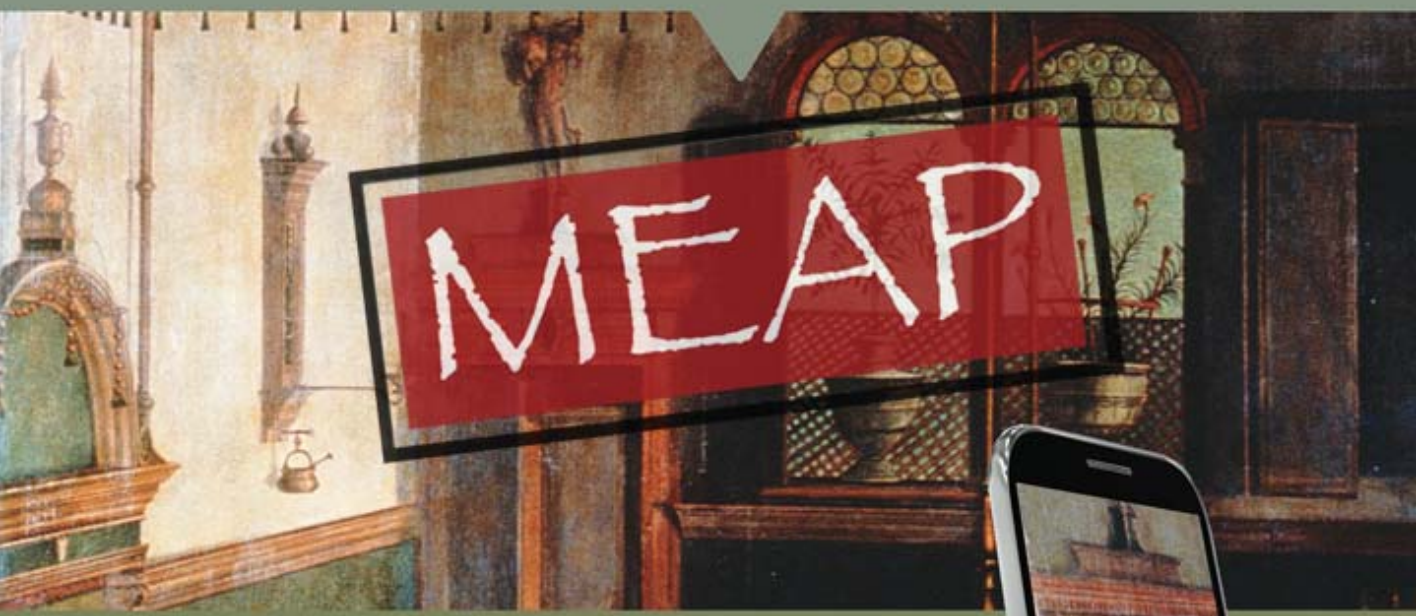
Responsive WordPress

Creating modern, scalable WordPress themes



Tracy F. Rotton

MANNING

MEAP Edition
Manning Early Access Program
Responsive WordPress
Creating modern, scalable WordPress themes
Version 4

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *Responsive WordPress: Creating modern, scalable WordPress themes.* It's thrilling for me to see this title become available, but I know that there is still much more work to be done. Responsive Web design is a hot topic these days, and for good reason. Mobile Web usage continues to grow at an astounding rate, and providing a great experience to your mobile visitors is more important than ever before. And of course WordPress continues to grow in popularity as well as functionality, making it one of the most popular Web content platforms on the planet. It's certainly an exciting time to be working with these technologies, and I'm very happy to have the opportunity to write this book.

This book is for Web designers and developers who want to build a WordPress theme for our modern, multi-device world. A strong knowledge of HTML and CSS is necessary, as is some familiarity with JavaScript using jQuery.

We're starting the MEAP by releasing chapters 1 through 3. Chapter 1 takes a look at the history of responsive Web design, how it came to be, and where WordPress fits into the grand scheme of things. We'll take a look at the early history of the "mobile" Web, learn about Ethan Marcotte's seminal blog post that got this whole thing started, and what it means to be "mobile first." We'll also look at WordPress themes available now that are responsive, and explore a little about what makes them so.

Chapter 2 moves on to some of the technical fundamentals of responsive Web design, such as the meta viewport tag, the fluid grid, CSS3 media queries, and responsive images. We'll also look at some of the key characteristics often found in WordPress themes and examine some of the code we're likely to come across. Common WordPress traits such as menus, sidebars and widget areas and how WordPress handles images will all be covered.

In chapter 3, we begin to design the theme that we are going to spend the rest of the book building. We'll start by figuring out what we want our theme to be, move on to creating wireframes, and then finally a visual design. Along the way, we'll look at tools, both traditional and cutting-edge, that can be used to plan your responsive theme.

Looking ahead, future chapters will take you through the following: creating a static prototype, an introduction to CSS preprocessors and how they're used in responsive Web design, building a responsive grid for your theme, creating our WordPress theme development environment, and so on. The goal is if you follow along with the project in this book, you'll have a fully responsive and ready-to-implement WordPress theme when you are done.

All throughout the process, I'll be watching the Author Online forum for your feedback.

Responsive Web design is an ever-changing process that is still in its infancy. New information, techniques, and best practices are being developed every day. As I write this, I will do my best to keep current with changes in technology that might affect the content.

Thank you again for purchasing this MEAP. I hope you find it useful to your work.

Tracy F. Rotton

brief contents

1

Our Mobile, WordPress-Powered Web

This chapter covers

- A brief history of responsive web design
- The “Mobile First” philosophy
- Why responsive web design is important to WordPress

In the beginning(of website development, that is), web designers and developers had to design websites according to a certain “lowest common denominator” of possible monitor sizes out in the wild. This number grew over time, and for a while stabilized at 1,024 pixels wide by 768 pixels high. This suited people just fine, and websites remained geared towards this assumed minimum standard for years. But eventually, that was to change, and change radically.

In January 2007, the late Apple, Inc. CEO Steve Jobs introduced a device unlike anything the world had seen before: the iPhone. A mere six years later, it and its Android and other smartphone counterparts are ubiquitous today, but at the time it represented a sea change in how we interacted with the larger connected world. Indeed, Jobs didn’t just introduce a new kind of cell phone, but something he termed a “an iPod, a phone and an Internet communicator.”

The iPhone, compared with the prevailing style of feature phones and Blackberries of the day, was extremely minimalistic. With a single button to bring you back to a home screen, the physical form of the device was nothing more than a three and a half inch monitor with a resolution of 320 pixels wide by 480 pixels high. Icons that you touched with only your finger (no stylus, thank you very much) were the primary means of interfacing with it. If you needed to do something more, enter a phone number, perhaps, or send a text message,

a virtual keyboard would appear. Furthermore, this keyboard would change to match the context of the task at hand: a numeric keypad for entering phone numbers; URL-friendly characters when entering web addresses, etc.

1.1 The Growing Mobile Web

The first iPhones were expensive, unsubsidized and ran on the painfully slow AT&T EDGE network. But they did come with the novel feature that for $30/month, you could access an unlimited amount of Internet data. The previous paradigm of expensive and confusing mobile data plans shifted forever, replaced (at least for a time) with economical, all-you-can-eat plans that meant you didn't have to watch every megabyte you downloaded. Since the iPhone's introduction, this has changed somewhat, and now tiered data plans are more the norm. But at the time, it was a one-size fits all kind of mobile data world.

All of this led the way for other device manufacturers, Internet content providers, and cellular carriers to begin building devices more like the iPhone than the feature phones that had dominated the cell phone market until then. In the same year as the iPhone announcement, Google introduced the Android operating system as an iOS competitor, and the first Android-powered phone came on the market in 2008.

Today, iOS and Android phones are everywhere you look, with screen sizes that range from the 3.5" iPhone 4 to the 5.5" Samsung Galaxy Note II. In 2010, Apple released the iPad, essentially a 10" tablet version of the iPhone, except without the cellular phone capabilities, although cellular data plans *were* available. Numerous Android-powered tablet devices followed the iPad, with sizes ranging from 7" to 10". Windows Phone 8, Blackberry 10 and Symbian and the hardware they run on also contributed to this ever-growing ecosystem of mobile devices. And yet, for the longest time, web developers essentially ignored the challenges that this new multi-resolution landscape presented them.

1.1.1 Early Mobile Websites

Before the introduction of the iPhone, there had been some early attempts at mobile website development through a technology called Wireless Application Protocol (WAP). The mobile landscape, such as it was, was dominated by the waning influence of Palm Pilot devices and feature phones which could send text message, *maybe* take pictures, and had very rudimentary web browsing capabilities that existed on very slow networks. Websites built to serve this tiny mobile market never gained any significant popularity, however, because of the high fees cellular carriers charged subscribers for Internet access, and because wireless devices, if they had Internet browsing capabilities at all, required WAP sites coded in Wireless Markup Language (WML), and these sites were incompatible with "ordinary" websites.

Other factors contributed to the overall stagnation of advancement in web standards. The dominance of Internet Explorer 6, with is quirks and lack of standards support, had kept the development of new web technologies at a practical stand still for the better part of the 2000's. And Flash, the multimedia technology pioneered by Macromedia (which was later

acquired by Adobe), was the primary means of distributing rich content such as video and browser-based games online. The iPhone famously did not, and still doesn't, support Flash, and even Adobe has since abandoned all efforts to provide a mobile version of Flash.

To be fair, it wasn't immediately apparent just how much of a disruption the iPhone was going to be to the web status quo. In his 2007 MacWorld Expo keynote, Jobs demonstrated how the iPhone could surf the web by double-tapping certain sections of a zoomed-out page to magnify the particular article one wanted to read. There was also the ability, thanks to "multi-touch", to stretch the view through a reverse-pinching gesture, and zoom back out again with a pinch. The implication, of course, was we don't need "mobile" websites, thanks to the iPhone. The iPhone could browse *any* website that your standard desktop computer could.

But after a short time, it became obvious that the Jobsian ideal of a pinch-and-zoom world was, to say the least, impractical and surfing a website in this manner was painful at best. For his demonstration in the keynote, Jobs selected articles from the online edition of the New York Times, a site whose content is broken down into neatly arranged columns of articles that become readable upon magnification. However, the same double-tap to zoom didn't work if a website happened to not use columns, and instead consisted of text that stretched the width of the page.

In an attempt to address the needs of the ever-growing smartphone population, web developers started creating special "mobile-only" sites that specifically targeted the iPhone and, eventually, other mobile devices. A typical experience often worked like this:

1. A user on an iPhone would visit a "desktop" website.
2. That website would use server-side technology to detect that he was on a website, usually through a process known as user-agent sniffing.
3. The website would then redirect the iPhone visitor to the "mobile" version of the website, often with a URL along the lines of "http://m.mywebsite.com".

These websites were constructed using the same HTML, CSS and JavaScript as your ordinary desktop sites, except they were structured in such a way as to be readable without all the pinching and zooming. While certainly a step in the right direction, these mobile sites did have some non-trivial flaws. The first was for the developer, who now had to maintain two different code bases to deliver the same content. If different databases were used, then making sure they stayed in synchronization became an issue. This led to increased development and maintenance costs that were prohibitive enough to deter many companies from bothering to develop a mobile site.

Most of the other problems with mobile sites were more on the user experience end. Often these sites were a stripped-down version of their desktop counterparts, and the information one could find on the desktop site often did not exist on the mobile one. Sometimes there was a link on the mobile page that allowed a smartphone user to access the desktop site, but not always. Besides, even if one clicked it, they'd be back to the whole pinch-and-zoom interface. Not ideal.

Another problem was in the technology that redirected to the mobile site in the first place. Where user-agent sniffing was involved, the server might redirect the user if it determined that the browser being used was Mobile Safari, the version of the Safari web browser that Apple ships with its iOS devices. This was fine prior to 2010, but with the introduction of the iPad and its 1024x768 display, which also happened to run Mobile Safari, iPad users found themselves being redirected to mobile sites, which looked ridiculous given that the iPad had the same resolution as many older desktop monitors. And if the mobile site *didn't* have a link to the desktop version, iPad users were stuck in a mobile experience (figure 1.1).

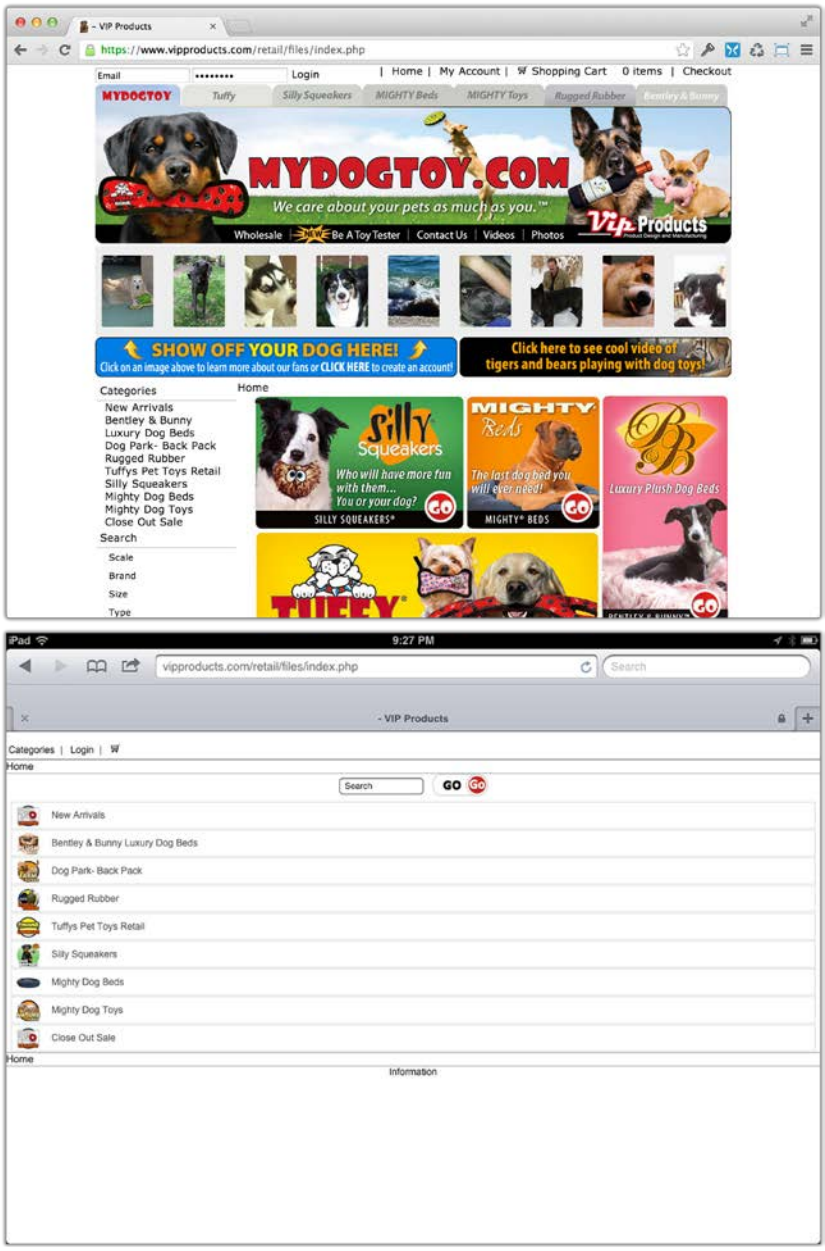


Figure 1.1 Same website, same size windows. The top is using Google Chrome on a MacBook Pro while the bottom is using Mobile Safari on the iPad 2. If an iPad was your primary means of accessing the web, websites like this might leave you feeling like a second-class webizen.

The last major problem has been termed the "server attention span" problem, and it goes something like this: Have you ever been reading Twitter on your mobile phone and tapped a link, only to be redirected to a mobile home page instead? And once there, there was probably no easy way, if any way at all, to find the article you clicked the link for in the first place. Yeah, that's pretty annoying, as the web comic XKCD illustrates in figure 1.2.

