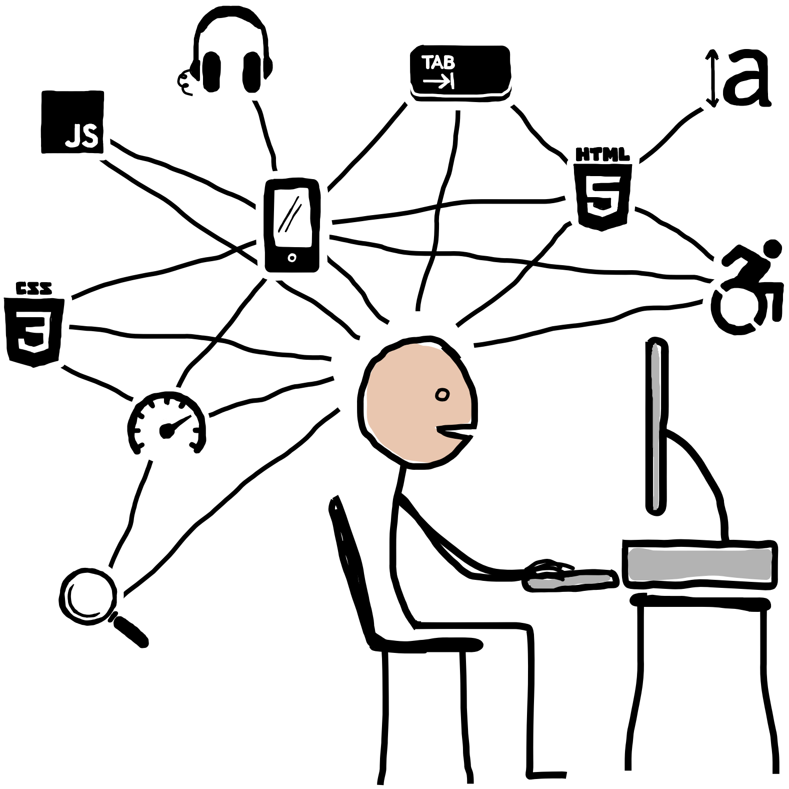
INCLUSIVE DESIGN PATTERNS

Coding Accessibility Into Web Design



SMASHING MAGAZINE

by Heydon Pickering

For my wife, Fan, who has the power
to make me smile, in spite of the world.

Published 2016 by Smashing Magazine GmbH,
Freiburg, Germany.
ISBN: 978-3-945749-43-2.

Cover and illustrations created by Heydon Pickering.
Editing & proofreading: Owen Gregory.
Print layout: Markus Seyfferth.
eBook production: Cosima Mielke.
Typefaces used: Elena by Nicole Dotin and Mija by Miguel Hernández.

Inclusive Design Patterns was written by Heydon Pickering and reviewed by Rodney Rehm and Steve Faulkner.

Table Of Contents

About The Author

Heydon Pickering is a utilitarian designer, writer and public speaker, the accessibility editor for Smashing Magazine,[1] and a consultant working with the Paciello Group.[2] He's interested in new and innovative ways to make the web an inclusive place. User research, systems thinking, and plain old semantic HTML all play their part. When Heydon isn't writing, coding or illustrating, he does some mental health campaigning, experiments with sound design, and thrashes out doom metal riffs on his detuned SG copy. He's almost entirely fueled by original recipe Guinness and Naga chillies.

About The Reviewers

Rodney Rehm is a web developer based in southern Germany. After being a full-stack freelancer for a decade, he moved on to working on the front-ends of Qivicon, Deutsche Telekom's Smart Home platform. He created *URI.js*[3] and the *ally.js*[4] accessibility library, made libsass run in the browser,[5] and wrote the world's first buggyfill.[6]

1 https://www.smashingmagazine.com/tag/accessibility/
2 https://www.paciellogroup.com/
3 http://medialize.github.io/URI.js/
4 http://allyjs.io/
5 https://github.com/medialize/sass.js/
6 https://github.com/rodneyrehm/viewport-units-buggyfill

Steve Faulkner is the senior web accessibility consultant and technical director of TPG Europe. He joined the Paciello Group in 2006 and was previously a senior web accessibility consultant at Vision Australia.[7] He is the creator and lead developer of the Web Accessibility Toolbar[8] accessibility testing tool. Steve is a member of several groups, including the W3C Web Platforms Working Group and the W3C ARIA Working Group. He is an editor of several specifications at the W3C[9] including:

- HTML 5.1,[10]
- ARIA in HTML,[11]
- Notes on Using ARIA in HTML,[12]
- HTML5: Techniques for useful text alternatives.[13]

He also develops and maintains *HTML5accessibility.com*.

7 https://www.visionaustralia.org/
8 https://www.paciellogroup.com/resources/wat/
9 http://w3.org
10 http://w3c.github.io/html/
11 http://w3c.github.io/html-aria/
12 http://w3c.github.io/aria-in-html/
13 http://www.w3.org/TR/2014/WD-html-alt-techniques-20141023/

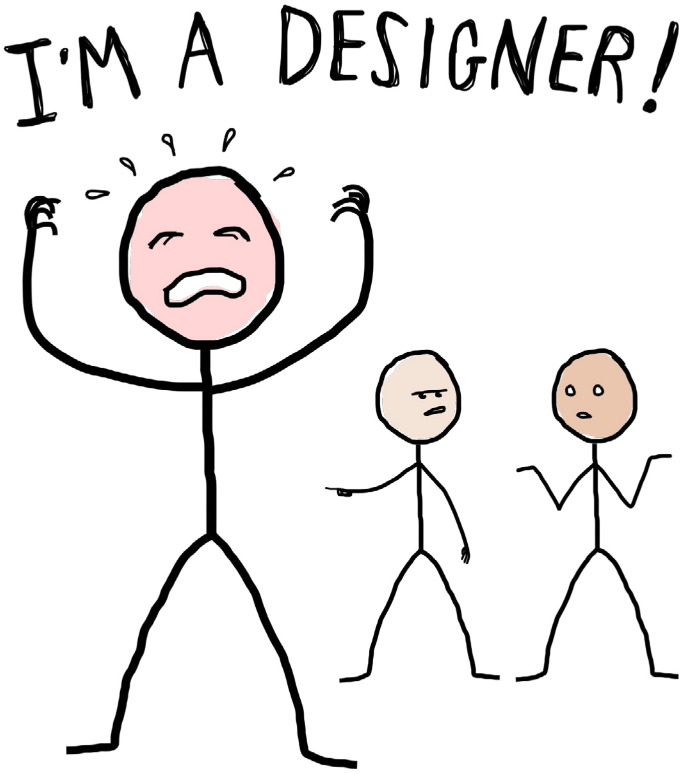
Foreword

I've always had a good word for Heydon Pickering. However, laws of libel prevent me from writing it here, so instead let me tell you about this book: it's very good. Heydon clearly and comprehensively shows you how to use the web properly; that is, how to take advantage of its built-in powers of reach, inclusivity and accessibility, so you don't lock out any potential customer or visitor, regardless of how they choose to (or need to) access your content.

I've been writing about the same for many years, but I've still picked up a number of useful tips and tricks when reading through the drafts. It's a little unfortunate that Heydon didn't accept some of my amendments to make his musical taste less — what's the word? — embarrassing, but don't let that put you off reading. There's a load of useful advice contained in this book, and it would be really good if you were to follow it and make your corner of the web a better place for all. Just tweet *me* if you want music suggestions.

Bruce Lawson,[14] *Deputy CTO, Opera*

14 http://www.brucelawson.co.uk



"What's their problem?"
"I dunno. All I asked them is where the toilets are."

Introduction

Imagine we meet at a party. Not long into conversation you ask me what I do. I simply say, “I’m a designer.” Before you have the chance to ask me what *type* of designer I am, our mutual friend, the host, taps me on the elbow.

“Can I borrow you a moment?”

They make their apologies and usher me away. You’re rather relieved, actually, since the sort of person who proudly pronounces themselves “a designer” is probably a bit of a self-regarding bore. Nonetheless, you’re left briefly wondering what I actually do to pay the bills. Perhaps he makes designer egg cups? Sews postmodernist swimwear? Builds helicopter missile systems? He could have meant *anything*.

By the end of the party we’re not reacquainted, but in a conversation with our mutual friend over the phone the following day, they apologize to you again for the previous night’s interruption. After a moment, you recall.

“Oh, your friend the designer. Well, never mind. What does he actually ***design*** *anyway?”*

“Oh, he's a web designer.”

"Oh, ***right****."*

"Yeah."

Now, the way you might react to this new information is really going to depend on your relationship to the sprawling, amorphous public information matrix we call "The Web." If you're more a consumer of the web than a producer *for* the web, *web designer* is probably sufficient: "OK, so they design web stuff. Websites and the like."

But I'm willing to bet this isn't you, otherwise you wouldn't be reading this book. In which case, you're no doubt aware just how controversial, contested, and frequently contrary the term *design* is in the context of the web. The number of people contributing to the web at any given moment is unfathomable, and so is the variety of their talents and skills. So which ones are the designers, and what do they do?

For a long time, we've been getting the answers to these questions epically, tragicomically *wrong*. We've misconceived and mistreated the medium, both making hard work for ourselves and shortchanging our audiences as a result.

Reconceiving Web Design

Our most persistent error as an industry has been to apply the largely incompatible principles of *printed* graphical communication design to the web. A print designer's domain is purely visual, confined to predictable and predetermined spaces, using agreed materials and consistently reproducible resolutions and colors. Print design is the production of static, immutable artifacts.

That's not like the web *at all*, but it doesn't stop us pouring wasted energy into pointless frippery like pixel perfection and organizing ourselves into those two ill-defined, arbitrary groups, designers and developers. It's as if content editors and project managers don't even exist!

The truth is, design work is *deliberation*. It's the pursuit of the best solution to a given problem. By relegating design to the realm of visual aesthetics, so much of the web goes undesigned. This can only lead to inaccessibility, poor performance and, of course, a general lack of utility.

The aim of this book is to help you nurture design thinking that's suited to the web and, as such, acknowledges some (astonishingly recent!) developments in the evolution of the web design discipline:

- The web is made of code and must be designed, therefore designing *with* code is working with the right materials. This is the best course of action.

- Content — what we *write* or otherwise *express* via the web — must be subject to design thinking and, in fact, all other design decisions should facilitate that.

- Web pages are not immutable artifacts. They should be tolerant of changing, dynamic content. This content should be managed in terms of discrete components which can be reused as agreed patterns.

- The potential audience of a website or app is *anyone human*. Inclusivity of ability, preference and circumstance is paramount. Where people differ — and they always do — inclusive interfaces are robust interfaces.

Inclusive Design

For the subject and purpose of this particular book, that last principle is the most important. Before we can contemplate inclusive patterns, inclusive design must first be defined. It's really more of a mindset than anything reducible to discrete skills, so I'm going to illustrate it by way of an analogy.

Here's a property representing a street address in JSON:

```
"address": "84, Beacon St, Boston, MA 02108,
 United States"
```

Now, if I wanted to print the address to screen using a templating library like Handlebars, I might write something like `{{this.address}}`. Fine, but what if I needed to refer to the country part of my address on its own? With the data in its current form — one string — I'd have to write a helper to extract the "United States" part.

```
Handlebars.registerHelper('getCountry', function(address) {
  return address.split(',').pop();
});
```

This is what some might refer to as *hacky*: a fragile and unfortunately complex workaround. A hack is a symptom of *bad design*. Not only is the employment of a helper function relatively computationally heavy, it's also unreliable. Why? Because not all addresses end in a country. British addresses, for instance, usually end in a postcode:

```
"address": "85-87, Gwydir St, Cambridge, England, CB1 2LG"
```

A more robust solution might be to make the address property an object and store each part of the address as a property on that object:

```
"address": {
   "building": "85-87",
   "street": "Gwydir St",
   "city": "Cambridge",
   "country": "UK",
   "zipOrPostcode": "CB1 2LG"
}
```

Now I can access the country simply with `address.country`.

This seems like a better approach. But is it? Using a hugely international set of addresses, you'll find they vary so much that prescribing an unbending set of properties is just not tenable. Simply capturing a single string makes more sense. You'd have to forfeit being able to extract countries, but sometimes that's the way it goes.

In any case, thinking about the structure of data and trying to arrive at an optimal solution *is designing* and there's not an Adobe license or copy of Sketch in sight. In fact, it's really a kind of *inclusive* design: here, the right solution is one which is inclusive of different kinds of addresses. We'd like to be able to deal with consistent, uniform addresses, but that's not reality. Instead, we make allowances.

Replace "structure of data" with "interface" and swap "addresses" for "people" in the previous paragraph, and you've got **inclusive interface design** in a nutshell.

The best part is that designing inclusive interfaces, like designing robust data schemas, doesn't have to be any harder or more complex than making exclusive or otherwise obsolete ones. It's just different. By looking at common web interface patterns through the lens of inclusivity, this book will help you quickly learn how to apply and reapply conventions that will earn you a broader and less frustrated audience.

The Inclusive Button

Let's look at one more simple example, a prototypical interface pattern much like those to follow. This time, we'll look at an interactive element, a button, from the perspectives of three types of designer. The purpose of this example is to show you how a little bit of knowledge about the medium can lead to a simpler and more inclusive solution.

THE GRAPHIC DESIGNER

The first designer comes from a graphic design background. Their files never have a resolution lower than 300dpi and they have a working knowledge of color theory. Their typography and illustration skills are vivacious. To them, a button is a visual artifact, producible in Adobe Illustrator or Sketch. They are concerned with how that button resembles a real button and how it simultaneously fits into the larger design's branding. They have no idea how to put that button on a web page or to make it do anything.

THE DESIGNER WHO CODES

The second designer has many of the same skills exhibited by the first, but differs in one important aspect: they have enough knowledge of HTML, CSS and JavaScript to make their button appear inside a web page, and attach a JavaScript event listener to it.

The HTML looks like this:

```
<div class="button"></div>
```

and the CSS like this:

```
.button {
  width: 200px;
  height: 70px;
  background: url('../images/button.png');
}
```

The JavaScript would probably be written using jQuery or perhaps AngularJS's API. Using the Web API ("vanilla" JavaScript), it might look a bit like this:

```
button.addEventListener('click', function() {
   // the event fired on click
});
```

Designer number two is a designer who has learned to code inasmuch as they've acquired the ability to bring their ideas to the web — to put them in web pages. Under some circumstances, they may even work for the odd user. Unfortunately, aside from the event listener, all they're really doing is encoding graphic design, not *designing with code*. As the third designer is only too aware, there are few direct translations between graphic design and web interface design.

THE INCLUSIVE DESIGNER

The third designer sees the second designer's button from a number of perspectives, each the imagined point of view of a potential user. Accordingly, the current implementation presents a number of problems.

One problem regards the not insignificant number of users who zoom their web pages to ease reading. The image used is not a vector, so cannot be scaled without becoming degraded and blurry as a result. That's if the user operates full page zoom. If the user adjusts their browser's default font size independently, the image (which has been defined using pixels rather than relative units) will not scale at all.

Another problem is encountered when a user switches off image loading in their mobile browser, to save bandwidth: the button, constituted entirely by a background image, will be invisible. Users having trouble differentiating foreground and background imagery may turn on Windows' high contrast mode.[1] This, under some conditions, will also eliminate background images.[2]

And it doesn't stop there. The `<div>` element, unlike the purpose-built `<button>` element, is not (in its current form) focusable or operable by keyboard. Some folks choose to navigate and operate web pages with a keyboard. Others *must* use a keyboard, because a mouse requires a finer motor control than they can muster.

Screen reader users include those with severe vision impairments, and others who find that having a web page read out to them using a synthetic voice aids their comprehension. The second designer's button design leaves these users entirely bereft. Most desktop screen reader users are also keyboard users, so they encounter the same problems as the previous group. In addition, the `<div>` element is semantically inert, offering no information non-visually that it is, indeed, a button.

Since the button's label of "Start" is in a background image as part of CSS's presentation layer, this too is unavailable to assistive technologies. It's also why the button is untranslatable into different languages, making it exclude international audiences. That's a lot of people missing out!

1 http://smashed.by/hicontrast
2 http://terrillthompson.com/blog/182

The inclusive designer anticipates these problems, because experience tells them that people differ, well, in lots of different ways. Far from being daunted or frustrated, they know that by exploiting standard conventions they can achieve more by doing less. In other words, they know when to design and when to employ what is *already designed* — a simple HTML button element, provided as standard by the HTML specification. Resizable, translatable, focusable, interoperable, stylable, restylable, maintainable, mutable, *simple*.

```
<button>Start</button>
```

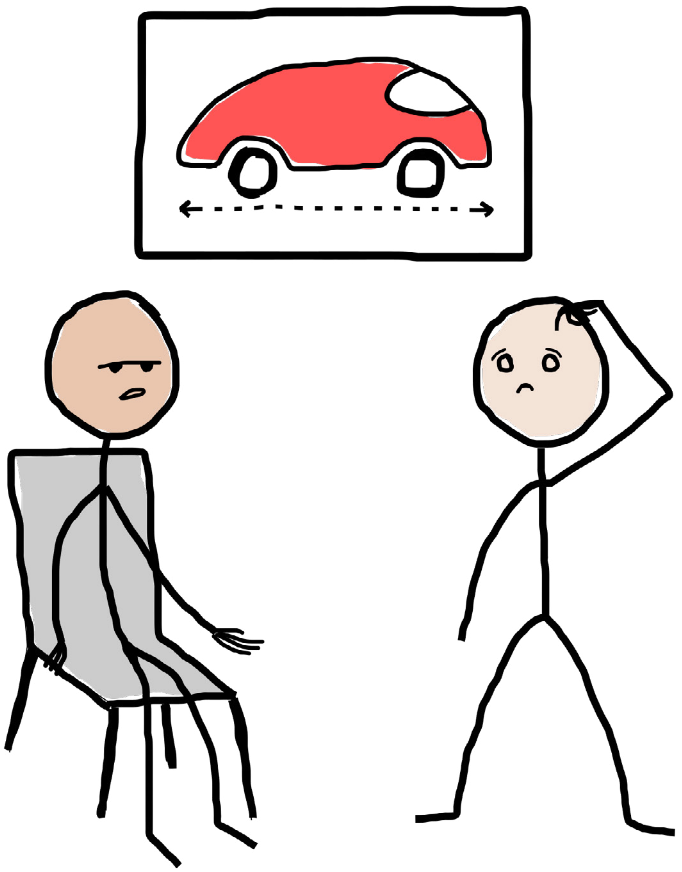
Not all inclusive design solutions are as simple as choosing the right HTML element for the job. Even so, *combining* simple, standard elements and conventional structures empathetically needn't be difficult, nor impede artistic flair: our `<button>` can still be presented in a near-infinite number of ways.

“*There are 140,737,479,966,720 combinations of hexcodes. Obviously not all of them are accessible. If only 1% of all color combinations are accessible then there are still almost 141 million combinations to choose from. This seems more than adequate to paint any bikeshed you will come across for the rest of your career.*”

— “The Veil Of Ignorance”, Adam Morse[3]

3 http://mrmrs.io/writing/2016/03/23/the-veil-of-ignorance/

With experience, identifying the countless *inclusive* solutions to interface design problems can become second nature. The patterns described in this book are reusable as exemplars, but mostly they're for practice. When new problems arise, requiring different inclusive patterns to be formulated, you will have learned how to think about formulating them. You will have become an inclusive designer.



“Yes, the design is slick. But how do people get in?
Where are the doors?”
“How do the *who* do *what* now?”

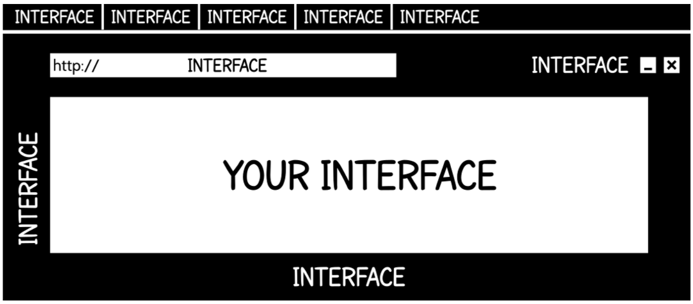
The Document

While other chapters of this book will look into discrete interface patterns — conventions realized as modules or components — it would be foolhardy to not recognize that each of these will ultimately belong to a web document. HTML pages vary dramatically in shape and size, and can include any combination of patterns; but there are a handful of document-level best practices we should stick to.

The aim here is not to go in search of the ultimate boilerplate but to configure a parent web page for an inclusive interface.

The Doctype

Whether your web page is an old-school static resource or a single-page app, it is unavoidably a document. The clue is in the "DOC" part which should appear in the first line of code: `<!DOCTYPE html>`. This serves as an important reminder that even when you're designing a highly tactile and dynamic interface, you are still really just putting content into a browser window. Never forget that the browser itself is an interface, which can be augmented and configured in a number of ways by the user — as well as helping interpret the content you provide for third-party assistive technologies like screen readers. Your subsidiary interface should be tolerant of users' differing setups and configurations.



Your interface is not the only interface users are interacting with when their browsers are open.

In addition, the *omission* of the aforementioned doctype declaration can result in unexpected and broken behavior for users. Without a doctype declared, the browser simply doesn't know how to interpret the content and can regress into a non-compliant and incompatible mode, often called *quirks mode*.[4] Layout and interaction can become error-prone and unpredictable. If things get plain odd when testing a web page, I always check for a `doctype` before trying anything else. I've been stung too many times.

The `lang` Attribute

If the `doctype` tells the browser what *kind* of document it is serving (HTML5 in the above example), then the `<html>` element's `lang` attribute tells it which language it is written in.

4 http://smashed.by/quirksmode

I'm not talking HTML or XHTML; I mean English or French.

```
<html lang="en"> <!-- language set to English -->
```

Though frequently omitted, declaring a language for a web page could scarcely be more important. Not only does it make the page more indexable by search engines, but it also becomes easier to translate by user's operating third-party tools such as Google's Translate API.[5] It also helps the user to *write* in the page's language. Firefox, for instance, changes dictionaries on `<textarea>`s so that spelling errors are highlighted appropriately.

Perhaps most starkly, a page that does not have a language declared — or has the wrong language declared for the content — will not trigger the adoption of an appropriate synthetic voice profile when used with a screen reader. That is, if `<html lang="en">` is present but the text is actually in French, you'd hear a voice profile called Jack doing a bad impression of French, rather than a French profile called Jaques using authentic French pronunciation.

```
<p>Il ne faut pas mettre tout dans le même sac!</p>
```

In addition to screen readers and computer braille displays benefiting from proper language declaration, it also helps

5 https://cloud.google.com/translate/docs

browsers choose and render system fonts with the appropriate character sets. For instance, `lang="zh-Hans”` will invoke the rendering of a simplified Chinese font. Garbled text, rendered in unbefitting characters is not optimally inclusive of your readership *to say the least*.

It is possible to switch languages within a page using the `lang` attribute on child elements within the <`body`>. For instance, I may want to quote some French within an English language page:

```
<blockquote lang="fr">
   <p>Ceci n’est pas une pipe</p>
   <p>— <cite>René Magritte</cite></p>
</blockquote>
```

In my CSS I can select any sections declared as French using the `:lang` pseudo-class, applying a font well suited to the French character set, thereby improving legibility:

```
:lang(fr) {
   font-family: French Font, fallback, sans-serif;
}
```

The `lang` attribute can be used to improve the readability, translatability and compatibility of web documents’ written content, helping open it up to an international audience. It’s easy to include the `lang` attribute, so just go ahead and include it.

Responsive Design

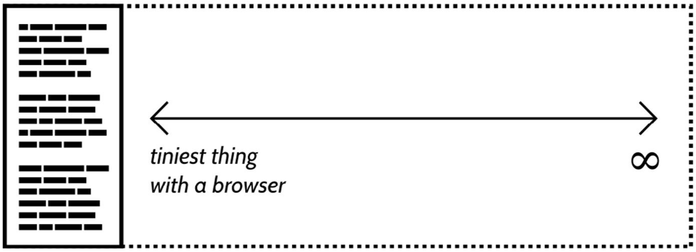
Responsive design is a big part of inclusive design. By designing documents that can respond and adapt to their environment, they become compatible with the ever proliferating choice of devices on the market. This book is not the place to discuss the ins and outs of responsive design, but you should be aware of some responsive design principles that support inclusive design.

CONTENT BREAKPOINTS

Targeting the specific viewports of specific devices (employing device breakpoints) is an exercise in futility if you want to create a truly inclusive experience. There are just too many viewport variations available to support each one. Instead, you should create an entirely flexible design from the outset and insert breakpoints only where the content breaks the layout — hence *content breakpoint* or *tweakpoint*.

By employing content breakpoints, you can ensure successful layouts for a range of devices far greater than you would ever be able to manually test with. Unless your superpower is prescience and you can anticipate the device setup of each and every one of your users, this is the only way to go.

Locating content breakpoints is easy using Firefox's responsive design mode.[6] Just press *Cmd + Option + M*, then resize the simulated viewport incrementally until the content is forced to collide, overlap or wrap. The responsive design mode interface displays the current dimensions, allowing you to record and attend to that breakpoint.



Your responsive design should cover every width. That doesn't mean you need a breakpoint for every width.

Simple interfaces are usable interfaces, and much less work in terms of managing breakpoints. As a rule of thumb, you should never set a fixed width or height on any element: the intrinsic flexibility of boxes means they can tolerate the same content in different spaces. Web pages behave like this without author CSS, and should continue to respect this fundamental behavior where CSS is included.

6 http://smashed.by/rdmode

ALLOW PINCH-TO-ZOOM

The `viewport` meta tag is where responsive design is, somewhat magically, enabled. Yet it's also where we habitually disable users' ability to zoom content, making experiences, well, *less* magical. In an informal poll I took on Twitter asking my followers about the biggest mistakes designers make regarding inclusion, the suppression of pinch-zoom on handheld devices was by far the most cited.

To be clear, then, the first of what follows is unacceptable; the second is correct.

```
<!-- don't use this -->
<meta name="viewport" content="width=device-width, initial-
scale=1.0, minimum-scale=1.0, maximum-scale=1.0,
user-scalable=no">

<!-- use this -->
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

Adrian Roselli provides a comprehensive list of reasons why disabling this feature undermines inclusion:[7]

- The text may be too small for the user to read.
- The user may want to see more detail in an image.
- Selecting words to copy/paste may be easier for users when the text is larger.

7 http://adrianroselli.com/2015/10/dont-disable-zoom.html

- The user wants to crop animated elements out of the view to reduce distraction.
- The developer did a poor job of responsive design, and the user needs to zoom just to use the page (this happens!).
- There is a browser bug or quirk (still a bug) that causes the default zoom level to be odd.
- It can be confounding for users when a pinch/spread gesture is interpreted as something else.

You may recall my remark earlier that the browser itself is an interface. To achieve inclusive design you should act as a facilitator, allowing users to configure the way they view and interact with the content you provide them. The fewer decisions you make for them, the more they can make for themselves.

If your layout breaks badly when a user zooms, *the design* is at fault, not the user, and removing their ability to zoom is not a solution. As a rule of thumb, avoid using positioned elements, especially with `position: fixed`. When content is enlarged, any elements stuck to a certain part of the screen are liable to become blind spots.

Font Size

Desktop browsers tend to render fonts at 16px by default. There is a rationale for reasonably large defaults: anything else risks alienating a huge swath of users, many from older populations whose eyesight has deteriorated. “But my audience is young and hip!” I hear you say. Sure, but generous font sizes don’t *offend* young, keen-eyed folks. The key to inclusive design isn’t to target specific groups, it’s to *not exclude* groups arbitrarily — there’s nothing to gain.

You may already be accustomed to the convention of setting font size using a percentage, on the root (`<html>`) element like so:

```css
html {
  font-size: 100%;
}
```

In the example above, `100%` is equal to `16px` *if* the user has not adjusted the font size manually, either in their operating system or browser settings (e.g. under *Preferences* → *Content* in Firefox). So, `100%` is really “one hundred percent of the default size or the size the user has chosen.” If I were to set the font size explicitly as 16px at the root level, I would diminish the user’s ability to adjust font size to their liking.

```css
html {
   /* do not do this */
  font-size: 16px;
}
```

It is true that a greater number of users will operate full-page zoom using *Cmd (or Ctrl) and* **+**, but modern browsers and operating systems still support text-only resizing, and so should you. To make sure `font-size`, `padding` and `margin` all resize proportionately, each should be set using the relative units `rem` or `em`.

This greatly reduces the complexity of your media queries. In the example to follow, the `line-height` and `margin` of the `p` will scale proportionately, because they are both set relative to the `font-size`. Where everything is set proportionately, whole pages can be scaled simply by adjusting the root font size, as in the media query that follows.

(**Note:** To ensure the media query triggers at the correct point relative to user-defined font size, we are using the em unit there, too. The rem unit causes problems in Safari, as "PX, EM or REM Media Queries?"[8] by Zell Liew attests.)

8 http://zellwk.com/blog/media-query-units/

```
p {
   margin: 1.5rem 0;
   font-size: 1rem;
   line-height: 1.5;
}

@media (max-width: 60em) {
   html {
         font-size: 80%;
   }
}
```

An <`h2`> given `font-size: 2rem;` is two times larger than the size set at the root. If the user has gone to *Settings* → *Content* in Firefox and set the default font size to 20px, `2rem` will mean 2 × 20 — or 40px. Do not make the mistake of converting theoretical rem units into pixels via a CSS preprocessor. It's what ends up in the compiled CSS, parsable by browsers, that counts.

VIEWPORT UNITS

Viewport units present the opportunity to set text that scales with the height (`vh`) or width (`vw`) of the viewport. Essentially, this means you can employ implicitly responsive text without the need for media queries like that of the previous code example.

You just need to address one issue: elements set using viewport units *cannot* be scaled using full-page zoom. Zoom can be reinstated by entering a viewport unit-based value into a sum with an em-based value. This has the added advantage of ensuring a minimum font size for the page. That is, `1em + (0 × 1vw)` is still `1em`.

```
html { font-size: calc(1em + 1vw); }
```

With this algorithm in place, everything scales incrementally and proportionately relative to the viewport. It saves you a lot of manual coding without sacrificing accessibility.

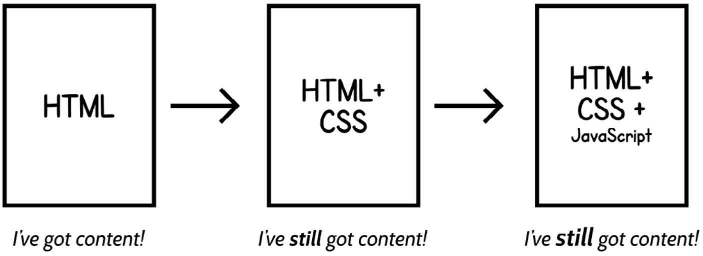
The only modern browser that does not support viewport units at the time of writing is Opera Mini. Not to worry: `font-size: calc(1em + 1vw);` is a progressive enhancement. Where it is not recognized, browsers fall back to the user agent's default. In other words, Opera Mini would display the body text as it would have liked to begin with.

Progressive Enhancement

Progressive enhancement, like responsive design, is a cornerstone of inclusive design. Much maligned as the arduous task of making web applications still do things when a unicorn-rare user switches JavaScript off, it has much broader implications.

Progressive enhancement is about building a strong foundation of content, logical and robust in form, which is resilient to a multitude of network and scripting failures. It's not just about JavaScript or CSS being unavailable, but when and how they are unavailable, for how long, and when they should be *made* available, and in what order.

Patterns in this book are founded on well-formed and semantic HTML structures, enhanced by CSS and JavaScript. Where possible, integrated JavaScript widgets degrade into well-structured static content or interactive form elements, meaning users without JavaScript or CSS — temporarily or otherwise — can traverse and use content. Whether JavaScript is available or not, semantic HTML ensures an inclusive experience for assistive technology users and makes interaction behavior more predictable and efficient.



Enhancement is a great thing, but only where it's really needed.

In a progressively enhanced setup, scripts should be inserted at the end of the document, just before the closing `</body>` tag. This allows the main DOM content to be rendered before the scripts are executed.

```
  <script>// TODO: enhancement</script>
</body>
```

Managing Assets

In terms of our document setup, it's critical we make sure the resources we are using to enhance the page content do not stand in the way of that content. On slow networks, the content should arrive as soon as possible. It's what the user went to the page for, after all.

Web fonts are typically large assets which should be treated as an enhancement. In particular, *FOIT* (flash of invisible text) should be avoided: if the font resource is indefinitely stalled (it happens!), users of some devices and browsers will be stuck with a page that has no visible text. That's pretty uninclusive of users on temperamental networks.

The trick is to load the page *then* the font, using the `onload` event as a watershed. For this to work, the fonts will have to be base64-encoded and embedded into the style sheet in question. In Keith Clark's example,[9] <`link`> has an `onload` handler which switches the media type from `none` to `all`. If JavaScript is entirely unavailable, the CSS is loaded regardless, thanks to <`noscript`>.

9 http://smashed.by/render-block

```
<link rel="stylesheet" href="fonts.css" media="none"
onload="if(media!='all')media='all'">
<noscript><link rel="stylesheet" href="fonts.css">
</noscript>
```

The base64-encoded font would be included inside an `@font-face` declaration block, like so:

```
@font-face {
   font-family: Merriweather;
   font-style: normal;
   font-weight: 400;
   src: local('Merriweather'),
url('data:application/x-font-woff;charset=utf-8;base64...');
}
```

A more comprehensive way of overcoming FOIT is offered by Bram Stein[10] and requires a small JavaScript dependency. *Font Face Observer* allows you to watch and wait on the completion of a font resource loading using a simple promise-based interface:

```
var observer = new FontFaceObserver('MyWebSerif');

observer.check().then(function() {
   document.documentElement.className += "fonts-loaded";
});
```

10 https://github.com/bramstein/fontfaceobserver

It’s simple to then supply the font in your CSS, using the `.fonts-loaded` class selector shown above:

```
html {
    /* system font with fallback */
   font-family: MySystemSerif, serif;
}

html.fonts-loaded {
    /* web font with fallbacks */
   font-family: MyWebSerif, MySystemSerif, serif;
}
```

In defeating FOIT, you have to accept a lesser evil called *FOUT* (flash of unstyled text). This is a rather unjust moniker since all fonts are styled (i.e. they have their own form). However, there can be an unpleasant and disorienting jump when the web font supplants the system font after it has loaded. This is due to one font being intrinsically larger or smaller than the other, causing lines to wrap in different places.

To mitigate this undesired effect, your best strategy is to choose fallback system fonts with intrinsic dimensions (metrics) which are similar to the web font.

Subsetting Fonts

A font that supports a large, international character set is an inclusive font. Nevertheless, you should only include a subset containing the characters you need or your users may suffer performance issues. The difference in file size between an entire font and one subsetted to just the ampersand glyph you want will be many orders of magnitude!

When including fonts using Google Fonts,[11] you can append a `text` parameter to the URI listing just the characters you need. For example, if you know your `<h2>` headings will use just the uppercase letters from a particular font, you can include the link like so:

```
<link href="https://fonts.googleapis.com/
css?family=Roboto:900&text=ABCDEFGHIJKLMNOPQRSTUVWXYZ"
rel="stylesheet" type="text/css">
```

If you are serving your own fonts, you can first subset them using Font Squirrel's generator.[12] In the CSS font stack, characters not supported by a priority font are filled by a fallback. In practice, this means a well-matched system font can supply the more unusual characters, allowing you to heavily subset the web font.

11 https://www.google.com/fonts
12 http://www.fontsquirrel.com/tools/webfont-generator

By using Font Squirrel's generator, you can roll back your web font's character set to just the Basic Latin Unicode block,[13] for example.

