A BOOK APART

Brief books for people who make websites

No 13

Scott Jehl

RESPONSIBLE RESPONSIVE DESIGN

FOREWORD BY Ethan Marcotte

Publisher: Jeffrey Zeldman
Designer: Jason Santa Maria
Managing Director: Katel LeDû
Editor: Tina Lee
Technical Editor: Ethan Marcotte
Copyeditors: Caren Litherland, Rachel Kaufman
Proofreader: Caren Litherland
Compositor: Rob Weychert
Ebook Production: India Amos

ISBN: 978-1-9375571-6-4

A Book Apart
New York, New York
http://abookapart.com

10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

FOREWORD

THE WEB IS A SUCKER for a good metaphor. In its early days, it was our printing press; as it got older, it was our playground, then our marketplace; now, it's our photo albums, our diaries, our travelogues, our shared moments and videos and GIFs and… and, and, and. What's more, thanks to the explosive popularity of handheld, web-ready devices, the web is accessed more broadly today than at any point in its short lifespan. I think it's fair to say the web is more than the sum of its underpinnings, evolving from a tangle of wires flinging packets over HTTP to a place where we publish, we sell, we connect, we work, and we play.

But here's the thing: the web is far more fragile than we might like to admit. It's fraught with uncertainty—a connection could be dropped, or a network's latency could be too high—which means entire elements of our designs might never reach our users. Of course, it's tempting to see this as a temporary problem, one that'll gradually resolve itself as devices get better, or as networks get stronger. But between the aging infrastructure of developed economies and the popularity of cheaper, low-powered mobile devices in younger, emerging markets, it feels like we're watching a new normal emerge for the web—a medium that's accessed across the planet, but is also much, much slower than we previously thought.

This might sound scary. But that's not how this story ends.

When "mobile" first happened, we were given an opportunity: instead of defaulting to device-specific sites, we realized we could use flexible layouts and media queries to make responsive designs, layouts that could adapt to a nearly infinite array of differently sized screens.

So now, too, we have another opportunity: to ensure our layouts aren't just responsive, but sustainable—fit to deliver compelling content and rich interfaces not only to the latest devices or the widest networks, but to every glowing screen, everywhere.

Thankfully, Scott Jehl is here to show us the way.

I've had the pleasure of working with Scott on a number of responsive redesigns, and I've never encountered a designer who possesses such a keen awareness of—and respect for—the web's fragility. And in this little book, Scott will share that expertise with you, dear reader, teaching you how to build nimble, lightweight interfaces that are ready for the web's volatility.

In the past few years of designing responsively, we've been learning to let go of our need to control the width and height of our layouts. Now, Scott Jehl shows us the next step: to build responsive designs in a responsible way, to ensure they're ready not just for differently sized screens, but for the changing shape of a universal, device-agnostic web.

Let's go.

—Ethan Marcotte

INTRODUCTION

IN EARLY 2012, my wife and I rented an apartment in Siem Reap, Cambodia. She was volunteering at a children's hospital; I was clocking in remotely to build websites with my Filament Group colleagues back in the United States. I worked this way for months as we traveled the region, passing through some of the most resource-strapped places in the developing world—Laos, Indonesia, Sri Lanka, and Nepal. Each stop offered an opportunity to use the web under the same, often constrained conditions that people who live there do. It tested my assumptions as a designer and my patience as a user.

You've likely read that mobile services are the primary means of internet access for many in developing parts of the world, and my casual observations confirmed that. Glass cases displaying mobile devices I'd never seen before filled street market stalls (and helped stock my backpack with test devices). But while seemingly everyone had an internet-capable phone, I was surprised at how frequently people used cell networks to connect other devices to the web. A prepaid SIM card and a USB dongle was the usual means to get a laptop online. So it was for me too.

Using the web this way was an exercise in patience. I wasted hours toggling between partially loaded browser tabs and hitting refresh to watch another web app's loading message spin atop a blank white page, eating away at the limited data I was allotted within my prepaid SIM card. As an advocate of best practices like progressive enhancement and responsive design, I would sometimes indulge in the thought that if only these sites had been built the "right way," these problems wouldn't exist. But if I was honest, I'd concede that many such best practices weren't working as well as they could. Sadly, it appeared that the basic promise of access on the web is one we have yet to fulfill.

I'm not the first to notice. A 2014 *Wired* article described several Facebook executives' experience using their own service during a visit to Nigeria, where over 30% of internet users are on Facebook (http://bkaprt.com/rrd/0-01/):

We fired it up, and we wait... and we wait... It took a really long time. Even simple things like uploading a photo—things most Facebook users do—just weren't working. That was a pretty harsh experience for us. We'd been building an app for users like us. But we were the exception, not the rule.

We web developers tend to be an exceptional bunch. Our work demands fast, reliable networks to stream enormous amounts of data, and we have access to the latest, most capable devices. But while many of us work in relatively ideal conditions, we can't just build for users like us; we can't forget that for most of the world, the web doesn't function like this.

You may think, "But that's not my audience." And you may be right, but consider that more of the world's web traffic this year will come from cheaper, under-featured devices in emerging markets (http://bkaprt.com/rrd/0-02/). Even in some of the most developed regions, mobile connections are often slow, intermittent, and unreliable, as data plans become more expensive and limited. A quick Twitter search confirms that London's notoriously bad cell service persists, and heck, I rarely get better than an ancient EDGE connection where I live in Florida—*EDGE!*

Accessing the web reliably and efficiently isn't a given for many of our neighbors, our users, our customers. As web designers, we're well poised to improve this situation. I mention *customers* to emphasize that pushing for better access is not only an appeal for empathy, but also an opportunity to expand the reach of our services, making them more resilient for everyone.

This book is about accessibility: broadening access to the services we make without compromising features that push the web ahead. Diversity is a defining feature of the web, not a bug. We should strive to make our content and services accessible to all capable devices. If that sounds hard, well, sometimes it is. I intend to convince you that it's possible, and that it's worth it.

Let's debrief on what our users are up to, shall we?

Our diversifying web

The proof is in the numbers. In 2011, Apple sold more iOS devices than all of the computers it sold in twenty-eight years

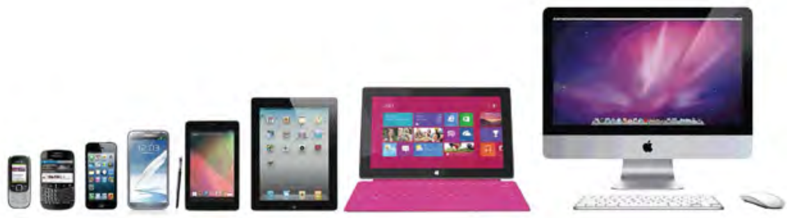


FIG 0.1: A sampling of the variety of screen sizes we now need to support.

(http://bkaprt.com/rrd/0-03/). In 2013, global mobile data usage grew by 81% (http://bkaprt.com/rrd/0-04/). As of January 2014, 58% of Americans owned a smartphone and 42% owned a tablet, four years after the iPad's release (http://bkaprt.com/rrd/0-05/). The speed of such growth is astonishing, but it's not just mobile.

Our devices represent a broadening spectrum of form factors, feature support, environmental constraints, and uses (**FIG 0.1**). The variance in screen size alone is staggering—consider this graphic overlaying the screen dimensions of the twenty most-used devices in early 2013 (**FIG 0.2**).

Screen dimensions say nothing about a display's resolution, which may be higher than standard definition; nor do they predict a browser's viewport size, which often differs from the screen's. As designer Cennydd Bowles says, when you consider the near-infinite variability of browser viewports, the sizes we need to care about are even broader in range than screen dimension rankings might suggest (**FIG 0.3**).