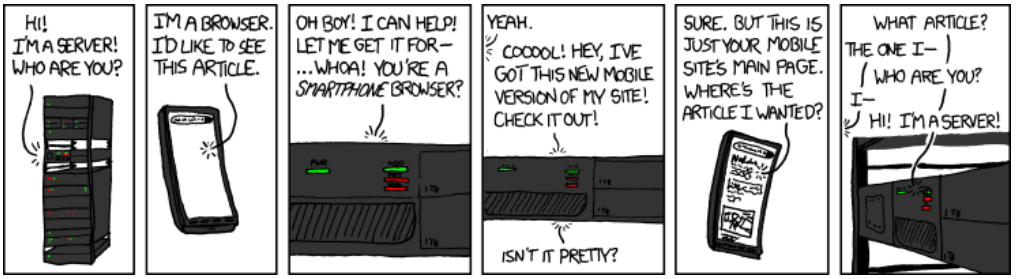


Figure 1.2 The server attention span problem, artfully illustrated by XKCD at http://xkcd.com/8691.

Mobile sites certainly have their share of problems, but at the very least they made websites more readable on smartphones, sort of. But a bigger shift in web development was on the horizon.

1.1.2 Introducing Responsive Web Design

By 2010, the scene had been set for something drastic to happen in web design philosophy. After years of stagnation and political in fighting amongst the various web governing bodies, the technologies that made up the web began to move forward again. A new standard in the HTML markup language began to emerge in the form of HTML5 with a bevy of new features and APIs such as native video and audio handling, new and more semantic markup elements, and other "extras" such as geolocation and local browser storage.

The styling syntax of the web was also moving into a new phase. CSS3 introduced abilities that had until that time been the sole domain of images: rounded corners, drop shadows and alpha-channel transparency on colors greatly reduced the need for many arduous hours of cutting up Photoshop comps just so you could display a banner headline in a particular font with a particular shading. We were truly reaching a golden age of front-end development!

But what many of us web developers didn't know about CSS3 was a little gem of a notion called media queries which are cues in the CSS to apply different properties when certain conditions, such as browser width, are met. It would take one guy and his blog post to expose this seemingly innocuous piece of code and exploit it in a wonderful way.

ETHAN MARCOTTE'S REVOLUTION

Rarely does publishing a single blog post cause a revolution, but it can be argued that that's what Ethan Marcotte did in May of 2010 when he published the article "Responsive Web

Design" in the front-end web development blog *A List Apart* (http://alistapart.com/article/responsive-web-design). This groundbreaking article described a new technique of combining fluid layouts with CSS3 media queries (we'll explore the technical aspects of these a bit more in chapter 2) to form a web page that conforms to the device width.

Marcotte followed up his article with a book of the same name, containing detailed instructions on how to build fluid grids, work with flexible typography and deal with responsive images and media. He then put his money where his mouth was when his web development agency, Filament Group, launched a completely responsive redesign of the Boston Globe website. In less than a year, responsive web design went from completely unknown concept to the driving design philosophy behind the website of a major American newspaper (figure 1.3).

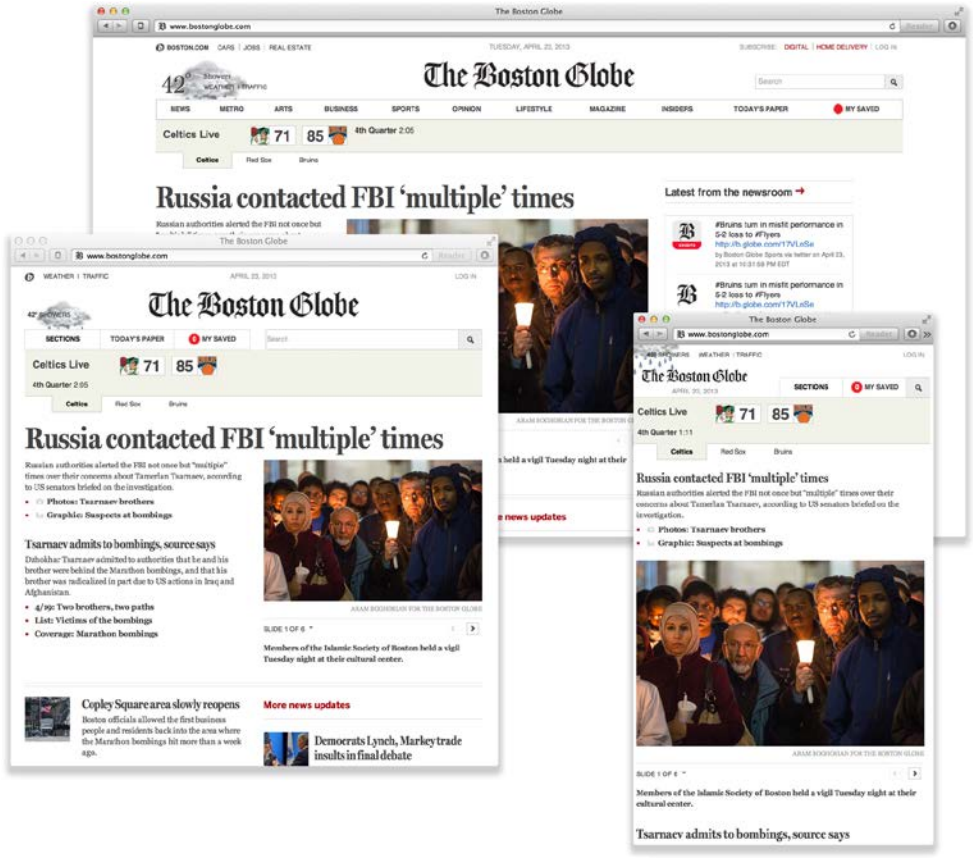


Figure 1.3 The Boston Globe website, http://www.bostonglobe.com, was the first significant website to embrace responsive web design.

It was around the time of the Boston Globe launch that I picked up Marcotte's book, and with it my view of web development changed forever. This was exactly the kind of thinking that was needed in the web development community: The idea that the web wasn't built on a fixed and reliable canvas. In fact, that had *never* been true; we had just pretended it to be so because of the limitation of our tools.

Instead, the web is fluid and ever changing. More importantly, it was about time we not only realized that we had no control over the visitor's choice of browsing experience, but instead embraced that uncertainty to deliver the best user experience possible. We were now building websites in three dimensions: width, height and *change:* change in browser size, change in browser capabilities, and change in the means people use to connect with the web, often from one visit to the next.

The driving force behind responsive web design isn't altogether that complex. Use a media query to detect a certain condition about your browser and, if true, alter your CSS in a prescribed way. In a nutshell, that's what responsive web design boils down to. Of course, the implementation is far more nuanced than that, as we will see. But wrap your head around CSS3 media queries, and you've got the basic gist of it.

Adaptive Web Design

From time to time, you may hear the term "adaptive web design" and might be wondering how that differs from responsive web design. Unfortunately, the distinction isn't always very clear, and adaptive web design can be defined in several different and contradictory ways.

Sometimes adaptive web design is used to refer to a website that uses media queries to adjust its layout at different widths, but unlike responsive web design which includes a fluid grid that expands and contracts to every possible pixel width, adaptive web design instead targets distinct widths (such as those common to iOS devices). Ergo, there is no fluidity in the design, only a change in layout to meet the specific needs of particular devices.

However, the man most often credited with coining the term adaptive web design, Aaron Gustafson, defines it differently (see "On Adaptive vs. Responsive Web Design," http://blog.easy-designs.net/archives/2011/11/16/on-adaptive-vs-responsive-web-design/, and *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement,* Easy Readers, 2011). In his view, adaptive web design is less concerned with the layout of a website, but rather in using various means of progressive enhancement in order to tailor the overall experience to the browser visiting the site. That is to say, the layouts may change (and may or may not be fluid), but furthermore, hover interactions that would be appropriate in a desktop environment would be dropped for more touch-friendly interfaces on a touch device. Also, browsers that support more advanced features, such as the HTML5 <canvas>element and geolocation features will get more content geared towards these expanded capabilities than less modern browsers.

In this way, the theme that we build will meet this second definition of adaptive web design, and for that reason that's the definition that we'll be using. But as responsive web design is the more common description of websites that can transform themselves from mobile to tablet to desktop, we will predominately be using that term.

IT CAN'T ALWAYS BE THAT SIMPLE, RIGHT?

So if everything I need to know about responsive web design is "media queries", why do I need this book? - Because the challenges of putting the pieces of a responsive website together are more complex than just that. Surely we can't slap some CSS inside a media query and call it a day. As with the early attempts at mobile websites, responsive web design comes with its own share of drawbacks.

Because of their added complexity, responsive websites are more costly to plan out than traditional desktop sites. This only makes sense, as you are designing multiple contexts of a website (what it looks like on a mobile device versus a desktop computer versus something in between). Often these additional costs and time in the planning, design and development phases can be a hard sell to clients. Web development has always had a dose of client education in the job description, and this is just the latest topic about which we need to inform.

Also problematic is supporting legacy browsers. If you read "Internet Explorer" into that last sentence, go get yourself a cookie. While rival web browsers such as Google's Chrome and Mozilla's Firefox are gaining marketshare over the venerable offering from Redmond, Internet Explorer is still the monkey on every web developer's back. If you have absolutely no need to support any version of Internet Explorer before version 9 (and have proof of this), then you can skip over every section on old Internet Explorer polyfills I've included in this book. Internet Explorer 9 happily supports CSS3 media queries (figure 1.4), and while it's support for other CSS3 properties is at times incomplete, it'll still handle most of what you're going to throw at it.

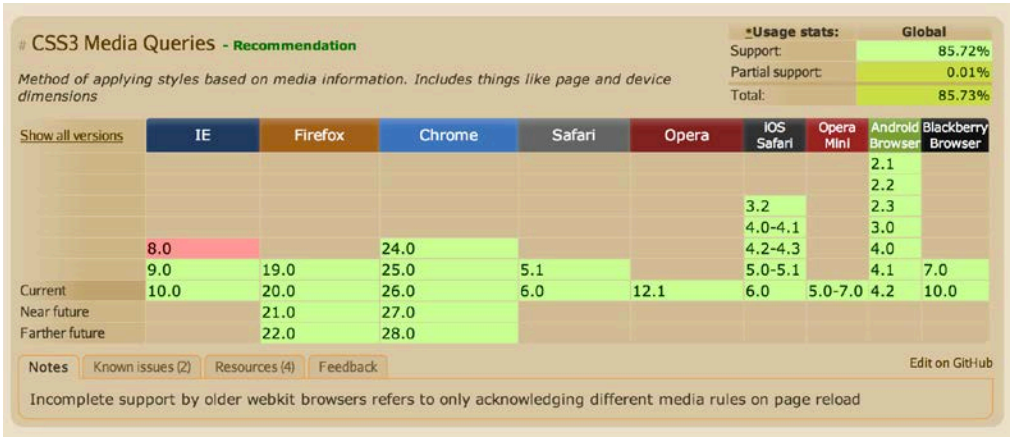


Figure 1.4 http://caniuse.com, a good reference for browser support of various HTML and CSS features, shows broad support for CSS3 media queries across every major browser – except IE8 and below.

But if you live in the real world, and are forced to make things work in version 8 or (sigh) version 7, then you're adding hours to your development time. Realize that now so that you won't be surprised later. Hopefully you've completely moved away from having to deal with Internet Explorer 6; even Microsoft has given up on supporting it. But if that's still in your list of required browser support for some odd reason, you probably shouldn't build a responsive website. Sorry.

REMEMBER THE "MOBILE" PART

There is one other major drawback to responsive web design that its critics (and yes, they exist) often cite as a reason to not use it, and that is the issue of page weight. Many responsive websites are built in a manner referred to as the "desktop-down" approach that basically took a fully featured and rich desktop experience and just squeezed it down into dimensions that would fit on a smartphone. As a result, everything, and I mean *everything*, that existed on the desktop site would have to load on the phone: Lots of content; large images, sometimes full bleed; video; multiple fonts; lots and lots of HTTP requests; etc. On a slow connection, this could take minutes to load in a worst-case scenario. On a data-capped plan, you could blow through megabytes without even realizing it, resulting in speed throttling or surcharges. And sometimes, if an image didn't make sense at a smaller resolution, it would be hidden via CSS with the "display: none" property, meaning that the browser had to download a possibly large image file which the user wouldn't even see!

If the idea of responsive web design was to make content more accessible on phones and other mobile devices, then it stood to reason that these needs of mobile visitors should come to the forefront. This lead to the philosophy of "mobile first." The term, first coined by Luke Wroblewski in a post on his blog (http://www.lukew.com/ff/entry.asp?933) and further explored his book of the same name (A Book Apart, 2011), was based in part on the

progressive enhancement strategy that dictated you should give all browsers a certain baseline experience, then increase the bells and whistles as the various visiting browsers could support them.

What does it take to build a site mobile first? It means that you need to plan your content strategy upfront; don't feel like every piece of information needs to be served to every device. Also, be considerate of your visitor's bandwidth constraints by serving up appropriately scaled images and media. We'll see that not all of this can be done solely through front-end technologies. Even in responsive web design there are some things that the server can do better than the client (client in this context meaning the web browser software). But realizing that the needs of mobile users are not always the same as desktop, or tablet, users is the first important step towards a mobile first, responsive website.

1.1.3 Don't Forget About the Server

A fun trick to do on a responsive website is to take the side edge of your browser and drag it so that the browser window grows and shrinks in width, watching how the elements on the page shift in response. Unless the site is coded with a plethora of "display: none;" properties in its CSS (and we'll see how to avoid those in part two of this book), all of the elements that are in the "desktop" view will appear in the "mobile" view as well, albeit in a different layout. The browser can't make any determination on its own of what pieces of content make sense in one view versus another.

But web pages are served from a source, and the server in question *can* perform this sort of logic. However, any logic regarding what to serve to the browser that's performed on the server end is not going to be affected by this browser resizing. Some might say that that's not truly "responsive," and that true responsiveness involves front-end technologies only.

Such thinking ignores the whole point of what is trying to be accomplished. Responsive design isn't about making elements fly and resize on a page at will; it's about presenting one version of a website that can conform to the needs of the visitor, and every need must be considered and respected. The vast majority of the time, it will be front-end tools that will help us accomplish this. But sometimes, we need the logic of the server to control what gets sent to the browser in the first place. And while it's not realistic to avoid every single use of the "display: none;" property, the true skill comes in knowing when to worry about optimizing our pages on the server end versus altering the display on the front-end. In the end, it's about delivering the best experience possible to our visitors, wherever they might be.

1.2 So Let's Talk About WordPress

Now that we've looked at responsive web design, where it came from and where it might be heading, let's take a look at how a once-humble blogging engine turned into one of the powerhouse CMS platforms on the planet, and how responsive web design figures in its present and future.

1.2.1 Humble Origins

In 2003, WordPress co-founder Matt Mullenweg decided that b2, the software he used for his weblog, needed updating, but its developer seemed to have disappeared. So Mullenweg and fellow developer Mike Little decided to fork it and start their own project. Thus WordPress was born.WordPress began its life as a straight blogging platform, and as such, its capabilities were limited. But over time, features were added such as widget support, custom post types, post formats, etc., all of which added polish and power to the WordPress core.

Also available to WordPress sites was a bevy of downloadable themes and plugins to make up for desired functionality that the core installation of WordPress lacked. Themes could easily be switched by a few clicks of the mouse, while plugins ranged from small monotaskers to multi-featured mini platforms of their own. All of this contributed to the WordPress ecosystem in a way that allowed WordPress to be nearly anything the developer wanted it to be. It became, over time, a robust content management system. In fact, in his 2012 State of the Word address at WordCamp San Francisco, Mullenweg noted that 66% of the websites being run by WordPress were used as a CMS, and *not* as a blog.

As the WordPress feature set grew, so did its popularity. As of the writing of this book, estimates on the market share enjoyed by WordPress vary from about 14% to 22%. That is to say, somewhere in the neighborhood of one out of every five websites around the world is powered by WordPress, including such high profile sites as the Hollywood trade publication *Variety*, the rock band the Rolling Stones (figure 1.5), and the blogs of the *Washington Post*.

