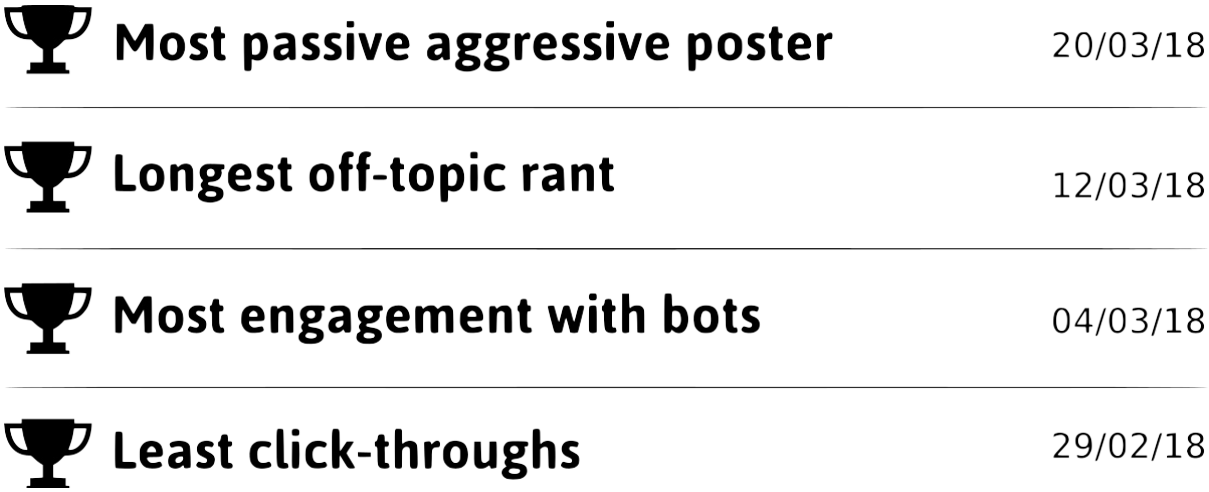
Instead, it's better the messages just disappear by themselves — after an appropriate amount of time. Here's a small script that lets you create notifications regions by type ('error', 'award', or 'info', say) and inject/remove notification messages after a chosen amount of time.

```
function Notifier(regionEl, duration, type) {
  this.regionEl = regionEl;
  this.duration = duration || 10000;
  // Info type as default
  this.type = type || 'info';
}

Notifier.prototype.notify = function(message) {
  let note = document.createElement('p');

  note.innerHTML = `
    <svg viewBox="0 0 20 20" focusable="false">
      <use xlink:href="#${this.type}"></use>
    </svg>
    <span class="visually-hidden">${this.type}:</span>
    ${message}
  `;

  this.regionEl.appendChild(note);

  // Remove after set amount of time
  window.setTimeout(() => {
    this.regionEl.removeChild(note)
  }, this.duration);
}

// Initialize
const infoNotifications = new Notifier(
  document.getElementById('notifications'),
  5000
);

// Call the notify method with a message
infoNotifications.notify('Heydon666 has joined this
group.');
```

(**Note:** The ‘type’ string is used both for the inline SVG reference and as the alternative text for the icon.)

But what if the user misses notifications come and go? Not a problem. Notifications should only refer to things that are discoverable elsewhere in the updated interface.

A couple of examples: If the notification refers to **@Heydon666** coming online, you’ll be able to discover they’re around because they have appeared in the list of active users, or have their status updated. For the ‘awards’ example, the interface should keep track in the user’s profile page. A chronology of awards is typical.



Conclusion

Thanks to the marvelous “you add it, I say it” nature of ARIA live regions, the technical implementation of inclusive notification could hardly be simpler. That leaves you to perfect the clarity of form and language.

The biggest and most important task actually has nothing to do with the notification component itself. It’s all in the structure and

presentation of the permanent history into which each notification message only offers a fleeting glimpse. As ever, structuring content is paramount, even where it pertains to dynamic events inside realtime web applications.

Checklist

- Don't use `aria-atomic="true"` unless you want all the contents of a live region announced whenever there's any change within it.
- Be judicious in your use of visually hidden live regions. Most content should be seen *and* heard.
- Distinguish parts of your interface in content or with content and style, but never just with style.
- Do not announce everything that changes on the page. A very popular carousel plugin that shall remain nameless announces the arrival of each slide as it comes into view. A huge irritant, and only comparable to a sighted user's experience if the carousel is set to track their eye movements and always remain at the center of their gaze.
- Be *very* wary of Desktop notifications for your site. I have never come across a site for which I wanted these needy and obstructive messages to be permitted.
- Be aware that live regions are a relatively new technology. In testing, I've found some users *assume* their focus has been moved by the application when they hear their screen readers announce new content. Although it's easy enough for the user to discover their context has not changed, be clear about notifications by using terminology like 'notification', 'update', or 'alert' if necessary.

Data Tables

The first thing I was told when I embarked on learning web standards about twelve years ago was, “don’t use tables for layout.” This was sound advice in spirit, but not very well qualified. As a result, there have been some unfortunate interpretations. Using table markup inevitably results in a visual layout, which has led some to abandon HTML tables altogether. Tables: *bad*.

The lesson in “don’t use tables for layout” is not to use HTML elements in ways for which they were not intended. Twelve years ago, the idea that I would be coding HTML ‘wrong’ was enough to put me off making such classic blunders. Vanity is not a real reason, though.

The real reason — the reason it’s a bad practice — is how it affects the user. Table markup, starting at `<table>` and including `<th>`, `<td>` *et al*, tells browsers to pass on certain information and produce certain behaviors. Someone using assistive software such as a screen reader will become subject to this information and behavior.

When table markup contains non-tabular content, it messes with blind

users' expectations. It's not a page layout to them; it's a data table that doesn't make sense. If they're sighted or partially sighted and running a screen reader it's *both*, which is arguably even more confusing.

Our way of judging web technologies is oddly epochal. We believe that one epoch — the epoch of CSS Flexbox, for example — should end as it ushers in the new epoch of CSS Grid. But like `<div>`-based page layouts and data tables, these are actually complementary things that can co-exist. You just need to know where to use one, and where the other.

In this chapter, I'll be exploring how to create inclusive data tables: ones that are screen reader accessible, responsive, and as ergonomic as possible for everyone. First, though, I want to show you a trick for fixing an old layout table.

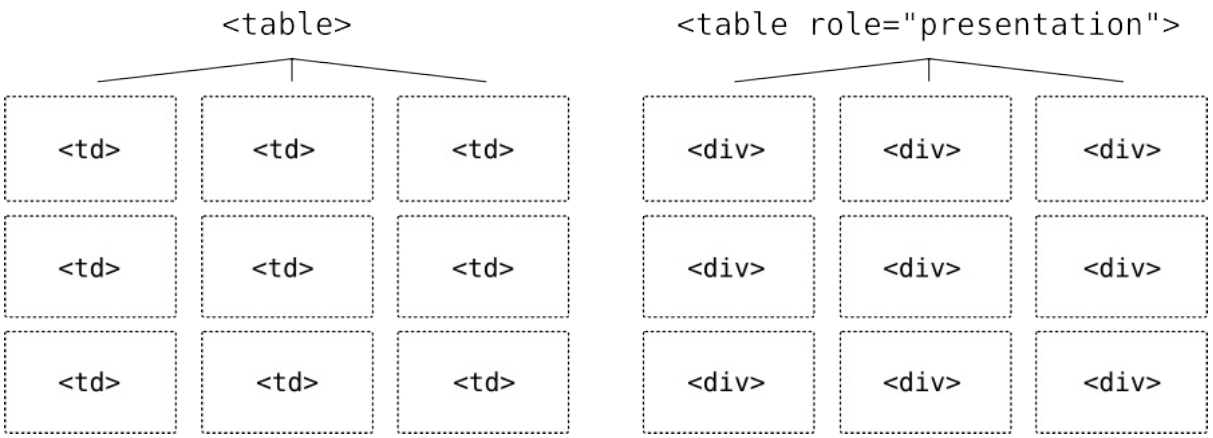
The presentation role

WAI-ARIA can be a helpful tool because it allows you to add and extend semantic information in HTML. For example, adding `aria-pressed` to a standard button makes it a toggle button to browsers and, therefore, assistive software. But did you know you can also use WAI-ARIA to take semantics away? That is, the following two elements are each semantically indeterminate to a screen reader. Neither are a 'button'.

```
<button role="presentation">Press me</button>

<span>Press me</span>
```

Most of the time you'll only want to add semantics where they are useful, rather than choosing elements for their appearance and removing the semantics where they aren't needed. But sometimes reverse engineering accessibility information is the most efficient way to make good of a bad decision like a layout table.

Applying `role="presentation"` to a `<table>` element removes all of that table's semantics, and therefore elicited behaviors, in screen readers. It is as if it was constructed using semantically unassuming `<div>`s all along.



Note that `role="presentation"` and `role="none"` are synonymous. The first is more longstanding and better supported.

In 2018, there are much better layout solutions than `<table>`s anyway, so there's no advantage in using them for any new layout you're trying out.

True data tables

A typical layout table consists of a `<table>` container, some `<tr>`s, and some `<td>`s inside them.

```
<table>
    <tr>
        <td><img src="some/image" alt=""></td>
        <td>Lorem ipsum dolor sit amet.</td>
    </tr>
    <tr>
        <td><img src="some/other/image" alt=""></td>
        <td>Integer vitae blandit nisi.</td>
    </tr>
</table>
```

The semantics issue to one side, these are all the elements you really need to achieve a visual layout. You have your rows and columns, like a grid.

Unfortunately, even where our *intention* is to manufacture a data table, we still tend to think visually only: “If it looks like a table, it’s fine.” But the following does not make an accessible table.

```
<table>
    <tr>
        <td>Column header 1</td>
        <td>Column header 2</td>
    </tr>
    <tr>
        <td>Row one, first cell</td>
        <td>Row one, second cell</td>
    </tr>
</table>
```

Why? Because our column headers — semantically speaking — are

just bog standard table elements. There's nothing here to explicitly say they are headers except the text (which is likely to be less clear in a real example than "Column header 1"). Instead, we need to make them `<th>` elements.

```
<table>
    <tr>
        <th>Column header 1</th>
        <th>Column header 2</th>
    </tr>
    <tr>
        <td>Row one, first cell</td>
        <td>Row one, second cell</td>
    </tr>
</table>
```

Using column headers in this way is not just to be 'semantically correct'. There is a manifest effect on screen reader behavior. Now, if I use my screen reader to navigate to a row cell, it will read out the header under which it sits, letting me know which column I am currently in.

Row headers

It's possible to have both column and row headers in data tables. I can't think of any kind of data for which row headers are strictly *necessary* for comprehension, but sometimes it feels like the *key* value for a table row should be on the left, and highlighted as such.

Band	Singer	Inception	Label
Napalm Death	Barney Greenway	1981	Century Media
Carcass	Jeff Walker	1985	Earache
Extreme Noise Terror	Dean Jones	1985	Candlelight
Discordance Axis	Jon Chang	1982	Hydrahead

The trouble is, unless you state it explicitly, it isn't clear whether a header labels cells below it or to its right. That's where the `scope` attribute comes in. For column headers you use `scope="col"` and for row headers you use `scope="row"`.

Here's an example for fuel prices that I was working on for Bulb recently.

```
<table>
    <tbody>
        <tr>
            <th scope="col">Region</th>
            <th scope="col">Electricity</th>
            <th scope="col">Gas</th>
        </tr>
        <tr>
            <th scope="row">East England</th>
            <td>10.40</td>
            <td>2.31</td>
        </tr>
        <tr>
            <th scope="row">East Midlands</th>
            <td>10.55</td>
            <td>2.77</td>
        </tr>
        <tr>
            <th scope="row">London</th>
            <td>10.10</td>
            <td>2.48</td>
        </tr>
    </tbody>
</table>
```

Note that not setting row headers does not make a nonsense of the data; it just adds extra clarity and context. For a table that uses both column and row headers, some screen readers will announce both the column and row labels for each of the data cells.