Now that's fragmentation! Luckily, the problem of delivering a design that adapts fluidly across various viewport sizes has been more or less solved.

Responsive design: a responsible starting point

"This is our way forward. Rather than tailoring disconnected designs to each of an ever-increasing number of web devices, we can treat them as facets of the same experience."
—ETHAN MARCOTTE, "Responsive Web Design," *A List Apart*

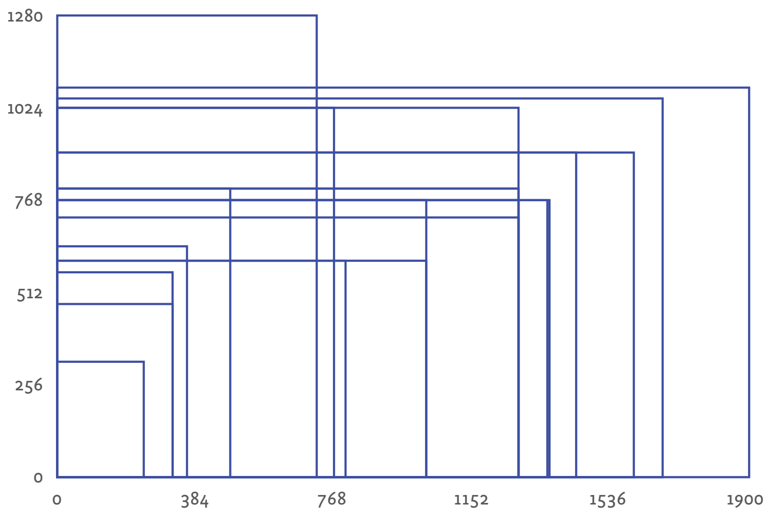


FIG 0.2: Wildly varying screen sizes of the top twenty most popular devices (http://bkaprt.com/rrd/0-06/).

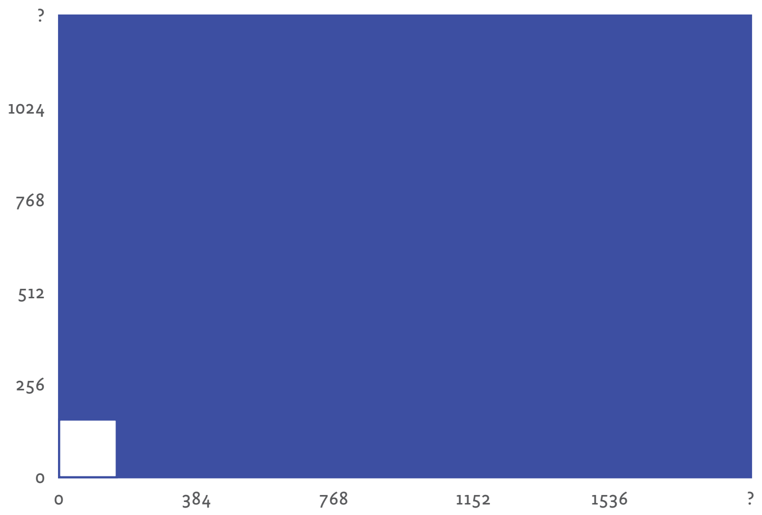


FIG 0.3: Viewport size fragmentation on the web, redrawn from a tweet by Cennydd Bowles (http://bkaprt.com/rrd/0-07/).

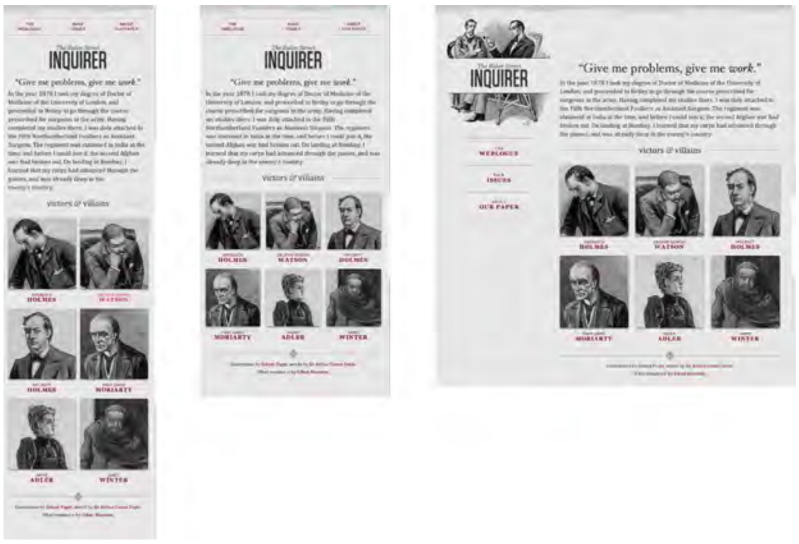


FIG 0.4: Ethan Marcotte's example of a responsive layout from his *A List Apart* article (http://bkaprt.com/rrd/0-08/).

In 2010, Ethan Marcotte coined the term responsive web design to describe an approach to web layout that combines fluid grids, fluid images, and CSS3 media queries to deliver layouts that respond (downright magically) to their environment (**FIG 0.4**).

If this book's title led you to believe that responsive web design is not responsible, now is a good time to clarify. Responsive web design is responsible web design. The end. Thank you for reading!

Okay, seriously, I'll explain.

Marcotte's clever combination of web standards technologies gives us a sustainable way to deliver cross-device visual layouts. But Marcotte would be the first to emphasize that responsive layout is one of many variables we must consider when building multi-device sites and applications. Layouts are the start. We need to expand beyond the viewport and consider how we support myriad device capabilities, how we retain accessibility in complex interfaces, and how we deliver assets over the wire.

As Trent Walton says in his essay “Device Agnostic”: “Like cars designed to perform in extreme heat or on icy roads, websites should be built to face the reality of the web’s inherent variability” (http://bkaprt.com/rrd/0-09/). Thankfully, being responsive from a layout perspective does not preclude us from being responsive from a performance, interactivity, or accessibility perspective.

Responsive and responsible

To deliver on the promise of a broadly accessible, delightful, and sustainable web, we need to combine responsive design with additional responsible practices. A responsible responsive design equally considers the following throughout a project:

- **Usability**: The way a website’s user interface is presented to the user, and how that UI responds to browsing conditions and user interactions.
- **Access**: The ability for users of all devices, browsers, and assistive technologies to access and understand a site’s features and content.
- **Sustainability**: The ability for the technology driving a site or application to work for devices that exist today and to continue to be usable and accessible to users, devices, and browsers in the future.
- **Performance**: The speed at which a site’s features and content are perceived to be delivered to the user and the efficiency with which they operate within the user interface.

That’s comprehensive, no? While I ponder renaming this book *Welcome to the Internet by Scott*, let’s take a deeper look at some challenges we face in delivering responsibly.

Designing for usability: sensors, input mechanisms, and interactivity

Gone are the days of building websites that only need to work with a mouse (if those days ever existed). We need to care about things like touch, keyboard, stylus, and more—which we may

FIG 0.5: The Windows 8 OS runs on devices that support several input modes, from touch to mouse to keyboard. Photograph by Kārlis Dambrāns (http://bkaprt.com/rrd/0-10/).

encounter in a mix of mobile devices, tablets, or laptops. Many of the most popular devices we use now support touch interactivity. For instance, the Windows 8 operating system supports touch interaction on both laptops and tablets (**FIG 0.5**). Microsoft Kinect tracks hand and arm gestures in midair (**FIG 0.6**). In response to these new input mechanisms, we can't rely solely on traditional mouse cursor interactions like hover; instead, our interfaces must be ready to respond to various input mechanisms within our multi-device universe.

