A BOOK APART

Brief books for people who make websites

No 15

SAMPLE CHAPTER

Ethan Marcotte

RESPONSIVE DESIGN: PATTERNS & PRINCIPLES

FOREWORD BY Mandy Brown

2 NAVIGATION

> *I may not have gone where I intended to go, but I think I have ended up where I needed to be."*
> —DOUGLAS ADAMS, The Long Dark Tea-Time of the Soul

I've been reading about a man named Pius "Mau" Piailug who, in 1976, navigated a large voyaging canoe across the Pacific, traveling more than 3,000 miles from Hawai'i to Tahiti. Piailug sailed without the aid of maps, computer-assisted navigation, or any other equipment—instead, he used the stars, sun, and moon to guide him. In fact, the instrument most helpful to him never made it onto his vessel: a star compass, a ring of shells, coral, or pebbles placed around a center point (**FIG 2.1**). The simple-looking instrument helped young navigators of Piailug's tradition understand the relationship between the horizon—the outer ring of the compass—and the canoe in the center. This, coupled with years of training at sea, was what helped Piailug complete his journey, and prove that traditional navigation was still relevant in a modern world.

I think of Piailug's journey often, and of his star compass in particular. Because if we've done our job right, a website's

FIG 2.1: Mau Piailug using a star compass to teach navigation, as he was taught in his youth. Photograph by Monte Costa (http://bkaprt.com/rdpp/02-01/).

navigation should act as a kind of compass: it helps new users orient themselves within a site hierarchy, and guides them to their destination. But with all the different tiers and types of menus our sites contain, designing intuitive, usable navigation can feel like a formidable task.

And those challenges compound themselves when you're designing responsively. How can a complex menu adapt to a smaller screen? What if we want to display more (or less) information depending on the dimensions of the display? Most critically, though, a responsive navigation system doesn't need to look or work the same at every breakpoint, but it does need to offer access to the same content across devices.

These questions might feel daunting, but they demonstrate why navigation is a great example of the *small layout systems* we're going to focus on in this book. The responsive design of a site's navigation poses an almost entirely different challenge than a page's top-level grid. In dealing with challenges of layout, interaction, and visual density, we're forced to ask ourselves: how can we design navigation that's as usable as it is responsive?

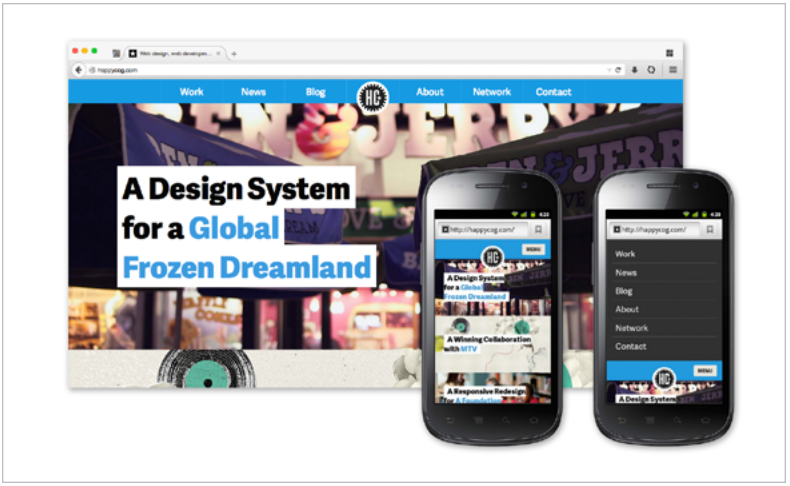

FIG 2.2: The responsive navigation for Happy Cog's site seen at sizes wide and small (http://bkaprt.com/rdpp/02-02/).

Thankfully, there are many great attempts at answering that question. In this chapter, we'll look at design patterns both common and not-so-common, and see if we can't find our way through the challenges of responsively designing navigation.

