Creating a Mobile-First Responsive Web Design

By Brad FrostPublished April 16, 2012Updated April 16, 2012

Introduction

We're going to walk through how to create an adaptive web experience that's designed mobile-first. This article and demo will go over the following:

- Why we need to create mobile-first, responsive, adaptive experiences
- How to structure HTML for an adaptive site in order to optimize performance and prioritize flexibility
- How to write CSS that defines shared styles first, builds up styles for larger screens with media queries, and uses relative units
- How to write unobtrusive Javascript to conditionally load in content fragments, take advantage of touch events and geolocation
- What we could do to further enhance our adaptive experience

The Need for Adaptivity

As the web landscape becomes increasingly complex, it's becoming extremely important to deliver solid web experiences to a growing number of contexts. Thankfully, responsive web design gives web creators some tools for making layouts that respond to any screen size. We'll use fluid grids, flexible images and media queries to get the layout looking great regardless of the size of the device's screen dimensions.

However, mobile context is much more than just screen size. Our mobile devices are with us wherever we go, unlocking entire new use cases. Because we constantly have our mobile devices with us, connectivity can be all over the board, ranging from strong wi-fi signals on the couch to 3G or EDGE when out and about. In addition, touch screens open new opportunities to interact directly with content and mobile ergonomics lead to different considerations when designing layout and functionality.

In order to create a site that's truly designed for mobile context and not just for small screens, we want to ensure that we tackle the many challenges of mobile development upfront. The constraints of the mobile context force us to focus on what content is essential and how to present that content as quickly as possible. Building fast-loading, optimized experiences mobile first has a trickle down (or up, depending on how you look at it) effect for tablet, desktop and other emerging contexts.

What We're Making: The Humble Product Detail Page

View the demo

The demo we're making is a simple e-commerce product detail page for a fictitious t-shirt company. Why choose this? E-commerce sites can have many use cases across contexts. For example, 70% of smartphone owners use their mobile phones to influence in-store purchases. So while we'll make sure that purchasing the product is as easy as possible, we'll also try to make the product reviews accessible and utilize the user's location to enhance the mobile experience.

Structure

Authoring lean, semantic HTML5 markup keeps adaptive experiences manageable and accessible, and also provides opportunities for enhanced experiences (quick example: using proper HTML5 input types brings up the appropriate virtual keyboard on many touch devices). Semantic markup is extremely portable and can be accessed by many mobile devices, tablets, desktop browsers and future web-enabled devices, regardless of feature set or capability.

Setting the viewport

In order to accommodate for sites not optimized for mobile screens, many modern mobile browsers set a larger browser viewport, which allows for better viewing of non-mobile-optimized sites. Users can then pinch-to-zoom in on the content they want. That's fine for non-mobile experiences, but because we're optimizing our experience for mobile browsers, we'll use the viewport meta tag to set the screen width to the device width:

```
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

It's important to note that we're not disabling the user's ability to zoom the page (which you could do by adding user-scalable=no to the content attribute), even though we're optimizing the content for small screens. It's recommended to keep user zooming enabled to keep things as accessible as possible. However, there are use cases to disable user-zooming, such as if you're including fixed positioned elements.

Content fragments

In order to keep the experience as lightweight as possible and to improve the perceived loading time, we're creating two additional HTML documents for our auxiliary content, reviews.html andrelated.html. Because this content isn't required for the main use case (buying the product) and includes a number of images, we won't load it by default to keep the initial page size down. By default the content is accessible via links on the page, but if a certain level of javascript support is present, we'll conditionally load the content when the user requests it or when the resolution reaches a certain breakpoint.

HTML Special Characters

A simple technique to reduce the need for background images (thereby saving HTTP requests) is to use HTML special characters for simple shapes. In the case of our rating stars, we're using ★ to create a solid star (★) and ☆ to create empty stars (☆) for our ratings. And because it's HTML and not an image, it stays crisp even on high resolution screens.

The tel: URI Scheme

Another simple yet effective technique we're including in our footer is a clickable link to the customer service number. This is accomplished by using the tel URI scheme, which looks like this:

```
<a href="tel:+18005550199">1-800-555-0199</a>
```

We sometimes forget that mobile devices can make phone calls, and also that some desktop configurations can launch VoIP applications to initiate a phone call. We're including an easy way for users to facilitate a phone call, which in some cases might make sense (i.e. a mobile user who might prefer finishing the transaction over the phone versus going through a checkout flow on their mobile device).

Now that we have a strong, semantic foundation in place, let's move onto adding style enhancements.

Style

When crafting our CSS, we'll do everything in our power to keep things lightweight and as fluid as possible. We understand that all these devices have many different screen sizes, and that tomorrow's devices won't have the same resolutions as today's. Because screen size is an unknown, we'll use the content itself to determine how the layout should adjust to its container.

Separate style sheet for larger screens

We're creating two separate CSS files, style.css and enhanced.css in order to deliver basic styles for screens less than 40.5em and using media queries to serve up enhanced styles for screens larger than 40.5em.