A disparity often exists between powerful native applications and the limited APIs we see on the web and, in truth, that can be a barrier to building web-based applications. Fortunately, many browsers are quickly gaining access to native operating system features like GPS location, contacts, calendar, notifications, file systems, and the camera. These standardized interfaces allow us to communicate with local device features without using plugins like Flash or Java, which rarely work across devices anyhow. In addition to local data APIs, browsers are increasingly able

FIG 0.6: The Microsoft Kinect tracks full-body movement, which may hint at future interactive models for the web. Photograph by Scott and Elaine van der Chijs (http://bkaprt.com/rrd/0-11/).

to access information from device sensors like proximity, GPS, accelerometer, battery levels, and even ambient light. With each new feature, the web platform gains a foothold.

Building for access: considering assistive technology and cross-device continuity

Because assistive technologies increasingly come preinstalled on devices, we often need to take steps to ensure that our sites retain their meaning when browsed in non-visual contexts. Now standard on all Apple computers and iOS devices, screen-reading software VoiceOver sits atop the browser and provides gesture-based navigation as it reads a page aloud. Its multitouch rotor-gesture system offers ways to navigate the web via things like headings and links, which gives us more reasons to be vigilant about the markup we use to communicate our content (**FIG 0.7**).

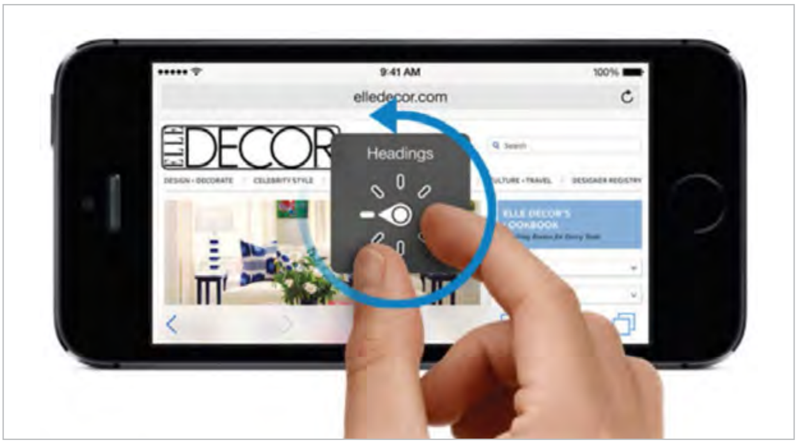


FIG 0.7: VoiceOver rotor on the iPhone (http://bkaprt.com/rrd/0-12/).

Assistive technology isn't only for those with permanent disabilities; voice and audio may be the preferred and safest interaction modes for any user in certain circumstances. Perhaps the most widely used screen reader is Apple's Siri, which is handy for people who are temporarily unable to look at their screen (while driving, for example) or prefer the convenience of voice interaction over touch typing. As web applications continue to make inroads on our native operating systems, we can expect that software like this will only become more prevalent.

Beyond delivering a usable experience in isolated contexts, we should keep in mind that people increasingly hop from one device to another and expect consistently accessible content. The 2012 Google study *The New Multi-Screen World* revealed that people use multiple devices throughout a single day, in many cases to complete a single task (**FIG 0.8**). The study found that 65% of shoppers who add an item to their cart on a handheld device later complete their transaction on a laptop computer. Perhaps the shoppers were interrupted by a call or preferred to go through the checkout process on a device with a keyboard. Whatever the reason, we must meet our users wherever they happen to be.

Google

The New Multi-screen World:

Understanding Cross-platform Consumer Behavior

U.S., August 2012

FIG 0.8: Google's 2012 study *The New Multi-Screen World* (http://bkaprt.com/rrd/0-13/).

Browsers: what's old is new again

While modern browsers like Google Chrome, Firefox, and, yes, even Internet Explorer press ahead with new features, many devices in the wild and in stores are locked into browsers that are no longer in development. For example, version 2 of the Android operating system continues to be incredibly popular worldwide, despite being more than two major releases behind the latest and including a built-in browser that hasn't seen updates since 2011 (http://bkaprt.com/rrd/0-14/)! Seasoned (read: old) developers like me may recall a similar situation with IE6's drawn-out reign. Unfortunately, long-term support of browser versions is inevitably cut in favor of the next new platform or a company's shifted priorities, leaving existing users in the lurch.

Browser lock-in presents challenges, then—but also opportunities. Users often seek out other browsers for their platforms, some of which offer unusual features and selling points. For example, millions of people who prefer web pages that load faster and consume less of their data plans (crazy, right?) choose a browser like Opera Mini, which requests web content through remote proxy servers that optimize each page's download size (**FIG 0.9**). Proxy-based browsers support little or no JavaScript

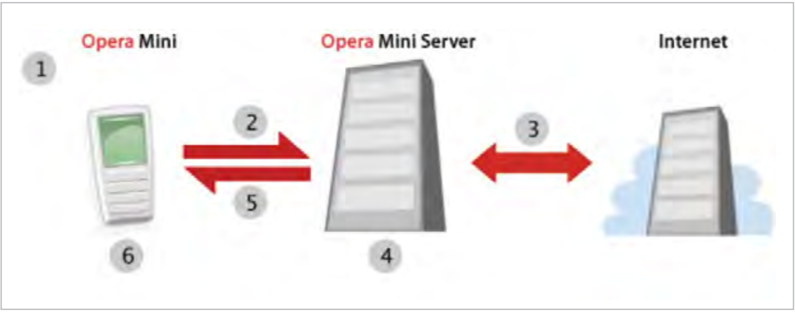


FIG 0.9: An Opera infographic demonstrating how the Opera Mini browser accesses the web (http://bkaprt.com/rrd/0-15/).

interactivity on the device itself; ironically, practices that benefit older browsers, such as delivering functional HTML, also help the millions running these new proxy browsers!

Prioritizing performance: networks, weight, and performance impact

Mobile network constraints grow more nuanced and challenging, even as they slowly improve as a whole. In the meantime, network connections are intermittent and lagging even in developed countries. To ensure high performance, we must reassess the ways we deliver our assets, reduce the weight and number of those assets, and remove potential points of failure that block access to our content.

Any unused code we deliver wastes our users' time and money, and we have plenty of room for improvement. In his April 2013 post "What are Responsive Websites Made Of?," Guy Podjarny evaluated file transfer sizes of 500 responsive websites and found that 86% sent roughly equivalent assets to all viewport sizes (http://bkaprt.com/rrd/0-16/). Rather than serving optimized images, for example, sites were delivering large-screen images (thus relying on browsers to scale them down for smaller screens), along with plenty of CSS, JavaScript, and other assets that were only necessary in some contexts.

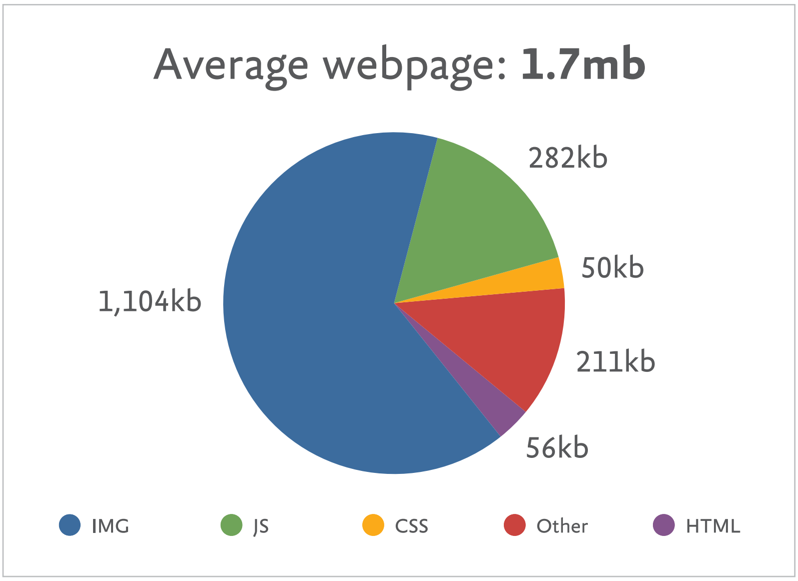


FIG 0.10: Average website weight, April 2014 (http://bkaprt.com/rrd/0-17/).

