2 SUSTAINABLE DETECTION

GIVEN THE DIVERSE NATURE of browsers today, the ability to detect browser features and constraints is vital to delivering an appropriate user experience. We have many ways to approach this detection, some more responsible than others.

DEVICE DETECTION: THE EVOLUTION OF A STOPGAP

Among topics of great debate in web development, perhaps the most contentious is the practice of device detection. Its mere mention in a gathering of peers gets my stomach tingling in anticipation of the fiery opinions that await. In truth, a little device detection is sometimes necessary in a complex cross-device codebase, but with each site I build, I find fewer reasons to use it.

This is a good thing, as any approach that includes device-specific logic risks threatening the sustainability of our codebase in the long term. Let's explore some reasons why that is.

Detecting all the things

When a user first requests a page, we know precious little about their browsing environment. We don't know the size of their device's screen, or if their device even has a screen. We don't know their browser's capabilities. Fortunately, we can detect these qualities after delivering code to the browser, but in some cases that's later than we'd prefer.

One thing we can universally detect upon first arrival is a browser's user agent information, included in every request that a browser—or user agent—makes. This string of text packs a variety of information, including the browser's make and version, like Firefox 14 or Chrome 25, and its operating system, like Apple iOS. Crafty developers realized early on that if they gathered data about various browsers and their capabilities and stored them on their server (in what's known as a device database), they could query that information when a user visits their site to get a good idea of the sort of browser they're dealing with. This process is called *user agent sniffing* or, more broadly, device detection.

Sniffing up the wrong tree

Perhaps the most common criticism of user agent sniffing is that the information a browser provides isn't always reliable. Browsers, networks, and even users sometimes modify user agent information for myriad reasons, which makes it difficult to know if you're dealing with the browser you think you are. Let's start with a few popular mobile browsers' preference panels: Android's default browser, Opera Mini, the BlackBerry browser, and others provide an easy means of changing the name the browser reports itself as. You'll sometimes see this disguised as "Request desktop site" or with more granular settings like those in the Android browser, but the ability to change user agent information exists to give users the tools to fight against sites that deliver limited content and functionality to particular browsers (**FIG 2.1**).

Similarly, a browser's default user agent string is crowded with mentions of other browsers in hopes that they will prevent

FIG 2.1: Android, Opera, and Firefox user agent settings.

its users from being locked out of the best versions of certain sites. For example, in addition to several appropriate bits of information, the UA string of my current browser (Chrome 34) mentions Mozilla, Webkit, KHTML, Gecko, and Safari—all terms describing non-Chrome browsers:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_5)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/34.0.1847.131 Safari/537.36
```

Some browsers go even further and deliberately obscure information in their user agent string to trick sites into sending them the experience other browsers get! The user agent string for the vastly improved Internet Explorer 11 never mentions Internet Explorer; instead, it tries to trick device-detection libraries into thinking the browser is Firefox or Webkit, which developers came to recognize as the only browsers that support advanced features necessary to deliver a better experience. (In recent versions of IE, this is thankfully no longer true.) In her *A List Apart* article "Testing Websites in Game Console Browsers," Anna Debenham notes a similar situation with the Sony PlayStation Vita's browser: "The Vita's browser is a WebKit-based version of NetFront. Strangely, it identifies itself as Silk in its user agent string, which is the browser for Amazon's Kindle Fire" (http://bkaprt.com/rrd/2-01/) (**FIG 2.2**).

Browser developers have an interest in ensuring the survival of their software. Ironically, the more web developers deliver their content and features unevenly based on user agent

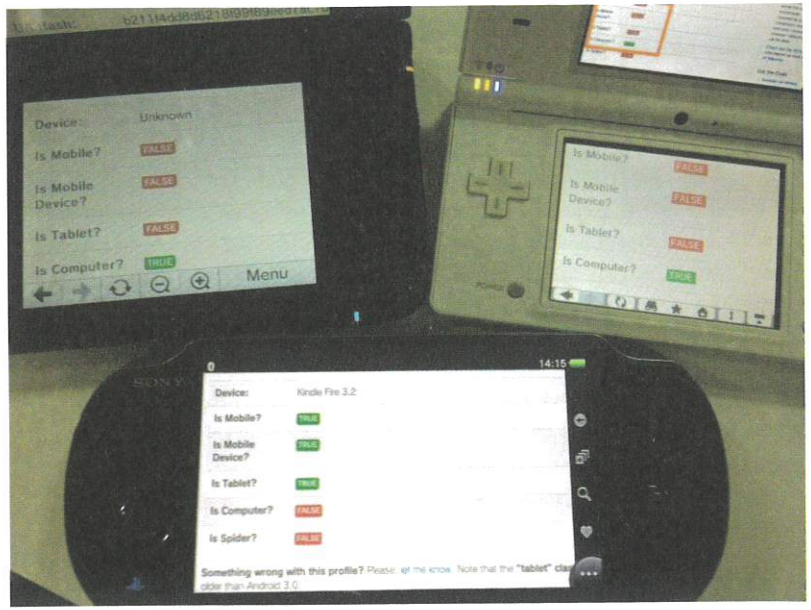

FIG 2.2: Messy device detection results across mobile devices (http://bkaprt.com/rrd/2-02/).

information, the less meaningful user agent information will continue to be.

"Set it and forget it?" Forget it!

But reliability is a minor problem compared to sustainability. We can only write detection logic against browsers and devices that exist now, which makes device detection utterly useless for gracefully accepting new browsers.

Most critically, relying too heavily on device detection can lead us to make dangerous assumptions based on information that's not always up to date. Device detection provides, at best, stock information about a device or browser, meaning any optimizations we make based on that static data may not reflect the live, dynamic nature of a user's actual browsing environment.

These are some examples of variables that a device database can never accurately convey (FIG. 2.3):

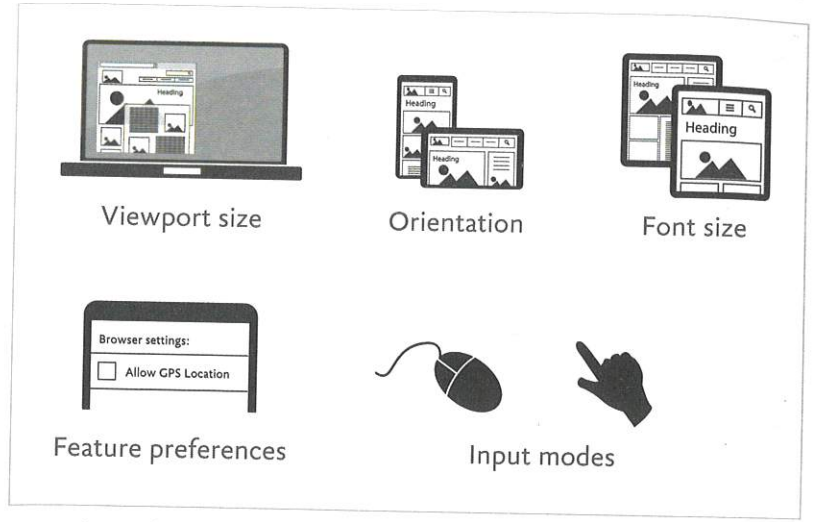

FIG 2.3: Assumptions to avoid based on a device's user agent string.

- **Viewport size.** While a device database may return somewhat reliable information about a device's screen, screen size often differs from a browser's viewport size. For responsive layouts, it's the viewport size we care about. We should also avoid assuming anything about a user's connection speed based on screen size—smartphones are commonly used over fast Wi-Fi connections, while laptops and tablets can be tethered to a slow cell network—or worse: bus Wi-Fi.
- **Device orientation.** Those viewport considerations are twice as difficult when you consider display differences between portrait and landscape (FIG 2.4). Even if we know the dimensions of a screen, we have no way of knowing (on the server side) the device's orientation. We need to ship CSS that accommodates viewport variability.
- **Font size.** The common practice of using em-based units for media queries means that users' preferred default font size determines the layout they get, so a browser on a laptop with a large font size may need a smartphone-ish layout. (As we'll discuss in a bit, CSS media queries handle this naturally.)

FIG 2.4: The *Boston Globe* website shown in two screen orientations on the same device.

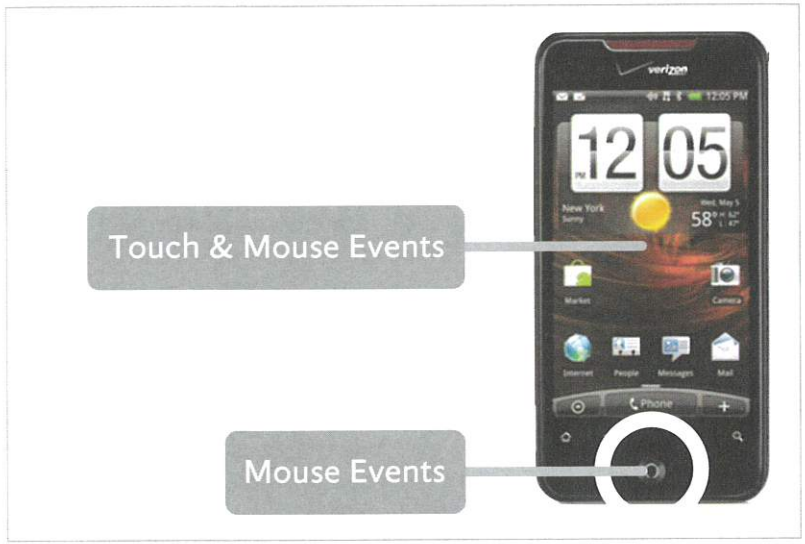

FIG 2.5: An Android 2.3 device with multiple input mechanisms.

- **Custom preferences.** People commonly override their browser defaults and turn off features on their phones. A browser may support a feature, but a server has no way of knowing whether that feature has been disabled by the user.
- **Input modes.** Device databases can often tell us if a device has a touch screen. But as you may recall, just because a device has a touch screen doesn't mean that it supports touch events, or that touch is the only input mechanism the device offers (FIG 2.5). And of course, touch support is now built into devices that have large screens as well, such as Google's Chromebook laptop, so it's unsafe to infer any sort of relationship between touch support and screen size.

So when we build cross-device experiences, we want to be mindful of these factors and be wary of assumptions based on stock device conditions. Device detection is a risky bet, and it's only going to get riskier.

GOOD NEWS: WE'RE IN CONTROL

The move away from browser-specific code has been long and slow, but support and tools for making sustainable, feature- and condition-based decisions have dramatically improved in recent years, and they get better every day. Client-side technology like HTML, CSS, and JavaScript allows us to see what's *actually* happening in that dynamic browser environment and to make decisions that are more contextual and appropriate. In a word: responsible.

Features and constraints, not devices

"An over-emphasis on context can focus design solutions too much on assumed mobile situations instead of on the true richness of mobile web use happening today."

—LUKE WROBLEWSKI, http://bkaprt.com/rrd/2-03/

One crutch we'd do well to abandon is the assumption that device form-factors are exclusively tied to specific browser

features or network conditions. In reality, these features routinely overlap across common device categories.

"Touch and viewport size aren't connected. The most popular touch devices may currently be phones and tablets, but you can also find touch screen offerings for 27" monitors and beyond."
—TRENT WALTON, "Type & Touch" (http://bkaprt.com/rrd/2-04/)

Once-convenient mobile and desktop categories have lost any meaning for our work. We sometimes hear "mobile" to describe a device's roving physical context, yet vast amounts of smartphone and tablet use happen while people are at home on their couch. We may think of mobile as a connection-speed limitation, yet devices of all kinds are as likely to be tethered to high-speed Wi-Fi as they are to a high-latency cell tower (FIG 2.6). And we may take mobile to mean devices with features like a smaller screen size and an ability to react to touch, or constraints like poor rendering capabilities, but each day devices are released that break free from the neat categorizations we try to impose.

Attempting to classify devices and browsers by form factor alone distracts us from the parameters that are actually important when we design for the web: *features* (like CSS properties and JavaScript APIs) and *constraints* (like viewport size, unpredictable connectivity, or off-line use). Designing for features and constraints allows us to see how patterns that may otherwise seem distinct are shared across devices, and to build in a modular manner to create unique experiences that feel appropriate to each device.

Querying media responsibly

Perhaps the most memorable tenet from Ethan Marcotte's original responsive design workflow is CSS3 media queries, the conditional CSS statements we use to deliver styles to some contexts and not others. Marcotte's initial article used media queries in a desktop-first manner, which means that we build the largest layout first and use media queries to override that layout all the way down to a small screen.

FIG 2.6: Wi-Fi not guaranteed: an ad for a USB dongle that enables web access over a SIM card on mobile networks. Photograph by Frankie Roberto (http://bkaprt.com/rrd/2-05/).

Shifting the responsive direction

Toward the end of his book *Responsive Web Design*, Marcotte remarked that shifting our media queries to follow a mobile-first, or small-screen-first, philosophy would give our users a more responsible, sustainable experience. To paraphrase Luke Wroblewski, a mobile-first workflow helps us to prioritize content, since there's not enough room on a small screen for non-critical content. Thinking mobile-first also pairs nicely with the mindset of progressive enhancement, aka starting small and layering in more complex layout as space permits.

"The absence of support for @media queries is in fact the first @media query."
—BRYAN RIEGER, http://bkaprt.com/rrd/2-06/

A mobile-first responsive stylesheet begins with styles that are shared across all experiences, forming the foundation of the smallest screen layout. These styles are followed by a series of mostly `min-width` media queries to scale that layout up to greater viewport sizes and pixel depths. At a high level, the CSS looks something like this:

```
/* styles for small viewports here */
.logo {
  width: 50%;
  float: left;
}
.nav {
  width: 50%;
  float: right;
}

@media (min-width: 50em) {
  /* styles for viewport widths 50em and up here */
}

@media (min-width: 65em) {
  /* styles for viewports 65em and up here */
}
```

What about max?

When building mobile first, `max-width` queries are still quite helpful. For example, if a design variation only occurs within a certain width range, that's a great candidate for `max-width`. You can combine `min` and `max` to isolate styles from CSS inheritance at bigger breakpoints, making for smaller, simpler CSS:

```
@media (min-width: 50em) {
  .header {
    position: static;
  }
}

@media (min-width: 54em) and (max-width: 65em) {
  .header {
    position: relative;
  }
}
```



```
@media (min-width: 65em) {
  /* .header is static positioned here */
}
```

What's with those ems, anyway?

You may have noticed that in addition to shifting the responsive direction, the breakpoint widths above use em units rather than pixels. Ems are flexible units that are sized relative to an element's container in a layout. By using ems, we can design responsive breakpoints proportionally to our fluid, scalable content, which also tends to be designed with scalable units like `em` and `%`.

Converting pixel breakpoints to ems is easy: divide the pixel-based value by 16, the default equivalent size of `1em` in most web browsers:

```
@media (min-width: 800px){
...
}
@media (min-width: 50em){  /* 800px / 16px */
...
}
```

If em breakpoints aren't your bag, pixels can work fine—I just prefer to use proportional units across a layout. The more important thing is to avoid basing breakpoints on device widths and instead focus on breakpoints that are appropriate to your site's content. For more information on `em` media queries, check out Lyza Gardner's article "The EMs have it: Proportional Media Queries FTW!" (http://bkaprt.com/rrd/2-07/).

Broadly qualifying CSS application

Not every mobile browser supports the CSS we rely on, like floats, positioning, or animation. If your styles for a small-screen experience are significantly complex, you might consider

broadly qualifying their application to newer, media-query-supporting browsers. Wrapping the mobile-first styles in a media query such as `only all` is one reliable way to do this. Though a bit confusing to look at, the `only all` query applies in any browser that supports CSS3 media queries. While `all` is a CSS media type that refers to any browser that supports CSS 1.0, the `only` prefix requires media query support to understand—which means that its defined styles are recognized by modern browsers. Here's how our mobile-first stylesheet looks when qualified for media-query-supporting browsers:

```
@media only all {
  /* styles for qualified small viewports here */
}

@media (min-width: 50em) {
  /* styles for viewport widths 50em and up here */
}

@media (min-width: 65em) {
  /* styles for viewports 65em and up here */
}
```

Retaining some style in basic browsers

To maintain some level of branded experience in browsers that don't support media queries, I find it useful to tease out a small amount of the safer styles from your first CSS breakpoint and place them before the `only all` media query so they apply everywhere.

Safe styles—like `font-weight`, `margin`, `padding`, `border`, `line-height`, `text-align`, and more—can be sent to any browser without introducing problems (FIG 2.7).

