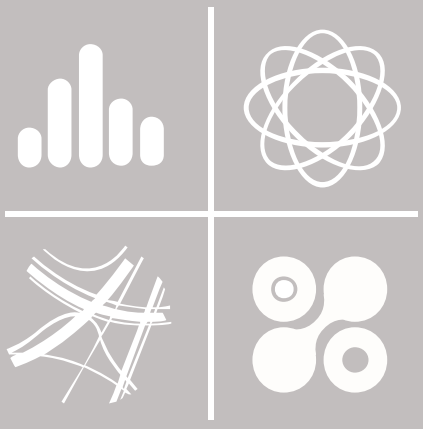


The Mobile Book

By Smashing Magazine

Published 2012 by Smashing Media GmbH, Freiburg, Germany.
Printed in EU.

Cover Design and Illustrations created by Mike Kus.
Proofreading: Andrew Lobo, Clarissa Peterson, Iris Lješnjanin. Author Bios: Cosima Mielke.
Editing and Quality Control: Vitaly Friedman, Iris Lješnjanin.
eBook Production: Talita Telma-Stöckle, Andrew Rogerson, Markus Seyfferth.
Marketing: Stephan Poppe.

Design and Layout: Ricardo Gimenes, Mike Kus, Melanie Lang, Markus Seyfferth.
Typefaces used: Elena by Nicole Dotin (Process Foundry), Whitney by Jonathan Hoefler and Tobias Frere-Jones, Skolar by David Březina (Type Together).

The Mobile Book was written by Peter-Paul Koch, Stephanie Rieger, Trent Walton, Brad Frost, Dave Olsen, Dennis Kardys, Josh Clark. The extra eBook chapters of the book were written by Greg Nudelman, Rian van der Merwe, Nathan Barry, Arturo Toledo, Sebastiaan de With, Remy Sharp.

Reviewers: Andreas Constantinou, Brad Frost, Bruce Lawson, Bryan Rieger, Dave Olsen, Dennis Bournique, Francisco Inchauste, Krijn Hoetmer, Lyza Danger Gardner, Peter Beverloo, Rian van der Merwe, Scott Jenson, Tom Giannattasio, Tim Kadlec, Trent Walton, Vasilis van Gemert.

Idea and concept: Vitaly Friedman, Sven Lennartz.

All links featured in this book can be found at smashed.by/links.

The Mobile Book: smashed.by/the-mobile-book.

TABLE OF CONTENTS

5 Foreword *by Jeremy Keith*

PART I—THE MOBILE LANDSCAPE

9 What's Going On In Mobile? *by Peter-Paul Koch*

55 The Future Of Mobile *by Stephanie Rieger*

PART II—RESPONSIVE WEB DESIGN

91 Responsive Design Strategy *by Trent Walton*

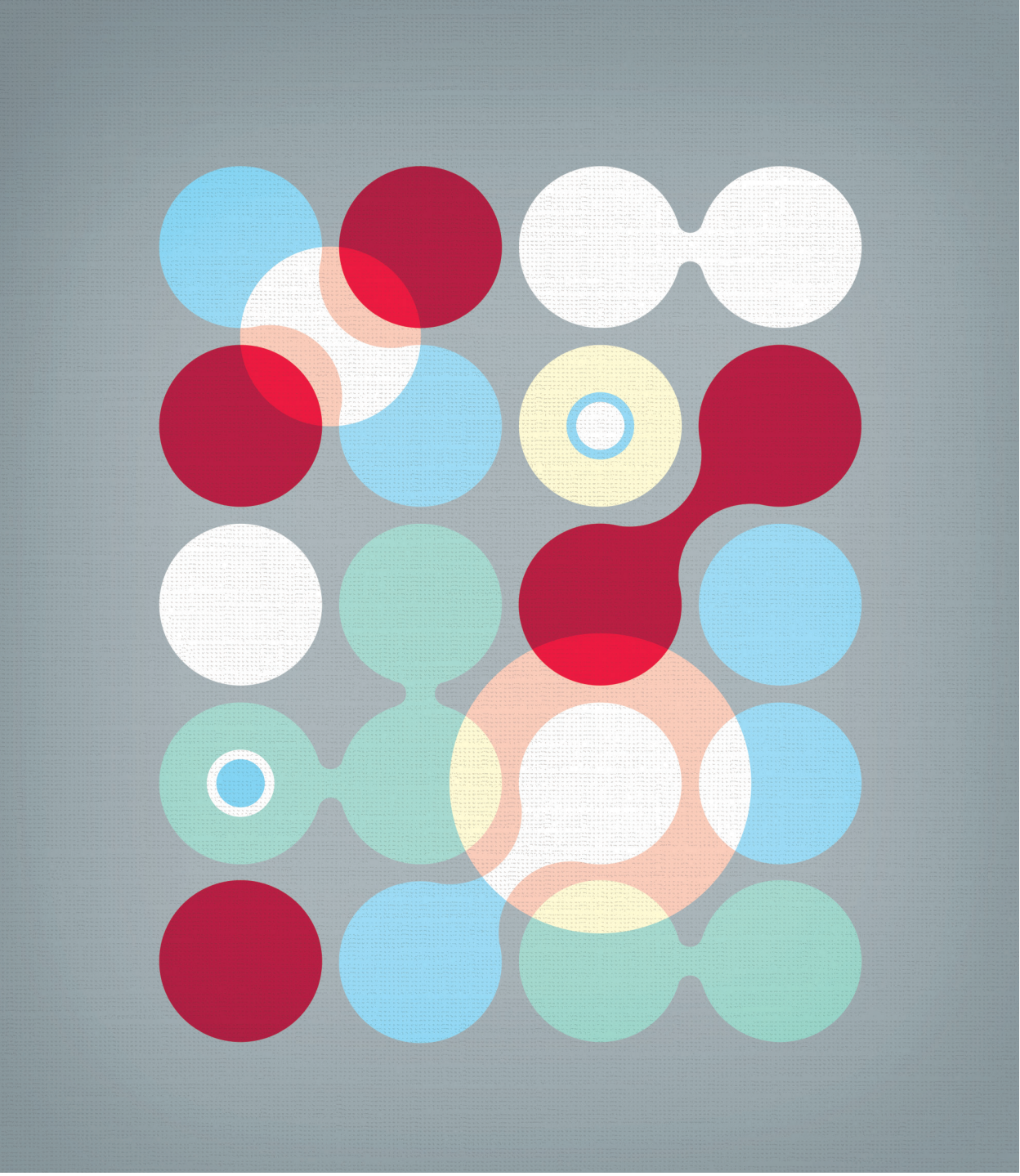
129 Responsive Design Patterns *by Brad Frost*

175 Optimizing For Mobile *by Dave Olson*

PART III—UX DESIGN FOR MOBILE

225 Hands-On Design For Mobile *by Dennis Kardys*

289 Designing For Touch *by Josh Clark*



Responsive Web Design

iii. Responsive Design Strategy
iv. Responsive Design Patterns
v. Optimization For Mobile

CHAPTER THREE · **BY TRENT WALTON**

RESPONSIVE DESIGN STRATEGY

I AM ALLERGIC TO CEDAR POLLEN. Here in the Texas Hill Country, cedar trees pollenate during the winter, triggering a cedar fever blight that makes over half the population miserable, cranky and desperate to know how bad pollen levels will be each day. Some days it's so bad that I'll wear a bandana or surgical mask over my face if I'm going to be outside for too long.