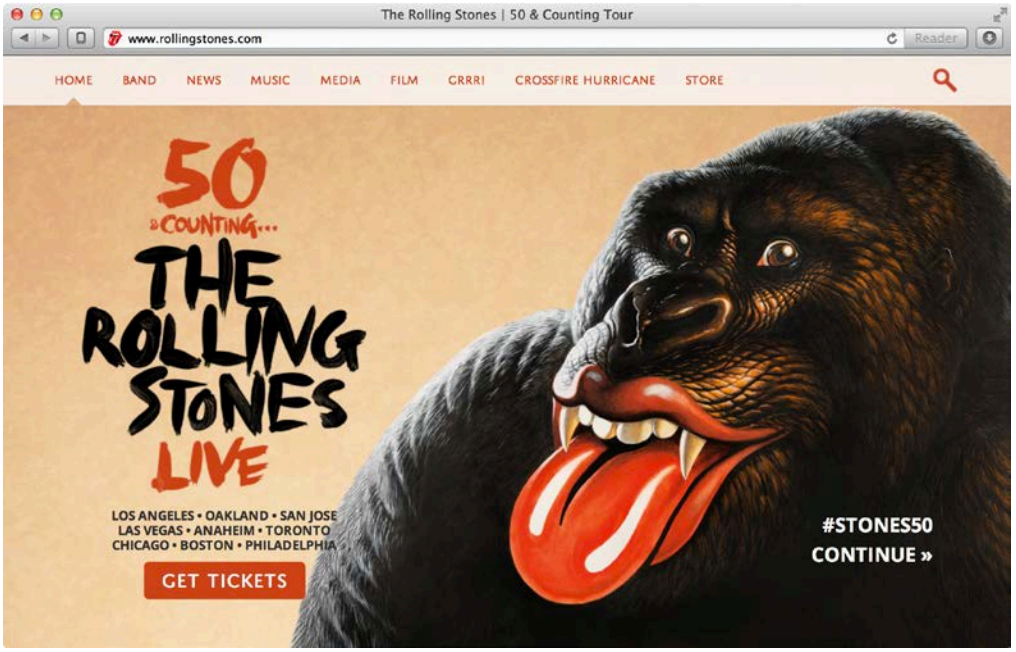


Figure 1.5 It's not your typical blog, but the Rolling Stones roll with WordPress.

1.2.2 WordPress.org versus WordPress.com

In August of 2005, Mullenweg founded the company Automattic which launched WordPress.com, a hosting solution which allowed anyone who wanted create their own WordPress website without having to download and install the server themselves on their own webhost. A basic blog with a subdomain of wordpress.com is free for anyone who wants an account, and additional services such as your own domain name and other added capabilities can be had as paid upgrades.

While WordPress.com has contributed greatly to the popularity of WordPress as a platform, it is still limited in the things you can do. A paid upgrade will allow you to customize a selected theme's appearance, but you can't upload a new theme of your very own. Nor can you upload your own plugins (although WordPress.com offers access to a multitude of popular features that plugins would otherwise provide). To do any of these things, you're going to have to go the self-hosted route.

When it comes to terminologies, it's important to realize the distinction between WordPress.com (sometimes referred to as simply .com) and WordPress.org (likewise referred to as .org). WordPress.com is a *hosted* solution: You don't have to download any software, and all of the headaches related to that are off your plate. But as we've seen, the true cost

of that is flexibility; you're going to give up a lot of control when you're hosting your site on WordPress.com. WordPress.org, on the other hand, is a *self-hosted* solution: You are responsible for finding your own webhost, downloading and installing the software, managing the database, and installing any themes and plugins you want to use with your site. Many times, a popular webhost will help you with installing the core software, often with via a single button.

In this book, we will be concerned with WordPress.org and the self-hosted software. We'll want to get under the hood of WordPress and get our hands on the code involved with building our own theme, including writing our own templates from scratch. If you've never installed WordPress on your own before, Appendix A will walk you through the steps to do so on a Unix-based platform.

1.3 Responsive Design Meets WordPress

In April of 2012, I gave a talk to the WordPress D.C. meet up group on responsive web design and WordPress. For my talk, I wanted to point to some available WordPress themes that were good examples of responsive design. I struggled to come up with four, and I wasn't totally convinced that they were really great examples. I included the WordPress default theme at the time, Twenty Eleven, even though its responsiveness was extremely minimal. In fact, the horizontal primary navigation wasn't responsive at all, so part of my talk was taking the audience through making it responsive, step-by-step.

Fast-forward to a year later when I'm writing this book, and the selection of responsive WordPress themes has exploded. Free responsive WordPress themes are coming to the WordPress.org theme repository on a steady basis, and many premium theme sellers offer a selection of high-quality responsive themes to meet the varying needs of businesses, photographers, casual bloggers and more.

1.3.1 Default WordPress Themes

WordPress would not be the web platform juggernaut it is today if it didn't look good, and that's where themes come in. Themes are the set of template files, stylesheets, JavaScript, images and other logic that provides the front end "look and feel" of a WordPress site. Without a theme, the data that comes out of WordPress doesn't look very pretty. In fact, it doesn't look like anything at all. When you remove all of the themes from the `wp-content/themes/` directory, all you get is a blank white browser window, lovingly referred to as the "white screen of death" (see figure 1.6).

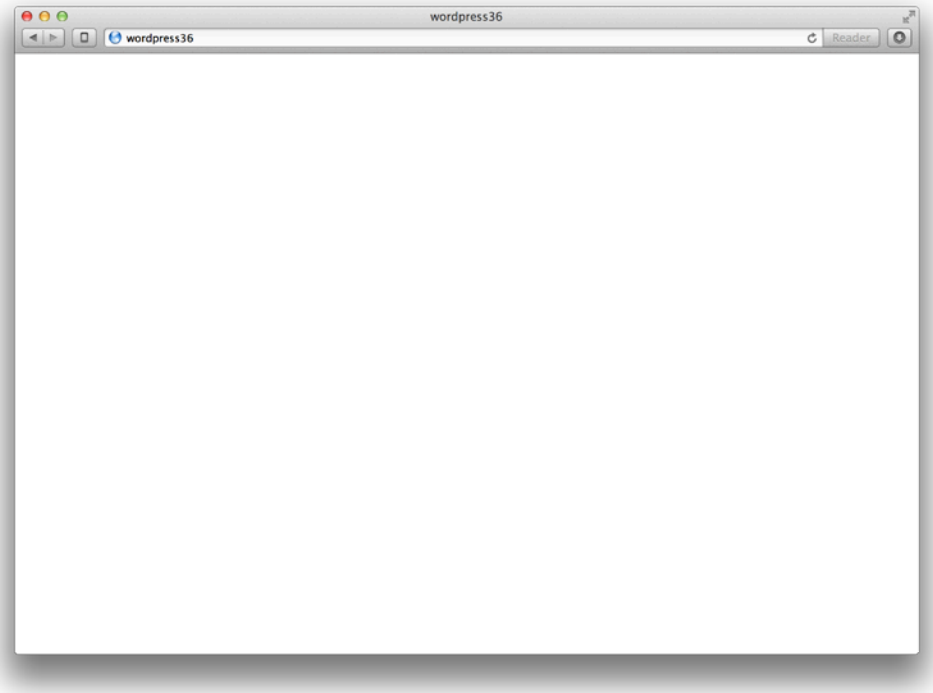


Figure 1.6 White screen of death is white.

Any theme included in the core distribution of WordPress is considered a "default" theme and since version 3.0, released in June of 2010, the core team has committed to releasing one new default theme per year. As of the publication of this book, there have been four themes that have been released as default themes: Twenty Ten, Twenty Eleven, Twenty Twelve and Twenty Thirteen. (Notice a pattern?)

TWENTY ELEVEN

Twenty Eleven was the first of these default themes to make any attempt at responsiveness, although that attempt was quite small. The responsiveness boils down to a desktop-down approach, where at a maximum width of 800 pixels, the sidebar stops floating alongside the main content and is instead stacked beneath it. At smaller breakpoints, adjustments are made to the font size, the layout of gallery posts and margins and paddings.

Given the fact that responsive design, as a concept, had only been around for a few months, Twenty Eleven's incorporation of even this most basic of responsive principles represented a large step in the acceptance of this development philosophy. After all, Twenty Eleven was the default browser for every new WordPress installation of version 3.2, as well

as being available for all of the blogs hosted by WordPress.com. Thousands of websites could now boast a (somewhat) responsive web design with just a five-minute WordPress installation (figure 1.7).

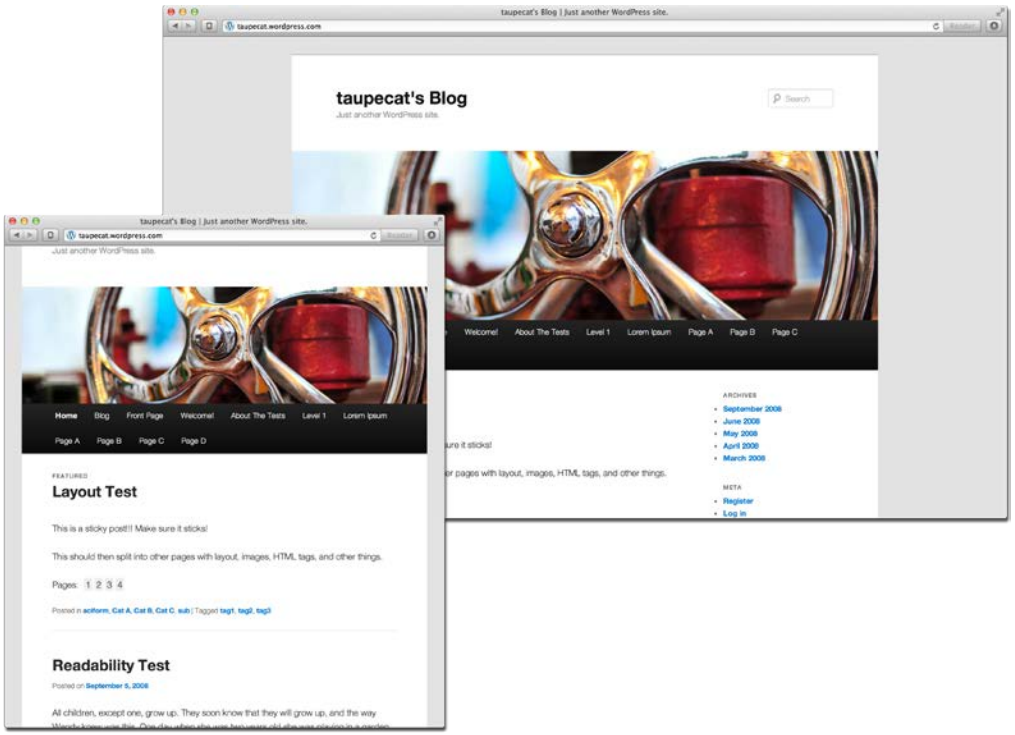


Figure 1.7 It wasn't earth-shattering, but it *was* ground-breaking when WordPress included a responsive default theme, Twenty Eleven, in its core installation.

TWENTY TWELVE AND TWENTY THIRTEEN

The next two WordPress default themes were significantly more focused on responsive design. Twenty Twelve (figure 1.8), released with WordPress 3.5 in December of 2012, was a mobile-first theme and was designed as a way of showcasing WordPress' emerging role as a CMS. At mobile widths, the main navigation collapsed down into a "Menu" button that expanded and contracted as the button was pressed. At a breakpoint of 600 pixels, the navigation expanded into a more typical horizontal layout. Also at 600 pixels, the sidebars (if any) were set to float along the right side of the main content.

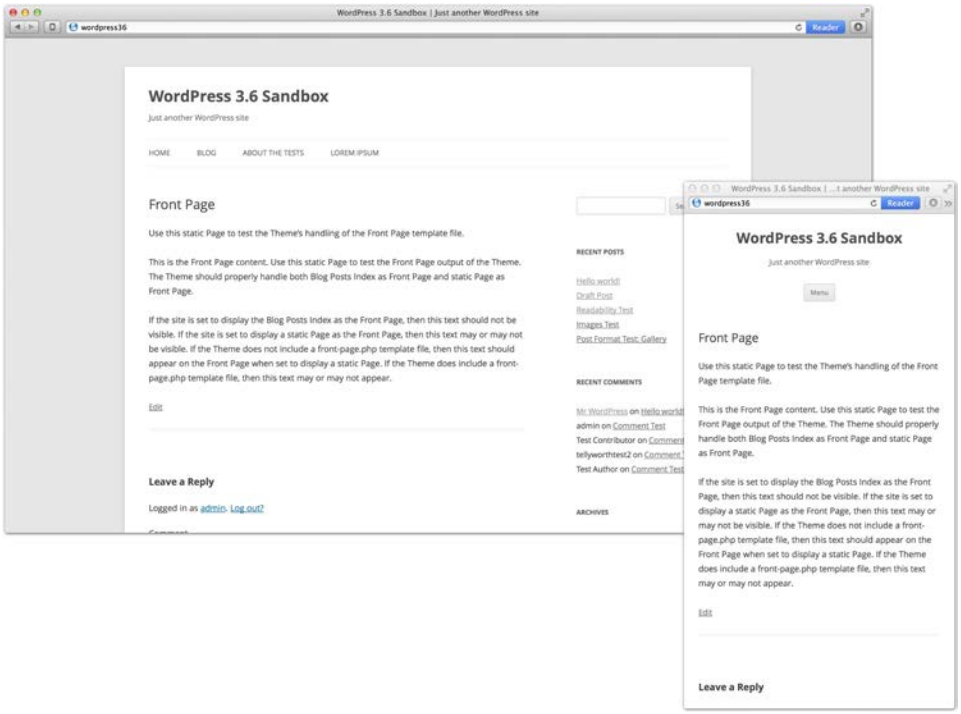


Figure 1.8 Twenty Twelve was the first WordPress default theme to boast a fully mobile-first responsive design.

Like its predecessor, Twenty Thirteen (figure 1.9) also sported a complete responsive design, however it took a desktop-down instead of a mobile-first approach. It also featured a bold color scheme (previous WordPress default themes were best characterized as monochromatic), increased support for WordPress post formats, and accessibility enhancements. And like Twenty Twelve before it, the main navigation collapsed down into a drop-down list that was activated through a "menu" toggle. Twenty Thirteen was released with WordPress 3.6.

Desktop-Down Versus Mobile-First

In this section, I've written quite a bit about the mobile-first philosophy and how a responsive website can be built either desktop-down or mobile-first. But how does one tell the difference between the two? The clues are in the CSS3 media queries that enable the responsiveness site. We'll examine media queries in chapter 2, so we'll go for a non-technical definition right now.

In a mobile-first implementation, you build the CSS for the smallest width by default, and add in styles for wider browsers by way of breakpoints. These breakpoints have the functional equivalent of saying, “Do everything you did before, and then at widths equal or greater than X, do some more stuff.” (Where X is your breakpoint’s width, of course.)

In a desktop-down implementation, as you might imagine, the process is reversed. The CSS declares all of the styles for the widest browser width by default, and breakpoints are kicked in as the browser narrows, again through media queries.

So which one is better? They each have their pros and cons, of course. A mobile-first approach is likely to result in slightly lower CSS file sizes because many of the styles that would need to be set for wider widths won’t have to be overwritten further down in the file. Think things like floats, which are not the default of any element, so if they’re set for a wider browser by default, they’ll likely need to be unset later down the line.

On the other hand, since older versions of Internet Explorer don’t understand media queries, in a mobile-first implementation users of those browsers will see the mobile version of the site instead of the desktop version. There are JavaScript polyfills that can address this limitation, however they tend to make an already slow browser run even slower.

Which one should you use on your project? That depends on the needs of the project and demands of the client. If you are okay with Internet Explorer seeing a mobile version of the site (or okay with the polyfill issues), then I would recommend mobile-first over desktop-down. I find that it results in cleaner, more manageable CSS overall. However, if the Internet Explorer issue is critical to you, you might want to consider building desktop-down. In this book, we will be using a mobile-first approach, and I will also show you how to use respond.js, the JavaScript that makes it possible for Internet Explorer to understand media queries.