Note

Using tables with screen readers

Complex interfaces and widgets tend to have special behaviors and associated keyboard shortcuts in screen readers, and tables are no different.

JAWS, NVDA, and VoiceOver each provide the T key to move between tables on the page. To navigate between table cells, you use your arrow keys. When arriving at a table, you are typically informed of how many columns and rows it contains. The `<caption>`, if present, is also read out.

When you switch between cells across columns, the new column header is announced, along with the numeric placement of the column (e.g. “column 3 of 4”), and the content of the cell itself. When you switch between cells across rows, the new row header is announced, along with the numeric placement of the row (e.g. “row 5 of 8”), and the content of the cell itself.

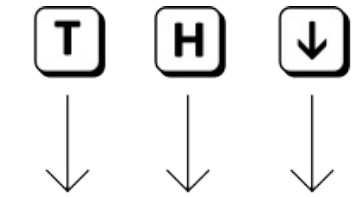
Video: Demonstration of traversing a data table with VoiceOver on OSX with Safari

Captions

There used to be two ways to provide descriptive information directly to tables: `<caption>` and `<summary>`. The `<summary>` element was deprecated in HTML5, so should be avoided. The `<caption>` element is superior regardless, because it provides a visual and screen reader accessible label. The `<summary>` element works more like an `alt` attribute and is not visible. Since the table itself provides textual information, such a summary should not be necessary.

Not all tables necessarily need captions, but it's recommended you either provide a caption or precede the table with a heading. That is, unless the table is inside a `<figure>` with a `<figcaption>`. As the name suggests, the `<figcaption>` is a kind of caption on its own, and will suffice.

The advantage of a caption over a heading is that it is read out when a screen reader user encounters the table directly, using the T shortcut key. Fortunately HTML5 lets you place headings *inside* captions, which is the best of both worlds and highly recommended where you know what level the heading should be ahead of time.



Grindcore bands

Band	Singer	Inception	Label
Napalm Death	Barney Greenway	1981	Century Media
Carcass	Jeff Walker	1985	Earache

By using a heading inside the table `<caption>`, there are now three ways to discover the table: by table shortcut, heading shortcut, or just by browsing downwards.

A data-driven table component

That pretty much covers basic tables and how to make them accessible. The trouble is, they're such a pain to code by hand, and most WYSIWYG tools for creating tables do not output decent markup, with the necessary headers in the correct places.

Instead, let's create a component that accepts data and outputs an accessible table automatically. In React, we can supply the headers and rows as props. In the headers' case we just need an array. For the rows: an array of arrays (or "two-dimensional" array).

```
const headers = ['Band', 'Singer', 'Inception',
'Label'];

const rows = [
  ['Napalm Death', 'Barney Greenway', '1981', 'Century
Media'],
  ['Carcass', 'Jeff Walker', '1985', 'Earache'],
  ['Extreme Noise Terror', 'Dean Jones', '1985',
'Candlelight'],
  ['Discordance Axis', 'Jon Chang', '1992',
'Hydrahead']
];
```

Now the `Table` component just needs those `const`s passed in.

```
<Table rows={rows} headers={headers} />
```

One of the best and worst things about HTML is that it's forgiving. You can write badly formed, inaccessible HTML and the browser will still render it without error. This makes the web platform inclusive of beginners, and those creating rule-breaking experiments. But it doesn't hold us to account when we're trying to create well-formed code that's compatible with all parsers, including assistive technologies.

By deferring the well-formed part to arrays, which expect a very specific structure, we can catch errors there. Where the arrays are well-

formed, we can generate accessible markup from them, automatically.

Here's how the basic component that handles this might look:

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          {this.props.headers.map((header, i) =>
            <th scope="col" key={i}>{header}</th>
          )}
        </tr>
        {this.props.rows.map((row, i) =>
          <tr key={i}>
            {row.map((cell, i) =>
              <td key={i}>{cell}</td>
            )}
          </tr>
        )}
      </table>
    );
  }
}
```

If you don't supply arrays for the `headers` and `rows` props things are going to go spectacularly wrong, so if you dig *'not a function'* errors, you're in for a fun time.

Perhaps, though, it would be better to catch those errors early and output a more helpful message. That's where 'prop types' can be useful.

```
Table.propTypes = {
  headers: PropTypes.array.required,
  rows: PropTypes.array.required
};
```

Of course, if you're using Typescript, you'll probably be handing this with an interface instead. I personally find the extreme rigidity and perplexing syntax of Typescript in React *a bit much*. I'm told it's great for when you're writing complex enterprise software, but if you mostly deal with small projects and codebases, life is probably too short.

Supporting row headers

Supporting the option of row headers is a cinch. We just need to know if the author has included a `rowHeaders` prop. Then we can transform the first cell of each row into a `<th>` with `scope="row"`.

```
<tr key={i}>
  {row.map((cell, i) =>
    (this.props.rowHeaders && i < 1) ? (
      <th scope="row" key={i}>{cell}</th>
    ) : (
      <td key={i}>{cell}</td>
    )
  )}
</tr>
```

In my table about grindcore bands, this makes a lot of sense since the bands named down the left hand side are the basis for all the other information.

Here's the full script for the basic table component, coming in at just 25 lines. Use it however you wish.

```
export default class Table extends React.Component {
  render() {
    return (
      <table>
        <caption>{this.props.caption}</caption>
        <tr>
          {this.props.headers.map((header, i) =>
            <th scope="col" key={i}>{header}</th>
          )}
        </tr>
        {this.props.rows.map((row, i) =>
          <tr key={i}>
            {row.map((cell, i) =>
              (this.props.rowHeaders && i < 1) ? (
                <th scope="row" key={i}>{cell}</th>
              ) : (
                <td key={i}>{cell}</td>
              )
            )}
          </tr>
        )}
      </table>
    );
  }
}

// Data
const headers = ['Band', 'Singer', 'Inception',
'Label'];

const rows = [
  ['Napalm Death', 'Barney Greenway', '1981', 'Century
Media'],
  ['Carcass', 'Jeff Walker', '1985', 'Earache'],
```

```
  ['Extreme Noise Terror', 'Dean Jones', '1985',
'Candlelight'],
  ['Discordance Axis', 'Jon Chang', '1992',
'Hydrahead']
];

// Initialization
<Table rows={rows} headers={headers} rowHeaders
caption="Grindcore bands" />
```

Going responsive

Responsive tables are one of those areas where the accessible solution is more about what you don't do than what you do. As Adrian Roselli recently noted, using CSS display properties to change table layout has a tendency to remove the underlying table semantics. This probably shouldn't happen, because it messes with the separation of concerns principle. **But it happens anyway**.

This isn't the only reason it's a bad idea to change the way tables are displayed. Visually speaking, it's not really the same table — or much of a table at all — if the columns and rows collapse on top of one another. Instead, we want to provide access to the same visual and semantic structure regardless of the space available.

It's as simple as letting the table's parent element scroll horizontally.

```
.table-container {
  overflow-x: auto;
}
```

Keyboard support

Okay, it's not *quite* that simple. As you may recall from A Content Slider, we need to make the scrollable element focusable so it can be operated by keyboard. That's just a case of adding `tabindex="0"`. But since screen reader users will be able to focus it too, we need to provide some context for them.

In this case, I'll use the table's `<caption>` to label the scrollable region using `aria-labelledby`.

```
<div class="table-container" tabindex="0" role="group"
aria-labelledby="caption">
  <table>
    <caption id="caption">Grindcore bands</caption>
    <!-- table content -->
  </table>
</div>
```

You can't use `aria-labelledby` just anywhere. The element has to have an appropriate `role`. Here I'm using the fairly generic `group` role for this purpose. From the spec' on `group`: *"A set of user interface objects which are not intended to be included in a page summary or table of contents by assistive technologies."*

Only focusable where scrollable

Of course, we don't want to make the table container focusable *unless* its contents overflow. Otherwise we're adding a tab stop to the focus order which doesn't do anything. In my opinion, that would be a fail under WCAG 2.4.3: Focus Order. Giving keyboard users elements to focus which don't actually do anything is confusing and obstructive.

What we can do is detect whether the content overflows on page load (or the component mounting) by adding `tabindex="0"` only if `scrollWidth` exceeds `clientWidth` for the container. We can use a ref (`this.container`) for this purpose.

```
componentDidMount() {
  const {scrollWidth, clientWidth} = this.container;
  let scrollable = scrollWidth > clientWidth;
  this.setState({
    tabindex: scrollable ? '0' : null
  });
}
```

(Thanks to Almero Steyn for the note on string `ref` deprecation. As he pointed out, in React 16.3 you define the `ref` in the constructor like `this.container = React.createRef();`. Then you just need to add `ref={this.container}` on the container element.)

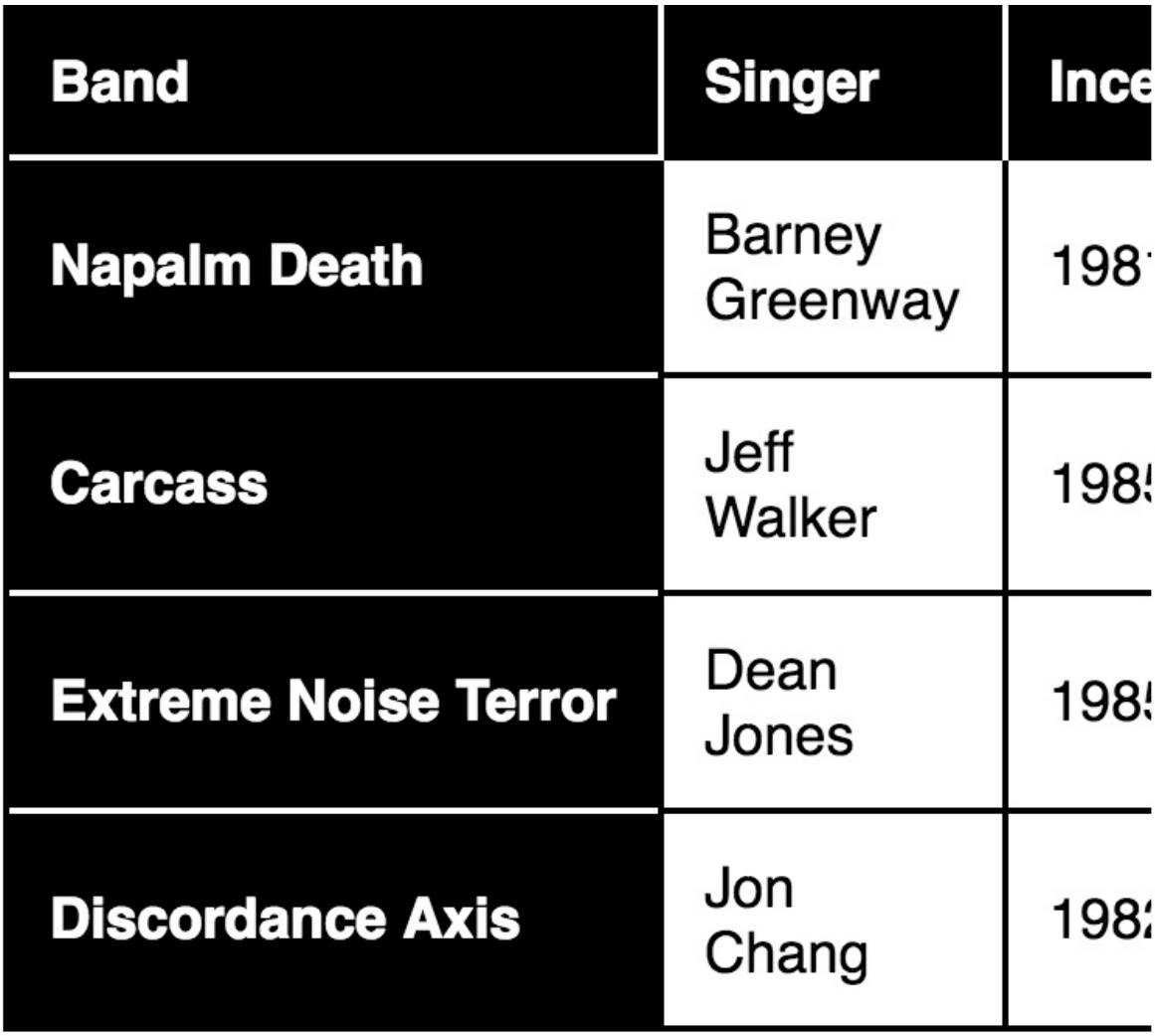
Here's a truncated version of the script, showing how I use state to switch the `tabindex` value via the `componentDidMount` lifecycle function.