THE SHOW/HIDE TOGGLE

Open up design agency Happy Cog's responsive site (**FIG 2.2**). On wider screens, the entire navigation is visible, but on smaller viewports, where screen real estate is at a premium, the top of the design only shows a Menu link. If you tap, click on, or select that link with your keyboard, the full menu appears.

This is one of the most common ways of handling complex navigation systems in a responsive design: when the menu doesn't fit, conceal it. This pattern requires two elements at minimum: the navigation, which is concealed at certain breakpoints; and a "trigger" element, which the user interacts with to

reveal the navigation. In fact, we took the same approach with the menu on responsivewebdesign.com (**FIG 2.3**). The design's fairly modest, but I'll briefly walk you through the code to demonstrate how this pattern's often implemented..

First, at the top of the page, we have this markup:

```
<div class="head">
  <h1 class="logo">
  <a href="/"><img src="/images/logo-rwd.png"
    alt="Responsive Web Design" /></a>
  </h1>

  <div id="nav" class="nav">
    <nav>
    <h1><a class="skip" href="#menu">Explore this
      site:</a></h1>

      <ul id="menu">
        <li><a href="/workshop/">Workshop</a></li>
        <li><a href="/events/">Public Events</a>
        </li>
        <li><a href="/podcast/">Podcast</a></li>
        <li><a href="/newsletter/">Newsletter</a>
        </li>
        <li><a href="/about/">About</a></li>
      </ul>
    </nav>
  </div><!-- /end .nav -->
</div>

<!-- [ The page's main content goes here. ] -->
```

I've simplified things a little, but there's not much more to it: the document leads off with our logo, a link to skip to the navigation, and then the navigation itself, marked up as an unordered list. But that HTML is, as you might have guessed, just the foundation. To enhance the menu further, let's begin with a simple JavaScript test:

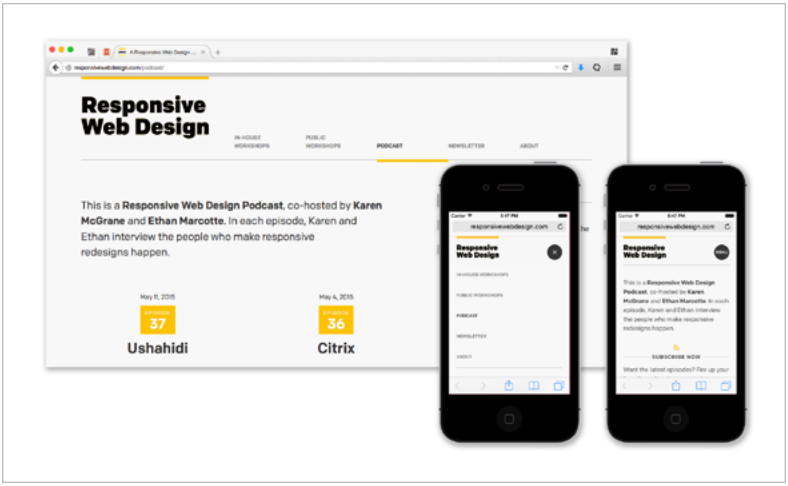

FIG 2.3: The responsive masthead for, uh, responsivewebdesign.com. Straightforward design, with a simple toggle to show (or hide) the navigation.

```
// Is this browser sufficiently modern to continue?
if ( !( "querySelector" in document
  && "addEventListener" in window
  && "getComputedStyle" in window) ) {
    return;
}

window.document.documentElement.className +=
  " enhanced";
```

We're asking the user's browser if it supports the DOM features we'll need elsewhere in our JavaScript—features like `document.querySelector`, `window.addEventListener`, and `window.getComputedStyle`. If they're not found, then that `return;` keeps the browser from executing the rest of our JavaScript. The result is that older browsers are left with a perfectly usable experience, albeit a less JavaScript-enabled one (**FIG 2.4**). And when those features are found, our JavaScript applies a class of `enhanced`

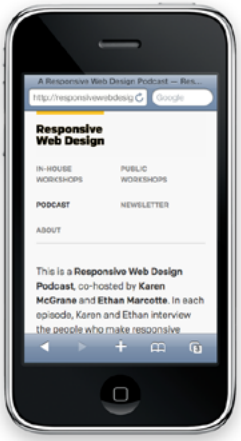

FIG 2.4: No JavaScript? No problem: our navigation's still accessible, even to modern browsers that can't load our code.

to the HTML element (`window.document.documentElement.className += " enhanced";`).