```
body {
   font-family: SubsettedWebFont, ExtensiveSystemFont,
sans-serif;
}
```

The <title> Element

The `<title>` element, found in the `<head>`, should be familiar to you already as the element browsers glean the text from for their tab labeling, and search engines the text for their results links. Certainly, unlabeled browser tabs are a frustrating shortcoming for most anyone, but there are additional implications for assistive technology users. Throughout this book you'll hear a lot about *accessible names*. These are the assistive technology compatible labels for the various elements of your web pages. The accessible name for a document, `<iframe>` or embedded SVG element is provided by their `<title>`. It should describe the purpose of the contents of said element.

The `<title>` is announced as soon as a new web document is loaded, so it is your opportunity to provide a succinct

13 https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block)

summary of the page. Conventional practice is to describe the page and append author and site information.

For example, “Inclusive Design Template | Heydon’s Site.” For a search results page, you should include the search term the user provided; something like, “Website name | Search results for search phrase.”

The <main> Element

Some patterns in this book, like **navigation regions**, should appear consistently between pages of the site as landmarks. Others will contribute to the modular, often dynamic content that makes up unique pages. In conventional web page anatomy, this content is designated to a main content area.

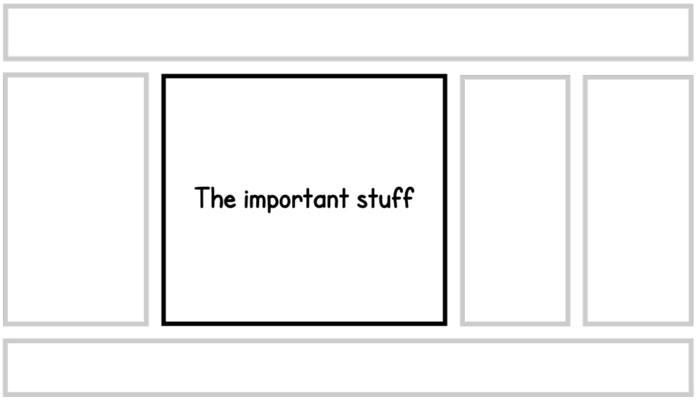
```
<main id="main">
  <!-- this page’s unique content -->
</main>
```

The concept of main content is not a new one, but it is only recently that we’ve been afforded a tool to inclusively separate it from surrounding page apparatus like headers, navigation regions and footers. The `<main>` element defines a region recognized and communicated by screen readers. Screen readers like JAWS also offer a keyboard shortcut (the *Q* key) to access `<main>`, allowing the user to bypass a page’s preamble and go straight to the content they came for. In a single-page application, your singular <main> should be where each of your functionality-heavy views is rendered. In a static blog or brochure website, <main> would contain blog posts and other informational content. A products page would describe the products within <main>.

Since <main> is designed to contain the salient content of a page, it can make the writing of a print style sheet easier. If your navigation, header, footer and sidebar (<aside>) regions are correctly made the siblings of <main>, you can target them quite easily with CSS:

```css
@media print {
   body > *:not(main) {
         display: none;
   }
}
```

I'm not one to print web pages. I gave up on home printers a long time ago because they always seemed to break after ten minutes of use. But not everyone is like me, and making sure printed pages do not contain page elements which are only relevant when browsing on-screen is an act of considerate, inclusive design. It's currently the easiest offline solution available, because it means the reader can print to PDF and save to their local drive.



Another advantage of being able to eliminate subsidiary content is the ability to improve the on-screen reading experience. Extensions are available[14] *to apply your own CSS on a per-domain basis.*

SKIP LINKS

Skip links are a classic allowance made in the name of inclusive design: they feel like an awful cludge, but their impact on the experience of some types of users is tried and tested.

A skip link appears above all other content on the page and points to the main content area. But who's it for? Conventional wisdom says screen reader users but, as I've already covered, they have other means to reach the `<main>` element. The principal beneficiaries of skip links are sighted

14 http://smashed.by/stylish

keyboard users. Such people are not afforded the same shortcuts as screen reader operators, so it's them we're mostly helping to skip over navigation and other header content.

Skip links should not appear visually by default because they have very limited utility to mouse and touch users and would only serve to confound them. To make skip links available to keyboard users, we should bring them into view on focus:

```
[href="#main"] {
  position: absolute;
  top: 0;
  right: 100%; /* moves off screen */
}

[href="#main"]:focus {
  right: auto;
}
```

When a keyboard user enters a new page, the document itself will be the first thing to receive focus. With the above provision in place, when the user hits their *Tab* key they'll focus the first interactive element on the page: the skip link. Focusing the skip link reveals it visually, giving the user the option of skipping to the main content if so desired. Tabbing again will hide the skip link and focus on the next interactive element on the page (probably the homepage link or the first link in the navigation list).

Putting The Page Together

OK, let's see how our inclusive document is shaping up.

```html
<!DOCTYPE html>
<!-- the main language of the page declared -->
<html lang="en">
   <head>
         <meta charset="utf-8">

         <!-- a viewport declaration which does not disable zooming -->
         <meta name="viewport" content="width=device-width, initial-
scale=1.0">
         <!-- a non-blocking base64-encoded font resource -->
         <link rel="stylesheet" href="fonts.css" media="none"
         onload="if(media!='all')media='all'">
         <noscript><link rel="stylesheet" href="fonts.css"></noscript>

         <!-- a non-blocking stylesheet -->
         <link rel="stylesheet" href="main.css" media="none"
         onload="if(media!='all')media='all'">
         <noscript><link rel="stylesheet" href="main.css"></noscript>

         <!-- a descriptive label for the page -->
         <title>Inclusive Design Template | Heydon's Site</title>
   </head>
   <body>
         <!-- a handy skip link for keyboard users -->
         <a href="#main">skip to main content</a>

         <!-- logo / page navigation etc. goes here -->

         <main id="main">
            <!-- unique content of page goes here -->
         </main>

         <!-- a non-blocking javascript resource -->
         <script src="scripts.js"></script>
   </body>
</html>
```

A Note On Frameworks, Preprocessors And Task Runners

It seems to me that developer ergonomics should be less important than our users' needs.

— Paul Lewis[15]

A lot of web design articles and books are about improving your workflow and making your life easier as a developer. Should you wish to adopt a framework or employ a preprocessor to speed up your development process, be my guest. However, this book is not about you; it's about your audience.

As such, any time spent on tools will only be in cases where they can have a direct impact on the quality of the user experience. Under all other circumstances we'll be exploring the possibilities offered by standard web technologies like HTML, CSS, JavaScript and SVG. How these underlying technologies actually interact with your users will be the focus throughout (alongside organizational and writing techniques that benefit users too, of course).

You can take the lessons learned here and apply them within any framework sufficiently flexible to allow you to write and organize good interfaces.

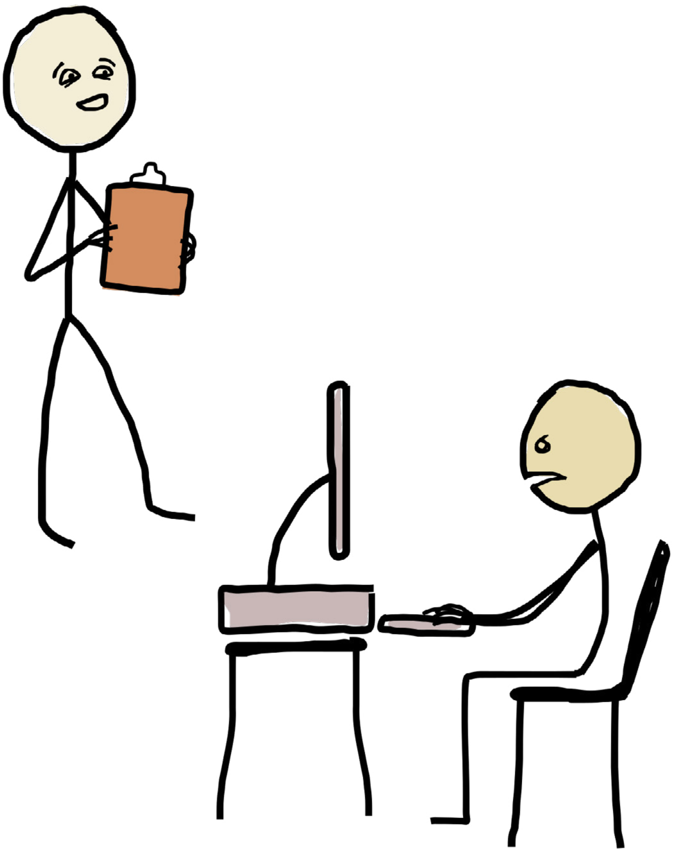
15 https://aerotwist.com/blog/react-plus-performance-equals-what/

Any opinionated frameworks not easily configurable for inclusive design techniques should be avoided. They lead you to build poor products.

To help me stay focused on the true task at hand, I have a points system. In all cases, whether I'm applying visual design, writing JavaScript behavior or structuring content, I ask who benefits from the *way* I'm approaching it.

- **1 point**: it benefits me
- **10 points**: it benefits a user/reader like me and with my setup
- **100 points**: it benefits me, people like me, *and* users/readers unlike me, with differing setups

One hundred points is what we're aiming for here.



“So, does it delight you? Are you delighted?”
“I’ll let you know when I figure out how it works.”

A Paragraph

Let's start small. In Brad Frost's atomic design terminology,[15] a paragraph would be an *atom*: an indivisible building block with which complex patterns (*molecules*, *organisms* and *pages*) are created.

Some say a picture paints a thousand words, but I prefer the contrary expression: "Never send a picture to do the job of words." Text is the most direct and efficient way of communicating information and — though imagery and other media have supplemental value for those who learn better visually or have literacy issues — it should be treated as *primary*. This applies whether you're in the business of publishing lengthy, florid prose or dotting isolated descriptions and messages around an application.

With so many exciting opportunities available to us to build interactive mechanisms, it's tempting to neglect the design of our body text. Yet not only is a readable web page an inclusive one, there's also some interactivity to address in the forms of zoom functionality and inline hyperlinks.

15 http://bradfrost.com/blog/post/atomic-web-design/#atoms

The Typeface

There is a lot of conflicting advice on what the characteristics of a readable body text typeface should be. One well-supported claim is that sans serif fonts are more readable than serif ones. Nonetheless, it is entirely possible to design an excruciatingly unreadable font which is technically sans serif.

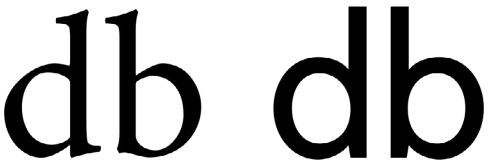
Instead of relying on conventional wisdom, here are a few things to ask about a typeface under consideration:

- Does it have any ornamentation that gets in the way of comprehension?
- Are the metrics (such as x-height)[16] consistent between letterforms?
- Are individual letterforms distinct in shape or can they be confused with others?
- Does the typeface support all of the characters and font styles that are needed?

The British Dyslexia Association notes some specific characteristics[17] that can aid legibility. Generous ascenders (e.g. the vertical line in *d*) and descenders (e.g. the down-pointing line in *y*) proved popular among dyslexic readers in their research. A *d* and *b* which were *not* an exact mirror image

16 https://en.wikipedia.org/wiki/X-height
17 http://bdatech.org/what-technology/typefaces-for-dyslexia/

of one another also helped, as well as a clear distinction between an uppercase *I*, lowercase *l* and *1*. Kerning (the spacing between characters) benefited from being reasonably generous, particularly between *r* and *n*. Otherwise "modern" could be read as "modem."



Although sans serif fonts are generally thought to be more readable, their simplicity makes them more vulnerable to having similar letterforms.

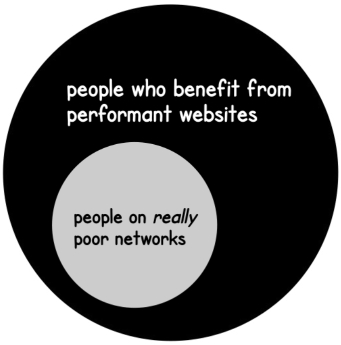
As with all things, successful comprehension is a question of offering enough but not too much information. Accordingly, serif typefaces are generally regarded as overcomplex, but a purposeful use of serifs to differentiate characters is actually beneficial.

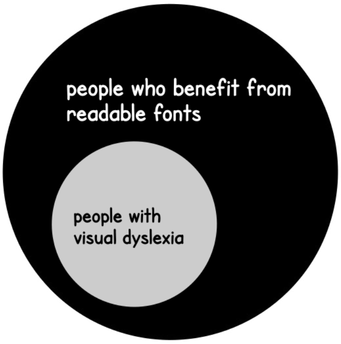
It's important to note that *legible* body text for folks with reading disorders such as dyslexia is also *pleasant* body text for those who have less trouble reading. By choosing a typeface that we feel the average user could read, we'd be consciously alienating a section of our users. Instead, by selecting a typeface which is workable for those who *struggle to read*, we arrive at a choice that works for everyone. This is efficient and effective inclusive design.

The average user is created from the combination of all users. What we get is, in fact, a completely different user. None of our users is like the average user. Therefore, when designing for that artificial individual we create something that doesn't fit anyone's needs.

– "Designing for the extremes," Susana Gonzalez Ruiz

As the excellent article "Designing for the extremes"[18] points out, designing first for users in extreme situations helps us better serve everyone. The article uses bandwidth as another exemplar: a web app that's performant on the most unreliable mobile network is, therefore, performant everywhere.


people who benefit from performant websites
people on *really* poor networks


people who benefit from readable fonts
people with visual dyslexia

However, it's important not to think about extreme cases in isolation and to target specific groups. In an experimental study carried out in Spain[19] (PDF) which investigated the readability of different fonts to dyslexic people, a typeface

18 http://sugoru.com/2013/07/14/designing-for-the-extremes/
19 http://smashed.by/dyslexiastudy

designed *specifically for* dyslexic readers performed very poorly. It would appear that the typeface designers, though well-meaning, made some generalizations about the target audience. As a result, they alienated the majority of dyslexic readers as well as creating a “clunky and difficult to read” font for everyone. In the article “A Typeface For Dyslexics? Don’t Buy Into The Hype,”[20] the creator of the Lucida typeface, Chuck Bigelow, corroborates these findings with his own research.

Typesetting

Having chosen an inclusive typeface, we ought to do it justice with good composition. Inclusive typesetting aids readability and is the task of choosing a measure, justification and leading which suit the typeface at hand. These provisions, according to Bigelow and others, are likely more important than the choice of font (unless the font is eye-wateringly illegible, of course).

MEASURE

A paragraph’s measure is the length, in characters, of one line. In text that isn’t justified (see “Justification” below) this will vary, so you should measure the longest line of a sample paragraph. Lines that are too long are difficult to read because, on

20 http://smashed.by/dyslexiafont

reaching the end of the line, scanning back to find the start of the following line becomes problematic. Lines that are too short require darting your eyes back and forth too frequently — something that becomes tiresome quickly.

In Robert Bringhurst's book "The Elements of Typographic Style," he recommends a measure between 45 and 75 characters. In CSS, `1rem` roughly corresponds to the width of the typeface's lowercase *m*, so a paragraph which is `60rem` wide could be said to have a measure of 60 — nicely within range for comfort.

Setting paragraphs' measure directly is unwise, since in responsive and modular layouts they should wrap according to their containing element. Bearing this in mind, no container of text in your layout should exceed your stated measure. For instance, your <`main`> element should be set with an appropriate `max-width`.

```
main {
  max-width: 60rem;
}
```

This has a useful outcome regarding media queries which augment the root font size: the measure adjusts proportionately. For example, where I increase the font size for wider desktop screens, the measure — which is defined using a unit relative to the font size (`rem`) — also increases.

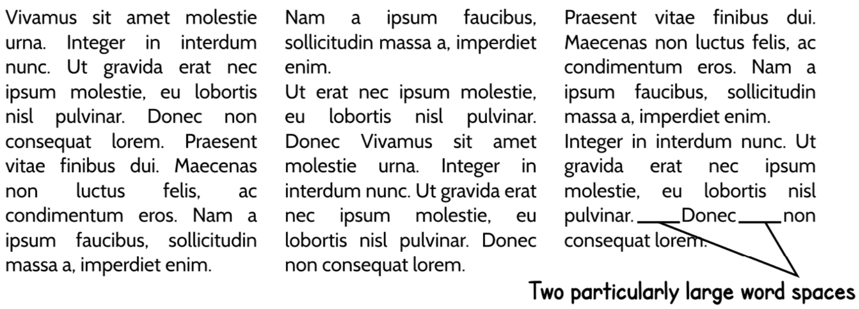
```
html {
   font-size: 100%; /* the default, so the block can be
removed */
}

main {
   max-width: 60rem;
}

@media (min-width: 120rem) {
   html {
         font-size: 150%;
   }
}
```

By virtue of relativity, we don't just set an inclusive measure but *ensure it* within responsive layouts too.

JUSTIFICATION

Justification, possible by setting `text-align: justify;` in your CSS, is generally considered bad practice for web content. Justification means making each line the same length, which makes paragraphs look neater but can severely impair readability. If you'll excuse the pun, I've struggled to justify using justified text on many occasions because I just like the way it looks. I'd be the first to admit I was putting vanity before inclusion. Bad dog!

In text justification, to make each line of equal length, words need to be redistributed within their lines, creating distractingly uneven word spacing.


Vivamus sit amet molestie urna. Integer in interdum nunc. Ut gravida erat nec ipsum molestie, eu lobortis nisl pulvinar. Donec non consequat lorem. Praesent vitae finibus dui. Maecenas non luctus felis, ac condimentum eros. Nam a ipsum faucibus, sollicitudin massa a, imperdiet enim.

Nam a ipsum faucibus, sollicitudin massa a, imperdiet enim.
Ut erat nec ipsum molestie, eu lobortis nisl pulvinar. Donec Vivamus sit amet molestie urna. Integer in interdum nunc. Ut gravida erat nec ipsum molestie, eu lobortis nisl pulvinar. Donec non consequat lorem.

Praesent vitae finibus dui. Maecenas non luctus felis, ac condimentum eros. Nam a ipsum faucibus, sollicitudin massa a, imperdiet enim.
Integer in interdum nunc. Ut gravida erat nec ipsum molestie, eu lobortis nisl pulvinar. Donec non consequat lorem.

Justified text produces the worst results when using a narrow measure, such as in the column-based layout illustrated.

In desktop publishing, this issue is mitigated by hyphenation: breaking up larger words to redistribute them more evenly. Unfortunately, CSS hyphenation, available via the `hyphens`[21] property, is poorly implemented across browsers. One of the most popular browsers, Chrome, does not support hyphenation at all. To make matters worse, JavaScript polyfills[22] for hyphenation tend to be extremely weighty because they must refer to a large library of agreed hyphenation patterns. Not all browsers even support hyphenation for languages other than English.

For good readability without JavaScript dependencies and their attendant performance issues, the default `text-align` value of `left` is preferable. This results in a ragged right-

21 https://developer.mozilla.org/en/docs/Web/CSS/hyphens
22 https://github.com/mnater/Hyphenator

hand side to your paragraphs, but users won't care. They'll be too busy effortlessly digesting your content.

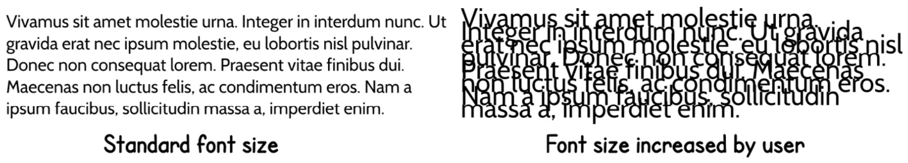
LEADING (LINE-HEIGHT)

Leading relates to the height of individual lines. It is the vertical measure between one line's baseline and the next. In the W3C's WCAG 2.0 accessibility guideline *1.4.8 Visual Presentation*,[23] it is recommended that a generous "space-and-a-half" of leading is applied to paragraphs. In CSS this can be expressed using the `line-height` property. For instance, if the font size was `16px`, the leading should be *at least* `24px`. However, as stated earlier in "The Document," it's rarely advisable to use fixed units like pixels because it makes managing comfortable proportions a headache. Instead, `line-height` should be set as a unitless proportion:

```
/* (life is too short)
p {
    font-size: 16px;
    line-height: 24px;
}
*/

p {
   font-size: 1rem; /* the default, so this line not needed */
   line-height: 1.5;
}
```

23 http://smashed.by/levelaaa

Now, as the paragraph font size is increased or decreased (either in a media query or by the user changing their font size settings), a comfortable, proportionate line height is ensured. When a pixel-based line height is used, the following is the result:


Vivamus sit amet molestie urna. Integer in interdum nunc. Ut
gravida erat nec ipsum molestie, eu lobortis nisl pulvinar.
Donec non consequat lorem. Praesent vitae finibus dui.
Maecenas non luctus felis, ac condimentum eros. Nam a
ipsum faucibus, sollicitudin massa a, imperdiet enim.

Vivamus sit amet molestie urna.
Integer in interdum nunc. Ut gravida
erat nec ipsum molestie, eu lobortis nisl
pulvinar. Donec non consequat lorem.
Praesent vitae finibus dui. Maecenas
non luctus felis, ac condimentum eros.
Nam a ipsum faucibus, sollicitudin
massa a, imperdiet enim.

The danger of a line height that is not relative to the font size.

CONTRAST

Why so many type designers are enamored with pale gray text on a white background, I'll never know. Quite apart from it causing readability issues, it just looks feeble. In any case, low-contrast combinations of text and background colors should be avoided.

Since it's easy to become wedded to brand colors early in a project, instead of testing color contrast after the fact, I recommend designing an accessible color scheme from the outset. Free tools like Contrast-A[24] (requires Flash) and Color Safe[25] are available to help you there. If you end up in

24 http://www.dasplankton.de/ContrastA/
25 http://colorsafe.co/

the unfortunate position that colors were chosen without recourse to contrast accessibility, tools like Color Contrast Analyser[26] can help you identify problem areas. I'm glad it's not me who'll have to inform your brand guru!

What's less known is that very high contrast can actually *diminish* readability for some users. Sufferers of scotopic sensitivity syndrome[27] (or Irlen syndrome) are sensitive to glare, and stark contrast can result in blurred text or text that appears to move around. It is advisable to slightly temper the contrast between your paragraph text and background color.

```
main {
   background: #eee;
}

p {
   color: #222;
}
```

As I'll restate throughout the book, there's rarely an inclusive design provision that benefits only one kind of user or reader. At this juncture I'd like to thank you in advance for dampening contrast in this way. That's because bright light has a habit of triggering migraines for me. Thank you.

26 https://www.paciellogroup.com/resources/contrastanalyser/
27 http://irlen.com/what-is-irlen-syndrome/

Inline Links

CSS is a blessing and a curse. It gives us the freedom to exercise our creativity, and it *gives us the freedom to exercise our creativity*. I'm not saying every website and app should look the same, but we have a habit of breaking with conventions just to appear radical. Here's the thing: in a world of artifice and ornamentation, *utility* is radical. And utility depends on cognition, which depends on convention. The trick is to embrace convention without dispensing with personality.

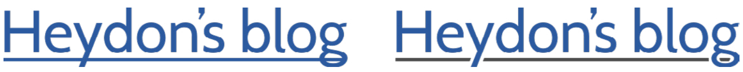
Save for button-like call-to-action links which play by different rules of affordance, it is conventional for links to be underlined. Not only does capitalizing on this convention help folks to identify links within paragraph text, but it also alleviates a more extreme concern of inclusive design: color blindness. By differentiating inline links by color alone, some readers will not be able to pick them out from the surrounding text, so there should be another means.

All browsers render text with `text-decoration: underline` by default, making inline link design a classic case of doing *nothing* being a perfectly good resolution. But it's possible to improve on `text-decoration: underline`. With custom CSS, we can keep the symbolic role of an underline but implement it better.

IMPROVED LINK UNDERLINES

Since an underline tends to sit directly below the typeface's baseline, it cuts through the descenders of letterforms like *g* and *j*, diminishing the readability of linked text. Ideally, the underline should not intersect the form of descenders but leave space for them. A solution partly developed at Medium[28] does just this, using a CSS background gradient and text shadow. Here is my adapted version:

```
p a {
    text-decoration: none;
    text-shadow: 0.05em 0 0 #fff, -0.05em 0 0 #fff,
          0 0.05em 0 #fff, 0 -0.05em 0 #fff,
          0.1em 0 0 #fff, -0.1em 0 0 #fff,
          0 0.1em 0 #fff, 0 -0.1em 0 #fff;
    background-image: linear-gradient(to right, currentColor
0%, currentColor 100%);
    background-repeat: repeat-x;
    background-position: bottom 0.05em center;
    background-size: 100% 0.05em;
}
```



When using custom, gradient-based underlines, you have the luxury of being able to control their color independently of the font color.

28 http://smashed.by/underlines

In this implementation, a `linear-gradient` background with a height of just `0.05em` forms the underline. Descenders are not crossed thanks to a text shadow which protects the letterforms like a moat protects a castle.

Note that unlike `text-decoration: underline;`, this underline is adjustable. You can tweak the vertical positioning using `background-position`, and the thickness using `background-size` to find the most readable solution for the typeface. Be wary, though, of older browsers that do not support gradient backgrounds. These should have a `text-decoration` fallback:

```
.ie-lte-9 a {
  text-decoration: underline;
}
```

INDICATING FOCUS

One approach to inclusive interaction design is to look at interfaces from the perspective of human interface devices (HIDs): different devices which allow input from and/or output to a human user. A mouse is an HID. So are keyboards, touch displays, refreshable braille displays and keyboards, joysticks[29] and switches.[30]

29 http://www.bltt.org/hardware/joysticks.htm
30 https://en.wikipedia.org/wiki/Switch_access

Designing inclusively is to support as many of these as possible and provide alternative content and interaction mechanisms where this isn't possible.

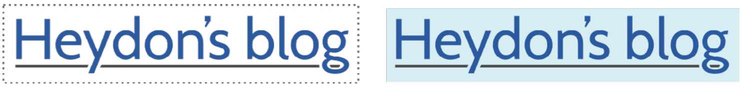
Making an element keyboard-accessible is quite straight-forward:

- Make sure the element is focusable.
- Make sure the focusable element is visible.
- Make sure the focus state of the element is visible.

Links, like the <`button`> element addressed in the introduction, are implicitly focusable. Thanks to standard browser behavior, any <`a`> element with an `href` attribute which is the next focusable element within the document will gain focus when the user hits the *Tab* key. It then becomes operable: you can follow the link by pressing *Enter*.

Even so, if links are not styled so their focus state is indicated, it's impossible for the user to see which element is indeed in current focus. This is where *focus styles* come in. Browsers implement their own focus styles ranging from dotted outlines to fuzzy blue halos. You can replace these default styles, but you should *never* just remove them. We can strengthen and simultaneously normalize the appearance of focus styles for our paragraph links across browsers by using a `background-color` solution.

```
p a:focus {
   outline: none;
   background-color: #cef;
   text-shadow: 0.05em 0 0 #cef, -0.05em 0 0 #cef,
      0 0.05em 0 #cef, 0 -0.05em 0 #cef,
      0.1em 0 0 #cef, -0.1em 0 0 #cef,
      0 0.1em 0 #cef, 0 -0.1em 0 #cef;
}
```



A background color creates a filled box which is easier to identify as focus moves between elements than the dotted outline some browsers provide by default.

This is much clearer than the thin dotted outline that many browsers apply by default. It's a style implemented on the official UK government website, GOV.UK;[31] an exemplar of inclusive service design, with a responsibility to be usable by every British citizen.

The subject of focus styles will come up in many patterns to follow, as more and different interactive elements are added to our vocabulary.

31 https://www.gov.uk/

Automated Icons

Folks who update and maintain websites can be non-technical and frequently have no confidence (nor interest) in visual design. An inclusive editorial system should be streamlined so that it doesn't force editors to think about and deal in code or formatting. It should enable them to get on with what they're good at without the fear of breaking anything.

By employing a modicum of fairly advanced CSS, it's possible to provide iconography within paragraph text which is inclusive on two counts:

- It does not complicate the editorial process.
- It provides accessible information to users.

When linking to an external resource on a separate domain, it's polite to inform the user that following the link will take them out of the current context. This is especially helpful to screen reader users who will have become accustomed to the current site's features and layout. In other words, you avoid a "Where the hell am I?" moment.

IDENTIFYING EXTERNAL LINKS

The first step is to identify which links qualify as external. Any link `href` which begins with *http* (i.e. is not a relative link) *and* does not link to the current domain passes muster.

My blog's local domain is *heydonworks.com*, so I'd build a selector like this:

```
[href^="http"]:not([href*="heydonworks.com"])
```

Translated, the selector reads, "Links with `href` values that start with *http* but do not contain the string *heydonworks.com*".

PROVIDING THE ICON

As discussed, an inclusive editorial experience is one where the editor is not asked to think and work in an unfamiliar way. In which case, we don't want to ask editors to have to upload and link icon images or — worse still — edit the source to add a `class` attribute to each external link. Instead, we can use CSS pseudo-content to automatically add the icon based on the matched selector:

```
[href^="http"]:not([href*="heydonworks.com"])::after {
  display: inline-block;
  width: 1em;
  height: 1em;
  background-image: url('path/to/external-icon.svg');
  background-repeat: no-repeat;
  background-position: center;
  background-size: 75% auto;
}
```

Note the use of relative units and a percentage-based `background-size` to make sure the icon scales with the text when it is resized by the user or in a media query.

ALTERNATIVE TEXT

That just leaves adding some hidden text to also inform screen reader users that the link is external. This is achievable using the `content` property and a few supplementary rules to move it out of sight without affecting layout.

```
[href^="http"]:not([href*="heydonworks.com"])::after {
  display: inline-block;
  width: 1em;
  height: 1em;
  background-image: url('path/to/external-icon.svg');
  background-repeat: no-repeat;
  background-position: center;
  background-size: 75% auto;
   /* alternative text rules */
  content: '(external link)';
  overflow: hidden;
  white-space: nowrap;
  text-indent: 1em; /* the width of the icon */
}
```

It's a common misconception that pseudo-content (in this case, "(external link)") is not announced in screen readers.[32]

32 http://tink.uk/accessibility-support-for-css-generated-content/

Not all screen reader and browser combinations support pseudo-content, but most do. Those that don't support it provide a less informative experience, but not a broken one.

Writing Paragraphs

It would be remiss of me to talk about inclusive paragraph design without mentioning how to actually *write* approachable, accessible content. Copywriting requires design just as much as form and composition, but in a culture dominated by visual design it tends to be put to one side; deferred until frankly less important decisions about aesthetics and ornamentation have been settled.

I'm a big believer in content-first design.[33] As stated in the principles outlined in the introduction, all other design activities should be treated as merely supportive of the content and its formation. What makes things hard is that, unlike dealing in accessible interaction states or color contrast levels, accessible wording is not easy to measure. Nonetheless, there are some general guidelines that are applicable to paragraph text:

- Write short paragraphs, sentences and words.
  "I purchased a mammalian companion of the canine variety" is never better than "I bought a dog."

33 http://alistapart.com/blog/post/content-first-design

- Be wary of passive sentences.[34] They can often be replaced by a more direct alternative. There's no need to say the convoluted, "a pork pie was eaten by Harry" when you can just as well say, "Harry ate a pork pie."

- Eliminate redundancy. It's rarely worth saying the same thing twice in different ways. Avoid repetition. (See what I did there?)

- Vary the length of sentences and paragraphs. This reduces monotony, encouraging focus.

Acknowledging the importance of good copy design will form a part of many patterns in this book. For example, "A Blog Post" will cover testing content readability against the Flesch-Kincaid scale.[35] For now, note this down: *everything but content is an enhancement*. Therefore, your prototypes and minimum viable product should contain *real* and preferably well-considered copy, ready to be tested with your users. If you're going to get something right from the outset make it the content, *not* the typography, layout or color scheme.

34 https://en.wikipedia.org/wiki/English_passive_voice
35 http://smashed.by/flesch

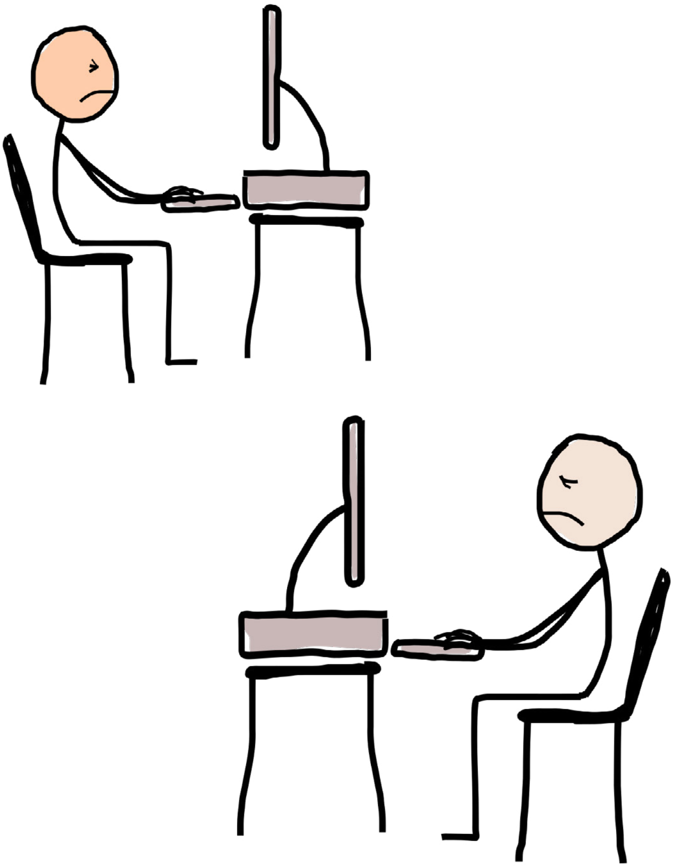
Summary

Well, there's more that goes into humble paragraph text design than perhaps you first imagined. By looking at typefaces, leading, measure, justification, contrast, focus indication and more, we've set ourselves on the right path to formulate paragraphs suited for a hugely diverse audience.

By tackling specific issues we know would affect folks with limited vision, dyslexia, Irlen syndrome, low literacy and limited technical knowledge, a comfortable reading and interaction experience has been ensured for most everyone else.

THINGS TO AVOID

- Wording and typefaces designed to impress rather than communicate.
- A measure that's too long or too short.
- Reasonably high contrast, without defaulting to stark black on white.
- Eliminating focus styles with `outline: none;`.



“Stupid date picker! Why can’t I just *type* a date?”
“Stupid text field! Why can’t I just *click* a date?”

A Blog Post

Keeping things prosaic for now, let's look at paragraphs in context.

I've had a blog for several years and it's been through four to five design iterations. Like many designer-bloggers, I seem to be afflicted with obsessive tinkering and I'm still not happy with the result. Time is a luxury we often can't afford on client projects — which is why client work sometimes gets finished, I suppose.

In my opinion, the most effective blog designs are spartan, focusing on content and removing as many distractions as possible. It's difficult to overestimate the effect of simplicity on both performance and accessibility. Not incorporating complex and ambitious layouts also puts you on the path to a more robust, flexible reading experience. When the reader adjusts something, there's simply less to break as a result.

Here, we'll investigate the composition of an inclusive blog post as found at its permalink, forming the principal content of the parent page. Semantic structure and source order must be taken into account. I'll also cover effective link text, incorporating video content, and the creation of an automated *flow system* to ease the work of content editors.

The <main> Element

You may recall the `<main>` element from our setup in “The Document.” Every web page should have a `<main>` element to identify the unique content of that page. The element helps us to think about what the main content of our page actually is, and can later be used as a navigational aid by screen reader operators.

At a blog permalink, the salient content is the blog article itself, so it should be placed inside `<main>`. Note the `id` below, which allows keyboard users to navigate to the article via a skip link (also discussed in “The Document”):

```
<main id="main">
  <!-- blog article here -->
</main>
```

When incorporating complex grid systems, we tend to rely heavily on `<div>`-based scaffolding. This may help us to corral the visual structure of the page, but it has no impact on the *semantic* structure. That is, the following code example is semantically identical to the preceding one. The source code is just more complex.