```
class Table extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      tabindex: null
    }
  }
  componentDidMount() {
    let container =
ReactDOM.findDOMNode(this.refs.container);
    let scrollable = container.scrollWidth >
container.clientWidth;
    this.setState({
      tabindex: scrollable ? '0' : null
    });
  }
  render() {
    const captionID = 'caption-' +
Math.random().toString(36).substr(2, 9);
    return (
      <div
        className="table-container"
        ref="container"
        tabIndex={this.state.tabindex}
        aria-labelledby={captionID}
      >
        <!-- table here -->
      </div>
    );
  }
}
```

Perceived affordance

It's not enough that users can scroll the table. They also need to know they *can* scroll the table. Fortunately, given our table cell border style,

it should be obvious when the table is cut off, indicating that some content is out of view.



Band	Singer	Ince
Napalm Death	Barney Greenway	198
Carcass	Jeff Walker	198
Extreme Noise Terror	Dean Jones	198
Discordance Axis	Jon Chang	198

We can do one better, just to be safe, and hook into the state to display a message in the caption:

```
{this.state.tabindex === '0' &&
  <div>
    <small>(scroll to see more)</small>
  </div>
}
```

Grindcore bands
(scroll to see more)

Band	Singer	Incepti
Napalm Death	Barney Greenway	1981
Carcass	Jeff Walker	1985
Extreme Noise Terror	Dean Jones	1985
Discordance Axis	Jon Chang	1982

This text will also form part of the scrollable container's label (via the `aria-labelledby` association discussed earlier). In a screen reader, when the scrollable container is focused you will hear something similar to *"Grindcore bands, open parenthesis, scroll to see more, close parenthesis, group."* In other words, this extra message adds clarification non-visually too.

Very narrow viewports

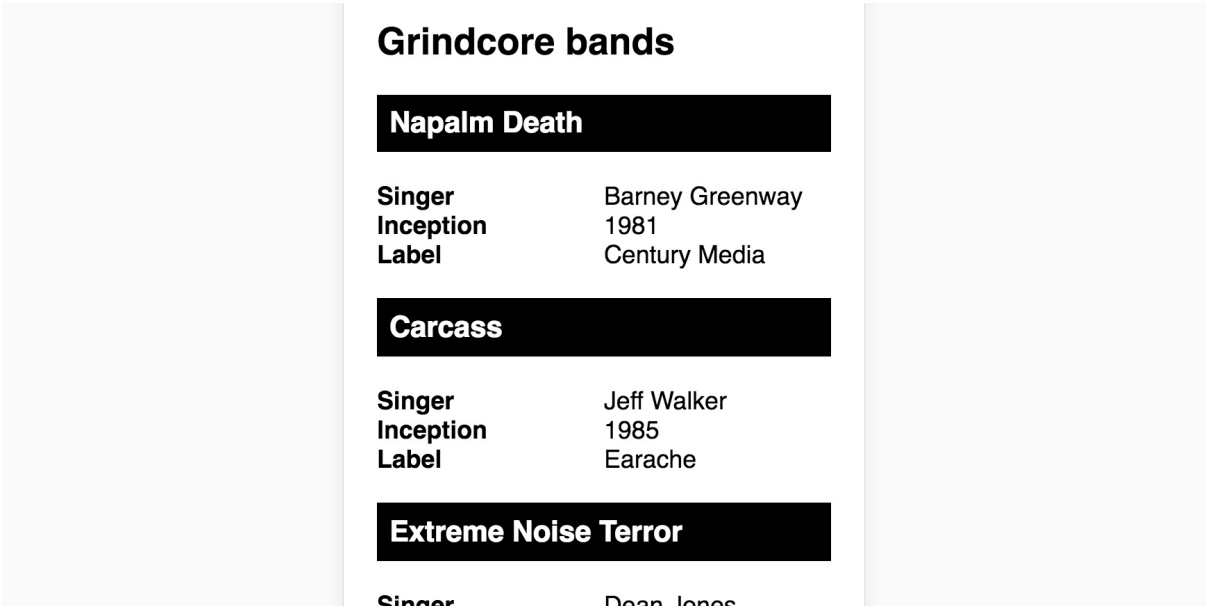
The preceding works for wide tables (with many columns) or narrow

viewports. *Very* narrow viewports might want something a bit more radical, though. If you can barely see one column at a time, the viewing experience is pretty terrible — even if you *can* physically scroll the other columns into view by touch.

Instead, for very narrow (one column) viewports, we can present the data using a different structure, with headings and definition lists.

- `<caption>` → `<h2>`
- `<th scope="row">` → `<h3>`
- `<th scope="col">` → `<dt>`
- `<td>` → `<dd>`

This structure is much more suited to mobile, where users are more accustomed to scrolling vertically. It's also accessible, just in a different way.



Here's what the JSX might look like:

```
<div className="lists-container">
  <h2>{this.props.caption}</h2>
  {this.props.rows.map((row, i) =>
    <div key={i}>
      <h3>{row[0]}</h3>
      <dl>
        {this.props.headers.map((header, i) =>
          i > 0 &&
          <React.Fragment key={i}>
            <dt>{header}</dt>
            <dd>{row[i]}</dd>
          </React.Fragment>
        )}
      </dl>
    </div>
  )}
</div>
```

Note the use of `Fragment`. This allows us to output the unwrapped sibling `<dt>` and `<dd>` elements for our `<dl>` structure. A recent change to the spec' has made it permissible to wrap `<dt>`/`<dd>` pairs in `<div>`s (thanks to Gunnar for contacting me about this). But we can't be sure it won't cause parsing issues for now, and we don't need the wrappers here anyway.

All that's left to do is show/hide the equivalent interfaces at the appropriate viewport widths. For example:

```
@media (min-width: 400px) {
  .table-container {
    display: block;
  }

  .lists-container {
    display: none;
  }
}
```

For extremely large data sets, having both interfaces in the DOM will bloat an already large DOM tree. However, in most cases this is the more performant solution compared with dynamically reconstituting the DOM via `matchMedia` or (worse still) listening to the `resize` event.

If you're loading dynamic data, you don't have to worry about the two interfaces staying in sync: they are based directly on the same source.

Sortable tables

Let's give users some control over how the content is sorted. After all, we already have the data in a sortable format — a two-dimensional array.

Of course, with such a small data set, just for demonstration purposes, sorting is not really needed. But let's implement it anyway, for cases where it does make things easier. The great thing about React props, is we can easily turn the functionality on or off.

Inside each column header we can provide a sorting button:

Band ↕	Singer ↕	Inception ↕	Label ↕

These can toggle between sorting the data by the column in either an ascending or descending order. Communicating the sorting method is the job of the `aria-sort` property. Note that it works most reliably in conjunction with an explicit `role="columnheader"`.

Here's the inception column, communicating an ascending sort (lowest value top) to screen readers. The other possible values are `descending` and `none`.

```
<th scope="col" role="columnheader" aria-
sort="ascending">
    Inception
    <button>sort</button>
</th>
```

Not all screen readers support `aria-sort`, but a sorting button label of *"sort by [column label]"* makes things clear enough to those who do not have the sorting state reported. You could go one better by adapting the label to *"sort by [column label] in ['ascending'|'descending'] order"*.

```
<button onClick={() => this.sortBy(i)}>
  <span className="visually-hidden">
    sort by {header} in
    {this.state.sortDir !== 'ascending'
      ? 'ascending'
      : 'descending'}
    order
  </span>
</button>
```

Iconography

Visually, the sort order should be fairly clear by glancing down the column in hand, but we can go one better by providing icons that communicate one of three states:

- ↕ = it's sortable, but not sorted
- ↑ = It's sorted by this column, in ascending order
- ↓ = It's sorted by this column, is descending order

As ever, it's advantageous to use an SVG.

- SVGs scale without degredation, making zoom more pleasant
- SVGs using `currentColor` respect Windows High Contrast settings
- SVGs can be constructed efficiently from shape and line elements
- SVGs are markup and their different parts can be targeted individually

That last advantage is not something I've explored in this book before,

but is ideal here because each arrow is made of two or more lines. Consider the following `Arrow` component.

```
const Arrow = props => {
  let ascending = props.sortDir === 'ascending';
  return (
    <svg viewBox="0 0 100 200" width="100"
height="200">
      {!(!ascending && props.isCurrent) &&
        <polyline points="20 50, 50 20, 80 50">
</polyline>
      }
      <line x1="50" y1="20" x2="50" y2="180"></line>
      {!(ascending && props.isCurrent) &&
        <polyline points="20 150, 50 180, 80 150">
</polyline>
      }
    </svg>
  );
}
```

Logic is passed in from the parent component via props (`sortDir` and `current`) to conditionally show the different `polyline` arrow heads. For example, the final `polyline` is only shown if the following are true.

- The sort order isn't ascending
- This isn't the current sorting column

Warning: Technically here I am using the arrow to express the button's current state, not the state pressing it will achieve. In many circumstances (and as discussed in Toggle Buttons) this is a mistake. The important thing here is the *change* in arrow direction as one

toggles, communicating the switch in polarity.

Note

A note on the grid role

WAI-ARIA provides a role, `grid`, that is closely associated with tables. This role is intended to be paired with specific keyboard behavior, letting keyboard users navigate table cells as they would be able via screen reader software (using their arrow keys).

You do *not* need to use the `grid` role to make most tables accessible to screen readers. The `grid`-related behavior should only be implemented where users *not* running screen reader software need to easily access each cell to interact with it. One example might be a date picker where each date is clickable within a grid representation of a calendar month.

Performance

The sorting function itself should look something like this, and uses the `sort` method. Note that the Edge browser does not support returning Booleans for sort methods, hence the explicit `1`, `-1`, or `0` return values.

