

RESPONSIBLE DESIGN

“My love for responsive centers around the idea that my website will meet you wherever you are—from mobile to full-blown desktop and anywhere in between.”

—TRENT WALTON, “Fit To Scale” (<http://bkaprt.com/trd/1-01/>)

RESPONSIVE DESIGN’S core tenets (fluid grids, fluid images, and media queries) go a long way toward providing a holistic package for cross-device interface design. But responsive design itself relies on features that may not work as expected—or at all. Our sites need to react to unexpected user behaviors, network conditions, and unique support scenarios.

In this chapter, we’ll dig into two responsible tenets: usability and accessibility. We’ll cover higher-level considerations before getting into nitty-gritty code you can implement now and expect to last. To start, let’s talk design.

DESIGNING FOR USABILITY

When we consider usability in responsive design, we think about how to present a design's content and features across a range of screen sizes and devices. Do the interface components yield to the content when screen real estate is tight? Do the components function intuitively in response to various input modes? Are the content and hierarchy easy to parse? Do the line lengths foster readability across screen sizes?

Get into the browser quickly

"Let's change the phrase 'designing in the browser' to 'deciding in the browser.'"

—DAN MALL, The Pastry Box Project (<http://bkaprt.com/rrd/1-02/>)

At Filament Group, we start most of our projects in Adobe Illustrator, where we iterate on high-level visual design concepts. We then try to move to code as soon as possible. At this stage, we aim to design the fewest number of interface variations that communicate a plan for layout and interactivity across viewports—mere suggestions for how the site will look and feel on any given device. Decisions about how features react to different input mechanisms and browser capabilities, as well as the particular viewport sizes that should receive each layout variation, remain to be determined. The goal is to move into the browser as quickly as we can to make design and interaction decisions in context, which translates to more informed recommendations for our clients.

Find your breakpoints

The viewport sizes at which we change from one fluid layout to another using media queries are called *breakpoints*. Here are two examples:

```
/* first breakpoint */
@media (min-width: 520px){
  ...styles for 520px widths and up go here!
}
/* second breakpoint */
@media (min-width: 735px){
  ...styles for 735px widths and up go here!
}
```

While it's tempting to choose breakpoints early in the design process, perhaps based on the dimensions of popular devices we know we need to support, the truth is that we shouldn't choose breakpoints at all. Instead, we should find them, using our content as a guide.

"Start with the small screen first, then expand until it looks like shit. TIME FOR A BREAKPOINT!"

—STEPHEN HAY, <http://bkaprt.com/rrd/1-03/>

A layout's design and content should shape and inform a layout's breakpoints. As Hay notes, the easiest way to find breakpoints is simply to resize the browser viewport until the content becomes awkward (that's the technical term) to use or read—and presto, a breakpoint.

In addition to a gut check, you might opt for a slightly more objective guideline. Per Richard Rutter's homage to Robert Bringhurst, *The Elements of Typographic Style Applied to the Web* (<http://bkaprt.com/rrd/1-05/>), an optimal *measure*—the number of characters per line in a column of text—for immersive reading is widely thought to fall between 45 and 75 characters, including spaces (FIG 1.1). If you're resizing a layout outward, watch for when a column of text approaches that range: it's probably a good place to adjust your layout.

As you work with complex responsive designs, you'll find that breakpoints often occur at different times for different portions of a layout, and that some are more significant than others.

2.1.2 Choose a comfortable measure

“Anything from 45 to 75 characters is widely regarded as a satisfactory length of line for a single-column page set in a serifed text face in a text size. The 66-character line (counting both letters and spaces) is widely regarded as ideal. For multiple column work, a better average is 40 to 50 characters.”

FIG 1.1: Here, a seventy-character line length makes for comfortable reading (<http://bkaprt.com/rrd/1-04/>).

Major breakpoints mark big shifts, usually to add columns or dramatically change the presentation of more than one component; *minor* breakpoints involve smaller design tweaks (such as changing a component’s **font-size** to prevent text wrapping) that take full advantage of the spaces between the major breakpoints. In general, I find that major layout breakpoints are decided early in development, while minor ones act as finishing touches. The fewer breakpoints we use, the easier a responsive design will be to maintain.

Let’s look at an example. On the *Boston Globe* website, we have two or three major layout breakpoints, but the more complicated components break more often. The site’s masthead component has four major breakpoints, as well as some minor ones for slight adjustments to prevent text wrapping (FIG 1.2).

Design modularly

As in the masthead example, I find it helpful to compile the multiple configurations of each modular component in isolation; that way, I can test its usability and document its variations in one place. Developer Dave Rupert of Paravel explored this concept in his post “Responsive Deliverables” (<http://bkaprt.com/rrd/1-06/>). “Responsive deliverables should look a lot like fully



First breakpoint: navigation and search options toggle on tap.



Second breakpoint: logo moves left to split the width with the navigation.



Third breakpoint: logo moves back to center, search box visible at all times.



Fourth breakpoint: search box moves left of logo, navigation expands.



Final breakpoint: search box widens, classified links visible at all times on top left.

FIG 1.2: Major and minor breakpoints of the Boston Globe’s masthead.

functioning Twitter Bootstrap-style (<http://bkaprt.com/rrd/1-07/>) systems custom tailored for your clients’ needs,” Rupert writes. In other words, we should build and document our components from the inside out, as standalone pieces that play nicely with others.

Same content, reduced noise

You've figured out how to find horizontal breakpoints across a range of viewport sizes. How do you fit all that content on small screens without making things noisy? Responsive design has (undeservedly) received a bad rap because of sites that attempt to avoid messy situations by hiding parts of the content from users—denying access to content that was ostensibly important enough to include in the first place. Remember, if it's useful to some people, it's likely useful to everyone. As Luke Wroblewski's book *Mobile First* instructs, rather than hide content that's inconvenient to display, it's best to reorganize the design to retain usability on smaller viewports.

Fortunately, we have many design patterns that work around small-screen constraints in interesting, intuitive, and responsible ways.