And with that, your perception of me as a rugged Texas mountain man is shattered. Alas, moving on…

During this season, the first thing I do is check the weather section of my local news website to know the pollen levels for the day. I've visited this website every winter day for three years until recently, when the news outlet launched device-specific versions of the website with a completely different information architecture and hierarchy from one device to the next. Even worse, it decided to pare down core content in the various versions from desktop to tablet to mobile. When I pulled up the website on my phone, I got a mobile version with 10 general headlines and no clear path to

the pollen levels, even after multiple guesses with the drill-down navigation. This "update" wasn't about optimization, and the overall strategy was ill-conceived. I've not been back to the website since.

Creating a new version of your website for each new device that hits the market is neither a responsible nor a sustainable practice. This is why a responsive approach has become the default for how I build for the Web. If I were advising a small community center or a mega-corporation on its Web presence, my advice would be the same: responsive Web design is the best, most sensible place to start for any project. In the case of that news outlet, it would have one version to maintain, one version to budget for and one version of consistent content for users. Thus, when new devices hit the market (like seven-inch tablets), it wouldn't have to segment its Web budget again to start from scratch on a whole new experience.

At its core, *responsive Web design* (RWD) is defined by three key components: flexible grids, flexible images and media queries. In the next three chapters, we will discover that these components are just the tip of the iceberg. And with the ever-increasing number of devices flooding the market, RWD is the most effective way to address them all at once. We will always be able to enhance or optimize further through techniques such as app caching and browser detection, but RWD is definitely the best place to start.

So, where should we start? How about with a confession?

From Pixels to Proportions

As much as I believed RWD was the brilliant future of our craft, every cell in my Web designer body screamed out in terror when I first saw it in action.

I wanted to cower under my desk after reading Ethan Marcotte's visionary A List Apart article "Responsive Web Design"[1] in

1. A List Apart, "Responsive Web Design," smashed.by/rwd.

May of 2010. Ethan had coined the term, connecting the dots from flexible grids to flexible images to media queries. He even built us a demo showing how they work together magically, reflowing the page into any sized viewport. What wasn't so quick and easy, however, was my acceptance of this new reality.

My fear was rooted in how I perceived and understood the Web. Maybe my working style was a lot like yours: my team and I would work on content and design based on the understanding of a single fixed-width layout, usually designed in Photoshop. Designing in Photoshop was where we could exercise complete control, and I took solace in that. What we saw in the PSD was, for the most part, exactly what we'd execute with the front-end code. In this way, pixel perfection was a point of pride for us, and the idea that what we'd be designing would flex, resize and rearrange itself based on the viewport's size was terrifying. How could we ensure quality? How could we retain control?

But that level of control has always been a myth. We all experience this everyday as we work to accommodate varying browser support, font rendering, monitor sizes, screen color set-ups and connection speeds. Moreover, now that users are visiting from pocket-sized to television-sized screens, it's time to accept that Web design is about so much more than creating desktop-friendly layouts. Ultimately, I realized that we must trade the control we thought we had in Photoshop for a new kind of control—using flexible grids, fluid images and media queries to build not a page, but a *network of content* that can be rearranged at any screen size to best convey the message.

RESPONSIVE WEB DESIGN ISN'T BOLT-ON

I've found that adapting approaches, workflows and strategies to get the most out of RWD doesn't typically happen overnight.

Unlike easy progressive enhancements for rounded corners (`border-radius`) and Web fonts, RWD can't be quickly added, tested and then removed from an existing website. Consider how easy it would be to open a website that you built a year ago and add `border-radius` to a button.

RWD isn't bolt-on in the same way. It's not something that can be easily added to a nearly finished website because its flexible foundation requires us to rethink the very units of measure that we build on, from pixels to proportions—things such as `ems` (based on the current font size), `rems` (based on the root font size) and percentages.

All the things we've become accustomed to designing in certain ways (such as calendars, checkout flows, tables and navigation patterns) can still exist, but they must be reconsidered to work optimally in a responsive setting. But before we get too far ahead of ourselves, let's gain a fresh perspective by briefly looking at each of RWD's core components.

Flexible Grids

If there's anything I've had to learn the hard way, it's that this core component of RWD—*flexible grids*—extends beyond grid structure to type, vertical spacing, positioning and even decorative items such as `border-radius` and `text-shadow`. Establishing this flexible foundation, regardless of whether you plan to add media queries, is a great way to help future-proof a website, preparing it for proportional scaling.

Think about it this way: take a button with a font size of `1 em` and 20 pixels of padding. Then, use a media query to increase the font size to 1.5 `em` at widths greater than 600 pixels. You will have to declare a larger pixel value for the padding if you want the button to grow proportionally. If you had declared 1.250 `em` instead of 20 pixels

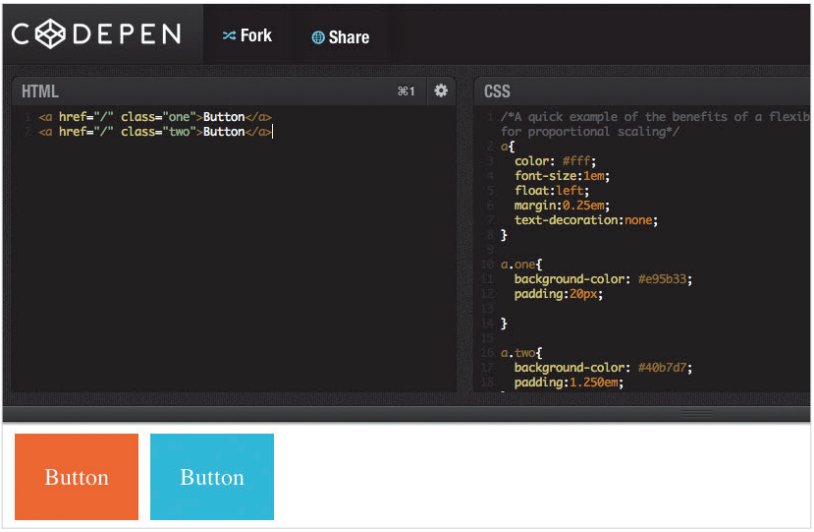


FIGURE 3.1. Basic setup for two buttons.

for padding, then that proportional growth would have happened automatically with less code.[2]

Because I used so many pixel values on the first responsive website that I built, I had to go through each media query and update just about every property to adjust those pixel values to suit. The style sheet just looked wrong. After establishing that flexible foundation, I was able to go back and simply change the body's font size and watch all of the `em`-unit-sized pieces of the website scale proportionally. This process facilitated a two-thirds reduction in code, all thanks to our dear friend, the flexible foundation. Think twice about using pixel values in your CSS.