Of course, the web's burgeoning weight problem isn't exclusive to responsive design. It has festered in the fixed-desktop web for some time; as of April 2014, the average website weighed a whopping 1.7 megabytes (**FIG 0.10**).

Unoptimized, heavy websites can mean long load times for users. A 2012 StrangeLoop survey of the Alexa Top 2000 sites showed that the average load time was six to ten seconds in Internet Explorer 7 with a Wi-Fi connection (never mind how slow it may be on a mobile connection) (**FIG 0.11**)! The cost of poor performance translates directly to users and, as a result, to businesses. In 2012, Walmart found that for every second shaved off load time, it gained 2% in conversions; incremental revenue went up 1% with each 100 millisecond decrease (http://bkaprt.com/rrd/0-18/).

In addition, as cell network speeds and reliability continue to constrain mobile use, the cost of data itself has become more prohibitive. If you purchased an iPhone from the Apple store

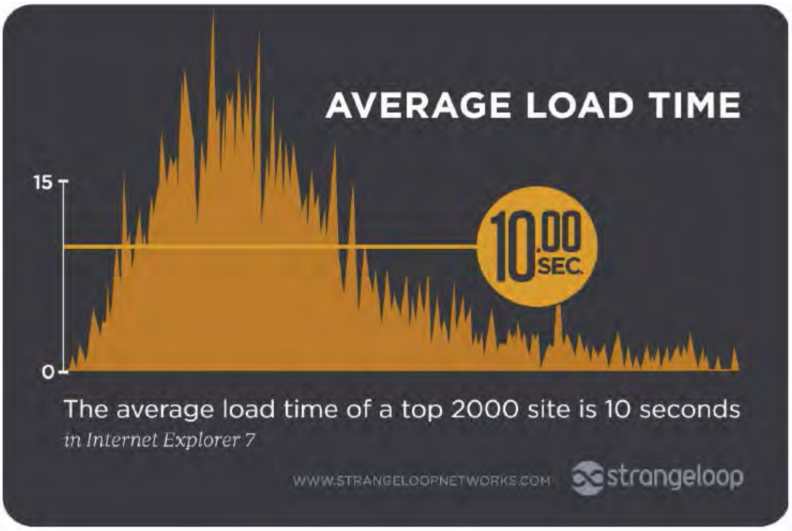


FIG 0.11: A site's transfer size affects how quickly (or slowly) it loads (http://bkaprt.com/rrd/0-19/).

in the United States in 2014, the cheapest Verizon plan at $60/month netted you only 250 megabytes of data. In light of that 1.7-megabyte average page weight, it's easy to see how quickly your monthly quota is spent.

Because responsible network use is so vital to performance, this book dedicates entire chapters to minimizing our delivery weights *and* optimizing the ways we deliver code so that sites are usable as fast as possible.

Embracing unpredictability

The web has always been a hostile design medium. As multi-device use rises, scenarios like low bandwidth, small screens, unpredictable screen orientation, or non-visual browsing aren't exceptions anymore; they're everyday contexts. Designing for an inflexible set of conditions leads to problems for our users: the interfaces we design and develop are increasingly interacted with in ways we can neither predict nor control.

To deliver on today's web successfully, we must think responsively down to the smallest detail and prepare our code to counter the unexpected. And to do that, we need to consider both past and possible usage patterns. The need for responsive sites that prioritize performance, access, usability, and sustainability is clear, but executing on those goals is no easy task. Throughout this book, we'll explore the challenges we face as we implement responsive design. By following responsible practices and patterns, we can eliminate many accessibility and performance issues before they occur to deliver appropriate, optimized experiences, regardless of a browser's features and constraints.

Our goal is to create delightful, inclusive experiences—a tall order—so let's forge ahead and get acquainted with some approaches to the challenges we face.

1

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/rrd/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.

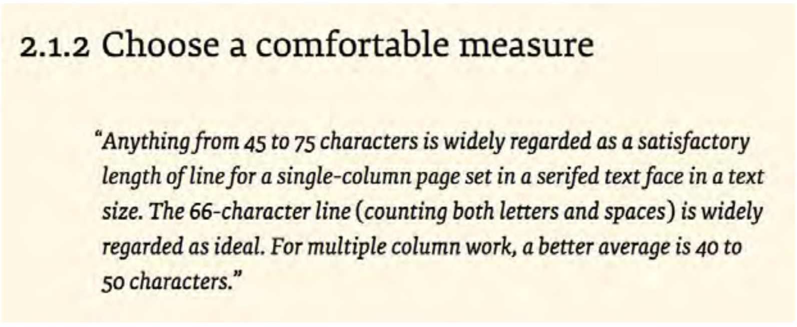
"Start with the small screen first, then expand until it looks like shit. TIME FOR A BREAKPOINT!"

—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."

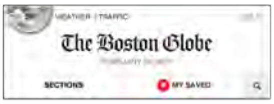
FIG 1.1: Here, a seventy-character line length makes for comfortable reading (http://bkaprt.com/rrd/1-04/).

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.

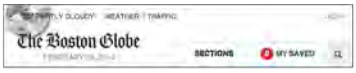
Let's look at an example. On the *Boston Globe* website, we have two or three major layout breakpoints, but the more complicated components break more often. The site's masthead component has four major breakpoints, as well as some minor ones for slight adjustments to prevent text wrapping (**FIG 1.2**).

Design modularly

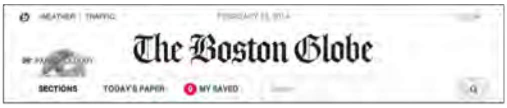
As in the masthead example, I find it helpful to compile the multiple configurations of each modular component in isolation; that way, I can test its usability and document its variations in one place. Developer Dave Rupert of Paravel explored this concept in his post "Responsive Deliverables" (http://bkaprt.com/rrd/1-06/).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



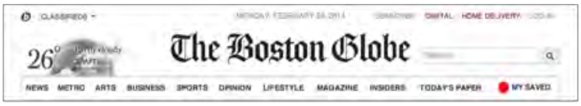
First breakpoint: navigation and search options toggle on tap.



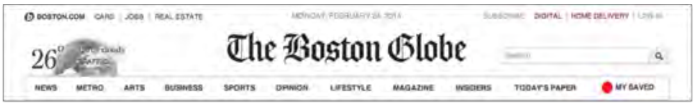
Second breakpoint: logo moves left to split the width with the navigation.