Progressive disclosure

One such pattern is *progressive disclosure*, a fancy term for showing content on demand. To be clear, not all hiding is bad; it's only bad if the user has no way to access the hidden content. The idea behind progressive disclosure is simple: hide portions of content, but provide interface cues so that users can view it when they wish (FIG 1.3).

Progressive disclosure is most often a simple show-and-hide like the example above, but we have plenty of ways to visually toggle content. For instance, this property listing component does a 3D flip upon tap or click to reveal additional information about a property, such as its address and location on a map (FIG 1.4). For browsers without 3D CSS animation support, users can toggle to the map without an animated transition, while basic browsers display the map at all times, just beneath the property information.

Off-canvas layout, a term coined by Luke Wroblewski in his article "Off-Canvas Multi-Device Layouts," describes another notable approach to minimizing complexity on small screens

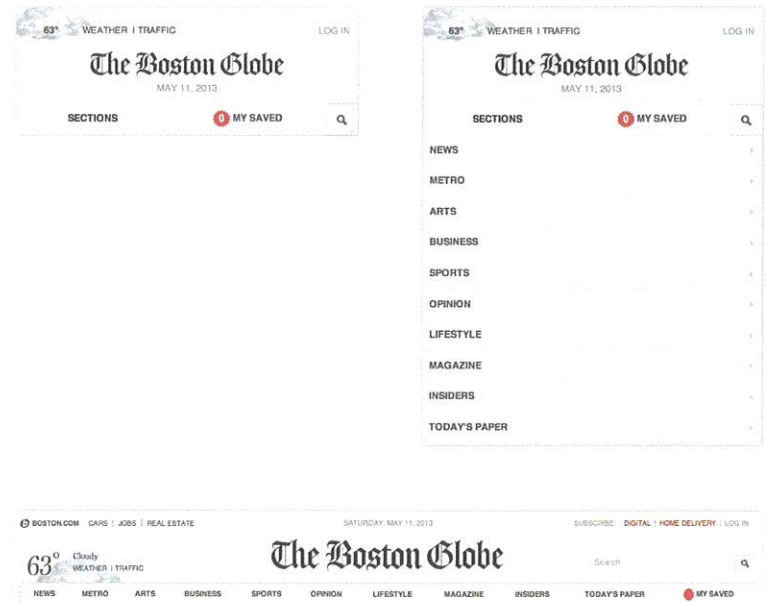


FIG 1.3: Boston Globe's navigation uses progressive disclosure on small viewports.

(<http://bkaprt.com/rrd/1-08/>). Wroblewski documents several patterns for positioning lower-priority interface components offscreen until users cue them by tapping a menu icon or similar item; the formerly offscreen content then enters the viewport, overlapping or pushing aside the primary content (FIG 1.5). This on-demand approach is becoming common in small-screen layouts.

Responsive tables

Tabular data is one of the toughest content types to present on a small screen. It's often essential that the user see column and

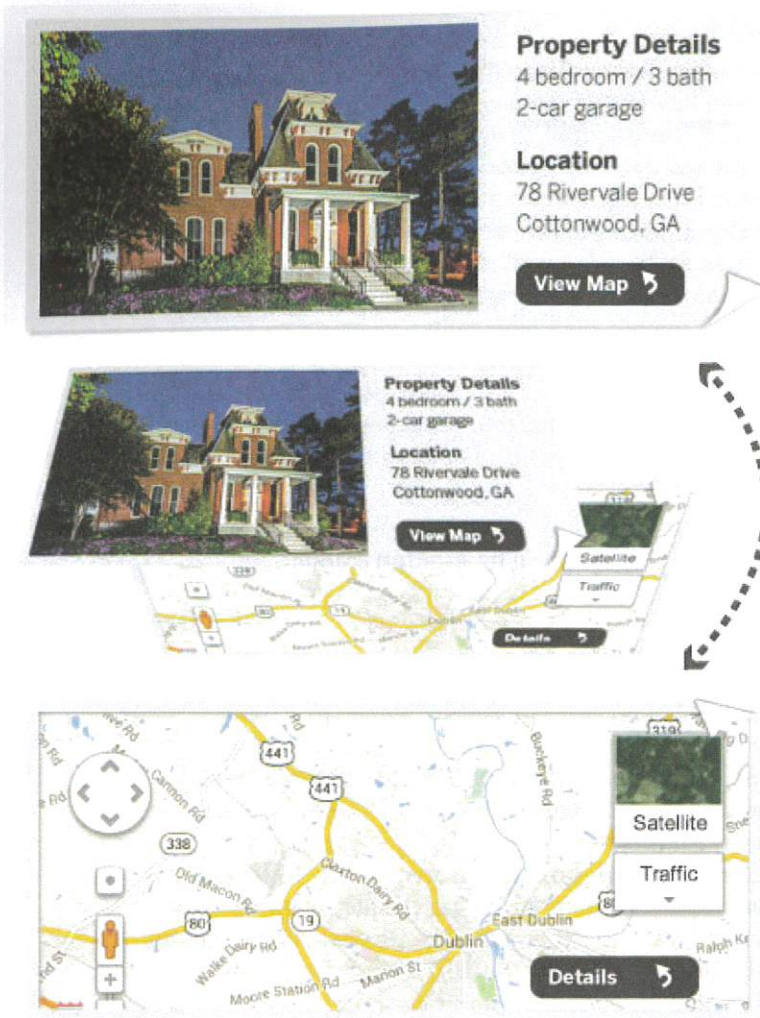


FIG 1.4: Progressively disclosed content flips in 3D to display more information.