Now, on to the grid itself. I'll assume you've already chosen your favorite calculator and have mastered Ethan's handy-dandy formula:

2. A demo: smashed.by/grids-demo.

```
target ÷ context = result
```

If you need to find the proportional amount for four equally sized 230-pixel columns within a 1000-pixel container, you'd plug in the following:

```
230px ÷ 1000px = 0.23 or 23%
```

Great! So, we'll set each of the four columns at 23%. Next up, let's say we have 10-pixel margins on each side.

```
10px ÷ 1000px = 0.01 or 1%
```

We'll plug in the 1% margin and watch it flex—all too easy. But what happens if the container (i.e. the context) is 1080 pixels?

```
230px ÷ 1080px = 0.21296296296296
```

OK, I know what you're thinking, but leave those numbers intact. Resist the urge to trim, as unsightly as they may be. If your calculator gives you 0.21296296296296, don't shorten the percentage to 21.3%. We want the truest proportional representation possible. Plus, computers are smarter at math than you are, and they know it, so don't do them any favors.

Note: Aside from setting margins to separate grid columns, you could also take advantage of `box-sizing: border-box`. This allows for the addition of padding and border *without* expanding the width of the box.[3]

OK, grids? Flexing. Next, let's tackle images.

3. Irish, Paul. "{ box-sizing: border-box } FTW," smashed.by/box-sizing-ftw.

Flexible Images (and Media)

Implementing flexible images is a breeze.

```
img {
     width: 100%;
}
```

Here, images are set to fill the width of their containers. But what happens if the image's actual size is much smaller than the container itself and we are worried about it becoming horribly pixelated? This is where `max-width` becomes incredibly useful.

```
img {
     max-width: 100%;
}
```

With `max-width`, image sizes still fill the width of their containers, but they won't render larger than their actual size. In other words, a 400-pixel image will scale perfectly to fill a 300-pixel container but will not suffer a resolution blowout if it continues to scale to a 500-pixel container, stopping at 400 pixels.

PROTECTING IMAGE ASPECT RATIOS

What happens to an image whose width and height attributes are defined in the HTML when `max-width`: 100% is applied to them? As Bruce Lawson reminds us in his post on preserving image aspect ratio,[4] we must remember to set `height`: `auto` to ensure that images aren't horrendously squeezed in only one direction at a time. That way we'll be able to override that static height and enable all images to flex proportionally in all directions.

4. Lawson, Bruce. "Responsive Web Design: preserving images' aspect ratio," smashed.by/asp-ratio.

EXTENDING THINGS TO MEDIA

The convention is to extend this to other forms of media elements by adding to our code to cover design elements such as video and Flash objects.

```
img, embed, object, video {
        max-width: 100%;
}
```

Great! All our bases are covered. With fluid images being so easy to implement, the rest must be just as easy, right? You'd think so, but gaining fluid-width video embedding isn't always as automagical as we'd like. Thankfully, Dave Rupert and Chris Coyier have built a solution in the form of a jQuery plugin named FitVids.[5]

FitVids automates the intrinsic ratio method created by Thierry Koblentz[6] to achieve fluid-width videos in a RWD. Essentially, Thierry suggests adding a "magic wrapper" (the parent) around the container for the media element (the child)—a container that proportionally resizes itself according to the width of its parent. This means creating a box with the proper ratio (4:3, 16:9, etc.) and then making the video inside that box stretch to fit the dimensions of the box. With this technique,

FIGURE 3.2. Fitvids.js let you easily embed videos with flexible width.

5. FitVids.js, smashed.by/fitvids.

6. Koblentz, Thierry. "Creating Intrinsic Ratios for Video," A List Apart, smashed.by/intrinsic-videos.

browsers are able to determine a video's dimensions based on the width of its containing block. With intrinsic dimensions, a new width will trigger a new calculation of height, allowing videos to resize and scale the way images do.

Here is the markup:

```
<div class="wrapper-with-intrinsic-ratio">
   <div class="element-to-stretch"></div>
</div>
```

And here is the CSS magic:

```
.wrapper-with-intrinsic-ratio {
                    position: relative;
                    padding-top: 25px;
                    padding-bottom: 56.25%;
                    height: 0;
}
.element-to-stretch {
                    position: absolute;
                    top: 0;
                    left: 0;
                    width: 100%;
                    height: 100%;
                    background: teal;
}
```

To create a box with an intrinsic ratio, all we need to do is apply `top` or `bottom` padding to a `div`. Whatever the width of the viewport, the teal box should keep its aspect ratio (in this case, 16:9 = 0.5625, which corresponds to 56.25%). Ordinarily, you would also need to

consider the height of the chrome and player controls (in this case, 25 pixels for the YouTube chrome).

ABOUT IMAGE OPTIMIZATION

Later chapters will dig into various responsive image techniques and proposals, but be sure that the images you start with are optimized as much as possible. ImageOptim[7] has become an indispensable part of my workflow. It strips out unnecessary comments and color profiles from PNG, JPG and GIF images. Smush.it[8] is a similar tool, but with a Web uploader interface.

Another tip is to edit the actual images themselves. Jeremy Keith recently wrote a post[9] about his optimization of the 2012 dConstruct website, for which he took the speakers' photos and blurred the outer edges with a photo editor. Anything that wasn't the focal point (i.e. the speaker's face) was blurred. This reduced the number of jagged artifacts in the raster images, thus reducing overall file size by up to 85%. And because we tend to focus on human faces, the technique isn't really noticeable. Brilliant!

MOVING TOWARDS RESOLUTION INDEPENDENCE

As easy as it was to gain fluid-width images, reevaluating how we're using those images will be important. Let me pose a few problems that could arise.

