IMPLEMENTING RESPONSIVE DESIGN

Building sites for an anywhere, everywhere web

Tim Kadlec

IMPLEMENTING RESPONSIVE DESIGN

Building sites for an anywhere, everywhere web

Tim Kadlec

New Riders
VOICES THAT MATTER™

IMPLEMENTING RESPONSIVE DESIGN: BUILDING SITES FOR AN ANYWHERE, EVERYWHERE WEB
Tim Kadlec

NEW RIDERS
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at: www.newriders.com
To report errors, please send a note to
errata@peachpit.com

New Riders is an imprint of Peachpit, a division of Pearson Education.

Project Editor: Michael J. Nolan
Development Editor: Margaret S. Anderson/ Stellarvisions
Technical Editor: Jason Grigsby
Production Editor: Rebecca Winter
Copyeditor: Gretchen Dykstra
Indexer: Joy Dean Lee
Proofreader: Rose Weisburd
Cover Designer: Aren Straiger
Interior Designer: Mimi Heft
Compositor: Danielle Foster

Find code and examples available at the companion website, www.implementingresponsivedesign.com.

NOTICE OF LIABILITY
The information in this book is distributed on an "As Is" basis without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

ISBN 13: 978-0-321-82168-3
ISBN 10: 0-321-82168-8

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

For my wife and our
beautiful daughters.

Acknowledgements

It is frequently said that writing a book is a lonely, solitary act. Perhaps that is true in some cases, but it certainly wasn't the case with this book. If this book is any good, it's because of all the hard work, patience and feedback provided by everyone who helped along the way.

I owe a huge thank you to…

Michael Nolan, who invited me to write a book in the first place. Thanks for being willing to gamble on me.

Margaret Anderson and Gretchen Dykstra for overlooking my horrible misuse of punctuation and for generally making it sound like I know how to write much better than I do.

Danielle Foster for making the book look so fantastic, and putting up with a few last minute adjustments. Also, to Rose Weisburd, Joy Dean Lee, Aren Straiger, Mimi Heft, Rebecca Winter, Glenn Bisignani and the rest of the team at New Riders for helping make this book come to life.

Ed Merritt, Brad Frost, Guy Podjarny, Henny Swan, Luke Wroblewski, Tom Maslen and Erik Runyon for their incredible contributions. By being willing to share their expertise and experiences, they've made this a much richer book than it would have otherwise been.

Jason Grigsby for making sure I wasn't making things up along the way and for providing valuable (and frequently hilarious) feedback and encouragement throughout. Not only is Jason one of the smartest people I know, but he's also one of the most helpful. I'm thankful to be able to call him a friend.

Aaron Gustafson for writing such a great foreword. I've been learning from Aaron since I first started working on the web—to say I'm humbled and honored that he agreed to write the foreword is an understatement.

Stephen Hay, Stephanie Rieger, Bryan Rieger, Brad Frost, Derek Pennycuff, Ethan Marcotte, Chris Robinson, Paul Thompson, Erik Wiedeman, Sara Wachter-Boettcher, Lyza Danger Gardner, Kristofer Layon, Zoe Gillenwater, Jeff Bruss, Bill Zoelle, James King, Michael Lehman, Mat Marquis, Nishant Kothary, Andy Clarke, Ronan Cremin, Denise Jacobs and Cennydd Bowles for the insights, feedback and encouragement they provided along the way. This book owes a great deal to their collective awesomeness.

To everyone whose conversations, both in person and online, inspired the discussion that takes place in this book. This is an awesome community we have going and I'm proud to be a part of it.

My mom and dad for their love and words of encouragement throughout.

My lovely daughters for reminding me it was ok to take a break every once in awhile to play and for filling each day with laughs, kisses and hugs.

And my incredible wife, Kate. This book, and anything else I do that is any good, is a direct result of her loving support and encouragement. There are no words powerful enough to express how thankful I am for her.

Foreword

By Aaron Gustafson

A few years back, photography legend Chase Jarvis smartly observed that "the best camera is the one that's with you." It was a mildly shocking assertion at the time, but it rings true: the perfect shot is rarely planned. Rather, it sneaks up on you.

Perhaps the light is perfectly accentuating the fall foliage on your late afternoon stroll. Or perhaps your infant daughter just pulled herself up on two legs for the first time. In moments like these, it doesn't matter that your Leica is sitting on a shelf in the other room or that you left your Rebel in the car—what matters is that you have a camera, however crude, in your pocket and can capture this serendipitous and ephemeral moment.

Riffing on Jarvis's idea, Stephanie Rieger has made the case that the best browser is the one you have with you. After all, life is unpredictable. Opportunities are fleeting. Inspiration strikes fast and hard.

Imagine yourself as a cancer researcher. You've been poring over a mountain of research for months, looking for a way to increase interferon-gamma production in an effort to boost the body's natural ability to inhibit the development of tumors. Your gut tells you that you're close to an answer, but it's just out of reach. Then one morning, while washing the exhaustion off in a nice hot shower, it hits you. Eureka! You think you've got it—you just need to refer back to that paper you read last week.

Dripping, you leap from the tub and land on the bath mat. Without even grabbing a towel, you pluck your mobile off the counter and head to the journal's site, only to find yourself re-routed to a "lite" version of the website that shows you only general information about the publication and prompts you to subscribe.

Your fingers leave wet streaks across the screen as you frantically scroll down the page to find the inevitable link to "View Full Site" and click it. As the screen loads, you find yourself hovering 30,000 feet above a patchwork quilt of a homepage that could only have been designed by committee.

Several minutes of pinching, zooming, and typing later, you finally find the article, only to discover it's a PDF and nearly impossible to read on your tiny screen. Dejected, you put down the phone and sulk back into the shower, hoping it will wash away your disappointment.

Sadly, browsing the web on mobile is all too often a frustrating (and occasionally dehumanizing) endeavor. But it doesn't have to be.

In the pages of this very book, my friend Tim clearly outlines the steps you can (and indeed should) take to ensure that the sites you help create offer each user a fantastic experience, tailored to the capabilities of her device and respectful of her time, patience, and data limits. Don't let his small town charm fool you: Tim knows this stuff inside and out. I learned a ton from this book and I know you will too.

Aaron Gustafson is the author *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement* (Easy Readers, 2011)

This page intentionally left blank

Contributions

The discussion around responsive design moves fast. Very fast. This book is intended to be a synthesis of the incredible discussion that is taking place in our community about this topic. To that end, I asked several people if they would be willing to contribute short pieces based on their recent projects and research.

Here are the contributions you'll find, in order of their appearance in the book:

- Vertical Media Queries, by Ed Merritt, page 70
- Performance Implications of Responsive Design, by Guy Podjarny, page 102
- Small Phone, Big Expectations, by Tom Maslen, page 136
- Responsive Design and Accessibility, by Henny Swan, page 141
- Selling Responsive Design, by Brad Frost, page 159
- RESS in the Wild, by Erik Runyon, page 210
- Beyond Layout, by Luke Wroblewski, page 242

Each of the seven contributors featured are experimenting with the cutting edge of responsive design. They're implementing the techniques discussed in this book, and pushing the discussion forward. I'm incredibly honored to be able to include their contributions—contributions based on hard-earned experience—in this book.

Contents

CHAPTER 3: MEDIA QUERIES 53

CHAPTER 4: RESPONSIVE MEDIA 95

This page intentionally left blank

CHAPTER 4
RESPONSIVE MEDIA

> Look! We’ve figured it seventeen different ways, and every time we figured it, it was no good, because no matter how we figured it, somebody don’t like the way we figured it.
>
> —BUDDY HACKETT AS BENJY BENJAMIN IN IT’S A MAD MAD MAD MAD WORLD

When it comes to rich experiences online, we have a love/hate relationship. On one hand, beautiful images and interesting videos help to provide a deeper, more pleasant experience. On the other hand, including many images and videos on a page results in a slow loading time, which can be very frustrating. It takes careful consideration and planning to give our users the best of both worlds: a beautiful experience that loads as quickly as possible.