```
<div class="grid_50">
   <div class="grid_50_inner">
         <main id="main">
                  <div class="main_inner">
                           <!-- blog article here -->
                  </div>
         </main>
   </div>
</div>
```

In general, I recommend against dealing with third-party grid systems. They're typically rather large assets themselves and encourage convoluted, redundant markup. Not only does this produce a larger document to download, but also it can have additional, unpleasant side effects. For example, in testing I've experienced a screen reader producing erroneous output as it struggled to parse a massive DOM.

If you're building a highly complex layout, a grid system may help you. However, *you should not be building a highly complex layout*. Simple user interfaces are usable interfaces, accessible interfaces, maintainable interfaces, and performant interfaces.

In print design, where the relative position of grid elements is reliable and consistent, you can rely on juxtaposition to convey meaning and use terminology like, "the photograph to the right." On the responsive web you don't have this luxury. Occasionally, you'll still want to put some things next to some other things, if only to be intensive with the available space. But there's no use trying to be exacting. I'll talk more about the terseness with which you can create responsive, self-managing grids[36] using flexbox in the "A Filter Widget" chapter.

Heading Structure

I'm concerned we only usually look at HTML semantics in terms of discrete elements providing meaning, and not by looking at their relationships to one another. While it's true the `<main>` element itself provides idiosyncratic behaviors, these benefits can be diminished with improper placement; for example, by using multiple `<main>` elements on a single page or by using `<main>` to wrap the entire DOM.

Heading elements in particular derive much of their meaning from their relationships with one another.

THE <H1> ELEMENT

The `<h1>` element represents your document's first-level heading. It is the top heading of the document. At our permalink, it should be the title of the blog article itself. The `<h1>` text effectively labels the supersection that is the document body, so it's not logical to have more than one per page.

36 http://www.heydonworks.com/article/flexbox-grid-finesse

```
<main id="main">
   <h1>How To Mark Up Inclusive Blog Articles</h1>
</main>
```

Like the `<main>` element, the `<h1>` exposes a navigation shortcut for assistive technology users. Both NVDA and JAWS users can press the `1` key and their screen readers will transport them directly to the `<h1>`, where “How To Mark Up Inclusive Blog Articles, heading level one” will be announced.

Practically speaking, the `<main>` and `<h1>` offer similar functionality, so why employ both? Because not all screen reader users are alike. In my informal research into screen reader users’ behaviors and preferences,[37] some respondents were familiar and comfortable with landmark (region) navigation, but the majority used headings to get around. Providing unobtrusive options for users is sometimes called *multimodality*. It’s why we should also provide transcripts alongside video content. More on that shortly.

A number of survey respondents also professed to navigate documents from element to element, reading from the start to the end using their screen reader’s down arrow key (or equivalent). Source order is an important dimension of structure and our blog article should make sense read from

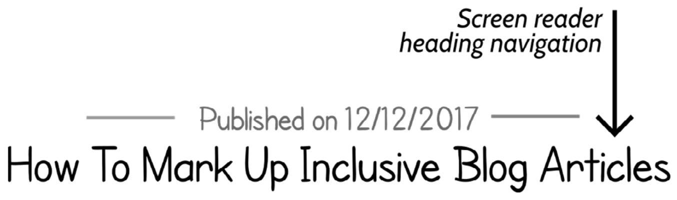
37 http://smashed.by/srsurvey

top to bottom. Though it can be made to look visually compelling either way, it is therefore important not to open an article with its publish date, but with its heading.

```
<!-- don’t use this -->
<main id="main">
  <div class="meta">Published on <time
datetime="2017-12-12">12/12/2017</time></div>
  <h1>How To Mark Up Inclusive Blog Articles</h1>
  <!-- article content here -->
</main>

<!-- use this -->
<main id="main">
  <h1>How To Mark Up Inclusive Blog Articles</h1>
  <div class="meta">Published on <time
datetime="2017-12-12">12/12/2017</time></div>
  <!-- article content here -->
</main>
```

In the first, commented out example, someone navigating by heading using the generic *h* key would be unaware of the publish date because the screen reader would take them silently past it.

Note that the first structure is unlikely to produce an error in an automated accessibility testing tool. That does not mean the second structure is not preferable. Inclusive design is about appreciating how people really use interfaces, not just fixing technical errors.


Screen reader heading navigation
Published on 12/12/2017
How To Mark Up Inclusive Blog Articles

How heading navigation bypasses section content that precedes sections' headings.

SUBSECTIONS

When it comes to structure, depth is just as important as source order. By dividing our blog article into subsections, we can describe which parts of it belong to which other parts, building a thematic map or outline. Despite the inclusion of nestable sectioning elements in the HTML5 specification, like `<section>`, the only elements currently able to describe depth in an inclusive fashion are `<h1>` to `<h6>`. That's not to say you shouldn't use `<section>` or `<article>`, but the information they provide does not serve this particular purpose. The HTML5 outline algorithm was specified to automate section structure based on sectioning elements, removing the need for `<h1>`–`<h6>`. However, it is not implemented in any user agent,[38] and there's no sign it ever will be.

38 http://smashed.by/html5doc

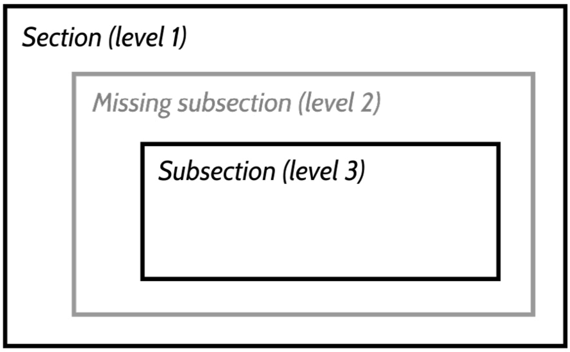
As stated, our blog article should begin with an <h1>. You could stop there, but the lack of further structural information would produce comprehension issues for a broad range of users. You should always aim to divide your content into digestible chunks. That's ultimately what HTML is for: forming structure through differentiation.

To create a subsection within the article, use <h2>. The 2 in <h2> says "two levels deep," or "one down from the first level." Visually, headings are conventionally differentiated by font size: the deeper the level, the smaller the `font-size`.

```
h1 { font-size: 3em; }
h2 { font-size: 2.25em; }
h3 { font-size: 1.5em; }
```

This is where a lot of developers make a mistake: they choose a heading element which they believe best fits the perceived importance of the section it represents. For example, a developer might choose an <h3> in place of an <h2> because they deem the section unimportant and would like to reduce the `font-size` accordingly. This deceives both visual and non-visual readers because it breaks the perceived structure — a level has been skipped.

```
<h1>How To Mark Up Blog Articles</h1>
<!-- introductory content -->
<h3>A quick note on the word 'semantic'</h3>
<!-- wait! Where's the second nesting level? -->
```



Missing layers in document structure can be disorienting to users trying to build a mental map of the document.

Some developers handle heading styles separately using CSS classes. This gives them the freedom to change font sizes independent of semantic structure.

```
<h2 class="h3">What am I?</h2>
```

While this method eschews having to make structural errors to achieve visual effects, exercise caution: in most cases, an `<h2>` should *look* like an `<h2>`. That's what it *is*, after all! Part of inclusion is achieving a parity of experience across different user needs. By managing content visually one way and non-visually another way, you begin to segregate seeing and non-seeing users. A syndicated version of your content, subject to a different style sheet, would diverge in visual structure too.

SUBTITLES

Blog articles in which the author has not been able to say everything they want in a title alone sometimes employ subtitles or straplines. This also applies to books…

The previous example could be rewritten to take the title "How To Mark Up Blog Articles" and the subtitle "In Seven Simple Steps." The question is how to mark this up. I've seen many an implementation which chooses to put the subtitle in a separate heading element, like so:

```
<main id="main">
   <h1>How To Mark Up Blog Articles</h1>
   <h2>In Seven Simple Steps</h2> <!-- this should not be a
heading -->
   <div class="meta">Published on <time
datetime="2017-12-12">12/12/2017</time></div>
   <!-- article content here -->
</main>
```

Now that you're familiar with the structural contribution of headings, you should be able to see the problem: the `<h2>` creates an immediate subsection, "In Seven Easy Steps." That's a pretty weird, fragmentary heading for a section and would make little sense wherever a table of contents is generated:

- How To Mark Up Blog Articles
    - In Seven Simple Steps
    - Headings
        - Subsections
    - Link text

For a short time, the <`hgroup`> element was the prescribed solution. By wrapping the <`h1`> and <`h2`> in an <`hgroup`>, you could theoretically remove the <`h2`> from the outline, preventing it from creating a subheading:

```
<!-- do not use -->
<hgroup>
   <h1>How To Mark Up Blog Articles</h1>
   <h2>In Seven Simple Steps</h2>
</hgroup>
```

As this MDN article on <`hgroup`>[39] attests, however, <`hgroup`> has been removed from the HTML5 specification. You cannot therefore depend on it becoming implemented uniformly or reliably in browsers.

Finding the right solution depends on what we want to achieve. If we want “In Seven Easy Steps” to be part of the title, we can separate it with a semantically inert <`span`> element and drop it onto a new line using `h1 span { display: block; }`.

39 http://smashed.by/hgroup

```
<h1>
   How To Mark Up Blog Articles <span>In Seven Simple Steps
</span>
</h1>
```

This would produce the following, corrected outline:

- How To Mark Up Blog Articles In Seven Simple Steps
    - Headings
        - Subsections
- Link text

Instead, you might deem the subtitle unimportant and want to remove its text from the outline altogether. In which case, the following solution could work:

```
<h1>How To Mark Up Blog Articles</h1>
<p><span class="visually-hidden">Subtitle:</span> In Seven
Simple Steps</p>
```

Note the use of the <span> with the `.visually-hidden` class. Visually, the role of the subtitle would presumably be indicated by its styling. In a non-visual context, this would not be apparent: Traversing from the <h1> to the <p> would have a screen reader believe that they had begun reading the article body. By providing the prefix "Subtitle:" we clarify the role of the <p> element non-visually, in much the same way as the implicit role of a <button> element makes screen readers announce "Button."

The `.visually-hidden` class invokes a special set of properties, carefully devised to hide the element visually without it becoming silenced in screen reader software. Using `display: none;`, `visibility: hidden;`, `height: 0;` or `width: 0;` wouldn't work — it would make the span unavailable both visually *and* aurally.

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

Having such a utility class in your style sheet and at your disposal is always recommended. In later chapters, we'll call on `.visually-hidden` a number of times.

Should a <`subtitle`> element ever be specified, the clarifying "Subtitle:" prefix would probably be provided for you as a feature of the element's role, and therefore automatically available in the "accessibility tree"[40] — the version of the DOM used by assistive technologies where roles, properties, states, labels and values are exposed. Aside from providing interactive behaviors, this is one of the simple benefits of using semantic elements where they are specified and supported.

40 http://smashed.by/a11ytree

The <article> Element

There is a lot of confusion surrounding HTML5 sectioning elements. For example, you may be asking, "If this is an article, shouldn't it be in an `<article>` element?"

```
<main id="main">
   <article>
         <h1>How To Mark Up Inclusive Blog Articles</h1>
         <div class="meta">Published on <time
datetime="2017-12-12">12/12/2017</time></div>
         <!-- article content here -->
   </article>
</main>
```

The answer is: probably not, but maybe. Theoretically, the `<article>` *should* start a new subsection by deferring to the HTML5 outline algorithm.[41] It would be poorly placed here because we haven't even begun the main content of the document (which is the outermost section). But, since the outline algorithm is not actually implemented in any user agents,[42] this has no effect on assistive technology users. No harm done. Unlike the `<main>` element, `<article>` is not a landmark, so few screen readers provide it as a navigational tool. Since it's the singular child of `<main>`, it would take the user to the same place anyway.

41 http://smashed.by/html5outline
42 http://smashed.by/html5doc

Altogether, deploying `<article>` doesn't look very appealing, but there is a *small* advantage in certain contexts. The JAWS screen reader (and *only* the JAWS screen reader) announces "Article" on entering the element and "Article end" on exiting it. Arguably, this is useful information where there are several `<article>`s on the same page and the user is reading from top to bottom, element by element. The following markup is serviceable for a blog's homepage, where multiple recent posts are listed.

```
<article>
   <!-- first article's content here -->
</article>
<article>
   <!-- second article's content here -->
</article>
<article>
   <!-- third article's content here -->
</article>
```

Try to think in terms of user experience when choosing semantic elements. Sometimes they may be the technically correct element to use, but they're not supported in any way so have no impact on anyone. Other times they *do* have an impact but the part they play is confusing, inconsistent or obstructive.

In her talk "Burn Your Select Tags" (video),[43] Alice Bartlett shares the research she undertook at the UK Government Digital Service[44] into `<select>` element usability. Universally implemented and semantic though it is, there's compelling evidence that you should avoid using the `<select>` element wherever possible. Not only are there discrete technical shortcomings (such as certain devices suppressing the zoom of `<option>` overlays), but they just don't seem to be *understood* by all sorts of people.

Progressive Enhancement And Interoperability

The sound semantic structure we've been fostering in our blog article doesn't just benefit screen reader users. If a browser fails to load your style sheet, it will still differentiate key elements of the interface using its own default user agent style sheet. Headings, for instance, will appear in bold text and have font sizes indicative of the section depth they represent.

In addition, services that syndicate your content, such as feed readers, will be able to apply their own structural styles, enabling a comfortable reading experience within their own

43 https://www.youtube.com/watch?v=CUkMCQR4TpY
44 https://gds.blog.gov.uk/

applications. Content that works well in different contexts and via different inputs can be said to be *interoperable*. Good semantic structure works well with screen readers, feed readers, search engines and all sorts of other parsers looking for structural meaning in your content.

A good way to test that your content has a sound structure is to turn CSS off and see if the page is still readable. Are the order and hierarchy apparent? Is the page still navigable and usable?

A NOTE ON SINGLE-PAGE APPLICATIONS

Single-page applications are typified by their use of client-side JavaScript to render and rerender content. Where content is subject to a lot of change as a result of user interaction, this approach enables an immediacy not possible when making round-trips to the server.

However, when the content you are serving is static and intended for reading alone, relying on client-side JavaScript to render that content is an underperforming and relatively unreliable method. Content should be available to users who do not have JavaScript running or for whom JavaScript has thrown an error, been blocked or failed to load.

Providing server-rendered content also means that content will be found by folks using search engines, and is parsable by third parties using command line tools like cURL.[45] It's for all these reasons that Tantek Çelik, cofounder of the IndieWeb movement, believes JavaScript-dependent static content is not in keeping with web design fundamentals. In short:

If it's not curlable, it's not on the web."

— "js;dr = JavaScript required; Didn't Read," Tantek Çelik[46]

When a site like smashingmagazine.com sends prerendered content to the client, I can print that content to my console using the `curl` command. If JavaScript was used to construct the content in the client, all I would see is the code for the outer web page, with its link(s) to the JavaScript which `curl` is unable to run.

```
curl https://www.smashingmagazine.com
```

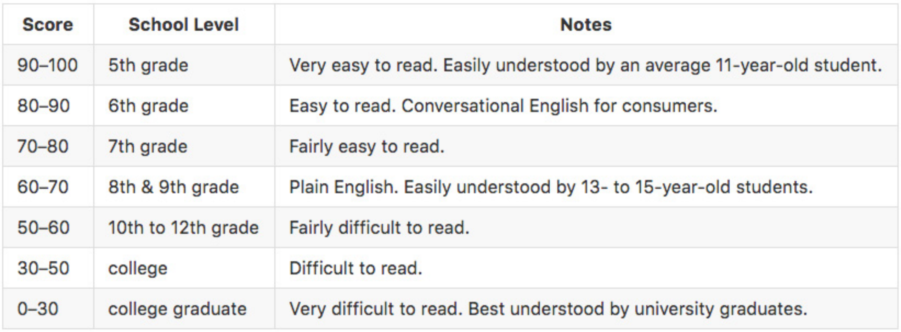
Flesch-Kincaid Readability Tests

In "A Paragraph" I offered some basic guidance on writing easily digestible body text. Now that we're putting a few paragraphs together within a blog article, it's probably a good time to review the readability of your content.

45 https://curl.haxx.se/docs/httpscripting.html
46 http://tantek.com/2015/069/t1/js-dr-javascript-required-dead

There are two tests you can perform on your content which assess word, sentence and paragraph length to determine the readability of your text.

The *Flesch reading ease* test gives you a score between 0 and 100 where a higher score means more easily readable and, therefore, more inclusive of different readers. The *Flesch-Kincaid grade level* test is similar but produces a US school grade level. The idea is that the younger the audience who can successfully read it, the more readable it is. Here's a summary from the tests' Wikipedia page:[47]



Score	School Level	Notes
90–100	5th grade	Very easy to read. Easily understood by an average 11-year-old student.
80–90	6th grade	Easy to read. Conversational English for consumers.
70–80	7th grade	Fairly easy to read.
60–70	8th & 9th grade	Plain English. Easily understood by 13- to 15-year-old students.
50–60	10th to 12th grade	Fairly difficult to read.
30–50	college	Difficult to read.
0–30	college graduate	Very difficult to read. Best understood by university graduates.

There are a number of tools to test your content against Flesch-Kincaid, including the TRAY readability tool[48] (a Chrome browser extension) or, if you want something to run from the command line, I've created Readability

47 http://smashed.by/flesch
48 http://smashed.by/tray

Checker CLI[49] — a Node.js CLI (command line interface). Just install `readability-checker` globally and point it at a web page.

```
npm i readability-checker -g

readability http://your-site.com/about.html
```

Here's what it will return:

- Flesch reading ease score (out of 100; higher means more readable).
- Notes about the score.
- A list of the longer words (more than four syllables).
- A list of the longer sentences (more than 35 words).

Recently, products like Tenon[50] have begun to emerge: quality assurance tools focused on accessibility and inclusion. Amid a maintenance-centric quality assurance culture largely designed by developers *for* developers, this is most welcome. If poor code style like inconsistent indentation breaks a build, that's one thing; but if a build breaks because the product is in danger of alienating a user? All the better. I see no reason why poor readability should not also be considered a point of failure.

49 https://github.com/Heydon/readabilityCheckerCLI
50 http://tenon.io/

Note, however, that readability is rather subjective. Scientific though a formula for determining readability may sound, you cannot rely on Flesch-Kinkaid tests alone. Use them as an early warning system to highlight potential issues, and test with real users as well.

Heading And Link Text

If you work as a front-end engineer, you are no doubt familiar with the virtues of modular design. By creating reusable modules (or components), you can develop more rapidly, with tried-and-tested code, and eliminate redundancy. A good module has an independent purpose that can be lent seamlessly to a number of different contexts.

For the purposes of inclusive design, sometimes our text has to have a certain autonomy as well. Take the following `<h2>` heading as an example:

```
<h2>Free, you say? Then yes, please!</h2>
```

In the context of surrounding paragraph text, it is no doubt possible to infer what the heading is referring to. That is, reading from the previous section into this one, you'd have a good idea of what is free. But this isn't much help in a table of contents, listing headings outside of body text. *What's* free?

Direct, descriptive headings clarify the ensuing content, which aids comprehension across the board. In other words, being cryptic or whimsical doesn't get you very far! There's also a specific implication for screen reader users: screen readers tend to catalog headings dynamically and offer them as a list to choose from. For instance, in NVDA, I can open the *Elements List* dialog with *Insert* + F7 and browse a list of headings on the page. Running VoiceOver you can open the *rotor* (*Ctrl* + *Option* + *U* on Mac OS X) and see headings listed in a similar fashion.

Accordingly, the following heading content would be of greater utility (at least if this section of the blog article is indeed about flapjacks, otherwise it would be rather misleading!).

```
<h2>Free flapjacks, you say? Yes, please!</h2>
```

OBNOXIOUS LINK LABELS

Links, like headings, are made available as a list in NVDA's elements dialog, and other screen readers aggregate them in a similar fashion. Like headings, link text should be autonomously meaningful. You may have encountered the habit of bloggers using runs of inline links like so:

I have a lot of support to back up my amazing ideas!

Cute, but each link has an entirely meaningless label when taken out of context, not to mention that even read sequentially “a link lot link of link” doesn’t tell us anything useful. In addition, it’s not always clear visually that there are separate links present. That depends on you identifying breaks in the underline. It’s even worse if you haven’t actually *provided* a link underline.

If you’re going to cite the work of other authors, the polite thing to do is mention their works by name. The links in the following example all indicate where they actually take you, and the commas — plus the “and” — help to separate them.

I have a lot of support to back up my amazing ideas, including *Why Heydon Is Right* by John Thoughtleader, *In Support Of Heydon* by Jane Unicornfield, and *When Heydon Talks It’s Like He Vomits Fragrant Rainbows* by Harry Surname.

Video

As a writer, it is the differentiation and clarification of runs and blocks of words that still excites me most about HTML, old news though this basic feature is. But I appreciate that not everyone is a confident writer or an avid reader. The addition of images, sound and video to blog posts gives color and variety to content, as well as providing more options to those with different abilities and preferences, or in different circumstances.

Take video. Sometimes it's just better to be *shown* rather than told about how something works. It's simply a better way of learning certain things. Other times I might want to be told but *not* shown, because I need to be looking at something different at the same time. For example, while I'm working on a visual design for something, I might want to simultaneously listen to the dialog from a conference talk video. On a bus, and without a set of headphones handy, I might want to be shown *and* told, but without blaring audio at my fellow passengers. Furthermore, watching speakers in the video with captions also available can assist me as a non-native speaker. A well-captioned video is just the ticket.

Of course, having captions also caters for those who are deaf and hard of hearing, but not *just* them. It's better to see captions as simply "another way to consume the same content." Then you get a better idea of just how many people can make use of them. But captions are only available if the video itself can be downloaded or streamed. This is why video that contains dialog should always be accompanied by a transcript.

A video blog post should feature the video prominently, but provide a transcript of the video underneath it. Here are a few pointers on posting videos.

THE PLAYER

Make sure the video player is keyboard and screen reader accessible. The fastest way to know if a player is worthy of your use is to try tabbing into it with your keyboard. If you find focus bypasses the player altogether, then game over. Retreat. Find another player.

Second prize for Worst Keyboard Accessibility Implementation goes to players which you can tab into and use, but without any visual feedback. Remember the "Indicating Focus" section in the last chapter? In short: no focus styles, no good. That said, if the various player controls are implemented in HTML and CSS and you can provide your *own* focus styles, the player may be salvageable. That is, if it's screen reader accessible too.

To get up and running with screen reader testing, I recommend you do the following:

- Install the free NVDA screen reader[51] for Windows.
- Run it with Firefox exclusively (for maximum compatibility).
- Watch this excellent video introduction from Deque Systems.[52]
- Refer to this list of NVDA keyboard commands[53] from WebAIM.
- Remember that your experience as a screen reader user will differ dramatically from blind users, who have dramatically different strategies even among themselves. (Refer to the responses of my screen reader survey[54] to gain an impression of the diversity.)

Alternatively, if you are a Mac user, simply press *Cmd + F5* to activate the built-in VoiceOver screen reader and follow along with the tutorial. Make sure you use VoiceOver with Safari for the most reliable experience.

51 http://www.nvaccess.org/
52 http://smashed.by/nvdatesting
53 http://webaim.org/resources/shortcuts/nvda
54 http://smashed.by/srsurvey

Fortunately, testing an interactive widget like a video player doesn't require a working knowledge of the vast array of keyboard commands offered in NVDA and other readers. Just tab to the controls as you would when testing keyboard support and listen for aural feedback. For instance, the play button should say "Play button" (or similar) when focused, or "Pause button" if the video is already playing. Of course, you should make sure the button can indeed be switched from a *press-to-play* to a *press-to-pause* state. This should work by pressing either the *Space* or *Enter* keys whether or not a screen reader is running.

RECOMMENDED PLAYERS

- YouTube's embeddable player
- Able Player[55]
- Accessible HTML5 Video Player[56] (from PayPal)
- Accessible and Responsive HTML5 Video Player[57] (by Laura Kalbag)

CLOSED CAPTIONS

Closed captions are captions which exist in a separate, associated file and can be switched on or off by the user, rather than being embedded into the video itself. You can manually

55 https://ableplayer.github.io/ableplayer/
56 https://github.com/paypal/accessible-html5-video-player
57 https://ind.ie/blog/accessible-video-player/

add caption files to HTML5 `<video>`[58] using a WebVTT file[59] as the `src` of a `<track>` element. But the easiest way (that I'm aware of) to *compose* closed captions is to use YouTube's intuitive GUI, for which Google offers a help page[60] to get you started.

There are certain conventions you should follow when writing captions. The most important of these are to identify:

1. Changes of speaker
2. Background/incidental sound

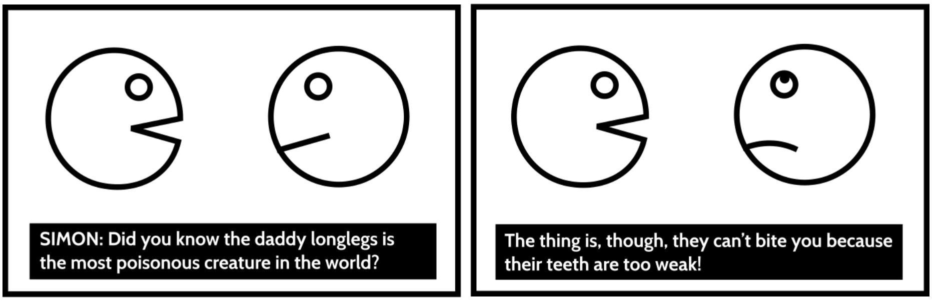
SPEAKER IDENTIFICATION AND SOUND EFFECTS

Let's say our video involves two men called Simon and Rupert. They're sitting at a table in a pub. Simon begins the conversation by claiming that the daddy long-legs spider[61] has the world's most poisonous venom and continues to describe how the spider is, nevertheless, harmless because its teeth are not strong enough to pierce human skin.

He means *fangs*, of course. Imagine a spider with teeth!

58 http://smashed.by/html5track
59 https://w3c.github.io/webvtt/#styling
60 https://support.google.com/youtube/answer/2734796?hl=en-GB
61 https://en.wikipedia.org/wiki/Pholcidae

In the following illustration, I've chosen to identify the speaker with their name written in capital letters. This is not the *only* way to identify speakers, but whatever convention you use, make sure it is consistent. Note that the name is not needed for the second caption as the speaker has not changed.

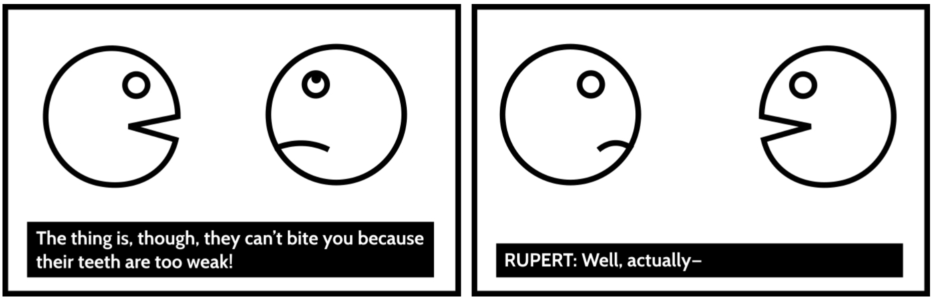

SIMON: Did you know the daddy longlegs is the most poisonous creature in the world?
The thing is, though, they can't bite you because their teeth are too weak!

In the left frame, the left speaker is identified in the caption using an all-caps name followed by a colon.

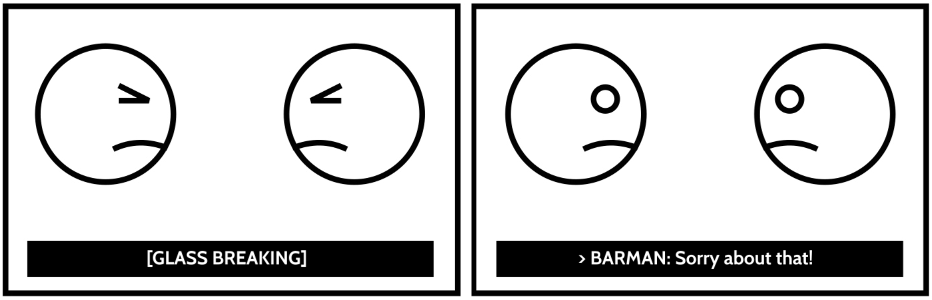
Rupert is aware that this factoid is just an internet rumor and interjects to correct Simon on his false claim.[62] Since Rupert is a new speaker, he must be identified. We use the same convention.

Before Rupert can finish his sentence, the two speakers are interrupted by a loud crash as the barman drops a glass (out of frame).

62 http://smashed.by/livescience


The thing is, though, they can't bite you because their teeth are too weak!
RUPERT: Well, actually–

Often, the description of such a noise is both in caps and bookended by [and] (square brackets). The barman immediately apologizes. I identify that he is out of frame by prefixing a greater-than symbol.



The BBC provides extensive guidelines[63] on different options to identify speakers, announcements, sounds, music, and much more.

63 http://bbc.github.io/subtitle-guidelines

THE TRANSCRIPT

The transcript would form the main textual content of your video posting. Sometimes it's easier to write the transcript first, then turn the transcript into captions. In any case, it should take the form of a linearized version of the captions. The only substitution I have made, for clarity where there is no visual context, is that the barman is identified as being out of shot:

SIMON: "Did you know the daddy long-legs spider is the most poisonous creature in the world? The thing is, though, they can't bite you because their teeth are too weak."
RUPERT: "Well, actually—"
[GLASS BREAKING]
BARMAN (out of shot): "Sorry about that!"

Establishing A Flow System

Inclusive web design isn't just about providing robust, accessible experiences for users. It's also about facilitating and easing the *contribution* of content by site owners and editors.

Like a grid system, a *flow system* uses CSS to manage layout. Unlike a grid system, a flow system deals with the relations between flow elements which appear one after another in a single column — headings, paragraphs, lists, images, videos, and so on. A good flow system produces a regular, wellspaced, readable procession of elements regardless of their order and combination. This gives editors the freedom to just go ahead and write.

EVERYTHING DIVISIBLE BY LINE-HEIGHT

Though a strict vertical rhythm[64] is notoriously tough to establish and maintain, it's good practice to at least use your paragraph line height as a basis for vertical spacing. So, if paragraphs have a `line-height` of `1.5`, one unit of vertical whitespace should be `1.5rem`.

To enforce a regularity independent of the flow elements incorporated in the editor's WYSIWYG or Markdown editor, this basic spacing should be applied generally, with just a few exceptions.

```
main * + * {
  margin-top: 1.5rem;
}

li, dt, dd, br, th, td {
  margin-top: 0;
}
```

Note the use of the *owl selector*[65] in the first line. This ensures that top margins only occur *between* successive elements.

64 http://smashed.by/verhythm
65 http://alistapart.com/article/axiomatic-css-and-lobotomized-owls

If instead we applied margin directly to elements, there would be the danger of it doubling up with the padding of containers:

```
.container {
  padding: 1.5rem;
}

.container p {
  margin-top: 1.5rem;
}

/* visible space inside top of .container would now be 3rem */
```

GROUPING AND SEPARATION

Currently, every flow element is separated by the same `1.5rem` margin, with a few exceptions. List items (<`li`>s) do not take margin at all and consequently look grouped together. So they should: for the sake of comprehension on the part of the reader, related elements should be visually proximate.

By changing the unit of measurement from `rem` to `em`, we can achieve a similar effect for subsections of content within our blog: a margin set in `em` on a heading will be relative to its own font size. Since heading text is larger than body text, this means the heading will appear closer to the content introduced below it.

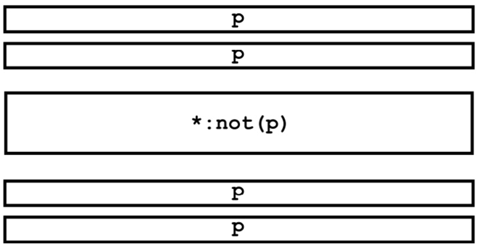
```
* + h2,
* + h3 {
    /* em, not rem, now */
   margin-top: 1.5em;
}
```

To separate supplementary content such as blockquotes and illustrations from the paragraph text, you can apply a greater margin for anything that isn't a `<p>`:

```
main * + *:not(p) {
   margin: 3rem 0;
}
```

To maintain vertical rhythm, a multiple of our paragraph line height is used (1.5 × 2 = 3). Note that the standard `1.5rem` top margin of the element following will be superseded, resulting in a `3rem` margin as prescribed both above and below the element in question. This is thanks to collapsing margin behavior.[66] The editor can insert a `<blockquote>`, `<figure>`, `<audio>` or any other element and the spacing will communicate a break from the prose.

66 http://smashed.by/cssbox



Successive paragraphs have less margin between them than paragraphs and other elements. In CSS, there's little more inclusive than the universal (star) selector. It can help us manage content and its interrelationships inclusively.

A NOTE ON DEFENSIVE CODING

I've always found the term WYSIWYG (what you see is what you get) a bit misleading, since WYSIWYG editors almost invariably have a habit of leaving lots of undesired, invisible junk in the source.

```
<p></p>
<p></p>
<p></p>
<!-- ad infinitum -->
```

In some circumstances this can have an impact on the spacing and vertical rhythm of the rendered content. We can employ some defensive coding here to remove empty elements from the flow. The `display:none;` declaration eliminates the element's layout, including any margins attributed to it.

```
main :empty {
  display: none;
}
```

In general, it's best to encourage the creation of semantic, well-structured content with a syntax like Markdown[67] or Textile.[68] They offer a simple syntax which corresponds to semantic HTML elements such as headings and list items. Where needed, most editors (like the Penflip[69] editor used to write this book) also allow raw HTML input, meaning editors versed in HTML have the option to add more complex content.

In lieu of training editorial staff to make the switch to structured content and to patch legacy WYSIWYG-based systems, a defensive strategy can help reduce headaches. Time spent fixing layout bugs when entering content is wasted time and contributes to a frustrating and alienating editorial experience.

Summary

In the conception of our blog post pattern, I covered one of the most fundamental aspects of inclusive design: *structure*.

67 https://en.wikipedia.org/wiki/Markdown
68 http://txstyle.org/
69 https://www.penflip.com/

By incorporating accessible landmarks and a sound section structure, the content — blog post or otherwise — becomes more navigable and interoperable by a diversity of users and parsers. This is bolstered by giving well-written and context-independent structural and navigational cues. We then moved on to supplementing this robust structure with additional media such as video.

Not just one structure but an infinite number of legitimate configurations of body text, headings and media elements were catered for in our robust and inclusive CSS flow system. This ensured that editors would be able to write freely without having to worry about breaking the visual design.

THINGS TO AVOID

- A lack of navigable, structural cues or bypass blocks[70] like landmarks and headings.
- Whimsical, cryptic or partial labels for headings and links.
- CSS margin declarations not tolerant of and reactive to changing contexts.
- Making your static content site dependent on client-side JavaScript.

70 http://smashed.by/navskip

Evaluation By Pattern

When I started working on this book, I prepared a rough outline first. This skeleton was organized around theme, or principle. That is, I had a chapter stubbed out about keyboard interaction, one about color, another on writing, and so on.

It's good to be prepared, but better to be prepared in the right way. As I embarked on fleshing out the various chapters, it quickly became apparent that I'd wrong-footed the structure entirely. It just wasn't compatible with the way I instinctively think and work.

The trouble was, when I set about describing how to make something keyboard-accessible, I had to bite my tongue not to talk about touch interaction, screen reader compatibility and copywriting as well. It didn't feel right to expound on making something inclusive in one regard without taking care of all the other problems that might undermine the same example. I'd be leaving the job unfinished.

When we design interfaces, we don't deal in abstract principles, we create working *things*. Our terminology for these things fluctuates between *modules*, *components*, or the looser term *patterns*, but essentially we mean the parts that contribute to an interface.

When creating patterns, we don't labor according to any one guiding principle; we don't create keyboard-accessible patterns or internationalized patterns. We want these patterns to have *all* the good qualities and, of course, none of the bad ones.

I ditched the contrived inclusion-by-principle structure early on in favor of talking through discrete patterns — motifs and components you might find in a working website or application. My writing began to flow much more easily because I was essentially documenting my design process: the work of conceiving and realizing inclusive content and functionality.

Since I'm speaking to you as designers, developers and makers yourselves, I figured this would make for clearer and more transferable information for you too.

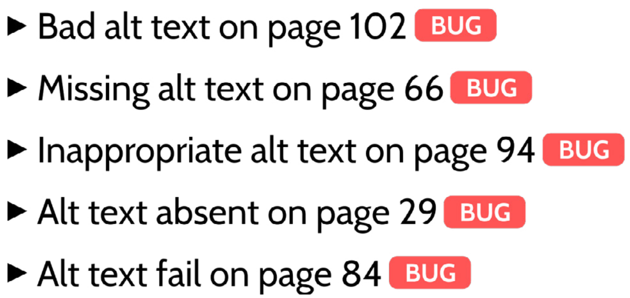
The Problem With Evaluating By Principle

The reason I started out writing by principle is because I often have to wear my accessibility engineer hat and perform accessibility audits. I evaluate the accessibility of a website or application, identifying problems and offering guidance for 'remediation' (making things unbroken).

When performing an audit, I defer to WCAG (Web Content Accessibility Guidelines). Though different countries have their own accessibility legislation, WCAG offers the de facto international rule set and is ratified by the W3C standards body. On the whole, it's better I don't *just* tell folks how *I* think they should have designed their interface!

That WCAG is organized by principle — *perceivable, operable, understandable, robust*– is to be expected. WCAG doesn't know in advance precisely which elements, composites and conventions you are going to employ so it has to remain abstract. However, this has led manual evaluation experts and automated test tools to *report* according to principle.

We're fairly wedded to reporting accessibility failures by principle because it's easier that way to reassure ourselves each and every independent issue has been covered. I'm just not convinced development teams find information presented in this way easily actionable. When developer focus moves between integrated components, what good are a hundred disparate and application-wide missing alternative text tickets?



- ▶ Bad alt text on page 102 BUG
- ▶ Missing alt text on page 66 BUG
- ▶ Inappropriate alt text on page 94 BUG
- ▶ Alt text absent on page 29 BUG
- ▶ Alt text fail on page 84 BUG

SIGH! Time to put the headphones on and fire up some Napalm Death.

But that's not all. A lot of the time, one issue apparently belonging to one principle or theme is related — either by cause or correlation — to another. Contriving to turn one instance of poor design into two or more reported issues not only complicates the remediation process, but does nothing to educate the developer about how these problems really come about.

THE BUTTON EXAMPLE (AGAIN)

Let's take another look at the button example from the book's introduction. Except this time let's imagine the faux button is for upvoting some content.

The button markup looks like this:

```
<div class="upvote" data-action="upvote"></div>
```

And the button appears like so:

The separation between style and behavior between the `class` and `data-action` attributes is nice and all, but it hardly makes up for the button's grave shortcomings when it comes to inclusion.

This time, let's map those shortcomings to WCAG's success criteria to establish where there are failings. First, under *perceivable*, the button fails *1.1.1 Non-text Content*[71] because it has no alternative text to supplement the image of text in the background. Second, under *operable*, it fails *2.1.1 Keyboard*[72] because the button uses an unfocusable <`div`> element, which — even if you made it to be focusable with `tabindex="0"` — would still need some JavaScript to trigger click events on pressing the *Enter* and *Space* keys.

Finally, under *robust*, the button fails *4.1.2 Name, Role, Value*[73] because there's nothing in the markup to communicate in assistive technologies that the button is, indeed, a button.

71 https://www.w3.org/TR/WCAG20/#text-equiv
72 https://www.w3.org/TR/WCAG20/#keyboard-operation
73 https://www.w3.org/TR/WCAG20/#ensure-compat

FIXING THE BUTTON

Now imagine an accessibility consultant has reported all of these failures in a spreadsheet under columns organized by principle and success criterion. This spreadsheet is then handed off to their client's project manager who sets about writing bug tickets for the developers.

One developer is randomly assigned a ticket labeled "Upvote Button Missing Label." They open that ticket and, like any self-respecting developer, endeavor to fix and close it. Not especially familiar with accessible naming techniques, they do a bit of casual research and land on some possible solutions. Supplying a text node is out because this would be visible over the top of the background image's text. In the end, they opt for `aria-label`.