```
/* styles for small viewports here */
body {
  font-family: sans-serif;
  margin: 0;
}
```


FIG 2.7: The basic experience of the *Boston Globe* website on an older BlackBerry.

```
a {
  font-color: #a00;
}
section {
  margin: 1em;
  border-bottom: 1px solid #aaa;
}

@media only all {
  /* styles for qualified small viewports here */
}
/* more... */
```

A quick (responsible) reminder: if you choose to deliver styles to basic browsers, be sure to test them!

Bullet-proofing the viewport

Traditionally (if such a term can be used for this stuff) in responsive layouts, we've used a `meta` element to specify the width that browsers should use to render a page when it first loads, such as the popular `width=device-width` declaration:

```
<meta name="viewport" content="width=device-width; »
  initial-scale=1">
```

This approach has worked fine for us so far, but it's not particularly sustainable: for starters, the W3C never standardized it; what's more, `meta` elements are a strange place to define a visual style. Thankfully, the W3C has standardized an approach to specifying viewport style information such as `width` and `scale`, and it's handled via CSS instead of HTML. To ensure that our viewport settings continue to work in future browser versions, we want to include these rules in our CSS:

```
@-webkit-viewport{width:device-width}
@-moz-viewport{width:device-width}
@-ms-viewport{width:device-width}
@-o-viewport{width:device-width}
@viewport{width:device-width}
```

For browsers that don't support `@viewport`, we should continue to include the `meta viewport` element. Trent Walton wrote a handy post about this, and includes tips for getting our responsive sites to work well with IE10's "snap mode" on Windows 8 (http://bkaprt.com/rrd/2-08/). (Unsurprising spoiler: getting things up to speed in IE10 requires more than the code above.)

Querying other media

Querying the width and height of a viewport with `min-width` and `max-width` goes a long way toward producing a usable layout, but there are many more conditions we can test to layer enhancements contextually. For instance, to deliver higher-dpi images to HD screens of 1.5× resolution and up, we can use a `min-resolution` media query of `144dpi` (twice that of standard `72dpi`). To cover some existing browsers currently transitioning to the standard syntax, we can also include a WebKit-prefixed fallback property (`-webkit-min-device-pixel-ratio`) in our query:

```
@media (-webkit-min-device-pixel-ratio: 1.5),
       (min-resolution: 144dpi) {
  /* Styles for HD screens here */
}
```

In the near future, media queries will support several more interesting features, such as detecting whether touch- or hover-based input mechanisms are supported via `@media (pointer:fine) {...}` and `@media (hover) {...}`, detecting JavaScript support via `@media (script){ ... }`, and even detecting ambient light with `luminosity`. To track their implementation status, keep an eye on Can I use... (http://bkaprt.com/rrd/2-09/), and for some great articles describing the "good and bad" of Level 4 media queries, see Stu Cox's article of that name (http://bkaprt.com/rrd/2-10/).

DETECTING FEATURES WITH JAVASCRIPT

As new features arrive in browsers, we often need to qualify their use at a more granular level. JavaScript feature detection has long been a part of web development, thanks to proprietary feature differences in early browsers. Back then and (to a lesser degree) to this day, to get code to work in more than one browser it was necessary to check whether even the most common functions were defined before using them. For example, if we wanted to listen for an event like `click`, we would first need to check which event API the browser supported:

```
// if standard event listeners are supported
if( document.addEventListener ){
  document.addEventListener( "click", myCallback, »
    false );
}
// if not, try the Internet Explorer attachEvent method
else if( document.attachEvent ){
  document.attachEvent( "onclick", myCallback );
}
```

Detecting JavaScript features

Thankfully, in recent years the web standards movement has nudged browsers into supporting common APIs for features like event handling, which greatly reduces the number of browser-specific forks we must apply in our code and makes it more sustainable in the long term.

Now it's more common to use JavaScript feature detection to determine whether a feature is supported, before using that feature to create enhancements on top of an already functional HTML experience. For example, the following JavaScript function detects whether the standard HTML `canvas` element (a sort of artboard element that offers an API for drawing graphics with JavaScript) is supported:

```
function canvasSupported() {
  var elem = document.createElement('canvas');
  return !!(elem.getContext && elem.getContext('2d'));
}
```

This could be used before loading and running a pile of `canvas`-dependent code:

```
if( canvasSupported() ){
  // use canvas API safely here!
}
```


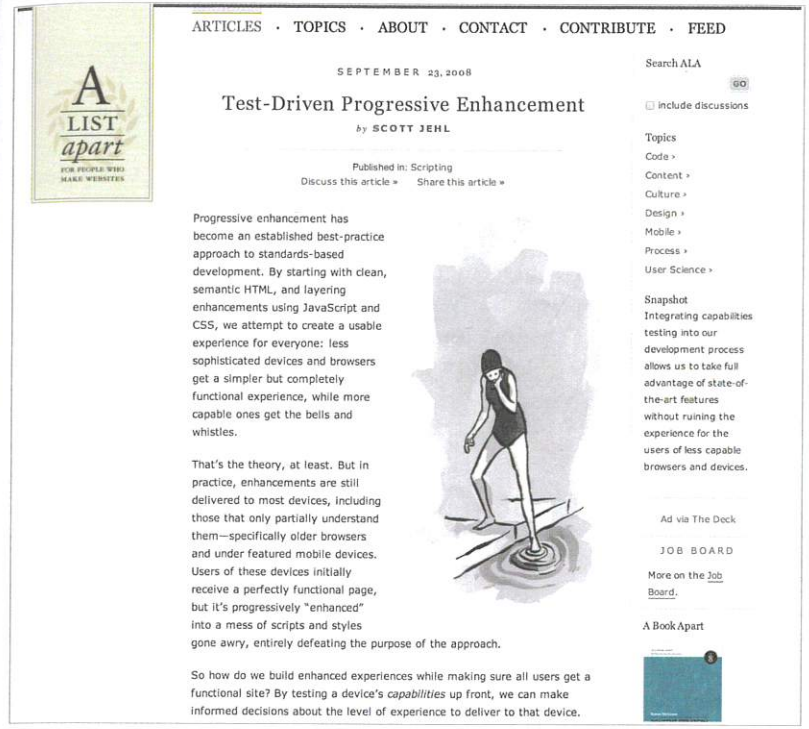

FIG 2.8: My 2008 *A List Apart* article "Test-Driven Progressive Enhancement" (http://bkaprt.com/rrd/2-11/).

Detecting CSS features

While detecting features in JavaScript isn't new, using JavaScript to detect CSS feature support began relatively recently. I first used CSS feature detection this way in the examples for my 2008 *A List Apart* article "Test-Driven Progressive Enhancement," which advocated the idea of running a series of diagnostic tests on a browser before applying CSS and JavaScript enhancements to a page (**FIG 2.8**).

At the time, new browsers included great new CSS capabilities like `float` and `position`, even though browsers with poor support for these features were widely used. This made it difficult to apply modern CSS to a site without breaking the experience for users running older browsers.

One example from the article was the following test to see if a browser *properly* supports the standard CSS box model, which incorporates `padding`, `width`, and `border` into the measured dimensions of an element. At the time, two different box model variations were actively supported across popular browsers, and writing CSS against one model would cause layouts to break in browsers (read: old versions of Internet Explorer) that supported the other.