Using the methods outlined in the first three chapters, we've built ourselves a responsive site. It looks good on desktops, on tablet devices, and on smartphones. Users can resize the browser window to their hearts' content, and the layout will adjust accordingly. If delivering a responsive approach were this easy, this book would be short indeed. There's still plenty of room to tidy things up. The images, in particular, are an issue.

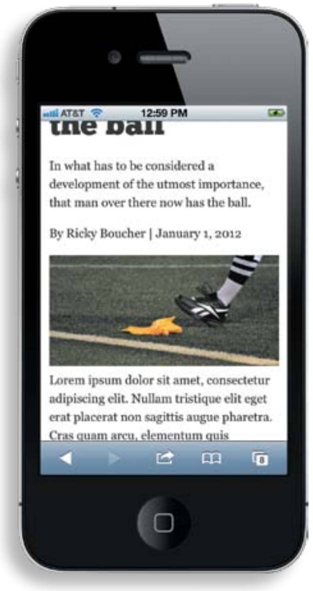
In this chapter, we'll discuss:

- Why performance matters
- How to conditionally load images
- What responsive image solutions are available, and their limitations
- How to swap out background images without downloading multiple images
- How to conditionally load web fonts
- What's ahead for responsive images
- How to make embedded video scale while maintaining its aspect ratio
- What to do with responsive advertising

What's the problem?

Once we hit the final breakpoint (1300px), the images associated with the "More in Football" section look a little worse for wear. Other than that, the images appear sharp and crisp.

We could probably improve the lead-in photograph for small screens. If the small version of the image were more tightly cropped, the image would maintain its initial impact, even when scaled down on the small screen. As it is, the flag and foot start to get lost at such a small size (**Figure 4.1**).



the ball

In what has to be considered a development of the utmost importance, that man over there now has the ball.

By Ricky Boucher | January 1, 2012

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam tristique elit eget erat placerat non sagittis augue pharetra. Cras quam arcu, elementum quis

Figure 4.1 On small-screens, the flag and foot in the main image start to lose their impact.

The main problem though is not in how the images look, but in how much they *weigh*, how much demand they place on performance. Currently, the same images are being loaded regardless of the device in use. That means, for example, the 624px lead-in image is being downloaded even on small screens where a 350px image is all that's needed. The page performance is suffering, and that's a big deal to people visiting the site.

Performance

Unfortunately, performance is treated as an afterthought on many projects. A quick look at the data reveals that it should be anything but.

Most of us working with the Web have faster connections than the average Internet user. As a result, we experience the Web differently. Our users, however, are keenly aware of how painful it is to use a poorly performing site.

In 2009, the major shopping comparison site Shopzilla improved its page load time from 4 to 6 seconds to 1.5 seconds. The results were stunning. The site's conversion rate increased by 7 to 12 percent and page views jumped a whopping 25 percent.[1]

[1] "Shopzilla Site Redesign–We get what we measure" at www.scribd.com/doc/16877317/Shopzillas-Site-Redo-You-Get-What-You-Measure

Mozilla found similar results when it trimmed page load time by 2.2 seconds: download conversions went up by 15.4 percent, which translated into an estimated 10.28 million additional downloads of Firefox per year![2]

The situation is much more dire for mobile phones. Networks are slower, hardware is less capable, and you have to deal with the messy world of data limitations and transcoding methods. In spite of all this, user expectations remain the same. In fact, 71 percent of mobile users expect sites to load on their phones as quickly as or faster than on their home computers.[3]

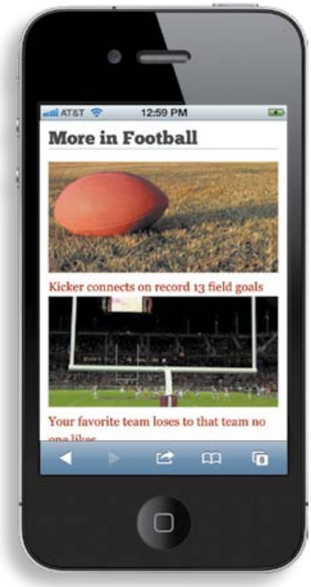
This is bad news for our site as it currently stands. Both the logo and article lead-in photo are very large. The article lead-in photo is 624px wide and weighs around 50KB. The small-screen layout could get away with using a much smaller image (somewhere around 300px), but we're still passing along the large desktop image instead of something more appropriate. Removing the amount of data sent down the pipe is an important consideration, and one we can't afford to ignore.

A quick assessment of the page reveals the following images that could be optimized:

- **The images for the "More in Football" section.** Each of these is only 300px, but they're really not needed on the small screen. In fact, they take up a lot of screen real estate and look out of proportion with the content (**Figure 4.2**). On the small screen, users have a better experience if only the headlines are displayed—not the images.
- **The article lead-in image.** The lead-in image is a whopping 624px and weighs in at just under 50KB. On small screens, an image half the size would work just as well. In addition, if the small-screen version of the image was more tightly cropped, the visual focus on the flag would be stronger.
- **The logo.** The logo weighs in at 10KB, so it's much lighter than the lead-in article. It is, again, about twice as big as it needs to be.

[2] "Firefox & Page Load Speed–Part II" at http://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-–-part-ii/

[3] "What Users Want from Mobile" at www.gomez.com/resources/whitepapers/survey-report-what-users-want-from-mobile/



More in Football

Kicker connects on record 13 field goals

Your favorite team loses to that team no one likes

Figure 4.2 The images in the “More Football” section take up a lot of precious screen real estate on small-screen devices.

Selectively serving images to mobile

Let’s start by removing the images in the “More in Football” section from the core experience. It might be tempting to just use `display:none` and call it a day, but that doesn’t fix the problem, it only hides it.

An image set to `display:none` will still be requested and downloaded by the browser. So while the image won’t show up on the screen, the issue of the extra request and weight is still there. Instead, as usual, the correct approach is to start with mobile first and then progressively enhance the experience.

Begin by removing the images from the HTML entirely:

```
<ul class="slats">
    <li class="group">
        <a href="#">
            <h3>Kicker connects on record 13 field goals</h3>
        </a>
    </li>
    <li class="group">
        <a href="#">
            <h3>Your favorite team loses to that team no one likes</h3>
        </a>
    </li>
    <li class="group">
        <a href="#">
            <h3>The Scarecrows Win 42-0</h3>
        </a>
    </li>
</ul>
```

● ***Custom data attributes***
Preceded by a `data-` prefix, these attributes store custom data private to the page, often for scripting purposes.

Obviously, the images will not load with this HTML. On the small-screen display, that's the way it'll stay. For the larger sizes, a little JavaScript will bring the images back. Using the HTML5 `data-*` attributes as hooks, it's easy to tell the JavaScript which images to load:

```
<ul class="slats">
    <li data-src="images/ball.jpg" class="group">
        <a href="#">
            <h3>Kicker connects on record 13 field goals</h3>
        </a>
    </li>
    <li data-src="images/goal_post.jpg" class="group">
        <a href="#">
            <h3>Your favorite team loses to that team no one likes</h3>
        </a>
    </li>
    <li data-src="images/ball_field.jpg" class="group">
        <a href="#">
            <h3>The Scarecrows Win 42-0</h3>
        </a>
    </li>
</ul>
```

JavaScript

The first thing to add is a quick utility function to help select elements. It's not necessary, but it's definitely useful to have around:

```
q : function(query) {
    if (document.querySelectorAll) {
        var res = document.querySelectorAll(query);
    } else {
        var d = document,
        a = d.styleSheets[0] || d.createStyleSheet();
        a.addRule(query,'f:b');
        for(var l=d.all,b=0,c=[],f=l.length;b<f;b++) {
            l[b].currentStyle.f && c.push(l[b]);
            a.removeRule(0);
            var res = c;
        }
        return res;
    }
}
```

If you're unfamiliar with native JavaScript, that might look a bit messy. That's OK. All the function does is take a selector, and return the elements that match it. If you can grasp the code, that's great. If not, as long as you understand what it accomplishes, that's enough for our purposes.