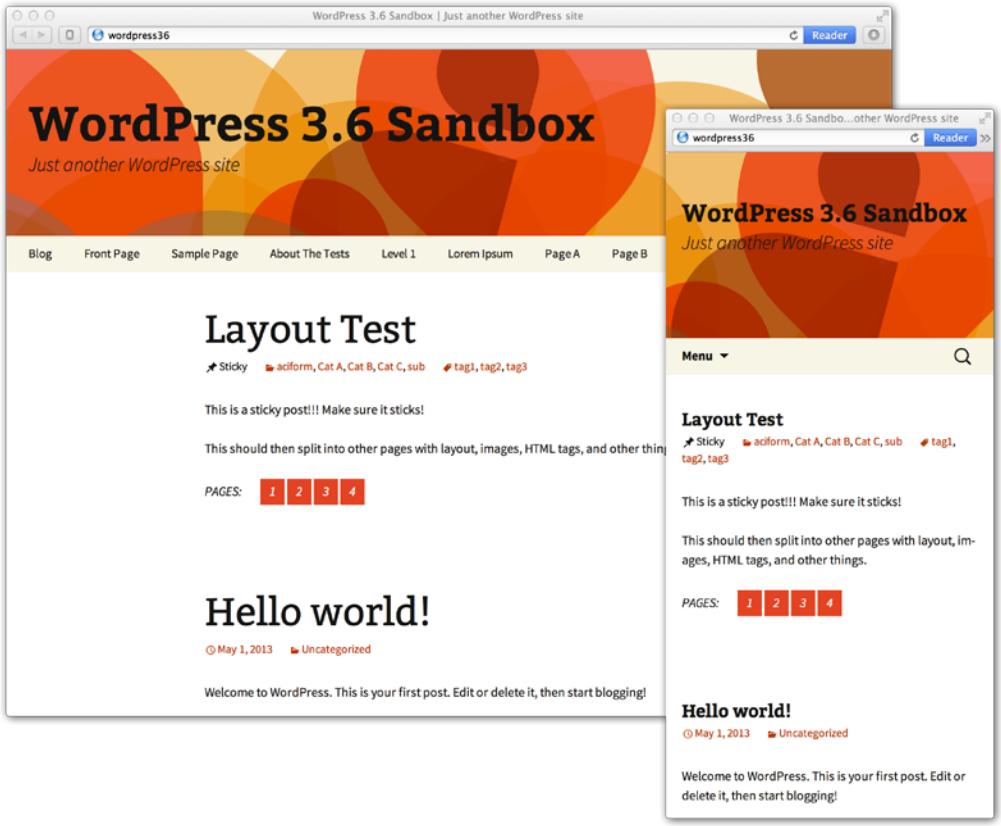


Figure 1.9 Twenty Thirteen is also fully responsive, and far more colorful than prior WordPress default themes.

It's probably safe to assume that at this point, all future WordPress default themes will be responsive as well.

WORKING WITH DEFAULT THEMES

All of the WordPress default themes are available for download on the WordPress theme repository at http://wordpress.org/themes. More than just high-quality, ready-to-use themes that can be put on any WordPress site and look great, their code is meant as a means of learning how to build your own themes. Inside the directory for each of the themes, you'll see all the necessary files required for building your own theme from scratch, and you can copy and paste snippets of code as needed.

From time to time in the course of our responsive WordPress theme project, we will be examining the code from Twenty Eleven, Twenty Twelve and Twenty Thirteen to see how each of these themes handled some of the challenges we come across.

1.3.2 Third Party Responsive WordPress Themes

WordPress is such a popular platform in part because of its rich ecosystem of third party designers and developers who create great plugins and themes for it. These include premium theme developers, who build and sell feature-rich themes on their own sites, and the WordPress free theme repository, which offers over one thousand free themes.

PREMIUM THEME SHOPS

Premium theme shops have certainly jumped on the responsive web design bandwagon in recent years. Popular sites such as WooThemes, the Theme Foundry and Graph Paper Press (figure 1.10), to name only three, offer high-quality WordPress themes on a fee per theme or subscription basis. These themes are often built atop their own theme platforms and offer advanced features and customization not found in many free themes. As part of the theme's price, they usually offer dedicated support to their customers.

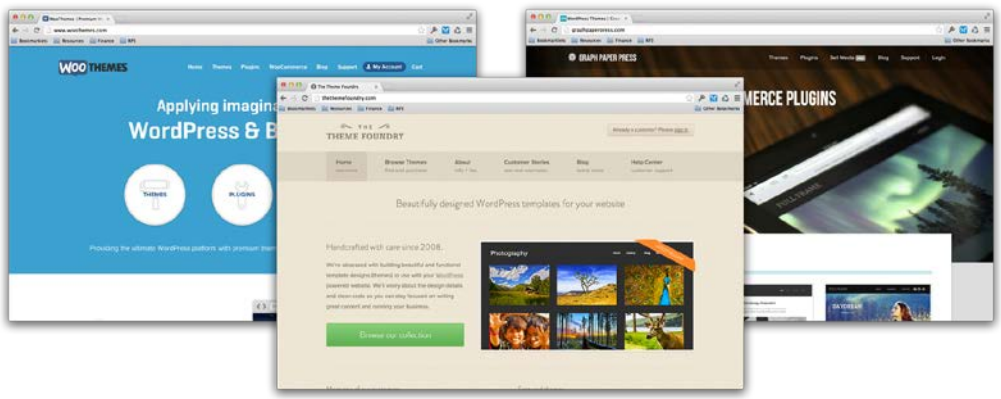


Figure 1.10 Premium theme shops such as WooThemes, the Theme Foundry and Graph Paper Press have begun offering a bevy of responsive WordPress.

In addition to shops that offer their own creations, there are other sites that serve as a marketplace that allow individual theme authors to sell their own themes without the hassle and expense of setting up their own business. Again, responsive themes still make up only a portion of these sites' offerings, but that percentage is growing as responsive web design gains in popularity.

WORDPRESS THEME REPOSITORY

In addition to the default themes we discussed above, there are over 1,700 other themes available in the official WordPress free theme repository from theme designers and developers all over the world. Since the theme repository has been around for a number of years, the vast majority of the themes it contains aren't responsive. However, increasingly, new, responsive themes are being added.

One such theme is called, simply, Responsive(figure 1.11). Responsive is intended to serve as a parent theme on which child themes are built, and comes with a wide selection of user interface elements predefined for use in your website. Much like responsive frameworks such as Foundation by Zurb and Twitter Bootstrap, which we'll look at in part two of this book, Responsive offers a quick jumping off point that does a lot of the basic responsive work for you.

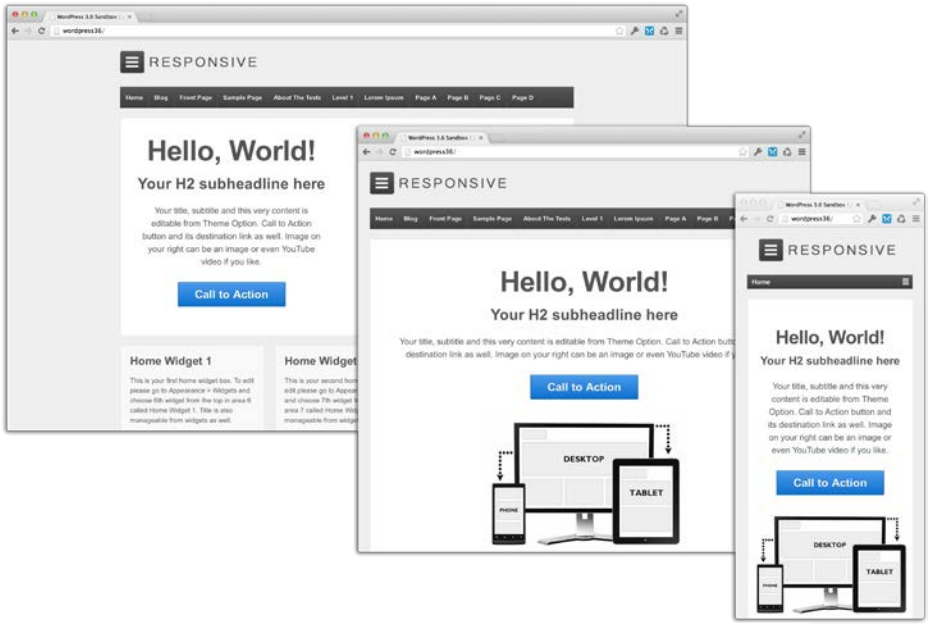


Figure 1.11 Responsive themes such as this are starting to appear in the WordPress theme repository in a big way.

What Are Child Themes?

A child theme is any theme that uses another theme as a template, referred to as the parent. In a parent/child theme structure, special comments in the child theme tell WordPress to look to the parent theme for all the templates and functionality, unless specifically overwritten by the child theme.

Child themes are a good way of customizing the look of your website without having to worry about whether those customizations will be destroyed the next time you go to upgrade your downloaded theme.

In this book, we will be building our theme from the ground up, and won't be using parent/child themes at all. However, all of the principles we'll cover could easily be applied to making a child theme of any parent theme responsive.

If you find a non-responsive theme from the repository that you really like, it is usually possible to retrofit it through the parent/child theme structure. Retrofitting a non-responsive website can be difficult and you should keep in mind that you will almost certainly be building your theme using a desktop-down approach. Also, you probably won't be saving as much time as you think by starting off with an existing parent theme. Often the work of taking something that wasn't designed to be responsive and making it so is as time consuming or more than starting from scratch.

A word of warning regarding free themes, however: If you do a Google search on "free WordPress themes," you're likely to get dozens of links to shady operations and spammy-looking sites. Avoid those. Often the themes that you download from those websites come with embedded spam links and even malware or are pirated copies of a premium theme that someone else worked hard to put together. The WordPress.org theme repository, on the other hand, fully reviews and vets all of the themes submitted to it carefully to make sure, as much as is possible, that every theme you download there is safe and meets certain standards in coding quality. Additionally, many reputable premium theme shops such as WooThemes and Graph Paper Press have a selection of high-quality free themes in addition to their paid offerings.

In short, do your homework before you download any theme from any site. Check to see if the author and/or shop are reputable, if the theme you're considering has had a history of security issues, and if the developer provides support through the WordPress.org support forums. Security is a vast topic that we won't get into much in this book, but it's vitally important because downloading an infected theme from a less-than-reliable source can instantly compromise your entire site.

1.4 Summary

In this chapter, we learned about the problems that responsive web design is trying to solve by presenting a better alternative to mobile-only websites. We also learned that even though responsive web design is a "one website to rule them all" solution, it's still important to keep mobile users foremost in our minds when building our website.

We also learned a bit about WordPress, its history and its popularity as more than a simple blogging platform. We've seen how WordPress theme developers have embraced responsive web design in their own work, and where we can go to find existing responsive WordPress themes.

In our next chapter we'll take a more technical look at both responsive design and WordPress and what makes each of them tick. We'll learn about the CSS3 media queries we introduced here, as well as fluid grids and responsive images. Also, we'll look at WordPress and some of the typical patterns and structures that we'll be working with as we build our responsive theme in part two of this book.

2

The Fundamentals

This chapter covers

- The meta viewport tag
- Fluid grids and CSS media queries
- Responsive images
- Typical WordPress widget and navigation construction
- How WordPress handles images

Responsive websites all have various required pieces, regardless of whether they're driven by WordPress or not. Put the pieces together correctly, and you'll have a website that will expand and contract to meet the needs of whatever browser may come to visit it. But if any piece is out of place or not configured correctly, you can render your site unusable on one or multiple platforms. These elements, including the meta viewport tag, the fluid grid, and CSS media queries, send specific instructions to the browser on how to behave whether the viewport of your browser is smaller than 400 pixels, or larger than 1,200.

And then there's WordPress, which –rightly or wrongly –is known for having a certain "look" that can be attributed to certain design patterns that are common in its themes. This look, which comes from sidebars, widgets and navigation, are so ingrained to the WordPress way of being that the proposal to remove any one of them from a default theme can cause heated debate amongst contributors. Take the Twenty Thirteen theme released in WordPress 3.6, for instance. There was debate about removing the right sidebar because that, given the design, a sidebar doesn't make as much sense as it did in previous default themes. Ultimately the core theme team chose to keep it because it was such an expected part of the WordPress experience, but it was demoted in prominence so that the footer serves as the primary widget area and the sidebar only appears if widgets are assigned to it. In the words of lead developer Mark Jaquith when he introduced the new default theme to

the world, “Twenty Thirteen really prefers a single column layout… but it supports a sidebar, if you really insist.” (http://make.wordpress.org/core/2013/02/18/introducing-twenty-thirteen/) (Figure 2.1)

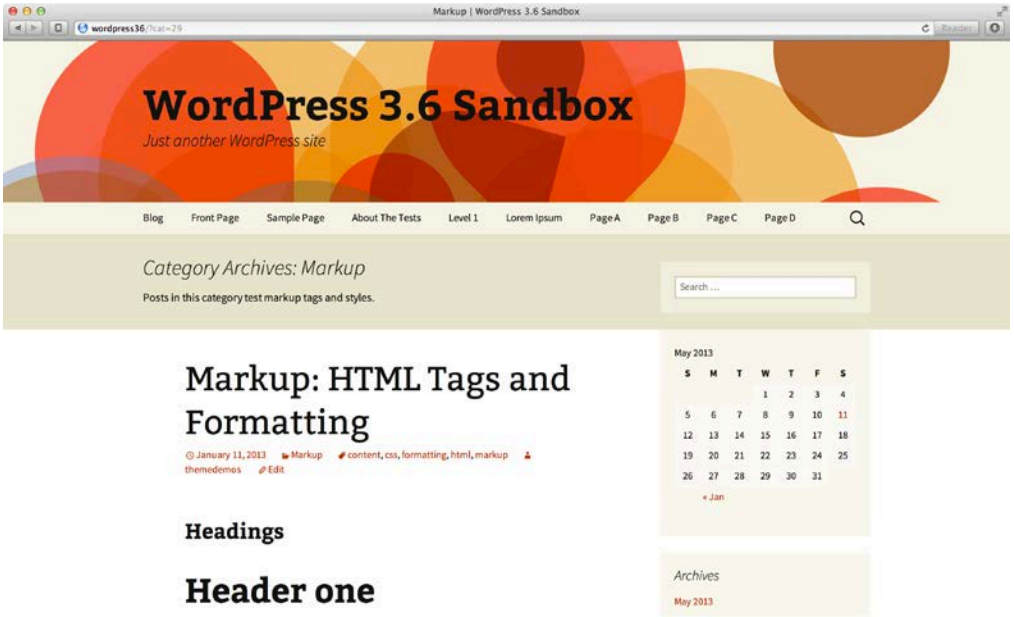


Figure 2.1 There are those who would say that a sidebar in a predominantly single-column theme looks out of place. But sidebars are so typically WordPress that omitting it from a default theme is a step too far for others.

So in this chapter, we’ll be taking a thorough look at the elements that often define the WordPress experience. Once we’ve gotten a handle on these fundamental aspects of responsive web design and WordPress theming, we’ll take them and use them to build our responsive WordPress theme in part two of this book.

2.1 Essential Components of Responsive Web Design

Just as elements form the building blocks of the universe, there are certain elements that form the building blocks of responsive web development. Without these, the website will not behave as expected across the different platforms and devices that may access it. If some of these components are in place, yet others are missing, the website can be rendered unusable on some browsers. I have often seen websites that contain a meta viewport tag, but not CSS3 media queries, leaving me unable to view the website on my smartphone because I could not zoom out to view the content. This happens most often when

developers build a site from a framework without a clear understanding of what all the components are for.

Our goal in this section is to gain a certain basic understanding of what these elements are and how to use them. I won't be going into great detail on any one particular item; to get a more thorough understanding on these topics, please check the bibliography for other books that cover these topics in-depth.

2.1.1 The Meta Viewport Tag

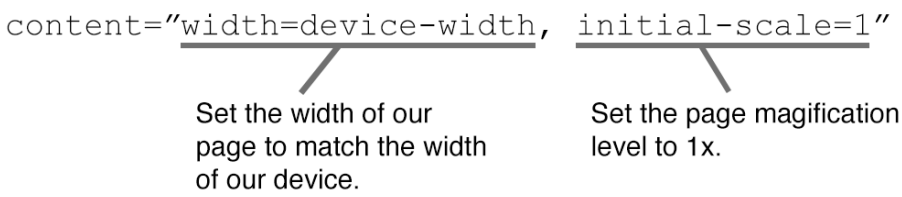
HTML5, despite being the new hotness around the year 2010, isn't as integral in building a responsive website as one might initially think. Most of what we need to make a site responsive is in the CSS, and secondarily the JavaScript. One HTML element we *do* need, however, is a certain `<meta>` tag that will tell our browsers, both mobile and otherwise, the particular magnification setting we need in order to display our website properly.

> **Meta Tags**
>
> Meta tags are elements in your HTML document that describe the contents and characteristics of that document. They can convey information to the web browser, search engines and social media sites. They do not, however, contain any content that renders directly to the viewable area of the browser, and are always placed in the <head> section of the document.

By default, iOS and Android browsers "zoom out" on a web page when it's initially loaded. To prevent this, we can tell the web browser to *not* zoom out for our web pages by adding the following tag in the `<head>` of our document:

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

What is this tag telling the browser, exactly?