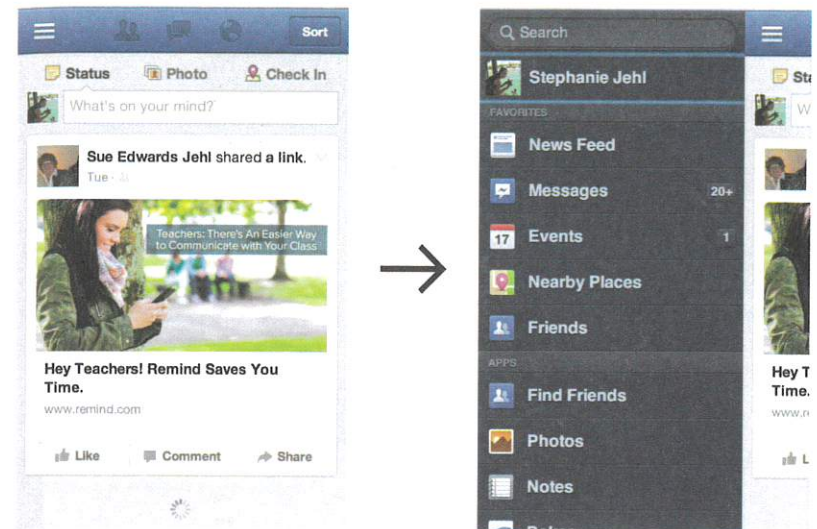


FIG 1.5: Tapping the menu icon reveals Facebook's off-canvas navigation from the screen's left edge.

row headers associated with a table cell, and yet we can only fit so many rows and columns on screen (FIG 1.6).

At Filament, we experimented a lot and found a couple of patterns that worked well enough to include in the jQuery Mobile framework. The first pattern, Reflow (<http://bkaprt.com/rrd/1-09/>), reformats the table from a multi-column view to a list view; each cell becomes its own row, with a row header to its left. (FIG 1.7).

To pull this off, Reflow uses CSS to set each cell in the table to `display: block`, creating a new row, and JavaScript to grab each of the table's column headers and insert them in each cell to serve as the labels (while hiding the additional labels from screen readers). Reflow suits simple tables that act like formatted lists, but its small-screen presentation falls short when you need to compare data points across rows.

FIG 1.6: Large tables can cause usability trouble on small screens.



Rank	1	Rank	Movie Title	Year	Rating	Reviews
Movie Title	Citizen Kane	1	Citizen Kane	1941	100%	74
Year	1941	2	Casablanca	1942	97%	64
Rating	100%	3	The Godfather	1972	97%	87
Reviews	74	4	Gone with the Wind	1939	96%	87
		5	Lawrence of Arabia	1962	94%	87
		6	Dr. Strangelove Or How I Learned to Stop Worrying and Love the Bomb	1964	92%	74
Rank	2	7	The Graduate	1967	91%	122
Movie Title	Casablanca	8	The Wizard of Oz	1939	90%	72
Year	1942	9	Singin' in the Rain	1952	89%	85
Rating	97%	10	Inception	2010	84%	78

FIG 1.7: An example of the jQuery Mobile Reflow table pattern, with the same table shown at narrow and wide widths.



FIG 1.8: An example of the jQuery mobile Column Toggle table pattern, with the same table shown at narrow and wide widths.

The Column Toggle (<http://bkaprt.com/rrd/1-10/>) pattern picks up that slack. It works by selectively showing columns in a table as horizontal space allows. If there isn't room, CSS hides the column data, but a menu offers users the chance to override the CSS and display the column anyway, eventually causing the table to expand wide enough to warrant horizontal scrolling (FIG 1.8).

These are only two of the numerous potential patterns for responsibly presenting tabular content. For more examples, check out Brad Frost's project Responsive Patterns (<http://bkaprt.com/rrd/1-11/>). You'll find everything from horizontal navigation components that collapse into menus when space is tight to CSS-Flexbox-driven grids for complex page layouts.

DESIGNING FOR TOUCH (AND EVERYTHING ELSE)

A responsive layout is but one step. Even if your site flows beautifully from one screen size to the next, you're not doing your job if someone can't use it. Touch isn't only the domain of

small screens; many devices offer touch alongside other input mechanisms. But as the number of people on touch devices surges, we must add touch to our arsenal of common interactions like mouse, focus, and keyboard. While the intricacies of touch can be daunting, we don't need to completely overhaul our designs to be touch-friendly. Far from it: one of the joys of responsible design is how it builds on our everyday tool set. Two basic measures pack a wallop on the usability of an existing, mouse-based interface:

- Make sure any content that offers mouse-centric interactivity (like **hover**) is also accessible in browsers where a mouse pointer may not exist.
- Don't assume touch will be used, but design as if it will be. Let's see how these play out with the following considerations.

Save hover for shortcuts

The absence of **mouseover** (or **hover**) interactions is one of the biggest changes when learning to support touch. In fact, the lack of **mouseover** support on many touch devices is a primary reason that many sites designed for the desktop web falter in touch contexts, resulting in usability problems that prevent users from accessing certain features. You can't rely on **mouseover** for vital design interactions, but you can use it as a nice-to-have alternate way to reach otherwise accessible content.

One example is the navigation for the Global News Canada website, designed by Upstatement and developed by the Filament Group team (FIG 1.9). The global navigation links users to National, Locals, and Watch section homepages when clicked or tapped. These links also feature split-button drop menus that toggle between sections on hover. On a touch screen, one tap directly sends users to that section's homepage, so we came up with an alternative mechanism to toggle between menus and account for all breakpoints. The split buttons with arrows next to each navigation link do just that, offering tap or click access to the drop menu.