First, using images instead of CSS to create buttons with effects such as borders, shadows and gradients makes it harder to resize and reshape the buttons. The need for this ability increases in a responsive setting because size and layout change from one viewport to another. Embedding text in images (such as JPEGs for advertisements) is always risky, but the negative effects are compounded on fluid-width

7. ImageOptim, smashed.by/imgopt.
8. Smush.it, smashed.by/smush-it.
9. Keith, Jeremy. "dContruct optimisation," smashed.by/dconstruct.

pages. The font size of text embedded in an image will scale with the container's size. Without proper planning and care, the text could become illegibly small or blurry as the image resizes.

With the introduction of high-pixel-density displays (such as Apple's Retina), the more we can use resolution-independent techniques, the more future-proof our websites will be.

Consider icons. If we're using CSS sprites, we could create multiple versions of a sprite and load an actual-sized or a double-sized version to ensure crisp rendering; but creating multiple-sized assets isn't as future-proof as seeking true resolution-independence through either SVG sprites or icon fonts.

I've recently taken to using icon fonts whenever possible. They're the most easily scalable solution because we're using only `font-size` instead of `background-position` and `background-size`, which are necessary for sprites. We can change the color on the fly (although fonts are limited to a single color), and they'll look crisp regardless of the pixel density of the display.

Implementing icon fonts is relatively easy. Use a service such as Shifticons[10] or Pictos[11] or roll your own. Chris Coyier has a nice write-up on icon font usage[12] that he's been updating regularly as best practices are hashed out. Chris maps icons to Unicode symbols and includes the `aria-hidden` attribute as an accessibility upgrade. Here's what an implementation might look like if we wanted to add an icon before a link to the weather section of a website:

```
<a href="/"><i aria-hidden="true" data-icon="&#x2601;"></
i><span class="screen-reader-text">Weather</span></a>
```

10. Shifticons, smashed.by/shifticons.
11. Pictos, smashed.by/pictos.
12. Coyier, Chris. "HTML for Icon Font Usage," smashed.by/font-usage.

And the CSS would be this:

```
@font-face {
   font-family: 'Icon-Font-Name';
   src: url('icon-font.eot');
   src: local('☺'),
   url('icon-font.woff') format('woff'),
   url('icon-fontweb.ttf') format('truetype'),
   url('icon-font.svg#webfontIyfZbseF') format('svg');
}

[data-icon]:before {
   font-family: 'Icon-Font-Name';
   content: attr(data-icon);
}
```

One other icon font service I'm giddy about is Symbolset.[13] In addition to using Unicode and CSS classes, Symbolset also lets designers apply icons to plain-text keywords, replacing them with matching icons. For example, one could replace the word "cart" with an actual shopping cart icon. This method is particularly helpful when an icon will appear without accompanying text, which is fairly common on responsive websites that constrict to a narrow layout and on which space is at a premium.

Media Queries

Sooner or later, fluid-layout break columns will become too cramped, navigation items will wrap in an unsightly manner, and paragraphs will become difficult to read. We can use media queries to repair broken layouts by having them target device or browser

13. Symbolset, smashed.by/symset.

properties. Media queries have two parts: the media type and the query. Let's look at an example.

```
@media screen and (min-width: 1000px) {
    body {
        font-size: 125%;
    }
}
```

The `screen` type is the one we are most familiar with. But we could also target other output media, such as `print`, `projection` and `tv`. Their support is very limited (so far), but keep them in mind nevertheless. For a complete current list, refer to the W3C's CSS2.1 specification.[14]

The `min-width` is the query itself. Width- and height-based queries are common, as are `device-width` and `device-height`. For a complete current list of queries and information on whether they support minimum and maximum values, refer to the W3C's media queries specification.[15]

With media queries, not only are we able to repair layouts when they break at a given view, but we can harness their power to ensure ideal experiences across the ever-increasing array of device types and screen sizes.

Take a basic two-column layout. When the flexible grid scales down to a narrower view—let's say 600 pixels wide—the columns would get too cramped. We could use media queries to stack the columns on top of each other.[16]

14. CSS 2.1: Media Types, "Recognized media types," smashed.by/media-types.
15. CSS3: "Media Queries," smashed.by/media-queries.
16. Demo: smashed.by/stack-demo.

FIGURE 3.3. Maintaining an ideal experience for users on small devices.

HTML:

```
<div id="container">
   <div class="row">
      <div class="col one">
         <p>Nullam quis risus eget urna mollis ornare
vel eu leo. Donec id elit non mi porta gravida at eget
metus. Curabitur blandit tempus porttitor. </p>
      </div>
      <div class="col two">
         <p>ullam quis risus eget urna mollis ornare vel
```

```
eu leo. Donec id elit non mi porta gravida at eget me-
tus. Curabitur blandit tempus porttitor. </p>
      </div>
   </div>
</div>
```

CSS:

```
.col {
    float: left;
    margin: 2%;
    width: 46%;
}

@media screen and (max-width: 600px) {
    .col {
        float: none;
        margin: 0;
        width: 100%;
   }
}
```

Success! With this example, we've altered the architecture of the page, maintaining an ideal experience for users on small devices and narrow viewports.

ESTABLISHING BREAKPOINTS

Most importantly, set media query breakpoints based on the content and not any particular device's screen size. While 480 and 768 pixel breakpoints are common today, they aren't universal, and they become less and less ubiquitous with each new tablet or smartphone that hits the market. Simply add a breakpoint when

the layout breaks. Use media queries to prevent navigation items from wrapping awkwardly, columns from cramping and text from becoming difficult to read at any given viewport size.

With all of these possibilities and problems to solve, managing media queries can quickly become unwieldy. Because of this, I try to consolidate media queries around a few key breakpoints. For example, If I realize that I added breakpoints to accommodate content at 650 pixels and again at 670 pixels, then I might seek to unify those changes under just one. Sure, we ought to be willing to extend things beyond that, but consolidating will help everyone—from copywriters to designers to developers—to maintain and communicate about the website.

As you can imagine, the more media queries and breakpoints you add, the more difficult keeping track of them will be. Even when a flexible foundation is firmly in place, the CSS code can pile up. While there are infinite ways to wrangle things, below are a few that have worked for me.

Top Down

Regardless of what view I design first, I always structure my CSS for mobile first by coding for the narrowest view first, and then adding in `min-width` media queries to build towards a full-width desktop view. For this, the simplest method, everything should live on *one* style sheet.

Another benefit to ordering things in this way is that it reinforces a mobile-first strategy any time the website undergoes maintenance or gets a new feature. This limitation of space can serve as an editor of sorts, continually honing a website's objectives and message.

Module-Based Organization

For large websites, having everything in one file can make it difficult to navigate the code as well as to keep your head wrapped

around which media queries affect the layout at which times. In situations such as these, I'll first establish CSS for the basic layout and typography. Then, I'll import style sheets for each part of the page that I plan to modularize. For example, a news website could have sheets like `headlines.css`, `weather.css`, `newsfeed.css`, `menu.css` and `ads.css`.