The first parameter in our content, `width=device-width`, tells the browser that the width of our document is the width of the device, no more and no less. So instead of the browser

pretending that (in the case of an iPhone in portrait orientation) it is 980 pixels wide in its initial, zoomed out state, it will always consider the page width to be the actual width of the device (in our example, 320 pixels).

HiDPI Displays

HiDPI displays (or, in Apple lingo, Retina displays) do not influence the device-width number. For example, web pages displayed on both the iPhone 3Gs (a non-Retina device) and the iPhone 4 (which *is* Retina) will appear to be the same size with the meta viewport tag set.

Although this has no bearing on the device width, it brings up its own considerations when dealing with images that must be taken into account when building a responsive website. We'll take a look at options for this situation in chapter ten of this book.

The second parameter, `initial-scale=1`, tells the browser what to use as the zoom factor, namely 1. This means that the document will be displayed as its actual size upon initial load, rather than some zoom factor less than 1.

Some documentation will advocate the use of a third parameter, `maximum-scale=1`, which will prevent the user from being able to zoom in on any portion of the page. But this can be problematic where accessibility is concerned because even the best crafted responsive website may not be sufficiently readable to people with low vision. Preventing any means for them to magnify the text (or any other element of the page) would make the website more difficult to use. Besides, it offers no benefit in the responsiveness of the site nor in the display upon initial load.

2.1.2 Fluid Grids

Now that we've told the browser that this site is responsive, it's time to start structuring our site responsively. In the early days of the web, websites were built to accommodate a certain minimal accepted standard monitor size. This minimum kept getting larger as the years went on, but for a long time it was assumed that most monitors surfing the web were probably going to be 1,024 pixels wide by 768 pixels high. And for a while, that suited everyone just fine.

In fact, web designers started building grid systems that specifically addressed the 1024x768 "default". These grid systems were then used by developers to neatly arrange content on the screen in a way that was reusable on many websites. Because of its divisibility by a wide variety of factors, 960 pixels became a common width for grid systems. This flexibility gave the designer many choices in how many columns to use. However, because most of these grid systems were built in CSS using absolute pixels, there was still the problem that if a user narrowed his browser beneath a certain width, a horizontal scroll bar would appear at the bottom of the window and the content would appear cut off.

What's a Grid System?

A grid system is a tool for arranging content on a page in a structure that's visually organized. Web grids typically divide the page into 12, 16 or 24 columns (plus reasonable margins) and all of the text, images and other content on that page will be arranged in such a way as to fit into those columns or spans of those columns. Grid systems are one of the many tools that print designers brought with them as they made the transition to web design.

The 960 Grid System (http://960.gs), one of the more popular CSS grid systems on the web, has a showcase of websites that use it, one of which is the website for the American Institute of Architects (http://aia.org) (figure 2.2).

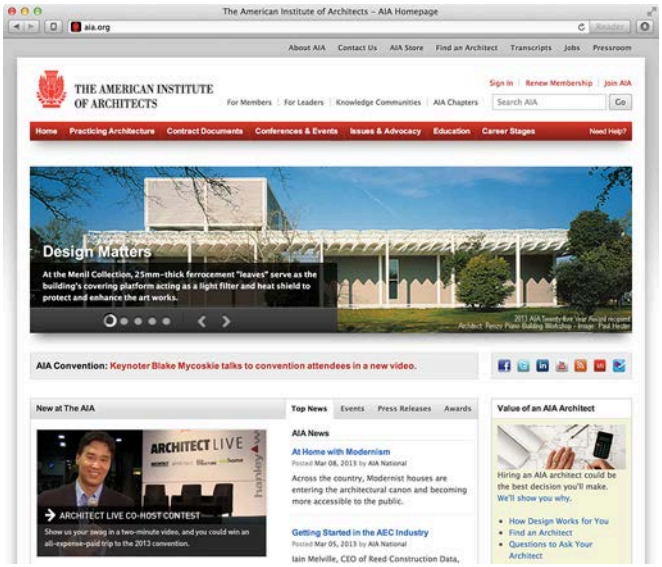


Figure 2.2 The website for the American Institute of Architects is built using the popular 960 Grid System.

While the website looks fine on monitors that are at least 1,024 pixels wide, content is cut off when the browser goes much below that width (figure 2.3).

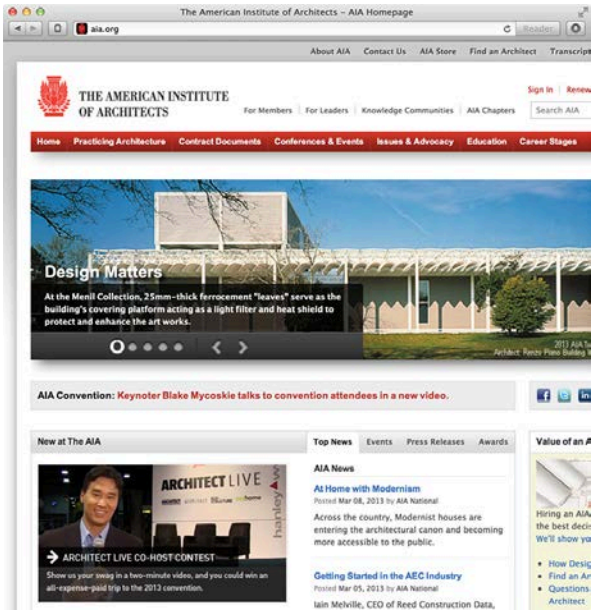


Figure 2.3 If we decrease the width of our browser, the website's content gets cut off.

With the plethora of mobile devices and tablets of various sizes in active use, we can no longer assume that the most modern browser that will visit our website will be of a certain minimum size. What we need is a flexible grid system that can expand and contract with the browser.

THE RESPONSIVE FORMULA

Enter the responsive formula. This formula, in of itself, won't make your site responsive, but it is a necessary first step.

```
target ÷ context = result
```

This formula actually predates the debut of responsive web design. It is, in fact, the necessary means for creating any sort of fluid grid. Where your target and context are set in absolute pixels (perhaps drawn from the dimensions found in a Photoshop comp file), the result will be a proportion. Multiply that proportion by 100, and you now have a percentage value that you can use in your CSS.

Take a typical Photoshop comp file (a layered Photoshop file that shows the visual design of a web page, sometimes also referred to as a mockup) given to us by the designer, such as the rather simplistic example in figure 2.4. Fortunately, they gave us all the specs for a two column (main column plus sidebar) written out in the comp, like so:

- Both columns have a 16 pixel border
- Both columns have 16 pixels of padding, both vertically and horizontally
- The main column should be 630 pixels wide, *including* the border and padding
- The sidebar should be 305 pixels wide, also including the border and padding
- There should be a 25 pixel gutter between the columns
- The total width of the page is 960 pixels wide

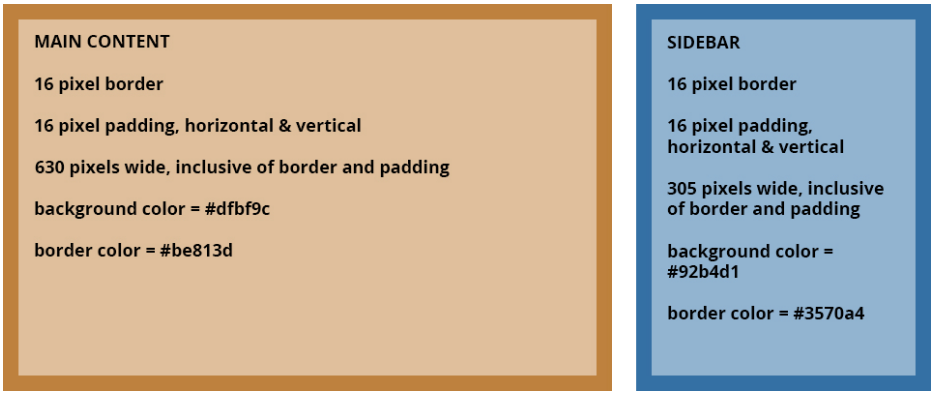


Figure 2.4 Our designer was very helpful to include all the necessary specifications for this page directly in the comp.

630, 305, 25 and 16 are our targets for our main column, sidebar, gutter and padding, respectively. The context for our horizontal values is 960, the total width of our content area. To express this in a fluid way, we would calculate the desired percentages based on the values our designer has given us:

```
630 ÷ 960 = 0.6526 = 65.625%
305 ÷ 960 = 0.31770833333333 = 31.770833333333%
25 ÷ 960 = 0.02604166666667 = 2.604166666667%
16 ÷ 960 = 0.01666666666667 = 1.666666666667%
```

We would then take these calculated percentages, and enter them into our stylesheet as in Listing 2.1.

Listing 2.1Defining Fluid Columns in CSS

```
.main-column, .right-column { #A
     border: 16px solid; #B
     box-sizing: border-box;
     float: left;
     padding: 16px 1.666666666667%; /* 16px / 960px */ #C
}
```

```
.main-column {
     background-color: #dfbf9c;
     border-color: #be813d;
     margin-right: 2.604166666667%; /* 25px / 960px */#D
     width: 65.625%; /* 630px / 960px */
}
.right-column {
     background-color: #92b4d1;
     border-color: #3570a4;
     width: 31.770833333333%; /* 305px / 960px */
}
```

#A Everything that is the same for both columns can go here. There's no sense repeating yourself (which just adds to the page weight and makes it more difficult to change values later).
#B We don't *really* want our borders to grow and shrink depending on the browser width. In general, we want borders to stay a fixed size, so let's just use pixels here.
#C In order to help other developers (not to mention your future self) who might work on this code understand the irrational percentages your formula may come up with, it's very helpful to put the target divided by context numbers in comments next to the property. Also, it usually wouldn't sense to express the top and bottom padding value in terms of a percentage of the width, so we'll leave that in pixels as well, for now.
#D Determining flexible margin and padding values can be tricky. See "Fluid Margins and Paddings" below for a full explanation.

Don't be scared by the lengthy decimals this formula can produce. After all, you're feeding these values into a computer that handles them with ease. Each browser rounds numbers to its own precision, and if you try to round too much you could end up columns that don't quite fit side-by-side on your canvas.

And if you're thinking about how many times you might end up having to calculate this formula, don't worry about that either. In chapter five, I'm going to introduce you to tools that will do much of the heavy lifting for of you. But for now, it's important to learn and understand this formula, because it is fundamental to building a strong responsive site.

FLUID MARGINS AND PADDINGS

Calculating fluid margins and paddings takes a bit of practice, and made a little more complicated with the rise in popularity of the `box-sizing: border-box` property and value. When Ethan Marcotte wrote *Responsive Web Design*, he stated (quite accurately) that:

1. When setting flexible **margins** on an element, your context is the width of the element's container.
2. When setting flexible **padding** on an element, your context is *the width of the element itself*. Which makes sense, if you think about the box model: we're describing the padding in relationship to the width of the box itself. (*Responsive Web Design,* Marcotte, p.35)

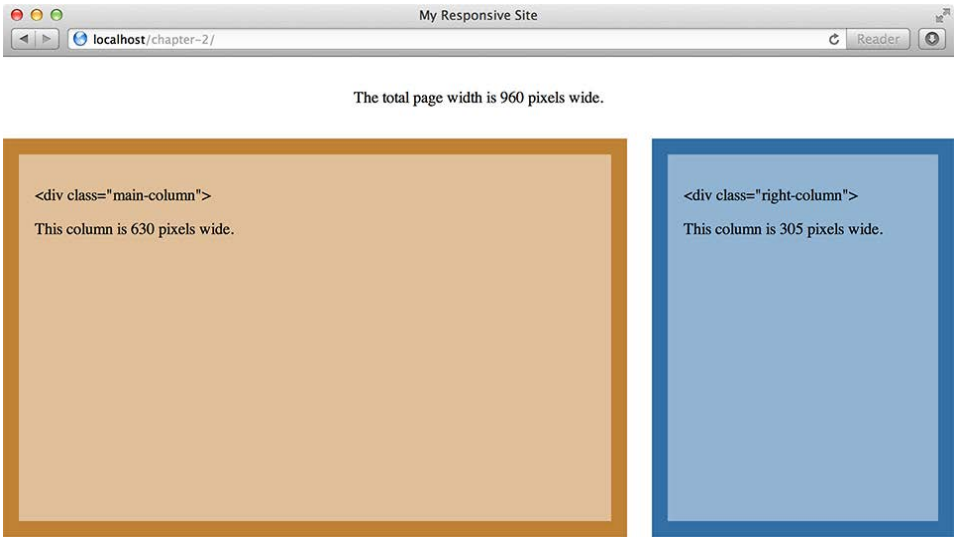
But the `box-sizing: border-box` declaration turns the traditional CSS box model on its head. Instead of the `width` of an element being only that portion taken up by the content, we can tell CSS to apply the `width` to the whole element, clear out to its border. This has some profound implications in building our responsive grid:

1. We no longer have to subtract our desired padding and border size from the desired overall size of our container. CSS will take care of that for us.
2. The flexible padding we're looking for now uses the element's container as its context.

We'll examine the promises and pitfalls of flexible margins and paddings in relation to the `box-sizing: border-box` declaration when we build our responsive theme in part 2 of this book.

OUR FLUID PAGE

Now if we were to look at our newly constructed website with its percentage-based dimensions, we'll see that the columns neatly expand and contract along with the width of the browser (figure 2.5).



#A I'm using a snippet of JavaScript to tell me exactly how many pixels wide my viewport and columns are. Right now, it's set to 960 pixels, which is the context on which I've based my percentages.
#B As a result, the main column is exactly 630 pixels, which was my original target.
#C The same goes for my sidebar, whose width at the moment is the same as our target.

Figure 2.5 If we set our browser width so that it matches our context, we see that the columns are the exact size we targeted.

If we take the browser window and make it span the entire monitor, the columns and the margin between them grow in the same proportions as the original 960 pixel view (figure 2.6).

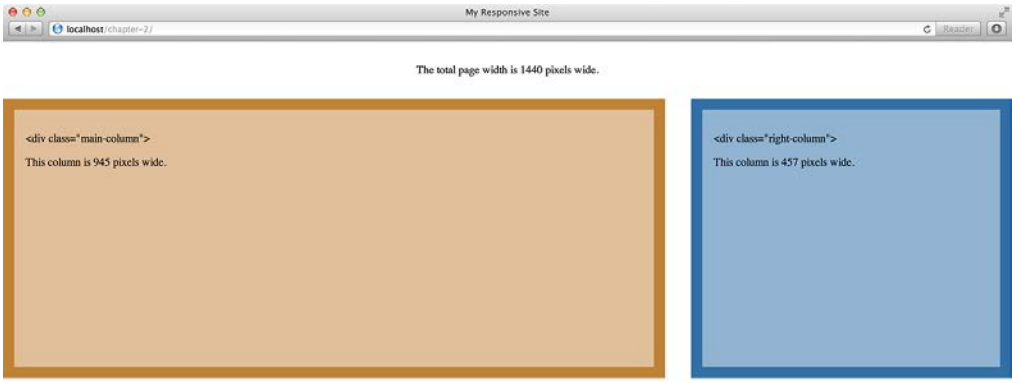


Figure 2.6 Even when the browser grows to fill the entire monitor, in this case 1,440 pixels, the columns remain in the same proportions as before.

We can use a little math to verify this. If we take our responsive formula put in the new values we see from the JavaScript, we come out with the following result:

```
945 pixels ÷ 1,440 pixels = 0.65625 = 65.625%
457 pixels ÷ 1,440 pixels = 0.31736111111111 = 31.736111111111%
```

The sidebar's value may be slightly off from our original calculations, but that's just rounding at work. This is all the more reason to let the browser do the rounding for you.

When shrinking the width of the browser, we see the columns narrow in the same proportions as our wider widths (figure 2.7).

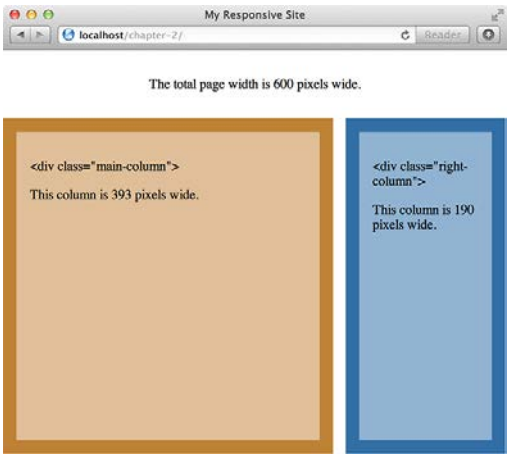


Figure 2.7 We've narrowed the browser down to 600 pixels, yet the columns are still in the same proportion as the previous two figures.