```
  sortBy(i) {
    let sortDir;
    let ascending = this.state.sortDir === 'ascending';
    if (i === this.state.sortedBy) {
      sortDir = !ascending ? 'ascending' :
'descending';
    } else {
      sortDir = 'ascending';
    }
    this.setState(prevState => ({
      rows: prevState.rows.slice(0).sort((a, b) => {
        if (sortDir === 'ascending') {
          return a[i] > b[i] ? 1 : a[i] < b[i] ? -1 :
0;
        } else {
          return a[i] < b[i] ? 1 : a[i] > b[i] ? -1 :
0;
        }
      }),
      sortedBy: i,
      sortDir: sortDir
    }));
  }
```

Note the use of `slice(0)`. If this were not present, the `sort` method would augment the original data directly (which is an unusual characteristic peculiar to `sort`). This would mean both the table *and* the mobile-width list structure would be rebuilt in the DOM. Since there are no sorting controls provided for the list structure, this is an unnecessary performance hit.

Demo

The complete demo, including row headers, selective scrolling, the alternative representation for mobile, and the sorting functionality is available on Github.

- Code on Github
- Demo page

Conclusion

Yes, it's still okay to use tables. Just don't use them if you don't need them and, when you *do* need them, structure them in a logical and expected way.

Checklist

- Don't use tables just for layout or, to be more clear, **don't use tables for anything but tabular data**.
- Always include at least column headers *or* row headers.
- Sorting functionality is nice, but don't include it if it isn't needed. The 'Grindcore bands' example doesn't really need sorting because there's not much data in total. Allow switching it on or off with a `sortable` prop.
- Make sure the visual design of the table is clear, with obvious divisions between cells, and highlighted headers. To make it easier to scan rows, you may want to consider alternating row colors for a

‘zebra’ effect.

Modal Dialogs

One component I get asked to write about a lot is the **modal dialog**. I have mixed feelings about this, because the proliferation of dialogs in web user interface design is something of a scourge. In my talk "Writing Less Damned Code" I joke that the fewer dialogs you use in your project, the more there are available for Twitter's web UI — an interface almost entirely made of dialogs.

As with the tabbed interface and content slider components, it's important to address dialogs *because* they're contentious and problematic. Pretending they don't exist isn't going to make them go away, and it isn't doing anything to improves the ones that inevitably stick around.

The three main things I want to focus on here are:

1. Strong cases against dialogs, for many of the popular use cases
2. Strong alternatives for these vetoed use cases
3. Strong dialogs, for the remaining cases (or for when you lose the

argument in point(1))

I've written about the technical implementation of accessible ARIA dialogs before, in *Apps For All*. Here, I'll explore some alternative takes in light of recent advancements in the web platform, performance considerations, and in the context of inclusive design thinking and process.

Dialogs and modal dialogs

Put simply, a modal dialog is one that changes the interface *mode* to prioritize itself. To achieve this, it temporarily disables the rest of the interface. Here's Therese Fessenden of the Nielsen Norman Group on modal dialogs:

A modal dialog is like my cat, Emma — who meows at 7am every morning to prompt me to feed her. I might be trying to sleep or get ready for the day, but my cat will place herself in front of me, then meow louder and incessantly until I look at her.

And here they are on non-modal dialogs, by way of comparison:

A nonmodal dialog is like a kitty who patiently sits near the dinner table during a meal, waiting on the off chance that food scraps may fall from the table. When Emma is doing this, I can eat, have a conversation, and enjoy dinner without much interruption.

It's a cute and pretty serviceable analogy I think, but it makes an

important assumption: in either case, the dialog in question has not been invoked by the user. Where this is the case, the modal dialog would indeed be more intrusive and irritating. It creates an impasse and forces you to deal with it. The *"Please turn off your ad blocker to view our ad-ridden content or please leave"* modal is an infamous example.

That's not to say the uninvited non-modal dialog isn't irritating as well. Have you ever been happy to continue reading and operating an interface with dialogs floating around on top of it? Of course not: you curse as you close them all first, and hope they don't come back.

Dialogs should *only* be invoked as the direct result of a user action or critical change in system state. And they should *only* be invoked if that action or change of state necessitates immediate further action (such as a confirmation or critical choice). To ensure this further action is

taken immediately, all other functionality in the interface must be put on hold.

If you're joining up the dots, you should have already reached an important conclusion: **there's never any need for non-modal dialogs**. Don't use them, unless you're looking for disgruntled and distrusting users.

Note that the draggable boxes of applications like make8bitart.com are not dialogs in terms of purpose, and would not, therefore, take ARIA's `role="dialog"` or associated behaviors. They may float, have close buttons, and be 'draggable', but they're really just positionable sidebars.

The `confirm()` method

As I've already attested, modal dialogs are only suitable when the application needs urgent input from the user. This makes a strong case for using the `confirm()` method, which provides this functionality natively. When you use native controls and features, your interface is

more efficient, reliable, and familiar. You sacrifice a branding opportunity, since these interfaces are not styleable, but does that really matter?

`confirm()` has all the expected semantics and behaviors of an accessible modal out of the box, which are as follows:

- Focus is moved from the last focused element to the dialog when it is opened.
- The dialog content is announced immediately and the controls are identified.
- Both clicking 'Cancel' and hitting Esc close the dialog.
- Focus is moved back to the element that had focus before the dialog opened upon closing the dialog.
- While the dialog is open, no content on the page is identifiable to assistive technologies.
- While the dialog is open, no interactive content outside of the dialog is focusable.

You may recall I mentioned the possibility of implementing confirmation dialogs for the todo list chapter. Well, let's do that now. It doesn't take much to adapt the Vue script. All I need to do is intervene within the `remove()` method:

```
remove(index, name) {
  if (window.confirm(`Are you sure you want to delete
"${name}"?`)) {
    this.todos.splice(index, 1);
    this.feedback = `${name} deleted`;
    document.getElementById('todos-label').focus();
  }
}
```

(**Note:** Default behavior is to move focus back to the invoking control when the `confirm()` closes. In our case, that control will have disappeared. As in the original **A Todo List** demo, focus is moved back to the heading above the list.)

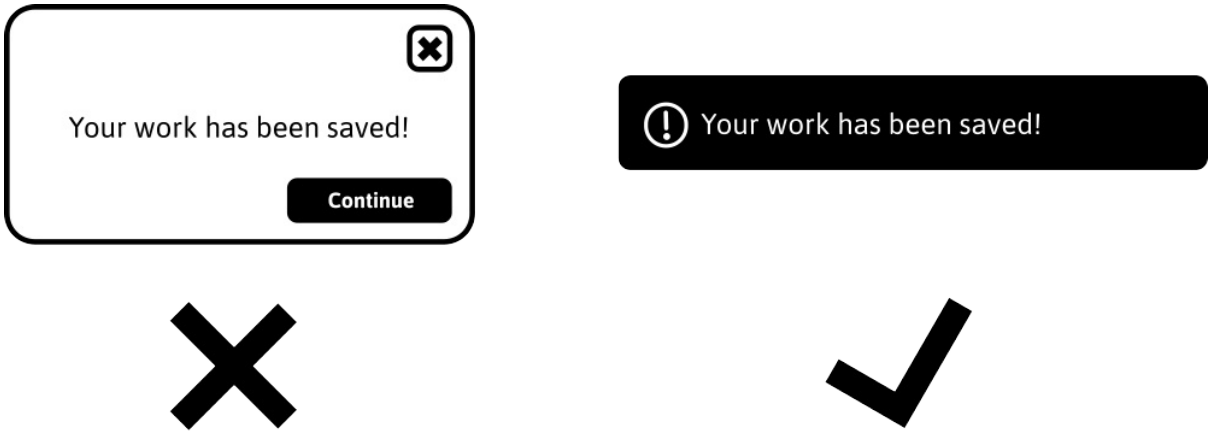
Demo: A todo list, with confirmation dialogs for deletion

Questions, not statements

Although you can write anything as your message in a `confirm()` modal, the provision of **OK** and **Cancel** buttons implies that a question is beng asked: *"Do you want to do this, or not?"*. Of course, it doesn't have to be phrased as a question (*"You are about to delete the [x] todo item"*) but it's still soliciting for an answer. That's why it exists.

In short: if your dialog is just some text and a close button, it probably shouldn't be a dialog. For updates and status messages, I have you covered in the Notifications chapter. As I explain there, diverting the user's focus and forcing them to take action is a jarring way to simply

inform them of something that has already happened.



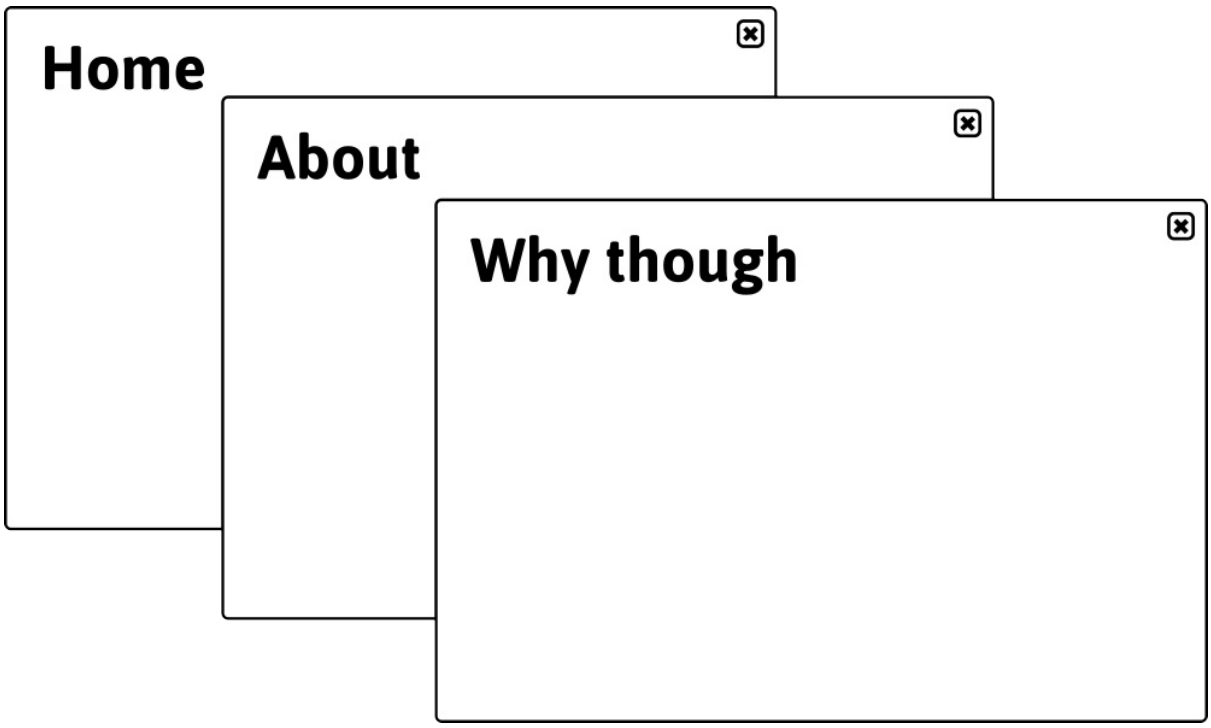
Oh, and *if* your dialog has explicit **OK** and **Cancel** buttons, or similar, it doesn't need a close button as well. As ubiquitous as they are, those little ✖ buttons are a sign of your modal dialog's obsolescence.

Modals or screens?

One of the nice things in the last `confirm()` example is how I can write imperatively. By using just a simple `if` clause, I invoke a complete—and quite accessible!—intermediary UI.

But what if I wanted to customize my modal and make it do *more*? I wouldn't be the first. I've seen whole forms placed inside modal dialogs — even 10,000-word terms and conditions statements, complete with the necessary `overflow-y: scroll` styling.

Just don't. If the content is long or complex, it needs its own page or screen. Placing a whole page's worth of content in a positioned box *above* another page is, frankly, dysfunctional behavior. It's ugly, awkward to cram into small screens, and completely unnecessary.



One arguable advantage of having a modal is that the (dimmed) content behind the modal indicates that the user is still on that page, and will be returned to it once the modal has been attended to. But too often the modal takes the user elsewhere anyway, so it might as well be an intermediary page instead.

In single-page applications especially, it is trivial to route the user between screens according to their decision-making and the resultant changes to state. A timeline showing steps to complete and an accessible indicator of the current location is how we solved this problem at Bulb. The markup (with a few Styled Components classes removed for brevity) is like this:

```
<aside aria-labelledby="your-progress">
  <h2 id="your-progress" class="visually-hidden">Your
progress</h2>
  <ol>
    <li class="current">
      <span>Quote</span>
      <span class="visually-hidden">(you are here)
</span>
    </li>
    <li><span class="text">My Information</span></li>
    <li><span class="text">Payment Details</span></li>
  </ol>
</aside>
```

- The container is an `<aside>`, meaning it's listed among the page's landmarks and is easily discoverable non-visually.
- The `<h2>` is not visible, because it is only really needed for clarification (and a navigational target) in a non-visual medium.
- The steps are an ordered list, indicating a linear continuum.
- The 'current' step has some additional visually hidden text reading "(you are here)". The ARIA alternative would be to include the attribution `aria-current="step"`, but we found this to be less well supported.

I think we can agree this is better than a succession of modal dialogs stacking up. But that's not to say others haven't taken that approach in the past. I've seen it in client websites, for example. Chances are you will need a modal dialog as part of your design system. Just don't use it for more than it should have to handle.

Custom modals

Well, it's happened: a requirement for the design system to include a custom modal dialog has arrived (that was quick). So let's set about making one as efficiently and accessibly as possible. The implementation to follow will closely resemble the behavior (and brevity) of a `confirm()`, but using my own HTML, CSS, and JavaScript.

There are a number of problems to solve in order to make the custom modal dialog similarly robust as `confirm()` and I'll attend to these in turn.

The markup

The markup for a straightforward dialog is quite simple. The container needs `role="dialog"` and the buttons have to be—you guessed it —`<button>` elements. The text acts as a label for the container by connecting it up with `aria-labelledby`. I'm creating the dialog on the fly, with JavaScript, because we don't need it until we need it.