Armed with that function, the part that actually loads the images is pretty straightforward:

```
1.  //load in the images
2.  var lazy = Utils.q('[data-src]');
3.  for (var i = 0; i < lazy.length; i++) {
4.      var source = lazy[i].getAttribute('data-src');
5.      //create the image
6.      var img = new Image();
7.      img.src = source;
8.      //insert it inside of the link
9.      lazy[i].insertBefore(img, lazy[i].firstChild);
10. };
```

Line 2 grabs any elements with a `data-src` attribute applied. Then, in line 3 the script loops through those elements. In lines 4–7, the script creates a new image for each element using the value of the `data-src` attribute. The script then inserts the new image (line 9) as the first element within the link.

With this JavaScript applied, the images aren't requested right away. Instead, they're loaded after the page has finished loading, which is what we want. Now, we just have to tell the script not to load for small screens.

Guy Podjarny

PERFORMANCE IMPLICATIONS OF RESPONSIVE DESIGN

Guy Podjarny, or Guypo for short, is a web performance researcher and evangelist, constantly chasing the elusive instant web. He focuses heavily on mobile web performance, and regularly digs into the guts of mobile browsers. He is also the author of Mobitest, a free mobile measurement tool, and contributes to various open source tools. Guypo was previously the co-founder and CTO of Blaze.io, later acquired by Akamai, where he now works as a Chief Product Architect.

Responsive Web Design (RWD) tackles many problems, and it's easy to get lost in questions around how maintainable, future-friendly, or cool your responsive website will be. In the midst of all of these, it's important to not lose sight of how *fast* will it be. Performance is a critical component in your user's experience, and many case studies demonstrate how it affects your users' satisfaction and your bottom line.

Today, smartphone browsers are often redirected to dedicated mobile websites, known as *mdot* sites, which tend to be significantly lighter in content and visuals than their desktop counterparts. This translates to having fewer images, scripts and stylesheets to download, which helps make those websites faster. The equation is simple—downloading fewer bytes with fewer requests is faster than having more of both.

Responsive websites, however, don't follow this pattern. I recently ran a performance test on 347 responsive websites (All the websites listed on http://mediaqueri.es/ in March, 2012). I loaded the homepage of each in a Google Chrome browser window resized to 4 different sizes, ranging from 320x480 to 1600x1200. Each page was loaded multiple times using www.webpagetest.org, a web performance measurement tool.

The results were depressing. Despite changing their look across window sizes, the weight and load time of the website hardly changed. 86% of the websites weighed roughly the same when loaded in the smallest window, compared to the largest one. In other words, despite the fact the websites *look* like an mdot site when loaded on a small screen, they are still downloading the full website content, and are thus painfully slow.

While every website is different, three causes for this over-downloading repeated across practically all websites.

- Download and Hide
- Download and Shrink
- Excess DOM

Download and Hide is by far the top reason for this bloat. Responsive websites usually return a single HTML to any client. Even on "Mobile First" websites, this HTML contains or references all that's needed to provide the richest experience on the biggest display. On a smaller screen, sections that shouldn't be shown are hidden using the `display:none` style rule.

Unfortunately, `display:none` doesn't help performance one bit, and resources referenced in a hidden part of the page are downloaded just the same. Scripts within hidden sections still run. DOM elements are still created. As a result, even when hiding the majority of your page's content, the browser will still evaluate the page in resources and download all the resources it can find.

Download and Shrink is a conceptually similar problem. RWD uses fluid images to better match the different screen sizes. While visually appealing, this means the desktop-grade image is downloaded every time, even when loaded on a much smaller screen. Users cannot appreciate the high quality image on the smaller screen, making the excess bytes a complete waste.

Excess DOM is the third episode of the same story. RWD websites return the same HTML to all clients. Browsers parse and process hidden areas of the DOM despite being hidden. As a result, loading a responsive website on a small screen results in a DOM that is far more complicated than what the user experience demands. A more complicated DOM leads to higher memory consumption, expensive reflows, and a generally slower website.

These problems are not simple to solve, since they're the result of how RWD and browsers work today. However, there are a few practices that can help you keep your performance under control:

- Use Responsive Images
- Build Mobile First
- Measure

Responsive Images are already discussed in this book at length, and help address the "Download and Shrink" problem. Since images are the bulk of the bytes on each page, this is the easiest way to significantly reduce your page's weight. Note that CSS images should be responsive as well, and can be replaced using media queries.

Build Mobile First means going a step beyond designing a Mobile First website, and actually coding a dedicated website for the lowest resolution you care about. Once implemented, this website should perform as well as other mdot sites, and be reasonably lightweight. From that point on, only enhance the page using JavaScript or CSS, to avoid over-downloading. Clients that have no JavaScript support will get your basic experience, which should be good enough for these edge cases. Note that enhancing with JavaScript and keeping performance high isn't simple, and best practices for it are not fully established yet—which brings me to my next point.

Measure. Treat performance as a core part of your website's quality, and don't ship without understanding and accepting its performance. If you know your mobile website weighs over 1 MB, you're likely to delay its launch until you do something about it. Measurement tools vary, but I would recommend Mobitest for testing on real devices (http://akamai.com/mobitest) and WebPageTest for testing on desktop browsers (www.webpagetest.org), resized them using the `setviewportsize` command.

In summary, Responsive Web Design is a powerful and forward thinking technique, but it also carries with it significant performance implications. Make sure you understand these challenges and design to avoid them, so that users won't abandon your website before they got to experience your amazing visuals and content.

Introducing matchMedia

In Chapter 3, “Media Queries,” the script we built to toggle the display of the navigation on small screens checked to see if the list items in the navigation were floated. If they were, the collapse feature was created. This time, let’s use the handy `matchMedia()` method.

The `matchMedia()` method is a native JavaScript method that lets you pass in a CSS media query and receive information about whether or not the media query is a match.

To be specific, the function returns a MediaQueryList object. That object has two properties: `matches` and `media`. The `matches` property returns either `true` (if the media query matches) or `false` (if it doesn’t). The `media` property returns the media query you just passed in. For example, the `media` property for `window.matchMedia("(min-width: 200px)")` would return `"(min-width: 200px)"`.

`matchMedia()` is supported natively by Chrome, Safari 5.1+, Firefox 9, Android 3+, and iOS5+. Paul Irish has created a handy polyfill for browsers that don’t support the method.

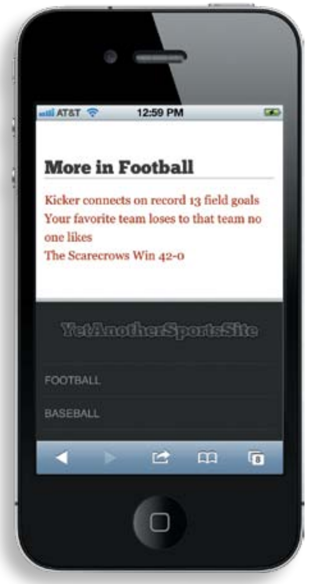
▶ **Note**
Irish’s polyfill is available on GitHub at https://github.com/paulirish/matchMedia.js or in the example files on the companion site at http://www.implementingresponsivedesign.com.