Breakpoint-Based Organization

Another option is to import style sheets for each key breakpoint. This can help designers mentally separate key shifts in the layout that occur at various screen sizes. It would look something like `style.css`, `540.css`, `720.css`, `1000.css` and so on. I prefer starting with (and importing) narrow one-column views as the base and building up from there. Of course, when a website goes into production, it's not uncommon for all of the files to be merged into one and minified.

Media Query Splitting

Another approach, outlined by Simurai,[17] is to set some breakpoints 1 pixel apart and to split the styles instead of overriding from one media query to the next. This would alleviate the need to reset something like `box-shadow: 0 1px 3px #333` to `box-shadow: none` when one is set in an alternate media query. To my mind, this technique is probably most useful in conjunction with a mobile-first, minimum-width approach, but it would surely ease some of the mental strain from tracking media queries during development.

There is an infinite number of ways to go about organizing media queries, so don't feel like you need to lock into a single one. To complement a good system of organization, there are some useful tools to keep track of which media queries are triggered and when:

17. "Media Query splitting," smashed.by/mq-splitting.

- Responsive.is (smashed.by/inbrowser)
  Resizes a website in-browser to typical preset device sizes and orientations.
- Responsivepx (smashed.by/respx)
  Remy Sharp's tool helps you pinpoint pixel values for when a responsive layout breaks.
- Aptus (smashed.by/aptus)
  A Mac app that acts as a dedicated browser for testing a website at any size. My favorite part is that it stores custom breakpoints so that you can quickly preview and diagnose layout issues in one click.

VERTICAL MEDIA QUERIES

If you're like me, most of your media query attention centers on width. But what can be done with height-based media queries? I've recently found a few uses.

One of my favorite things to do with media queries is to increase the font size in larger views. One problem I'd run into, however, was that I didn't want short but wide screens to trigger the largest of font sizes, because then everything would just look awkward. I used a `min-height` media query to solve that:

```
@media screen and (min-width: 1000px)
              and (min-height: 700px) {
   body {
       font-size: 137.5%;
   }
}
```

With this, only views wider than 1000 pixels and taller than 700 pixels would get the extra-large type. Another helpful use of this I've found is to manage the fold. I'm not trying to incite a debate on

whether or not the fold exists, but let's suppose you'd like to ensure that certain important pieces of content are visible upon initial loading without any scrolling. You could use vertical media queries to reduce vertical padding or margins or even to crop images on the page.

RELATIVE UNITS FOR MEDIA QUERIES

Up to this point, we've been using pixel values for our media queries. I do this regularly when building out a website, but recently I've been completing the pixel-value purge and using `em` units for media queries as well. As Lyza Gardner of Cloud Four points out,[18] this can help facilitate proportional scaling, which is especially handy when users zoom in the browser. To do this, we'd simply calculate the `em`-unit value based on the baseline body font size, which is 100% or `1 em`.

Suppose we start from a baseline of 16 pixels, and we have the media query `@media all` and `(min-width: 400px)` in our CSS. To calculate the appropriate `em` values, we would just calculate 400 pixels ÷ 16 pixels = 25, resulting in `@media all` and `(min-width: 25em)`. The difference is that if users zoom in (i.e. make the text larger in their browser), then they would no longer be using the 16 pixels baseline that we used for pixel-based media queries. With the `em`-based media query, we're better equipped for the whatever baseline they define in their browser.

My favorite tool for quick calculations is PXtoEM.[19]

MEDIA QUERY SUPPORT

Browser support for media queries is great until you get to IE 8 and below. But never fear. We've got a few options for handling this, which I choose from depending on a project's scope and requirements.

18. Gardner, Lyza. "The EMs have it: Proportional Media Queries FTW!," Cloud Four, smashed.by/queries-ftw.
19. PXtoEM, smashed.by/pxtoem.

Scott Jehl's Respond.js[20] was built as part of the Filament Group's work for the Boston Globe. The group has kindly placed it on GitHub as a polyfill for `min-width` and `max-width` media queries.

Another option is to structure CSS mobile first, using `min-width` media queries to re-architect the layout from single-column narrow views up to the full-width desktop. IE 6 through 8 will get that mobile single-column narrow view, but with your amazing design chops, I'm sure that's nothing to frown about.

A third option is to build fixed-width layouts for older browsers, with targeted LT (i.e. less than) IE 9 style sheets that mimic the layout at full width (or some arbitrary container width, such as 960 pixels). I like this option because it helps separate the layouts for older browsers from all of the progressive enhancement we do for the main version. Because so many of the newer CSS properties we enjoy are supported only by browsers that *also* support media queries, we can push things further with the main style sheet.

Getting to Work

We've broken down each of the three core responsive components and looked at tips for each of them. Now that the dots have been connected, let's dive into examining how teams can begin to work together to build their next website responsively.

DECOMPARTMENTALIZE THE WORKFLOW

In the production model that I was used to, a project was passed down the assembly line from content to design to development to launch. While it might not have been the best approach, it worked because each step was a prerequisite for the next. The content would be set; the designers would lay things out; and then the developers would code according to the image mockups.

20. Respond.js, smashed.by/github-respond.

With RWD, we're designing that aforementioned network of content, rather than a page, so I now favor an approach with rapid iteration and a lot of show and tell. This requires all team members to work even harder to foster healthy collaborative relationships.

If you're a designer but don't yet know how to code, I'd recommend learning how to use basic browser development tools, such as WebKit's Web inspector. Requesting bumps in pixel values in the margin or padding will mean less to those who code in percentages and `em` units.

If you're a developer working with designers, make sure they get easy access to view the website while in production. With RWD, screenshots won't fully convey the progress that's been made or make it easy for collaborators to understand where things need to be improved.

RWD will further blur the line between the Web designer and developer's roles, so be willing to expand your skill set by taking advantage of these collaborative working situations.

BUILDING THAT NETWORK OF CONTENT

Now that your workflow is in order and everybody loves each other, what's next? One of the biggest challenges for a team new to RWD (and we're all new) is discussing the content and hierarchy for a layout that has become a moving target. It's easy to revert to thinking about content arrayed across a full-width desktop view and to forget that space will come at a premium in alternate views. Because of this, I like to start with low-fidelity mockups. These could be paper sketches, gray-box wireframes, anything you want. The trick is to keep them simple, because at this point it's less about how things look and more about providing visuals to generate a discussion about how content should fit in the space at any given width and how to maintain the hierarchy.