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 in **FIGURE 1.3**, 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

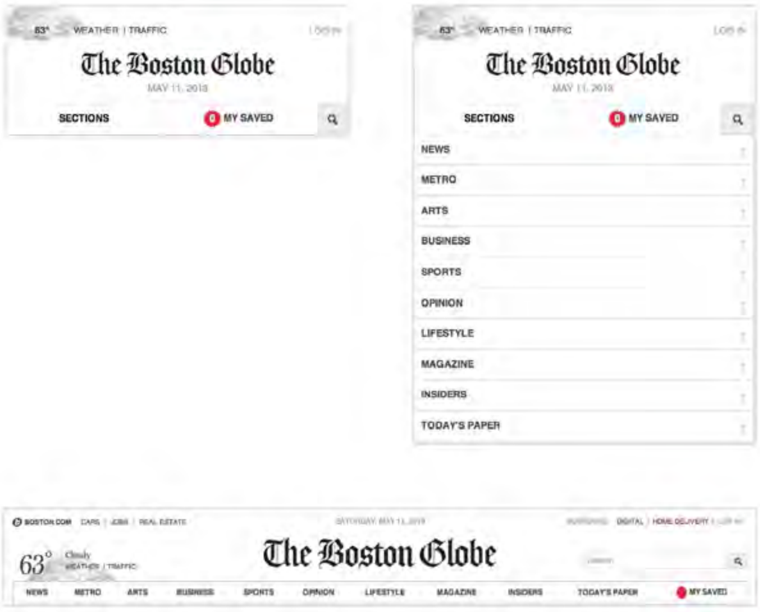


FIG 1.3: *Boston Globe*'s navigation uses progressive disclosure on small viewports.

(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

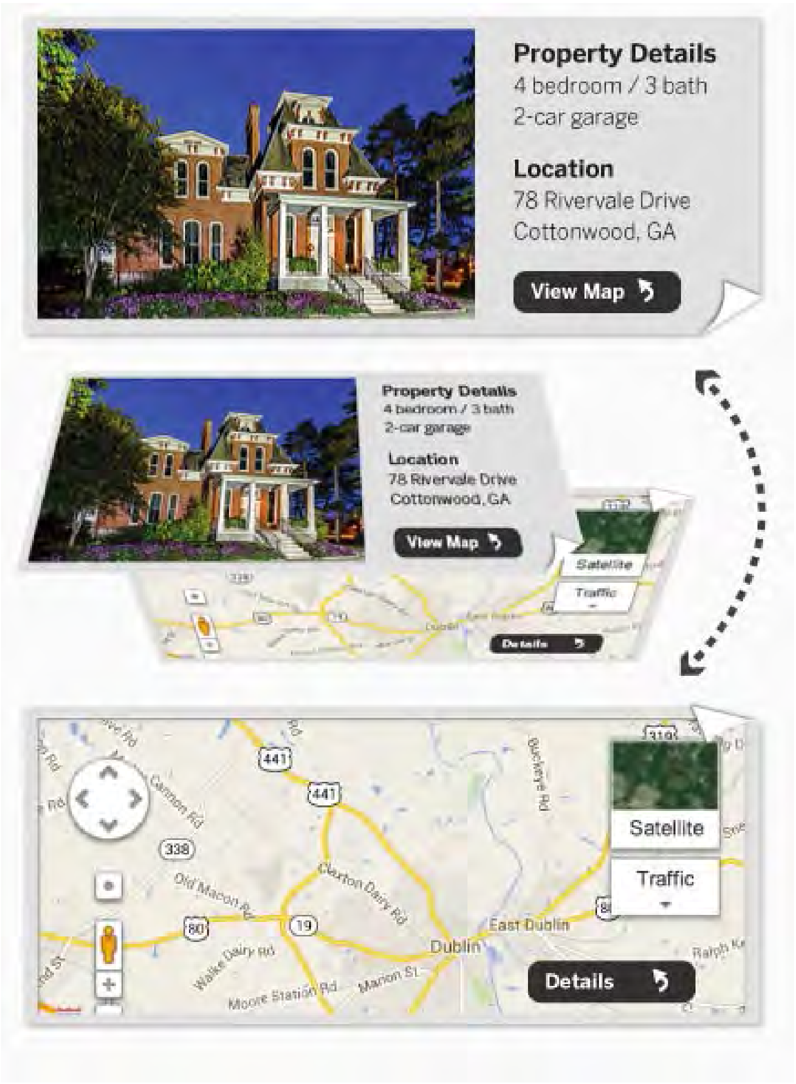


FIG 1.4: Progressively disclosed content flips in 3D to display more information.

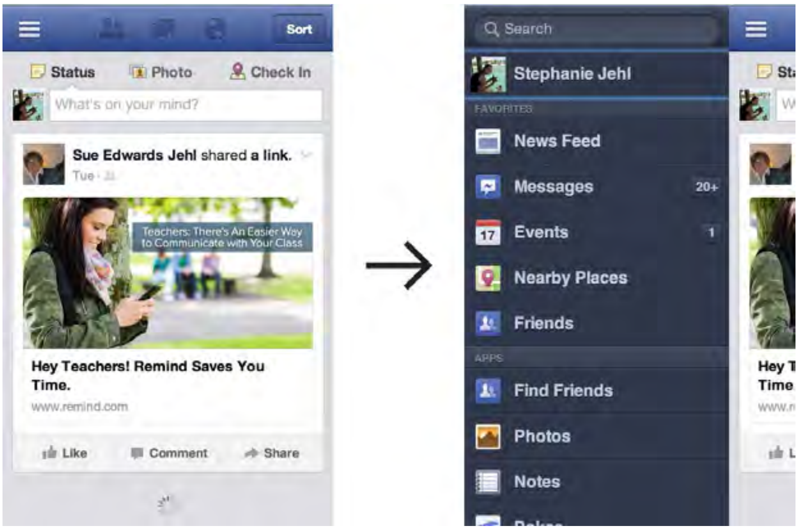


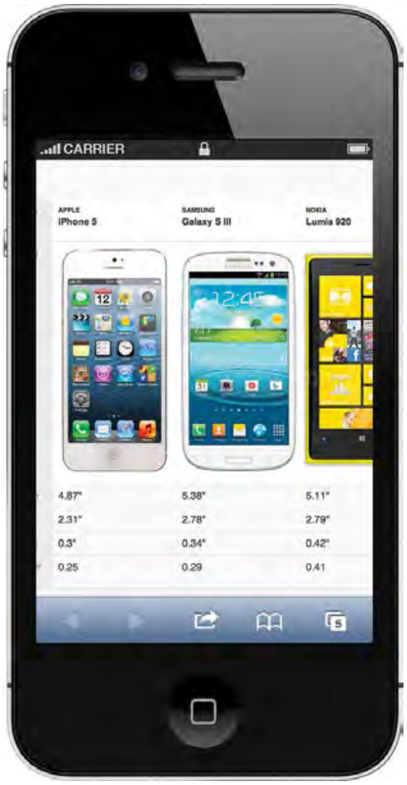
FIG 1.5: Tapping the menu icon reveals Facebook's off-canvas navigation from the screen's left edge.