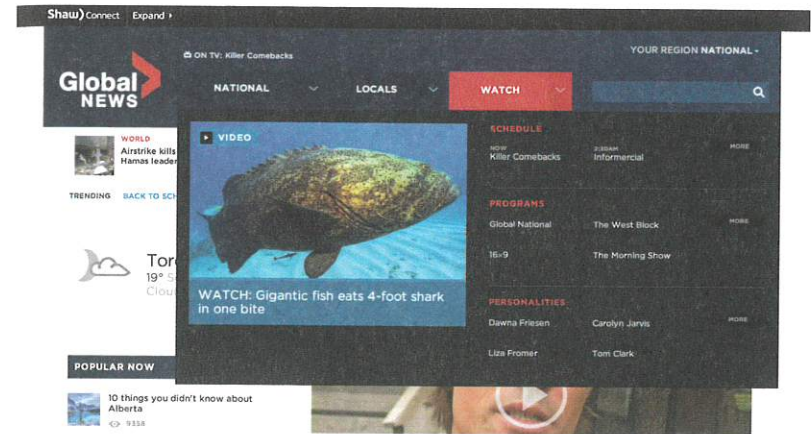


FIG 1.9: The split-button menus on GlobalNews.ca work for touch and mouseover.

Keep in touch

One rule of thumb(s): the devices accessing your site may or may not have touch screens, but always design as if they will. Fingers aren't precise, so we need to enlarge button and link target areas to make them easier to tap. How much bigger is an open discussion, though Apple's guidelines suggest 44×44 pixels as the minimum size for usable buttons. Based on findings from MIT's Touch Lab (<http://bkaprt.com/rrd/1-12/>), the *Smashing Magazine* article "Finger-Friendly Design: Ideal Mobile Touchscreen Target Sizes" by author Anthony T suggests slightly larger targets at 45-57 pixels, and 72 pixels for buttons for thumb use, like the ones located near the bottom of a handheld device's screen (FIG 1.10).

Don't forget your white space! Equally important as the size of touchable elements is the space around those elements. A smaller button surrounded by dead space can be as easy to use as a larger element, so the size of the button within its tappable footprint becomes a question of visual emphasis.

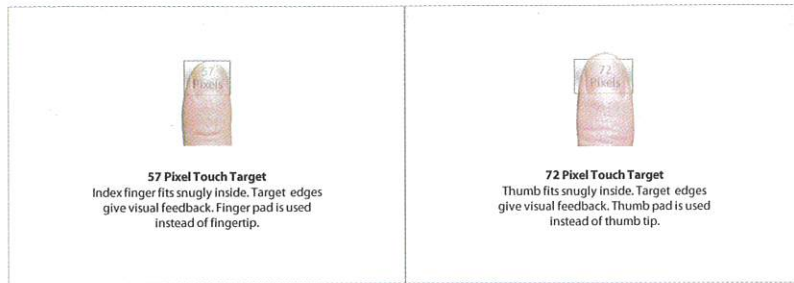


FIG 1.10: Illustrations from *Smashing Magazine's* article (<http://bkaprt.com/rrd/1-13/>).

The usual gestures

Touch screens offer the potential for richer interactions than tap alone—many touch gestures have become commonplace, particularly in native apps. This diagram by Craig Villamor, Dan Willis, and Luke Wroblewski demonstrates some popular gestures in touch interaction (FIG 1.11).

You're probably familiar with most of these gestures, which are used by operating systems on several devices (including iOS). Within browsers, these gestures are often paired with convenient default behavior that varies from device to device; some gestures share the same behavior. For example, a double tap or pinch or spread in iOS Safari causes the browser to zoom in or out on a particular region. Dragging or flicking in any direction causes the page to scroll; and a press, or touch-hold, often exposes a context menu akin to what you'd see when right-clicking with a mouse.

Native gestures like these have all sorts of implications for how we can responsibly develop for touch. Users form expectations about their devices' native features, so we don't want to disable or repurpose a feature like touch-hold if we can avoid it. While browsers do let us use touch events like `touchstart`, `touchmove`, and `touchend` (or the new standard pointer events `pointerdown`, `pointermove`, `pointerup`, etc.) to specify gestures with JavaScript, how can we do so without conflicting with native touch behavior?

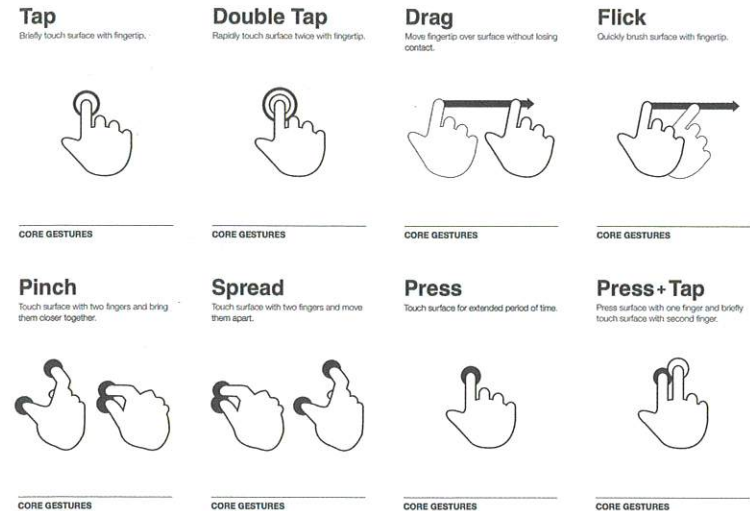
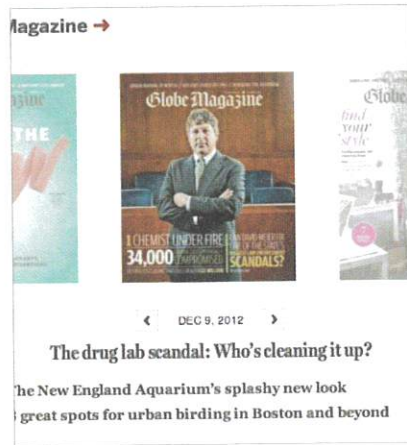


FIG 1.11: Touch Gesture diagram (<http://bkaprt.com/rrd/1-14/>).

Web-safe gestures: do they exist?