There's no time like the present to abandon static ways of thinking about how layout and content interact. Test ideas by quickly sketching alternate views, trying to poke holes in the plan. After all, what might seem like a great idea for a full-width layout might not make sense at narrower views. This will also help the team pinpoint where the layout might run out of space sooner than later. Perhaps the content needs to be edited down. Maybe the game plan needs to be rethought. This is one of my favorite parts of the process. It's laid back and collaborative, so invention is bound to happen.

DESIGNING RESPONSIVELY

Once the team has all of its best guesses for Plan A down on paper, it's time to start designing. Similar to the hierarchy sketching stage, picking any view and designing it is OK. I like to start with the full-width desktop view and then, before getting too far, gather the team for a review. As mentioned earlier, I still favor a mobile-first approach from a content and coding perspective; I've just found that my team likes to start at full size during the design phase. If we find that we're designing ourselves into a responsive corner, we'll quickly sketch out how things could shift and reflow at various widths. It's also not uncommon for coding to begin at this point. If there's something we're unsure we'll be able to develop, then we take to websites such as JS Bin[21] and CodePen[22] to experiment with potential solutions.

For some designers, Photoshop may not be necessary for anything other than asset creation (logos, textures, etc.). Designing in the browser is both a viable and a smart option, especially in a responsive scenario. I've recently noticed that we rarely wait until Photoshop comps are completed before getting into the code. Because any one comp can capture only a sliver of the vast array of

21. JS Bin, smashed.by/jsbin.
22. CodePen, smashed.by/code-pen.

what users will experience at various screen sizes, the only true way to evaluate a responsive design is in the browser.

Because of that, Samantha Warren has created Style Tiles,[23] which can help teams collaborate on a design without committing to a series of fully fleshed out Photoshop mockups.

FIGURE 3.5. Style Tiles, a new design process for responsive websites, smashed.by/style-tiles.

Any time I have presented a handful of breakpoint-based comps for a page, I find myself fielding a series of "What if" questions that wouldn't have been asked had the design been presented in the browser. Style Tiles could be used to get approval on core design elements, before the process of developing a fully formed layout begins. Those elements can then inform the process once approval has been gained.

This is why gradually transitioning the team to a more *browser-centric workflow* can pay off in the long run. Full disclosure: I still like my Photoshop comps for quick ideas and brainstorming because I find myself in such a different mental mode in an image editor than in a code editor. However, once ideas take shape, I transition to code

23. Style Tiles, smashed.by/style-tiles.

as soon as possible to really kick the tires. I also find that saving all of the precise nudging and typesetting for code makes the most sense. After all, you're not gaining any ground by perfecting a JPG layout when it's going to wind up in code anyway. Save all the blood, sweat and tears for the browser.

Tending to the Network

As we further develop the HTML and CSS for our responsive project, a few things can be done to help preserve the hierarchy and keep this painstakingly developed network of content intact.

FIGURE 3.6. An introduction to content choreography on Trent Walton's blog: smashed.by/cont-chor.

CONTENT CHOREOGRAPHY

Earlier, we used media queries to re-stack elements on the page when columns became cramped in narrow views. Taking things further, we shouldn't just consider the space available; we must also consider the content. Imagine a four-column website at full width: as the view narrows, four will become three, then two, then one. A common solution is to stack them on top of each other in chunks.

Now, there's a good chance this rearrangement is consistent with the content hierarchy on the page, but what happens if the first column is really tall? Is the content in column two less important than *all* of the content in column one? In some cases it might be nice to interdigitate content by folding elements into each other as the viewport narrows.

On the next page are a few techniques we can use to better choreograph how content flows from one view to the next. No, these aren't names for 1950s dance moves; they're just horribly coined terms from yours truly. Hey, what do you want from me?

Imagine this for that local news outlet's website I referred to at the beginning, with the forecast (including pollen levels) in the top right. In a narrower view, we'll want the forecast to be directly below the headline graphic, instead of being shuffled down to the bottom of the first column.

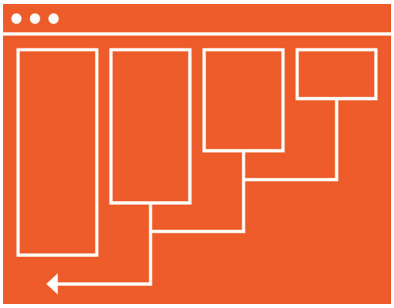


FIGURE 3.7. Often elements are stacked on top of each other in smaller views.

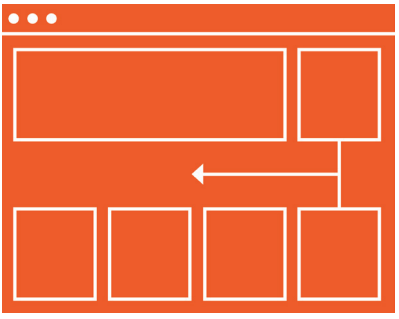


FIGURE 3.8. Sometimes it might be more useful to interdigitate content by folding elements into each other as the viewport narrows.

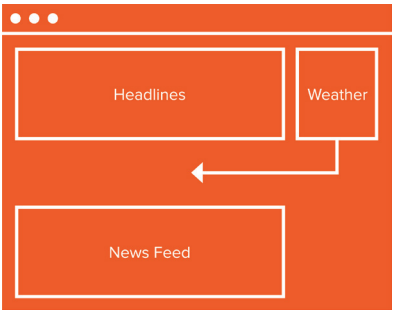


FIGURE 3.9. For example, the "weather" section could be placed under the main headlines rathen than pushed to the bottom of the page.

The Bunch and Slide

One way to handle this is by grouping columns in the markup in a particular way. We could create a row for content that we want to bunch together (in this case, the headlines and local weather). Then, before continuing with the rest of the content, we'd simply close the row and start with yet another container row for the rest of the content down the page.[24]