Why run that test? Well, this JavaScript test lets us build our navigation with a pinch of progressive enhancement: we can design a simpler but *usable* experience that's universally accessible by default, and then enhance the experience *only* for browsers and devices that will actually benefit from it. If the test successfully runs, then the `enhanced` class on the HTML element tells us a given browser is receiving the "enhanced" experience.

This is a fairly common approach for responsive sites, especially at a certain scale. For example, the BBC News team built their responsive design upon a foundation of progressive enhancement (**FIG 2.5**), using a little JavaScript test similar to the one we've used above, which allows them to determine whether a browser "cuts the mustard" (http://bkaprt.com/rdpp/02-03/):

> *We make [the browser landscape] manageable in the same [way] you and everyone else in the industry does it: by having a lowest common denominator and developing towards that. So we've taken the decision to split the entire browser market into two, which we are currently calling "feature browsers" and "smart browsers". ...as the [site] loads we earmark incapable*

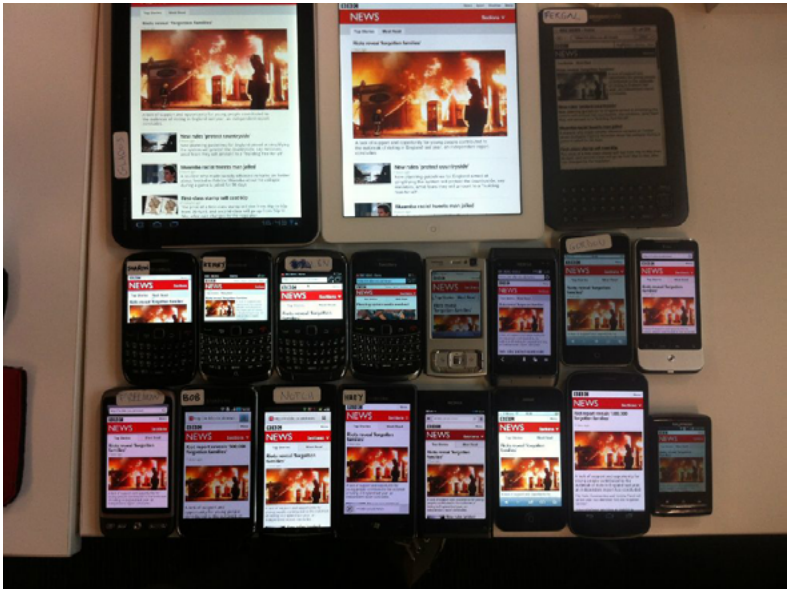

FIG 2.5: The BBC News site is accessible—and responsive—on every internet-connected device, but the experience is slightly enhanced on more modern browsers. Photograph courtesy Responsive News (http://bkaprt.com/rdpp/02-04/).

> *browsers with the above code and exclude the bulk of the JavaScript-powered UI from them, leaving them with a clean, concise* core *experience.*

Instead of tracking myriad combinations of browsers and devices, the BBC can think of their design as existing in one of two broad *experience* tiers: a baseline responsive experience, and a slightly more advanced experience that's only served to the browsers that can handle it.