```
<div class="upvote" data-action="upvote" aria-label="upvote">
</div>
```

Unknown to this developer, another developer is assigned "Upvote Button Not Keyboard-Accessible." The ticket comes with some guidance from the accessibility consultant about how to make the <`div`> keyboard-accessible using `tabindex` and some JavaScript to trigger click events with the keyboard. The project's JavaScript swells slightly and the markup ends up like this:

```
<div class="upvote" data-action="upvote" aria-label="upvote"
tabindex="0"></div>
```

Meanwhile, a third developer is dutifully studying "Upvote Button Has Improper Role" and — seeing how much work has gone into this `<div>` already — tops it off with an explicit ARIA role of "button".

```
<div class="upvote" data-action="upvote" aria-label="upvote"
tabindex="0" role="button"></div>
```

A BETTER WAY

The problem with this contrived and fragmented way of evaluating interfaces is that nobody has the chance to point out and avoid what are simply bad *ideas*, like using a `<div>` as a button control. The way failures are reported and developers are tasked means you always end up just patching what's already there. What you end up with is frequently bloated and less robust.

In reporting by pattern instead, the consultant has the opportunity to recommend alternative approaches and techniques. In this case, the various failures organized under the *upvote button pattern* would surely lead them to recommend the use of a standard `<button>` element in its place.

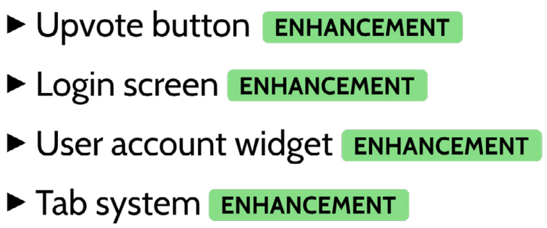
```
<button data-action="upvote" aria-label="upvote"></button>
```

To complete the redesigned pattern they might recommend the use of an SVG to represent the icon. As I cover in “A Menu Button,” SVG benefits from being scalable, small in file size and — unlike icon fonts — not breakable by user font preferences.

```
<button data-action="upvote" aria-label="upvote">
  <svg>
    <use xlink:href="#upvote"></use>
  </svg>
</button>
```

To a developer used to working in terms of modules and components, this represents a complete, integrated solution to a problem. The accessibility consultant (reporter) has met them on their home turf and helped them to improve the interface in an apprehensible and tangible way.

But that’s not all. Because the inclusive upvote button replaces the previous implementation one-for-one, the consultant has also helped the organization improve its pattern library. The next time an upvote button (or similar) is needed, the developer has an exemplar at the ready. This could apply to current or future projects.


▶ Upvote button ENHANCEMENT
▶ Login screen ENHANCEMENT
▶ User account widget ENHANCEMENT
▶ Tab system ENHANCEMENT

Tickets organized by pattern can be easily reconciled with a pattern library.

As Joe Dolson writes in “I’m an accessibility consultant. Stop hiring me,”[74] remediating broken websites and apps is unpleasant work for both the consultant and the client. Much more effective and rewarding is training and education in how to *design* inclusive products from the outset. Remediating by pattern brings design thinking into the remediation process, helping the client to fix their current product while instilling the confidence to make inclusive design decisions in the future.

This is an effective way to make my role as a remediator obsolete and, like Joe, I welcome it.

74 http://smashed.by/stop-hiring-me

Custom Elements And Shadow DOM

This is probably as good a place as any to approach the issue of inclusion and web components. Since many organizations are beginning to embrace the web component specifications[75] to compose their patterns, we should look at what needs to be done differently or additionally in terms of inclusive design. Of the Shadow DOM, Custom Elements, HTML elements, and HTML templates specifications, only Custom Elements[76] and (to a lesser extent) Shadow DOM[77] directly pose potential problems. Only they affect the behavior of the interface as it is manipulated by the user.

The joy of custom elements is that you can tie custom style and behavior to an element of your own conception. You could, for example, write all the code for a toggle button and attach it to a `<toggle-button>` element: a neat package of functionality with a descriptive name.

```
//  Custom Elements v0 syntax
document.registerElement('toggle-button', {
  prototype: toggleButton
});

//  Custom Elements v1 syntax
customElements.define('toggle-button', toggleButton);
```

75 https://github.com/w3c/webcomponents
76 https://w3c.github.io/webcomponents/spec/custom/
77 https://w3c.github.io/webcomponents/spec/shadow/

(**Note**: For a comparison of *Custom Elements versions 0* and 1, see Shawn Allen's "All about HTML Custom Elements."[78])

That new element, although accepted as a legitimate element in browsers supporting `registerElement` or `customElements.define`, does not have the standard behaviors of a <`button`> element. To put it another way, it is not an instance of the `HTMLButtonElement`[79] prototype. In fact, it is merely an instance of `HTMLElement`,[80] much like a <`div`>.

If you've been following along, you'll know what that means: it's not focusable or actionable, and doesn't have the `button` role that announces it as a button in screen readers — just like the remediated <`div`> button I was talking about earlier in the chapter.

In the case of our custom element, these features can be added much more neatly and robustly. Using Custom Elements v0, we add them in the `createdCallback` lifecycle callback:[81]

78 https://github.com/shawnbot/custom-elements
79 http://smashed.by/htmlbtn
80 http://smashed.by/stop-hiring-me
81 http://smashed.by/lifecycle

```
toggleButton.createdCallback = function() {
    //  accept keyboard focus
    this.setAttribute('tabindex', '0');
    //  make this element appear as a button in the
accessibility tree
    this.setAttribute('role', 'button');
    //  dispatch clicks events with keyboard
    this.addEventListener('keydown', function(event) {
        if (event.keyCode === 13 || event.keyCode === 32) {
            var click = new MouseEvent('click', {
                'view': this.ownerDocument.defaultView,
                'bubbles': true,
                'cancelable': true,
            });
            this.dispatchEvent(click);
        }
    });
};
```

Since our toggle button will initialize in the off (unpressed) state, we should also add `aria-pressed="false":`

```
this.setAttribute('aria-pressed', 'false');
```

The advantage of adding these attributes and behaviors as part of the element's definition is that the interface remains clean to authors — they only have to instantiate <`toggle-button`>. All we need to do now is handle that `click` event to switch the state. First we add the listener by placing this inside the `createdCallback` too:

```
this.addEventListener('click', function() {
  this.toggle();
});
```

Then we need to create the actual `toggle()` method. With the following in place, the state will change either on `click` or by scripting `toggleInstance.toggle()` (where `toggleInstance` is an instance of <`toggle-button`>).

```
this.toggle = function() {
    var isPressed = this.getAttribute('aria-pressed') ===
'true';
    this.setAttribute('aria-pressed', String(!isPressed));
};
```

In terms of semantics and behavior, our <`toggle-button`> now has all the accessibility provisions of <`button`>, but also the toggle functionality which makes it deserving of its own element definition. In the tradition of <`button`> we have created an element which has accessibility *built in*, meaning it can simply be deployed to an interface without having to consider keyboard and screen reader inclusion as separate concerns. Here's a demo of this little toggle button implementation, using Custom Elements v0.[82]

An easier way to get the basic functionality of a <`button`> before extending it is to literally *extend* the

82 http://codepen.io/heydon/pen/ZOqwqQ

`HTMLButtonElement` prototype. Using Custom Elements v1 with ES2015/ES6 syntax, we can write what follows. Note that in Custom Elements v1, `createdCallback` is not necessary:

```
class ToggleButton extends HTMLButtonElement {
  constructor() {
    super();

    this.setAttribute('aria-pressed', 'false');
    this.toggle = () => {
      var isPressed = this.getAttribute('aria-pressed') ===
'true';
      this.setAttribute('aria-pressed', String(!isPressed));
    };

    this.addEventListener("click", () => {
      this.toggle();
    });
  }
}

customElements.define('toggle-button', ToggleButton, {
extends: 'button' });
```

As you can see, now that I'm piggybacking the `HTMLButtonElement`, I have been able to exclude the `tabindex` and `role` attributes, as well as the ugly `keydown` event dispatching script. We are doing as we set out to do from the very beginning: using what's already at our disposal. It's more reliable, efficient, and just plain easier.

SHADOW BOUNDARY ISSUES

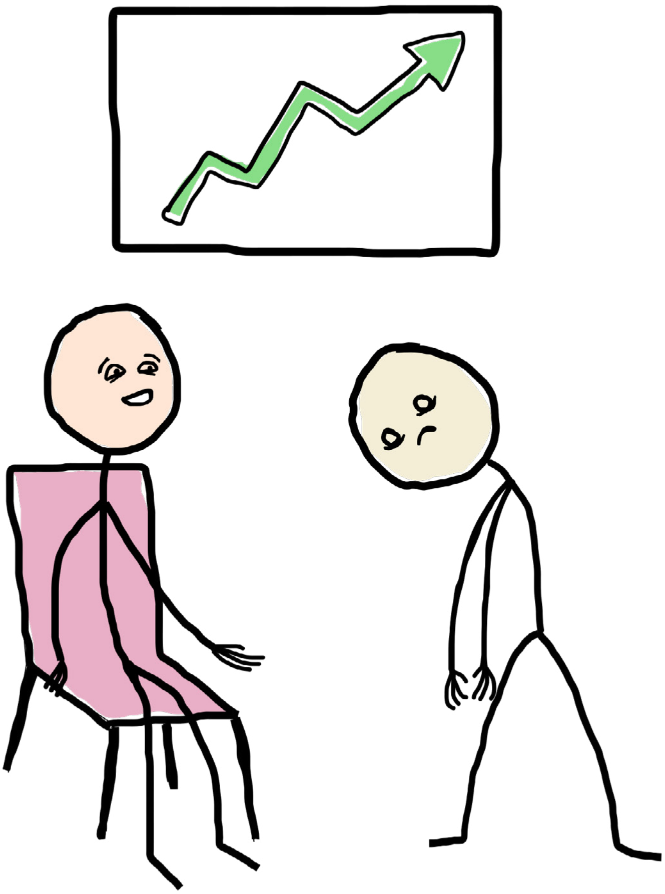
For the most part, creating inclusive web components is just creating accessible, semantic HTML as usual but inside a new API. Two notable differences I'm aware of are the effect the shadow boundary (the boundary between the DOM and Shadow DOM) has on ARIA relationship attributes,[83] and positive `tabindex` values.

As Steve Faulkner writes in "Some stuff that doesn't work between the DOM and Shadow DOM,"[84] the shadow boundary effectively severs relationships built using attributes like `aria-labelledby`, `aria-describedby` and `aria-controls`, where one element in the relationship is the other side of a shadow boundary.

There are common misconceptions that all custom elements have shadow DOM included, and that you can only use Shadow DOM with Custom Elements. No: they're autonomous technologies. The toggle button example does not use Shadow DOM, so could still associate itself with an element of ID `#toggleTarget` using `aria-controls="toggleTarget"`. Nonetheless, if it did employ Shadow DOM, this would become impossible.

83 http://smashed.by/ariarel
84 http://smashed.by/shadowdom

As for `tabindex`, using a positive integer like `tabindex="4"` will refer to the fourth in focus order within the shadow DOM and *not* the fourth in focus order within the parent document. This may have some useful applications, but use of explicit `tabindex` ordering is not advised in any case. Focus order should follow source order for logical keyboard operation. This is most easily achieved simply by using implicitly focusable elements and — where necessary — elements with `tabindex="0"`. Both of which are placed in default focus order according to source order, unaffected by Shadow DOM subtrees.



“Well done! It’s the most successful app of all time.”
“Yeah… but it’s not *pixel perfect*.”

Navigation Regions

THE LAST COUPLE OF PATTERNS dealt a lot with content, how to manage it and write it. But not all web page content is really content, as such. Navigation landmarks, for example, help your audience to traverse your web pages and the content they offer. In other words, they're a bit *meta*: they're content that helps with content.

Navigation regions are a tool with a long history, making them a convention you can rely on for inclusive page design. In fact, even the first web page ever created[85] features a rudimentary navigation region consisting of a set of links within a definition list (`<dl>`). Let's take some time to examine the design of navigation patterns as applied to site-wide, and page-specific table of contents schemas. This will incorporate writing structured markup, clear labels and robust CSS. How to inclusively indicate the current page — so that everyone knows where they are! — will also be covered.

The Navigation Landmark

Discrete areas of web pages are variously referred to as *regions*, *blocks* or *modules* and denote the visual boxes that demarcate parts of that page. The WAI-ARIA specification offers a handful of landmark roles[86] such as `role="main"`,

85 http://info.cern.ch/hypertext/WWW/TheProject.html
86 http://smashed.by/landmark-roles

which define semantics and behaviors to make common regions accessible in assistive technologies.

The `<main>` element and `role="main"` WAI-ARIA role map to each other, offering the same semantics and behaviors. You can use either but `<main>` is terser, as the following example attests.

```
<div role="main" id="main">
  &hellip;
</div>

<!-- or... -->

<main id="main">
  &hellip;
</main>
```

For the purposes of inclusion, our navigation regions should also be navigation *landmarks*. Navigation landmarks, denoted either by the `role="navigation"` WAI-ARIA role or the `<nav>` element, differ in two key ways to main landmarks:

- You can have more than one navigation landmark per page.
- The content inside navigation landmarks that appear on multiple pages (such as a common site navigation block) should be consistent between those pages.

Site-Wide Navigation

Let's deal with the most common type of navigation landmark first, establishing some of the features common to all navigation landmarks while we're at it.

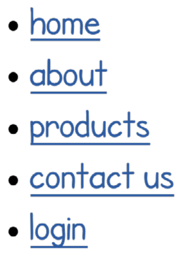
For the purposes of accessible UX, progressive enhancement and backwards compatibility, landmark regions should contain an unordered list of links. For *site* navigation, these links would pertain to the different pages available in your site or application.

```
<nav>
  <ul>
    <li><a href="/">home</a></li>
    <li><a href="/about">about</a></li>
    <li><a href="/products">products</a></li>
    <li><a href="/contact">contact us</a></li>
    <li><a href="/login">login</a></li>
  </ul>
</nav>
```

By using an unordered list, we evoke the rudimentary navigation schemas possible in HTML4 and XHTML and build on a recognized convention. The list groups the links, conveying accessibly that they have a common and shared purpose. Not only that, but when CSS fails, the familiar form of a bulleted list containing blue, underlined text acts as a visual signifier that this content has a navigational purpose.

Just by using a list, we're being inclusive of older users and devices, assistive technology users and those who are experiencing a CSS failure.

Wrapping the list in a landmark offers additional semantics and behaviors. If I was to enter the landmark while running a screen reader and focus the first link, I would hear a lot of useful information. Firstly, I'd hear "Navigation landmark," followed by "List, one of five" and finally "Link, home." By then, I'd know that I'm in a navigation landmark, that it contains five links in total, and that I can follow the first link immediately if I wish.



- home
- about
- products
- contact us
- login

We see interfaces with broken or missing CSS frequently enough to recognize this as a navigation block.

That's not all, though. The <`nav`> (or `role="navigation"`) landmark joins <`main`> in being discoverable when traversing the page by region — for instance by using the *D* key in the NVDA screen reader. Also, NVDA, JAWS and VoiceOver all provide a dialog listing landmarks as a menu, allowing you to switch between them directly.

- Banner
- Navigation
- Main
- Footer

OK, so a navigation menu that lists a navigation menu is *really* meta. Nonetheless, it's helpful to have your screen reader create an automated index of regions for the page in this way. Wading through content with only the hope of encountering an identifiable region requires faith that few users have to spare.

APPEARANCE AND PLACEMENT

The human brain uses patterns called *schemata*[87] to understand sense data. Schemata constitute prior experience against which current experience is evaluated.

In programming terms, schemata are a kind of cache for *understanding*. So long as a familiar thing is being experienced, little more effort is required to understand it.

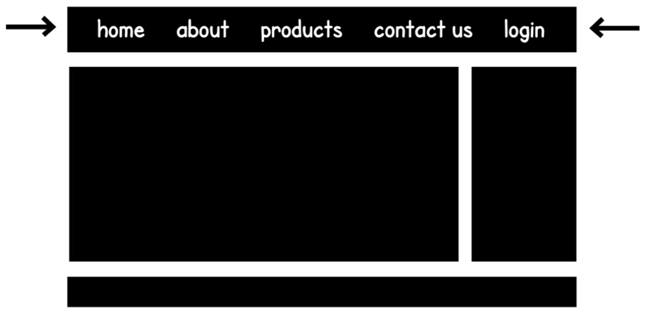
By the same token, if something genuinely new is experienced, there's nothing in the cache to be relied on and the sense data must be evaluated in full.

In design, by making things that behave in a certain way *appear* as expected, we help our users make the most of their cognitive cache. In other words, we don't make them think.[88] This is a well-known usability principle, but it's worth

87 https://en.wikipedia.org/wiki/Schema_%28psychology%29
88 http://smashed.by/dontmakemethink

restating — especially in the context of inclusive design, where we're trying to cater for extremes of cognitive impairment.

In the case of the site navigation landmark, you would do well to give it a familiar form consisting of adjacent, enlarged links. Position it in the header of each page, above the main content and preferably before anything else.



I'm not suggesting your navigation bar has to use a white, cursive font on a black background! It's the familiar positioning and shape that counts.

CSS POSITIONING

The purpose of placing site navigation at the top of the page is not only for cognition, but interaction. As we've already established, keyboard users navigate pages one interactive element at a time, in order. Putting page navigation at the top means they don't have to traverse one page to get to another.

In practice, so long as source order matches reading order this is something which takes care of itself. Unfortunately, we have a habit of overengineering and overcomplicating things.

While attending the Future of Web Design conference in 2015, I sat in on an accessibility clinic run by Léonie Watson.[89] One of the attendees had come to Léonie with a problem: they couldn't work out why the site navigation in their prototype wasn't focusable. It turns out it was; it was just at the bottom of the source so it took thirty or forty presses of the *Tab* key to reach it and finally invoke the focus style.

```
   <nav>
         <!-- the last interactive elements on the page -->
   </nav>
</body>
```

With CSS position values of `absolute` or `fixed`, one can visually position an element anywhere in the viewport, regardless of the source order. Not only does this create a contrary user experience for keyboard users, but positioned elements don't reflow with the rest of the document. This is liable to create layout issues when the viewport or its content is resized. Elements will slip behind other elements or extend outside of the viewport and become obscured.

89 https://twitter.com/LeonieWatson

In general, avoid positioning in all but the rarest of cases. When relying on positioning to create modal dialogs, you should exercise extreme caution, testing against a range of viewport dimensions and magnification settings.

"YOU ARE HERE"

Half of knowing where you want to go is knowing where you are already. This is why department stores and shopping centres put "You are here" signs on their floor maps.

A friendly navigation schema does something similar by highlighting the current page: the page the user has open. Identifying the current page link in your `<nav>` region makes your site more usable. If you go about it the right way, this increase in usability can be inclusive of different types of setup and user.

DON'T DIFFERENTIATE BY COLOR ALONE

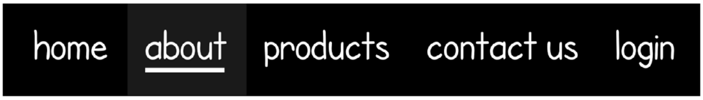
Color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.

— *WCAG2.0 1.4.1 Use of Color*[90]

90 https://www.w3.org/TR/WCAG20/#visual-audio-contrast

Typically, the current page link is distinguished by color, which means users with certain varieties of color blindness may not be able to differentiate this link from the others. It's best to provide an additional adornment, such as a text underline, as an alternative means of differentiation.

```
a.current-page {
  display: inline-block;
  padding: 0.5rem;
  text-decoration: underline;
  background: $highlight-color;
}
```

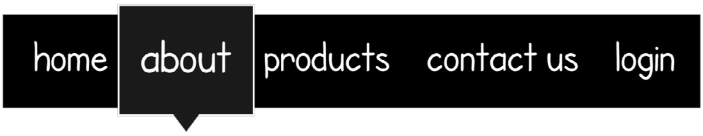


A navigation bar with the active "about" link underlined.

As illustrated, a clear differentiation by hue can be a faint differentiation by shade to some color blind users. Mac users who want to see what their web pages look like without color can go to *System Preferences* → *Accessibility* → *Display* and check *Use grayscale*.

A more ambitious solution might be to use some scaling and a little pointer, provided in pseudo-content.

```
a.current-page {
   display: inline-block;
   padding: 0.5rem;
   background: $highlight-color;
   transform: scale(1.2);
}

a.current-page::after {
   content: '';
   position: absolute;
   left: 0;
   right: 0;
   bottom: -0.25em;
   height: 0.25rem;
   background: url('images/pointer.svg') center no-repeat;
   background-size: auto 100%;
}
```



The Nav bar with an active "about" inside a white bordered box.

Any method is acceptable so long as the link text remains unobscured and readable, and the hue is not the only means of differentiation.

SCREEN READER AND KEYBOARD SUPPORT

There's no standard way to identify current page links non-visually as yet. Instead, we'll have to employ a workaround. In "The Accessible Current Page Link Conundrum,"[91] I explored a few possibilities and a productive discussion took place in the comments. The following is an exposition of the better ideas that emerged there.

The first thing you need to do is provide some text to be read out when the screen reader operator focuses the link. It's important that this text does *not* replace the existing link text and is, instead, appended or prepended to it. One approach is to insert a visually hidden, screen reader announceable `<span>` containing a cipher, like "Current page":

```
<nav>
   <ul>
      <li><a href="/">home</a></li>
      <li><a href="/about"><span class="visually-
hidden">Current page</span> about</a></li>
      <li><a href="/products">products</a></li>
      <li><a href="/contact">contact us</a></li>
      <li><a href="/login">login</a></li>
   </ul>
</nav>
```

91 http://smashed.by/conundrum

(**Note**: The CSS for the `visually-hidden` class is set out in "A Blog Post.")

Whether it is to say "Current page," "This page" or "You are here" is somewhat open to debate, but it should be fairly succinct. I've prepended the `<span>` so the screen reader says "Current page about" rather than "About current page." The latter is ambiguous: does this link refer to a description *of* the current page?

Another approach, which goes some way to mitigate ambiguity, is to use a proxy element and `aria-describedby`.[92] The `aria-describedby` attribute imports and appends a description which is read after a pause. In the following example, focusing on the current (about page) link would trigger the announcement, "About [pause] Current page."

```
<nav>
   <ul>
      <li><a href="/">home</a></li>
      <li><a href="/about" aria-describedby="current">about</
a></li>
      <li><a href="/products">products</a></li>
      <li><a href="/contact">contact us</a></li>
      <li><a href="/login">login</a></li>
   </ul>
   <div hidden id="current">Current page</div>
</nav>
```

92 http://w3c.github.io/aria/aria/aria.html#aria-describedby

A few notes:

- `aria-describedby` is a *relationship attribute*: it associates the described element with the descriptive element via `id`. The description is the descriptive element's text node.
- The HTML5 `hidden` attribute is used to hide the description both visually and from screen readers. It works like `display: none;` but is syntactically neater. Despite being hidden, because it *has a relationship* to the link, the link still has access to "Current page" and it will be announced on focusing the link. (Note that Internet Explorer does not support `hidden` at the time of writing, but you can force it to by adding `[hidden] { display: none; }` to your style sheet.
- Moving the `aria-describedby="current"` attribute from one element to another between pages is easy enough, and it can be used as a styling hook just as a class might be: `[aria-describedby="current"]`.

FROM REDUNDANT LINK TO SKIP LINK

Some unfinished business: on the about page, the current page link, now accessibly identified, points to the same about page. In practice, this means that clicking the link will reload the page, dumping the keyboard user's focus at the very top, on the document itself. For screen reader users, this will prompt the repeated announcement of document-level information such as the `<title>` and a set of meta information about the page such as how many links it contains.

The redundancy of the self-referential link has prompted some designers to remove it altogether. On the about page, that would leave us with:

```html
<nav>
  <ul>
    <li><a href="/">home</a></li>
    <li><a href="/products">products</a></li>
    <li><a href="/contact">contact us</a></li>
    <li><a href="/login">login</a></li>
  </ul>
</nav>
```

This produces an unsatisfactory wayfinding experience. The current page link is for context, remember; the "You are here" marker. Without it, you force the user into a rather absurd spot-the-difference game.

As a remedy, I've heard it suggested to remove the current page link's `href`, or to replace the `<a>` entirely with an uninteractive `<span>` element. These implementations are serviceable for sighted users, but screen reader users often traverse navigation landmarks from link to link, using their *Tab* key. If the current page indicator is not focusable, it would be skipped over as if it didn't exist.

My favorite solution is to make the current page link point to the main region of the page. This way, instead of just reloading the page, it takes you past the navigation landmark to the content, like a skip link.

```
<nav>
  <ul>
    <li><a href="/">home</a></li>
    <li><a href="#main">about</a></li>
    <li><a href="/products">products</a></li>
    <li><a href="/contact">contact us</a></li>
    <li><a href="/login">login</a></li>
  </ul>
</nav>
```

Not only is the link easy to style via the `[href="#main"]` attribute selector, but its behavior is communicated to assistive technologies: on focusing the home link, "Home, link" will be announced. Then, on focusing the about link, the user should hear "About, same page link." Since "same page" is vocalized already, the hidden <`span`> with "Current page" is arguably obsolete. We've married behavior and semantics just by altering the `href` value.

Credit should go to Daniel Göransson[93] for suggesting this solution. Sometimes discussions in article comment sections can be surprisingly productive!

Be wary of using this technique in single-page applications. When you are emulating the behavior of loading whole new pages, the announcement of "same page link" exposes a white lie and would make the experience confusing. For single-page applications, the `aria-describedby` approach would therefore be more suitable.

REMOVING REDUNDANCY

It's common practice to place your navigation landmark within the page's *banner* landmark, which also features your company or project logo. Here's how that might be devised. Note that the <`header`> element has the explicit ARIA `banner` role. This is because multiple <`header`> elements can be used on one page, but only one should be the singular banner landmark. Though some browsers are clever enough to calculate which <`header`> is the right one, this helps others along.

93 https://twitter.com/danielgoransson

```
<header role="banner">
  <a href="/home">
    <img src="images/logo.svg" alt="My Project home">
  </a>
  <nav>
    <ul>
      <li><a href="#main">home</a></li>
      <li><a href="/about">about</a></li>
      <li><a href="/products">products</a></li>
      <li><a href="/contact">contact us</a></li>
      <li><a href="/login">login</a></li>
    </ul>
  </nav>
</header>
```

Since usability convention dictates that the logo image should double as a link to the home page, the logo has a navigational role: the same navigational role as the first of the navigation landmark's links. Not only does this create needless and potentially confounding redundancy, but we've committed the cardinal usability sin of giving two things that *appear* differently the same functionality.

One resolution might be to remove the home link from the navigation landmark. However, navigation landmarks need to be complete and autonomous because they can be accessed directly, using shortcuts. Instead, we can turn the navigation block's home link into the logo, compacting the content by removing the now unnecessary `role="banner"` landmark.

```
<nav>
  <ul>
    <li>
      <a href="#main">
        <img src="images/logo.svg" alt="My Project home">
      </a>
    </li>
    <li><a href="/about">about</a></li>
    <li><a href="/products">products</a></li>
    <li><a href="/contact">contact us</a></li>
    <li><a href="/login">login</a></li>
  </ul>
</nav>
```

(**Note:** I am still using the same-page link technique here. Imagining that we are currently *on* the home page, the logo's link goes to `#main`.)

Where an image is the content of a link, the alternative text (`alt` value) should describe the purpose of the link, not the nature of the image. In other words, "My Project logo" would not suffice. "My Project home" or just "Home" are both appropriate here.

By creating a canonical link to the homepage, we've not only made navigation simpler to understand but foregone a *Tab* stop and diminished the verbosity of screen reader output. It's important that screen reader users get all the information they need, but redundant information is noise, something which is more difficult to ignore than visual artifacts.

In terms of responsive design, it’s possible to target the logo using `:first-child` and, for example, place it on its own row within smaller viewports:

```
@media (max-width: 20em) {
   nav li:first-child {
      display: block;
      text-align: center;
   }
}
```

Tables Of Contents

I like pages with links at the top of the page. It’s really helpful on long pages with a lot of sections. I can figure out what’s on the page without a lot of work.”

— Lea, a persona from “A Web For Everyone”[94]

Resources like Wikipedia which trade in typically very lengthy and detailed articles, have long been incorporating tables of content because they provide two related benefits:

- Summarizing long-form content.
- Providing navigation to specific sections of long-form content.

94 http://rosenfeldmedia.com/books/a-web-for-everyone/

I'm fond of tables of contents for one reason most of all: they're not drop-down/pull-down/pop-up submenus. I mean, these don't even have a name anyone can agree on! *Drop-down submenus* (let's call them that) attempt to solve the same problem of grouping links to related content, but are flawed in a number of ways.

First, they hide navigational content from view within a precarious, interaction-activated menu. Second, they encourage information architects to spawn numerous, independent pages of content, without the clear hierarchy of grouping related sections under a main heading on a shared page. Though technically they can be made accessible, it's not a trivial task to support mouse, keyboard and touch interaction simultaneously without diminishing the quality of any one mode. The submenus they disclose need to be positioned absolutely, which invites layout issues across different viewports, and they're cumbersome when it comes to graceful degradation in the absence of JavaScript, CSS or both.

Sometimes it's better not to make something just-about-inclusive, but to think about the root problem and devise a completely different solution — a simpler and more robust one. Tables of content are the antidote to drop-down submenus. Drop-down submenus will not, therefore, have a pattern of their own in this book. For further reading, consult Nielsen Norman Group's "Drop-Down Menus: Use Sparingly."[95]

95 http://smashed.by/drop-down

BASIC STRUCTURE

Wikipedia provides the <h2> "Contents" to locate the table of contents in assistive technology. Since tables of contents that link to page sections are de facto navigation landmarks, we can do one better than Wikipedia and include `role="navigation"` on the parent element. Each link points to a fragment identifier[96] that corresponds with a section of the page.

Contents
Our history
What we do
News
Endorsements

Web page tables of contents usually follow the Wikipedia model: a basic bulleted list of links inside a box.

```
<div class="toc" role="navigation">
   <h2>Contents</h2>
   <ul>
      <li><a href="#history">Our history</a></li>
      <li><a href="#services">What we do</a></li>
      <li><a href="#endorsements">News</a></li>
      <li><a href="#endorsements">Endorsements</a></li>
   </ul>
</div>
```

96 https://en.wikipedia.org/wiki/Fragment_identifier

SEQUENTIAL FOCUS NAVIGATION

The links within a table of contents point to element `id`s representing targets within the page, such as `#services`, which update the document's URL with the corresponding hash cipher.

```
<h2 id="services">Services</h2>
```

Despite these targets (usually `<section>`s or headings) not typically being focusable elements, browsers employ *sequential focus navigation* whereby the first focusable element within or after the target is made the *next* focusable element in sequence. The upshot is that users navigating by keyboard can follow a link in a table of contents and be transported to that section safe in the knowledge that keyboard functionality will be localized to the section.

MAXIMIZING SUPPORT

Until recently, a bug in Chrome[97] meant that sequential focus navigation did not work, but that bug has (finally!) been fixed. To make sequential focus navigation work in Internet Explorer, you have to overcome a longstanding `hasLayout` bug. Fortunately, this is possible by simply adding `tabindex="-1"` to the element carrying the target `id`.

97 http://smashed.by/focus-target

```
<h2 id="services" tabindex="-1">Services</h2>
```

LINKJACKING

Some implementations of in-page linking are enhanced by JavaScript to provide a smooth scrolling effect as the targeted section comes into view. For this to work correctly, the standard browser behavior of jumping instantaneously to the target needs to be suppressed. The behavior of *linking* is effectively replaced with *scrolling* and sequential focus navigation no longer takes place. Keyboard users become stranded.

Now that standard behavior has been usurped, the usurper needs to emulate it for us. In other words, we need to use JavaScript to fix the JavaScript. As is often the case in making keyboard-inclusive JavaScript interfaces, explicit focus management must be employed to move the keyboard user to the intended location. A typical implementation is achieved by animating `scrollTop` using jQuery. Fortunately, jQuery's `animate` method provides a callback, meaning focus can be applied to the target element after the animation has finished.

```
function isSameResource(urlOne, urlTwo) {
   var fragmentPattern = /#._$/;
   var resourceOne = urlOne.replace(fragmentPattern, '');
   var resourceTwo = urlTwo.replace(fragmentPattern, '');
   return resourceOne === resourceTwo;
}

function getFragmentTarget(id) {
   if (id.slice(0, 1) === '#') {
      id = id.slice(1);
   }

   // we’re looking for <div id="{id}"> or <a name="{id}">
   return document.getElementById(id)
      || document.querySelector('a[name="' + id + '"]');
}

// handle activation of all fragment links on current page
$(document.body).on('click', 'a[href*="#"]:not([href="#"])',
function(event) {
   if (event.isDefaultPrevented()) {
    // some other event handler might have already
handled this event
      return;
   }

   if (!isSameResource(location.href, this.href)) {
      return;
   }

   var target = getFragmentTarget(this.hash);
   if (!target) {
      return;
   }

    // prevent browser from jumping to the fragment
    // because we want to scroll it into view first
   event.preventDefault();
   // smooth scroll document to target element over 1 second,
   // not using Element.scrollIntoView() because not widely
supported yet
  $('html, body').animate(
     { scrollTop: $(target).offset().top },
     1000, function() {
   // now that we've scrolled the target into view,
   // let the browser do its regular thing and update the URL
        window.location.hash = target.id || target.name;
     }
  );
});
```

Note the line starting `window.location.hash` which updates the URL after focus with the hash fragment. This reinstates standard behavior too, making sure the user can still bookmark and share the subsection. This ability to record and retrieve subsections of documents is an integral part of the web experience, and one some users would miss if absent.

LABELING NAVIGATION LANDMARKS

We began by formulating a site-wide navigation landmark. Now we've spawned an in-page navigation landmark, introducing a second region to our pages. While some landmarks like `banner` and `main` can only occur once per page, it's quite legitimate to use multiple navigation landmarks if the content merits them.

There's a world of difference, however, between technical compliance and ensuring a good user experience. To make the most of these two complementary navigation tools, the user needs to know which is which. Visually speaking this is trivial, since their style and placement demarcate them. The site navigation should appear above the `<main>` content and the table of contents within it (preferably directly below the document's principal `<h1>` heading, as it would in a Microsoft Word document).

Similarly, when browsing the page from top to bottom, the screen reader user would encounter the site navigation first, which is some indication of role and identity. A heading, such as the "Contents" `<h2>` in the previous example, helps if the user is navigating by heading shortcut.

However, as you may remember from "A Blog Post," screen readers aggregate headings, links and landmarks into lists, providing tables of contents of their own. So that headings and links make sense decontextualized in such a fashion, you need to write autonomous, self-describing labels.

Unlike headings and links, landmarks do not constitute their own labels; they are simply identified by their role.

Accordingly, two navigation landmarks would be listed as:

- Navigation
- Navigation

Never fear: thanks to WAI-ARIA's global `aria-labelledby` relationship attribute, we can provide an auxiliary label via the existing heading element. In the following example, the text node "Contents" (of the `<h2>`) becomes the label of the region using its `id` as the `aria-labelledby` value.