Let's compile a list of web-safe gestures we can use in our sites (spoiler: it's short). Based on the native gestures in today's popular devices, we have tap, two-finger tap, horizontal drag, and horizontal flick. Yet within this small list, we still have potential for conflict. For instance, Chrome on iOS and Android allows users to horizontally swipe to switch between open tabs, while iOS Safari uses the same gesture to go back or forward in browser history, which means our use of those gestures can lead to unexpected behavior. Horizontal drag gestures can also introduce issues even in touch browsers that don't use them for native navigation. For example, if a page's content stretches wider than the browser's viewport, which often happens after zooming in, a horizontal touch-drag is typically used to scroll the page right or left, so we have to be careful that our custom touch gestures don't interfere.

FIG 1.12: The multiple-input-mode carousels on the *Boston Globe* site.



Keep in mind that I've deemed these gestures safe only because I'm unaware of any touch-based browsers that use them—yet. The moment iOS implements two-finger tap, anything we've built may conflict with native behavior, and that's not future-friendly at all. This doesn't mean we should avoid building custom gestures, but it highlights the importance of developing for many input modes. If one fails for any reason, we'll have alternate ways to access our content.

In practice, this means ensuring there's always a click-and-keyboard-based interface for interaction. For example, the carousel of magazine covers on the *Boston Globe* site has several interactive options (FIG 1.12). You can click the arrows beneath the carousel, click the covers to the right or left of the featured image, use the right and left arrow keys on your keyboard, or touch-drag the carousel on a touch device. Think of touch gestures as a nice-to-have enhancement on top of broadly supported input modes.

Perhaps a bigger problem with touch gestures is discovery, as touch gestures often lack any visual interface to hint at their presence. We ran into this dilemma when building the *Boston Globe's* saved articles feature, which allows you to save articles to your account so you can read them later. On small screens, the Save buttons hide by default but can be toggled into view with a

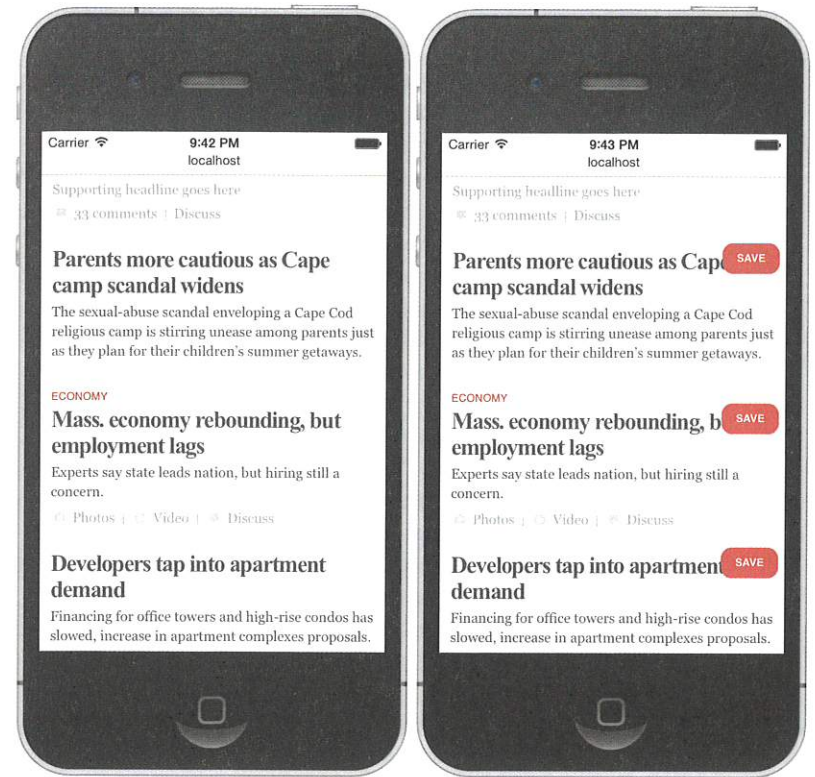


FIG 1.13: *Boston Globe's* Save buttons become visible via two-finger tap.

two-finger tap (FIG 1.13). Of course, there is no easy way to know that unless you visit the help section and read the instructions!

Scripting touch interactivity

Touch-screen browsers are typically capable of using components designed for mouse input, so outside of accommodating touch from a design perspective, you may not need to do anything special with JavaScript to ensure touch support. However, touch-specific events do exist, and the advantage of scripting with them is often a matter of richness and enhancement. When

developing components, for example, it's particularly nice to write code that listens for touch events because they respond immediately to touch interaction. By comparison, in many touch browsers, mouse events like `click` and `mouseup` typically fire 300 milliseconds or more after a user taps the screen (the device waits to make sure that a double tap isn't happening before it handles the `click`), so any site that's coded to respond to mouse events alone will suffer minor but noticeable delays. That said, scripting touch gestures can be tricky because most browsers that support touch emit both mouse and touch events whenever a touch occurs. Further complicating things, browsers sometimes use different touch-event names (such as the widely used `touchstart` rather than the emerging standard, `pointerdown`).

Whatever touch-screen optimizations we make, it's crucial not to hinder people's ability to interact with content using non-touch input mechanisms like the mouse and keyboard. A common, responsible approach to ensure that touch interactions work as fast as possible is to set up event listeners for both mouse and touch events. During a particular interaction, the logic would handle whichever event type happens first and ignore the other to prevent the possibility of running any scripting twice. Sounds straightforward, but it's not. That's why I recommend using a well-tested, open-source JavaScript library to do the hard work for you. I use Tappy.js (<http://bkaprt.com/rrd/1-15/>), a script I created to allow you to listen for a custom `tap` event when writing jQuery code. Here's Tappy in play:

```
$( ".myBtn" ).bind( "tap", function(){
    alert( "tap!" );
});
```

Behind the scenes, that `tap` event is listening for touch, keyboard, or mouse clicks to perform a specific behavior. (In this case, it throws an alert that says, "tap!" I'm sure you can find better uses for it, of course.)

For a library that offers a more advanced set of touch features, check out FastClick (<http://bkaprt.com/rrd/1-16/>), created and maintained by the talented team at *Financial Times*.