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

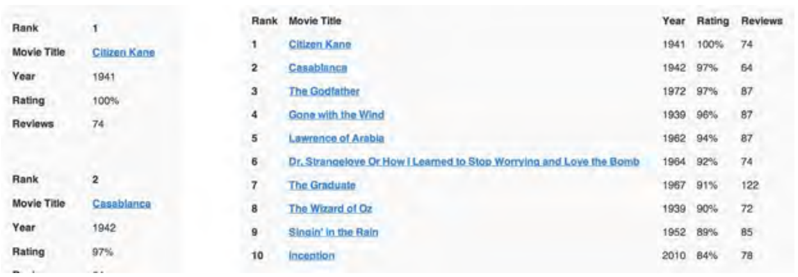
At Filament, we experimented a lot and found a couple of patterns that worked well enough to include in the jQuery Mobile framework. The first pattern, Reflow (http://bkaprt.com/rrd/1-09/), reformats the table from a multi-column view to a list view; each cell becomes its own row, with a row header to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



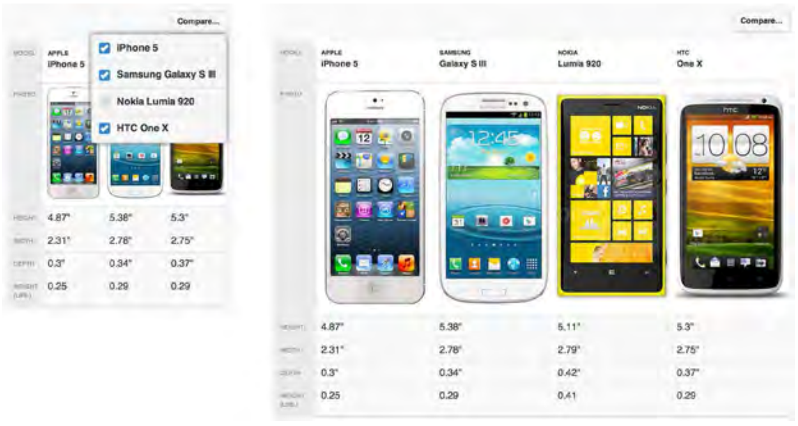
APPLE iPhone 5	SAMSUNG Galaxy S III	NOKIA Lumia 920
4.87"	5.38"	5.11"
2.31"	2.78"	2.79"
0.3"	0.34"	0.42"
0.25	0.29	0.41



Rank	1
Movie Title	Citizen Kane
Year	1941
Rating	100%
Reviews	74
Rank	2
Movie Title	Casablanca
Year	1942
Rating	97%

Rank	Movie Title	Year	Rating	Reviews
1	Citizen Kane	1941	100%	74
2	Casablanca	1942	97%	64
3	The Godfather	1972	97%	87
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
7	The Graduate	1967	91%	122
8	The Wizard of Oz	1939	90%	72
9	Singin' in the Rain	1952	89%	85
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.



MODEL	APPLE iPhone 5	SAMSUNG Galaxy S III	HTC One X
PHOTO			
HEIGHT	4.87"	5.38"	5.3"
WIDTH	2.31"	2.78"	2.75"
DEPTH	0.3"	0.34"	0.37"
WEIGHT (LBS.)	0.25	0.29	0.29

MODEL	APPLE iPhone 5	SAMSUNG Galaxy S III	NOKIA Lumia 920	HTC One X
PHOTO				
HEIGHT	4.87"	5.38"	5.11"	5.3"
WIDTH	2.31"	2.78"	2.79"	2.75"
DEPTH	0.3"	0.34"	0.42"	0.37"
WEIGHT (LBS.)	0.25	0.29	0.41	0.29

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 menus.

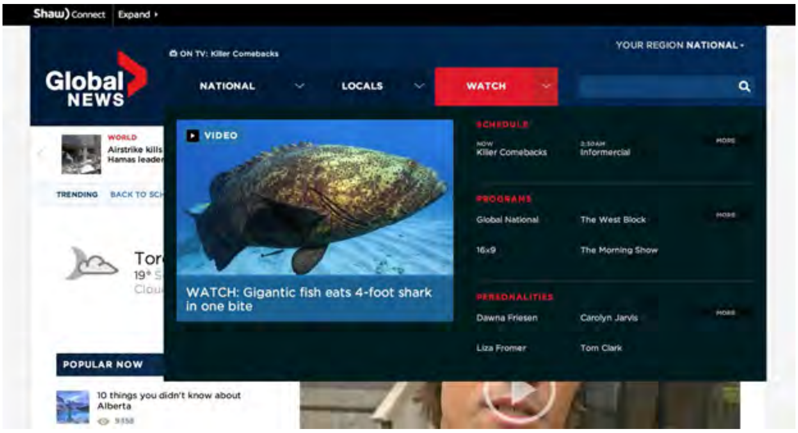


FIG 1.9: The split-button menus on GlobalNews.ca work for touch and `mouseover`.

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.

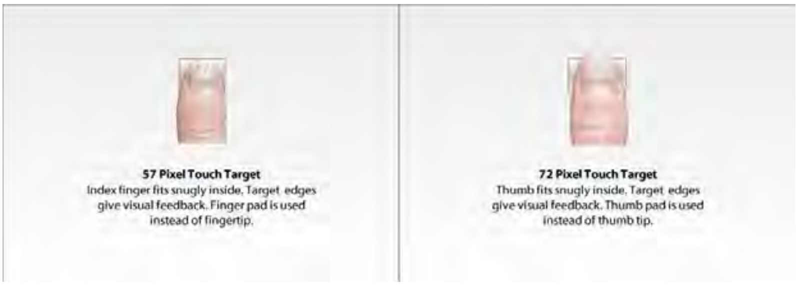

57 Pixel Touch Target
Index finger fits snugly inside. Target edges give visual feedback. Finger pad is used instead of fingertip.
72 Pixel Touch Target
Thumb fits snugly inside. Target edges give visual feedback. Thumb pad is used instead of thumb tip.

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?

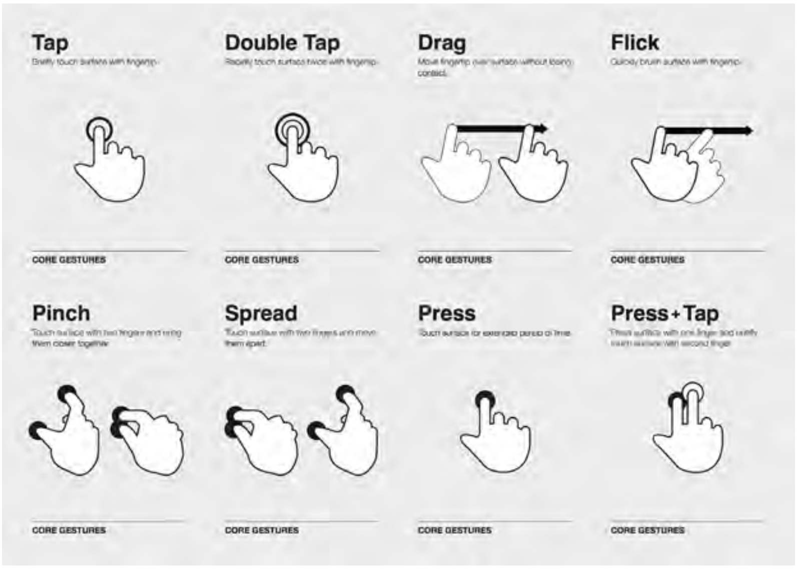
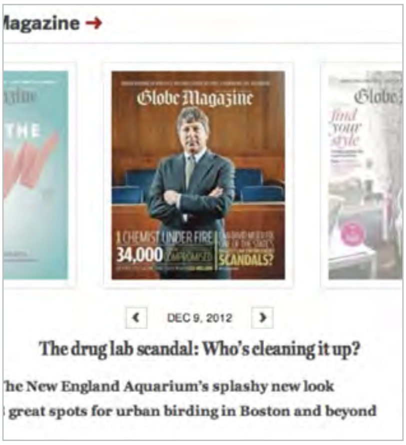

Tap
Briefly touch surface with fingertip.
Double Tap
Rapidly touch surface twice with fingertip.
Drag
Move fingertip over surface without losing contact.
Flick
Quickly brush surface with fingertip.
CORE GESTURES
Pinch
Touch surface with two fingers and bring them closer together.
Spread
Touch surface with two fingers and move them apart.
Press
Touch surface for extended period of time.
Press+Tap
Press surface with one finger and briefly touch surface with second finger.
CORE GESTURES

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.