```
<nav class="toc" aria-labelledby="contents-heading">
  <h2 id="contents-heading">Contents</h2>
  <ul>
    <li><a href="#history">Our history</a></li>
    <li><a href="#services">The services we offer</a></li>
    <li><a href="#endorsements">Visit our office</a></li>
    <li><a href="#endorsements">Endorsements</a></li>
  </ul>
</nav>
```

Though a heading is recommended to support screen reader heading navigation, and to add lexical clarity for users of all kinds, it's not obligatory. In which case, the `aria-label` attribute can be attached directly to the landmark instead.

```
<nav class="toc" aria-label="contents">
  <ul>
    <li><a href="#history">Our history</a></li>
    <li><a href="#services">The services we offer</a></li>
    <li><a href="#endorsements">Visit our office</a></li>
    <li><a href="#endorsements">Endorsements</a></li>
  </ul>
</nav>
```

In either implementation, there are two upshots:

1. When a screen reader user focuses a link inside the navigation landmark, the label is added to the contextual information announced. By focusing the "Our history" link in the previous example, "*contents* navigation landmark, list, one of four items, our history, link" (or similar) would be announced.

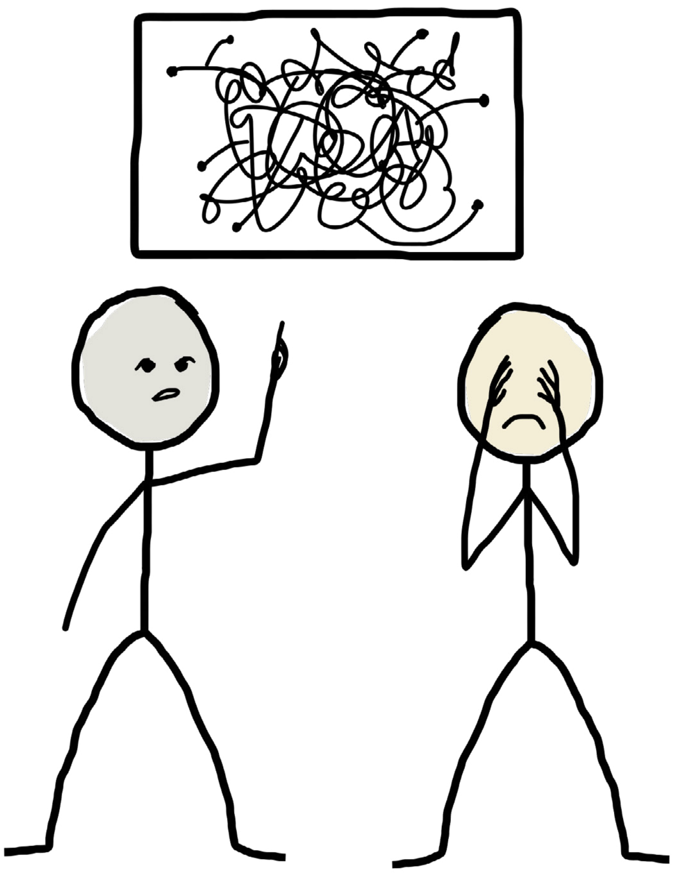
2. In the screen readers VoiceOver, NVDA and JAWS, the navigation landmark will be distinctly labeled as "Contents navigation" in the landmark elements list.

Summary

For this pattern, we progressively enhanced one of HTML's primitives, lists, to create the inclusive means to navigate within and between web pages. Along the way, I covered design provisions transferable to many other patterns, including logical source order and the virtue of eliminating redundancy. When JavaScript was introduced to enhance scrolling behavior, we ensured it would both remain functional to keyboard users and degrade gracefully in the absence of the script.

THINGS TO AVOID

- Unconventionally designed or difficult to discover menu systems.
- Relying on color alone to indicate the current page or section.
- Hijacking link behavior without considering focus management.
- Foregoing unique labels where more than one `<nav>` is present.



“This is the current database structure.
Our designers may be able to code, but
our coders clearly can’t design.”

A Menu Button

Sometimes we hide menus away (like the navigation regions of the last chapter), putting the content center stage. Revealing the menu should be at the user's discretion, so we provide a button. Were menu buttons straightforward and uncontroversial, I could have folded this pattern into the last one, but, alas, there are a number of challenges and concerns to address: the method of rendering the menu icon; how to make it easily intelligible; how to label it accessibly; the communication of open and closed menu states; the ergonomics of touch operation. All these things have to be taken into account.

First of all, though, it needs to be stated that — like drop-down menus — if you don't need a menu button, don't involve one. As a rule of thumb, if the menu has fewer than five items, just lay them out; make them available to the user at all times. In desktop viewports, there's rarely *any* reason to hide a navigation menu away, regardless of the number of items it contains. Hiding functionality away from users and requiring them to perform an additional action to reveal that functionality is always a last resort.

If only a few menu items are needed, the menu button is a solution to a problem that does not exist. Louie A points out in "Why and How to Avoid Hamburger Menus"[98] that foregoing

98 http://smashed.by/avoid-hamburger

the button is often a question of information architecture. Nonetheless, not all applications are reducible to only a handful of views or actions and, since menu buttons are close to ubiquitous, we should formulate an inclusive implementation.

Appearance

It is well established that iconography can improve and accelerate comprehension for sighted users. To pick just one study, “Icons Improve Older and Younger Adults Comprehension of Medication Information” (PDF)[99] found that medication dosage and routine were more quickly understood when icons were used to represent the salient information. A non-textual mode of communication also crosses language barriers, helping to internationalize an interface, and assists those with poor literacy.

happy
blij
高兴
счастливый
glücklich
مبارک

A smiley face says “happy” to more people than any one language can manage.

99 http://smashed.by/iconimprove

Some icons are more intelligible than others and that's a question of their place within a sign system.[100] In the "Navigation Regions" pattern we acknowledged the ubiquity of lists being used to enumerate navigation options. Since the three-line (≡) hamburger icon or navicon is a co-opted list icon, it is meaningful based on our knowledge of lists and their relationship to navigation.

With that in mind, the icon should be fairly widely understood. This rather depends, though, on the menu that's being revealed actually having the appearance of a list. Making list items bear the same shape and background color as the horizontal strips in the icon itself makes the icon truly iconic (representative by physical approximation). As always, convention is a friend to inclusion, so extrapolating on the classic three-line symbol by adding lines or changing line orientation is to be avoided.

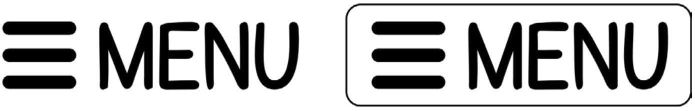
Leave creativity to the bad designers. This is not the place to do something different. If a convention exists, use it."

— Mark Boulton, "Icons, Symbols and a semiotic Web"[101]

100 https://en.wikipedia.org/wiki/Sign_system
101 http://smashed.by/iconsymbols

Not everyone will see navigation when looking at something as pictorially reductive as three horizontal lines, and James Foster's research[102] ratifies this assumption. In A/B testing he found that the icon accompanied by the text "menu" was better understood.

He also found that menu buttons with a button-like shape (thanks to a bordered outline) were more apprehensible. In general, all the buttons in your interface should *look* like buttons, otherwise they lose perceived affordance:[103] the appearance that they can be used. Perceived affordance is a cornerstone of cognitive accessibility for interaction design.



By drawing a perimeter line around the icon and text of the control it appears as a button.

By having the icon and the "menu" text present, we accelerate the comprehension of well-versed mobile interface users without alienating newcomers. The voice activation term "menu" is also made clear to users of products like Dragon NaturallySpeaking.[104]

102 http://exisweb.net/mobile-menu-abtest
103 http://www.jnd.org/dn.mss/affordances_and.html
104 http://smashed.by/voice

Rendering The Icon

You'd be forgiven for thinking you were spoiled for choice when it comes to rendering the icon: image tag; background image; Unicode character; icon font character; SVG — take your pick! Except, don't. Not all of these options are as robust and inclusive as you might suppose. Let's look at some of the pros and cons for each.

BACKGROUND IMAGE

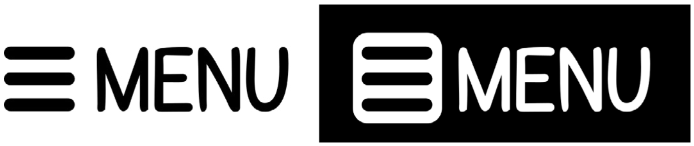
Now that background images can be scaled using the `background-size` property, their viability in responsive design has improved. However, as you may recall from the <`button`> example in the introduction, background images are eliminated when users switch into Windows high contrast mode.[105] If we provide the "menu" text (which will have its color inverted by high contrast mode to still be readable against an inverted background color) this isn't a deal breaker. But without the additional text, the button has no visual presence at all.

IMAGE

A *.png* image which defines three black lines separated by transparent spaces has a similar issue: when the background becomes black, you get three black lines on a black

105 http://webaim.org/blog/high-contrast/

background. You can, of course, provide a white, rather than transparent, background for your icon, but on a black background this becomes three black stripes in a white box, which is a slightly different proposition.



Encapsulated in white, on the high contrast mode's black background, the icon appears differently and seems separate from the text label of "menu."

A GLYPH FROM AN ICON FONT

Fortunately, icon fonts are text, so they *behave* like text. When high contrast mode inverts the color of the "menu" text, it will also invert the color of the icon. Another advantage is that icon fonts scale without degradation, much like SVG.

Icon fonts become problematic when users choose their own fonts for web pages, as described by Seren D in "Death To Icon Fonts."[106] It's important for inclusion to allow users to choose fonts which they are comfortable reading, especially if they experience difficulty with dyslexia. Your style

106 https://speakerdeck.com/ninjanails/death-to-icon-fonts

sheets should be sensitive to this preference. The problem comes when the user-defined font does not support the characters used for the icons, and leaves generic "Glyph not defined" boxes in their place.

The familiar rectangular box indicates a glyph that is not supported and constitutes a fallback.

As with any web font, an icon font is a resource which may be blocked. This will result in the generic-boxes-all-over-the-place effect. Opera Mini doesn't load web fonts as a matter of course. This is workable where there are fallback system fonts defined, but icon fonts tend to use esoteric Unicode points which have no equivalent in normal system fonts. Bruce Lawson of Opera has a great article on "Making websites that work well on Opera Mini."[107] Successful Opera Mini support is a good litmus test of high performance.

Zach Leatherman of Filament Group writes that sometimes operating systems *do* use Unicode's private use area,[108] but for their own purposes. In practice, this means our menu icon font could fail to load and fall back to displaying a cat's face. I imagine this might have a negative impact on cognition.

107 https://dev.opera.com/articles/making-sites-work-opera-mini/
108 https://www.filamentgroup.com/lab/bulletproof_icon_fonts.html

UNICODE

What if, instead of downloading an icon font, we used a standard Unicode symbol to represent the icon? In terms of performance and font stacking behavior, this is an improvement — and there is indeed an approximate Unicode symbol: U+2630.[109]

There are a couple of issues. The first is that not all devices support a Unicode set extensive enough to include this character, the "trigram for heaven." The second is that, unlike an icon font which is mapped to Unicode points in the private use area,[110] this character is more likely to be interpreted by assistive technology. Therefore, unless intervention is taken, "Trigram for heaven" could be announced in some screen readers. To either English or Chinese screen reader users, this would be rather confusing.

To silence readout, you'd have to place the icon in an element with `aria-hidden="true"` specified, which precludes the use of CSS pseudo-content for rendering it. CSS pseudo-content is, typically, announced by screen readers and there's no "Don't say this" property that's supported well (although work is underway).[111]

109 http://www.fileformat.info/info/unicode/char/2630/index.htm
110 https://en.wikipedia.org/wiki/PrivateUseAreas#PrivateUseAreas
111 https://www.w3.org/TR/css3-speech/#speaking-props-speak

```
<button>
   <span aria-hidden="true">☰</span>
   Menu
</button>
```

SVG SPRITES

SVG sprites are fast becoming the de facto solution for icon rendering — and with good reason. As Google's 305-byte logo implementation[112] attests, they can make very small assets. They are scalable by nature and can even change color in accordance with changes to font color.

SVG sprites work best cross-browser when they're embedded in the page. This also eliminates a separate HTTP request. The following should appear directly inside the `<body>`:

```
<svg style="display: none;">
   <symbol id="navicon" viewBox="0 0 20 20">
      <path d="m0-0v4h20v-4h-20zm0 8v4h20v-4h-20zm0 8v4h20v-
4h-20z"/>
   </symbol>
</svg>
```

This hidden `<svg>` is for reference and defines the menu icon as a `<symbol>` containing the path that forms the icon's shape. Note that it is hidden with an inline `display: none;`

112 http://smashed.by/googlelogo

style. If it was hidden using CSS, the SVG would be visible to users whose CSS was not loaded. In fact, since `<symbol>` elements are not themselves rendered, there would be a blank gap in your design.

The `<symbol>` can be used within our menu button by referencing its `id` with a `<use>` element.

```
<button>
  <svg><use xlink:href="#navicon"></use></svg>
  menu
</button>
```

In the CSS, we change the icons default 20×20px size (set in the referenced SVG's `viewBox` definition) to fit and scale along with the "menu" text:

```
button svg {
  width: 1em;
  height: 1em;
}
```

To make the icon adopt the color of the button element's font — both with high contrast mode on or off — we can use CSS's `currentColor` value and set it on the `<path>`'s `fill` property. SVG is clearly the most robust solution and already has very good support.[113]

113 http://caniuse.com/#feat=svg

```
<svg style="display: none;">
  <symbol id="navicon" viewBox="0 0 20 20">
    <path d="m0-0v4h20v-4h-20zm0 8v4h20v-4h-20zm0 8v4h20v-
4h-20z" fill="currentColor" />
  </symbol>
</svg>
```

Labeling

All interactive elements should have an accessible name so they can be interpreted and communicated in assistive technologies. This relates to WCAG's *4.1.2 criterion, Name, Role, Value.*[114]

The simplest labels are both visible and parsable, based on textual content such as our "menu" text node. As discussed, there are cognitive benefits of including "menu," not to mention the `<button>` would remain intelligible even if one of the less reliable icon rendering techniques fails.

However, there are circumstances in which you might provide the icon alone. If so, making sure the button is identified as "menu button" to screen reader users is paramount. The techniques expounded here are applicable to any kind of iconic control, such as a play or stop button in a media player. Many lessons in this book are transferable between patterns and to new patterns in this way.

114 http://smashed.by/rsv

THE VISUALLY HIDDEN <SPAN>

This method uses a CSS hack on a <span> to hide the "menu" text label without it becoming unavailable to screen readers. For that, we can harness our trusty .visibility-hidden class (set out in "A Blog Post").

Used within our SVG implementation, the code would look like this:

```
<button>
  <svg><use xlink:href="#navicon"></use></svg>
  <span class="visually-hidden">menu</span>
</button>
```

THE ARIA-LABEL ATTRIBUTE

Another auxiliary method for labeling the icon is to use an aria-label[115] attribute. This global property attaches alternative text to elements much like the alt attribute, but isn't just applicable to <img> tags. We can attach an aria-label directly to the <button> element:

```
<button aria-label="menu">
  <svg><use xlink:href="#navicon"></use></svg>
</button>
```

115 http://smashed.by/aria-label

One of the advantages of `aria-label` is that is overrides the text node of the element, if present. So, if I were to use the Unicode rendering method, I could replace the potential "Trigram for heaven" readout with "menu":

```
<button aria-label="menu">
  &#x2630;
</button>
```

Since ▶ is read as "black right-pointing triangle" and × as "times" (multiplication), the same `aria-label` method could fix the labeling of your play and close buttons, among others. Nonetheless, because of the described benefits, use SVG to render all your icons if you are able to.

I wrote an article on the UX of `aria-label`[116] for Dev.Opera.

Older Browsers

Support for SVG is pretty much universal,[117] with the stipulation that IE9–11 cannot reference external files using `xlink:href` — not a problem with this implementation. Internet Explorer preceding version 9 and other older browsers need a fallback for inline SVG. This is where you can use the <`switch`> and <`foreignObject`> elements:

116 https://dev.opera.com/articles/ux-accessibility-aria-label/
117 http://caniuse.com/#feat=svg

```
<button aria-label="menu">
   <svg>
      <switch>
         <use xlink:href="#navicon"></use>
         <foreignObject>
                 <img src="path/to/navicon.png" alt="" />
         </foreignObject>
      </switch>
   </svg>
</button>
```

In the example above, the SVG renders via the <`use`> element if supported, but *switches* to rendering the fallback PNG if not. Note the empty or `null` value of the `alt` attribute on the fallback image: this tells screen readers not to acknowledge the image. Since the label of "menu" is provided via `aria-label`, the `alt` is not needed here. Omitting the `alt` attribute altogether will mean some screen readers will announce the image's file name which is irrelevant and, frankly, irritating. Only `alt=""` is reliable.

One issue with the <`img`/> fallback is that the resource will almost certainly download, whether or not the browser supports SVG, affecting a needless performance hit. Artur Ampilogov has a workaround using background images.[118]

118 http://smashed.by/svgfallback

Operation

Now let's talk about actually using the button. First of all, to create an inclusive UX, we need to take care with the placement of the <`button`> and the menu it discloses.

```
<nav aria-label="site">
  <button>
    <svg><use xlink:href="#navicon"></use></svg>
    menu
  </button>
  <ul hidden>
    <li><a href="#main">home</a></li>
    <li><a href="/about">about</a></li>
    <li><a href="/products">products</a></li>
    <li><a href="/contact">contact us</a></li>
    <li><a href="/login">login</a></li>
  </ul>
</nav>
```

- In the example above, we imagine that JavaScript has run, meaning it's safe to hide the menu and reveal the button (hence the `hidden` attribute that JavaScript has added to the menu <`ul`>).

- The <`button`> is placed *inside* the navigation landmark, meaning it will be available to screen reader users who go to the landmark via a shortcut. If the button were outside, users would arrive at an empty landmark with no way to populate it.

- The menu is placed directly *after* the `<button>` in the source order, meaning the first menu item will be the next focusable element after the menu is opened. Because the menu is hidden with `hidden`, its contents are not focusable when it is closed. Invisible elements are not for operation, so they should never be focusable.

In cases where it is impossible (due to design constraints, company politics or whatever's standing in your way) to place the subject menu directly after the menu button as prescribed, you can associate the two elements with `aria-controls`. As I wrote in "Aria-controls Is Poop,"[119] this should be avoided. The attribute is only exposed in JAWS and JAWs implementation is incomplete and unsatisfactory. It announces, "press JAWS key plus Alt plus M to go to controlled element" and offers no way to traverse back.

If you're looking for a way to *transport* a user to a menu (or any other remote element in the page), a link is your best bet:

```
<a href="#nav-menu">navigation menu</a>
<!-- lots of other DOM stuff here -->
<nav aria-label="site" id="nav-menu" tabindex="-1">
  <ul>
    <li><a href="#main">home</a></li>
    <li><a href="/about">about</a></li>
    <li><a href="/products">products</a></li>
```

119 http://www.heydonworks.com/article/aria-controls-is-poop

```
      <li><a href="/contact">contact us</a></li>
      <li><a href="/login">login</a></li>
   </ul>
</nav>
```

(Note the `tabindex="-1"` attribute which fixes sequential focus navigation, as discussed in “Navigation Regions”.)

Should you want to hide the menu until the user arrives at the landmark, you can use the `:target` pseudo-class. I removed the hidden attribute and would supply this CSS:

```
#nav-menu ul {
   display: none;
}

#nav-menu:target ul {
   display: block;
}
```

When you click the `href="#nav-menu"` link, the URL for the page is appended with the `#nav-menu` fragment identifier and the landmark becomes the target, making `display: block;` applicable.

Let’s put that to one side and go back to discussing our button. Since proximity isn’t an issue in our case, the button is really a better fit. For one thing it lets the user close, as well as open, the menu at will. Using JavaScript and WAI-ARIA we can communicate this change in state non-visually.

COMMUNICATING STATE

Communicating the state of functional elements within web interfaces is an important part of making those interfaces inclusive of anyone using assistive technology and who is therefore dependent on the accessibility tree.[120] Remember, the accessibility tree is a version of the DOM that exposes the accessible role, property, value and state information you provide in your markup to for non-visual use.

It is a popular misconception that screen readers are not reactive to JavaScript triggered DOM changes. All popular screen readers listen for changes and update their buffers (their interpreted versions of the DOM using the accessibility tree) each time a change takes place. Changes to text nodes and attributes update the buffer, as well as adding or removing elements.

WAI-ARIA offers a host of state attributes with `true` and `false` values to communicate the presence or absence of the state. I think `aria-expanded`[121] is probably the solution here because it prompts screen readers to explicitly announce "collapsed" (`false`) and "expanded" (`true`). Initially, the menu is not open, so `false` is applicable:

```
<nav aria-label="site">
```

120 http://smashed.by/a11ytree
121 http://smashed.by/ariaexpanded

```
   <button aria-expanded="false">
      <svg><use xlink:href="#navicon"></use></svg>
      menu
   </button>
   <ul hidden>
      <li><a href="#main">home</a></li>
      <li><a href="/about">about</a></li>
      <li><a href="/products">products</a></li>
      <li><a href="/contact">contact us</a></li>
      <li><a href="/login">login</a></li>
   </ul>
</nav>
```

When focusing the menu button, screen readers should announce something similar to "Site navigation, menu button, collapsed" — all the pertinent information about the region's menu system.

CRITICAL JAVASCRIPT

Switching the state of the menu accessibly requires JavaScript. Because the operation of the menu is a fairly fundamental feature, I've written a vanilla JavaScript implementation to embed at the bottom of the page.

As described, the menu will be visible and usable where JavaScript fails or is switched off, but this way it's less likely to fail: if the document itself loads, it's already there in its entirety — no jQuery or other dependency to wait on.

```
(function() {
    // get the button and menu nodes
   var button = document.querySelector('[aria-label="site"]
button');
   var menu = button.nextElementSibling;
    // set initial (closed menu) states
   button.setAttribute('aria-expanded', 'false');
   button.hidden = false;
   menu.hidden = true;
   button.addEventListener('click', function() {
    // toggle menu visibility
      var expanded = this.getAttribute('aria-expanded') ===
'true';
      this.setAttribute('aria-expanded', String(!expanded));
      menu.hidden = expanded;
   });
})();
```

A ROBUST DISCLOSURE

Note that I've not done anything fancy with CSS positioning or animation here. The menu just appears and disappears by switching the display state via the `hidden` attribute. This has three benefits:

- There's no CSS dependency for the menu system, meaning it will function regardless of a CSS failure.

- The menu emerges as part of the document flow, meaning a taller-than-viewport menu is still scrollable into view and does not obscure page content beneath it. Absolute positioning cannot assure us of this.

- As previously mentioned, the `hidden` attribute (like `display: none;`) on a parent element makes any interactive element children unfocusable. Setting `height: 0;` and animating to full height would not have this effect, meaning keyboard users would have to tab through invisible elements while the menu is closed.

It's possible that judicious animation effects can aid comprehension, if designed with care. Avoid the temptation, though, to add animations — especially ones which depend on CSS positioning — just to impress users. As I wrote in "The Precarious X In UX,"[122] only *other designers* tend to be enamored with the finer aesthetic points of the interface itself. Most real users just want to get things done. Catering to them is top priority to an inclusive designer.

122 http://www.heydonworks.com/article/the-precarious-x-in-ux

Here is a little revelation. People are not really into using products. Any time spent by a user operating an interface, twisting knobs, pulling levers or tapping buttons is time wasted. Rather, people are ***more interested in the end result*** *and in obtaining that result in the quickest, least intrusive and most efficient manner possible."*

— Goran Peuc "Nobody Wants To Use Your Product" [123]

Touch Targets

As stated, hidden content which depends on a user action to be revealed is a last resort. Accordingly, the menu should be ever present at reasonably wide viewports. At narrower widths, touch operation is more likely — especially when we encroach on mobile and handheld dimensions. The ease of touch interaction for our hamburger menu needs consideration. We especially want to be inclusive of users with limited dexterity due to rheumatic issues.

Small touch targets for interactive elements make operation needlessly challenging, so what is the minimum size we should aim for? The vast proliferation of device resolutions makes it near impossible to establish a universal figure.

123 http://smashed.by/nobodyproduct

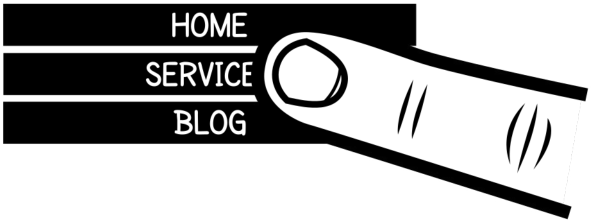
Apple and Android differ in their advice on touch targets with Apple recommending 44 points × 44 points (a density-independent unit of measure related to pixels and unique to Apple) and Android 48px × 48px. At *around* this size or larger, the visual focus feedback is not obscured by the user's finger, which would otherwise force them to use the precarious gestures described in Anthony Thomas' "Finger-Friendly Design: Ideal Mobile Touch Target Sizes:"[124]

> *Users use the fingertip to hit small touch targets because it gives them the visual feedback they need to know that they're hitting their target accurately. But when users have to reorient their finger, it slows their movement down, and forces them to work harder to hit their target."*

Patrick H Lauke has undertaken research for The W3C Mobile Accessibility Taskforce into touch / pointer target size.[125]

Touch target size issues are exacerbated in mobile-sized navigation regions because of the link's proximity. When the surface of your finger spans more than one element, it could be either element that is activated when pressure is applied. This is not the sort of gamble anyone is interested in taking.

124 http://smashed.by/targetsize
125 http://smashed.by/mobilea11y



"Aaargh! I keep hitting the wrong one. Ever heard of vertical padding?"

The BBC Mobile Accessibility Guidelines[126] recommend the inclusion of "inactive space" between elements to remedy this issue, but I suggest this would create a somewhat incomprehensible, broken-up appearance when it comes to navigation regions.

I would recommend mobile viewports display navigation items in a single vertical column — one link per line — and provide a generous enough vertical padding to make each link higher than the diameter of an adult finger pad.

126 http://smashed.by/bbcspacing

Summary

Though this pattern looked at menu buttons explicitly, it was really an exploration of the use of button controls, with icons, in general. By being mindful of the need for accessible labels, potential problems with Windows high contrast mode and ease of touch operation, we ensured our menu button — as well as the content it reveals — is inclusive of differing user settings, circumstances, devices, and assistive technology software.

THINGS TO AVOID

- Background images for icon rendering.
- Omitting accessible names and labels.
- Small touch (or hit) areas.
- Foregoing accessible state communication.

Inclusive Prototyping

THIS BOOK OFFERS A NUMBER OF PRECONCEIVED SOLUTIONS to certain problems, approached with inclusion in mind. Though you're likely to use some as presented (like the *navigation region* pattern) across many of your projects, they're really just examples. They're here to help you practice *thinking inclusively* as you formulate your own patterns to solve your own design problems. It's investing this thought in the early stages of the design process that leads to a robust product.

Drawing on what we learn about our potential audience in the discovery phase[127] of our project, we begin to think about what tasks our app will allow users to perform. In essence, we begin to stockpile verbs: create, sort, edit, buy, read, respond, move, capture, draw, upload.

It's imperative we think first about how these actions might be taken rather than how the experience might look and feel. Thinking about aesthetics and delight is far too high-level at this stage. We need to prototype for usability, and the more inclusive the prototyping experience, the more people will find the product usable.

127 http://www.uxapprentice.com/discovery/

Paper Prototyping

Almost all of the successful projects I've worked on as a designer and developer have included interactive paper prototyping in the early stages of the design process. For example, the Great British Public Toilet Map[128] started out with me drawing and cutting out pictures of toilets in the offices of Neontribe.[129]

Not to be confused with sketching, paper prototyping is about creating an *interactive* prototype from paper, which — with the help of someone to manually operate it — allows user testers to play with a rough-and-ready demo version of the candidate app.

There are a number of advantages to paper prototyping over high fidelity and code-based demos.

- **Most anyone can work with paper.** Coded prototypes require some skill in front-end development. Committing to work with paper means other team members, including back-end developers and others not confident in their design skills, can have a try. Importantly, it also means stakeholders and test users can be included in iteration.

128 https://greatbritishpublictoiletmap.rca.ac.uk/
129 https://www.neontribe.co.uk/

- **It's easy to iterate.** Paper prototypes are mostly just pen on paper, held together with sticky tack. By using sticky tack to piece components together, it's easy to move parts around. If you want new parts, just get out some spare bits of paper and your marker pens. This can be done during test sessions and can involve everyone present — infinitely preferable to sitting around twiddling thumbs while a front-end dev fiddles anxiously with their laptop.

- **It's not polished.** The trouble with high-fidelity mockups and prototypes is that they tend to look finished. People are polite and you won't get honest feedback about something that looks like it's had a lot of work put into it. That it's made of paper at all means it's clearly only an analogy of the real app, so feedback about aesthetic particulars is kept to a minimum. Focus remains on whether the idea solves the problem well.

WHAT YOU WILL NEED

Let's imagine you're embarking on a paper prototype. First, we'll take care of what you need in terms of equipment and why.

- Large pieces of paper (preferably A2)
- Scissors
- Sticky tack
- Permanent marker pens
- Sheets of acetate
- A dry-wipe (whiteboard-style) pen

LARGE PIECES OF PAPER

Whether your prototype is intended to demo an app for small handheld devices or large installation screens, it's better to begin large. A large viewport (paper sheet to which smaller parts of the paper app are adhered) means more people in your test environment can simultaneously see it as it is being used. That means everyone can feel involved, and more and better notes can be taken. Obviously, you'll need a table big enough to sit the prototype on!

SCISSORS

Just like a real app, the viewport will be divided into smaller sections or components. You'll need to cut some of your larger sheets into smaller ones to be placed on top of the base sheet. If you have a guillotine handy all the better, but don't worry too much about all the lines being parallel. The wonkier the prototype looks (within reason!), the less intimidating and more approachable it is.

STICKY TACK

Sticky tack is what makes the paper app modular and lets you quickly move components around, either as part of the app's operation or as part of layout iteration. Whenever a part of a component is liable to be moved or replaced, use a separate piece of paper and stick it in place with tack.

PERMANENT MARKER PENS

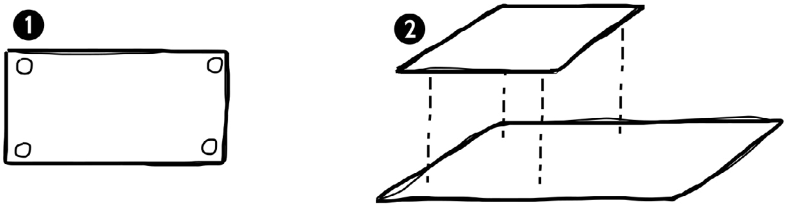
Of course, you'll need to write labels, draw borders around buttons and make various other markings, so some pens are essential. However, try to limit yourself to just a few colors — this isn't an art project. I usually make sure I have a couple of black pens, then choose a red for errors, green for success messages, and blue for links and buttons.

SHEETS OF ACETATE

If you're making an app, there are probably going to be some input fields. Acetate is great when it comes to prototyping input fields because it allows the user tester to write on the paper prototype non-permanently. They can correct their mistakes and the writing can be quickly removed ready for the next test session.

A DRY-WIPE PEN

Naturally, you'll need one of these for use with the acetate (see above). This is not used in the construction of the prototype, but is needed in the testing described below. Make sure the one you take to the test session has some ink in it!



After adhering sticky tack to each corner of your component (1), you can place it on a larger, parent component (2). You can always unstick it and move it later.

TESTING THE PAPER PROTOTYPE

Paper prototype testing is a slightly unusual ritual with a few things to remember and set up.

1. **Bring plenty of spares.** A good session leads to fevered iteration. Lots of spare paper and plenty of pens and scissors to go around are vital.

2. **Write a scenario.** Putting a user tester in front of your prototype cold will amount to stunned inaction. Give them a scenario to work through, something the app should be able to solve for them. It's important this scenario is fictional, and not drawn from the individual's real experiences for the sake of privacy.

3. **Appoint a computer.** Someone who built the prototype will have to operate it. That means reacting to (role-played) input by the user tester. When the tester presses a button, for instance, the computer will have to reorganize the app in response. If the button deletes something this would mean unsticking the paper component and putting it to one side.

4. **Give clear instructions**. Tell the user tester they can touch any items in the interface they think are interactive. Importantly, tell them to speak out loud what they are thinking as they use the prototype. This is important for understanding where there are usability failures and what needs to be overcome.

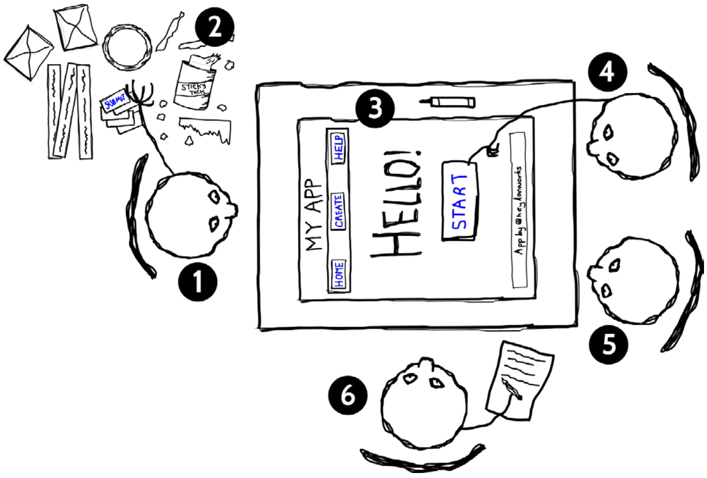
5. **Instill calm.** It's easy for the user to feel tested, so make it clear that you are not testing them but the app itself. If they don't know how to operate it, make it clear in advance that it's not their fault, but yours as the designer.

A T-shirt design I made for usability expert Lily Dart (http://lilydart.com/) during her time consulting my former employer, Neontribe.[130] *It bears her mantra, "Test usability, not users."*

130 https://www.neontribe.co.uk/

THE LAYOUT

To give you a better picture of the testing setup, here's an annotated sketch.



1. The person playing the computer.
2. A collection of prototype bits and pieces (other screens, yet-to-be-revealed functionality, etc.).
3. The app.
4. The user tester, operating the prototype.
5. It's beneficial to have a passive second tester, sat with the user tester for moral support and to encourage dialogue.
6. A note taker. Notice the neutral seating position between the computer and testers. By sitting here, between the two parties at either end, the testers should feel less scrutinized.

From Paper To Code

After a few sessions of paper prototype testing, you should have a lot of torn-up paper and a better idea of the app you should be building. The way you translate the surviving bits of paper into coded modules is critical for an inclusive end product. It's in the creation of your living styleguide[131] that you should be drawing on the inclusive coding practices you've been learning in this book.

In a large team, this can be a collaborative effort with some friendly competition to reach the best solutions. In her article, "From Pages To Patterns: An Exercise For Everyone,"[132] Charlotte Jackson recommends that after agreeing on which components are which and what they should be named, you should get straight into coding. Charlotte sets it out simply, like this:

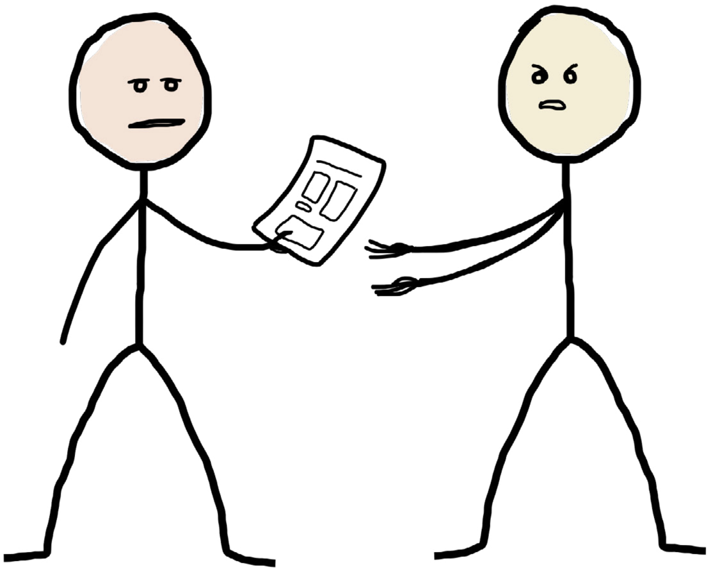
- Everyone grab a component.
- Code it up in HTML and CSS. Set a time limit and resist the temptation to perfect things. It's fine to throw away the code if the designs change.
- Compare and discuss your code.
- Repeat.

131 http://smashed.by/style-guide-tools
132 http://smashed.by/pages-to-patterns

I would suggest a couple of stipulations to really make the most of this in terms of inclusive design. First, I'd recommend several individuals (or teams) work on the same component so there are different solutions to compare. Second, accessibility should be a prerequisite for consideration. If a solution isn't keyboard-accessible, uses a poor structure or is not explicable by screen reader, it's out.

The quickest way to prototype in code is to use standard HTML elements such as `<input>`, `<select>` `<a>` and `<button>`. They give you the behaviors you need to make your component functional. It's usually only when we begin overengineering custom components that we encounter issues. By going straight from paper to HTML, we lay the foundations for efficient code and inclusive experiences. It is based on this foundation that we can enrich the experience with CSS and JavaScript.

After all, you can't even *have* CSS or JavaScript without some HTML on which to stick them. So, you might as well get the HTML right first.



“Here’s your website.”
“That’s just a piece of paper.”
“You asked me to design it, not make it.”

A List Of Products

"What religion are you?"

"Well if you're going to reduce my identity to my religion, then I'm Sikh. But I also like hiphop and NPR. And I'm restoring a 1967 Corvette in my spare time."

"Okay. So, one Sikh, and…"

— The Office, Season 4, Fun Run

Business tends to concern itself with targeting and communicating to a receptive audience. The creators of a photography website, for example, would seek to identify and target people who like photography and, well, websites. For most marketers, research doesn't stop there. They'll look into age groups, related interests, competing websites, income, aspirations and even gender.

Fortunately for you — the inclusive designer — the precarious and error-prone art of marketing is none of your concern because abilities, preferences and circumstances have little to do with interests, tastes or lifestyle choices. And while second guessing who might *want* to use our products is a game we have to play, making assumptions about *how* folks use them can only alienate potential fans and customers.

Targeting a so-called average user is a disastrous interface design strategy, because average users do not exist — except for the handful of people teetering on the apex of your bell curve. The popular utensil firm OXO knows this well and by catering to extreme situations and disabilities first, they've produced highly ergonomic products with mainstream appeal. In OXOs own words:[133]

> *When all users' needs are taken into consideration in the initial design process, the result is a product that can be used by the broadest spectrum of users."*

In the design of the hypothetical photography site, it's tempting for the business to try to write off support for users they don't think will be interested in it. That way, maybe they can save money. "Blind users aren't likely to be interested in photography, so why bother making the site screen reader accessible?" they might think.

Meet Sandra. She was a photographer until the age of thirty-seven, when she lost her sight. Her interest in photography never really diminished, though. She enjoys being an active member of the online photography community where she can trade the technical know-how she's picked up over the years for detailed descriptions of photographs.

133 http://smashed.by/oxo

There's always a Sandra, and she deserves an experience just as good as the Ryans, Barbaras and Xaviers of this world. But also consider the following:

- The features Sandra wants (a social dimension and text descriptions) are not exclusive to her or other blind users for reasons of *being* blind.
- Including these features does not alienate other types of user in favor of people like Sandra.
- Making these features accessible to blind and low-vision users has benefits for other types of user. For example, `alt` text describes images in text-only browsers or where image requests return a 403, 404, 500 or 503 status. Screen reader accessibility also requires keyboard accessibility, which helps folks with a range of rheumatic complaints, as well as conditions that cause inaccuracy with a mouse, such as Parkinson's disease or dyspraxia. And that doesn't begin to cover all of the folks who use the keyboard when browsing — clinical condition or no.

Everybody is a keyboard user when eating with their mouse hand.”

– Adrian Roselli[134]

No matter what the content or the commercial offer, there is no reason to second-guess how the user might operate the interface. In fact, doing so can only reduce the quality of the product. To save money, you are going to want to make just the one interface, not a different one for each imagined user group. So, you make it inclusive. You make inclusivity a *quality* of the product — not an expensive additional feature.

When inclusion is integrated into the design process in this way, it’s very little more work. In fact, using standard technologies in expected ways often means less work. For example, devising an inaccessible, custom-authored `<div class="heading-3">` is actually *more* typing than using a standard `<h3>`. It gets even more verbose when you’re wedded to using a `<div>` and have to fix the heading accessibility *after the fact* with ARIA attributes:

```
<!-- life is too short! -->
<div class="heading-3" role="heading" aria-level="3">Heading
text</div>
```

134 https://twitter.com/aardrian/status/388733408576159744

The Virtue Of Lists

For this particular pattern, I want you to imagine a hypothetical photography app has a built in shop, selling photographic prints. The product list is the set of prints returned after the user has entered a product search term. We need to look at how individual products might be marked up and presented, including illustrations and calls to action — the aim is to get users to buy things, after all.

I've expounded the virtues of lists in previous patterns. They itemize their contents accessibly, group content thematically, and break up otherwise structureless, undifferentiated runs of prose. So far, we've only included text and inline elements (links) to make terse list items, but it's perfectly acceptable to define more complex structures.

Though the pattern as documented in your pattern library is likely only to define the structure of one product item, it's important to acknowledge context and belonging first. Let's start a list of products and use the contents of the first list item as the product template.

Using an `<h3>` element, let's give our product a title.

```
<li>
   <h3>
      Naked Man In Garage Forecourt
      <a href="/photographer/kenny-mulbarton">by Kenny
Mulbarton</a>
   </h3>
</li>
```

Each product should have a heading of the same (third) level because they are equal in hierarchy as members of the same list. Though it's tempting to use a higher heading level like <h2> to pick out featured products, this should be avoided as it would make for a nonsensical structure. Better to show that a product is featured or recommended using some descriptive text, highlighted with CSS.

```
<li>
   <h3>
      <strong class="highlight">Featured:</strong> Naked Man
In Garage Forecourt
      <a href="/photographer/kenny-mulbarton">by Kenny
Mulbarton</a>
   </h3>
</li>
```

The highlight style makes featured items easy to identify visually while scanning and the "Featured" text provides the information non-visually to assistive technologies.

Some screen reader users will explore the product list by heading. With the `<h3>`s in place, they can move from product to product using either the *H* key (next heading) or 3 (next third-level heading). Other screen reader users might traverse the list using direct list navigation. For instance, JAWS offers the *I* (for *item*) shortcut key to move from one list item to the next. Giving users choice is the first of Henny Swan's[135] accessible UX principles, as developed at the BBC:

1. **Give users choice.**
2. Put users in control.
3. Design with familiarity in mind.
4. Prioritize features that add value.

In the absence of author CSS, the browser will display each list item with a bullet point and render the heading text in a large, bold font. The visual structure is therefore clear, even where CSS is (temporarily or permanently) unavailable or overridden with a user syle sheet. Separating content and presentation[136] like this also makes it easier for users to apply their own user styles[137] should they wish to.

135 http://www.iheni.com/
136 http://webaim.org/techniques/css/#sep
137 http://www.opera.com/docs/usercss/

If the user were to follow the link inside the heading to the photographer's own page, the photographic prints listed at *that* location should be similarly marked up and presented. As "A Web For Everyone"[138] implores:

> *Present things that are the same in the same way. One way to help users find their way around a site is to be consistent in how elements of the site are presented and labeled, which doesn't mean that the site must be boring with no variation or texture."*

Key Information

So far, our product only has a title. The user will no doubt be interested in information such as size, price and customer rating. We can group and label these key pieces of information clearly, using a definition list:

```
<li>
   <h3>
      Naked Man In Garage Forecourt
      <a href="/artist/kenny-mulbarton">by Kenny Mulbarton</a>
   </h3>
   <dl>
      <dt>Size:</dt>
      <dd>90cm × 30cm</dd>
      <dt>Price:</dt>
      <dd>€35.95</dd>
```

138 http://smashed.by/aweb4everyone

```
      <dt>Rating:</dt>
      <dd><img src="/images/rating_4_5.svg" alt="">4 out of 5
stars</dd>
   </dl>
</li>
```

NOTES

- Like an unordered list (`<ul>`), the definitions list (`<dl>`) has its items enumerated by screen readers so users know how many to expect. When first encountered, screen readers like VoiceOver will announce "Definition list", letting users know that they should expect key/value pairs.
- User agents style definition lists by indenting the `<dd>` elements, meaning a visual hierarchy is present in the absence of author CSS.
- "90cm × 30cm" is written using the correct (multiplication) Unicode character, so that "Thirty C M *times* ninety C M" is announced in screen readers. Using a lowercase *x* would be a poor substitution. Not only would it be announced as "ex", but it would also be visually unclear — especially when rendered with a serif font.

- We're using the well-established convention of displaying stars to denote the customer rating. To reinforce what the stars represent, we're also spelling out the rating adjacently. A null `alt` value (`alt=""`) omits the image from screen readers. A description of the rating in place of the image *and* next to the image would mean a tedious duplication for screen reader users.

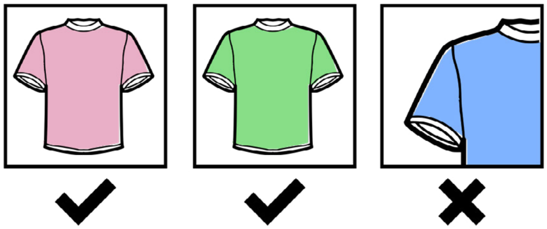
- To be more precise, the "Size" label should arguably be "Dimensions." However, size is a simpler and more widely understood word and concept which — practically speaking, given the context — means the same thing. Ashley Bischoff's presentation, "Embracing Plain Language for Better Accessibility"[139] offers some excellent advice on substituting the pretentious and robotic for the simple and human when choosing words.

Product Thumbnail

The intimate relationship between images and advertising hardly needs restating here. Suffice to say, unless you're selling an intangible product like a six-month subscription or eternal happiness, your users are going to want to get a good look at their potential purchase.

139 http://www.handcoding.com/presentations/plaina11y/#cover

The product thumbnail is essentially the *canonical* image of the product. As such, it should show the product in its entirety, unobscured and with no distracting artifacts. Similar products (such as green and blue versions of the same T-shirt design) should be presented the same way, from the same angle, denoting equivalency.



Art directed, creative product photography[140] may be evocative, but increases cognitive load[141] for users, forcing them to try to unpick exactly what form the product takes. It should be the preserve of the product's permalink page, if at all.

140 http://smashed.by/productphoto
141 https://en.wikipedia.org/wiki/Cognitive_load

ALTERNATIVE TEXT

Fortunately, our photographic print thumbnails are images *of* images so composition and lighting take care of themselves. That leaves the matter of prescribing alternative text so that the product appearance is described to blind and low-vision users.

Always including `alt` text is one of the first lessons we learn when embarking on accessible interface design, but it's precisely *what* text we include that makes the difference between accessible and inaccessible, inclusive and exclusive. To avoid duplication in the last example, the correct answer was — perhaps counterintuitively at first glance — explicitly *no* alternative text.

Aware that the design of `alt` text — like the design of anything else — should be informed by context, the WAI (Web Accessibility Initiative) offers the `alt` decision tree.[142] For our thumbnail, which does not contain text or describe an action, we have to ask: "Does the image contribute meaning to the current page or context?"

The answer is: it can, if we provide information which isn't already in the image's title. Since the title is very terse and literal, we can seize the opportunity:

142 https://www.w3.org/WAI/tutorials/images/decision-tree/

```html
<li>
   <h3>
      Naked Man In Garage Forecourt
      <a href="/artist/kenny-mulbarton">by Kenny Mulbarton</a>
   </h3>
   <img src="/images/naked-forecourt-man.jpg" alt="High-
contrast black and white image of a naked man
nonchalantly leaning against a gas pump." />
   <dl>
      <dt>Size:</dt>
      <dd>90cm × 30cm</dd>
      <dt>Price:</dt>
      <dd>€35.95</dd>
      <dt>Rating:</dt>
      <dd><img src="/images/rating_4_5.svg" alt="">4 out of 5
stars</dd>
   </dl>
</li>
```

Note that I've communicated an important category of the image (black and white photography) and also tried to capture the *mood* of the image ("nonchalantly leaning"). Neither piece of information is available elsewhere to blind users or users for whom images are otherwise unavailable. As a user on a slow network who has turned off images, or they've failed to load, the alternative text is visible to me. With descriptive alternative text in place, I might still be convinced to bookmark the photographic print to have a look when I'm on a Wi-Fi connection.

IMAGE PERFORMANCE

In “A Menu Button” we looked at the many ways to render the simple, three-lined navicon. One of the advantages of using SVG is that its file size can be extremely small, especially when using repeated geometric shapes like rectangles, by way of the `<rect>` element.

Unfortunately, emblematic impressions of the photographs we have for sale, reimagined using rectangles and circles, are not going to cut it here. We’ll have to defer to a weightier but more accurate raster format such as PNG or JPG. To get an impression of how slow loading a page full of raster thumbnails can be, find an existing site with the kind of content we’re discussing, then open Chrome’s developer tools. Next, press the *Toggle device toolbar* button (*Cmd + Shift + M*). Choose *Show throttling* from the menu and select “Regular 3G” — not the slowest or fastest connection.

Now reload the page and wait. And then wait some more.

But boredom isn’t the only issue. Loading this many images over a mobile network is costly too. To give you an impression, take the URL of the site you were just testing and head over to What Does My Site Cost[143] to find out how many cents it’ll cost in data to retrieve just that one page, replete with all the images.

143 https://whatdoesmysitecost.com/

Web developers tend to have expensive data contracts and spend most of their working day connected to lightning-fast broadband. The experience is both a privileged and an unrepresentative one. To be inclusive of those on poor networks, with smaller budgets or being subject to huge roaming charges, we need to mitigate the impact that loading all of these images is likely to have.

This book is not the place to discuss the ins and outs of handling image assets for performance but, briefly, let's consider some important techniques.

1. **Compression:** If you do nothing else, make sure your images are compressed. Compressing images needn't be the arduous, manual task it once was, now that we can make use of automation tools like ImageOptim.[144]

2. **Lazy loading:** There's no need to load images unless they are to be seen. Using lazy loading means not downloading the image resource unless or until the `<img/>` element enters the viewport. There are a number of scripts and plugins available, each working by replacing a dummy `src` value with a real one contained in a `data-` attribute.

144 https://imageoptim.com/

For our product list, each image's markup might look like:

```
<img src="dummy.jpg" width="400" height="300"
data-src="/images/naked-forecourt-man.jpg">
```

3. **The <picture> element:** This allows you to tailor images to viewports. Since smaller viewports are more likely to be used with mobile devices and networks, you can benefit from prescribing physically smaller — therefore less weighty and costly — images. We might, accordingly, incorporate markup similar to that in the following illustration:

```
<picture>
  <source media="(min-width: 800px;)"
  srcset="/images/naked-forecourt-man_large.jpg">
  <img src="/images/naked-forecourt-man_small.jpg"
  alt="High-contrast black and white image of a naked man
  nonchalantly leaning against a petrol pump.">
</picture>
```

Note that the `<img/>` element, included last, will have its `src` attribute automatically switched to that defined in the `<source>` element *only* if the viewport is more than `800px` wide. Be aware that this is not an exact science. It's entirely possible that some of your users will be connected via a slow mobile network on a device featuring a viewport greater than `800px` in width.

The “Buy Now” Action

Unless we’re *entirely* lacking in business acumen, we’re going to want to facilitate the user’s purchase of the print. Including a “Buy Now” button is perhaps the most straightforward way to go about this. Even so, there are some pitfalls to avoid.

BUTTONS VERSUS <BUTTON>S

The first pitfall regards the concept of buttons and their expected behavior. Despite so-called “Buy Now” buttons having the appearance of a button, their behavior links to a resource — they are hyperlinks.

```
<a href="/product/naked-man-in-garage"
class="button">Buy Now</a>
```

In a noble attempt to create a more accessible experience, this leads some developers to mistakenly provide the `role="button"` attribute. This overrides the implicit accessible role of “link” with an explicit role of `"button"`, meaning the link will be announced as a button in screen reader software. Since the linking *behavior* of loading a new page is still intact, this is deceptive and confusing for blind screen reader users and should be avoided.

```
<!-- don’t do this -->
<a href="/product/naked-man-in-garage"
role="button">Buy Now</a>
```

Since the visual appearance of a button suggests button-like behavior, it is recommended to preserve that aesthetic for actual <button>s. I suggest you maintain a separate call-to-action style for links which has more visual purchase than a standard link but which does not resemble a button.

```
a {
   /* standard link styles */
}

a.call-to-action {
   /* emphatic link styles */
}

button {
   /* <button> element styles */
}
```

basic link
action →
button

Three controls: a basic link with underline; an action link with a border; and a button with a blue background.

NOTES

- I highly recommend that, although each of these elements are treated differently, they share a common color — a color which denotes the shared *interactive* quality of the three elements. For the sake of easy cognition, if links should be blue, so should buttons: blue comes to mean clickable.

- I have qualified the `.call-to-action` class with the element name `a`. You may have read elsewhere that this is redundant, but it serves an important purpose: it restricts the use of `.call-to-action` styles to the `a` element. By using `.call-to-action` on a `<button>`, or an inaccessibly interactive `<div>` element, the style will not be invoked. This effectively prevents a developer from using the wrong element for the job. In the excellent article "How Our CSS Framework Helps Enforce Accessibility,"[145] Ian McBurnie details a number of similar provisions.

- Another way that `<button>`s are differentiated from links is that they don't have a `pointer` cursor style. In "Buttons shouldn't have a hand cursor,"[146] Adam Silver

145 http://smashed.by/enforcea11y
146 http://smashed.by/handcursor

explains why manually adding `cursor: pointer` to <`button`> elements is a usability mistake. It is not standard behavior to invoke this cursor style on <`button`>s, and it's risky to break with long-held convention.

EMBRACE STANDARD BEHAVIOR

A second pitfall can be laid out much more succinctly: if the behavior is to link, *use a link's behavior to do it*. It sounds simple when put like that, but we tend to overengineer interfaces. Client-side JavaScript frameworks especially have a habit of supplanting standard link behavior. This might help the framework developers because they like to work exclusively within the realm of JavaScript. It does not help users.

As well as basic linking behavior breaking when JavaScript is not available, even where it *is* available the experience is lacking: URLs only reveal "`javascript:;`" on hover and the link cannot be dragged into the browser tab list. The right-click context menu intended for links is not invoked either. A browser only knows to provide these things if you use the <`a`> element in an expected way.

```
<!-- this is robust and featureful -->
<a href="/product/naked-man-in-garage" class="call-to-
action">Buy Now</a>

<!-- this is not robust or featureful -->
<a href="javascript:;" data-action="naked-man-in-
garage" class="call-to-action">Buy Now</a>
```

UNIQUE, DESCRIPTIVE LINK TEXT

Let's not forget that each link for each product currently has an identical "Buy Now" label, meaning screen reader aggregated link lists will take the following unhelpful form:

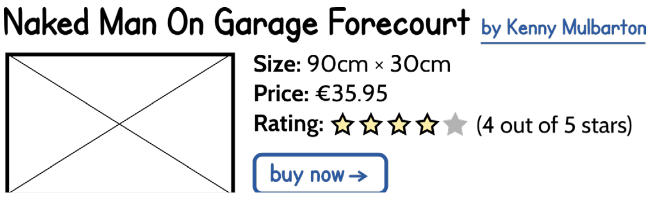
- Buy Now
- Buy Now
- Buy Now
- Buy Now
- Buy Now

Since the title of the product is already present visually and in context via the `<h3>`, we should remedy this with some visually hidden text, placed within the link.

```
<a href="/product/naked-man-in-garage" class="call-to-
action"> Buy <span class="visually-hidden">Naked Man In
Garage Forecourt</span> Now
</a>
```

Now the link text makes sense independent of context. The `visually-hidden` class should, of course, correspond to a screen reader accessible hiding technique,[147] as introduced in "A Blog Post." Note that having the product name in the link text has an added advantage: it is a well-known technique for improving search engine visibility. Google recommends using descriptive link text[148] because its crawler can see that (where link text and the destination page's <`title`> or <`h1`> are similar) relevance is likely to be high.

Here's how the overall layout is shaping up:


Naked Man On Garage Forecourt
by Kenny Mulbarton
Size: 90cm × 30cm
Price: €35.95
Rating: ☆☆☆☆★ (4 out of 5 stars)
buy now →

BLOCK-LEVEL LINKS

In HTML5, it has been made acceptable to place block-level content inside link (<`a`>) elements.[149] That is, I could wrap my entire product item's contents in a link. This has the advantage of increasing the clickable area of the item.

147 http://smashed.by/hidingcontent
148 http://smashed.by/linkarchitecture
149 http://html5doctor.com/block-level-links-in-html-5/

In our case, this comes at the cost of removing the child link to the photographer’s page.

```
<li>
   <a href="/product/naked-man-in-garage">
      <h3>Naked Man In Garage Forecourt</h3>
      <img src="/images/naked-forecourt-man.jpg" alt="high
contrast black and white image of a naked man nonchalantly
leaning against a petrol pump." />
      <dl>
         <dt>Size:</dt>
         <dd>90cm × 30cm</dd>
         <dt>Price:</dt>
         <dd>€35.95</dd>
         <dt>Rating:</dt>
         <dd><img src="/images/rating_4_5.svg" alt="">4 out
of 5 stars</dd>
      </dl>
   </a>
</li>
```

Now that the markup is not technically *nonconforming*, there are a number of more important **UX** concerns to address:

1. The link doesn’t have a dedicated label, either visually or determinable by assistive technologies. The outcome is that users will not know they can interact with the item until they *interact* with *it* (by hovering the cursor over it to produce a pointer, say, or by invoking a focus style).

2. This structure produces unexpected behaviors in some screen readers. For example, in VoiceOver, when I try to navigate to a product heading using *Ctrl* + *Alt* + *Cmd* + *H*, the surrounding link is focused and the heading semantics ("Third-level heading") are not announced.

3. In all assistive technologies, focusing the link is liable to trigger the readout of all its textual contents, which is rather verbose.

4. Operators of touch devices will undoubtedly find themselves accidentally following the link while pressing a part of the screen which *appears* to be uninteractive. Very annoying.

In general, it's better to avoid block-level links like this one. Just because the specification mandates that something is technically valid markup doesn't mean it produces an agreeable user experience. In his article "To Hell With Compliance,"[150] accessibility consultant and trainer Karl Groves sets out why the bureaucratic box-ticking of compliance testing gains us little ground towards successful inclusion.

150 http://www.karlgroves.com/2015/01/06/to-hell-with-compliance/

It's the bare minimum we can offer. Complaints toward online services are almost always from people unable to actually *use* an interface, but a 100%-compliant interface can still be entirely unusable. By the same token, an interface with the odd superficial error may be so simple, well structured and clear that it's still a pleasure to use.

Because so many accessibility errors relating to assistive technologies are markup errors, and because markup errors are so easy to identify, we've grown up in an accessibility remediation culture that is assistive technology obsessed and focused on discrete code errors.

Inclusive design has a different take. It acknowledges the importance of markup in making semantic, robust interfaces, but it is the user's ability to *actually get things done* that it makes the object. The quality of the markup is measured by what it can offer in terms of UX.

SERPs

Though our product pattern is now more than satisfactory, we need to accept that our own interface is not the only way our customers will find and consume our product information. A well-indexed site will also expose products in search engine results pages, or SERPs.

The advantage of using SERPs like Google's is that no matter the content you are searching, the interface remains the same. Notably, some assistive technology users rely heavily on Google Search because it is a prelearned interface. A common tactic is to use the site: prefix to return results confined to a chosen website:

```
site:shop.the-photography-site.com [search term]
```

We don't have control over the aesthetic with which Google lists our content. But we do have control of the wording. For example, the listed page's `<title>` doubles as the linked `<h3>` element of an individual search result. All the more reason to write descriptive, easily understood `<title>` text as described in "The Document." In fact, well-written `<title>` text can elevate your search rankings, but avoid spammy `<title>`s.[151] Not only will Google penalize you, but such `<title>`s tend to constitute unreadable, and therefore uninclusive, content.

Since we are dealing with products, we have an opportunity to enhance our product listing using structured data.[152] In brief, structured data increases the meta information parsable by machines like Googlebot. It's a little like WAI-ARIA, but for archiving rather than assistive technology support.

151 http://www.hobo-web.co.uk/title-tags/#spammy-title-tags
152 https://developers.google.com/structured-data/

Structured data affects the user directly since the search listing will be enhanced to include more detailed information, such as the price and rating of the product. By bringing this information to search engine results, we enhance the experience of those who choose to browse our content in this context.

USING THE PRODUCT VOCABULARY

Structured data is divided into *vocabularies* which apply to different varieties of content. For our purposes there is a product vocabulary, defining product-specific properties, to be found at http://schema.org/Product.

The best place to include structured data for an individual product is at its permalink page: the page “Buy Now” will take the user to see greater detail about the product and choose purchasing options. At this page, the markup will use a template which might look like this:

```
<main id="main">
   <h1>
      Naked Man In Garage Forecourt
      <a href="/artist/kenny-mulbarton">by Kenny Mulbarton</a>
   </h1>
   <img src="/images/naked-forecourt-man.jpg" alt="High-
contrast black and white image of a naked man
nonchalantly leaning against a petrol pump." />
```

```
   <dl>
      <dt>Size:</dt>
      <dd>90cm × 30cm</dd>
      <dt>Price:</dt>
      <dd>€35.95</dd>
      <dt>Rating:</dt>
      <dd><img src="/images/rating_4_5.svg" alt="">4 out of 5 
stars</dd>
   </dl>
   <h2>Choose a payment method</h2>
    <!-- purchasing widget here -->
</main>
```

Now, let's adapt the plain markup to include the structured data. I am basing this on an example provided by Google itself[153] using microdata (one version of structured data).

```
<main id="main" itemscope itemtype="http://schema.org/
Product">
   <h1>
      <span itemprop="name">Naked Man In Garage Forecourt</
span>
      <a href="/artist/kenny-mulbarton">by Kenny Mulbarton</a>
   </h1>
   <img itemprop="image" src="/images/naked-forecourt-man.
jpg" alt="High-contrast black and white image of a naked man 
nonchalantly leaning against a petrol pump." />
```

153 http://smashed.by/richsnippets

```
   <dl>
      <dt>Size:</dt>
      <dd>90cm × 30cm</dd>
      <dt>Price:</dt>
      <dd>
         <span itemprop="offers" itemscope itemtype="http://
schema.org/Offer">
            <meta itemprop="priceCurrency" content="EUR" />
            €<span itemprop="price">35.95</span>
         </span>
      </dd>
      <dt>Rating:</dt>
      <dd>
         <img src="/images/rating_4_5.svg" alt="">
         <span itemprop="aggregateRating" itemscope
itemtype="http://schema.org/AggregateRating">
            <span itemprop="ratingValue">4</span> stars,
based on <span itemprop="reviewCount">13</span> reviews
         </span>
      </dd>
   </dl>
   <h2>Choose a payment method</h2>
    <!-- purchasing widget here -->
</main>
```

NOTES

- The `itemscope` Boolean attribute and `itemtype` reference are placed on the wrapping <`main`> element to define the context for our product data.

- Additional <`span`>s are used to define the properties where necessary. The name of the product is defined by wrapping the product's title, "Naked Man In Garage Forecourt," in a <`span`> bearing `itemprop="name"`.

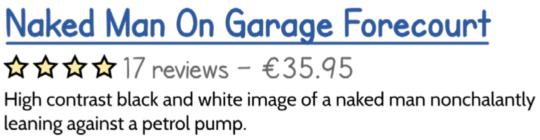
- The rating and price ('offer') are defined using the nested "Offer" and "Aggregate Rating" vocabularies.

- Some properties are not based on displayed content (text nodes), such as the three-letter ISO currency of "EUR." Instead, we supply this information using a <`meta`> tag. A full list of ISO currency codes is available on Wikipedia.[154]

- You can test your structured data[155] with Google's tool to make sure the properties are valid and applicable.

154 https://en.wikipedia.org/wiki/ISO_4217#Active_codes
155 https://developers.google.com/structured-data/testing-tool/

In Google’s results pages, the final product (no pun intended) will look something like the following:


Naked Man On Garage Forecourt
☆☆☆☆ 17 reviews – €35.95
High contrast black and white image of a naked man nonchalantly leaning against a petrol pump.

Fortunately, Google has ensured that the additional structured information is accessible to our potential visitors. For example, they use a custom <`g-review-stars`> element which incorporates an `aria-label` to spell out the rating.

```
<g-review-stars>
  <span class="_ayg" aria-label="Rated 4.0 out of 5">
    <span style="width:66px"></span>
  </span>
</g-review-stars>
```

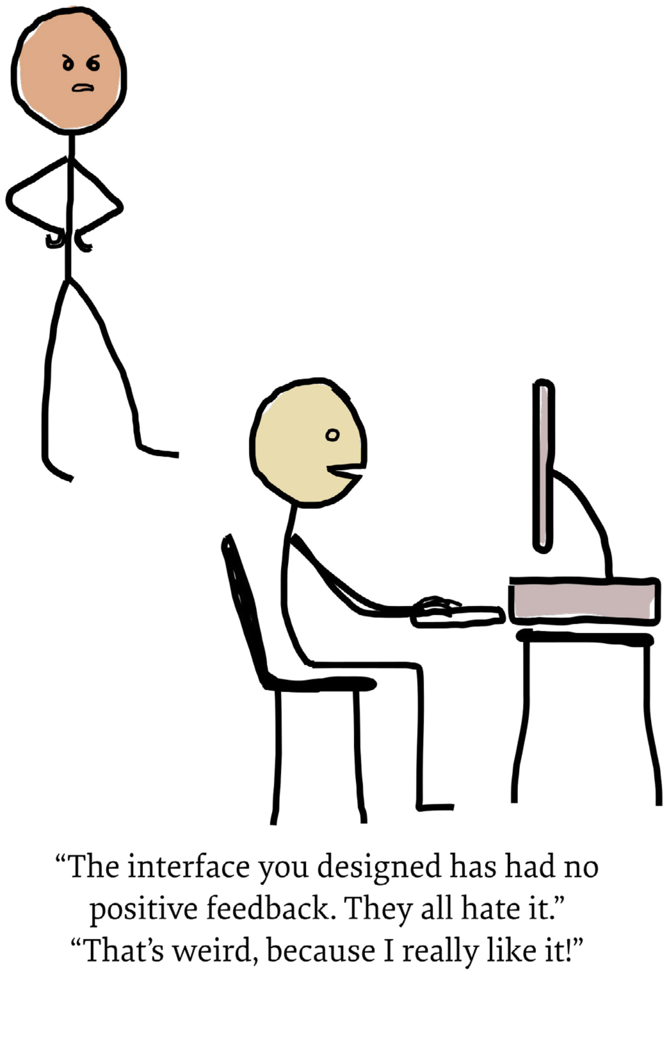
Summary

This pattern constitutes just one way to go about designing and coding a product list for a maximally inclusive experience. As with all the patterns herein, it’s really just an excuse to hone our inclusive design chops. As in previous patterns, the organization and structure of content is paramount.

However, this was the first time we looked deeply into image accessibility, from both the perspectives of alternative text composition and performance. In catering to blind consumers, those who cannot afford generous data contracts, and anyone accessing your content from *outside* your interface, it was a chance to really push the limits of inclusive design.

THINGS TO AVOID

- Inconsistently composed or unoptimized product images.
- Unhelpful or misleading alternative text.
- Block-level links containing lots of text content.
- Links to product permalinks without the product name in the link text.


“The interface you designed has had no positive feedback. They all hate it.”
“That’s weird, because I really like it!”

A Filter Widget

In the previous chapter we went commercial and looked at a pattern for displaying products. How we marked up individual product components and how we grouped them together were both pertinent to inclusive design. We even made sure they would be rich with accessible information *outside* of our own interface, as search engine results.

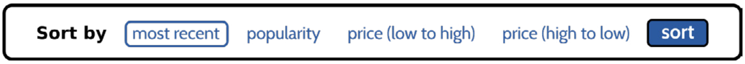
What we've yet to address is providing the tools for users to reorganize and sort lists of content. Filtering tools are important because they offer an additional dimension of control over search, facilitating users' ability to prioritize the information they're viewing. *Put users in control* is the second of Henny Swan's principles.

1. Give users choice.
2. Put users in control.
3. Design with familiarity in mind.
4. Prioritize features that add value.

Like the navigation regions we covered earlier in the book, filtering widgets are *meta* content, provided as a tool to change the content of the page. To make ours inclusive, we need to consider blind and keyboard users, potential performance issues and visual clarity.

The pattern will be founded on progressive enhancement and will work in the absence of both CSS and JavaScript, though we shall improve the experience inclusively with a careful application of those two technologies respectively.

How It Might Look



Note that the selected control is highlighted in a color-independent fashion so that our color-blind and low vision users can determine the selected option.

The Markup

Some complex JavaScript widgets, like tab interfaces,[156] depend on WAI-ARIA semantics and custom (JavaScript) keyboard bindings to be made fully accessible by keyboard and to screen readers. However, the first rule of ARIA use[157] states:

*If you **can** use a native HTML element or attribute with the semantics and behaviour you require **already built in**, instead of re-purposing an element and adding an ARIA role, state or property to make it accessible, **then do so**.*

156 http://heydonworks.com/practicalariaexamples/#tab-interface
157 https://www.w3.org/TR/aria -in-html/#first-rule-of-aria-use

Standard browser-enabled interactivity is more performant and robust than that facilitated by custom JavaScript. If you can use it, you should, so it's worth investigating.

Solutions usually rely on harnessing the behavior of standard HTML form elements. These have idiosyncratic semantics, key bindings and behaviors built in for our convenience. Just because JavaScript is (well, might be) available doesn't mean you should use it to reinvent the wheel.[158]

Our sorting widget consists of a set of mutually exclusive options each with their own unique label, and grouped under a common label. We can create this structure using a `<fieldset>`, `<legend>` and radio buttons. Imagining we're still dealing in products, this is how the widget might look:

```
<form role="form" class="sorter" method="get">
  <fieldset>
    <legend>Sort by</legend>
    <input type="radio" name="sort-method" id="most-recent"
value="most-recent" checked>
    <label for="most-recent">most recent</label>
    <input type="radio" name="sort-method" id="popularity"
value="popularity">
    <label for="popularity">popularity</label>
    <input type="radio" name="sort-method" id="price-low-
high" value="price-low-high">
```

158 http://www.heydonworks.com/article/reinventing-the-hyperlink

```
      <label for="price-low-high">price (low to high)</label>
      <input type="radio" name="sort-method" id="price-high-
low" value="price-high-low">
      <label for="price-high-low">price (high to low)</label>
   </fieldset>
   <button type="submit">sort</button>
</form>
```

NOTES

- Having the ARIA role of `form` on a `<form>` seems counterintuitive, but it turns the widget into a region, making it navigable in screen readers using shortcuts. Since the basic functionality works without JavaScript and triggers a page refresh, this helps users navigate back to the form from the top of the document.

- The `<form>` contains a single `<fieldset>` which is used to group the radio options under the label "Sort by", followed by a submit button. When a radio `<input>` is focused, the <legend> content is announced followed by the `<input>`'s `<label>`. To begin with, the first option ("most recent") is checked. Standard behavior is that only this checked `<input>` is focusable by Tab. Focusing it would trigger the announcement of "Sort by most recent, selected, radio button, one of four." "Sort by" is the group label; "most recent" is the element label;

"selected" is the element state; "radio button" is the element role. The number four is the total number of radio buttons which share a common `name="sort-method"`.

- The browser delegates arrow keys (left and right, up and down) for selecting different options. Pressing the right arrow key, for instance, will focus and select the next radio button with the `name="sort-method"` attribute - "popularity." In screen readers this will announce something similar to "Sort by popularity, selected, radio button, two of four." Note that "Sort by" is read whichever radio button is selected. This ensures that if I were to leave the widget and begin focusing other page elements and then return to it, I'd be reminded of the sorting function no matter which option is currently selected.

- We're using the `get` method on the form to rebuild the page from the server without relying on client-side JavaScript, at this stage. For example, submitting the form with the popularity option selected will build a page with `?sort-method=popularity` as the query parameter. We'll enhance the experience with JavaScript in the "JavaScript Enhancement" section to come.

CSS Enhancement

By designing our sorting widget with conventional, well-supported markup, we've already ensured it is accessible to keyboard and screen reader users. It can also be manipulated without a dependence on JavaScript and assumes a familiar form.

Despite the inclusive nature of standard radio buttons, they offer limited styling opportunities. This is the main reason why we dispense with them in favor of either completely inaccessible `<div>`- and `<span>`-based solutions, or comparatively fragile and complex WAI-ARIA implementations.

- However, as I wrote in "Replacing Radio Buttons Without Replacing Radio Buttons,"[159] you don't actually *have* to style the `<input type="radio">` element directly. You can hide it and style the `<label>` as a proxy.

In the following structure, the `for`/`id` relationship means that clicking (or pressing) the `<label>` operates the associated `<input>`. That is, the `<label>` extends the click area of the `<input>`.