DESIGNING FOR ACCESS

We've covered some major aspects of usability, such as designing for screen variation, finding breakpoints, and handling input modes inclusively. But for components to be usable across devices, we must make sure that they're accessible in browsers that don't support our ideal presentation or behavior, and for users who browse the web with assistive technology. For these reasons and more, you can't do a better service to your users than to start with plain old HTML. A major strength of HTML is its innate backward compatibility, which means pages built with even the latest iterations can still be accessed from almost any HTML-capable device.

While HTML documents are born quite accessible, they don't always stay that way: careless application of CSS and JavaScript can render formerly accessible content completely unusable, leaving users worse off than they were with the initial, bare-bones experience. For example, consider a drop menu whose content is hidden with `display: none`. With exceptions, screen readers will relay only the content that is presented on screen, so if precautions aren't in place, that menu's content will not only be hidden *visually*, it will also be hidden *audibly* from screen reader users. We must provide meaningful cues to alert all users—not just those browsing the web visually—that the menu content exists and can be shown (or heard) when desired.

As we continue to push HTML toward new interactivity, it's critical that we think of access as something we constantly risk losing, as something we must retain throughout our development process.

Ensure access with progressive enhancement

The idea that the web is born accessible pairs neatly with the concept of progressive enhancement, which advocates starting with functional, meaningful HTML and then unobtrusively layering presentation (CSS) and behavior (JS) on top for a richer, more dynamic user experience.

With power comes responsibility. Any time you venture beyond standard browser rendering of HTML into building

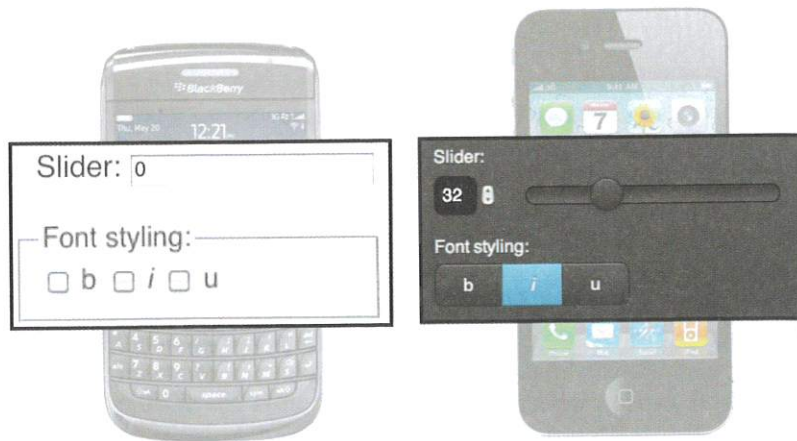


FIG 1.14: A view (left) of the underlying native controls behind an enhanced user interface (right).

your own presentation and interactivity, you're responsible for accessibility. This requires some planning. As developers, we must "see through" our visual interface designs to discover their underlying meaning in HTML.

In Filament Group's book *Designing with Progressive Enhancement*, we describe this process as *the x-ray perspective* (FIG 1.14):

The x-ray perspective is a methodology we've developed to evaluate a complex site design, break it down to its most basic modular parts, and build it back up in such a way that a single coded page will work for modern browsers with full functional capabilities as well as other browsers and devices that may understand only basic HTML.

The process of x-raying a design's parts may require a certain amount of creative thinking; it depends on how closely a custom control resembles a native equivalent. Some are fairly transparent: say, a button that acts as a checkbox `input`. In this case, a bit of CSS alone could render some `label` and `input` markup



FIG 1.15: A standard input and label styled as a button.

from a standard text and box presentation into the button-like component shown below (FIG 1.15):

```
<label class="checkbox">
  <input type="checkbox">Bold
</label>
```

A CSS-alone approach has triple benefits. It's simple, lightweight, and, most important, using native HTML form elements almost guarantees that the control will be accessible to users with disabilities. In other words, assistive technology like Apple's built-in VoiceOver screen reader will read the native control aloud as if the visual enhancements aren't even there: "bold, unchecked checkbox" by default and "bold, checked checkbox" when checked.

Easy, right? However, it can be difficult to maintain this level of accessibility with more complex custom components.

Responsibly enhance a complex control

Let's focus those x-ray specs on something more abstract, such as a slider (FIG 1.16):

A great feature in the HTML5 specification is the new set of form input types like `number`, `color`, and `search`. You can safely use these types today to deliver more specialized interactivity in supporting browsers; browsers that don't understand them will simply render the input as a standard `text` type.

Here's some markup for a `color` input:

```
<label for="color">Choose a color:</label>
<input type="color" id="color">
```

FIGURE 1.17 shows how it renders in Google Chrome, a supporting browser.

Results Shown:



FIG 1.16: A custom slider control with a numerical input.

Choose a color:



FIG 1.17: A color input with a color picker in Google Chrome.

FIGURE 1.18 shows it in iOS 7, a non-supporting browser.

Another new form input is `range`, which displays a slider control in most browsers. But the generated native slider leaves a lot to be desired from a design and usability perspective. For one, its appearance is vexing—sometimes impossible—to customize. Depending on the browser, the native slider lacks any text label to display the slider's value, making it useless for choosing precise values. For example, FIGURE 1.19 shows how a native `range` input with possible values of 0-10 renders in iOS 7 Safari.

```
<label for="value">Choose a value:</label>
<input type="range" id="value" min="0" max="10">
```



FIG 1.18: A color input falls back to a plain text input in iOS 7.

Choose a value:



FIG 1.19: A range input rendered in iOS 7 Safari, which gives no feedback about minimum, maximum, or current value.

Unless we're designing a music-volume control, this slider isn't helpful. If we want to create a usable, touch-friendly slider, we'll need to build it ourselves. Let's do so in a way that works for everyone.

FIG 1.20: Our foundational HTML markup as rendered in a browser.