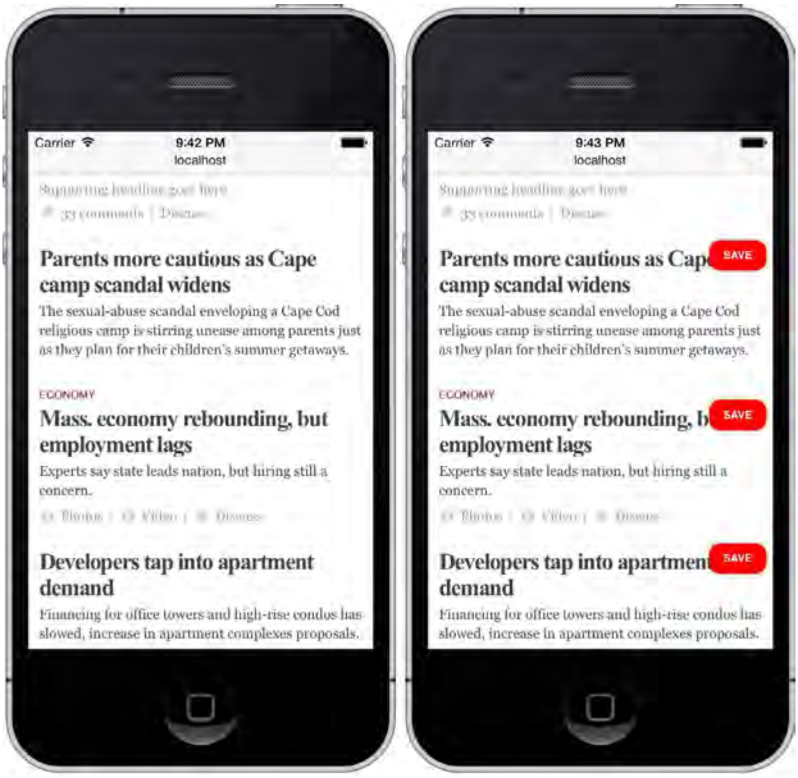
The drug lab scandal: Who's cleaning it up?
he New England Aquarium's splashy new look
great spots for urban birding in Boston and beyond

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


Carrier 9:42 PM
localhost

Supporting headline goes here
33 comments | Discuss

Parents more cautious as Cape camp scandal widens
The sexual-abuse scandal enveloping a Cape Cod religious camp is stirring unease among parents just as they plan for their children's summer getaways.

ECONOMY
Mass. economy rebounding, but employment lags
Experts say state leads nation, but hiring still a concern.
Photos | Video | Discuss

Developers tap into apartment demand
Financing for office towers and high-rise condos has slowed, increase in apartment complexes proposals.

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, barebones 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

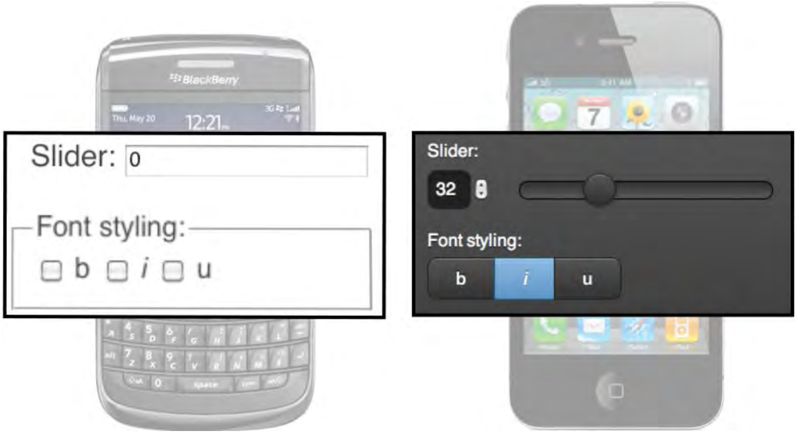


FIG 1.14: A view (left) of the underlying native controls behind an enhanced user interface (right).

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

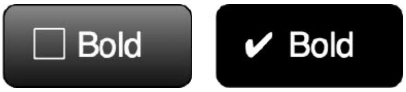


FIG 1.15: A standard input and label styled as a button.

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

```
<label class="check">
  <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.

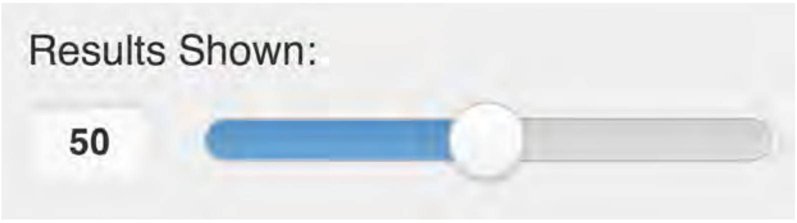


FIG 1.16: A custom slider control with a numerical input.

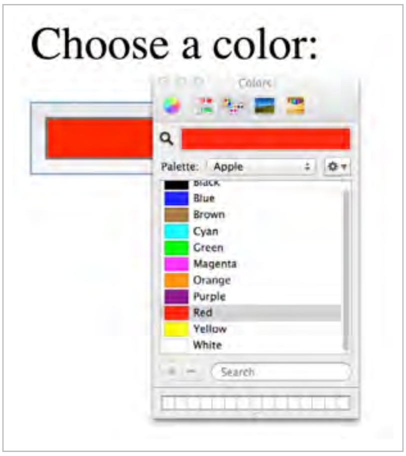


FIG 1.17: A color input with a color picker in Google Chrome.

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">
```

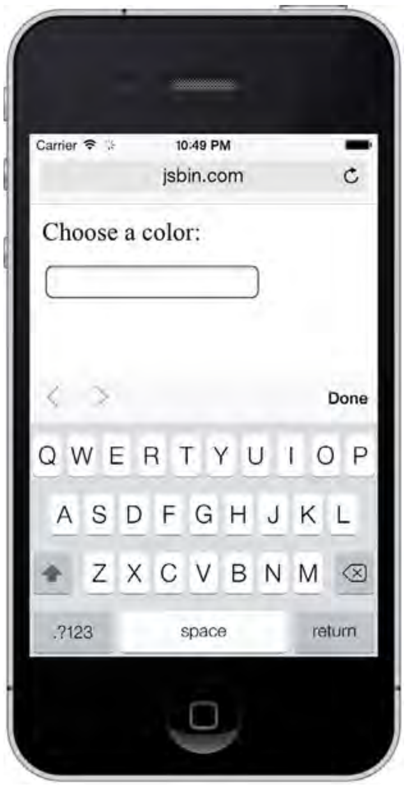


FIG 1.18: A color input falls back to a plain text input in iOS 7.

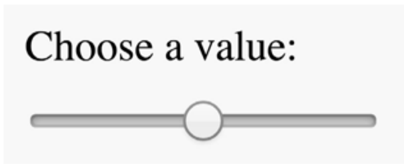


FIG 1.19: A `range` input rendered in iOS 7 Safari, which gives no feedback about minimum, maximum, or current value.

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.

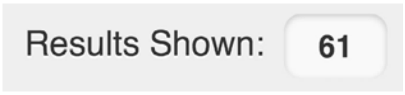


FIG 1.20: Our foundational HTML markup as rendered in a browser.

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

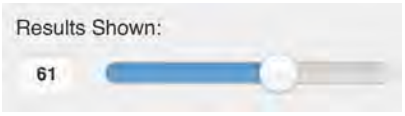


FIG 1.21: Our slider div, with the input element shown to the left.