On a much smaller scale, that's exactly what we're doing with the navigation on responsivewebdesign.com. With that `enhanced` class in place, we can write more advanced styles

directed at the browsers that pass our test, and build the more advanced view of our navigation:

```
.enhanced .nav .skip {
  position: absolute;
  right: 0;
  top: 1.4em;
  background: #363636;
  border-radius: 50%;
  width: 2.5em;
  height: 2.5em;
}
.enhanced .nav ul {
  max-height: 0;
  overflow: hidden;
}
```

If a browser passes our JavaScript test, this rule will use plain ol' absolute positioning to take that skip link before our navigation—`.nav .skip`—and stick it at the top of the page. At the same time, by applying a pinch of `background: #363636` and `border-radius: 50%`, we can turn that link into a big, gray, circular button. But the second rule is where things get interesting: it selects the `ul` inside `.nav`—that is, the unordered list that contains our navigation links—and uses `overflow: hidden` and `max-height: 0` to turn the list into a `0px`-tall box, effectively hiding our links from view. Hiding them, that is, until a class of `.open` is applied to the list:

```
.enhanced .nav ul.open {
  max-height: 20em;
}
```

With those rules, we now have two states for our navigation: completely hidden and expanded (**FIG 2.6**). Sounds great and all, but you might be wondering how we'll get that class on our `ul`. Well, that's where a little more JavaScript comes into play:

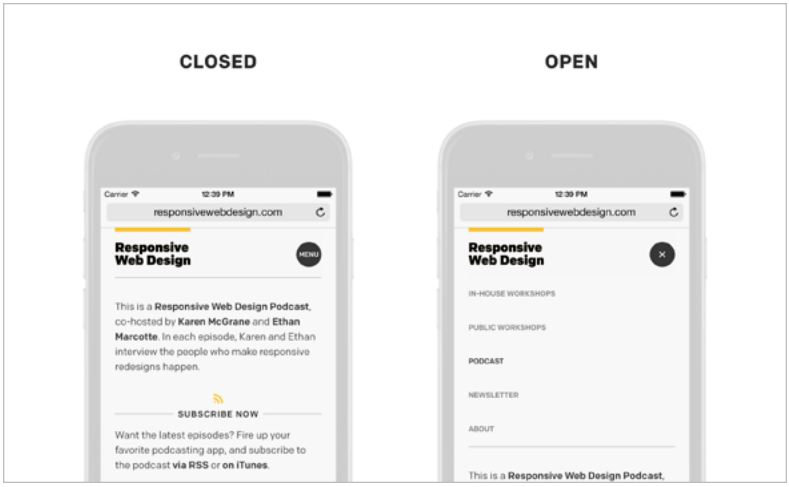

FIG 2.6: Our CSS now allows us to show or hide the navigation. But how do we make it interactive?

```
var nav = document.querySelector( ".nav ul" ),
navToggle = document.querySelector( ".nav .skip" );

if ( navToggle ) {
  navToggle.addEventListener( "click",
    function( e ) {
    if ( nav.className == "open" ) {
      nav.className = "";
    } else {
    nav.className = "open";
    }

    e.preventDefault();
  }, false );
}
```

If JavaScript's not your thing, don't worry—the code's more straightforward than it looks, I promise. Remember that skip

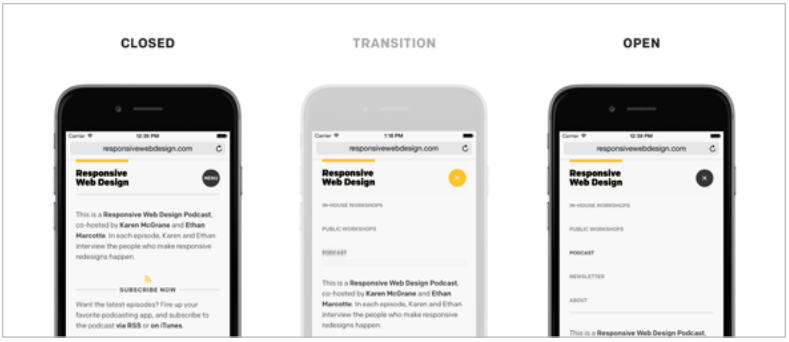

FIG 2.7: Why not include a little max-height transition, you say? Great idea!

link inside our `.nav` element? Well, we're using some JavaScript to look for it (`document.querySelector( ".nav .skip" )`) and, if it's found, add some functionality whenever it's clicked or tapped (`navToggle.addEventListener( "click", … );`). When a user taps or clicks on that link, our code checks to see if our unordered list has a class of `open` (`if ( nav.className === "open" ) { … }`). If it doesn't, the JavaScript adds the class to reveal the links; and if it *does*, it removes the class, and hides the navigation from view.