Again, some basic math will validate our findings:

```
393 pixels ÷ 600 pixels = 0.655 = 65.5%
190 pixels ÷ 600 pixels = 0.31666666666667 = 31.666666666667%
```

Close enough.

Fluid grids alone do not make for a responsive website. At some point, our columns will end up being too narrow to be of much use for anything. To really go full out responsive, we need to take this to the next level and bring in our new best friend: CSS3 media queries.

2.1.3 CSS3 Media Queries

The predecessor for CSS media queries first entered the W3C specification as "media types" in CSS 2.1. Originally, they were meant solely to distinguish between different content delivery systems: `screen` for your standard computer monitor, `print` for printing out a web page onto paper, etc. But in the CSS3 specification, media queries took on a new role: to distinguish between any number of different factors, including screen width and pixel density (resolution). When coupled with the `<meta name="viewport" … >` tag we talked about at the beginning of the chapter, they form a powerful combination that can alter the styling rules to your site based solely on the width of the browser.

To specify a media query, we must first test for the media type and then combine that with the query itself. While there are a variety of conditions you can test for with media queries, by far the most common application is testing for viewport width. When we set a media query to test for a particular width, we refer to it as our "breakpoint."

A Note About Breakpoints

When developers first started working with responsive web design, breakpoints were often set based on the viewports of the most popular devices in the "mobile" space, namely the iPhone and the iPad. As a result, you'll still see a lot of breakpoints set at 320 pixels, 768 pixels, etc. But as increasing numbers of Android devices came to market and grew in popularity, the variety of viewport widths became increasingly problematic.

Instead of basing your breakpoints on the widths of any particular device, they should be based on where it makes sense in the design. That is to say, where the design of a page begins to break down should drive your breakpoint decisions. Just where are those, exactly? Sometimes, it takes a little trial and error with building out the design. We'll look extensively on how to set breakpoints when we build our responsive WordPress theme in part two of this book.

Let's revisit our code from Listing 2.1. Suppose our designers had determined that at viewport widths narrower than 600 pixels, our columns were to stack one atop another, with the main content column on top. Only on viewport widths at or greater than 600 pixels

should the columns appear side-by-side. Minding the fact that we want our site to be mobile-first, we would rewrite our CSS as in Listing 2.2.

Listing 2.2 A Typical CSS Media Query

```
.main-column, .right-column {
     border: 1em solid;
     box-sizing: border-box;
     float: none; #A
     min-height: 15em;
     padding: 1em 1.666666666667%; /* 16px / 960px */
}
.main-column {
     background-color: #dfbf9c;
     border-color: #be813d;
     margin-right: 0;
     width: 100%;
}
.right-column {
     background-color: #92b4d1;
     border-color: #3570a4;
     width: 100%;
}
@media only screen and (min-width: 600px) { #B
     .main-column, .right-column {
            float: left;
            min-height: 25em;
     .main-column {
            margin-right: 2.604166666667%; /* 25px / 960px */
            width: 65.625%; /* 630px / 960px */
     }
     .right-column {
            width: 31.770833333333%; /* 305px / 960px */
     }
}
```

#A: We actually don't *need* to define our float property here, since most likely the element will already be set to the default "float: none;". However, I'm including it here for illustration purposes.
#B: What's with the "only screen" in our media query? Older versions of Internet Explorer choke on media queries (I know, you're so surprised). By adding the word "only" we're causing IE to ignore the media query altogether. When you're developing mobile-first, versions of IE prior to 9 will see the mobile version of your website unless you use some sort of polyfill.

When we reload the browser and narrow it below 600 pixels, our grid now becomes a stack, as in figure 2.8.

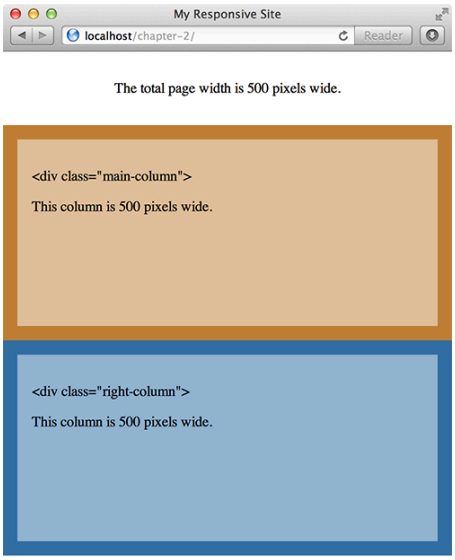


Figure 2.8 At 500 pixels, our browser window is now too narrow for side-by-side columns to work well. We've switched to our "mobile" view with the two columns stacked one upon the other instead.

There are a few things to note in our example. First, you only need to define the properties that will change at the breakpoint you've defined. So if we set our main content area's background color to be orange in the default state, it'll stay orange when we hit our breakpoint unless we've redefined it there. Also, *any* CSS property can change at a given breakpoint. Want a different background image for mobile screens versus large displays? No problem. Font sizes, font families, widths, heights, display properties… it's all up for grabs.

Some websites use media queries to achieve some clever effects (figure 2.9).

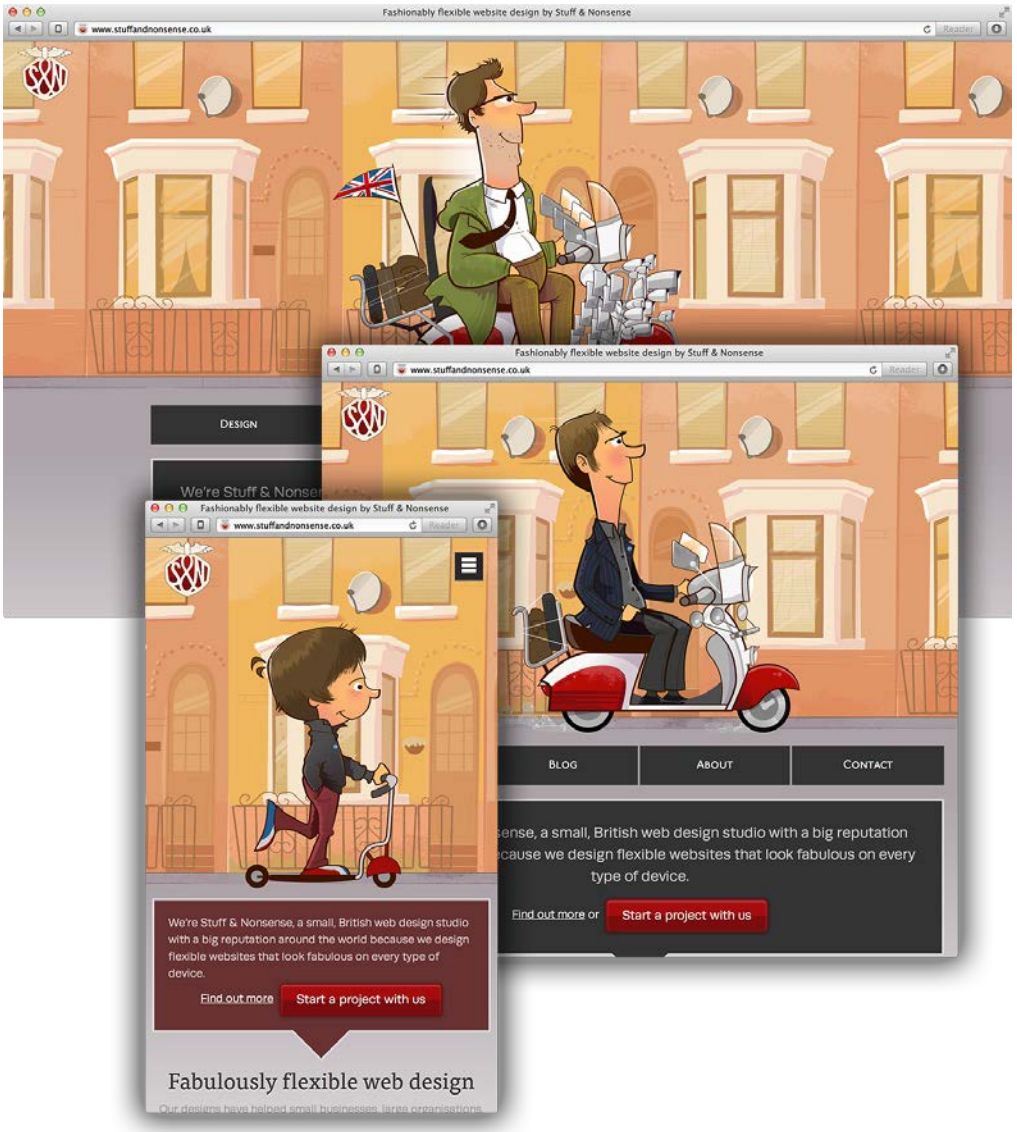


Figure 2.9 British web design firm Stuff and Nonsense uses media queries to change the feature image (via the background-image CSS property) on their home page.

2.1.4 Responsive Media

How best to deal with images and video in a responsive way is still in many ways an unsettled issue. The developer has many considerations when deciding how best to place images on a web page, or even when to do so:

- Mobile data speeds and bandwidth caps can make downloading heavy, image-rich web pages expensive, in both time and in the literal money way
- Images must be able to scale if they are going to be part of a fluid grid
- Scaling images either too large or too small can make the image look either pixelated, or too small to discern what the image is

In creating responsive images on our web page, we first have to unlearn a little of what we have learned. Once upon a time, a good front-end developer would never think to leave out the `width` and `height` attributes from the `<img …>` tag. By defining those attributes, we were telling the browser (which was likely on a painfully slow dial up connection) what the dimensions of our image were so that it could make room for it on the page. But times have changed. Connections (even mobile ones) are faster today than the dial up days of ten and fifteen years ago. Computer processing power has also increased, and with it, browsers' ability to calculate the space required by an image.

But unfortunately, WordPress didn't get the memo. Many of the tools it uses to display images on the page involve automatically inserting the `width` and `height` attributes, which are values in absolute pixels and thus not inherently responsive. Fortunately, we can override these controls by inserting a few lines of CSS:

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

By setting our image's `max-width` to `100%`, we have told it to never go beyond the boundaries of its parent element, which presumably will scale with the fluid grid we defined earlier. And by setting the `height` to `auto`, the image will scale proportionately and not become distorted.

Let's see how that works by adding a shot of the Hollywood sign to a web page. In figure 2.10, we see that our image, while lovely, is too big to fit in the container because the `width` and `height` attributes are hard set in the `<img …>` tag.

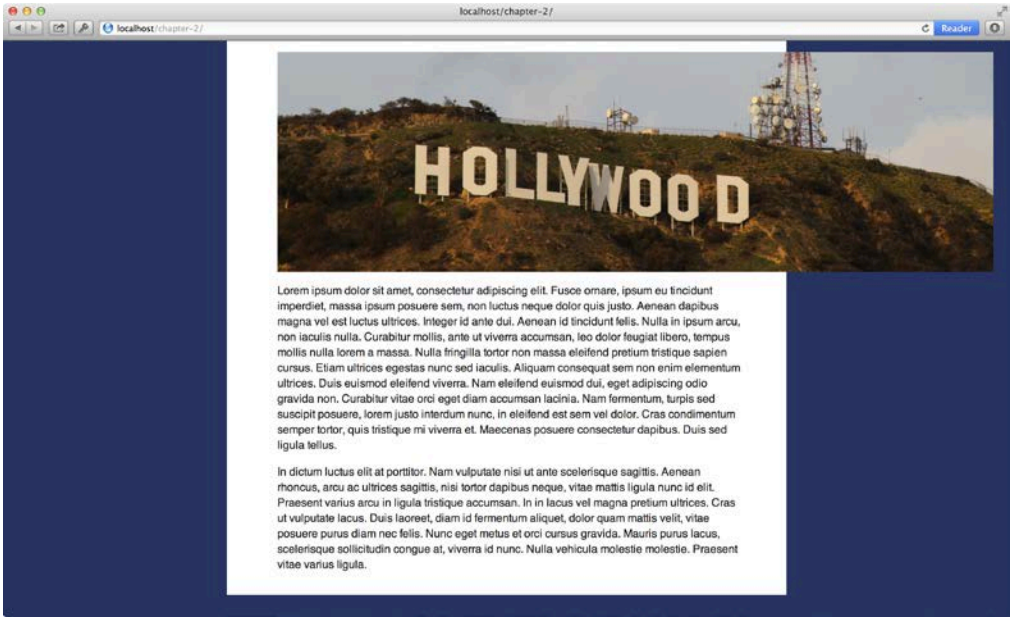


Figure 2.10 Hooray for Hollywood! But the sign is bursting out of its seams! (Photo by Brian Finifter.)

But if we scale down the width of our image via CSS with the following declaration:

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

we get a nice fit width-wise, but the web browser is still reading, and obeying, the `height` attribute we have inserted (figure 2.11).

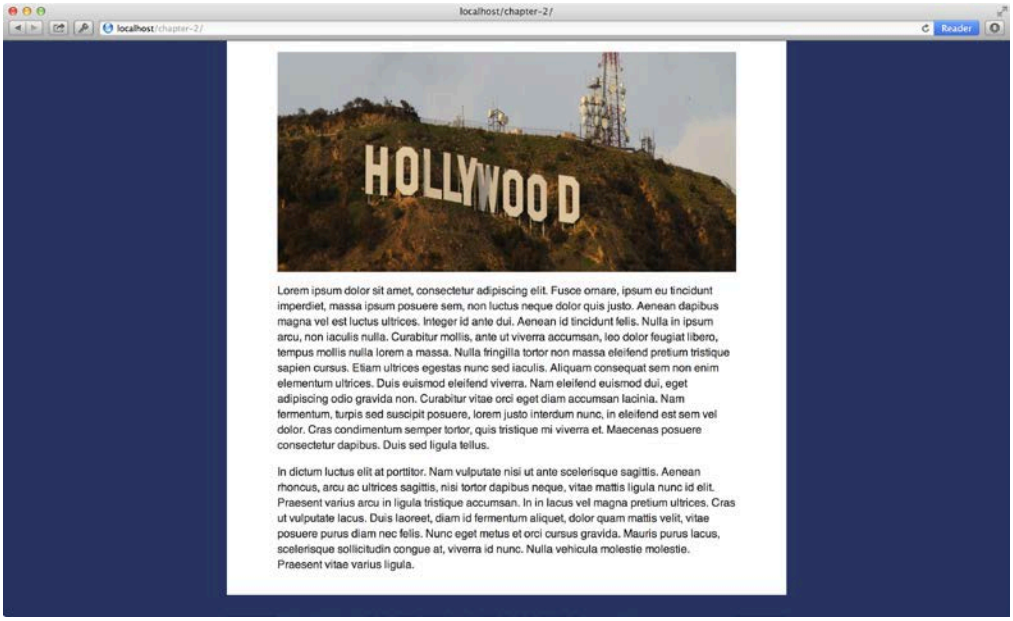


Figure 2.11 Our sign now fits the container, but it's looking just a little squished.

Changing our CSS to:

```
.header img {
  max-width: 100%;
  height: auto;
}
```

overrides the `height` attribute as well, and scales the sign to its proper proportions (figure 2.12).

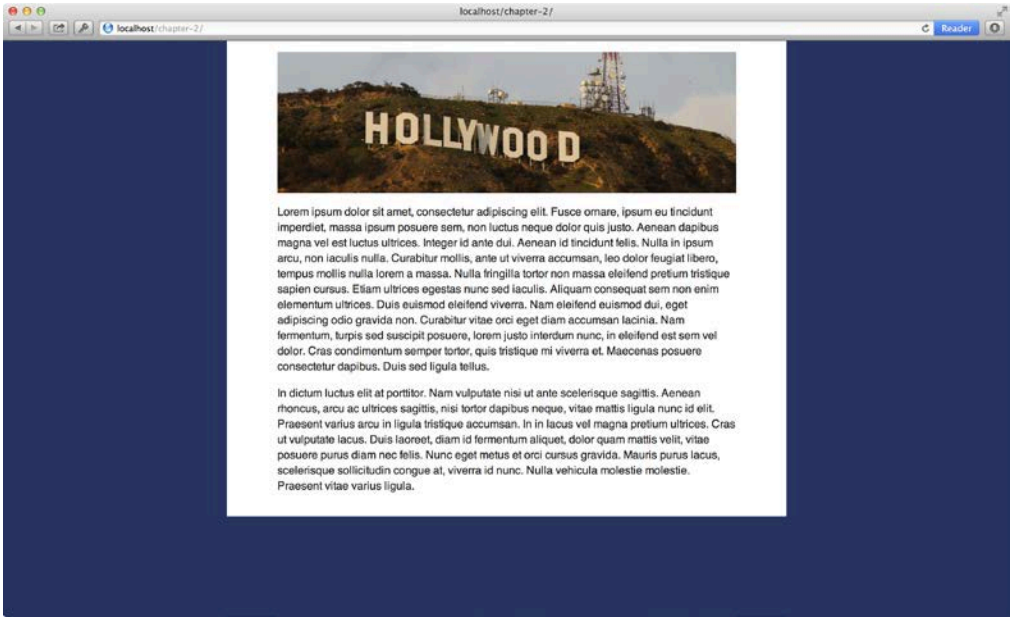


Figure 2.12 Our sign is now in its proper proportions, and fits in its container as well.