```
function boxmodel(){
    var newDiv = document.createElement('div');
    document.body.appendChild(newDiv);
    newDiv.style.width = '20px';
    newDiv.style.padding = '10px';
    var divWidth = newDiv.offsetWidth;
    document.body.removeChild(newDiv);
    return divWidth === 40;
}
```

Let's look at this more closely. The JavaScript function creates a new `div` element, appends it to the `body` element in the document, and gives the `div` some `width` and `padding`. The function then returns a statement that the `div`'s rendered width should equal 40. Those familiar with the standard CSS box model will recall that the `width` and `padding` of an element contribute to its calculated width on the screen, so this function tells you whether the browser calculates that width as expected.

In the article, I bundled this test and others for properties like `float` or `position` into a suite called `enhance.js`, which could be run as a broad diagnostic during page load. If the test passed, the script would add a class of `enhanced` to the HTML element that could be used to qualify the application of advanced CSS properties.

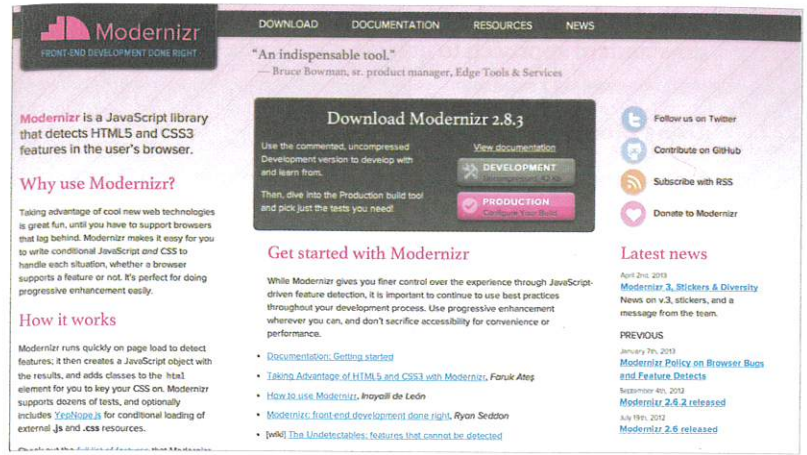

FIG 2.9: The Modernizr feature-testing framework.

```
.enhanced .main {
  float: left;
}
```

Qualifying CSS in this way felt like a sustainable step forward, but `enhance.js` was admittedly rough around the edges, since it couldn't detect and apply features at a granular level. Fortunately, developers much smarter than myself picked up the slack and took off running.

Feature detection frameworks

Almost any modern JavaScript framework uses feature tests within its internal codebase, but one framework stands alone in its mission to provide a standard approach to running tests in our sites: Modernizr (http://bkaprt.com/rrd/2-12/), created in 2009 by Paul Irish, Faruk Ateş, Alex Sexton, Ryan Seddon, and Alexander Farkas (**FIG 2.9**). Modernizr's simple workflow of adding specific classes to the `html` element to signify that a feature like CSS multi-columns is supported (`<html class="...css-columns...">`) makes the approach accessible to developers not

versed in JavaScript detection intricacies, and has become a pseudo-standard approach to qualified application of enhancements.

Using Modernizr

Using Modernizr out of the box is quite straightforward. Include the `modernizr.js` script in the head of an HTML document, and the script runs feature tests automatically.

```
<script src="js/modernizr.js"></script>
```

When Modernizr tests run, the framework retains a JavaScript property, stored on the globally available `Modernizr` object, of that test's name that equals `true` if it passes or `false` if it doesn't.

```
if( Modernizr.canvas ){
  // Canvas is supported!
}
```

When a test passes, Modernizr also adds a class of that test's name to the `html` element, which you can then use within your CSS selectors to qualify the use of certain features. Quite a lot easier than hand-coding those tests above, right?

While you can safely use many modern CSS features without qualification—like `box-shadow`, `border-radius`, or `transition`—relying too heavily on these features can introduce usability issues in browsers that don't support them. For instance, say you want to overlay text on an image. You want a text color that matches the image and a text shadow to pull the characters forward (**FIG 2.10**).

```
.img-title {
  color: #abb8c7;
  text-shadow: .1em .1em .3em rgba( 0, 0, 0, .6 );
}
```

In browsers without `text-shadow` support, the text is nearly invisible (**FIG 2.11**)!

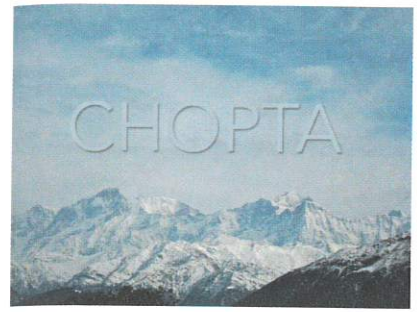

FIG 2.10: Our intended design.

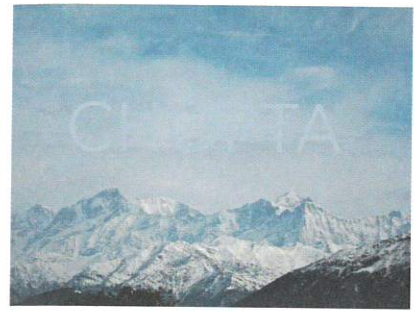

FIG 2.11: Our design as viewed in a non-`text-shadow`-supporting browser.

To keep this from happening, you may choose to default to a different presentation, perhaps using a color with higher contrast first and then feature detection to enhance to the ideal presentation.

```
.img-title {
  color: #203e5b;
}
.textshadow .img-title {
  color: #abb8c7;
  text-shadow: .1em .1em .3em rgba( 0, 0, 0, .6 );
}
```

And voilà! You have yourself an accessible experience in browsers new *and* old (**FIG 2.12–2.13**).

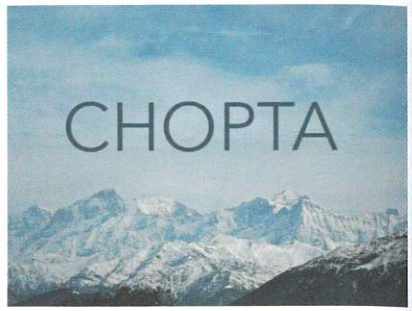

FIG 2.12: Default experience.

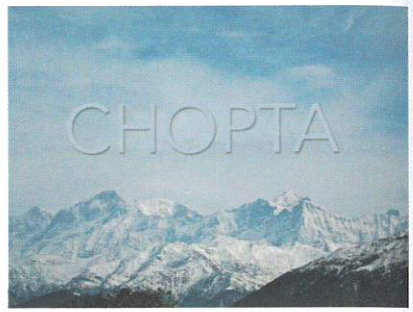

FIG 2.13: Enhanced experience.

Detecting CSS support without JavaScript

As useful as JavaScript-driven feature detection is, it comes with the downside of loading and running code for no purpose other than to qualify features we want to use. Ideally, we should standardize the ways we detect features as we do the features themselves; thanks to advocacy from developer Paul Irish, native support for a CSS feature-detection approach has been standardized by the W3C and is gradually becoming available in browsers.

The `@supports` feature (http://bkaprt.com/rrd/2-13/) follows a similar syntax to that of media queries. By passing any CSS property and value pair (say, `display: flex`) to the `@supports` rule, you can define entire style blocks to apply only in browsers that implement that CSS feature (or features). Here's an example:


```
@supports ( display: flex ) {
  #content {
    display: flex;
  }
  ...more flexbox styles here
}
```

`@supports` is pretty handy: it offloads feature detection work to the browser, removing the need for us to write custom—and often slow, unreliable—tests to produce similar results. Less work for developers, and better performance for users! In addition to the `@supports` syntax in CSS, you can pair a JavaScript API called `CSS.supports`. Here's an example of it in action, qualifying the use of `transition`:

```
if( CSS.supports( "(transition: none)" ) ){
  // CSS transitions are supported!
  // Perhaps you'd add some transition event listeners
     here...
}
```

Support for support

As is the nature of many CSS features, the `@supports` approach to feature queries will gracefully degrade by itself, meaning you can safely include it in a stylesheet. Browsers that don't understand `@supports` will ignore it and the styles it qualifies.

We can't say the same of the JavaScript method that pairs with `@supports`: funnily enough, before using the `CSS.supports` JavaScript API, you need to check if the browser supports `CSS.supports`! If you've been developing websites for a while, you're probably used to this sort of thing. Somewhat amazingly, though, two versions of `CSS.supports` already exist in the wild because some versions of the Opera browser have a non-standard implementation (`window.supportsCSS`). So here's a snippet that tries to assign a variable cssSupports to one or the other, if available:

```
var cssSupports = window.CSS && window.CSS.supports || »
  window.supportsCSS;
```

With this normalization in place, you can qualify your `CSS.supports` use as follows:

```
if( cssSupports && cssSupports( "(transition: none)" »
  ) ){
  // CSS transitions are supported!
}
```

Now to play devil's advocate for a moment: one potential issue with native feature detection like `@supports` is that it places trust in browsers to report honest results about their own implementation's standards compliance. For example, the Android 2 browser supports `history.pushState`—used for changing the browser's URL location to reflect updates made in the page since last load—but it doesn't update the actual page address until you refresh the page, making the implementation completely useless. From a web developer's perspective, any variation from a W3C spec in a browser's implementation could deem a feature unusable, so where do we draw the line for whether a feature is supported or not? The spec suggests that support is defined by a browser implementing a particular property and value "with a usable level of support," which, of course, is subjective (http://bkaprt.com/rrd/2-14/). Given that in the past, browser vendors have routinely adjusted their user agent strings to improve their relevance among competitors, there's also the potential for deliberately dishonest reporting. As for how accurately this detection feature will continue to work, the future remains to be seen.

That leads us well into our next section.

UA detection: the best when all else fails

Sometimes, the question of whether a feature is supported is more complicated than a simple yes or no.

Uneven browser support is particularly problematic when it comes to talking about "the undetectables": features that are hard to detect across browsers through feature detection alone (http://bkaprt.com/rrd/2-15/). Scarily, a significant subset of these undetectables can wreak havoc on the usability or accessibility of content when they're unsupported or, often worse, partially supported. For example, Windows Phone 7 (running Internet Explorer 9) supports `@font-face` for delivering custom fonts, but only with fonts that are installed on the device—defeating the purpose of the feature.

Many features are partially or improperly supported in browsers. That presents a tedious challenge to responsible design: we have no way of knowing whether those features are working properly without testing the browser in question ourselves.

In situations where support for a technology you need is uneven and undetectable, and the lack of (or partial) support can create an undesirable effect, it may be a wise choice to employ some browser-based (rather than feature-based) detection as a fallback. It's worth noting, *yelling* even, that user agent detection has serious drawbacks and tends to be very unsustainable. Avoid it if you can. That said, it's sometimes necessary. The responsible approach is to do what we can to exhaust all potential means of browser-agnostic detection before resorting to the user agent string. Here are a couple of examples incorporating that last resort.

Desperately qualifying overflow

The CSS `overflow` property allows us to control what happens when content overflows the boundaries of an element. Possible values include `visible` (which visually displays the overflowed content), `hidden` (which hides it), and `scroll` or `auto` (which allows the user to scroll through the element's content). For example, the following CSS when applied to an element with a class of .my-scrolling-region:

```
.my-scrolling-region {
  border: 1px solid #000;
  height: 200px;
  width: 300px;
  overflow: auto;
}
```

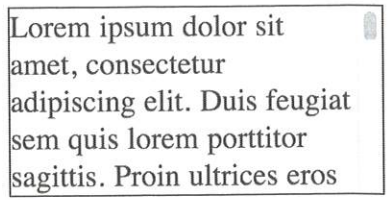

FIG 2.14: An example of the CSS overflow property.

...produces **FIG 2.14** in the browser, if the content happens to exceed the height of the element.

Unfortunately, simple as it may sound, *partial* support for `overflow` is prevalent on the web. For example, many mobile browsers treat `overflow: auto` the same as `overflow: hidden`, which crops content without offering users any means of accessing it. What's more, older versions of iOS require two fingers to scroll an `overflow` region (which presumably few iOS users even know to try).

These support shortcomings make `overflow` risky to use without qualification, but to make matters worse, `overflow` support is nearly impossible to detect! A test for whether the `overflow` property is supported will pass even if it's not supported *properly*, and trying to test for `overflow: auto` support specifically requires user interaction to verify (i.e., we don't know for sure if scrolling works until the user tries it). Because of this predicament, `overflow` is a good candidate for a little user agent detection (as a fallback). Overthrow (http://bkaprt.com/rrd/2-16/) is a script that helps us use `overflow` safely; when the script runs, it takes the following steps:

It first runs a feature test to try to detect whether `overflow` is supported. This test will fail reliably in browsers that don't support `overflow`, and pass in most modern browsers that correctly support it. Unfortunately, though, the test also fails in several browsers that are known to support `overflow` properly, requiring a fallback approach to get those browsers on board. That approach checks the browser's user agent string to detect eight or so browsers that are known to render `overflow` properly yet fail the feature test. The script assumes those specific browsers will continue to support the feature in future versions as well (a slightly risky assumption). In passing browsers, Overthrow adds a class of `overthrow-enabled` to the HTML element, which can be used to qualify `overflow` within a stylesheet.

I want to reemphasize that we've attempted to use a browser-agnostic means of detecting the feature *before* resorting to device-specific logic. That part is critical, as we want to make our code as future-ready and sustainable as we can. With that class in place, we can qualify the element from above to safely use `overflow`:

```
.overthrow-enabled .my-scrolling-region {
  overflow: auto;
  -webkit-overflow-scrolling: touch;
  -ms-overflow-style: auto;
  height: 200px;
}
```

The CSS shown here ensures that browsers that support `overflow` get a scrolling pane with a specific `height`, while others see the content in full without a set `height` that would require scrolling. Best of all, if the test malfunctions or fails to pass an `overflow`-supporting browser, the content will still be accessible. In addition to the `overflow` and `height`, I've added vendor-specific properties to apply momentum-based scrolling in WebKit and IE10 touch-based environments. **FIG 2.15** and **FIG 2.16** demonstrate supported versus unsupported environments—both perfectly usable.

Position: fixed? More like position: broken!

Another example of a dangerous undetectable is the CSS property `position:fixed`. Many recently popular mobile browsers (Android 2, Opera Mobile, older iOS versions) leave fixed-positioned content wherever it is at page load, meaning that content continues to sit on top of the content beneath it, obscuring access to the page (**FIG 2.17**).

FIG 2.15: The Overthrow site in a browser that supports overflow.

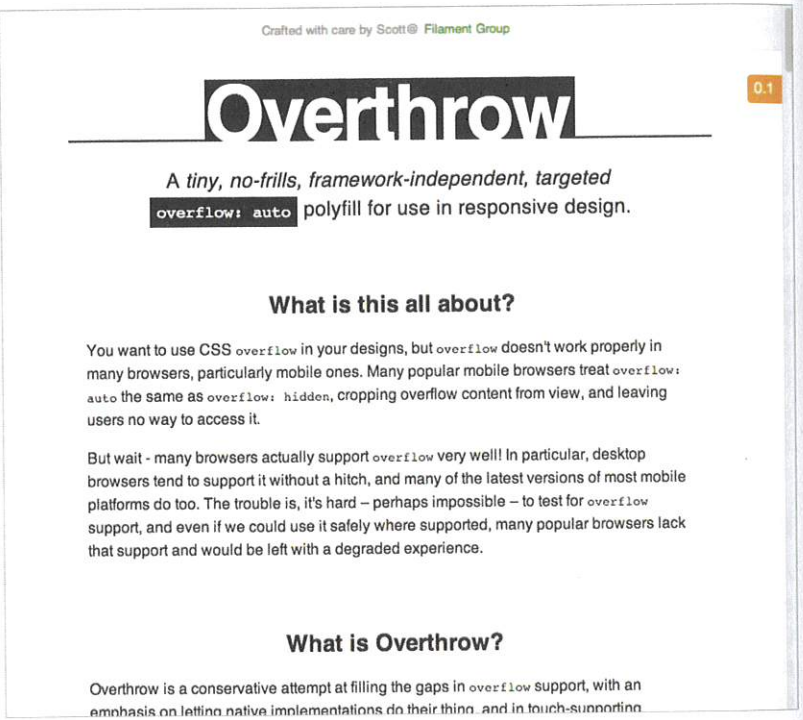

FIG 2.16: The Overthrow site in a browser that doesn't support overflow.

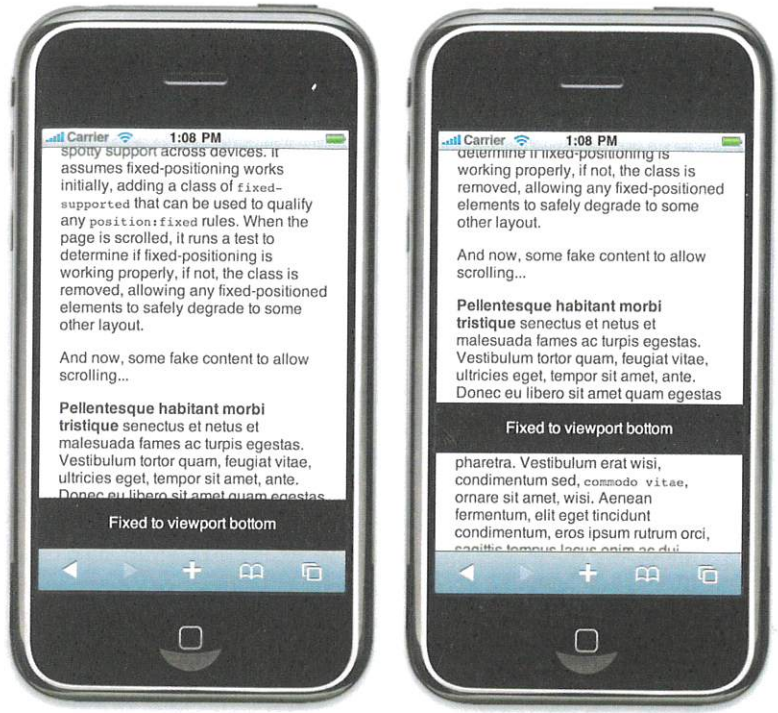

FIG 2.17: Intended behavior (left) vs. buggy behavior (right) in a browser with poor fixed-position support.

To combat this, check out Fixed-Fixed (http://bkaprt.com/rrd/2-17/). Similarly to Overthrow, Fixed-Fixed employs a simple CSS class qualifier you can use in your selectors; it also, like Overthrow, attempts to run a feature test before resorting to user-agent-based fallback detection if necessary. Here's an example:

```
.fixed-supported #header {
      position: fixed;
}
```

That's about it! In qualified browsers, the `#header` element is fixed to the top of the viewport; in others, it scrolls with the page.

Supporting the unsupported

If a browser doesn't support a particular feature, does that mean we have no way to use it in that browser? Not necessarily. In the past several years, the practice of emulating features in unsupported browsers, known as *shimming* or *polyfilling*, has become quite common. In fact, there's a workaround listed on the Modernizr site for almost every feature the library detects.

Shims tend to be quick hacks to enable a certain approach, while polyfills are more involved. Let's look at shims first.

Shims

Probably the most famous shim is the HTML5 shim, also called the HTML5 *shiv*, perhaps due to web developers' common disdain for older versions of Internet Explorer (more here: http://bkaprt.com/rrd/2-18/). IE versions older than 9 can't apply CSS styles to HTML elements that didn't exist at the time of the browser's release date, meaning HTML5 elements like `section` and `header` are unstyleable in one of the most widely used browsers on the web. Fortunately, a JavaScript workaround discovered by developer Sjoerd Visscher tricks IE into "learning" about any element that's generated with the method `document.createElement`, enabling IE to style those elements like any other. The workaround couldn't be easier: create an element of a given name using `document.createElement`, and all instances of that element IE subsequently encounters will be recognized as if natively supported, like magic.

Remy Sharp later created an open-source script (http://bkaprt.com/rrd/2-19/), now maintained by Alexander Farkas and others, that applies this workaround to the new HTML5 elements.

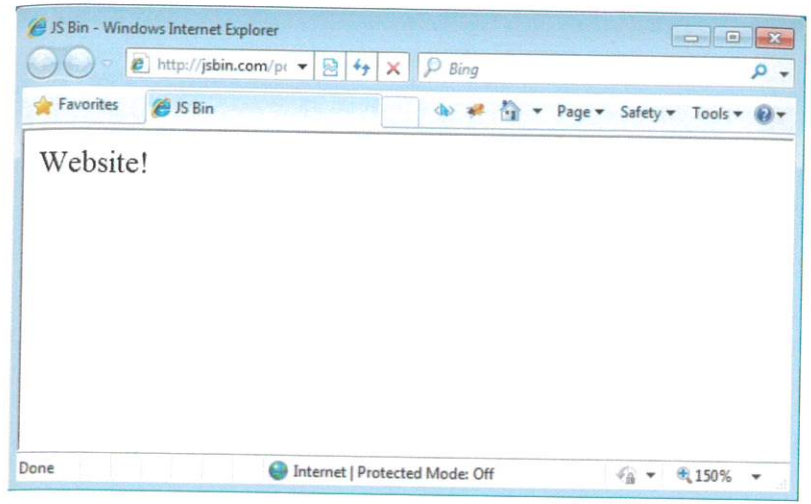

FIG 2.18: Unstyled, unrecognized HTML5 header element.

FIGURE 2.18 shows an example of HTML5 styling in IE8 without the shim.

```
<!DOCTYPE HTML>
<html>
<head>
  <style>
    header {
      font-size: 22px;
      color: green;
    }
  </style>
</head>
<body>
  <header>Website!</header>
</body>
</html>
```

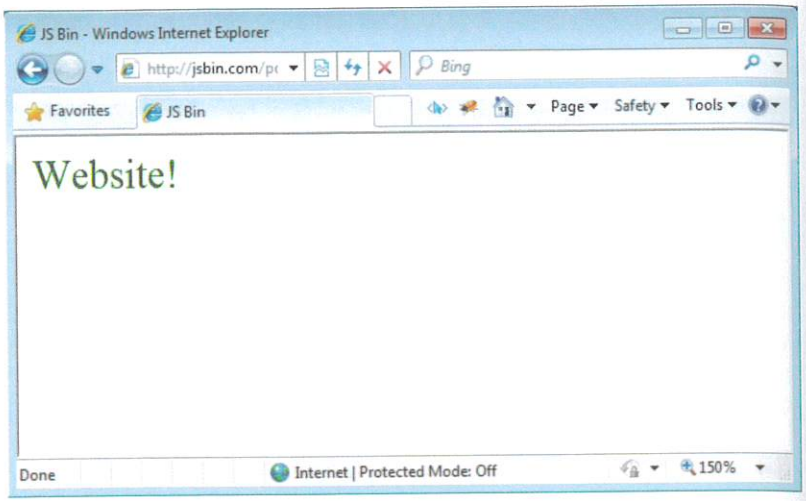

FIG 2.19: Styled, shimmed HTML5 header element.

FIGURE 2.19 shows how it renders with the shim.