And if we want to get a little fancy—and of *course* we want to get a little fancy—we can add a CSS transition on the `max-height`, allowing the list to subtly telescope in and out of view (**FIG 2.7**):

```
.enhanced .nav ul {
  max-height: 0;
  overflow: hidden;
  transition: max-height 0.25s ease-out;
}
```

And we're done! With a little bit of JavaScript, we can show (or hide) an element of our design when it's clicked on, all by adding (or removing) a class.

(Quick aside: while `overflow` is a CSS property older than time, it's worth noting that an astonishingly high number of

mobile browsers don't implement it correctly. If any part of your design uses `overflow: auto` to create scrollable areas, I recommend Filament Group's Overthrow.js library (http://bkaprt.com/rdpp/02-05/), which properly detects support for `overflow` while weeding out the browsers that claim to support the property but don't.)

While the show/hide toggle works beautifully, that doesn't mean the effect's necessarily appropriate for *all* breakpoints. The toggle's really only valuable on smaller viewports, where the layout's a bit tighter; when the viewport gets wider, we can display the entire navigation, locked-up with the logo (**FIG 2.8**). But since the entire effect is driven by our CSS, we can override it above a certain breakpoint with a media query:

```
@media screen and (min-width: 39em) {
  .page .nav ul {
    overflow: auto;
    max-height: inherit;
  }
}
```

Now, when the viewport reaches a minimum width of `39em`, we've disabled the `overflow: hidden` on the list, and returning its `max-height` to a normal, default value. As a result, our list is no longer hidden from view, allowing us to style it like a more traditional masthead.

...phew!

That might seem like a lot of work, but we're simply adding or removing a class with a little JavaScript, and using that class to control the visibility of our navigation. And really, that's the basic mechanism of nearly all show/hide toggles. MSNBC.com's responsive site does this very thing, in fact (**FIG 2.9**): on widescreen displays, tapping or clicking on the primary categories reveals secondary menus; but on smaller displays, tapping on an icon reveals the entire navigation, with submenus also expandable within it.

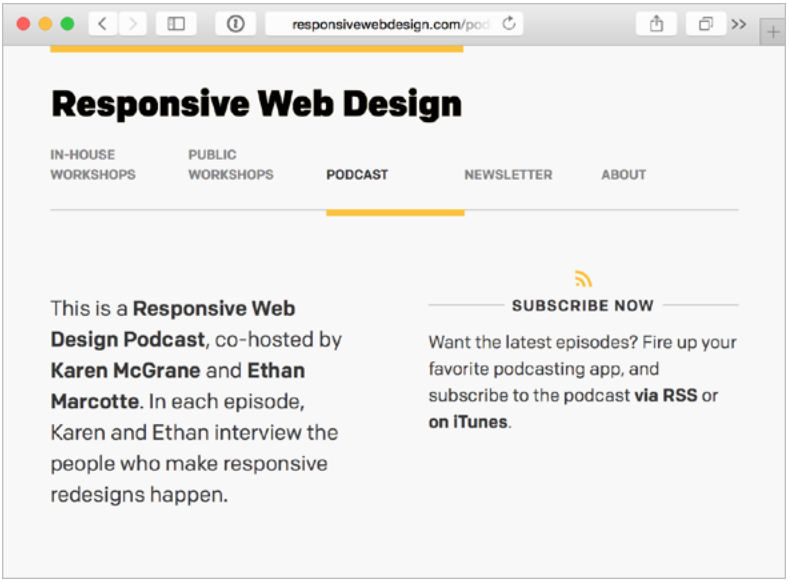

FIG 2.8: For wider screens, we can disable the show/hide toggle, and just keep our links in view.