```
<input type="radio" name="sort-method" id="most-recent"
value="most-recent" checked />
<label for="most-recent">most recent</label>
```

159 http://smashed.by/radio-buttons

- Not only does this help provide a larger, more ergonomic hit area for touch users, but we can use CSS to indicate the `<input>`'s `:focus` and `:checked` states in CSS. Note the use of adjacent sibling combinators to *pass* state to the label:

```
[type="radio"] + label {
   cursor: pointer;
    /* other basic styles */
}

[type="radio"]:focus + label {
    /* focus styles */
}

[type="radio"]:checked + label {
    /* selected styles */
}
```

Now that all the interaction and visual feedback is attached to the label, we can safely hide the obstinately ugly radio button from view.

```
.sorter [type="radio"] {
   position: absolute !important;
   width: 1px !important;
   height: 1px !important;
   padding:0 !important;
   border:0 !important;
   overflow: hidden !important;
   clip: rect(1px, 1px, 1px, 1px);
}
```

In summary: never fix with JavaScript and WAI-ARIA what can be achieved with HTML and CSS. "WTF, forms?"[160] by Mark Otto provides similar CSS enhancement solutions for checkboxes, `<select>` elements, and file inputs.

JavaScript Enhancement

The pattern we've devised is already fully functional and robust. It's fair to say it's a bit *clunky*, though. Where JavaScript is available to the user, we ought to enhance the experience where we can. But just because JavaScript is available, doesn't mean we should get carried away tearing everything up and rewriting it. The foundation is solid; there's no need to recreate in JavaScript the parts that already work without it.

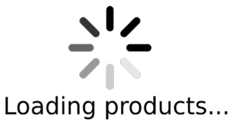
Each time a user chooses a new filtering option and hits the sort button, there's currently a page refresh. By now, you'll know what this means for screen reader and keyboard users: having to hear all the page information again or having to traverse through all the document's preamble to get to the content they were just interacting with. Or both.

160 http://wtfforms.com/

Where possible, we should aim to repopulate our list with XHR.[161] That way, new content can be added without reloading the page and the keyboard-dependent user can progress directly from the widget (where focus will remain) into the content.

WAITING

Telling users that content is pending and will arrive in a matter of time is the preserve of the ubiquitous loading symbol, which looks a bit like this:



The trouble is, this is only determinable by sighted users. It's important to communicate that content is being fetched to screen reader users as well. Otherwise they might be left wondering if hitting the sort button did anything at all.

The WAI-ARIA specification provides live regions[162] for just this kind of thing. Usually, content is only announced in screen readers when either:

161 http://smashed.by/xmlhtml
162 http://smashed.by/livereg

- An element is focused either by the user or programmatically.
- The user navigates to an element using their screen reader's own navigation commands (such as pressing *9* in NVDA to announce the following line).

But live regions announce their content simply when that content changes. In practice, this means we can provide commentary to screen reader users without asking them to leave their location in the page. After the sort button is pressed, we can populate a live region with the message, "Please wait. Loading products."

```
<div aria-live="assertive" role="alert">
  Please wait. Loading products.
</div>
```

Then, when the products are loaded, we can change the live region content to inform the user:

```
<div aria-live="assertive" role="alert">
  Loading complete. 23 products listed.
</div>
```

NOTES

- The `aria-live="polite"`[163] property and the `status`[164] role are equivalent. Both are provided to maximize compatibility across platforms and screen readers (in some setups, only one or the other is recognized).

- We would populate the live region with the message using simple JavaScript DOM manipulation: `liveElement.textContent = 'message'`.

- The equivalent `assertive` and `alert` values mean the current readout of the screen reader will be interrupted to announce the live region's new message. Since the user might move away from the widget to read some other content, this means they will be told that the products are ready immediately. Alternative `polite` and `status` values would mean the live region contents are announced only after the screen reader has finished announcing the content it is currently set to announce.

- For app notifications, you would need to style and display the live region's contents so that it doubled as a visual message. However, the loading symbol (and its

163 http://w3c.github.io/aria/aria/aria.html#aria-live
164 http://w3c.github.io/aria/aria/aria.html#status

eventual removal) already say what needs to be said. In which case, we can hide our live region from view, so that it is only available non-visually. Once again, we'd call on a screen reader accessible hiding method.[165]

- Deque offers a live region playground[166] for exploring a number of settings. If you're a Mac user, the quickest way to get started testing with this is to open it in Safari and turn on VoiceOver (*Cmd + F5*).

FOREGOING THE SUBMIT BUTTON

Now that we're not making use of our `get` functionality, we'll need to prevent the page refreshing by suppressing the submit button's default form submission functionality. We do this by catching the `submit` event.

```
var sortForm = document.querySelector('.sorter');
sortForm.addEventListener('submit', function(event) {

    // keep the browser from submitting the form,
    // because we're handling this with XHR
   event.preventDefault();

    // XHR request handling here
});
```

165 http://smashed.by/hidingcontent
166 http://smashed.by/contentfeedback

This begs the question whether we need the submit button at all. Perhaps, for greater immediacy, we could remove the submit button with JavaScript and attach the XHR functionality to the change event.

```
var sortForm = document.querySelector('.sorter');
sortForm.addEventListener('change', function(event) {
   if (event.target.type !== 'radio') {
      return;
   }

   this.submit();
}, true);
```

Since we've styled our radio controls to look like buttons and provided clear feedback via our loading symbol and live region, this seems like a fairly safe course of action. However, we must be mindful of the *give users control* principle mentioned at the beginning of this chapter. By removing an explicit submission action, it's possible that triggering the XHR will be unexpected to some users. They may feel their control has been usurped and this may diminish their trust in the interface.

In addition, note that keyboard users operating the widget must use their arrow keys to move through the radio options. Each arrow keypress not only focuses adjacent radio buttons but *selects them* as well.

This means moving through the options in either direction would fire the `change` event three times in total.

It's possible to limit this enhanced XHR-on-choosing-filter-option functionality to mouse and touch users by switching from using the `change` event to using `click`.

```
var sortForm = document.querySelector('.sorter');
sortForm.addEventListener('click', function(event) {
   if (event.target.type !== 'radio') {
      return;
   }

   this.submit();
}, true);
```

However, this would mean hiding the now redundant submit button and trusting that keyboard users would know to press Enter to submit the form. Note that you can't simply *remove* the submit button because some platforms (notably iOS) will not submit forms where a submit button isn't present. In which case, we'd have to use our special `.visually-hidden` class, plus `tabindex="-1"` to make sure the button isn't user-focusable:

```
<button type="submit" class="visually-hidden"
tabindex="-1">sort</button>
```

Even with these measures in place, whether users actually understand or *like* this enhancement is a question for user testing. Recruiting a diverse test group is the best way to properly confirm inclusive design decisions.

Include disabled participants as part of a wider user testing recruitment process. The numbers will be small, but aim to capture a range of disabilities and assistive technologies.

— Government Digital Services (UK)

The Government Service Design Manual has an informative section on accessibility testing[167] as a complement to standard user testing. For smaller budgets and timeframes, diverse users should be made part of the main test group.

Loading More Results

If we have a large inventory of products, some search terms and filtering options are likely to match a considerable number of items. Retrieving and rendering all of them at once would cause a serious performance bottleneck, as well as producing an intimidating and unwieldy page. Far better to load a smaller set of results, then retrieve more as and when the user needs them. I can think of a few different ways to do this. One of them — infinite scroll — has some serious problems regarding inclusivity.

167 http://smashed.by/a11ytesting

INFINITE SCROLL

The infinite scroll[168] pattern harnesses the user's scroll behavior to automatically load new content at the point that they reach the current content's end. The aim is to provide a stream of new content to the user without necessitating an action on their part. Unless implemented very carefully, infinite scroll tends to result in a frustrating experience for a number of different input modes.

As a mouse user, I might scroll the page by dragging the scrollbar's handle. When new results load, this handle will move upwards in accordance with the greater amount of content. Not realizing the handle has moved away from reach, I may click the scroll track instead and make the page lurch downwards from my current location: a hugely counterintuitive and annoying experience, wherein I'm liable to skip over unread content.

As Derek Featherstone writes,[169] infinite scrolling is also a frustrating experience for keyboard users. When items within the stream have interactive elements (such as our "Buy Now" buttons), focusable elements are added to the stream ad infinitum, making it impossible to tab past the main content to interactive elements that reside below it in the footer.

168 http://smashed.by/infinitescroll
169 http://simplyaccessible.com/article/infinite-scrolling/

Derek offers two solutions. He's so fond of one that he lists it twice:

1. Just don't implement infinite scrolling.
2. Replace automatic infinite scrolling with a "Load more results…" button or link that explicitly invites the user to add more. Once they do a few times, prompt them to ask if they'd like to turn auto-loading of more results on.
3. No, really, just don't implement infinite scrolling.

THE "LOAD MORE" BUTTON

The "Load more" button — and it *should* be a `<button>` — resides at the end of the current crop of results and clicking it sends an XHR request to load the next set.

```
  <li><!-- penultimate item --></li>
  <li><!-- last item --></li>
</ul>
<button data-load-more>Load more</button>
```

After the new content is rendered, it's important that keyboard focus is moved from the `data-load-more` button to the first of the newly loaded items. Otherwise, the user's view will not change and focus will remain on the `data-load-more` button which has now been pushed off-screen by the added content.

Note that most implementations *only* remove focus from the button, and don't move it elsewhere. This results in what's sometimes called *freak out mode* whereby the browser doesn't know what to focus and defaults to focusing the document body. The outcome is that keyboard (including screen reader) users are sent right to the start of the page content.

An effective course here is to switch focus to the title (`<h3>`) of the first newly returned product. This will place that product at the top of the viewport (which benefits all kinds of sighted users) and trigger the announcement of the `<h3>`'s text in screen readers, naming the first instance of the new content.

Note that `tabindex="-1"` has to be applied to the first item's `<h3>` to make it focusable with JavaScript's `focus()` method. Unlike `tabindex="0"`, elements are not focusable by keyboard users directly. In this case, user focus is not desirable because the `<h3>` is not an actionable (interactive) element.

We are only focusing the `<h3>` to place the user in the correct context and to trigger announcement. A press of the Tab key will take them to the artist link, and another press to the "Buy now" link.

```
<li>
   <h3 tabindex="-1"> <!-- make the title focusable -->
      Naked Man In Garage Forecourt
      <a href="/artist/kenny-mulbarton">by Kenny Mulbarton</a>
   </h3>
   <img src="/images/naked-forecourt-man.jpg" alt="high
contrast black and white image of a naked man nonchalantly
leaning against a petrol pump." />
   <dl>
      <dt>Size:</dt>
      <dd>30cm × 90cm</dd>
      <dt>Price:</dt>
      <dd>€35.95</dd>
      <dt>Rating:</dt>
      <dd><img src="/images/rating_4_5.svg" alt="">4 out of 5
stars</dd>
   </dl>
   <a href="/product/naked-man-in-garage" class="call-to-
action">
      Buy <span class="visually-hidden">Naked Man In Garage
Forecourt</span> Now
   </a>
</li>
<li>
   <!-- second newly returned product -->
</li>
<li>
   <!-- etc -->
</li>
```

As with choosing a filtering option, an XHR has to be handled. We should, therefore, supply a loading graphic and live region. In this case, we can replace the “Load more” button’s text node with that graphic. The order of events should be as follows:

1. The user clicks the “Load more” button.

2. The button’s text is changed to “Loading:” and is set to handle no further clicks. You can catch the click and simply `return` if the loading state is detected (for instance, by looking for “Loading” in the button’s `textContent`).

3. A hidden live region announces, “Loading more products.”

4. The XHR is handled.

5. On success, the content is rendered.

6. The live region announces, “Products loaded.”

7. Focus is moved from the button to the first of the newly loaded product items. A smooth scrolling action could be used here, as described in “Navigation Regions.”

8. The “Load more” button has its original text node reinstated and `click` events are handled again.

load more
loading

The "Load more" button in its initial state (left) and after being pressed (right).

Infinite scroll hijacks the user's scrolling action to perform an unexpected behavior, commandeering user control and diminishing the user experience. The "Load more" button invites the user to take an explicit, labeled action at their convenience and therefore conforms to the second of Henny Swan's UX principles, to *give users control*.

Display Options

Typically, designers see it as their job to foresee their user's needs and make intelligent decisions on their behalf. That's what design is, right? Perhaps, but we have to contend with:

- Users having different preferences.
- Users encountering different circumstances.

One example of making a decision on users' behalf is to disable zoom, using `user-scalable=no` on the viewport `<meta>` tag. This is to decide *for* the user what font size and magnification level suits them, and prevents them from adjusting it themselves. An audacious act, and foolhardy given the inevitable diversity of your audience.

We still have to make some decisions in isolation, of course, because not every part of a design can be informed directly by user research. But when we do, we can increase our confidence by deferring to:

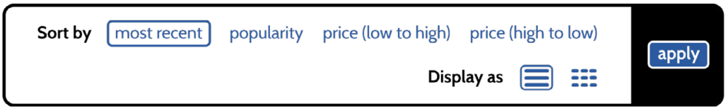
- **Convention**: Using widely adopted patterns, motifs or language.
- **Choice**: Allowing the user to decide how they want to consume something.

To complete our filtering widget pattern, we're going to give our users a choice over the way the filtered content is displayed.

LIST OR GRID?

In this final enhancement to our filtering interface, we'll give the user the choice between displaying results in list or grid format. That way, they can select a visual display that best suits their cognitive needs. A list is a simple format, but necessarily longer vertically; a grid gives a better overview but compresses more information into the viewport at once.

The adapted filter widget might look like the following:



The submit button now reads "apply:" a broader action which relates to both the sorting and display settings.

Here's the extended markup:

```
<form role="form" class="sorter" method="get">
   <fieldset>
      <legend>Sort by</legend>
      <input type="radio" name="sort-method" id="most-recent"
checked>
      <label for="most-recent">most recent</label>
      <input type="radio" name="sort-method" id="popularity">
      <label for="popularity">popularity</label>
      <input type="radio" name="sort-method" id="price-low-
high">
      <label for="price-low-high">price (low to high)</label>
      <input type="radio" name="sort-method" id="price-high-
low">
      <label for="price-high-low">price (high to low)</label>
   </fieldset>
   <fieldset>
      <legend>Display as</legend>
      <label for="list">
         <svg>
            <use xlink:href="#list-icon"></use>
            <text class="visually-hidden">a list</text>
         </svg>
      </label>
      <input type="radio" name="display-as" id="list"
value="list" checked>
      <label for="grid">
         <svg>
            <use xlink:href="#grid-icon"></use>
            <text class="visually-hidden">a grid</text>
         </svg>
      </label>
      <input type="radio" name="display-as" id="grid"
value="grid">
   </fieldset>
   <button type="submit">apply</button>
</form>
```

NOTES

- The "Display as" functionality is in its own `<fieldset>` to demarcate it from the principal filtering options.
- The list and grid icons are provided as inline SVGs, with visually hidden `<text>` elements providing each radio button's screen reader accessible label text.
- Focusing the selected display option radio button would announce the `<legend>` first, then the `<label>` and additional information in screen readers. So, if the list item is focused (and selected) it would read, "Display as a list, radio button, selected" or similar.

- I've changed the submit button label from "sort" to "apply" so that it *applies* to both settings, if you'll excuse the wordplay.

A SELF-GOVERNING GRID

Whether grid or list in terms of layout, the products should always be formed as a list (`<ul>`) in the markup. Users not actually *seeing* a list benefit from list semantics regardless. This also saves on client-side DOM manipulation: all we should need to change is a class on the parent `<ul>` item: `.list-display` for a single-column layout or `.grid-display` for multiple columns.

The question is: "How many columns is the right number?" In responsive design, the answer depends on how much space there is. We typically proceed to match column numbers to viewport widths — the wider the viewport, the greater the number of columns we can afford. This necessitates a lot of manual media query writing and an ever-watchful eye for layout problems.

Using Flexbox, we can instead use `flex-basis` to define an ideal width at an element level. By switching on `flex-grow` and `flex-shrink`, the grid elements can expand and collapse around this ideal, preserving a complete and orderly grid across an infinite range of viewports.

```
.grid-display {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}

.grid-display li {
  flex-grow: 1;
  flex-shrink: 1; /* the default value, in fact */
  flex-basis: 10em;
}
```

NOTES

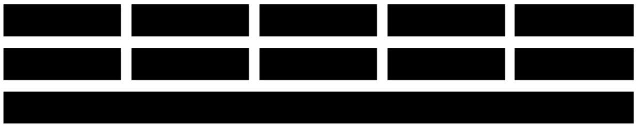
- The `flex-basis` value is key. This is the width the flexbox algorithm considers ideal for individual items. Each item will try to be 10em wide but will *flex* to share the available space completely (see `flex-grow` and `flex-shrink` below).
- The `flex-basis` value is set in `ems` so that it is relative to font size. This means that the automatic reflow of columns that we've established is sensitive to font size.
- The `flex-grow` value of `1` means items will grow beyond the `10em` width to fill the available space.
- The `flex-shrink` value of `1` (can be omitted, in fact, because it's the default) ensures items will shrink if needed.

- `flex-wrap` is switched on so items will wrap to the next row when there is not enough room to fit them alongside other items greater than 10em in width.

ENSURING ACCEPTABLE MEASURE

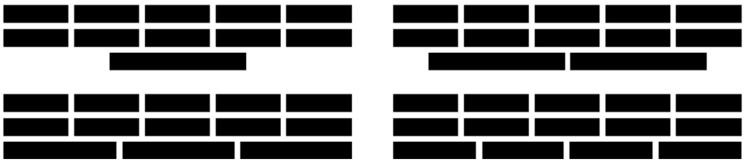
What we've established is a self-governing grid system. With very little code, we are able to ensure our content remains unbroken across an *infinite* range of viewport widths. That's a highly inclusive layout strategy.

All that's left is to ensure the content remains readable. If you remember from "A Paragraph", we shouldn't allow the measure to get too wide. That's currently a danger at wider viewports, especially when the wrapping algorithm places the final item on a single line.



In the adapted code example to follow, items are restricted to having a `max-width` of `20em`. My preference, for symmetry, is to group the content around the center line, hence the `justify-content: center;` declaration on the flex container.

```
.grid-display {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: center;
}

.grid-display li {
  flex-grow: 1;
  flex-basis: 10em;
  max-width: 20em;
}
```



Now the grid items are organized around the vertical center line and cannot become more than 20em wide.

Flexbox works algorithmically, which is what makes it so powerful. To get the best results for your own content, you'll need to tweak the `10em` and `20em` values.

(**Note:** IE10 and IE11 have an unpleasant bug whereby the children of focusable items within a flex container can become focusable themselves. The handy *a11y.js* library has a fix for this.)[170]

170 http://allyjs.io/api/fix/pointer-focus-children.html

RIGHT-TO-LEFT GRIDS

As we established in the chapter “The Document,” it’s important to declare the language of a web page because this gets the most out of assistive technologies and translation tools. An international audience is a big step for inclusion. Some languages, like Arabic, are supposed to be read right to left, unlike languages like English that are read left to right. Setting the text direction for the document is possible by including the `dir` attribute (alongside the `lang` attribute) on the HTML element.

```
<html lang="ar" dir="rtl">
```

Float-based CSS layouts are unaffected by this provision. That is, you have to manually reverse the layout of floated content.

```
.content {
  float: left;
  width: 60%;
}

[dir="rtl"] .content {
  float: right;
}

.sidebar {
  float: right;
  width: 40%;
}
```

```
[dir="rtl"] .sidebar {
  float: left;
}
```

Flexbox, rather beautifully, handles direction automatically. Any flex container inside a document with `dir` set to `rtl` will display each row of content from right to left as expected: it switches the `row` and `row-reverse` values of the `flex-direction` property.

Should you, despite the right-to-left direction setting, want your grid to be laid out in a left-to-right configuration, you can force this using the CSS `direction` property on the flex container:

```
.grid-display {
  display: flex;
  direction: ltr;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: center;
}
```

(*Note*: This automatic reversal is also true of <`table`> elements which switch their column order. We don't use <`table`> elements for layout because they provide incorrect semantic information to assistive technologies.)

Tolerating Dynamic Content

As ever, we should be mindful of the content that our proposed structure is designed for. Since we've covered wording and tone in previous chapters, let's take some time to consider fluctuating quantity.

One of the main reasons static mockups of interface components are defunct is that they tend to represent idealized content: people's names of a particular length, and descriptions all exactly five lines high, for instance. When the front-end is built, problems can occur where variable content has an unexpected effect on layout.

To make sure our product's grid interface can tolerate fluctuating content length at the prototyping stage, we can make use of a tool like *forceFeed.js*.[171] This script allows you to feed the proposed layout with randomized arrays of content within certain parameters.

For example, to test the interface's tolerance of different names, I might add the following *forceFeed.js* `data-forcefeed` attribute to the artist link's <`cite`>:

```
<a href="/artist/kenny-mulbarton">by <cite data-
forcefeed="words|2">Kenny Mulbarton</cite></a>
```

171 https://github.com/Heydon/forceFeed

The `words` parameter refers to an array of words of different lengths. The `2` parameter refers to the number of randomized words I want to include. Since most names are composed of two words — a forename and surname — two seems reasonable. To run this over all of the artist <`cite`>s in a page, I first include *forceFeed.js*, then script the following.

Note that I'm using a short set of *lorem ipsum* words just for brevity, but you can populate your array how you like.

```
window.words = ['lorem', 'ipsum', 'dolor', 'sit', 'amet',
'adipsing', 'consectetur', 'elit', 'sed', 'commodo',
'ligula', 'vitae', 'mollis', 'pellentesque', 'condimentum',
'sollicitudin', 'fermentum', 'enim', 'tincidunt'];

var cites = document.querySelectorAll('cite');

[].forEach.call(cites, function(cite) {
  cite.addAttribute('data-forcefeed', 'words|2');
});
forceFeed({words: window.words});
```

Testing the tolerance of these word combinations is a simple case of reloading the page repeatedly to see whether any randomized combinations of words appear to break the layout, by causing an ugly wrapping behavior, for example.

THE PRODUCT TITLE

There's another dimension to *forceFeed.js*. The script also allows you to add *between x* and *n* numbers of array items. This helps to test dynamic content for the title of the print.

Note that *forceFeed.js* needs a wrapper element to work, so let's pretend we've used a script to place a `<span>` around the title before the link.

```
<h3 tabindex="-1"> <!-- make the title focusable -->
   <span data-forcefeed="words|1|10">Naked Man In Garage
Forecourt</span>
   <a href="/artist/kenny-mulbarton">by <cite data-
forcefeed="words|1">Kenny Mulbarton</cite></a>
</h3>
```

When two numbers are provided, as in `words|1|10`, the script will place *between* the first number and second number of array items as the text node. Between one and ten words is the kind of variability we can expect for the titles.

Now it's back to refreshing the page and adapting the grid to tolerate any visual breakages better. Usually problems arise due to wrapping causing height differences between items, or failures to wrap meaning content breaks out of its box.

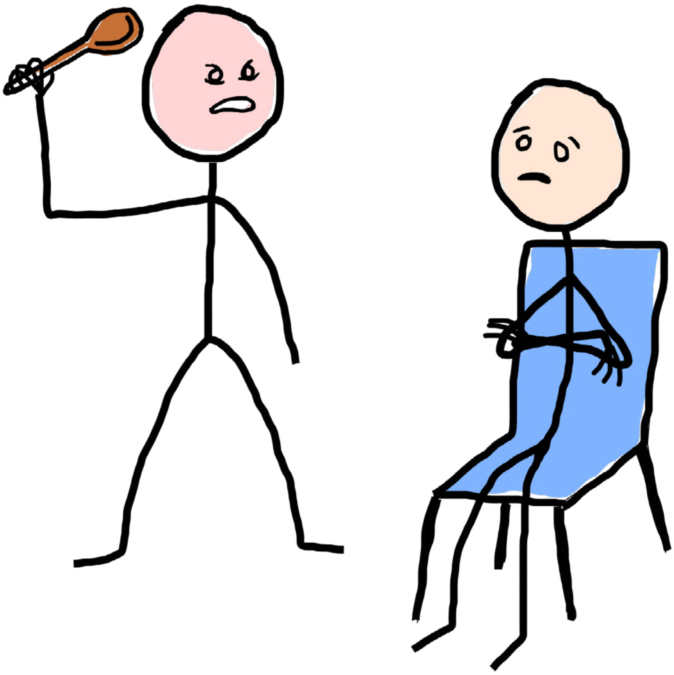
Summary

In this pattern we discovered two things. First, sometimes HTML already provides the behaviors we often find ourselves needlessly and flimsily recreating in JavaScript. Second, when using semantic HTML, CSS can be used as a form of progressive enhancement. In addition, we explored the importance of giving user's *choice* and *control* over how their content is arranged.

We also used some techniques to make sure our design was tolerant of dynamic and fluctuating content. As we've been discussing throughout the book, inclusive design also means a visual design which is not too strict about the nature of the content imparted to it.

THINGS TO AVOID

- Creating behaviors in JavaScript that HTML already offers.
- Grid systems which don't allow right-to-left language support.
- Infinite scrolling.
- Prototyping with idealized content.



“Your banking interface is terrible!”
“Sir, we make spoons. That’s a wooden spoon.”
“Well, that was your first mistake!”

A Registration Form

My first forays into web design at the turn of the century revolved around images, image maps to link parts of those images to other pages of images, and nested `<table>` elements to hold the whole debacle together. Underperforming, unresponsive, not cross-browser compatible: just some of the things you could condemn my early handiwork for.

My current output is hardly perfect, but it has benefited from a change in mindset; an acknowledgement that web pages aren't merely pinboards for images. Really, web pages are terminals: they accept information (input) and display information (output).

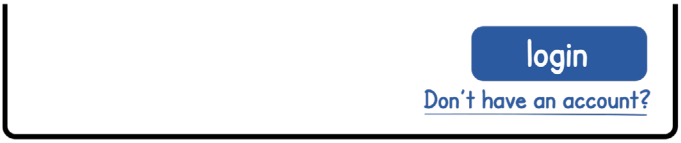
Web forms take care of the input part. Making sure forms are inclusive is paramount, because all people should be able to contribute to and not just consume the web. So, what are we dealing with? It's a mixed bag: form elements piggyback OS functionality and are keyboard and screen reader accessible as standard — at least when marked up correctly. So that's good. On the other hand, forms are a 10,000-volt electromagnet for attracting usability problems.

For this simple registration form pattern, we shall embrace standards while attempting to tiptoe nimbly around potential usability failures. The relationship between standards and usability is where we find inclusion.

The Form In Context

Before diving into constructing the registration form itself, it's worth acknowledging the context in which we are likely to find it. Despite the virtues of modularity, it can't be helped that some patterns are informed by context and are better designed where context is considered first.

Conventionally, registration forms are counterparts to login forms provided for users who already have accounts. In fact, it's usually the login form that you would encounter first, with an option to register instead. This registration option usually takes the form of some tiny link text reading "Don't have an account?" and placed *after* the login form's submit button.



The thinking here is often that the no account option doesn't need to be a big deal because *most* new users will have gone directly to the homepage, read the entire sales pitch for the application and pressed the sign-up call-to-action there.

Making assumptions about the way visitors enter and move through your app is dangerous territory anyway, but for certain users the usability here is even more aggravating. Screen reader operators traversing the page from top to bottom methodically will not realize the option to register is present unless they go past the login form. In addition, by linking to the registration form on a separate page, they'd have to make their way down to the form from the top of that newly loaded page.

This is one of the reasons we create the ability to bypass blocks of content using skip links, headings and landmark regions, as discussed in "A Blog Post." This comes under WCAG's *2.4.1 Bypass Blocks*[172] success criterion.

However, a clearer solution for *all* users would be to present them with a choice of login or registration at the outset. In the illustration below, "Login" is selected by default but in a clear relationship with the unselected register option.

Login
Register

172 https://www.w3.org/TR/WCAG20/#navigation-mechanisms

THE LOGIN/REGISTER TOOLBAR

It's important the choice being presented to the user is clear both visually and non-visually. But despite what certain vocal accessibility pundits might have you believe, there's really no right or wrong way to achieve this kind of thing. It's down to your discretion as a designer to find the most effective solution.

One option would be to design the login and register options as tabs in a fully-fledged WAI-ARIA tab interface. I talk about such interfaces at length in the book "Apps For All: Coding Accessible Web Applications."[173] Also available is an accompanying demo.[174]

Since we are only dealing with two options (tabs), I think this complex widget is overkill. A simpler way to communicate the choice and the current selection would be to construct a little `toolbar`,[175] like the following.

```
<h1>Welcome</h1>
<div role="toolbar" aria-label="Login or register">
   <button aria-pressed="true">Login</button>
   <button aria-pressed="false">Register</button>
</div>
```

173http://smashed.by/apps4all
174 http://smashed.by/tab-interface
175http://w3c.github.io/aria/aria/aria.html#toolbar

```
<div id="forms">
  <div id="login">
    <form>
            <!-- the login form -->
    </form>
  </div>
  <div id="register">
    <form>
            <!-- the registration form -->
    </form>
  </div>
</div>
```

NOTES

- Pressing a button changes that button to `aria-pressed="true"`[176] and reveals the corresponding form. A CSS style should be provided to show that this button is the selected one, possibly with the `[aria-pressed="true"]` attribute selector.

- When screen reader users focus the first button, “Login or register toolbar, login toggle button, selected” (or similar, depending on the screen reader) is announced. This informs users that they are interacting with a toolbar widget presenting them with a choice of “Login or register” and that the “login” option is currently activated.

176 https://www.w3.org/TR/wai-aria-1.1/#aria-pressed

- The displayed form (either login or register) is in focus order following the toolbar so is easily reached by keyboard or screen reader controls. No explicit relationship between the toolbar and forms area is therefore necessary. As previously discussed, `aria-controls` can provide an explicit relationship but it should not be relied upon because of low support. Source order is your friend in these situations.

The Basic Form

Now let's shift our focus to just the registration form itself. We've already touched on harnessing form controls in "A Filter Widget." This form takes a more familiar, well, *form*, facilitating text input by the user.

```html
<form id="register">
   <label for="email">Your email address</label>
   <input type="text" id="email" name="email">
   <label for="username">Choose a username</label>
   <input type="text" id="username" name="username"
placeholder="e.g. HotStuff666">
   <label for="password">Choose a password</label>
   <input type="password" id="password" name="password">
   <button type="submit">Register</button>
</form>
```

LABELING

In Inclusive Forms 101, we must ensure that all interactive elements have an accessible label associated with them. In the case of the submit button, the text node is the accessible label. That is, when you focus the button, "Register" is announced as the label.

For elements that accept user input like text inputs, an auxiliary label must be associated with it. The standard way to achieve this is by using a `<label>` element, which takes a `for` attribute. The `for` attribute associates the label with an input using its `id` value. This is one of the reasons you should ensure `id`s are unique. When they are not, you would fail WCAG's *4.1.1 Parsing*[177] rule.

In case of the password input, the agreed cipher — the matching content of the `for` and `id` values — is simply "password":

```
<label for="password">Choose a password</label>
<input type="password" id="password" name="password">
```

To appreciate why labels have to be explicitly associated with controls in this way, you have to appreciate how screen reader operators traverse forms. Unlike prose content, where users may use the down arrow to go from element to element, forms are operated by moving directly between

177 https://www.w3.org/TR/WCAG20/#ensure-compat

one field and the next. Label elements, therefore, are jumped over: if they weren't explicitly connected to the interactive elements they were describing, they'd be missed.

In my example, when a screen reader user focuses the password field, "Choose a password, secure input" (or similar) would be announced.

Senior accessibility engineer Léonie Watson notes that a role of `password` is being mooted[178] for the WAI-ARIA (2.0) specification. This would enable developers to communicate the security of a custom field without necessarily ensuring that security (masking the inputted characters) is actually present. WAI-ARIA only affects semantics, not general behavior. This is why standard elements and attributes such as `type="password"` are a safer choice where available. They pair semantics with standard behaviors automatically.

THE PLACEHOLDER ATTRIBUTE

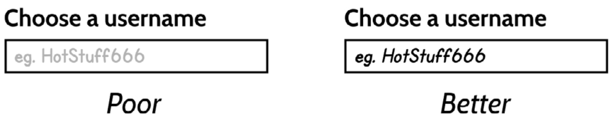
The `placeholder` attribute is a relatively new addition to the HTML specification. It was created in response to developers wanting to give hints for the type of content the user should provide. The key word here is *hints*: the `placeholder` is not a labeling method by itself and should only be used to provide supplemental information.

178 http://tink.uk/proposed-aria-password-role/

In the `#username` example, “Choose a username” is the (accessible) label and “e.g. HotStuff666” is provided just to get the user’s imagination kick-started.

```
<label for="username">Choose a username</label>
<input type="text" id="username" name="username"
placeholder="e.g. HotStuff666">
```

By default, the `placeholder` attribute is shown as gray text, which can cause contrast issues, especially where the input uses a background color. Instead of differentiating the placeholder by diminishing its contrast, I suggest using a different method, such as italicization.



The left example shows the placeholder text in default gray. The right uses a custom italic style with a darker color.

Styling the placeholder is possible in most browsers with the help of some standard and proprietary properties:

```
::placeholder {
  color: #000;
  font-style: italic;
}
```

```
::-webkit-input-placeholder {
  color: #000;
  font-style: italic;
}

::-moz-placeholder {
  color: #000;
  font-style: italic;
}
```

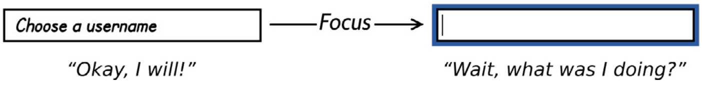
(**Note:** Each rule is in a separate declaration block rather than a comma-delimited list. Browsers will not parse a combined block if it contains unrecognized, proprietary selectors.)

Some screen readers and browsers do not support `placeholder`, so using it to supplant a proper label will result in missing information. But that's not the only problem: any sighted users interacting with the input will find the surrogate label disappears as soon as they start typing. This produces cognitive accessibility issues and is broadly uninclusive as an interface pattern. The browser's autocompletion routine could also populate multiple fields, of course. This too would eradicate visible labels and make it difficult for the user to check the automatic values against what the fields intended.

Using placeholder as a label is esp. bad when combined with autofill. Just had a whole form autofilled and I have no clue what anything is.

— Lea Verou, on Twitter[179]

Sometimes it's tempting to remove visible labels and rely on `placeholder` attributes to save space (or *screen estate* if you prefer) but that's hardly a good reason to diminish the usability of a form.



Left unfocused text field shows the placeholder. Right focused text field does not, leaving the user to ponder what it was for.

You should prioritize making room for the label from the outset, in mockups and prototypes. However, there exists an innovative solution which uses the label *as* the placeholder. The float label pattern[180] by Matt D Smith[181] animates the label out of the way of the user's cursor on focus. Be aware, though, that since it treats the label and placeholder as one and the same, there is no opportunity to supply a hint or example.

179 https://twitter.com/LeaVerou/status/758386597012185088
180 http://bradfrost.com/blog/post/float-label-pattern/
181 https://twitter.com/mds

A NOTE ON GROUPING

The <fieldset> element is proffered for grouping related form fields under a common label: a <legend>. In most screen readers, this results in the <legend>'s content being concatenated with each field's <label>. We *could* adapt our form to include this grouping mechanism like so:

```
<form id="register">
    <fieldset>
        <legend>Registration</legend>
        <label for="email">Your email address</label>
        <input type="email" id="email" name="email">
        <label for="username">Choose a username</label>
<input type="text" id="username" name="username"
placeholder="e.g. HotStuff666">
      <label for="password">Choose a password</label>
      <input type="password" id="password" name="password">
      <button type="submit">Register</button>
   </fieldset>
</form>
```

This is technically valid, but it creates a lot of unnecessary noise; that is, focusing each input in turn would announce the extended labels of "Registration: Your email address," "Registration: Choose a username," and so on.

When programming code day in and day out, we habituate ourselves to thinking of things as right or wrong, true or false. That a <fieldset> in this context is not technically invalid or nonconforming might lead us to think it is the opposite: virtuous, correct — *mandatory* even.

HTML does not hold fast to the clear, procedural logic of imperative programming languages, so it's better not to think of them as similar. Structures of HTML, like the structures of natural language it annotates, should be judged in similarly nuanced terms: what helps or hinders; what's too little or too much. In the current context, the `<fieldset>` and `<legend>` create too much verbosity for little to no gain, so probably shouldn't be there.

Bearing in mind that `<fieldset>`s are pointless without `<legend>`s, you can use the following three rules of thumb to decide if a `<fieldset>` is appropriate.

1. Is there more than one distinct set of fields in total, in the same form or context? *Yes*? Use `<fieldset>`s. *No*? Don't use `<fieldset>`s.

2. Does a set actually only have one field in it? *Yes*? You don't need a `<fieldset>`. *No*? Use a `<fieldset>` if (1) applies.

3. Can you think of a `<legend>` that would make sense or aid comprehension if used with each of the `<fieldset>`'s field labels? *Yes*? Use a `<fieldset>`. *No*? Don't use a `<fieldset>`.

Required Fields

Our register form has some required fields, which cannot be left blank. In fact, *all* the fields are required: for the sake of a simple UX we're not asking anything which isn't completely critical to signing up the user.

Denoting required fields inclusively is a mixture of standards and convention. For many people, an asterisk (`&#x002a;`) character suffixing the field label is familiar. I can place it in a `<strong>` element for the purposes of making it red, if I think that will increase its comprehensibility:

```
<label for="email">Your email address <strong class="red">*</
strong></label>
<input type="text" id="email" name="email">
```

For screen reader users, the label is announced as usual, including the asterisk, as "Your email address asterisk." The term "asterisk" in this context is well enough understood by screen reader users to mean "required," but it's not exactly robust. Imagine we were marking up a quiz question about a certain Gaulish cartoon character:[182]

```
<label for="quiz-question">What is the name of Goscinny and
Uderzo's famous cartoon Gaul?<strong class="red"*</strong>
</label>
<input type="text" id="quiz-question" name="quiz-question">
```

182 https://en.wikipedia.org/wiki/Asterix

More correctly, placing `aria-required="true"`[183] on the input itself will announce “Required” in the set language of the page. This just leaves us the job of silencing the asterisk, for which we can use `aria-hidden`. You can think of `aria-hidden="true"` as the aural equivalent of `display: none;`.