```
<!DOCTYPE HTML>
<html>
<head>
  <!--[if lt IE 9]>
  <script src="html5shiv.js"></script>
  <![endif]-->
  <style>
    header {
      font-size: 22px;
      color: green;
    }
  </style>
</head>
<body>
  <header>Website!</header>
</body>
</html>
```


With regard to responsible development, there is a minor but considerable downside to shimming HTML5 support: if the JavaScript fails to load in older IE browsers, HTML5 elements will not receive any CSS styles. This may not be a major problem if the only style we're applying is some color, as in the example above, but if a columnar page layout depends upon HTML5 element styling, the page elements will crash together in IE, which may hinder usability. To avoid this issue, it has become common to wrap HTML5 elements in a `div` with a class of that element name (`<div class="article"><article></article></div>`), and style that `div` element instead. This bloats the markup a little, but it does allow modern browsers to reap the semantic benefits of HTML5 elements without needing a JavaScript workaround to style the page.

Responsive design polyfills

The term *polyfill* was coined by Remy Sharp to describe an approach that Paul Irish sums up nicely as "a shim that mimics a future API providing fallback functionality to older browsers" (http://bkaprt.com/rrd/2-20/). A polyfill goes to some length to reproduce a standardized API with JavaScript, and is typically more than a quick-and-dirty workaround.

A responsible shim or polyfill should always try to discern if a feature is supported natively before reproducing its API. For performance reasons, a native implementation is always preferred, so it's also wise to consider whether the feature is truly necessary to polyfill in the first place. Nine times out of ten, it's more responsible to serve unsupported browsers a less-enhanced experience than to force ad hoc upgrades for features they don't support. The decision to use a polyfill should be based on three main points: how much the feature improves your audience's user experience, the cost to performance of including the polyfill in a page, and its ability to one day be removed seamlessly from your codebase.

For responsive design, I commonly find a few polyfills helpful.

MatchMedia: media queries in JavaScript

While media queries are mostly used for applying CSS, sometimes it's useful to know whether a media query applies to JavaScript logic as well. One example may be when requesting additional, appropriately sized images for a gallery. `MatchMedia` enables us to evaluate media queries in JavaScript.

To use it, simply pass any media type or query to the `window.matchMedia` function, and it will return an object with a `matches` property that is either `true` or `false` depending on whether the media applies at that time:

```
if( window.matchMedia( "(min-width: 45em)" ).matches ){
  // The viewport is at least 45em wide!
}
```

Okay, I didn't mention a slight wrinkle: `matchMedia` is not supported in every browser that supports CSS3 media queries. So, before using it we either need to check to see if it's supported at all or use a polyfill to make it work where it otherwise wouldn't. For those interested in the latter option, I wrote a polyfill for `matchMedia` a few years back, and Paul Irish was kind enough to set up a GitHub repository where we've continued to maintain the script (**FIG 2.20**).

To use the polyfill, simply reference the `matchMedia.js` file in your page to use `window.matchMedia` in any browser, even one that doesn't support CSS media queries! Not so fast, though: you still need to be in a media-query-supporting browser for any media query value to match (though media types like `screen` work in just about any device with a screen).

With the polyfill in place, you can now use `matchMedia` to test whether CSS3 media queries are natively supported, which could be useful if you want to qualify the addition of advanced scripting that should only apply in modern browsers. Just like in CSS itself, the `only all` media query can give us just that information.

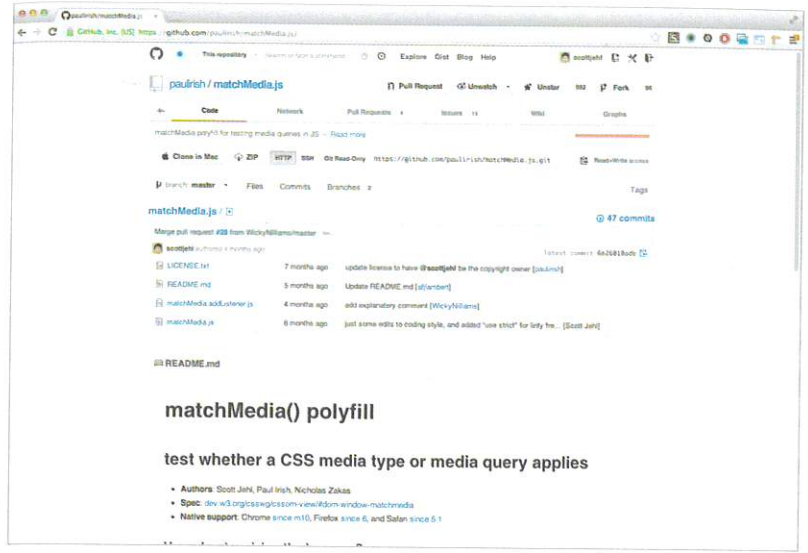

FIG 2.20: The `matchMedia.js` Project by Scott Jehl, Paul Irish, and Nicholas Zakas (http://bkaprt.com/rrd/2-21/).

```
if( window.matchMedia( "only all" ).matches ){
  // Media queries are natively supported!
}
```

Another potentially useful feature of the `matchMedia` API is its ability to accept *listeners*, allowing us to keep an ear out for changes to a particular `matchMedia` query's state after we check it the first time. To ensure it'll work broadly, the `matchMedia.js` polyfill has a listener extension to support this part of the API as well. Adding a `matchMedia` listener is pretty straightforward: call a `matchMedia` function as seen above and assign an `addListener` method to the end of it, like this:

```
window.matchMedia( "(min-width: 45em)" ).addListener( »
  callback );
```

In this case, `callback` is a function you can define that executes every time the media query changes its state between `true` and `false`. The first argument passed to the `callback` function contains a reference to the `matchMedia` object, allowing easy access to its `matches` property whenever the listener fires. Here's an example of how that function can plug in:

```
window.matchMedia( "(min-width: 45em)" )
  .addListener( function( mm ){
    if( mm.matches ){
      // The viewport is at least 45em in width!
    }
    else {
      // The viewport is less than 45em in width!
    }
} );
```

Media queries to IE: please respond, IE.

As you'll likely remember from earlier in this chapter, Internet Explorer versions 8 and older don't support CSS media queries. This means that a mobile-first responsive layout will render in a layout intended for small screens on a desktop computer—still usable, but not formatted in an ideal way for large-screen use (**FIG 2.21**).

This drawback might put a damper on the whole responsive design thing if it weren't for some reliable workarounds.

First, we have a small polyfill script, `respond.js` (http://bkaprt.com/rrd/2-22/), that I developed during the *Boston Globe* project to make old IE versions render responsive layouts as if they understood CSS3 media queries. `respond.js` works by reading every stylesheet referenced in a document to find all the media queries contained therein. The script parses the values of these media queries to look for either a minimum or maximum width that can be compared against the viewport window's dimensions. When it finds a query that matches, it injects the styles contained in that query into a style block in the page, allowing the styles to apply in browsers that do not understand media queries, and the script reruns this logic whenever the

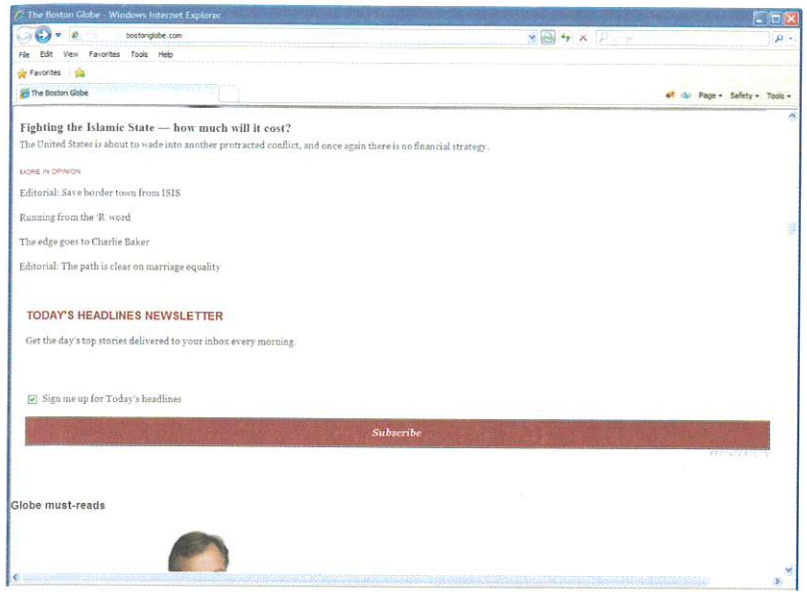

FIG 2.21: An example of the *Boston Globe* homepage in IE8.

browser is resized (and when a device's orientation changes). `respond.js` is intentionally limited in scope to keep it small and fast, so it only supports `min-width` and `max-width` media queries, which should be enough to pull off a reasonably responsive layout for users of old IE.

To use `respond.js`, reference the script in your page anywhere after your CSS references. I recommend using an IE conditional comment (a special comment syntax that old IE browsers are designed to ignore) around the script tag as well, so that the file is only requested in the versions of Internet Explorer that need it. This particular conditional comment says: "If the browser is IE less than version 9, parse the content of this comment like all other HTML on the page."