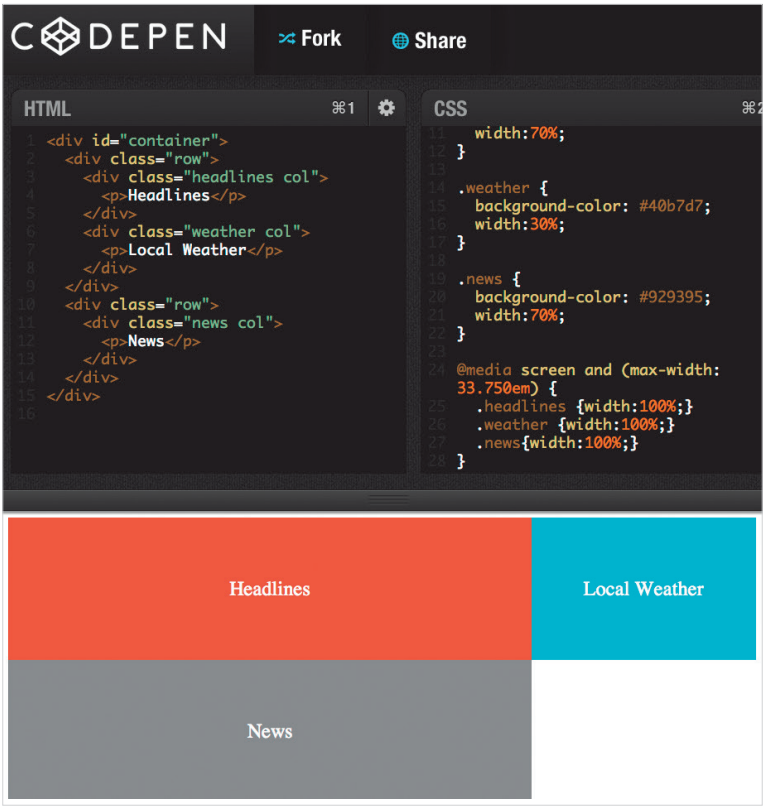


FIGURE 3.10. In this case headlines and weather are bunched together in one row, while the news container fills out the rest of the page.

24. Demo: smashed.by/bunch-slide.

```
<div id="container">
   <div class="row">
      <div class="headlines col">
         <p>Insert headline code here.</p>
      </div>
      <div class="weather col">
         <p>Insert local weather here.</p>
      </div>
   </div>
   <div class="row">
      <div class="news col">
         <p>Insert additional news stories here.</p>
      </div>
   </div>
</div>
```

And the CSS:

```
.col {
     float: left;
     display: inline;
}
.headlines {
     width: 70%;
}
.weather {
     width: 30%;
}
.news {
     width: 70%;
}
```

```
@media screen and (max-width: 33.750em) {
    .headlines {
        width: 100%;
    }
   .weather {
        width: 100%;
    }
   .news {
        width: 100%;
    }
}
```

The Slide and Float

Another option is to use media queries to manage floats. What if our objective were to place local weather *above* the headline graphic in one-column views because some twerp kept filling out a feedback form complaining about not being able to find pollen levels? First, we'd reorder the HTML to reflect our new change in hierarchy.[25]

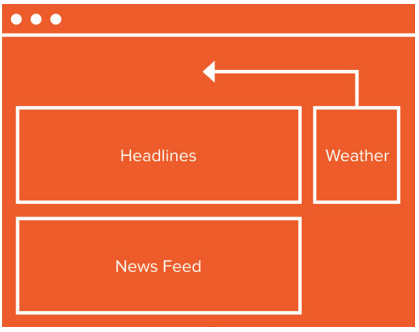


FIGURE 3.11. We could place the "weather" section above the headlines as well.

```
<div id="container">
   <div class="row">
      <div class="weather col">
         <p>Insert local weather here.</p>
```

25. Demo: smashed.by/hierarchy.

```
      </div>
     <div class="headlines col">
          <p>Insert headline code here.</p>
      </div>
   </div>
   <div class="row">
      <div class="news col">
         <p>Insert additional news stories here.</p>
      </div>
   </div>
</div>
```

Then, the CSS could look something like this:

```
.col {
      float: left;
      display: inline;
}

.headlines {
      width: 70%;
}

.weather {
      width: 30%;
      float: right;
}

.news {
      width: 70%;
}
```

```
@media screen and (max-width: 33.750em) {
        .headlines {
            width: 100%;
        }
        .weather {
            width: 100%;
        }
        .news {
            width: 100%;
        }
}
```

Voilà! For any width greater than 540 pixels (33.750 em), the weather would be floated to the right side of the page, adjacent to the headlines.

The Squishy Flexbox

Whereas our first two dance moves techniques allow for minimal content choreography, the sky is the limit with the flexible box layout. Let's revisit the initial markup:[26]

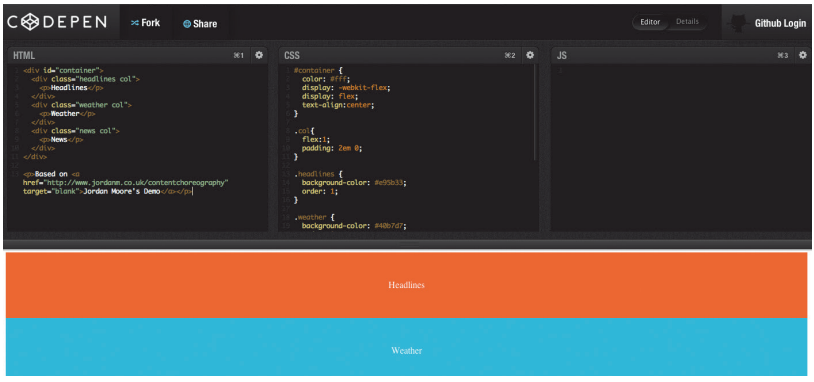


FIGURE 3.12. In this case headlines and weather are bunched together in one row, while the news container fills out the rest of the page.

26. Demo: smashed.by/squishy-flx.

```
<div id="container">
   <div class="headlines col">
      <p>Headlines</p>
   </div>
   <div class="weather col">
      <p>Weather</p>
   </div>
   <div class="news col">
      <p>News</p>
   </div>
</div>
```

Then, in the CSS, we could try this:

```
container {
   display: -webkit-flex;
   display: flex;
}
.col {
   flex: 1;
}
.headlines {
   order: 1;
}
.weather {
   order: 2;
}
.news {
   order: 3;
}
```

```
@media screen and (max-width: 33.750em) {
    #container {
        flex-flow: column wrap;
    }
   .headlines {
        order: 2;
    }
   .weather {
        order: 1;
    }
   .news {
        order: 3;
    }
}
```

You can rearrange the stacking order of elements on the page by changing the order.[27]

RESPONSIVE IMAGE HIERARCHY

Another important aspect of maintaining hierarchy is physical space. Simply put, the larger something is on the page, the more important we perceive it to be. Let's take this page as an example:

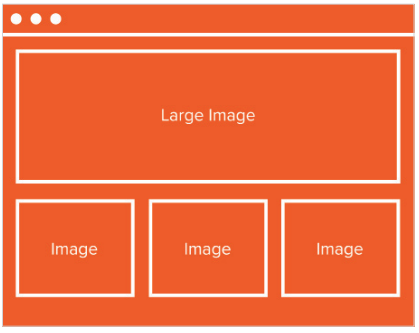


FIGURE 3.13. A layout with a large and 3 column images on wide views.

27. The flexbox specification is still being refined and has recently undergone a massive overhaul, so implement with care. To learn more, check out the specification at: smashed.by/flexbox. Also, Chris Coyier has a rather nice side-by-side demo comparing the old and new specification: smashed.by/new-flexbox.