```
// Unique identifier for the text's `id`
const unique = +new Date();

// Create the dialog
const dialog = document.createElement('div');
dialog.setAttribute('role', 'dialog');
dialog.setAttribute('aria-labelledby', `q-${unique}`);
dialog.innerHTML = `
  <p id="q-${unique}">${question}</p>
  <div class="buttons">
    <button class="okay">Okay</button>
    <button class="cancel">Cancel</button>
  </div>
`;

// Append the dialog to the <body>
document.body.appendChild(dialog);
```

Modality

This next part can be tricky, depending on your approach. We need to disable the rest of the page when the modal is active. But that doesn't just mean fading the page out (or applying a similar effect). It needs to be unidentifiable to assistive technologies, and none of the interactive contents can be clickable.

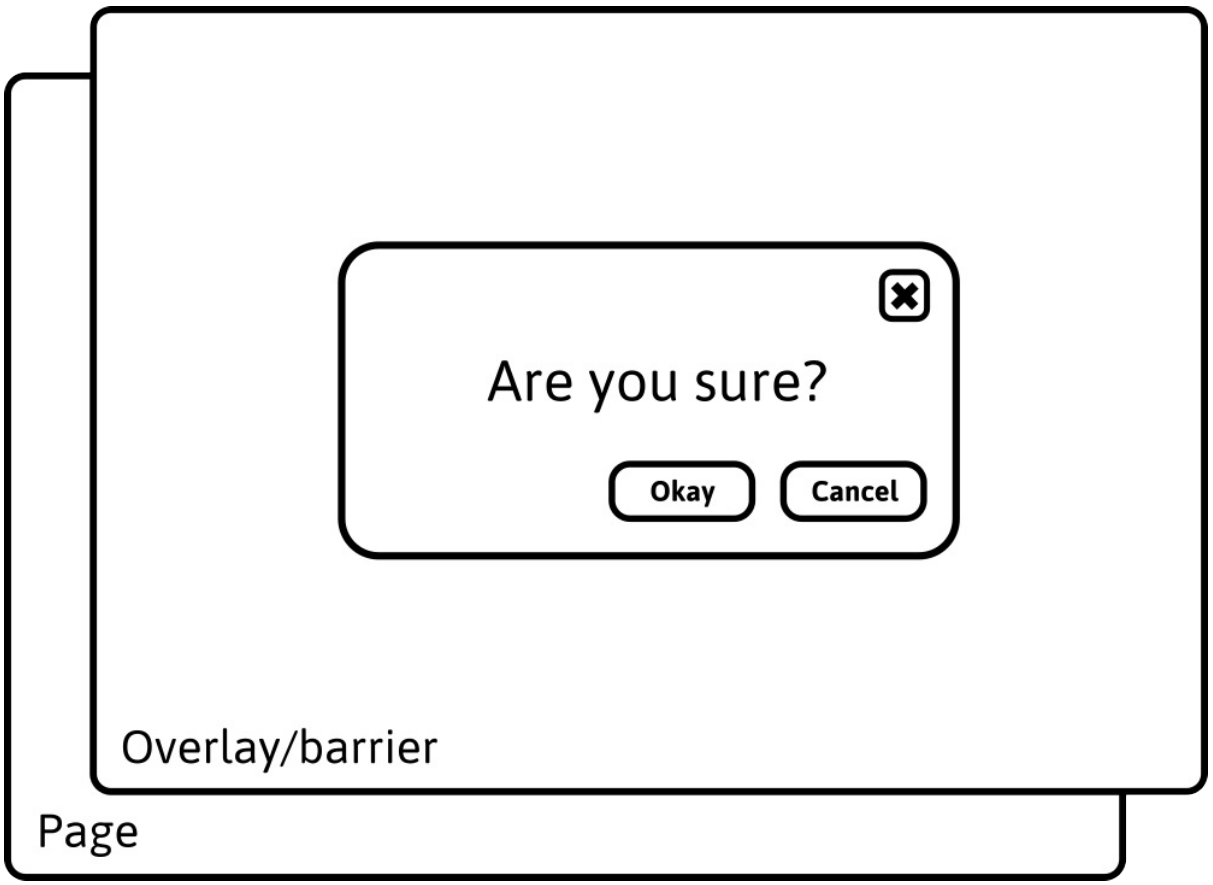
The first part is easy. By applying `aria-hidden="true"` to a container, all its contents become undetectable by assistive technologies. That is, it affects the whole 'subtree' of the DOM.

```
<div class="wrapper" aria-hidden="true">
  <!-- all the page contents -->
</div>
<div role="dialog" aria-labelledby="q-1234">
  <p id="q-1234">${question}</p>
  <div class="buttons">
    <button class="okay">Okay</button>
    <button class="cancel">Cancel</button>
  </div>
</div>
```

Of course, there needs to be an element that wraps literally everything apart from the dialog. That might be a problem. The focus issue is a lot harder, though. You need to be able to identify all the interactive elements and apply `tabindex="-1"`.

```
document.querySelectorAll('a, button, input, select,
textarea, [contenteditable]');
```

The trouble is, some of these elements might have `tabindex="-1"` already, and when it comes to reactivate the elements you'd end up making elements supposed to be inactive active. So you use `:not([tabindex="-1"])` with each of the selectors. It's a bit hacky, and there are still mouse users to worry about. You need to apply an overlay by positioning an element over the contents but under the dialog to act as a barrier for clicks.



Another way to go about it is to keep the overlay element, but 'trap' focus within the dialog so it's not possible to reach the other focusable elements by keyboard.

This too is a bit of a cludge. You need to listen for Tab and Shift + Tab key-presses and programmatically reroute focus between the buttons. It's also a substandard approach, because it makes reaching browser chrome like the address bar impossible by Tab. This is not the case when using a native `confirm()`.

The `inert` attribute

There is a better way. The inert attribute acts like `aria-hidden="true"` but takes care of disabling each element from both mouse and keyboard interaction. In effect, it does all of the messy stuff discussed just now but in one go. It is only supported in Chrome at the time of

writing, but there's a small polyfill available that does everything we need.

By targeting all direct children of `<body>` *before* creating and appending the dialog element, we can make everything outside the dialog `inert`. With this collection of nodes saved to memory, we can easily reactivate the content when the dialog closes.

```
const elems = document.querySelectorAll('body > *');
Array.prototype.forEach.call(elems, elem => {
  elem.setAttribute('inert', 'inert');
});
```

Focus

The demo implementation lets you provide a callback function as the second argument. This function fires after the `close()` function if **OK** has been pressed. As with `confirm()`, Esc also closes the dialog.

```
okay.onclick = () => {
  close();
  func();
}
cancel.onclick = () => close();
dialog.addEventListener('keydown', e => {
  if (e.keyCode == 27) {
    e.preventDefault();
    close();
  }
});
```

To trigger announcement of the dialog when it is opened, and to place the keyboard user within the dialog, the **OK** button is focused. This works because **OK** is inside the dialog, and the dialog is labeled (using `aria-labelledby`).

```
const trigger = document.activeElement;
okay.focus();
```

Importantly, the element that invoked the dialog in the first place (`const trigger`) is saved in memory so it can be refocused when the dialog is closed again. Incidentally, typing `document.activeElement` in your dev tools console is a quick way to find out which element is focused if it's not obvious (it should be!)

That's the default behavior; but it happens that, in the todo list demo with this custom modal dialog, the invoking element cannot be refocused when **OK** is pressed: the element has been removed from the DOM. As in the `confirm()` version, focus is redirected to the heading labeling the list: a reassuring affirmation of context followed by the ARIA live region announcement *"[item name] deleted"*.

```
remove(index, name) {
  this.dialog(`Are you sure you want to delete
"${name}"?`,
    () => {
      this.todos.splice(index, 1);
      document.getElementById('todos-label').focus();
      this.feedback = `${name} deleted`;
    }
  );
}
```

Demo: A todo list, with custom confirmation dialogs

The CSS

While the `confirm()` emerges from the top of the viewport, our custom dialog can appear wherever we like. Good old CSS Tricks offers a clever solution using `transform` to center the dialog regardless of its size.

```
[role="dialog"] {
  position: fixed;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

I finesse this in just one regard. I give the content element a `max-height` and `overflow-y: auto`. This way, the dialog is not allowed to grow larger than the viewport and become obscured. The text should be concise, so this should never be a problem anyway. But better safe than sorry.

```
[role="dialog"] .message {
  max-width: 50ch;
  max-height: 50vh;
  overflow-y: auto;
}
```

As you can see, it also has a `max-width` to prevent lines becoming too long and difficult to read. It's set in `ch` because 1 `ch` is roughly the width of one character, and measure (line width) is a question of characters.

Conclusion

One of the first painful lessons I learned from usability testing was that giving people lots of freedom to *explore* and *discover* interfaces is *not* actually very popular. People don't want to be misled or forced to do things they don't want to, but they do like you (and your interface) to be clear and assertive about what's needed to get the task in hand done.

You'd hope that the relationship between your interface and your user wouldn't have to resort to a dialog. The todo list example doesn't really need it because deleting a todo item you didn't mean to is hardly the end of the world. But when a dialog really is needed, there should be no ambiguity. Make it as straightforward and bold as possible. Let the user know that, for once, you really do need their input right there and then.

Checklist

- Don't use a dialog unless you are asking the user to resolve something crtical and urgent.
- So don't use non-modal dialogs.
- Keep dialogs brief; if the information is rich or the actions many, direct the user to a new page/screen.
- Use native methods like `confirm()` for maximum efficiency.
- Don't stray far from `confirm()` behavior if you intend on creating a custom modal dialog.

Cards

Some of the components I've explored here have specific standardized requirements in order to work as expected. Tab interfaces, for example, have a prescribed structure and a set of interaction behaviors as mandated by the WAI-ARIA specification.

It's at your discretion how closely you follow these requirements. Research may show that your audience doesn't do well with a tab interface *precisely* as recommended. Nonetheless, those requirements are there.

Other components, like the ubiquitous but multivarious card, do not have a standard to follow. There's no `<card>` element, nor an "ARIA card" design pattern. These are some of the more interesting components to work on. The card component is the last of the components in this book because it requires the most invention.

Each potential barrier to inclusion needs to be identified and addressed in turn. These barriers differ in line with the card's purpose and content. Some cards are just illustrated introductions to

permalinks like blog posts; others are more autonomous and offer a lot of functionality. In this chapter, I'll be looking into a few permutations of a simple card component, emphasizing a balance between sound HTML structure and ergonomic interaction.