With the `matchMedia` polyfill in place, telling the browser to insert only the images above the first breakpoint simply requires wrapping the code inside a `matchMedia` check:

```
if (window.matchMedia("(min-width: 37.5em)").matches) {
    //load in the images
    var lazy = Utils.q('[data-src]');
    for (var i = 0; i < lazy.length; i++) {
        var source = lazy[i].getAttribute('data-src');
        //create the image
        var img = new Image();
        img.src = source;
        //insert it inside of the link
        lazy[i].insertBefore(img, lazy[i].firstChild);
    };
}
```

● ***Polyfill***
A snippet of code that provides support for a feature the browser does not yet support natively.

Now when the page is loaded on a phone, or the screen is sized down, the images are no longer requested (**Figure 4.3**). This is a big win for performance on small screens. There are now three fewer HTTP requests, and the size of the page has been reduced by about 60KB (the size of those three images combined). Best of all, the headlines are still there and the links are completely functional. The experience doesn’t suffer at all.



More in Football

Kicker connects on record 13 field goals
Your favorite team loses to that team no one likes
The Scarecrows Win 42-0

Figure 4.3 On small screens, the images in the "More in Football" section won't be requested, greatly improving the performance of the page.

With those images out of the way, we can focus on the lead-in image and the logo. We want those images, the logo in particular, to show up no matter the resolution. So, instead of conditionally loading them, we'll load them every time, but sized appropriately. This is where things get hairy.

Responsive image strategies

They say there are only seven stories in the world, they just get told in different ways. In the same way, there are currently only three strategies for handling responsive images: fighting the browser, resignation, or going to the server.

Fighting the browser

Most front-end solutions attempt to fight the browser. They try their best to switch which image is loaded before the browser can download the wrong one.

This is an increasingly difficult task. Browsers want pages to load quickly, so they go to extreme lengths to download images as quickly as possible. Of course, this is a good thing—you want your site to load as quickly as possible. It's really only annoying when you want to beat them to it.

Resignation

A few strategies out there basically admit defeat to the browser. Typically the approach is to load the small-screen image first, by default. Then, if necessary, load the larger image for larger screens as well.

Obviously this is not ideal. Larger screen devices will be making two requests where only one is needed. That's something to avoid if possible. Performance is important on large-screen devices, too.

Going to the server

Finally, a few methods use the server and some form of detection to determine which image to load. This method doesn't have to race the browser, because all the logic is executed before the browser ever sees the HTML.

However, going to the server is also not particularly future friendly. Maintaining information about every device that might request your content will become increasingly difficult as they proliferate (thanks to the decrease in the cost of manufacturing computing devices). Information about devices will also be less reliable as more and more devices allow content to be viewed in different ways: projections, embedded webviews, or on another screen entirely.

Responsive image options

There are limitations to every approach to responsive images currently available. To illustrate this, let's look at a couple different techniques for setting responsive images, and evaluate whether they're right for the *Yet Another Sports Site* page.

Sencha.io Src

Sencha.io Src is as close as you're going to get to a plug-and-play solution for responsive images. The service, originally created by James Pearce, takes an image that you pass and returns it resized. To use it, you simply preface your image source with the Sencha.io Src address like so:

```
http://src.sencha.io/http://mysite.com/images/football.jpg
```

Sencha.io Src uses the user agent string of the device making the request to figure out what size the device is and then resizes your image accordingly. By default, it resizes the image to 100% of the screen width (though it will never size up).

▶ **Note**
You can find detailed documentation for Sencha.io Src at http://docs.sencha.io/0.3.3/index.html#!/guide/src

A great deal of customization is possible. For instance, if you want the service to resize your image to a specific width, you can pass that along as another parameter. For example, the following line of code resizes the image to 320px wide:

```
http://src.sencha.io/320/http://mysite.com/images/football.jpg
```

Sencha.io Src is also smart enough to cache the requests, so the image isn't generated each and every time the page loads.

Unfortunately, this probably isn't the best solution for *Yet Another Sports Site.* Sizing the images to 100% of the screen size only helps on small screens. On a large display, when the article spans two columns, the image remains its original size because Sencha.io Src looks at the screen width, not the width of the containing element. While it's possible to tell Sencha.io Src to use that width, it involves using the service's experimental client-side measurements feature and doing a bit of JavaScript hackery.

While the current version of the page doesn't run into the issue, Sencha.io Src is also limiting if you want to do more than just resize an image, for instance, if you want to recrop the image. Perhaps the "More in Football" images could become square thumbnails at some point. If they did, a simple resize wouldn't work. Some art direction capability is needed, and Sencha.io Src doesn't allow for that.

You might also be a bit uncomfortable using a third-party solution for this. If the company changes its policy or goes out of business, you could very well be out in the cold and looking for another solution entirely.

Adaptive Images

Another solution bordering on plug-and-play is Adaptive Images, created by Matt Wilcox. It determines the screen size and then creates, and caches, a resized version of your image.

▶ **Note**
The code for Adaptive Images can be found at http://adaptive-images.com

It's an excellent solution for an existing site where you may not have time to restructure your markup or code. Getting it up and running is a simple three-step process:

1. Place the .htaccess and adaptive-images.php files that are included in the download into your root folder.
2. Create an ai-cache folder and grant it write permissions.
3. Add the following line of JavaScript to the head of your document:

   ```
   <script>document.cookie='resolution='+Math.max(screen.width,
   screen.height)+'; path=/';</script>
   ```

That line grabs the resolution of the screen and stores it in a cookie for future reference.

While there are many options you can configure in the adaptive-images.php, much of the time you'll be able to get away with just setting the `$resolutions` variable to include your breakpoints:

```
$resolutions = array(860, 600, 320); // the resolution breakpoints to
use (screen widths, in pixels)
```

If you're paying close attention, you'll notice that the breakpoints are slightly off from the CSS of the *Yet Another Sports Site* page. There's no 320px breakpoint in the CSS, and the highest two breakpoints, 1300px and 940px, are not included in the `$resolutions` array. This is because of the way the script works.

The smallest breakpoint, in this case 320px, is the size at which the image will be created for any screen that does not exceed that width. So, for example, a 300px screen will receive a 320px image because it's the lowest size defined in the `$resolutions` array. A 321px screen, since it exceeds the 320px value defined in the array, will receive the next size image—in this case, 600px. If we left 600px as our first breakpoint, any device with a screen size below 600px would have received a 600px image.

We also don't need the two highest breakpoints, because again, the script will try to resize an image to the breakpoint size. It will never size the image larger than it already is, so really, anything above 624px (the physical dimension of the image) doesn't matter much—the script won't resize the image.

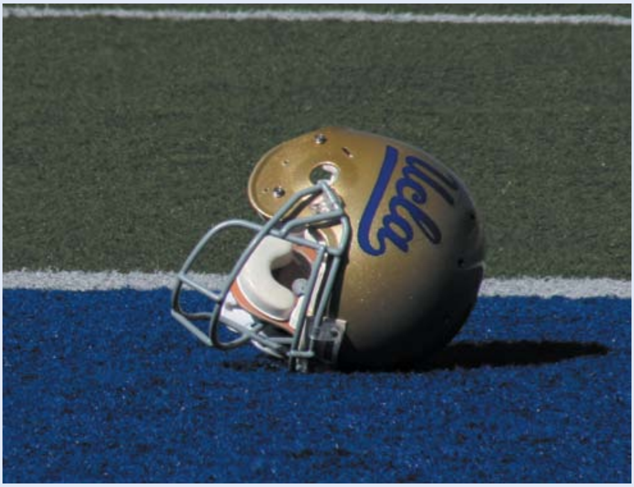
Once created, the images are stored in the ai-cache folder (you can change the name) so they don't have to be regenerated. There's also a configuration setting to control how long the browser should cache the image.