As we build our responsive WordPress theme in part two of this book, we will take a look at more techniques we can use to increase the responsiveness of our images.

2.2 Characteristics of a WordPress Theme

In the previous section, we looked at the elements that are fundamental to responsive websites. Omitting any one of these elements puts the site at risk of not only not being responsive, but also perhaps being unusable. WordPress themes, however, while often tending to share common traits, can come in a wide variety of configurations. In this section, we will look at certain elements that are common to WordPress themes, but are by no means required. Many, if not most, WordPress sites contain these features, but can easily exclude one or more, as in figure 2.13.

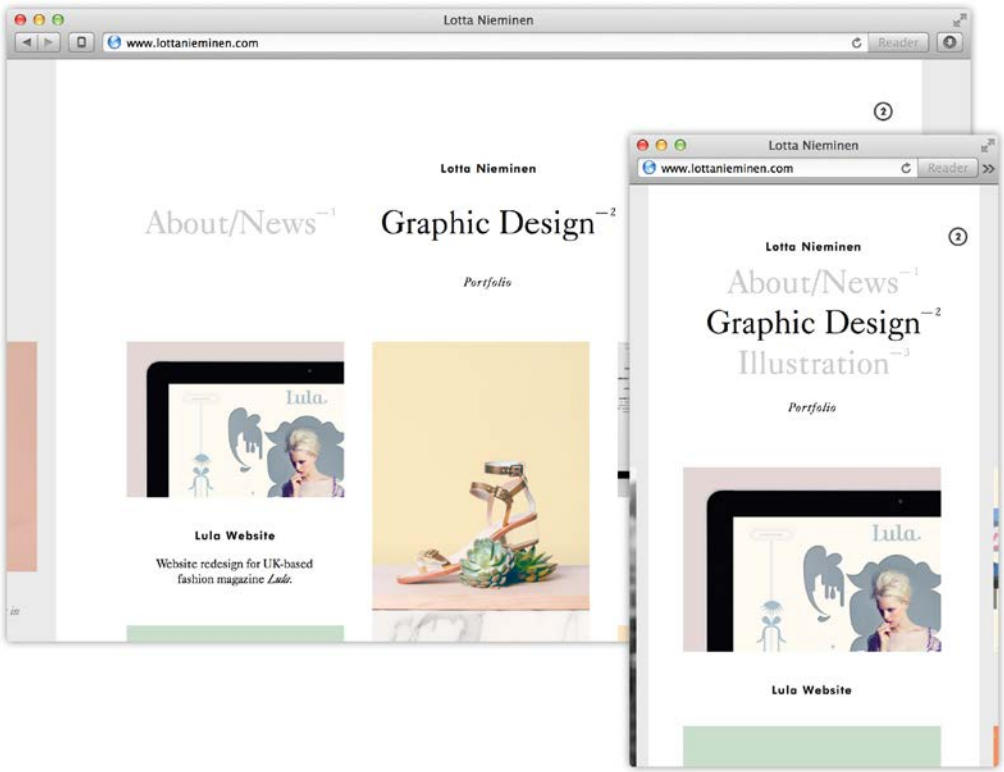


Figure 2.13 Despite its unconventional navigation and lack if sidebars, graphic designer Lotta Nieminen's online portfolio is still a responsive WordPress site.

2.2.1 WordPress Theme Files

In order for a WordPress theme to be recognized and usable by the system, it must have at least two key files:

- **style.css**, inside which certain identifying information is contained
- **index.php**, the default template file that serves as the "template file of last resort"

There are many other different types of template files that we will be coming across as we build our theme, including **header.php**, **footer.php**, **sidebar.php**, **single.php** and **archive.php**. You are likely to see these files in any theme that you download. You are also bound to find a file called **functions.php**, which is where we can define certain custom functionality that applies to the whole theme.

Inside these theme files go our HTML markup as well as special functions, called template tags, which perform various tasks like true/false tests (known as conditional tags) or information retrieval (such as the blog's name or tagline). As we build our WordPress theme

in part two, we'll look at many useful template tags, including ones that can help with mobile device detection.We will learn more about the files that go into creating a WordPress theme in chapter six.

2.2.2 Widgets and Sidebars

Widgets are a key feature of WordPress and containers for them can be found, in one form or another, on nearly every general-use WordPress theme developed for mass distribution. You are probably familiar with the typical widget configuration, often in a sidebar that runs along the side of the page in a smaller column than the main content. Sidebars could be on the right or the left, and occasionally were on both. The theme Coraline by Automattic displays a typical sidebar in figure 2.14.

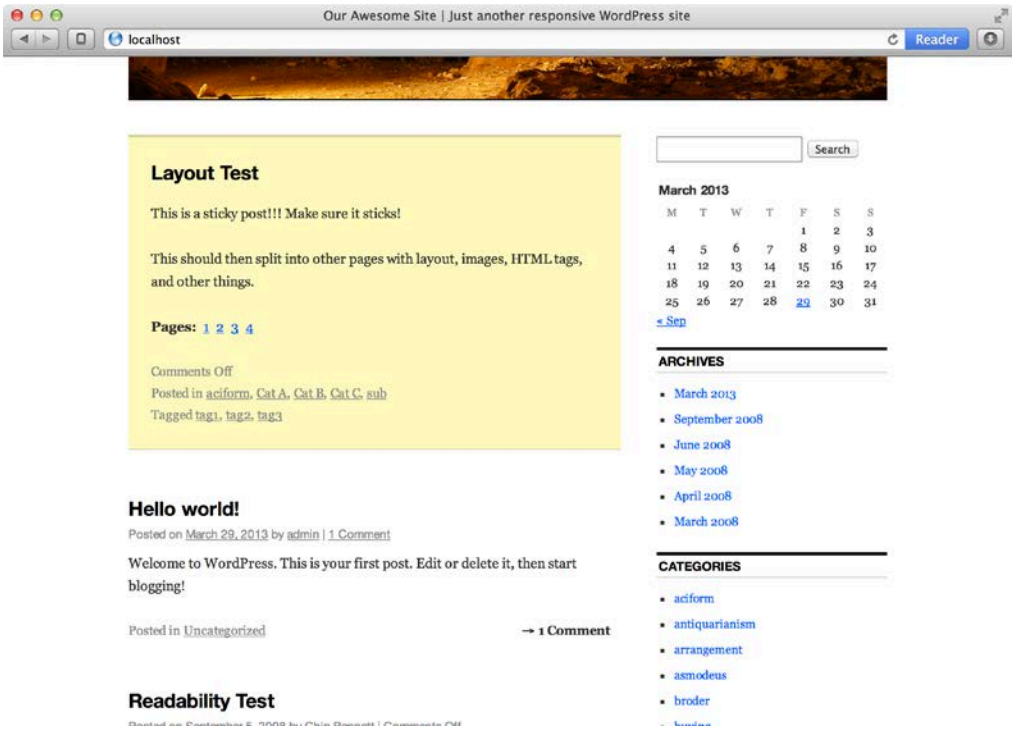


Figure 2.14 In this typical sidebar layout, widgets are displayed in a column along the right side of the main content.

Since this pattern was so common, the term "sidebar" has become synonymous with any space that contained widgets, even though increasingly widgets could be found in places other than a traditional sidebar. In the WordPress 3.0 default theme Twenty Ten, for example, the footer alone had four different places where widgets could be placed. For that

reason, it's more accurate to refer to these places as "widget areas," rather than sidebars. Yet the function to create a widget area in WordPress remains `register_sidebar()` (typically invoked in the theme's functions.php file)and the template tag to display one is `dynamic_sidebar()` (invoked wherever the widget area is to appear in the theme templates).

Like most aspects of WordPress, widgets have a certain set of defaults that make for a common set of markup in many WordPress themes. These defaults have overrides, of course, but the typical widget area when called by the `dynamic_sidebar()`template taglooks something like listing 2.3.

Listing 2.3 Typical Widget Area Markup

```
<li id="search-2" class="widget widget_search"> #A
<formrole="search" method="get" id="searchform" class="searchform"
action="http://localhost/">
<div>
<labelclass="screen-reader-text" for="s">Search for:</label>
<inputtype="text" value="" name="s" id="s" />
<inputtype="submit" id="searchsubmit" value="Search" />
</div>
</form>
</li>
<li id="archives-2" class="widget widget_archive">
<h2 class="widgettitle">Archives</h2> #B
<ul>
<li><ahref='http://localhost/?m=201303' title='March 2013'>March
2013</a></li>
<li><ahref='http://localhost/?m=200809' title='September 2008'>September
2008</a></li>
<li><ahref='http://localhost/?m=200806' title='June 2008'>June
2008</a></li>
<li><ahref='http://localhost/?m=200805' title='May 2008'>May 2008</a></li>
<li><ahref='http://localhost/?m=200804' title='April 2008'>April
2008</a></li>
<li><ahref='http://localhost/?m=200803' title='March 2008'>March
2008</a></li>
</ul>
</li>
```

#A These list items would be enclosed in an unordered list (<ul>), coded out separately in the template and not included in the content returned by the dynamic_sidebar() function.
#B This widget is shown displaying a title. The exact markup for the title is defined when the widget area is created.

2.2.3 Menus

Beginning in version 3.0, WordPress has had the ability to create custom navigation menus that were manageable through the administration dashboard. This was a major leap forward in the ease-of-use of WordPress because it meant that you no longer needed to edit code in order to create or change navigation menu options. Instead, all you had to do was to add and move around your menu options in a graphical interface (figure 2.15).

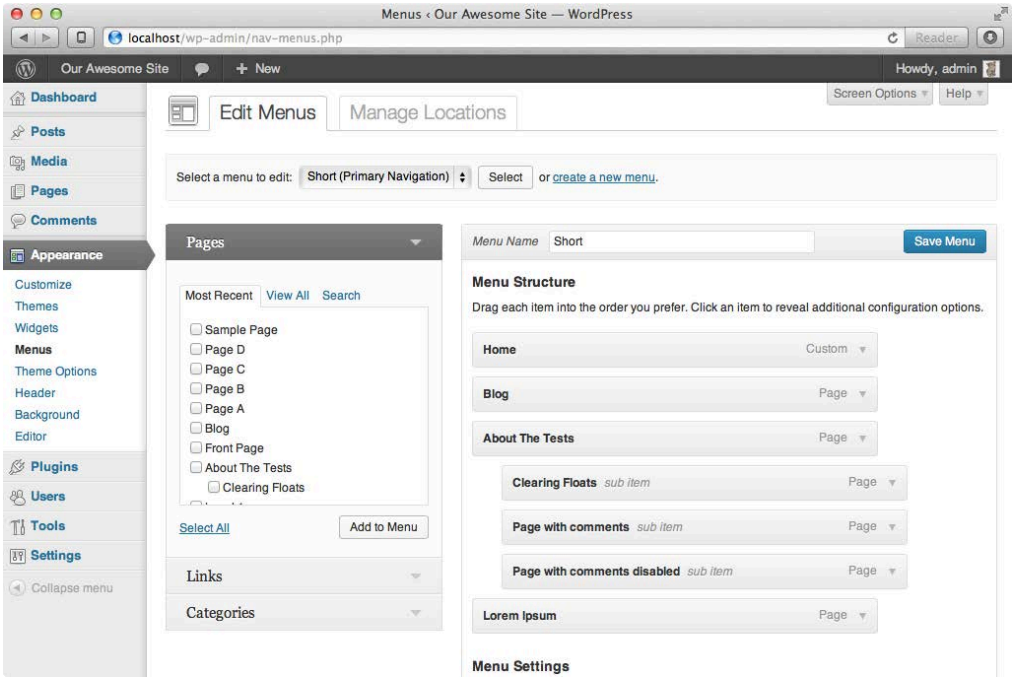


Figure 2.15 The WordPress administration interface allows you to build as many menus as necessary just by dragging and dropping the menu items in the desired ordered and assigning the menu to a location.

Similar to our widget areas, we can add menu locations to our theme with a combination of the `register_nav_menu()` function in our functions.php file, along with the template tag`wp_nav_menu()` in our theme's template. When called with its default options, `wp_nav_menu()`will generatemarkup that resembles listing 2.4.

Listing 2.4 Default Navigation Menu Markup

```
<div class="menu-header">
<ul id="menu-short" class="menu">
<li id="menu-item-828" class="menu-item menu-item-type-custom menu-item-
object-custom menu-item-828"><a
href="http://wpthemetestdata.wordpress.com/">Home</a></li>
<li id="menu-item-874" class="menu-item menu-item-type-post_type menu-item-
object-page menu-item-874"><a
href="http://localhost/?page_id=703">Blog</a></li>
<li id="menu-item-875" class="menu-item menu-item-type-post_type menu-item-
object-page menu-item-875"><a href="http://localhost/?page_id=832">About
The Tests</a>
<ul class="sub-menu">
<li id="menu-item-876" class="menu-item menu-item-type-post_type menu-item-
object-page menu-item-876"><a href="http://localhost/?page_id=501">Clearing
Floats</a></li>
```

```
<li id="menu-item-877" class="menu-item menu-item-type-post_type menu-item-
object-page menu-item-877"><a href="http://localhost/?page_id=155">Page
with comments</a></li>
</ul>
</li>
<li id="menu-item-879" class="menu-item menu-item-type-post_type menu-item-
object-page menu-item-879"><a href="http://localhost/?page_id=146">Lorem
Ipsum</a></li>
</ul>
</div>
```

A common design pattern, in WordPress and elsewhere, is to take this HTML structure and style it using CSS into a horizontal menu bar across the top of the page in the header. The challenge comes in how to make such a structure responsive. Obviously a menu that stretches the width of a browser window will have some issues when that browser window shrinks down to mobile sizes. Fortunately, there are several options on both the client and server sides we can explore in chapter eight to help us with this.

2.2.4 Media Library

Media items are a special default post type that are often images or another form of binary (that is, non-text) information. Technically, in WordPress lingo, these items are referred to as “attachments” because they are “attached” to traditional posts and pages, primarily through the “Add Media” button in the Add New Post/Page and Edit Post/Page administration pages. And since they are their own post type, they have an administration screen all to their own, show in figure 2.16.