Results Shown: 61

The first and most important step is to start with our pal, HTML. Deep down, a slider is a visualization of a numeric scale, so let's begin with an `input` element and give it a type of `number`, which is another HTML5 input that degrades to a `text` input in non-supporting browsers. Using `number` has the benefit of allowing us to use several standard, complementary attributes that shape the control's constraints: `min` and `max`. We'll use these attributes as our HTML starting point (FIG 1.20):

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="60" min="0" max="100" />
```

Now that we have our foundation, we can use JavaScript to create a slider component that will manipulate the `input`'s value when the user drags its handle.

The actual scripting to pull that off lies beyond this book's scope, but I will cover the resulting generated markup and how to make sure the slider doesn't hinder accessibility. First, the newly generated markup in bold:

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="60" min="0" max="100" />
<div class="slider">
  <a href="#" class="handle" style="left: 60%;"></a>
</div>
```

Let's walk through the changes. To create our slider handle and track, we need to use an element that is natively focusable via keyboard, in this case an `a` element, to which I assigned the

Results Shown:

61 

FIG 1.21: Our slider div, with the input element shown to the left.

class `handle` for reference. We also need a `div` container element for the `.handle` to be visually styled as a slider track. As a user drags the handle or taps their arrow keys, we use JavaScript to manipulate the handle's CSS `left` positioning with a percentage that reflects the distance the user has dragged, and update the value of our `input` control as well. I've included our new slider markup in bold (FIG 1.21):

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="61" min="0" max="100" />
<div class="slider">
  <a href="#" class="handle" style="left: 61%;"></a>
</div>
```

CSS styling aside, that's the bulk of the behavior a basic slider control needs to perform. But our work isn't done. Our page started out accessible, but with JavaScript we've introduced markup that's playing an unnatural role—that anchor element with a class of `.handle`. When a screen reader encounters this element, it will read it aloud as “number link” because it appears to be an ordinary link with an `href` value of `#`.

To prevent this markup from leading to a confusing experience, we have two options: we can either hide the slider from screen readers (since the text `input` already exists) or do additional work to make the slider itself meaningful to screen readers. I prefer the simplicity of hiding the new control; all we need to do is add an `aria-hidden` attribute to the `div`, which tells a screen reader to ignore the contents of that element when reading the page aloud:

```

<label for="results">Results Shown:</label>
<input type="range" id="results" name="results" »
  value="61" min="0" max="100" />
<div class="slider" aria-hidden>
  <a href="#" style="left: 61%;"></a>
</div>

```

Just like that, we've progressively enhanced our input into a better visual presentation without undermining accessibility. "But... ARIA *what?*" you may ask. Briefly, the W3C's Accessible Rich Internet Applications (ARIA) specification is a set of HTML attributes that embed semantic meaning in HTML elements that play a non-native role—whether that's an `a` acting as a menu button instead of a link (which would use ARIA's `role="button"` attribute) or a `ul` acting as a navigable tree component (the `role="tree"` attribute), as you'd see when browsing a list of files in an operating system window. There's even an ARIA role to describe a slider, if we wanted to go that route with our component above: `role="slider"`. In addition to those role-based attributes, ARIA provides *state* attributes that describe the state a control is in, such as `aria-expanded`, `aria-collapsed`, and `aria-hidden` (used above), and even attributes to describe the current and possible values of a custom slider control. Find out more about ARIA over at the W3C's site (<http://bkaprt.com/rrd/1-17/>).

Make data visualizations accessible

Data visualizations, like charts and graphs, are often delivered in ways that aren't terribly meaningful for those using assistive technology. For example, take a complex line chart in a *New York Times* article, delivered via an `img` element (FIG 1.22).

To a screen reader, all of the information in this chart is invisible. Now, a responsible developer might (at the very least!) go as far as adding an `alt` attribute to describe the chart's data, but such data can be impossible for a single string of text to describe meaningfully:

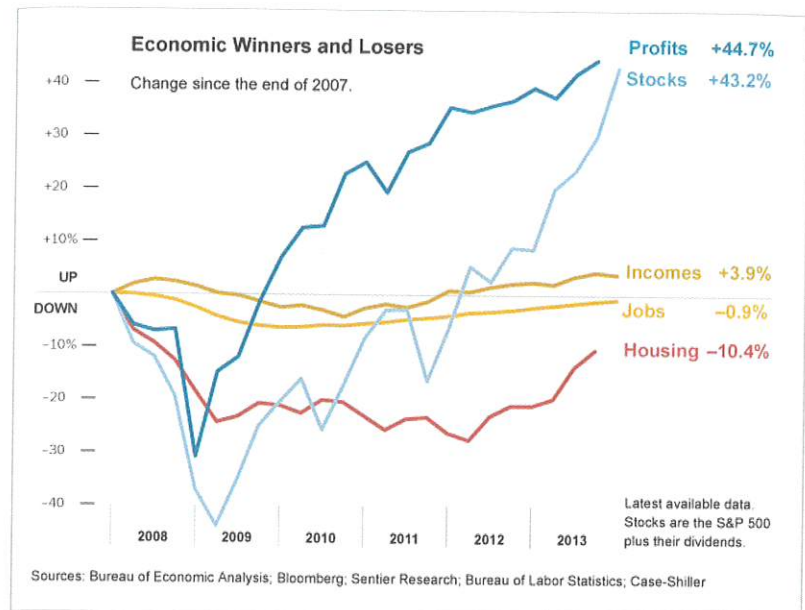


FIG 1.22: This complex line chart was delivered via an `img` element (<http://bkaprt.com/rrd/1-18/>).