```
<!--[if lt IE 9]><script src="respond.js"> »
  </script><![endif]-->
```

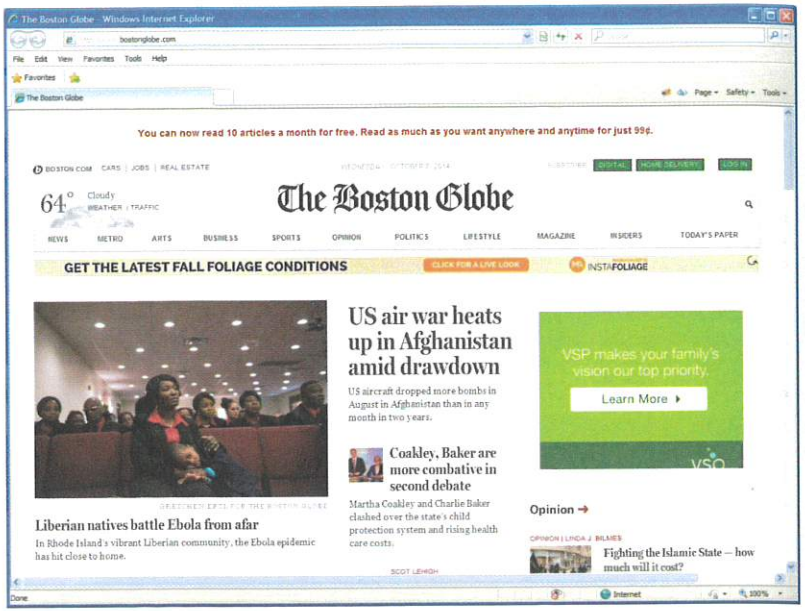

FIG 2.22: *The Boston Globe* website, viewed in IE8 with `respond.js` used for media query support.

By including this script, the *Boston Globe* homepage is more usable in old IE (**FIG 2.22**).

Avoiding the polyfill with static CSS

Another responsible approach to addressing old IE's lack of media query support is to serve IE additional CSS rules that essentially force it into rendering the styles from a responsive design's wider breakpoints. You can do this manually or with the help of CSS preprocessors such as Sass. For more on this approach, check out Jeremy Keith's 2013 article "Dealing with IE" (http://bkaprt.com/rrd/2-23/).

This approach is only able to serve users running old IE a fluid, but not responsive, layout, which may be fine depending on how broadly your fluid layout scales. However, depending on your user's screen size and your particular layout, it may or may not make for an ideal experience.

Avoiding doing anything at all

As a third option, you might simply do nothing at all and serve the responsive site to old IE as is. This leaves the layout in its default non-media-query state. Depending on the layout, this can be perfectly fine, especially if you set a reasonable `max-width` on the layout to keep the line lengths in check.

TESTING RESPONSIBLY

To ensure that a site works across a variety of screen sizes, input types, and browsers, you can't beat testing on real devices. To get a decent idea of the devices that it would make sense to amass for a personal testing lab, see Brad Frost's excellent post "Test on Real Mobile Devices without Breaking the Bank" (http://bkaprt.com/rrd/2-24/).

Devices are expensive to collect, so to test on an array of relevant devices, the average developer may need to search for a nearby community device lab, which is thankfully becoming more common (**FIG 2.23**). For information about device labs in your area, visit Open Device Lab (http://bkaprt.com/rrd/2-25/).

Testing on real devices is ideal, but we can't possibly expect to have access to even a fraction of the devices we need to care about. When you don't have access to a device, a device emulator is a brilliant solution. Emulated devices do come with drawbacks, such as misleading performance (because the browser is running on different hardware than it would normally run on), slow screen refresh rates that make animation difficult to test, connection speeds that are often faster than the device would typically have, and a lack of physical feedback that allows us to get a true sense for how a site feels on a particular device. But despite the downsides, emulators are a very reliable means of diagnosing issues with CSS layout and JavaScript.

These days I do most of my own emulated browser testing on BrowserStack (http://bkaprt.com/rrd/2-27/), which offers real-time browser testing on platforms like iOS, Android, and

FIG 2.23: Friends gathered around a collection of test devices and laptops. Photograph by Luke Wroblewski (http://bkaprt.com/rrd/2-26/).

Opera Mobile, as well as various Windows and Mac desktop browsers (FIG 2.24). BrowserStack even offers a way to easily test local sites on your machine, so you don't need to upload anything to test a page.

Also, I spend the vast majority of my development time in a browser with strong developer tools, like Google Chrome or Firefox, as their code inspectors give incredibly helpful insights into how a site's various components are working in unison, and even allow me to test features that aren't enabled in the browser by default, like touch events. I only branch out to other physical and emulated devices once a feature works to verify usability and performance, a process I repeat over and over throughout the development cycle.

As the number of web-accessing devices has grown, browser testing has become a nuanced activity, requiring developers to make subjective decisions about minor variations in the experience that individual devices receive. When pulling up a site on a

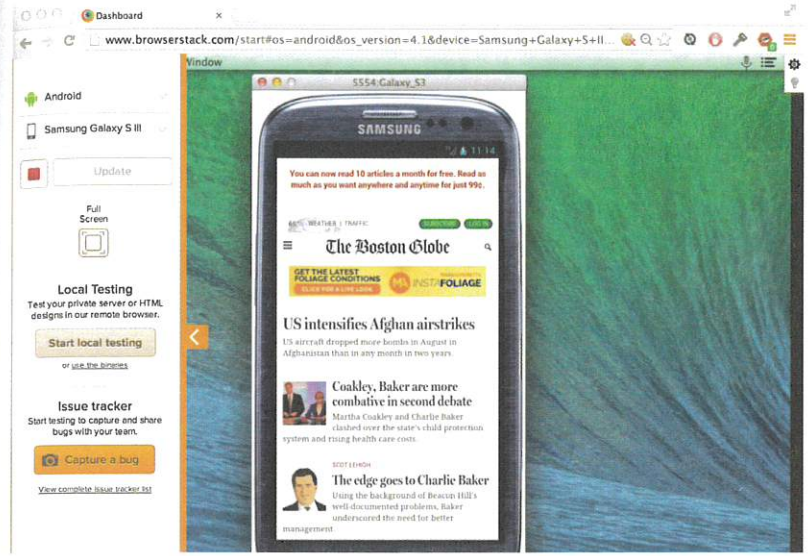

FIG 2.24: The Browserstack testing service.

particular device, I like to ask myself a series of questions about the site's design and functionality:

- Does the site load and present itself in a reasonable amount of time?
- Is the core content and functionality usable and accessible?
- Does the level of enhancement in the layout feel appropriate to the device?
- Is the text easy to scan? Do the line lengths promote readability?
- Is the site controllable and browsable via common input mechanisms on the device (touch, mouse, keyboard, etc.)?
- Are the actionable areas of the page easy to tap without tapping on adjacent items?
- Does the layout hold up to changes in orientation, viewport resizing, and font size?

- If the device has assistive technology installed (such as VoiceOver), does the content read back in meaningful ways?
- Does the page scroll efficiently? Do animations run smoothly?

The more devices we can test, the better our chances of reaching our users wherever they are.

NEXT UP

In this chapter, we covered many of the complexities of writing sustainable, cross-browser code. With that, we can proceed to our fourth tenet of responsible responsive design: performance. Because performance is a heavy topic—perhaps the one most in need of our attention when building responsive websites today—I've dedicated two chapters to its discussion.

Let's move ahead—with speed.