Let's imagine a basic card. It has an illustration, a 'title', a description, and an attribution. Importantly, it belongs to a list item because — like playing cards — you rarely see one card on its own. Other component articles have already explored the advantages of using lists to enhance assistive technology users' experience. Briefly:

- Screen readers provide shortcuts to lists and between list items
- Screen readers enumerate the items so users know how many are available

In this case, let's say the card is a teaser for a blog post. Note the heading: like the list markup, headings provide navigation cues to

screen reader users. Each card has a heading of the same level — `<h2>` in this case — because they belong to a flat list hierarchy. Also note that the image is treated as decorative in this example, and is silenced in screen reader output using an empty `alt` value. I'll tackle positive `alt` text later in the chapter.

```
<li>
  <img src="/path/to/image.png" alt="">
  <h2>Card design woes</h2>
  <p>Ten common pitfalls to avoid when designing card
components.</p>
  <small>By Heydon Pickering</small>
</li>
```

The question is: where do I place the link to that blog post? Which part of the card is interactive? One reasonable answer is "the whole card". And by wrapping the card contents in an `a` tag, it's quite possible.

```
<li>
  <a href="/card-design-woes">
    <img src="/path/to/image.png" alt="">
    <h2>Card design woes</h2>
    <p>Ten common pitfalls to avoid when designing card
components.</p>
    <small>By Heydon Pickering</small>
  </a>
</li>
```

This is not without its problems. Now, all of the card contents form the label of the link. So when a screen reader encounters it, the announcement might be something link *"Card design woes, ten*

common pitfalls to avoid when designing card components, by Heydon Pickering, link”.

It’s not *disastrous* in terms of comprehension, but verbose — especially if the card evolves to contain more content. It’s also quite unexpected to find a block element like an `<h2>` inside an inline element like an `<a>`, even though it’s technically permissible in HTML5.

If I were to start adding interactivity, like linking the author name, things start to get even more confusing. Some screen readers only read out the first element of a ‘block link’, reducing verbosity but making it easy to miss the additional functionality. You just wouldn’t *expect* there to be another link inside the first link and a blind user might Tab away none the wiser.

```
<li>
  <a href="/card-design-woes">
    <img src="/path/to/image.png" alt="">
    <h2>Card design woes</h2>
    <p>Ten common pitfalls to avoid when designing card
components.</p>
    <small>By <a href="/author/heydon">Heydon
Pickering</a></small>
  </a>
</li>
```

Many an inclusive design conundrum stems from the tension between logical document structure, compelling visual layout, and intuitive interaction. Where we dispense with any one of these, someone somewhere will have a diminished experience. Compromise is inevitable, but it should be an equitable sort of compromise.

I find the best approach is to start with a sound document structure,

then use CSS to solve visual layout issues and JavaScript to enhance behavior — *if beneficial*. For this simple card, the title/heading is the name of the article for which the card acts as a teaser. It makes sense, then, to use its text as the primary link.

```
<li>
  <img src="/path/to/image.png" alt="">
  <h2>
    <a href="/card-design-woes">Card design woes</a>
  </h2>
  <p>Ten common pitfalls to avoid when designing card
components.</p>
  <small>By Heydon Pickering</small>
</li>
```

The advantage here over having “read more” calls-to-action is that each link has a unique and descriptive label, which is useful when users are searching through aggregated lists of links. For example, pressing Insert + F7 in NVDA gives the user access to all links on the page.

However, I’d still like the card itself to be clickable. In the absence of a clear “read more” call to action in this case, it’s not obvious where to click, so “anywhere” solves the problem. It also makes the link easier to target by touch and mouse.

There are a couple of ways to solve this.

The pseudo-content trick

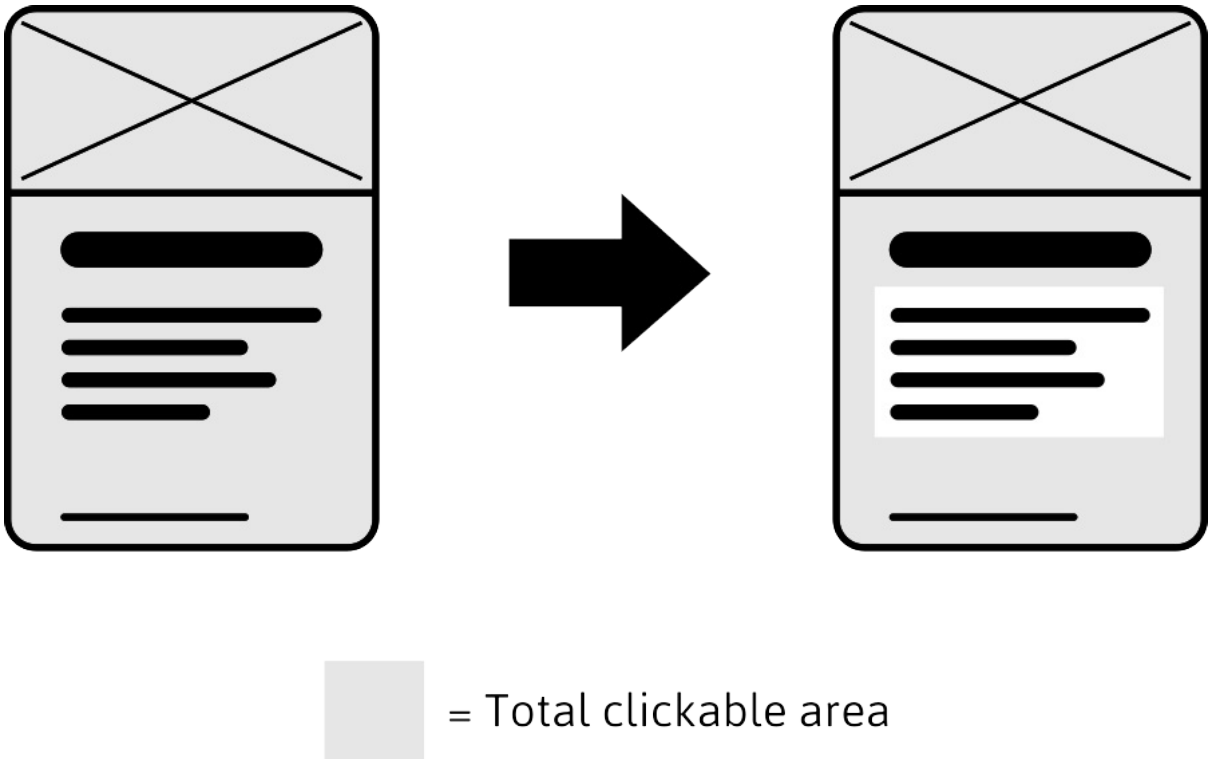
By taking the following steps, I can make the whole card clickable

without altering the markup we've established.

1. Give the card container element `position: relative`
2. Give the link's `::after` pseudo-content `position: absolute`
3. Give each of the link's `::after` pseudo-content `left`, `top`, `right`, and `bottom` properties a value of `0`

This stretches the link's layout over the whole card, making it clickable like a button.

On the one hand this is a sound solution because it doesn't rely on JavaScript (and why use JavaScript on a static site like a blog if you can avoid it?) On the other hand, it's not ideal because it's now difficult to select the text within the card (the link acts as a mask over the top of it). You can selectively 'raise' elements like the description by giving them `position: relative`, but their layout then becomes a gap in the card's overall clickable area.



Demo: The pseudo-content trick

The redundant click event

Alternatively, we can employ JavaScript and use the card container as a proxy for the link. A click handler on the card's container element simply triggers the click method on the link inside it. This does not affect the keyboard user, who remains content with the original link.

```
card.addEventListener(() => link.click());
```

Technically, because of event bubbling, if I click the link directly (making it the event's `target`) the event fires twice. Although no side effects were found in testing, you can suppress this like so:

```
card.addEventListener(e => {
  if (link !== e.target) {
    link.click();
  }
});
```

Now selecting the text is possible, but the `click` event is still fired and the link followed. We need to detect how long the user is taking between `mousedown` and `mouseup` and suppress the event if it's "likely to be selecting text" territory.

Here's the whole script used in the demo to follow. I found that a 200 millisecond threshold was about right. In any case, an unusually ponderous click is recoverable with a second attempt.

```
const cards = document.querySelectorAll('.card');
Array.prototype.forEach.call(cards, card => {
  let down, up, link = card.querySelector('h2 a');
  card.onmousedown = () => down = +new Date();
  card.onmouseup = () => {
    up = +new Date();
    if ((up - down) < 200) {
      link.click();
    }
  }
});
```

Demo: The redundant click solution

It's not highly probable the user would choose to select text from a card/teaser when they have access to the full content to which the card/teaser is pointing. But it may be disconcerting to them to find they *cannot* select the text. If it's *not* seemingly important, I recommend you use the pseudo-content trick, because this approach means the link's context menu appears wherever the user right clicks on the card: a nice feature.

Affordance

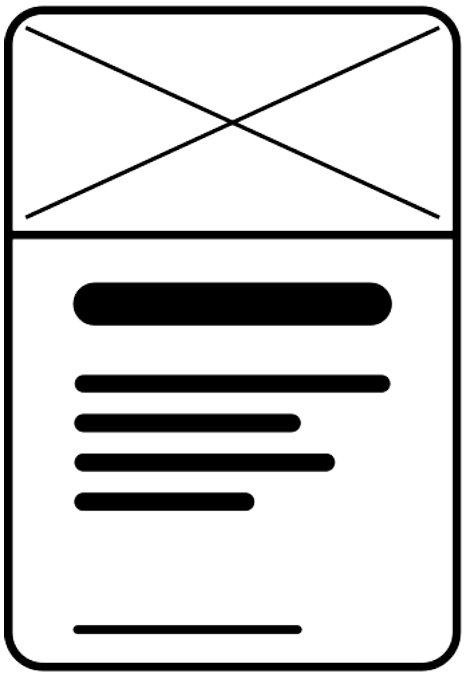
If the whole card is interactive, the user should know about it. We need to support perceived affordance.

Using the **pseudo-content trick**, the entire card already takes the `pointer` cursor style, because the card has the link stretched over it. This will have to be added manually for the **redundant click event** solution. We should add it *with* JavaScript because, if JavaScript fails, the style would be deceptive.

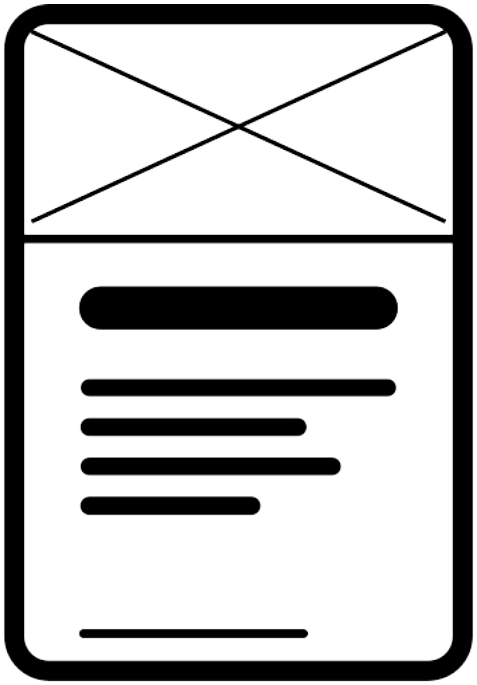
```
card.style.cursor = 'pointer';
```

In addition to the rounded, button-like design I've given my cards, a hover style makes things clearer still. I like to use `box-shadow` because — unlike `outline` — it respects the curves of the corners.

```
.card:hover {
  box-shadow: 0 0 0 0.25rem;
}
```



Default state



Hover/focus state

Where there are hover styles there should also be focus styles, which presents us with an interesting problem. Using `:focus`, we can only apply a style to the link itself. This isn't a big issue, but it would be nice if sighted keyboard users saw that nice big, card-sized style that mouse users see. Fortunately, this is possible using `:focus-within`:

```
.card a:focus {
  text-decoration: underline;
}

.card:focus-within {
  box-shadow: 0 0 0 0.25rem;
  outline: 2px solid transparent;
}

.card:focus-within a:focus {
  text-decoration: none;
}
```

I've progressively enhanced focus styles here using CSS's cascade. First I apply a basic focus style, to the link. Then I use `:focus-within` to match the `:hover` style. Finally, I *remove* the unnecessary basic `:focus` style *only* where `:focus-within` is supported. That is, if `:focus-within` is *not* supported, both the second and last blocks will be ignored. The upshot is that users of older browsers that do not support `:focus-within` will still see the fallback focus style.

Note I've also included a transparent `outline` style, because Windows High Contrast Mode suppresses `box-shadow`. The transparent color will change to become visible when high contrast mode is switched on.