```
<link rel="stylesheet" type="text/css" href="style.css" media="screen,
handheld" />
<link rel="stylesheet" type="text/css" href="enhanced.css" media="screen
and (min-width: 40.5em)" />
<!--[if (lt IE 9)&(!IEMobile)]>
<link rel="stylesheet" type="text/css" href="enhanced.css" />
<![endif]-->
```

We're using the conditional code gives us greater flexibility over our styles. Alternately, we could use respond.js to deliver enhanced styles to IE.

We're using the em unit instead of px to maintain consistency with the rest of our relative units and account for user settings like zoom level. Also, the content should determine the breakpoint (we're using 40.5em as a breakpoint) because device dimensions are too varied and are always changing so are therefore unreliable.

Mobile-First Styles

Starting with baseline shared styles and introducing more advanced layout rules when screen size permits keeps code simpler, smaller and more maintainable. Here's just a quick example to demonstrate this point:

```
/*Large screen styles first - Avoid*/
.product-img {
  width: 50%;
  float: left;
}
@media screen and (max-width: 40.5em) {
  .product-img {
    width: auto;
    float: none;
  }
}
```

We want to avoid complexity as much as we can, so here's what a mobile-first approach looks like:

```
@media screen and (min-width: 40.5em) {
  .product-img {
    width: 50%;
    float: left;
  }
}
```

Instead of declaring large screen rules first only to override them for smaller screens, we'll simply define rules as more real estate becomes available. The web by default is a fluid thing so we'll do our best to work with it instead of against it. It's important to note that some mobile browsers (Symbian browsers, Blackberry Bryan Rieger puts it, "the absence of support for @media queries is in fact the first @media query."

Applying Media Queries

We're continuing our mobile-first style when we apply our media queries. Our related product list starts off two to a row, but increases to 3 in a row when the screen size is at least 28.75em

wide (roughly the size of mobile phones in landscape mode) and then to 6 to a row when the screen size is at least 40.5em (roughly tablets in portrait mode or small desktop screens).

```
/*Default styles*/
.related-products li {
  float: left;
  width: 50%;
}

/*Display 3 per row for medium displays (like mobile phones in landscape or smaller tablets)*/
@media screen and (min-width: 28.75em) {
  .related-products li {
    width: 33.3333333%;
  }
}
/*Display 6 to a row for large displays (like medium tablets and up) */
@media screen and min-width: 40.5em) {
  .related-products li {
    width: 16.6666667%;
  }
}
```

Assuming small screen by default allows us to support more platforms and also makes it easy add more breakpoints without having to modify existing styles. Defining styles as they're needed also keeps file size down, reduces complexity and keeps code more maintainable.

Using Relative Units

We're using percentages and em units in our design in order to keep things as flexible as possible. Relative units are far more compatible with the tremendous variance brought on by screen size, pixel density and zoom level.

While media queries are responsive web design's secret sauce, we want our fluid grids to do most of the work. Maintaining a whole slew of set-width styles across many media queries can become unwieldy, so we'll make sure the stylesheet's foundation is entirely flexible. Ethan Marcotte provides a formula for converting dimensions and font sizes from pixel-based to relative units:

```
target ÷ context = result
```

Using CSS to Reduce Requests

Too many HTTP requests can be a huge killer for performance, especially on mobile. We're incorporating some CSS techniques to save HTTP requests which will improve the site's performance. Using CSS gradients instead of background images reduces the amount of image requests and gives us more control over the design. We're including the appropriate vendor

prefixes to ensure maximum compatibility (there are tools for this) and hoping that one day that these rules will become standardized to save us some time.

```
/*Using CSS gradients instead of background images*/
header[role="banner"] {
  position: relative;
  background: #111;
  background: +linear-gradient (top, #111 0%, #222 100%);
}
```

We're also using data URIs instead of background images for some of the smaller icons (for icons like search, social features and location). While data URIs might look a bit ugly and can increase up the stylesheet file size, the reduction of requests results in a faster perceived download time.