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:

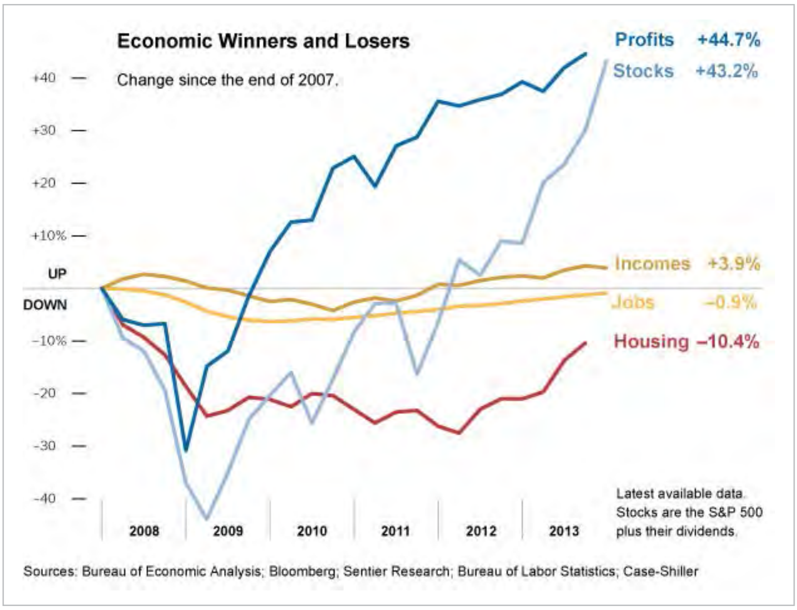


FIG 1.22: This complex line chart was delivered via an `img` element (http://bkaprt.com/rrd/1-18/).

```
<img src="chart.png" alt="Economic winners and losers, »
  Change since...">
```

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

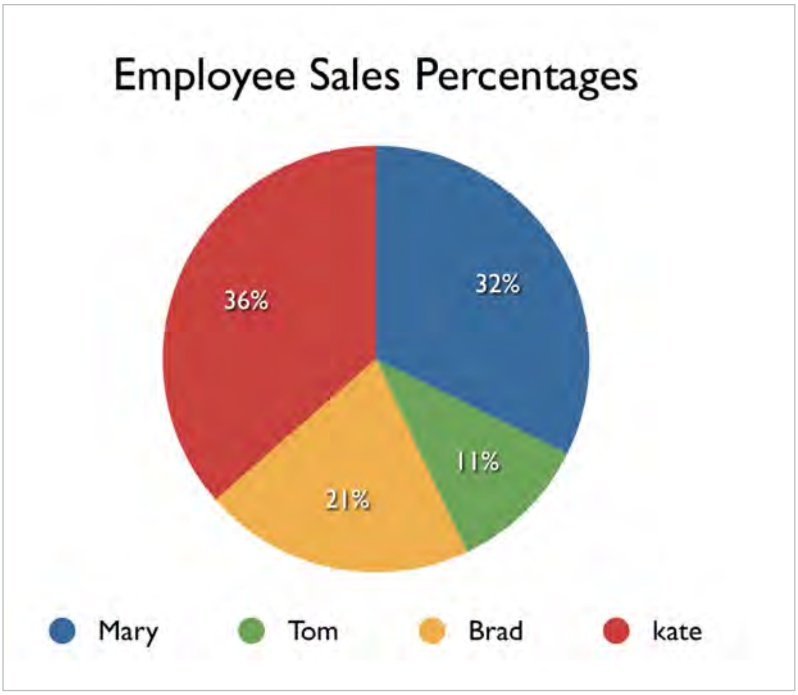
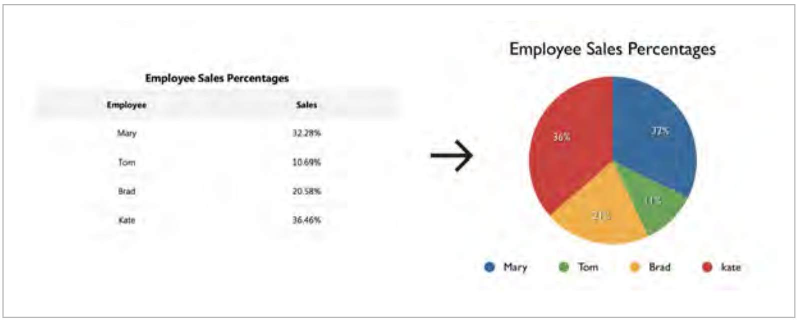


FIG 1.23: How can we meaningfully deliver complex graphics to screen readers?



Employee Sales Percentages

Employee	Sales
Mary	32.28%
Tom	10.69%
Brad	20.58%
Kate	36.46%

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

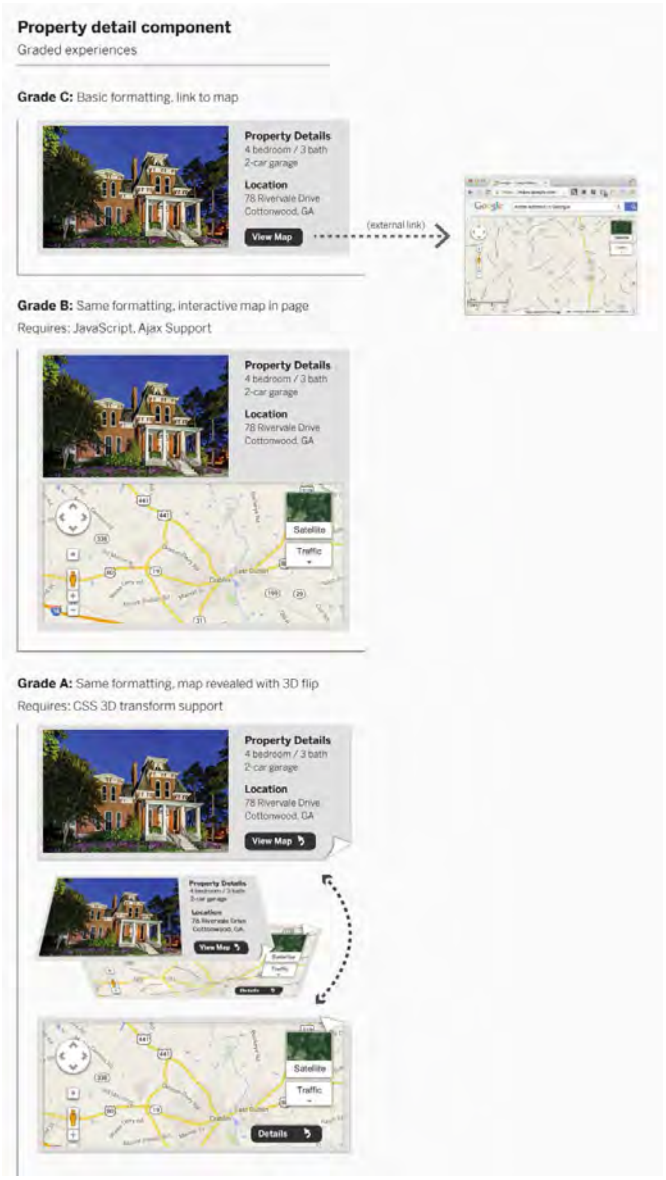

Property detail component
Graded experiences
Grade C: Basic formatting, link to map
Property Details
4 bedroom / 3 bath
2-car garage
Location
78 Rivervale Drive
Cottonwood, GA
View Map
(external link)
Grade B: Same formatting, interactive map in page
Requires: JavaScript, Ajax Support
Satellite
Traffic
Grade A: Same formatting, map revealed with 3D flip
Requires: CSS 3D transform support
Details

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.