```
<label for="email">Your email address <strong class="red"
aria-hidden="true">*</strong></label>
<input type="text" id="email" name="email" aria-
required="true">
```

(**Note:** Both `aria-hidden="true"` and `aria-required="true"` have explicit values rather than being written in the Boolean form that is possible with certain HTML5 attributes.[184] This is correct for ARIA attributes and significantly more reliable.)

A NOTE ON THE REQUIRED ATTRIBUTE

You may be aware that there is an HTML5 `required` attribute. Why aren’t we using this? Usually it is better to use the HTML5 base semantics rather than the WAI-ARIA extension, but only if vendor (read: browser and screen reader) support is acceptable.

183 http://w3c.github.io/aria/aria/aria.html#aria-required
184 http://smashed.by/html5forms

The `required` attribute is not implemented uniformly[185] across browsers. It also tends to invoke an undesirable feature: marking empty required fields as invalid from the outset. For our purposes this is rather verbose and aggressive.

Showing The Password

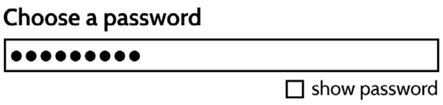
You'll recall the discussion on the cognitive stress of disappearing `placeholder` attributes, especially when they are unaccompanied by proper labels. Having to type a password without being able to see if you've hit the right keys is a matter of similar discomfort.

Some interfaces provide an auxiliary field, similar in all respects but its `id`, and implore you to type your proposed password a *second* time, so the system can compare the two. Bothersome and off-putting.

To preserve security but give users the option to check their entered password, instead we shall provide a checkbox to reveal the password at will.

```
<label for="password">Choose a password</label>
<input type="password" id="password" name="password">
<label><input type="checkbox" id="showPassword"> show
password</label>
```

185 http://caniuse.com/#feat=form-validation



The form with a "show password" checkbox to its bottom-right.

A short script simply toggles the field type from `password` to `text` and back again:

```
var password = document.getElementById('password');
var showPassword = document.getElementById('showPassword');

showPassword.addEventListener('change', function() {
  var type = this.checked ? 'text' : 'password';
  password.setAttribute('type', type);
});
```

Note that Internet Explorer and Microsoft Edge provide this functionality natively, using an interactive eye icon associated with the `::ms-reveal` pseudo-class.[186] Since we've provided our own (cross-browser) solution, it would be wise to suppress this feature:

```
input[type=password]::-ms-reveal {
  display: none;
}
```

186 https://developer.mozilla.org/en-US/docs/Web/CSS/::-ms-reveal

At this stage, it's become a habit of ours, as inclusive designers, to not only improve an interface's UX but to do so in a way that supports users in unusual circumstances or using auxiliary software. This little, screen reader accessible show-password provision is another reusable micropattern. We only have to think about and work on the accessibility aspect once.

Validation

Our biggest challenge is to provide an inclusive form validation experience. This involves a lot of moving parts and, without care, can produce a jarring and unusable experience for some users. The trick is in communicating two key messages during validation as separate concerns:

1. Something is broken (the form has errors).
2. What needs fixing (what will make the form valid).

As with the HTML5 `required` attribute, there are concerns about the support and uniformity of HTML5 form validation based on browser's interpretation of input types like `url`. There is also no functionality within HTML5 to validate a password field. In which case, we are going to rely on WAI-ARIA to indicate invalidity, with JavaScript for the actual validation (pattern matching).

SOMETHING IS BROKEN

When the user tries to submit the form, we need to check if there are any errors. If there are, we need to suppress form submission temporarily. Note that we are suppressing the submission of the form itself rather than merely the click handler on the submit button.

```
var form = document.getElementById('register');
form.addEventListener('submit', function(event) {
   if (errors) {
      event.preventDefault(); // do not submit
   }
});
```

This is all well and good, but literally nothing currently happens when the user tries to submit (where form errors are present). This is our cue to provide some feedback in the form of an error message. At this point, all we want to communicate is the presence of errors and that they need attention. A simple live region that is populated by the error message on an attempted submission will suffice.

Following is the initial markup. Note that I am placing the region *directly above* the submit button. Since the submit button is where the user is looking and working, this does the best job of bringing the error to their attention. Remember that users can (and will!) zoom their content. Therefore, an error which appears above the form may not even appear within the current viewport.

```
<div id="error" aria-live="assertive" role="alert"></div>
<button type="submit">Register</button>
```

Here's how we populate the live region using our script:

```
var form = document.getElementById('register');
form.addEventListener('submit ', function(event) {
   if (errors) {
      event.preventDefault(); // do not submit
      document.getElementById('error').textContent = 'Please 
make sure all your registration information is correct.'
   }
});
```

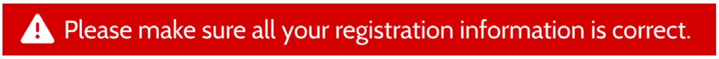
VISUAL DESIGN

The `#error` live region should only be visible when it is populated. To ensure that the empty box is not visible in its initial state (and does not otherwise affect the form's layout) you can employ the `:empty` pseudo-class:

```
#error:empty {
   display: none;
}
```

Conventionally, errors are displayed in red, so it's advisable to give the message box a red border or background. However, you should be wary of red being the only visual characteristic that classifies the message as an error. To support at the same time screen reader users and people who cannot see color, we can prepend a warning icon containing alternative text.

```
<div id="error" aria-live="assertive" role="alert">
  <p>
    <svg role="img" aria-label="error:">
      <use xlink:href="#error"></use>
    </svg>
    Please make sure your registration information is correct.
  </p>
</div>
```


Please make sure all your registration information is correct.

A red error message with white text prefixed with a triangular warning sign containing an exclamation mark.

When the form's submission event is suppressed, the live region is populated, switching its display state from `none` to `block` thanks to the `:empty` pseudo-class becoming inapplicable. This population of DOM content simultaneously triggers screen readers to announce the content, including the prepended alternative text: "error: Please make sure your registration information is correct." Note that the ARIA role of `img`[187] and `aria-label` force the <`svg`> element to behave like a standard <`img`/> element carrying an `alt` attribute with the content "error:".

The advantage of declaring the presence of errors using a live region is that we don't have to move the user to bring

187 https://www.w3.org/TR/wai-aria-1.1/#img

this information to their attention. Typically, form errors alert users by focusing the first invalid form field. This unexpected and unsolicited shift of position within the application risks disorientating users. In our case, the user remains focused on the submit button and is free to move back into the form to fix the errors when ready.

WHAT NEEDS FIXING

Now we can safely move on to handling the invalid fields. Each one needs two pieces of information, available visually and non-visually:

1. That the field is invalid.
2. What would make the field valid.

Except for the wording of error descriptions, the pattern will be the same for each invalid input, so we'll just use the password field as an exemplar. For (1) we can deploy the well-supported `aria-invalid` attribute.[188]

```
<label for="password">Choose a password</label>
<input type="text" id="password" name="password"
aria-invalid="true">
<label><input type="checkbox" id="showPassword">
show password</label>
```

188 http://smashed.by/invalidattr

That's about it, really. When the user moves back into the form and focuses the input it will now announce "Invalid" (or similar) in screen readers.

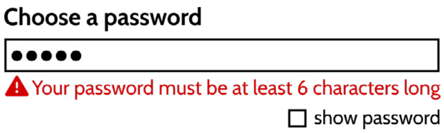
You can harness the `aria-invalid` attribute to provide a visual indication too. By linking the parsable state of `aria-invalid="true"` to a visual style, you save yourself the bother of managing style and state as separate concerns. Usually a separation of concerns is beneficial, but whenever a field is marked as invalid, that's when it should *look* invalid. Using the attribute selector for the state makes sure the form and underlying function of your interface don't get out of sync.

```
[aria-invalid="true"] {
  border-color: red;
  background: url () center right;
}
```

(Note the inclusion of a background icon for color-blind users who cannot perceive the red `border-color` change.)

Just knowing that the field is invalid is little use unless the user knows how to fix it too. For this purpose we can provide an accompanying description. In this case, the password is not acceptable because it is fewer than six characters long.

```
<label for="password">Choose a password</label>
<input type="text" id="password" aria-invalid="true" aria-
describedby="password-hint">
<div id="password-hint">Your password must be at least 6
characters long</div>
<label><input type="checkbox" id="showPassword"> show
password</label>
```


Choose a password

●●●●●

Your password must be at least 6 characters long

show password

The password input with error message directly below in red text, prefixed by a warning symbol.

The `#password-hint` element is connected to the input using the `aria-describedby`[189] attribute and the `password-hint` `id` as a cipher. That is, the description is connected much like the label. The only difference is in terms of order: the description is read last. Now, when a screen reader user focuses the input, this accessible description will be read out after the label, current value, input type and (invalid) state information. All the pieces are in place.

189 https://www.w3.org/TR/wai-aria-1.1/#aria-describedby

RERUNNING THE ROUTINE

Some fancy form validation scripts give you live feedback as you type your text entries, letting you know whether what you type is valid *as* you type it. This can become very difficult to manage. For entries requiring a certain number of characters, the first few keystrokes are *always* going to constitute an invalid entry. So, when do we send feedback to the user and how frequently?

Not wanting to be the overbearing waiter continually interrupting customers to check in with them, we didn't flag errors on first run. Instead, only when errors are present *after* attempted submission do we begin informing the user.

Once the user is actively engaged in *correcting* errors, I think it helpful to reward their efforts as they work. For fields now marked invalid, we could run our validation routine on each `input` event, switching `aria-invalid` from `false` to `true` where applicable.

```
var password = document.getElementById('password');
password.addEventListener('input', function() {
  validate(this);
});
```

DEBOUNCING

For users who type quickly, the above `validate()` function will fire very frequently. Not only will this produce some performance issues, but in implementations where a live region is populated and repopulated to declare invalidity, you put screen readers into 80's remix mode: "Y—your—y—your pass—your password must be at least 6 characters long."

We want to make sure the `validate()` function is only called when the user is idle. By debouncing, we divide clusters of keypress events into blocks and only execute `validate()` once per block.

Lodash's .`debounce`[190] utility accepts a `wait` parameter. In our case, we need to set this to be slightly shorter than what we imagine is the average interval between keystrokes. When debouncing, the single function execution can take place at the beginning (`leading`) or end (`trailing`) of the block. We need the validation to happen only when the user has stopped typing. Accordingly, in the `options` object, `options.leading` should be `false` and `options.trailing` `true`.

```
var password = document.getElementById('password');
var handleInput = _.debounce(validate.bind(null, this), 150,
{
   leading: false,
```

190 https://lodash.com/docs#debounce

```
   trailing: true,
});
password.addEventListener('input', handleInput);
```

Whether you actually want to implement a live region per field for feedback is up to you. Certainly, without one it won't be immediately obvious when the field has become valid or invalid. However, since the `aria-invalid` attribute will have been set to `true` regardless, when the user blurs and refocuses the field, they will be told it is valid. Most screen readers also provide shortcuts to reannounce element information. NVDA, for example, will speak the focused password input when pressing *Insert* + *Tab*.

Microcopy

As ever, we need to be mindful not just of the code and the visual design but our choice of words. In forms, where labels and instructions mean the difference between succeeding to complete a form and failing, we need to take extra care — especially if the site we're developing provides a crucial service, such as registering to vote.

The article "Five Ways To Prevent Bad Microcopy"[191] by Bill Beard offers some key pointers:

191 http://smashed.by/bad-microcopy

1. Get out your own head and get to know the user.
2. The user is a person. Talk to them like one.
3. Use copy as a guide, not a crutch.
4. Treat every moment like a branding moment, even when it's not.
5. If content is king, then treat context like a queen.

All good advice, and I recommend you read the article in full. But Id like to emphasize Bill's stipulation regarding point four:

> *Your brand's tone and voice are essential to consider when writing all of your copy, but it* ***should not get in the way*** *of a user who is trying to take action."*

In other words, don't put your brand before your *usability*. In an official capacity, WCAG echoes Bill's sentiments with the *2.4.6 Headings and Labels*[192] success criterion, imploring that "headings and labels describe topic or purpose." Branding whimsy in your copy can diverge too much from transparent wording, confusing and infuriating the user.

Take the password label in our registration form, for example. If our form is for an online fantasy role-playing game, it might be tempting to relabel the field "Secret incantation," "Spell of entry," or maybe "Cryptic charm"; all more

192 http://smashed.by/topic-purpose

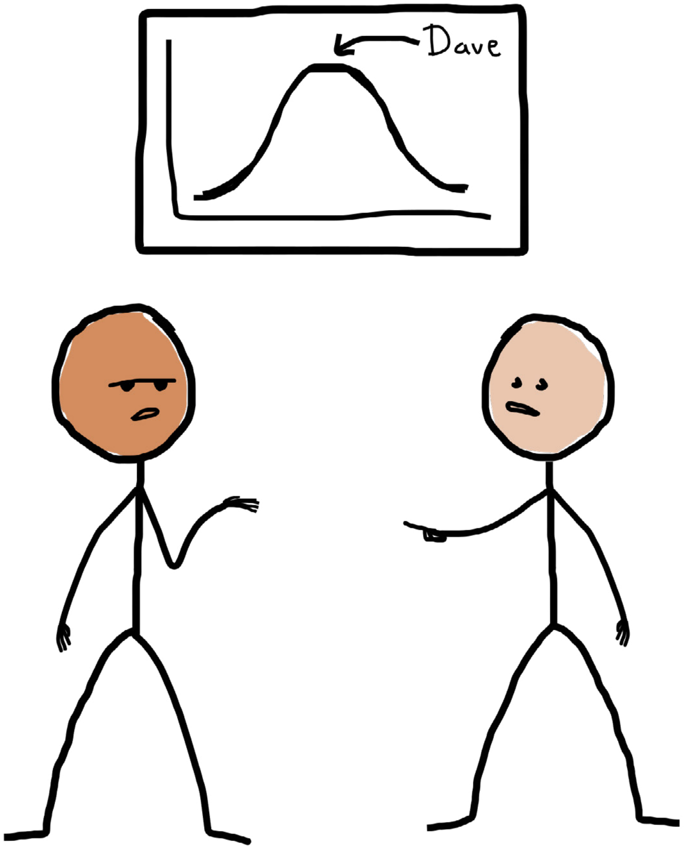
evocative than “Password” but in danger of leaving the user wondering what they’re really being asked for.

Summary

By exploring a specific form pattern, this chapter has given you most everything you need to develop inclusive forms in general. It’s by using standard form elements, effective labeling, and facilitating the correction of errors that a maximal number of users are able to access and contribute to your websites and apps. But by keeping the form simple and avoiding irritating experiences — like disappearing labels, and passwords that you cannot check — we’ve made sure using the form isn’t just possible but somewhat agreeable. Well, not entirely *dis*agreeable anyway!

THINGS TO AVOID

- Putting content that needs to be read *first* last in the source.
- Input focus styles that are too subtle.
- Missing, invisible or unassociated labels and descriptions.
- Uninteractive content in a <`form`> context that isn’t associated with form fields.



“We designed it for our average user: a guy called Dave.”
“So, Dave must be pretty happy then, right?”
“It’s a social media app that no one else can use.”

Test-Driven Markup

TEST-DRIVEN DEVELOPMENT[193] (TDD) allows developers working with frequent iterations to move forward with confidence. By writing tests first, to prescribe outcomes, then creating the functionality to achieve them, you can ensure successful builds behave in a predictable and reliable fashion. Thrill seekers who relish their "What the hell is going on?" moments may find TDD a little boring, but it's a solid approach.

In application development, tests are usually written against functions and what those functions are expected to produce. For example, if I wanted to test that the `add()` function from my *add.js* Node module worked correctly, I could use Mocha[194] and the Chai[195] assertion library:

```
var expect = require('chai').expect;
var add = require('../lib/add.js');
describe('Add module', function() {
   describe('The add() function', function() {
      it('should give 4 when adding 3 to 1', function() {
         add(3, 1).should.equal(4);
      });
   });
});
```

193 https://en.wikipedia.org/wiki/Test-driven_development
194 https://mochajs.org
195 http://chaijs.com/

Now I can just run `npm test` (or whatever alias I have set up) and I'll find out if my function is doing what I hoped it would.

That's all very well, but it's designed for the largely imperative language of JavaScript. What if we could write similar tests for declarative languages like HTML? Then we could ensure the correct *structure of markup* as we write it as well. Since well-formed markup is a big contributor to web accessibility, a kind of TDD for markup can keep us from building uninclusive patterns.

It turns out there's already a language perfectly suited for this task. Appropriately, it is itself a declarative language and, fortuitously, it does not take the form of an additional library dependency. I'm talking, of course, about CSS.

The Logic Of Selectors

Despite CSS being a declarative language, there is a logic in the formulation of selectors. Selectors are condition-based expressions which are used to match HTML structures. So good are CSS selectors at doing this that tools for *writing* markup like Emmet[196] are based on selector-like syntax.

196 http://emmet.io/

The inherent logic of CSS selectors becomes clear when you reimagine selectors in JavaScript syntax.

For example, I might want to match all buttons that are not disabled:

```
button:not(:disabled) { … }
```

Rewritten in JavaScript:

```
if (element.nodeName === 'button' && !element.disabled) { … }
```

Typically, CSS selectors are used to match expected patterns but, by the same token, we could use selectors to target broken or malformed patterns. Where these are present, an error style could be elicited to highlight the problem.

```
[undesired pattern] {
   /* error style, such as
   outline: 0.5em solid red;
   */
}
```

Now all we need is a method for describing the issue to the user on inspection of the highlighted element. Let's see how this might work using a tab interface as our subject pattern.

The Test-Driven Tab Interface

I detail the formulation of accessible, ARIA-enhanced tab interfaces in my book, "Apps For All: Coding Accessible Web Applications."[197] Also available are a demo and explanation.[198] If you're not familiar with this integrated pattern and its expected behaviors, please refer to either of these resources.

The following code example sets out a tab interface as marked up in its initial state. Our job is to write tests against the elements properties and relationships within the tab interface, to make sure it is structured as expected. With relatively complex patterns like tab interfaces, it's easy to slip up and miss out or badly form part of the structure. This is our opportunity to stop that from happening.

```
   <div class="tab-interface">
    <ul role="tablist">
        <li role="presentation"><a href="#panel1" id="tab1"
role="tab" aria-selected="true">First Tab</a></li>
        <li role="presentation"><a href="#panel2" id="tab2"
role="tab" tabindex="-1">Second Tab</a></li>
        <li role="presentation"><a href="#panel3" id="tab3"
role="tab" tabindex="-1">Third Tab</a></li>
    </ul>
```

197 http://smashed.by/apps4all
198 http://heydonworks.com/practical_aria_examples/#tab-interface

```
    <div role="tabpanel" id="panel1" aria-labelledby="tab1">
          <!-- tab panel 1 content -->
    </div>
    <div role="tabpanel" id="panel2" aria-labelledby="tab2">
          <!-- tab panel 2 content -->
    </div>
    <div role="tabpanel" id="panel3" aria-labelledby="tab3">
          <!-- tab panel 3 content -->
    </div>
</div>
```

THE TESTS

Let's start from the top. I know that my tabs will only function correctly in assistive technologies if they belong to an element with the special `tablist` role to group the individual tabs. My selector should identify the <`ul`> element within the `class="tab-interface"` container and highlight it if it doesn't have a `role="tablist"` attribute.

```
.tab-interface ul:not([role="tablist"]) {
   outline: 0.5em solid red;
}
```

Now, if no well-placed `tablist` semantics are present, an ugly red outline will appear on the list element.

As stated in the `tablist` role specification,[199] tablists *require* "owned elements" with the explicit `tab` role, in the form `role="tab"`. Some implementations place this attribute directly on each `<li>` element, but I prefer to use `<a>` elements so that the structure easily degrades as a list of links. See the "Navigation Regions" chapter for more on link lists and navigation.

Having decided that it is the `<a>` element which should take the `tab` role, I can add a corresponding test to my suite.

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]) {
  outline: 0.5em solid red;
}
```

Where JavaScript has added the tab interface semantics, the list semantics become redundant, which is why we suppress them with `role="presentation"` on each `<li>` element. This test selector follows similar logic to the previous test.

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"])
{
  outline: 0.5em solid red;
}
```

199 https://www.w3.org/TR/wai-aria/roles#tablist

In correctly behaving tab interfaces, there should always be one selected tab, defined accessibly using the `aria-selected`[200] state. This should be the only element focusable by the user with the Tab key. All other tabs should take the `tabindex="-1"` attribute and be selectable using only the arrow keys. This requires two more test selectors. The second is essentially a negated version of the first.

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"]),
[role="tablist"] a[aria-selected][tabindex="-1"],
[role="tablist"] a:not([aria-selected]):not([tabindex="-1"])
{
  outline: 0.5em solid red;
}
```

We can even detect incorrect patterns used to formulate the `tabpanel` `aria-labelledby` attibute values. The additional test selector in the following example matches tab panels with the correct role, but with an `aria-labelledby` value not commencing with "tab". This fuzzy matching is possible using the *starts with* `^` modifier in the attribute selector, `[aria-labelledby^="tab"]`.

200 http://w3c.github.io/aria/aria/aria.html#aria-selected

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"]),
[role="tablist"] a[aria-selected][tabindex="-1"],
[role="tablist"] a:not([aria-selected]):not([tabindex="-1"]),
[role="tabpanel"]:not([aria-labelledby^="tab"])
{
  outline: 0.5em solid red;
}
```

Of course, this catches the complete absence of `aria-labelledby` as well.

Finally, let's make sure all the tab panels have the `tabpanel` role. We know that the tab panel set should appear after the list, so we can use the general sibling combinator `~` in this test to ask if there are any `<div>`s following the `tablist` that are *not* `tabpanel`s.

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"]),
[role="tablist"] a[aria-selected][tabindex="-1"],
[role="tablist"] a:not([aria-selected]):not([tabindex="-1"]),
[role="tabpanel"]:not([aria-labelledby^="tab"]),
[role="tablist"] ~ div:not([role="tabpanel"])
{
  outline: 0.5em solid red;
}
```

Having content between the `tablist` and tab panels would fragment the interface and cause confusion. As an accompaniment to the last test, we can make sure a <`div`> with the `tabpanel` role is the first sibling element after the `tablist`. This test selector uses the adjacent sibling combinator[201] (or next-sibling selector) **+**:

```
.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"]),
[role="tablist"] a[aria-selected][tabindex="-1"],
[role="tablist"] a:not([aria-selected]):not([tabindex="-1"]),
[role="tabpanel"]:not([aria-labelledby^="tab"]),
[role="tablist"] ~ div:not([role="tabpanel"]),
[role="tablist"] + div:not([role="tabpanel"])
{
  outline: 0.5em solid red;
}
```

ERROR MESSAGES

So far, we've provided a red outline for any elements which match undesired patterns. But that's not much use unless we also provide an explanation of the error. In the experimental CSS testing bookmarklet *revenge.css*,[202] I used test selectors to identify generic accessibility-related markup errors and provided error messages using `pseudo-content`.

201 http://smashed.by/siblingselectors
202 http://heydonworks.com/revengecssbookmarklet/

For example, the following prints a lurid message for lists which are badly formed:

```
ol > *:not(li)::after,
ul > *:not(li)::after {
  content: 'Only <li> can be a direct child of <ul> or <ol>.';
}
```

However, actually printing that error to the screen is fraught with problems. Since it piggybacks a site's own CSS, I had to be very careful about overriding certain styles for `pseudo-content` that would mess up the appearance and readability of the error messages.

Where there are errors for successive sibling elements (like the erroneous `<li>` replacements the last example is designed to detect) space becomes an issue too.

A11y.css[203] — a more advanced CSS accessibility testing tool by Gaël Poupard[204] — overcomes this by positioning each `pseudo-content` message at the top of the document and reveals it on hover.

203 https://ffoodd.github.io/a11y.css/
204 https://twitter.com/ffoodd_fr

But why bother showing the error messages within the page itself at all? Instead, I can use the developer tools CSS inspector like a JavaScript console and provide the error messages there. The red outline should remain, but only as an indicator of which elements the developer should be inspecting.

In which case, I could just hide the pseudo-content message with `display:none;`:

```
ol > *:not(li)::after {            index.html: 15
  display: none;
  content: 'Only <li> elements are permitted as
  children of <ul> elements';
}
```

There are a couple of small issues here. The first is that when two errors apply to the same element, all but the last in the cascade will be grayed out (with a line-through style) in the CSS inspector. That is, it looks misleadingly as if all but the last error is not applicable. The other is that `display:none;` is supplemental to the error message — unnecessary noise.

THE ERROR PROPERTY

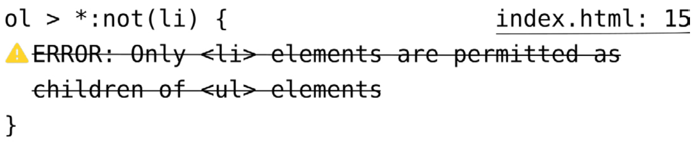
The is no such thing as the `ERROR` property in CSS and there are no plans for one — let me make that clear right away!

But there are a few benefits to using an unrecognized CSS property to log errors to the CSS inspector. The first is that `pseudo-content` becomes entirely redundant, along with the `display:none`; property, reducing an individual CSS error to this:

```
.tab-interface ul:not([role="tablist"]) {
  ERROR: The tab interface <ul> must have the tablist WAI-
ARIA role
}
```

Second, as an unrecognized property it is not entered into specificity calculations. Depending on the browser, this either means no more graying out, or a grayed-out style for *all* `ERROR` declarations: a slight improvement.

My favorite part is that the Chrome browser's way of highlighting unrecognized properties is to prefix them with a little warning sign ("⚠"). We automatically co-opt this sign to help highlight our errors, producing something like this:

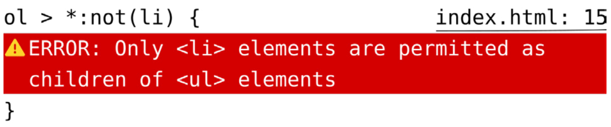


ol > *:not(li) { index.html: 15
⚠~~ERROR: Only <li> elements are permitted as children of <ul> elements~~
}

A declaration block from Chrome dev tools showing the error declaration with a line-through style.

Unfortunately, like overriden declarations, unrecognized declarations still take a line-through style as illustrated. For my money, this isn't a deal-breaker because the capitalized `ERROR` property name picks these declarations out quite clearly.

Nonetheless, with the help of Dan Smith,[205] I can provide a small Chrome extension[206] for removing this line-through style and replacing it with a more error-like white-on-red appearance:



```
ol > *:not(li) {                        index.html: 15
⚠ERROR: Only <li> elements are permitted as
  children of <ul> elements
}
```

Note that it's no longer necessary to encapsulate the error in single quotation marks as a string.

Be aware that this style will be honored by most unrecognized declarations. However, thanks to the extension's omission of the `.has-ignorable-error` class, the most common unrecognized properties — browser prefixes — are not styled this way.

205 https://twitter.com/dansketchpad
206 https://github.com/Heydon/css-error-property-style/blob/master/

```
.overloaded.not-parsed-ok.inactive:not(.has-ignorable-error)
{
   text-decoration: none !important;
   background: red !important;
   color: #fff !important;
}

.overloaded.not-parsed-ok.inactive:not(.has-ignorable-error)
.webkit-css-property {
   color: #fff !important;
}
```

PUTTING IT TOGETHER

Our tab interface pattern should now have a *tab-interface.css* file and, in an effort to establish a naming convention, an accompanying *tab-interface.test.css* file — something like this:

```
.tab-interface ul:not([role="tablist"]) {
   ERROR: The tab interface <ul> must have the tablist
WAI-ARIA role to be recognized in assistive technologies.;
}

[role="tablist"] a:not([role="tab"]) {
   ERROR: <a> elements within the tablist need to each have
the WAI-ARIA tab role to be counted as tabs in assistive
technologies.;
}

[role="tablist"] li:not([role="presentation"]) {
   ERROR: Remove the <li> semantics with the WAI-ARIA
presentation role. Where the tab interface is instated, these
semantics are irrelevant.;
}
```

```
[role="tablist"] a[aria-selected][tabindex="-1"] {
   ERROR: Remove the -1 tabindex value on the aria-selected
tab to make it focusable by the user. They should be able to
move to this tab only.;
}

[role="tablist"] a:not([aria-selected]):not([tabindex="-1"])
{
   ERROR: All unselected tabs should have the -1 tabindex value
and only be focusable using the left and right arrow keys.;
}
[role="tabpanel"]:not([aria-labelledby^="tab"]) {
   ERROR: Each tabpanel should have an aria-labelledby
attribute starting with "tab" followed by the corresponding
tab's number. This is the convention of tab systems in our
interface.;
}

[role="tablist"] ~ div:not([role="tabpanel"]) {
   ERROR: Each tabpanel needs to have the explicit tabpanel
WAI-ARIA role to be correctly associated with the
tablist that controls it.;
}

[role="tablist"] + div:not([role="tabpanel"]) {
   ERROR: The first element after the tablist should be a tab
panel with the tabpanel WAI-ARIA role. Screen reader
users must be able to move directly into the open tab panel
from the selected tab.;
}

.tab-interface ul:not([role="tablist"]),
[role="tablist"] a:not([role="tab"]),
[role="tablist"] li:not([role="presentation"]),
[role="tablist"] a[aria-selected][tabindex="-1"],
[role="tablist"] a:not([aria-selected]):not([tabindex="-1"]),
[role="tabpanel"]:not([aria-labelledby^="tab"]),
[role="tablist"] ~ div:not([role="tabpanel"]),
[role="tablist"] + div:not([role="tabpanel"])
{
  outline: 0.5em solid red;
}
```

Of course, none of this can technically *fail* a build in its current form; it just highlights errors visually, for the front-end developer to see. However, the *.test.css* files certainly don't want to be included in production. Using the *.test.css* convention helps when it comes to omit them from being copied to your production build folder.

Not One-Size-Fits-All

The difference between this test-driven markup approach and standard automated accessibility testing is clear: whether using a CSS-based bookmarklet like *a11y.css*[207] or an advanced API like *tenon.io*,[208] generic errors are disclosed. This testing is important, especially for WCAG compliance. But writing your own tests for your own tailored patterns and structures means you can be more granular and specific about their expected form.

207 https://ffoodd.github.io/a11y.css/
208 http://tenon.io/

Implementations of tab interfaces can differ. For instance, the relationship between the tabs and their panels can be achieved using `aria-controls` rather than `aria-labelledby`. The tests written here ensure that *my* implementation — my specific pattern — is structured in the prescribed way.

Where additional JavaScript and CSS hooks (`data` and `class` attributes) should be present, this is also an opportunity to test for them.

As you develop your own library of inclusive design patterns, I recommend you try writing some tests like these. As the pattern evolves over time and between colleagues, you can make sure its integrity remains intact. Where an error does emerge, you will have taken the opportunity — via the error message — to explain your decision-making and how the structure you've chosen makes the markup inclusive.

Further Reading

Thank you for reading my book!

One of the best things about writing a book about patterns is that there are always more patterns to talk about. And you know what that means: I can write a sequel! If you can think of any inclusive patterns you'd like me to explore, you can find me on Twitter as @heydonworks. Also, if you spot any inconsistencies or errors here (it happens!), please contact myself or Smashing Magazine so we can update the errata page.[209]

In the meantime, here's a list of resources I recommend for reading about and around inclusive design, covering everything from responsive design to UX, technical accessibility, performance, and internationalization.

209 http://smashed.by/errata

Books

- *A Web For Everyone* by Sarah Horton and Whitney Quesenbery: http://smashed.by/aweb4everyone
- *Apps For All* by Heydon Pickering: http://smashed.by/apps4all
- *Designing With Web Standards* by Jeffrey Zeldman with Ethan Marcotte http://smashed.by/webstandards
- *Adaptive Web Design* by Aaron Gustafson: http://adaptivewebdesign.info/
- *The Design Of Everyday Things* by Donald A Norman: http://smashed.by/everydaythings
- *Colour Accessibility* by Geri Coady https://gumroad.com/l/loura11y —
- *Don't Make Me Think* by Steve Krug: http://smashed.by/dontmakemethink
- *Responsible Responsive Design* by Scott Jehl http://smashed.by/resres
- *Web Performance* by Andy Davis: http://andydavies.me/books/#webperformance
- *Front-end Style Guides* by Anna Debenham: http://www.maban.co.uk/projects/front-end-style-guides/

- *International User Research* by Chui Chui Tan: https://gumroad.com/l/international-user-research
- *Service Design* – From Insight To Implementation by Andy Polaine, Lavrans Løvlie, Ben Reason: http://smashed.by/servicedesign
- *Design Meets Disability* by Graham Pullin: http://smashed.by/desdis
- *The Joy of UX – User Experience and Interactive Design for Developers* by David Platt: http://smashed.by/joyux
- *Including Your Missing 20% By Embedding Web And Mobile Accessibility* by Jonathan Hassell: http://www.hassellinclusion.com/landing/book/
- *Design For Real Life* by Eric Meyer & Sara Wachter-Boettcher: http://smashed.by/reallife
- *The Timeless Way Of Building* by Christopher Alexander: http://smashed.by/timelessway

Blogs

- Marco’s Accessibility Blog: https://www.marcozehe.de/
- Paciello Group’s blog: https://www.paciellogroup.com/blog/
- Accessibility Wins: https://a11ywins.tumblr.com/
- Tink.uk: http://tink.uk/ (Léonie Watson)
- Adrian Roselli: http://adrianroselli.com/
- Simply Accessible: http://simplyaccessible.com/articles/
- Web Axe: http://www.webaxe.org/ (Dennis Lembrée)
- Web Aim’s blog: http://webaim.org/blog/
- Karl Groves: http://www.karlgroves.com/
- SSB Bart Group’s blog: http://www.ssbbartgroup.com/blog/
- Deque’s blog: http://www.deque.com/blog/
- Terrill Thompson: http://terrillthompson.com/blog/
- Nielsen Norman Group: https://www.nngroup.com/articles/
- GOV.UK accessibility blog: https://accessibility.blog.gov.uk/
- lukew: http://www.lukew.com/ff/ (Luke Wroblewski)
- UX Booth: http://smashed.by/a11yforms