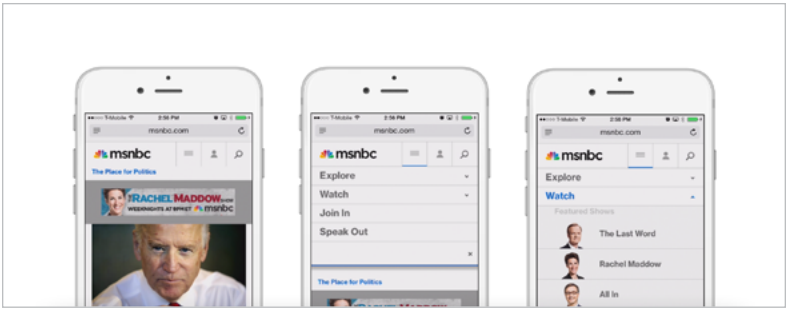

FIG 2.9: MSNBC's responsive navigation uses a top-level toggle to reveal its menu on smaller screens. Additionally, users can open nested menus by tapping or clicking on the relevant sections.

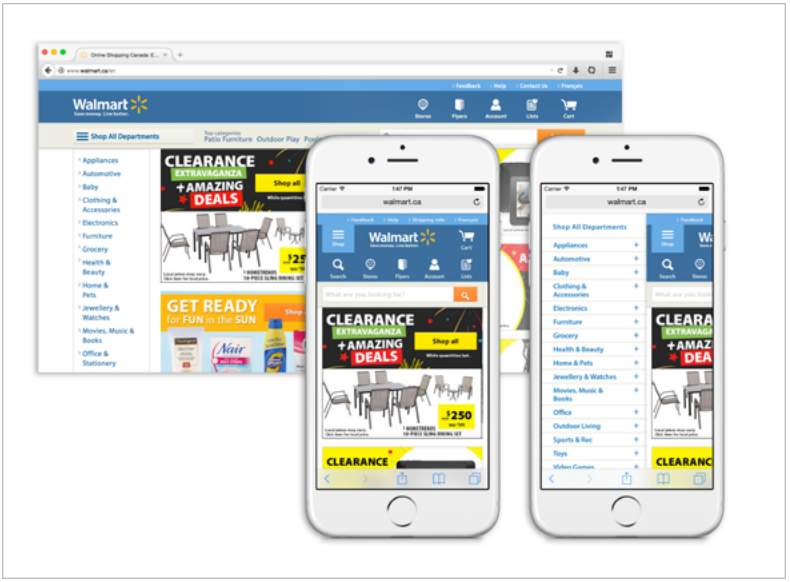

FIG 2.10: Walmart.ca's navigation is hidden "off-canvas" on narrower viewports, but visible by default on wider ones.

THE OFF-CANVAS MENU

A variant of the show/hide toggle is what's colloquially referred to as the *off-canvas menu*. While this pattern first gained traction in native mobile applications, it's recently seen use in responsive and mobile websites (http://bkaprt.com/rdpp/02-06/). As it happens, Walmart.ca adopted this approach in their recent responsive redesign (**FIG 2.10**). On wider screens, the navigation's visible at the left. But on smaller screens, tapping or clicking on the Shop icon causes the entire navigation to *slide* in from the left, positioned just beyond the visible canvas.

From a mechanical standpoint, this isn't considerably different from our old, trusty show/hide toggle: we're still concealing our navigation, and then asking our users to interact with an element to toggle its visibility. If executed well, the off-canvas menu can convey an extra layer of depth and dimensionality in

your layout. It does, however, require extra care to make sure it's built accessibly, and that it doesn't break the experience for all but the latest browsers (http://bkaprt.com/rdpp/02-07/).

Read the rest of this chapter and more when you buy the book!