```
/*Using a Data URI for Background Image*/
.find-nearby {
  background:
url(
no-repeat 100% 43%;
}
```

Behavior

Now that we have our structure and style in place, we'll add JavaScript enhancements to add functionality to the navigation, image gallery and auxiliary content.

Navigation

Navigation can be especially tricky for adaptive experiences. Top navigation is common for desktop sites, but top navigation can crowd the screen and push down the primary content on small screens. We want to highlight the product and not the site navigation, so we'll do our best to get the navigation out of the way. in our markup we've created a list called #nav-anchors, which will be used to toggle the visibility of the navigation and search bar for small screens.

```
<ul  id="nav-anchors" class="nav-anchors">
  <li><a href="#nav" id="menu-anchor">Menu</a></li>
  <li><a href="#search" id="search-anchor">Search</a></li>
</ul>
<form id="search" action="#" method="post" class="search reveal">
  <fieldset>
    <legend>Search the Site</legend>
    <input type="search" placeholder="Search Store" />
    <input type="submit" value="Search" />
  </fieldset>
</form>
<nav id="nav" class="nav reveal">
  <ul role="navigation">
    <li><a href="#">T-shirts</a></li>
    <li><a href="#">Hoodies</a></li>
    <li><a href="#">Pants</a></li>
  </ul>
</nav>
```

We'll add a resize listener which will determine whether there's enough room to show the navigation and search bar.

```
$(w).resize(function(){ //Update dimensions on resize
  sw = document.documentElement.clientWidth;
  sh = document.documentElement.clientHeight;
  checkMobile();
});

//Check if Mobile
function checkMobile() {
  mobile = (sw > breakpoint) ? false : true;

  if (!mobile) { //If Not Mobile
    $('[role="tabpanel"],#nav,#search').show(); //Show full navigation and search
  } else { //Hide
    if(!$('#nav-anchors a').hasClass('active')) {
      $('#nav,#search').hide(); //Hide full navigation and search
    }
  }
}
```

Image Gallery

By default the image gallery is simply a large image with thumbnail images that click through to their larger counterparts. This means that they're accessible to browsers and devices with poor or no JavaScript support.

```
<div id="product-img" class="product-img">
  <figure class="img-container" id="img-container">
    <img src="images/product_img_1.jpg" alt="Super Ffly T-shirt" />
  </figure>
  <nav>
    <ul>
      <li><a href="images/product_img_1.jpg"><img
src="images/product_img_1_thumb.jpg" alt="Super Ffly Men's Shirt"
/></a></li>
      <li><a href="images/product_img_2.jpg"><img
src="images/product_img_2_thumb.jpg" alt="Super Ffly Women's Shirt"
/></a></li>
      <li><a href="images/product_img_3.png"><img
src="images/product_img_3_thumb.jpg" alt="Ffly Logo" /></a></li>
    </ul>
  </nav>
</div>
```

We'll build an image carousel from the available thumbnail images:

```
function buildGallery() {
  container.html('<div id="img-list"><ul /></div>');
  imgList = $('#img-list');
  nav.find('a:first').addClass('active');

  //For Each Navigation Link
  nav.find('a').each(function() {
    var $this = $(this);
    var href = $this.attr('href');

    //Prepare list item with image source in data attribute
    arr += '<li data-imgsrc="'+href+'"></li>';
  });

  //Append to #img-list
  imgList.find('ul').append(arr);

  //Nav Thumbnail Click
  nav.on('click', 'a', function(e) {
    var pos = $(this).parent().index();
    e.preventDefault();
    loadImg(pos);
    if(swipeEnabled) {
      mySwipe.slide(index, 300);
    }
    updateNav(pos);
  });
}
```

To enhance the experience further, we're using Modernizr to detect for the presence of touch events and CSS transitions, and if they are supported, we'll load in a library called SwipeJS to

make a touch-friendly image carousel.

```
Modernizr.load({
  test: Modernizr.touch && Modernizr.csstransitions,
  yep: 'js/swipe.js',
  complete: function() {
    if (Modernizr.touch && Modernizr.csstransitions) {
      swipeEnabled = true;
      buildSwipe();
    }
  }
});

//Build Swipe Carousel
function buildSwipe() {
  //Initialize Swipe.js
  w.mySwipe = new Swipe(document.getElementById('img-list'), {
    callback: function(event, index, elem) {
      updateNav(index);
      loadImg(index + 1);
    }
  });
}
```

We now have an accessible image gallery with added enhancements for touch-enabled devices.

Related Content

In order to keep Initial page size down, we're not loading auxiliary content, namely the related t-shirts and product reviews, by default. Instead, they exist as their own HTML pages, which are accessed by links as a default behavior.