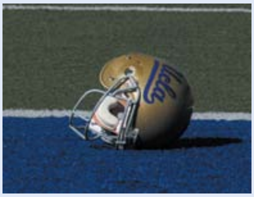
The installation is simple, and again, it's a great solution for existing sites that need to get something in place, but it's not without its faults. Unfortunately, no opportunity for art direction exists since the images are dynamically resized.

Art direction and responsive images

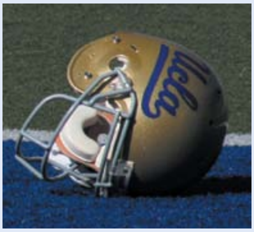
Much of the conversation about responsive images revolves around file size. While that's an important consideration, it isn't the only one. Sometimes, resizing an image for smaller screens can reduce its impact.

Consider this example photo of a football helmet.





The photo looks nice at its original size, and is well balanced. If we make the image smaller, suddenly the helmet is almost too small to be recognized.



This is an instance where art direction is necessary. Resizing the image alone causes it to lose its impact and recognizability. By tightening the crop instead, we keep the focus on the helmet despite the small image size.

The script also doesn't help you if the image is actually smaller at a large resolution. For the *Yet Another Sports Site* page, that's a problem. When the screen is above 1300px, the article goes to two columns and the image is placed inside one of them, reducing its size. Using the Adaptive Images script, the largest version of the image will still be downloaded.

● ***Content Delivery Network***
A collection of servers deployed in multiple locations to help deliver content more efficiently to users.

The other concern with this approach is that the URL stays the same, regardless of the size of the image being requested. This could cause issues with *Content Delivery Networks (CDNs)*. The first time a URL is requested, the CDN may cache it to improve the speed the next time that same resource is requested. If multiple requests for the same URL are made via the same CDN, the CDN may serve up the cached image, which may not be the size you actually want served.

What's ahead for responsive images?

Just to be clear: relying on a combination of server-side detection and JavaScript cookies is entirely a stopgap method. If there were something more permanent out there, I'd advocate it. Unfortunately every responsive image method available today is essentially a hack, a temporary solution to cover up the problem.

More long-term solutions, such as a new element, new attribute or new image format, have been discussed. In fact, if you're feeling a bit frisky, there's a fully functioning polyfill for one such as-yet non-existent element available on GitHub at https://github.com/scottjehl/picturefill. Unfortunately, the problem is far from being solved because the answer isn't as simple as "what is easy for developers to use."

In a blog post discussing the conflict of opinions between two popular proposed solutions, Jason Grigsby hit the problem on the head.[4] To improve performance, browsers want to be able to download images as soon as possible, before the layout of the page is known. Developers, on the other hand, rely on knowledge about the page layout to be able to determine which image to load. It's a difficult nut to crack.

I am confident that with time, a proper solution will emerge. In the meantime, as already mentioned, the best approach will vary depending on the project at hand.

[4] Read more about the real conflict behind <picture> and @srcset at http://blog.cloudfour.com/the-real-conflict-behind-picture-and-srcset/

Wait, what's the answer here?

Ultimately, no definitive solution currently exists for responsive images. Each method has advantages and disadvantages. The approach you take will depend on the project you're working on.

Of the two approaches we've discussed, settling on Adaptive Images is probably the best route to take since it doesn't require any reliance on a third-party source.

Background images

The folks over at *Yet Another Sports Site* are pretty happy with the site, but they'd like to see a visual indication in the header that helps visitors identify which section of the site they're in.

After 30 seconds of exhausting Photoshop work, we provide them with two silhouettes of footballs as shown in **Figure 4.4**.

They're happy with the way this looks on the large screen, but on anything smaller than the 53.75em breakpoint, where the logo starts to overlap, they'd like the background image to go away.

This is another area where building mobile up is helpful. Let's consider what would happen if we built the site desktop down using media queries.

Your base styles would be the place to include the background image and you would have to override it in a later media query. It would looked something like this:

```
/* base styles */
 header[role="banner"] .inner{
     background: url('../images/football_bg.png') bottom right no-repeat;
 }
....
@media all and (max-width: 53.75em) {
     header[role="banner"] .inner {
          background-image: none;
     }
}
```

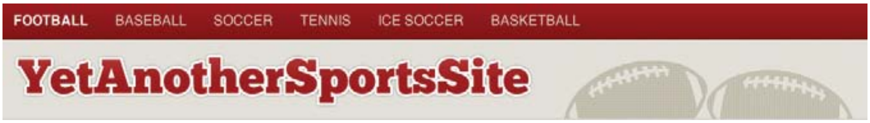


Figure 4.4 The header sporting its spiffy new background image.

On paper, this seems fine. But in reality, for many browsers, this would result in downloading the image even on a small screen device where it wouldn't be used. Most notable among these is the default browser on Android 2.x. Remember, while version 4 is current at the time of this writing, about 95 percent of Android devices are running an earlier version. This means that almost all Android traffic on mobile devices would be downloading the image without needing it.

To avoid this penalty, a better method would be to declare the background image within a media query like so:

```
/* base styles */
@media all and (min-width: 53.75em) {
     header[role="banner"] .inner{
          background: url('../images/football_bg.png') bottom right
          no-repeat;
     }
 }
 ....
 @media all and (max-width: 53.75em) {
     header[role="banner"] .inner {
          background-image: none;
     }
}
```

Doing that would be enough to get Android to play along nicely.

Note
If you want the juicy details about a variety of methods for replacing and hiding background images, take a look at http://timkadlec.com/2012/04/media-query-asset-downloading-results/ to see the tables of results from tests I've been running.

Since we built the page mobile up, the whole process is much simpler. The base experience doesn't need the background image, so we can introduce it in a media query later on:

```
/* base styles */
 @media all and (min-width: 53.75em) {
     header[role=banner] .inner{
          background: url('../images/football_bg.png') bottom right
no-repeat;
     }
}
```

Using this approach means that only browsers that need to display the background image will request it—performance problem solved!

Of course, once again, building mobile up means that Internet Explorer 8 and below won't see this background image by default. However, we already have an IE-specific stylesheet in place thanks to conditional comments. We can just add this declaration in there and we're good to go.

While we're at it

Currently, we're using web fonts to load the ChunkFive font that is being used in the header elements. The style declaration looks like this:

```
@font-face {
    font-family: 'ChunkFiveRegular';
    src: url('Chunkfive-webfont.eot');
    src: url('Chunkfive-webfont.eot?#iefix') format
    ('embedded-opentype'),
        url('Chunkfive-webfont.woff') format('woff'),
        url('Chunkfive-webfont.ttf') format('truetype'),
        url('Chunkfive-webfont.svg#ChunkFiveRegular') format('svg');
    font-weight: normal;
    font-style: normal;
}
```

▶ **Note**
Why all the different font files? You have browser differences to thank for that. While browser support is pretty good, they can't seem to agree on one format.

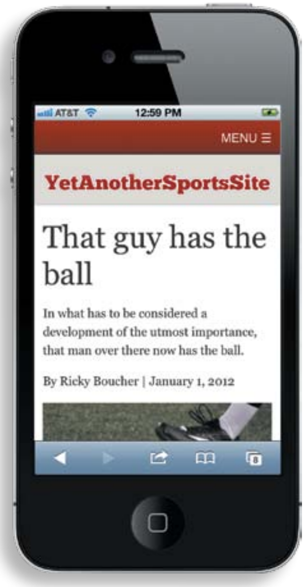
That declaration works nicely. The browser grabs the file it needs and renders the font. The sizes of the files aren't even that bad. There's a downside though. Currently, WebKit-based browsers won't display text styled with a web font until that web font has been downloaded. This means that if a user comes along on an Android, BlackBerry, or iPhone over a slow connection (or uses a laptop through a tethered connection for that matter), the header elements will take some time to actually display. This is a confusing experience for users and should be avoided.