Without :focus-within support

With :focus-within support

Warning: do not include the `:hover` style in the same blocks as `:focus-within`. The whole block will be rejected where `:focus-within` is not supported, and you'll lose the hover style along with it.

Content tolerance

One unsung aspect of inclusive design is the art of making interfaces tolerant of different levels of content. Wherever an interface 'breaks' when too much or too little content is provided, we are restricting what contributors can say.

Our cards need to be able to accept different length of title and description without the design become ugly or difficult to scan. First, I add some `<div>` containers for convenience (`.img` and `.text`):

```
<li class="card">
  <div class="img">
    <img src="/path/to/image.png" alt="">
  </div>
  <div class="text">
    <h2>
      <a href="/card-design-woes">Card design woes</a>
    </h2>
    <p>Ten common pitfalls to avoid when designing card
components.</p>
    <small>By Heydon Pickering</small>
  </div>
</li>
```

Then I make both the `.card` container and the `.text` wrapper inside it Flexbox contexts, using `flex-direction: column`.

```
.card, .card .text {
  display: flex;
  flex-direction: column;
}
```

Next, I force the textual elements to take up all the available space with `flex-grow: 1`:

```
.card .text {
  flex-grow: 1;
}
```

Finally, to give some balance, I take the last textual element and give it a top margin of `auto`:

```
.card .text :last-child {
  margin-top: auto;
}
```

This pushes the attribution element in the demo to the bottom of the card, regardless of its height.

All that's left to do is add a bottom margin to the second last element, to ensure a minimum level of separation. Using the owl selector to inject a common margin, it all comes together like this:

```
.card, .card .text {
  display: flex;
  flex-direction: column;
}

.card .text {
  flex-grow: 1;
}

.card .text > * + * {
  margin-top: 0.75rem;
}

.card .text :last-child {
  margin-top: auto;
}

.card .text :nth-last-child(2) {
  margin-bottom: 0.75rem;
}
```

Note that we are managing margins in an algorithmic way here, using position and context rather than specific element properties. No matter what the elements we place in this text container, the effect will be the same.

Progressive grid enhancement

All that's left to do is place the cards in a CSS Flexbox or CSS grid context wherein the cards will stretch to share the same height: the height of the card with the most content.

Grid and Flexbox can both have this effect, but I prefer Grid's wrapping

algorithm and `grid-gap` is the easiest way to distribute cards without having to use negative margin hacks.

By using `@supports` I can implement a simple, one-column layout then enhance it with grid where supported:

```
.card .text {
  max-width: 60ch;
}

.card + .card {
  margin-top: 1.5rem;
}

@supports (display: grid) {
  .cards > ul {
    display: grid;
    grid-template-columns: repeat(auto-fill,
minmax(15rem, 1fr));
    grid-gap: 1.5rem;
  }

  .card + .card {
    margin-top: 0;
  }
}
```

Note the `max-width` of (approximately) 60 characters on the text container. This prevents the line length for cards on large screens not supporting grid from becoming too long. They won't really look like cards in these conditions, of course, but at least they'll be readable.

More card

Eu dolore labore ad occaecat minim in minim ad ea commodo excepteur ullamco. Cupidatat tempor sint mollit in tempor ut fugiat excepteur laborum labore.

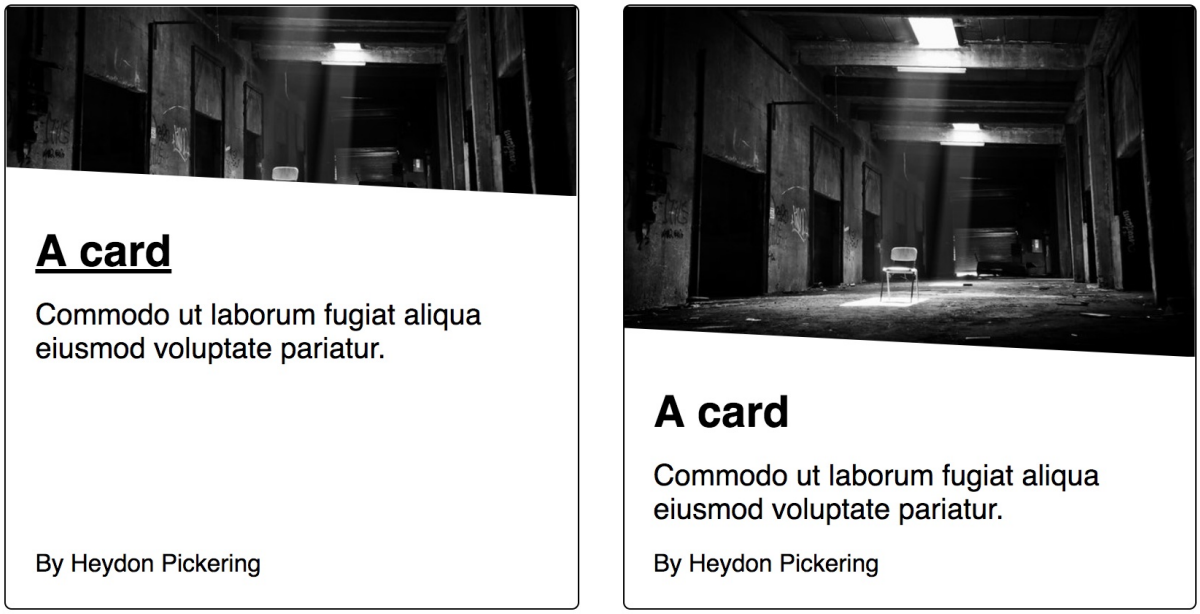
By Heydon Pickering

Fortunately, support for Grid is fairly extensive now.

The image dimensions

In addition to allowing flexible text content, we should handle different uploaded image dimensions. The `object-fit: cover` declaration makes light work of this when combined with a `width` and `height` of `100%`. This allows us to adjust the height of the image container to our liking without gaps showing or the image becoming squished.

```
.card .img {
  height: 5rem;
}

.card .img img {
  object-fit: cover;
  height: 100%;
  width: 100%;
}
```



The slight slant given to the image box is achieved using `clip-path`. This is a progressive enhancement too. No content is obscured where `clip-path` is not supported.

There's an inherent compromise in using `object-fit:cover`: To maintain the correct aspect ratio, the image will become cropped along two or more edges. Since the image is always centered within its container, it helps to curate images for which the center is the focus. The positioning can be adjusted using `object-position`, however.

The author link

What about that author link? The first thing to consider is whether it's necessary or desirable to make this link interactive within the card. Especially if the author's page is linked from the permalink to which the card is pointing. Only add tab stops where beneficial, because too many make navigation by keyboard slow and arduous.

For argument's sake, let's say there is a use case for linking the author within the card. This is viable alongside both the pseudo-content and

JavaScript techniques described above. A declaration of `position:relative` will raise the link above the pseudo-content in the first example. Contrary to popular belief, just the positioning is needed, and no `z-index`, because the author link is after the primary link in the source.

```
.card small a {
  position: relative;
}
```

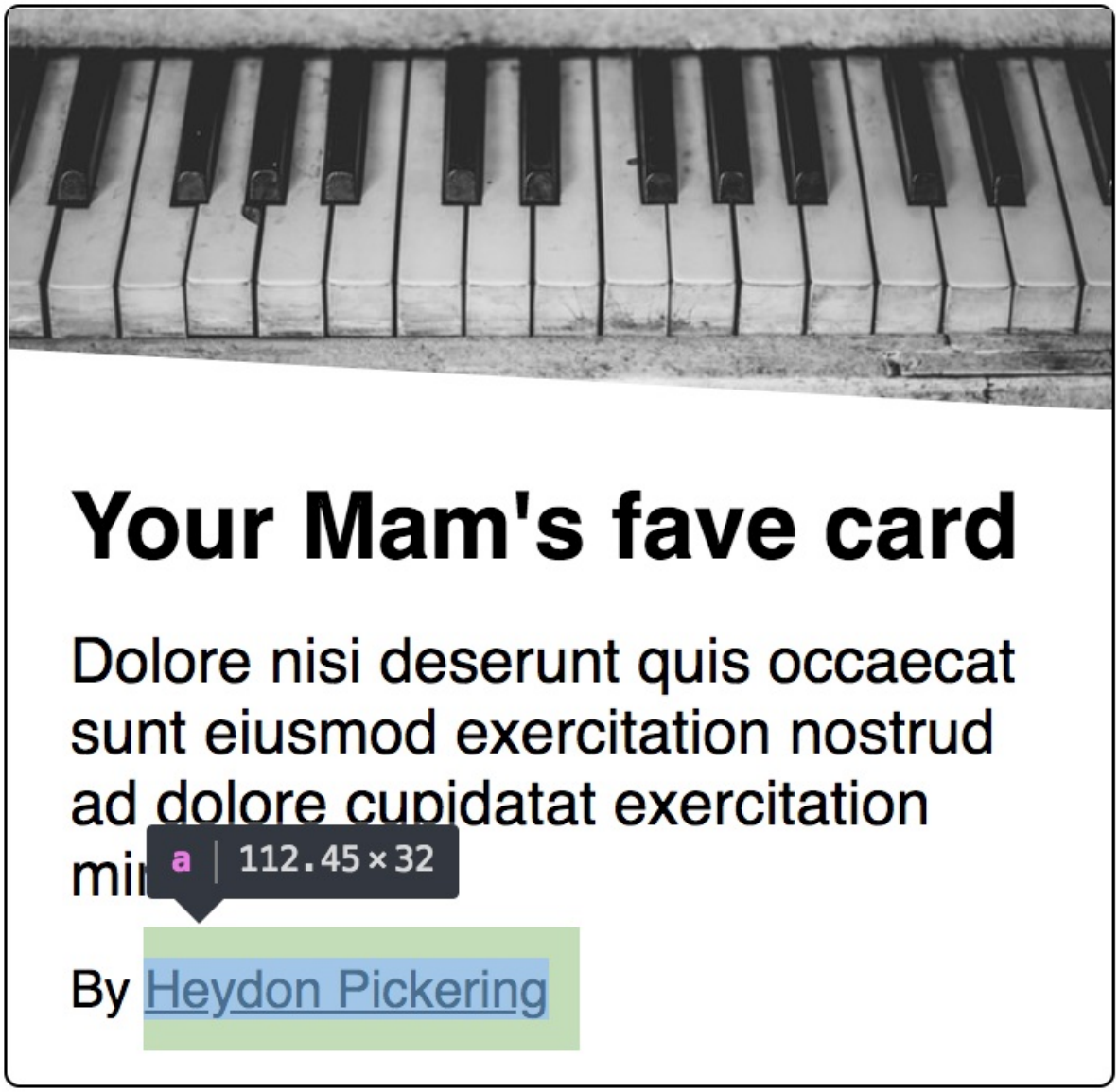
Demo: Author links demo (using pseudo-content)

(**Note:** the links are just dummy links pointing to hash fragments. Check the address bar to see which one you've pressed.)

An issue for some users will be trouble targeting the author link. In most scenarios, a couple of inaccurate clicks or taps are relatively harmless, but where the desired link is placed *over* another interactive element, the other element might get activated. In this case, that would mean loading the wrong page.

So, why don't we increase the 'hit area' of the author link to mitigate this? We can use `padding`. The left padding remains unaffected because this would push the link away from the preceding text.

```
.card small a {
  position: relative;
  padding: 0.5rem 0.5rem 0.5rem 0;
}
```



Many people find they have low accuracy when targeting items by touch, including those with Parkinson's disease and rheumatism. So it's also wise to increase the gap between single-column cards on small

screens, meaning it's easier to avoid activating a card while scrolling.

```
@media (max-width: 400px) {
  .cards > ul {
    grid-gap: 4.5rem;
  }
}
```

Calls to action

As I said already, multiple "read more" links are useless when taken out of context and aggregated into a glossary. Best to avoid that. However, it may prove instructive to have an explicit call-to-action. Without it, users may not be aware cards are interactive. Being *obvious* is usually the best approach in interface design.