```



```

How can we communicate this better? Ready those x-ray specs. As we did with the slider, perhaps we could choose a more meaningful starting point from which to create this graph. Consider the pie chart in FIGURE 1.23, for example. How might we build it in a way that provides more meaning to screen readers than an `img` tag can?

We can start with HTML that's meaningful to all users and present the chart as an enhancement. By peering through the chart to its underlying meaning, we might discover that a chart's bones could be described with an HTML table element. We

Employee Sales Percentages

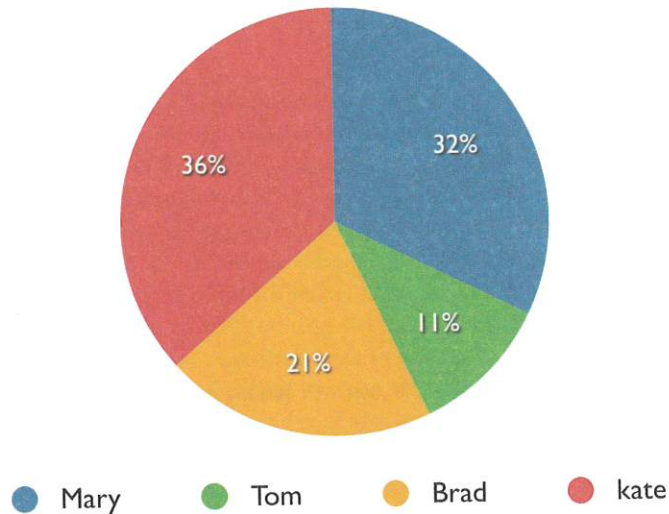


FIG 1.23: How can we meaningfully deliver complex graphics to screen readers?

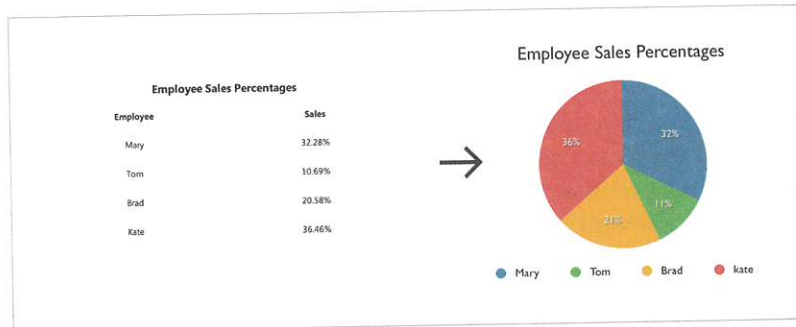


FIG 1.24: A canvas-generated chart visualization of the table on the left.

could then parse the HTML markup below with JavaScript to dynamically draw the chart with a technology like HTML5's `canvas` or SVG. Once the chart is generated, we might even choose to accessibly hide the table by positioning it off screen, deeming the chart a visual improvement over the table it replaces (FIG 1.24).

```
<table>
  <summary>Employee Sales Percentages</summary>
  <tr>
    <th>Employee</th>
    <th>Sales</th>
  </tr>
  <tr>
    <td>Mary</td>
    <td>32.28%</td>
  </tr>
  <tr>
    <td>Tom</td>
    <td>10.69%</td>
  </tr>
  <tr>
    <td>Brad</td>
    <td>20.58%</td>
  </tr>
  <tr>
    <td>Kate</td>
    <td>36.46%</td>
  </tr>
</table>
```

We've only scratched the surface of everything we should consider when building accessible, complex interfaces. But it's hard to go wrong when starting with markup that is valid, accessible, and functional on almost any device, and layer enhancements from there. It's a fine line between an enhancement and a hindrance, one that we as responsible developers must carefully walk.

Building this way is a clear win for access, but planning for such variation makes for an interesting challenge when it comes to communicating these expectations to our clients and QA testers. Perhaps a tweak to how we define support is in order...

An enhanced support strategy

In the article “Grade Components, Not Browsers,” I expanded on a great idea by my colleague Maggie Wachs about defining support granularly for each site component (rather than assigning a grade to a browser as a whole, as is common with approaches like Yahoo’s Graded Browser Support) (<http://bkaprt.com/rrd/1-19/>). The documentation we share with our clients assigns graded levels for each component based on its major tiers of enhancement.

As an example, the following image shows enhancement levels for a property detail component on a real estate website (FIG 1.25). The enhancement level that a browser receives depends on several conditions, such as support of features like Ajax and 3D CSS Transform.

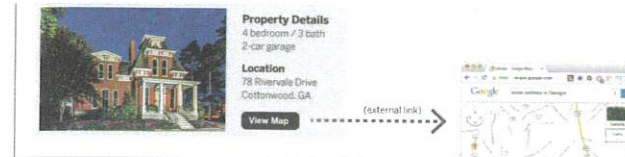
This documentation accomplishes a few things. For one, it helps us to itemize for our clients the particular conditions that enable portions of their site to work at an enhanced level, so everyone (designers, clients, and quality assurance testers) knows what to expect. It also acts as a reminder that some components may receive a higher grade than others, depending on the browser. In other words, feature support varies across even modern browsers, so a browser may receive a bells-and-whistles A-grade experience for one component and a less-enhanced B-grade experience for another.

When we document support this way, we shift the focus from the browser to its features and constraints. We start to think of support as less a binary switch—or even a scale—than a scatter plot. In this system, every browser that understands HTML is supported and is guaranteed access to the site’s primary features and content. As Jeremy Keith points out: “It’s our job to

Property detail component

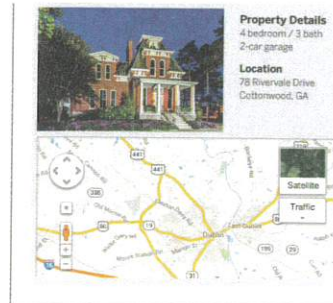
Graded experiences

Grade C: Basic formatting, link to map



Grade B: Same formatting, interactive map in page

Requires: JavaScript, Ajax Support



Grade A: Same formatting, map revealed with 3D flip

Requires: CSS 3D transform support



FIG 1.25: A graded documentation of a feature whose presentation varies across browsers.

explain how the web works...and how the unevenly-distributed nature of browser capabilities is not a bug, it's a feature" (<http://bkaprt.com/rrd/1-20/>).

Speaking of features, we need reliable, device-agnostic, and sustainable ways to detect them. Let's move on to look at the why and the how of doing so.