We can't determine bandwidth (yet—check out Chapter 9, "Responsive Experiences," for a preview of what's to come), but we know the likelihood of a slow network is highest with a mobile device. It would be nice to save the user the trouble and load the fonts for larger screens only.

The approach we used to conditionally load background images works for fonts as well. So, let's move the `@font-face` declaration inside of a media query. Doing so ensures that devices below that breakpoint will not attempt to grab the fonts:

```
@media all and (min-width: 37.5em) {
    ...
    @font-face {
        font-family: 'ChunkFiveRegular';
        src: url('Chunkfive-webfont.eot');
        src: url('Chunkfive-webfont.eot?#iefix')
        format('embedded-opentype'),
           url('Chunkfive-webfont.woff') format('woff'),
            url('Chunkfive-webfont.ttf') format('truetype'),
            url('Chunkfive-webfont.svg#ChunkFiveRegular')
format('svg');
        font-weight: normal;
        font-style: normal;
    }
}
```

With that small tweak in place, the web font will load only on screens larger than 37.5em (~600px). While it's still possible for a user with a slow connection to get stuck with the WebKit delayed loading bug, by removing the fonts from small-screen displays we've also removed the most likely victim: people using mobile devices (**Figure 4.5**).



That guy has the ball

In what has to be considered a development of the utmost importance, that man over there now has the ball.

By Ricky Boucher | January 1, 2012

Figure 4.5
Web fonts will no longer be loaded on small screens to improve performance.

High-resolution displays

Just in case you thought swapping images out based on screen size wasn't difficult enough, it turns out there is at least one more situation that might require different images: high-resolution displays. The problem really started with the Retina display on the iPhone 4, but it's been exacerbated by the iPad 3 and the latest versions of the MacBook Pro both supporting a Retina display.

The Retina display sports a whopping 326ppi (pixels per inch) pixel density, compared to 163ppi for the iPhone 3 display. This high density means that images can appear to be incredibly detailed and sharp—if they're optimized for the display. If they're not, they will appear grainy and blurry.

● ***Pixel density*** The number of pixels within a specified space. For example, 326ppi means there are 326 pixels within every inch of a display.

Creating images for high-resolution displays means creating larger images, which in turn means larger file sizes. Therein lies the rub. You don't want to pass these larger images to screens that don't need them. Currently, there isn't a great way to do that with content images: it's the same sort of problem we discussed previously when trying to load images appropriate for different screen widths.

For CSS images, you can use the `min-resolution` media query for all browsers except those running on WebKit. For WebKit-based browsers, you must use the `-webkit-min-device-pixel-ratio` media query.

The `-webkit-min-device-pixel-ratio` media query takes a decimal value representing the pixel ratio. To target the Retina display on the iPhone, iPad, or new MacBook Pro you need a value of at least 2.

The `min-resolution` media query takes one of two values. The first is the screen resolution in either dots per inch or dots per centimeter. Doing this requires a little math, and some of the early implementations were inaccurate. As a result, I recommend using the new dots per pixel (dppx) unit. Not only does it remove the need for any math (it lines up perfectly with the ratio value accepted by the `-webkit-min-device-pixel-ratio` media query), but it also avoids the older, incorrect implementations. Support for the dots per pixel unit is still a little sketchy, but since displaying Retina-ready images is a nice enhancement rather than an essential feature, I'm pretty comfortable using it.

```
1.  header[role="banner"] .inner {
2.       background: url('../images/football_bg_lowres.png') bottom right
         no-repeat;
3.  }
4.  @media only screen and (-webkit-min-device-pixel-ratio: 2),
5.       only screen and (min-resolution: 2dppx) {
6.            header[role="banner"] .inner {
7.                 background: url('../images/football_bg_highres.png')
                   bottom right no-repeat;
8.            }
9.  }
```

The above media query targets any device with a pixel ratio of at least 2. Lines 1–3 set the background image for low resolutions. Lines 4 and 5 target devices with a pixel ratio of at least 2. If the pixel ratio is at least 2, then lines 8–10 apply a higher-resolution image for the background.

SVG

One solution for both high-resolution displays and images that scale across screen sizes is Scalable Vector Graphics (SVG). SVG images are vector images whose behavior is defined in XML. This means they can scale well without actually increasing file size. It also means they can be programmatically altered and adjusted.

One great example of how SVG can improve an experience is the work Yiibu, a mobile company in Edinburgh, did for the Royal Observatory at Greenwich. The company was working on a project that involved a responsive site featuring images of constellation patterns that needed to scale down. When using regular images and scaling, the small-screen images lost much of their detail. Using SVG and some smart scaling, Yiibu was able to adjust the images for small screens so the detail was retained (**Figure 4.6**).

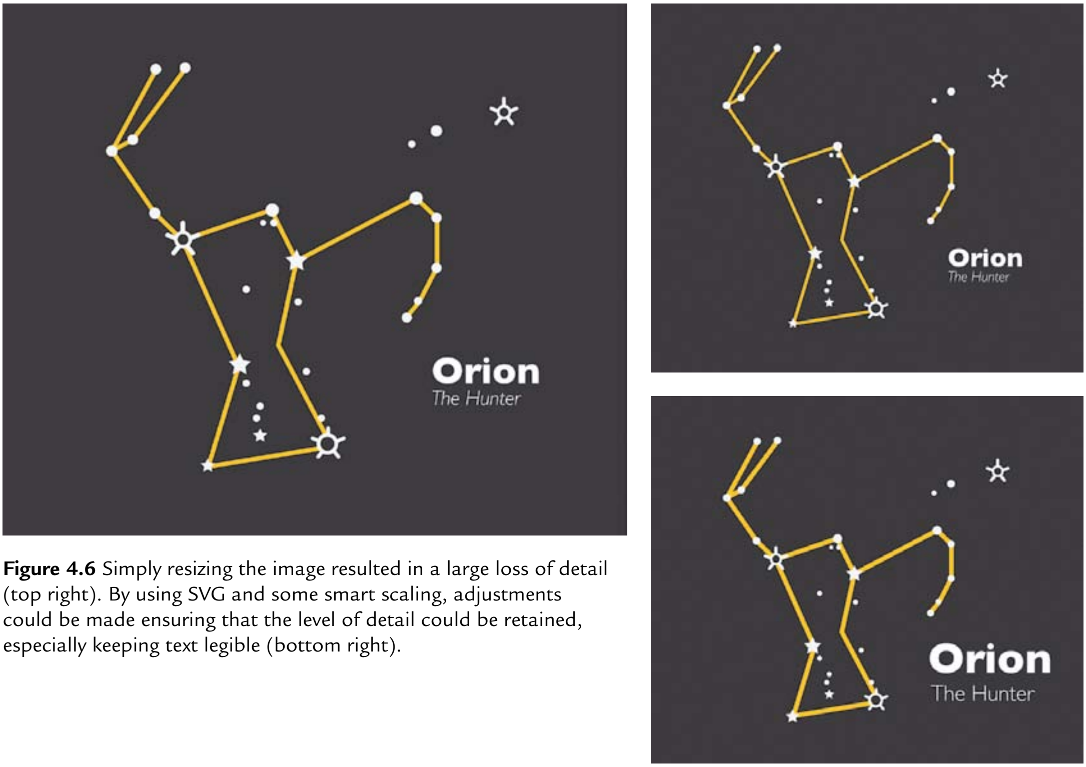


Figure 4.6 Simply resizing the image resulted in a large loss of detail (top right). By using SVG and some smart scaling, adjustments could be made ensuring that the level of detail could be retained, especially keeping text legible (bottom right).

There are two real issues standing in the way of SVG: browser support and lack of tools. As usual, Internet Explorer 8 and under don't play along. More importantly, neither does the default browser on Android 2.x—the most popular version of that platform. Those browsers that *do* support SVG images vary in their level, and quality, of support.