So, how do we supply these buttons but keep the descriptive link text? One possibility is to keep the title/heading as the primary link, and add a decorative 'read more' button separately.

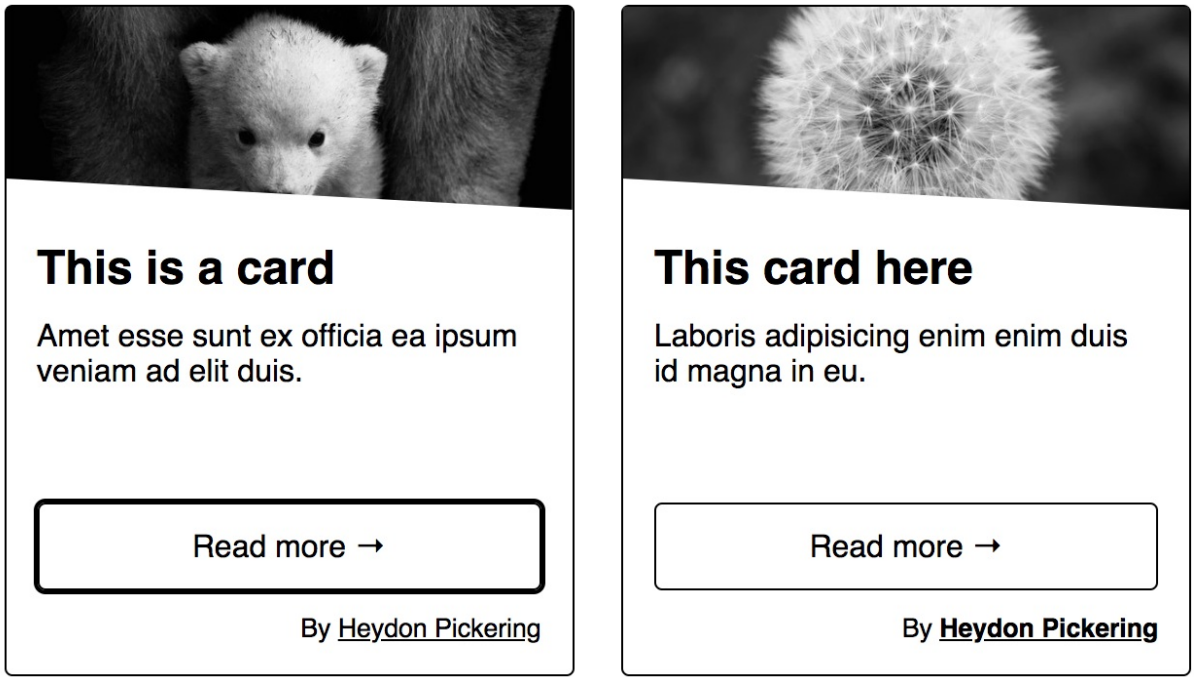
```
<li>
  <img src="/path/to/image.png" alt="">
  <h2>
    <a href="/card-design-woes">Card design woes</a>
  </h2>
  <p>Ten common pitfalls to avoid when designing card
components.</p>
  <span class="cta" aria-hidden="true">read more
</span>
  <small>By Heydon Pickering</small>
</li>
```

Had I made the call-to-action a link too, I'd be creating redundant functionality and an extra tab stop. Instead, the 'button' is just for show, and hidden from assistive technologies using `aria-hidden`. The trick is to make it appear the button is the interactive element: another job for `:focus-within`.

I look for focus within the `<h2>`, and use the general sibling combinator to delegate the focus style to the call-to-action button.

```
.card h2 a:focus {
  text-decoration: underline;
}

.card h2:focus-within ~ .cta {
  box-shadow: 0 0 0 0.125rem;
  outline: 2px solid transparent;
}

.card:focus-within h2 a:focus {
  text-decoration: none;
}
```

Visual focus order remains logical between the title (call-to-action) and author link:



For sighted screen reader users there's potential for a little confusion here, since "read more" will not be announced despite the element appearing to take focus. Fortunately, we can attach "read more" to the link as a description, using `aria-describedby`. Now users will hear *"Card design woes, link, read more"*. The description is always read last.

```
<li>
  <img src="/path/to/image.png" alt="">
  <h2>
    <a href="/card-design-woes" aria-describedby="desc-
card-design-woes">Card design woes</a>
  </h2>
  <p>Ten common pitfalls to avoid when designing card
components.</p>
  <span class="cta" aria-hidden="true" id="desc-card-
design-woes">read more</span>
  <small>By Heydon Pickering</small>
</li>
```

It works because, even where `aria-hidden="true"` is applied, the *relationship* created is still intact and the description available to the link. This is useful where you want to use an element for your description, but don't want assistive technologies to acknowledge the element directly. It would have been confusing to a screen reader user to be be able to browse to the call-to-action and hear the imperative "read more" without the element having a role or being interactive.

Note that, since the call-to-action says "read more" in each case, only one of the calls-to-action elements *needs* to be referenced by each of the cards' links. Within a templating loop, this is likely to be hard to implement, though.

Demo: Calls-to-action demo

Note

Unique strings

When creating dynamic content by iterating over data, there are certain things we can't do. One of these is to manually create `id` values.

For relationships built using `id`s to work (like the `aria-describedby` association in the previous example) those `id` ciphers need to be consistent and unique. There are a couple of ways you can do this.

The first is to create a unique string using some pseudo-randomization. The following snippet, based on a gist by Gordon Brander, is a neat solution and has been used already in this book:

```
const uniq = Math.random().toString(36).substr(2, 9);
```

Another solution is to stringify `Date` using the + operator, which is terser. Note that, unlike the above solution, this does result in a number — so it must prefixed with some alphabetic characters to make it a valid `id`.

```
const uniq = +new Date();
```

Using Vue.js, I can apply this unique string like so:

```
<!-- the id attribute -->
:id="'desc-' + uniq"

<!-- the aria-describedby attribute -->
:aria-describedby="'desc-' + uniq"
```

One disadvantage of this approach is that testing snapshots will constantly be out of sync, because a new unique string is generated for each build. It's for this reason that I prefer to use a simple 'slugify' function, based on a string already in the page. In this case, the title for the card seems apt.

Here's that function as a small utility module. It would convert *"My card component!"* to `my-card-component`:

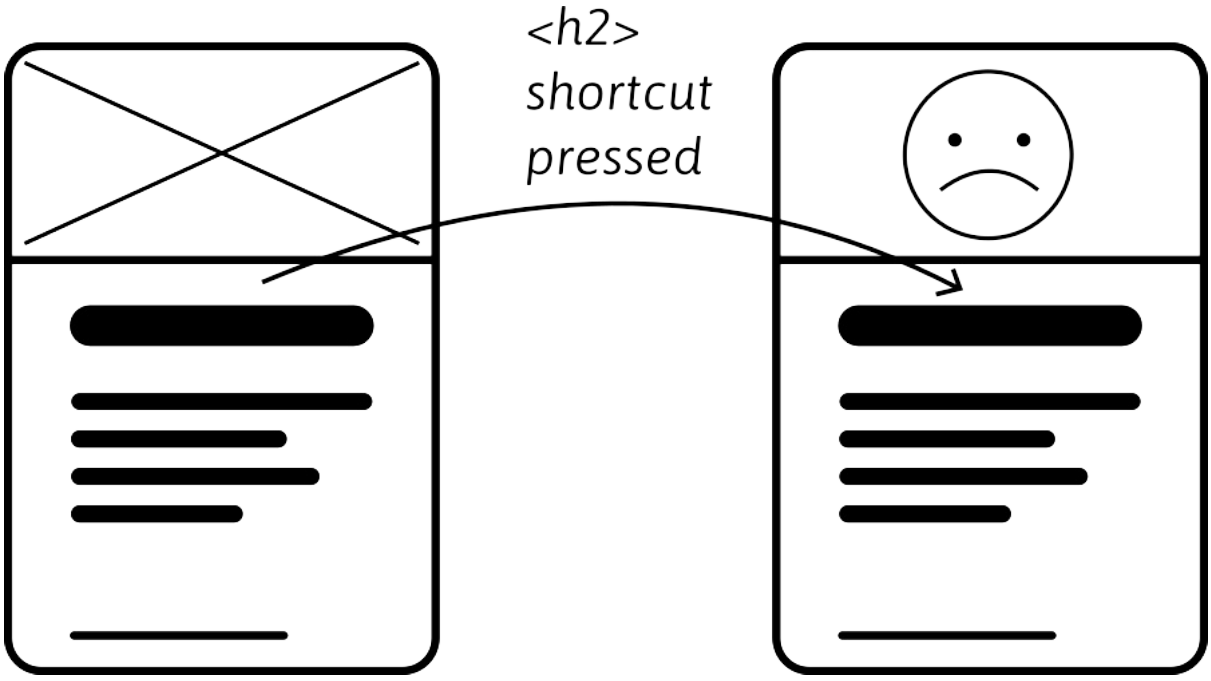
```
export default text => {
  return text
    .toString()
    .toLowerCase()
    .replace(/\s+/g, '-')
    .replace(/[^\w\-]+/g, '')
    .replace(/\-\-+/g, '-')
    .replace(/^-+/, '')
    .replace(/-+$/, '');
};
```

Alternative text

So far, I've been working with the assumption that the card's image is decorative and doesn't need alternative text, hence `alt=""`. If this empty `alt` was omitted, screen readers would identify the image and read (part of) the `src` attribute as a fallback, which is not what anyone wants here.

If the image *were* considered pertinent in terms of content (for example, the appearance of a product that's for sale) we should of course supply a suitable value to `alt`. But we have a problem, and one anticipated by Andy Kirk who contacted me about it on Patreon.

Currently, the image appears *before* the text. Since headings, like the `<h2>` here, introduce sections, having the image before the heading suggests that it does not belong to the section. On one hand, you could argue that the `<li>` groups the image with the text, but not all users would consume the structure this way. When a screen reader user operating NVDA presses `2` to go to the next `<h2>`, they would 'skip over' the image and miss it.



The `order` property

Fortunately, Flexbox's order property allows us to manipulate the source order, then correct for visual appearance.

First I switch the image and text containers around...

```
<li class="card">
  <div class="text">
    <h2>
        <a href="/card-design-woes">A great product</a>
    </h2>
    <p>Description of the great product</p>
    <small>By Great Products(TM)</small>
  </div>
  <div class="img">
    <img src="/path/to/image.png" alt="Description of
the great product's appearance">
  </div>
</li>
```

…then I just promote the image container to the top of the layout:

```
.card .text {
  order: 1;
}
```

Manipulating the order of elements using CSS can cause accessibility issues, especially where it means the focus order contradicts the visual layout. This can be confusing.

In this case, the focus order is not applicable because the image is not focusable. The experience for sighted screen reader users is a little odd but unlikely to cause major comprehension problems since each card is visually self-contained.

Screen readers like VoiceOver provide a visible ring, like a focus style, for each element the user browses — including non-focusable elements. This ring shows the user which element they are on, wherever they are.

Demo: Calls-to-action demo

Conclusion

Some of the ideas and techniques explored here may not be applicable to your particular card designs; others will. I'm not here to tell you exactly how to design a 'card' because I don't know your requirements. But I hope I've given you some ideas about how to solve problems you

might encounter, and how to enhance the interface in ways that are sensitive to a broad range of users.

In fact, none of the components in this book are offered as 'perfect', just-copy-and-paste exemplars. My real aim has been to show you how to **think inclusively** as you approach interface design. Please do take the ideas and the code here, because I've done my best to solve many of the common problems. But look out for other problems with these components, and with the new and different components you make in the future.

Checklist

- Use list markup to group your cards
- Make sure your cards don't break when lines of content wrap or images don't meet specific aspect ratio requirements
- Avoid too much functionality and reduce tab stops. Cards shouldn't be miniature web pages.
- Remember that headings should *begin* sections. Most everything that belongs to the section should follow the heading in the source.

That’s it. Go make a better web.