```
<section class="aux related-products" id="related-products">
  <header id="tab-related">
    <a href="related.html">
      <h2>Similar T-shirts</h2>
    </a>
  </header>
</section>
<section class="aux reviews" id="reviews">
  <header id="tab-reviews">
    <a href="reviews.html">
      <h2>8 Reviews</h2>
      <ol class="star">
        <li class="on">&#9733;</li>
        <li class="on">&#9733;</li>
        <li class="on">&#9733;</li>
        <li class="on">&#9733;</li>
        <li>&#9734;</li>
      </ol>
    </a>
  </header>
</section>
```

We'll pull in the related content when one of two conditions are met: When a small-screen user clicks the related shirts or product reviews links When the screen has enough room to load in the auxiliary content.

```
//Check if Mobile
function checkMobile() {
  if(sw > breakpoint) {
    mobile = false; //Not Mobile
  } else {
    mobile = true; //Mobile
  }

  if (!mobile) { //If Not Mobile
    loadAux(); //Load auxiliary content
  }
}

//Set up Auxiliary content
function loadAux() {
  var $aux = $('.aux');
  $aux.each(function(index) {
    var $this = $(this);
    var auxLink = $this.find('a');
    var auxFragment = auxLink.attr('href');
    var auxContent = $this.find('[role=tabpanel]');
    if (auxContent.size()===0 && $this.hasClass('loaded')===false) {
      loadContent(auxFragment,$this);
    }
  });
}

function loadContent(src,container) { // Load Tab Content
  container.addClass('loaded');
  $('<div role="tabpanel" />').load(src +' #content > div',function() {
    $(this).appendTo(container);
  });
}
```

note: we're using screen size to determine when to load in content, but this is in no way perfect. Keep an eye out for navigator.connection for a better way to determine whether it's worth introducing extra content.

Geolocation

Leveraging user location to deliver enhanced experiences is an important aspect of mobile development. Thankfully geolocation is one of the best supported Features across mobile browsers (as well as most desktop browsers). The fallback functionality could be a simple form where the user simply inputs their ZIP code to find store near them.

Next Steps

Adaptive Images

Our demo isn't incorporating many large images, but it's best practice to load in mobile optimized images by default then conditionally load in larger images only when needed. There

are lots of different techniques for responsive images, both client-side and server side. We've done a lot so far to be mindful of performance, and optimizing images is an easy way to optimize performance even further.

Less JS

Keeping pages as lightweight as possible is important for performance, so we should look to optimize scripts as much as possible. We're using the jQuery library for our demo, but we're definitely not using all of it. We could look into using Closure Compiler to strip out unused bits of the library to keep things as lightweight as possible while still taking advantage of what jQuery offers. Alternately, we could look into micro-frameworks like Zepto.js and others, but they typically don't necessarily offer the best cross-browser support. Writing vanilla Javascript could avoid additional heft but can be more difficult to author and harder to maintain. Ultimately, each approach has its pros and cons, just be sure to consider the tradeoffs when making these decisions.

Offline Access

It's increasingly important to make sure web experiences are accessible offline, especially when considering mobile users with variable connectivity. Thankfully, appcache and other offline techniques gives us a way to keep our resources accessible even when the user is offline.

Wrapping Up

We've created an experience that is mindful to user context and adapts both layout and functionality based the browser and device's features. We've also set up a foundation that can adapt to future devices and browsers. Here's some key takeaways:

- Author semantic HTML5 markup as a foundation for adaptive experiences.
- Create mobile first CSS to keep things lightweight, simple and maintainable.
- Use relative units like ems and percentages to keep styles as fluid and flexible as possible.
- Let content determine the breakpoints for media queries.
- Exploit opportunities to reduce HTTP requests by conditionally-loading content and using HTML characters, CSS gradients, Data URIs and more
- Author unobtrusive javascript and use tools like Modernizr to detect features.
- Take advantage of mobile-centric features like touch events, telephone links and geolocation to deliver enhanced experiences to mobile users.

Creating adaptive experiences allows your content to go more places, which means more opportunities to reach potential customers wherever they may be. By adhering to the principles of progressive enhancement and addressing constraints first, we're laying a future-friendly foundation that gives our site a better chance of working in future browsers and environments.