The most popular tools for image creation and manipulation, such as Photoshop, are not built with vector formats like SVG in mind. If you want to create SVG images, you need to find another tool to do it in.

As tools and browsers start to catch up, SVG images may become a very common tool in a web developer's toolbox.

Other fixed-width assets

Images aren't the only asset that present some problems for responsive sites. Let's look at two in particular: video and advertising.

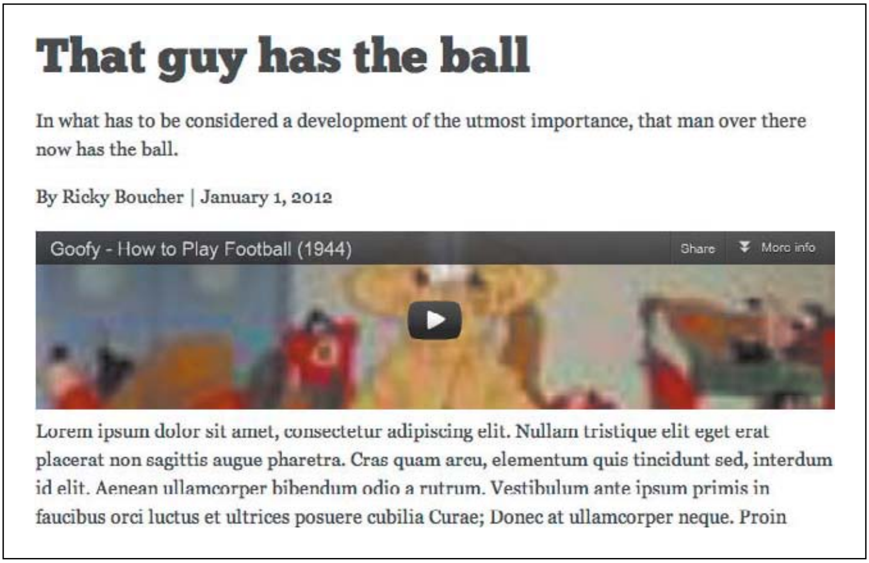
Video

Embedding videos in a responsive site is, perhaps surprisingly, a little more complicated than it first appears. If you're using HTML5 video, it's simple. You can use the same `max-width` technique we discussed for making images fluid:

```
video{
     max-width: 100%;
     height: auto;
}
```

Most sites, however, pull their videos from a third party (YouTube or Vimeo, for example) using an iFrame. If you apply the same trick, the width scales but the height retains its original value, breaking the aspect ratio (**Figure 4.7**).

Figure 4.7 Unfortunately, using `max-width: 100%` and `height: auto` on video embeds will result in the video breaking the aspect ratio.



That guy has the ball

In what has to be considered a development of the utmost importance, that man over there now has the ball.

By Ricky Boucher | January 1, 2012

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam tristique elit eget erat placerat non sagittis augue pharetra. Cras quam arcu, elementum quis tincidunt sed, interdum id elit. Aenean ullamcorper bibendum odio a rutrum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Donec at ullamcorper neque. Proin

The trick is something Thierry Koblentz called "intrinsic ratios."[5] The basic idea is that the box that contains the video should have the proper aspect ratio of the video (4:3, 16:9, and so on). Then, the video needs to fit the dimensions of the box. That way, when the width of the containing box changes, it maintains the aspect ratio and forces the video to adjust with it.

▶ **Note**
If you prefer, there's a helpful jQuery plug-in called FitVids that automates the process of making videos respond. Visit GitHub at https://github.com/davatron5000/FitVids.js to download it.

The first thing to do is create a wrapping element:

```
<div class="vid-wrapper">
     <iframe></iframe>
</div>
```

The wrapper serves as the containing box so it needs to maintain the proper aspect ratio. In this situation, the aspect ratio is 16:9. The video itself is positioned absolutely, so the wrapper needs an adequate amount of padding applied to maintain the ratio. To maintain the 16:9 ratio, divide 9 by 16, which gives you 56.25%.

```
1.    .vid-wrapper{
2.         width: 100%;
3.         position: relative;
4.         padding-bottom: 56.25%';
5.         height: 0;
6.    }
7.    .vid-wrapper iframe{
8.         position: absolute;
9.         top: 0;
10.        left: 0;
11.        width: 100%;
12.        height: 100%;
13.   }
```

The styles above also position the iFrame absolutely within the wrapper and set the height and width to 100% so it stretches to fill (lines 11–12). The wrapper itself is set to 100% of the article's width (line 2) so it adjusts as the screen size adjusts.

With these styles in place, the video responds to different screen sizes while maintaining its original aspect ratio.

[5] "Creating Intrinsic Ratios for Video" at www.alistapart.com/articles/creating-intrinsic-ratios-for-video/

ENHANCING THE EXPERIENCE

As always, it's worth taking a step back and considering how the experience can be enhanced. At the moment, the video is being downloaded on all devices. That might not be the best approach for the base experience. To speed up the core experience, it would be nice to display only a link to the video. Then, for larger screens, the video embed could be included.

To do this, start with a simple link:

```
<a id="video" href="http://www.youtube.com/watch?v=HwbE3bPvzr4">
Video highlights</a>
```

You can also add a few simple styles to make sure the text link doesn't look out of place:

```
.vid{
     display: block;
     padding: .3em;
     margin-bottom: 1em;
     background: url(../images/video.png) 5px center no-repeat #e3e0d9;
     padding-left: 35px;
     border: 1px solid rgb(175,175,175);
     color: #333;
}
```

There's nothing too fancy going on here. We gave the link a little padding and margin to set it apart from the rest of the content, and applied a background with a video icon set to the left (**Figure 4.8**).

Now, with JavaScript, convert the link to the appropriate embed.

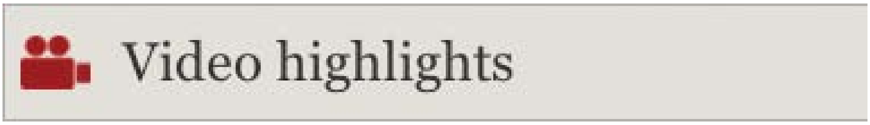


Figure 4.8 With some styles in place, the video link fits nicely in with the rest of the page.

Add the following function to the `Utils` object in yass.js:

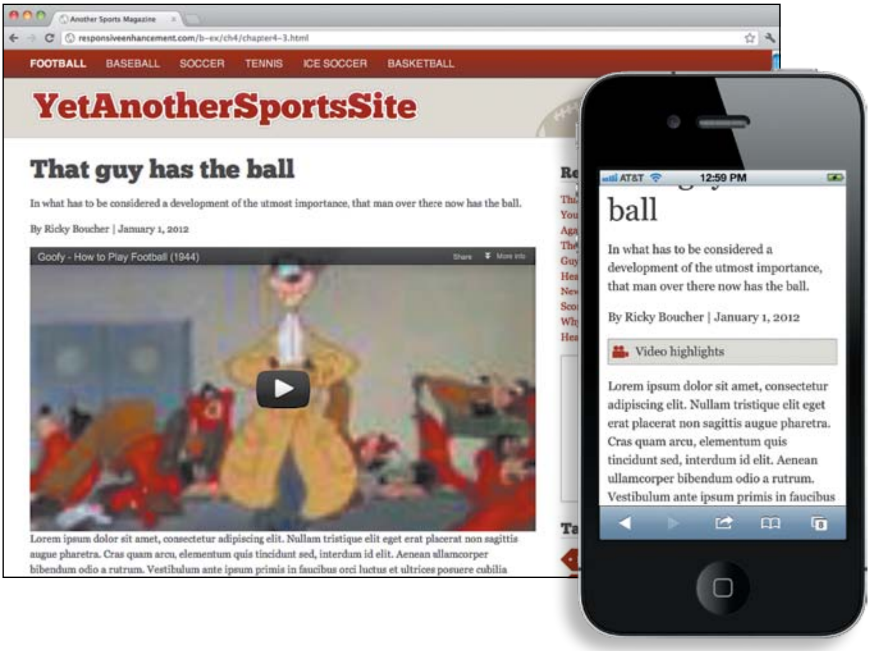
```
1.  getEmbed : function(url){
2.       var output = '';
3.       var youtubeUrl = url.match(/watch\?v=([a-zA-Z0-9\-_]+)/);
4.       var vimeoUrl = url.match(/^http:\/\/(www\.)?vimeo\.com\
         /(clip\:)?(\d+).*$/);
5.       if(youtubeUrl){
6.            output = '<div class="vid-wrapper"><iframe src="http://
              www.youtube.com/embed/'+youtubeUrl[1]+'?rel=0"
              frameborder="0" allowfullscreen></iframe></div>';
7.            return output;
8.       } else if(vimeoUrl){
9.            output =  '<div class="vid-wrapper"><iframe src="http://
              player.vimeo.com/video/'+vimeoUrl[3]+'" frameborder="0">
              </iframe></div>';
10.           return output;
11.      }
12. }
```

Let's walk through the function.