Obviously, the large panoramic banner is important. It's higher up on the page and larger than the three images below. With fluid images, what will happen to this layout as it is displayed at narrower views? Perhaps the layout displayed on the right?

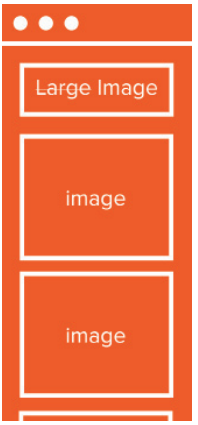


FIGURE 3.14. A layout with a large and 3 column images on narrow views.

No way! What happened to our visual hierarchy here? That banner graphic was the largest thing on the page, appropriately sized to suit the importance of the content relative to the rest of the page. In this narrower view, things are out of whack and all we see is a tiny strip of image, much smaller than the rest of the images on the page.

What can be done to resolve this? Once again with the flexible media save is my coworker Dave Rupert.[28] We can wrap the panoramic element in a `div`. Let's call it `.banner-item`.[29]

Then, we'll use `min-height`, `height` and `overflow: hidden` to keep things proportional. Here's what the code could look like:

```
.banner-item {
   overflow: hidden;
   min-height: 300px; /* 300px is arbitrary. */
}
.banner-item img {
   width: 100%;
}
/* 600px / 37.500em is about the width "locked in" at */
@media screen and (max-width: 37.500em) {
   .banner-item img {
```

28. Rupert, Dave. "Responsive Image Hierarchy," smashed.by/image-hierarchy.
29. Demo: smashed.by/flexmedia.

FIGURE 3.15. A technique that wraps the panoramic element in a `div` and uses `min-height`, `height` and `overflow: hidden` to keep things proportional.

```
        width: auto;
        max-width: none;
        height: 300px;
    }
}
```

With this media query, we've stopped the reduction in height at 300 pixels, maintaining proportional hierarchy.

Now that we've explored so many ways to approach flexible grids and layouts, let's look at how this affects text and readability.

FLUID TYPOGRAPHY

Amidst all of this, we want to ensure readability across all devices and widths. A key part of this is managing characters per line. In a fluid layout, browser width and typographic measure are linked: the wider the viewport, the more characters per line. The Elements of Typographic Style Applied to the Web[30] tells us that 45 to 75 characters per line (including spaces) is generally accepted as safe for comfortable reading. Bearing that in mind, we can do a few things to avoid extra-long lines of text in fluid layouts.

First, we could use media queries to adjust the width of the container. As the viewport gets wider, we could decrease the container's width to give lines of text less distance to span across.

Secondly, we could increase the font size in wider views. A trick I like is to place two asterisks in some placeholder text:

```
Lorem ipsum dolor sit amet, consectetur adip*isicing
elit, sed do eiusmod *tempor incididunt ut labore et
dolore magna aliqua.
```

As I widen the browser window, if at any point the two asterisks appear on the same line of text, then I know it's time to increase the font size. These adjustments are easy (usually one CSS property) because I've used percentage, `em` and `rem` (rather than pixel) units in the CSS.[31]

To my delight, Josh Brewer and Mark Christian released a jQuery plugin named Responsive Measure,[32] which helps to automate this by generating the font size needed to produce the ideal measure for text at a given width.

30. The Elements of Typographic Style Applied to the Web, "Choose a comfortable measure," smashed.by/comf-measure.

31. This technique is a small part of a post I wrote titled "Fluid Type:" smashed.by/fluid-type.

32. Responsive Measure, smashed.by/resp-measure.

This is great for paragraph text, but what if we want the text to behave more like a fluid image, scaling based on the width of the browser window, for more intensely typeset headlines? I had this problem on my own blog, and, fortunately for me, I'm friends with Dave Rupert. He built a jQuery plugin, FitText, to inflate Web type, and we released it on GitHub.[33]

Sean McBride has a great example of FitText in use.[34] There, Sean has applied FitText to an h1 heading whose width he has governed via `padding: 5em 40% 2.5em 5%`. The 40% padding on the right leaves room for the focal point of the photograph. The text scales to fill the width of the container with the addition of the following script in the head:

FIGURE 3.16. Increasing font size is easy because percentage, `em` and `rem` units have been used here.

```
<script>
    $(function() { $('.heading h1').fitText(5.7) });
</script>
```

33. FitText, smashed.by/fittext.
34. Demo: FitText, smashed.by/fittext-demo.

Actually, if you like FitText, then you'll love Molten Leading.[35] Dreamed up by Tim Brown and built by Mat Marquis, Molten Leading automatically adjusts line height to a minimum and maximum range within a minimum and maximum pixel value width range. In other words, it gives us a fluid way to set line height.

Even better news is that, one day, things like FitText could be made obsolete by `vw`, `vh` and `vmin` units of measure. It's pretty simple:

- 1 vw is 1% of the viewport's width;
- 1 vh is 1% of the viewport's height;
- vmin is 1 vw or vh, whichever is smaller.

Instead of using JavaScript and the percentage-based widths that FitText requires, these units would do the job all on their own. These units could also be used to more precisely govern typographic measure across all viewport widths once they gain more browser support.

CODE-BASED WEB DESIGN STANDARDS

As basic design elements such as typography, buttons, navigation and general page structure become finalized, it's worth considering building a page or an asset library website. This would help get things ready for production, but it could also help establish visual standards and assets for a project.[36] In my mind, I picture this as a light version of the fantastically comprehensive Twitter Bootstrap,[37] but unique to your project. Having design examples in a live code setting is the most accurate way to test and continually improve a website. The way I see it, using images to show a responsive layout is like test driving a car that's stuck in park.

35. Molten Leading, smashed.by/molten-leading.

36. You can review some examples of front-end style guides in a Gimme Bar collection: smashed.by/frontendguides.

37. Twitter Bootstrap, smashed.by/tw-bootstrap.

Onward!

Responsive Web design has restored my sense of wonder for our craft. We've reached a new frontier, and the uncharted territory ahead is gnarly. With a spirit of bold adventure, let's press on. How will online banking, extensive drill-down menus and content management systems look in a responsive setting? Brad Frost, the responsive Web's faithful docent, will take over in the next chapter, helping us to begin answering some of these questions.

ABOUT THE AUTHOR

Trent Walton is one of the founders of paravelinc.com, a small Web shop based out of the Texas Hill Country, where the chicken fried steaks are as big as your face. Together with Dave Rupert and Reagan Ray he is designing and building for the Web since 2002. The most important lesson he has learned in his career is that there is no substitute for being in control of your own destiny.