The function takes the URL of the video as its only parameter. It then determines if the URL is a YouTube video or a Vimeo video using regular expressions (lines 4–5). Depending on the URL type, it creates the embed markup including the containing element and returns it (lines 5–11).

Armed with the `getEmbed` function, it's easy to convert the video link to an embed. Throw the following JavaScript within the `matchMedia("(min-width: 37.5em)")` test:

```
//load in the video embed
var videoLink = document.getElementById('video');
if (videoLink) {
     var linkHref = videoLink.getAttribute('href');
     var result = Utils.getEmbed(linkHref);
     var parent = videoLink.parentNode;
     parent.innerHTML = result + videoLink.parentNode.innerHTML;
     parent.removeChild(document.getElementById('video'));
}
```



That guy has the ball

In what has to be considered a development of the utmost importance, that man over there now has the ball.

By Ricky Boucher | January 1, 2012

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam tristique elit eget erat placerat non sagittis augue pharetra. Cras quam arcu, elementum quis tincidunt sed, interdum id elit. Aenean ullamcorper bibendum odio a rutrum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia

Figure 4.9 On large screens (right) the video is embedded but small screens will see a link to the video instead.

The first two lines grab the link to the video and the link's `href`. On line 5, the link is passed to the `getEmbed` function we created. Once you have the result, lines 6–8 insert it into the article and remove the text link (**Figure 4.9**).

Now the video embed is responsive, and is pulled in only when the screen size is greater than 37.5em, ensuring that the base experience won't need to make the expensive HTTP requests to embed the video.

Advertising

Another fixed asset that presents some difficulties is advertising.

Like it or not, advertising is a key part of many businesses' revenue stream online. We won't get into a debate here about advertising-based revenue versus the pay-for-content model; that's a discussion that gets ugly quickly. The reality of the matter is that for many businesses, ad revenue is essential.

From a purely technical standpoint, advertising in a responsive layout isn't that difficult to implement. You could use JavaScript to conditionally load an ad unit based on the screen size. Rob Flaherty, a developer in New York City, demonstrated a basic method:[6]

```
1.   // Ad config
2.   var ads = {
3.        leaderboard: {
4.             width: 728,
5.             height: 90,
6.             breakpoint: false,
7.             url: '728x90.png'
8.        },
9.        rectangle: {
10.            width: 300,
11.            height: 250,
12.            breakpoint: 728,
13.            url: '300x250.png'
14.       },
15.       mobile: {
16.            width: 300,
17.            height: 50,
18.            breakpoint: 500 ,
19.            url: '300x50.png'
20.       }
21.  };
```

This configuration sets up three different ads (leaderboard, rectangle, and mobile). Each ad has a width (lines 4, 10, and 16), height (lines 5, 11, and 17), URL (lines 7, 13, and 19), and breakpoint at which point the ad should load (lines 6, 12, and 18). You could use the `matchMedia` function to determine which ad should be loaded based on the breakpoint.

Even better, the ad itself could be responsive. It could consist of HTML and CSS that allow it to adjust to different screen sizes. Going this route would eliminate the JavaScript dependency and potentially allow the ad to do some cool things by playing on its interactive nature.

[6] "Responsive Ad Demos" at www.ravelrumba.com/blog/responsive-ad-demos/

From a technical perspective, neither of these options is particularly difficult. The problem is that creating and displaying an ad has a lot of moving parts.

Most ads are served by third-party networks or the creative pieces are developed externally and then submitted according to the specifications of the site. At the moment, no major ad-serving networks accommodate varying ad sizes based on screen size.

Using an internal ad serving platform is a bit more flexible, but if the creative is developed outside your company, then you'll need to be willing to do some education. The people creating the ad materials may not be up to speed on what's going on.

More importantly, ads are currently sold much like they are in print: You pay based on the size and placement of the ad. So how exactly do you do that when the size and placement vary?

One solution is to sell ad groups instead of ads. For example, instead of selling a skyscraper ad, you sell a Premier Group ad (or whatever you want to call it). The Premier Group may consist of a skyscraper for screens above 900px wide, a boom box for screens above 600px but below 900px, and a small banner for screen sizes below that point.

Obviously, this won't be an easy transition. Creative teams, decision makers, and the salesforce all need to be educated on why this approach makes more sense than buying a defined ad space. It won't be an easy sell, but with time it should get easier.

The other consideration here is that some companies may want to target only a single form factor. Perhaps their service is something specific to mobile devices, and they decide they'd only like to serve their ads to those smaller screens. That of course throws a little wrinkle into the ad groups, as things start to get broken up.

Ultimately, I'd like to see the discussion of responsive advertising lead to fewer ad spots and a higher cost per ad. Sites whose revenues are ad-based frequently overload their pages with a plethora of ads. This makes the situation more difficult when trying to handle the small-screen experience. Do you hide all those ads, thereby limiting page views for your advertisers, or do you cram them all in there and ruin the experience for your visitors?

Instead of loading up pages with more and more ads, reduce the amount of ads on a page. Instead of ten ad slots at $1,000 per month, offer three at $4,000 each. Make the ad spaces something worth coveting. It benefits advertisers because they have fewer ads competing for attention, and it benefits users because they are greeted with a much better experience.

Unfortunately there's a chicken and the egg problem: advertising rates are currently a race to the bottom. Ads struggle to get quality click-through rates so the way to compete is to see how far you can lower the cost of entry. Someone has to be bold enough to make that first step.

Wrapping it up

Performance is an important consideration for any site. Loading images that are unnecessary or larger than needed can have a serious impact on page load time.

The CSS solution of `display:none` is not viable. It hides images from view, but they're still requested and downloaded. If you want images to show only above a certain breakpoint, the better bet is to load them conditionally, after the page load has occurred.

Responsive images are an unsolved problem. There have been many attempts at a solution but each has its own set of problems. The best thing you can do is take time before each project to consider which approach will work best for that site.

To hide background images without having to download them, include the image in a media query. Setting it in your base styles and then trying to hide it results in the image being downloaded in the majority of cases.

High-resolution displays, such as the Retina display on latest versions of the iPhone, iPad, and MacBook Pro, pose another challenge. There is a solution for CSS-based images, which can use the `min-resolution` media query.

Video and advertising are also concerns. For video, using the intrinsic ratio method can help you to scale the video appropriately across screen sizes. As always, be conscious of the performance. It may be best for users to simply link to the video on small screens and embed on larger ones.

For advertising, the technical challenges are not difficult to solve. If you're loading ads from your own system, JavaScript or some responsive HTML and CSS can help the ads change for different resolutions. The bigger problem arises in getting sales teams and third-party advertising networks to get on board.

This page intentionally left blank

Index