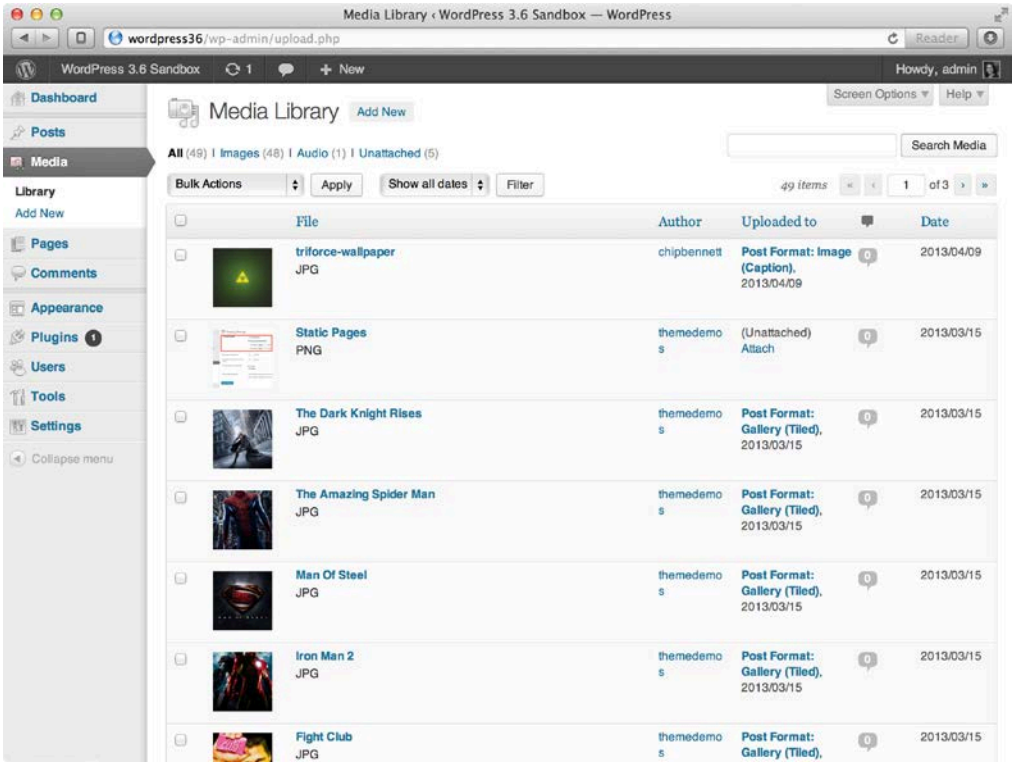


Figure 2.16 WordPress has a special page just for managing images (and other non-text content).

The most common media item is, of course, images. WordPress has a variety of ways in which it natively manages and displays images on the website, including some "special" image situations that get extra treatment. One "gotcha" you really need to worry about in a responsive environment is WordPress' tendency to specify the `width` and `height` attributes in the `<img …>` tags it generates. In order to keep our images fluid and responsive, we can override these attributes in our CSS, but we must remember to set *both* our `width` and `height` properties. One without the other leads to some oddly proportioned images, as we saw in our Hollywood sign example.

FEATURED IMAGES

Featured images are one of the special uses of media in WordPress. It is a key image that is associated with a post in a special way. These images are attached to the post through a special metabox on the post edit page, and are *not* included directly in the content editor.

Featured Image or Post Thumbnail?

When first introduced in WordPress 2.9, "Featured Images" were referred to as "Post Thumbnails," however the name was changed to "Featured Images" in version 3.0. As such, many of the functions used to work with featured images still bear variations of the name `post_thumbnail`. I will be referring to them as featured images throughout this book.

WordPress does not turn on a theme's support of featured images by default; that must be done by adding the following line in your theme's functions.php file:

```
add_theme_support( 'post-thumbnails' );
```

Once support has been added, a new dashboard metabox such as the one in figure 2.17 will appear on your post and page edit screens. You can select an image from your media library or add an image from another source as you would if you were inserting the image directly into the content.

Figure 2.17 The Featured Image dashboard metabox by default will appear in the right column of your add and edit posts and pages screens once support has been added for your theme.

To make this header image appear on our post page, we'll first want to test for its existence (featured images are, of course, entirely optional, even once theme support for them is enabled), and then send the image markup to the browser. The logic for testing and writing the featured image is in listing 2.5.

Listing 2.5 Testing For and Displaying a Featured Image in a Template

```
if ( has_post_thumbnail() ) { #A
  the_post_thumbnail('thumbnail'); #B
}
```

#A See if the featured image for this post exists.
#B Display the featured image.

The resulting markup that is sent to the browser will look something like listing 2.6.

Listing 2.6 Featured Image Markup Generated by WordPress

```
<img width="150" height="150" src="http://wordpress36/wp-
content/uploads/2011/07/dsc020853-150x150.jpg" class="attachment-thumbnail
wp-post-image" alt="Orange Iris" /> #A
```

#A Notice how the width and height attributes are set. We might have to do something about that at some point.

What the image actually *looks* like, and how it's displayed in the theme, is totally up to the theme and the CSS in it. For example, the above code in the Twenty Thirteen default theme would like something like figure 2.18:

Figure 2.18 Twenty Thirteen places its featured image above the post, but you don't have to do it that way.

HEADER IMAGE

Another specialized use of images in WordPress is as site header images (sometimes referred to as banners). Site headers are global elements that appear on every page in the site, not on particular posts or pages as with featured images. When supported in the theme, WordPress provides a special user interface to control which image (or if any image at all), and can be selected from a choice of defaults that the theme has established, or else uploaded by the user (figure 2.19).

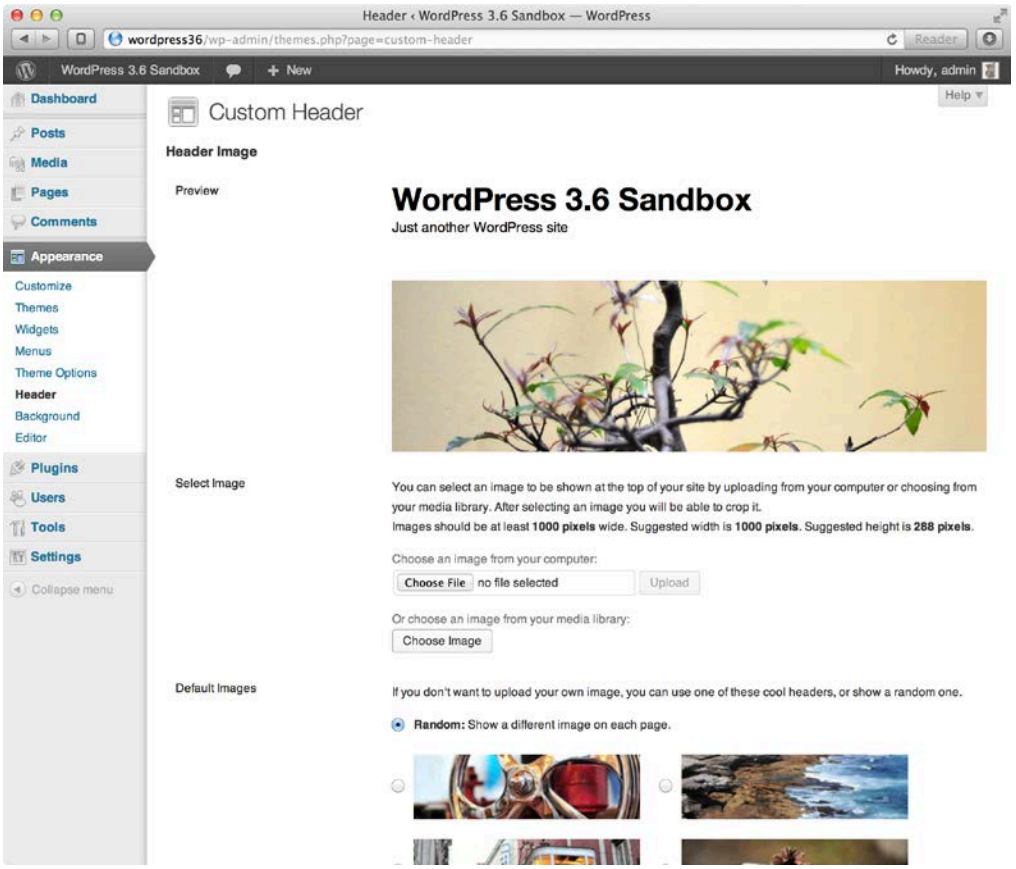


Figure 2.19 When supported by the theme, the user can choose from a variety of default header images, upload their own header image, or choose to display no header image at all.

Again, as with the featured image, we need to turn support for header images on in our theme's functions.php file. But unlike adding featured image support, when we enable custom headers on our site, we need to make some decisions via an array of arguments.

These arguments include defining the color of the text that will be overlayed on the header image, what our default image will be, the dimensions of our header image and more. A full listing of the available options and their default values can be found in the WordPress Codex at http://codex.wordpress.org/Custom_Headers.

Let's work with an example. Say we want a header image on our site that is at most 1,280 pixels wide and by default 200 pixels high. We want the user to be able to define the proportions of his header image, so we'll leave that "flexible." Not flexible or fluid in the responsive way, but in the meaning that when we are asked to crop the image, we won't be constrained to a certain aspect ratio. We know where we will keep a default image in the theme's directory structure, but we'll also define another possible header that will be available at the outset. And since the user can upload his own image which could be any color, the header image editor screen will allow them to change the text color on the banner, but we'll start with a default value of near-black.

Our code for our functions.php file would go a little something like in listing 2.7.

Listing 2.7 Enabling a Header Image on Our Site

```
add_action( 'after_setup_theme', 'my_custom_header_setup' ); #A

function my_custom_header_setup() {
  $args = array(
    'default-text-color'          => '262626', #B
    'default-image'               => '%s/images/headers/my-header-1.jpg' #C
    'height'                      => 200,
    'width'                       => 1280,
    'flex-height'                 => true
  );

  add_theme_support( 'custom-header', $args );

  register_default_headers( array( #D
    'my-header-1' => array(
      'url'           => '%s/images/headers/my-header-1.jpg',
      'thumbnail_url' => '%s/images/headers/my-header-1-thumb.jpg',
      'description'   => _x( 'My Header 1', 'A description.', 'my-theme' )
    ),
    'my-header-2' => array(
      'url'           => '%s/images/headers/my-header-2.jpg',
      'thumbnail_url' => '%s/images/headers/my-header-2-thumb.jpg',
      'description'   => _x( 'My Header 2', 'A description.', 'my-theme' )
    )
  ) );
}
```

#A This line is an action hook, telling WordPress to execute this code at a particular point in time.
#B This is our hex code for a dark gray color, minus the usual leading "#".
#C The characters "%s" are replaced with the location of your theme's directory. WordPress will look in the subdirectory structure you entered here, starting at your theme's main directory.
#D register_default_headers() is the means by which you are defining which default images will always be available to our site builder. It is essentially an array with certain required information.

To display the header image in our theme, we will need to call the function `header_image()` where we want the header image to display. However, unlike the other "display image" functions we've already covered, this function does *not* actually display the entire image tag. Instead, it only displays (or in the case of `get_header_image()`, returns) the path to the desired header image. This way, it can be included via CSS as the background element of a container, making it easy to add text on top or to style the header in responsive-friendly ways, say as in listing 2.8.

Listing 2.8 Example Usage of header_image()

```
<div class="header-banner" style="background-image: url(<?php
header_image(); ?> #A
<h1>My Awesome WordPress Site</h1>
<h2>Just another WordPress Site</h2>
</div>
```

#A Calling the background-image as the value of an inline style attribute is one way to set the header image as a background dynamically, but we'll look at more elegant ways to do this later.

This code might result in a header image that looks something like figure 2.20:

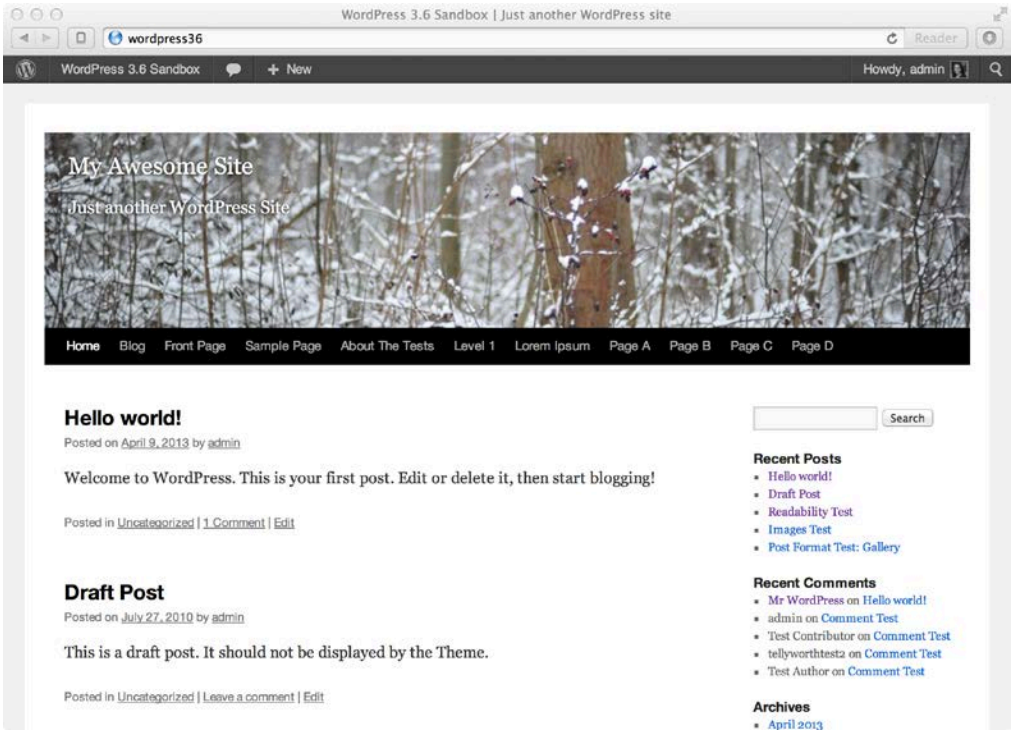


Figure 2.20 Our header image, defined as a background image for a header div.

2.3 Summary

We've now examined the properties and structure that all responsive websites must have in order to look great on a variety of browsers and devices. The meta viewport tag, fluid grids, media queries and responsive images all play a key role in making a web page render in desktop and mobile screens alike.

We've also taken a look at key features and patterns common in WordPress websites. Not every WordPress site you'll build in your career will contain every feature that WordPress has to offer, such as widget areas, horizontal navigation, featured images, etc., but with every project you start, you'll have these tools at your disposal.

All through this chapter, I've said how we'll be using what we've learned when we build our responsive WordPress theme. Well, that time is now. In chapter three, we'll start with the design process and examine some best practices for planning your responsive theme. From concept, to structure, to visual design, there's a lot of considerations that go into determining just what your WordPress theme will be. Let's get to it!

3

Planning Your Responsive Theme

This chapter covers

- Determining what kind of theme you're going to create
- Wireframing your responsive theme by hand, with online tools and in the browser
- Creating a visual design for your theme in Photoshop

In chapter one of this book, we took a look at the history of both responsive web design and WordPress. We looked at a few responsive WordPress themes and examined a little about what made them tick. Then in chapter two, we dove into the nuts and bolts of responsive design, learning some of the basic principles that we'll be working with throughout the rest of this book. We also learned about some of the common traits that are shared by many, if not most, WordPress themes, and which we're likely to come across when building our responsive theme.

So now we're ready to dive in to code, right? Have patience, young Padawan. Before we can start building our theme, we need to take some time to plan what we would like our awesome theme to be. And that requires asking some questions and forming an idea of just what we want our finished product to be.

If you're a designer, you likely already have workflows that you're comfortable with. By all means, start with using the tools you know, because they can likely be made to adapt to a responsive process. However, if you're not primarily a designer, or you are but you feel like your current tools are lacking in capabilities, here's an opportunity to explore new ones that you may not be familiar with.

A warning, though: responsive web design has muddied the design process waters considerably. The "one Photoshop comp per page" model that for years served as the backbone of the web design workflow now fails under the weight of multiple views. I'm almost afraid to write this chapter because the ground is shifting so much under our feet right now. For those reasons and more, we're going to focus less on the subtleties of the

responsive web design process, and more on the tools that are available to help you get there. In appendix x, there will be a list of further reading (both books and blog posts) that you can reference to help further guide you in finding the responsive web design workflow that will work best for you.

3.1 Determine Your Goals

WordPress themes are usually developed for one of two reasons. Either you need to design and/or build a custom theme for a client project, or you are building one for mass distribution of some sort, either as a free theme from the WordPress theme repository or to sell as a premium theme. Whichever situation is applicable to you, you'll need to answer some questions as we start with the design phase.

3.1.1 Building a Custom Theme for a Client

Working with clients has its trials and tribulations, to be sure, but it does have the decided benefit of giving you a solid starting point. Since a custom theme doesn't have the expectation of having to be all things to all people, you can design something that's going to meet the client's exact needs, no more and no less.

We'll start with the big questions: what does your client do and what is his or her website going to be? The answer to the first question will (hopefully) be clear; maybe the answer to the second will come out gradually over the course of several planning meetings. Either way, by the end of the initial phases of the planning process, you and the client should have a clear goal for what the website should be. Perhaps it's going to be a marketing tool for a restaurant. Or maybe it'll be a blog to promote topics related to the client's industry (see figure 3.1). It could even serve as an information-sharing tool amongst the members of an organization. A WordPress site can serve all these disparate kinds of websites if it is planned carefully enough in advance.

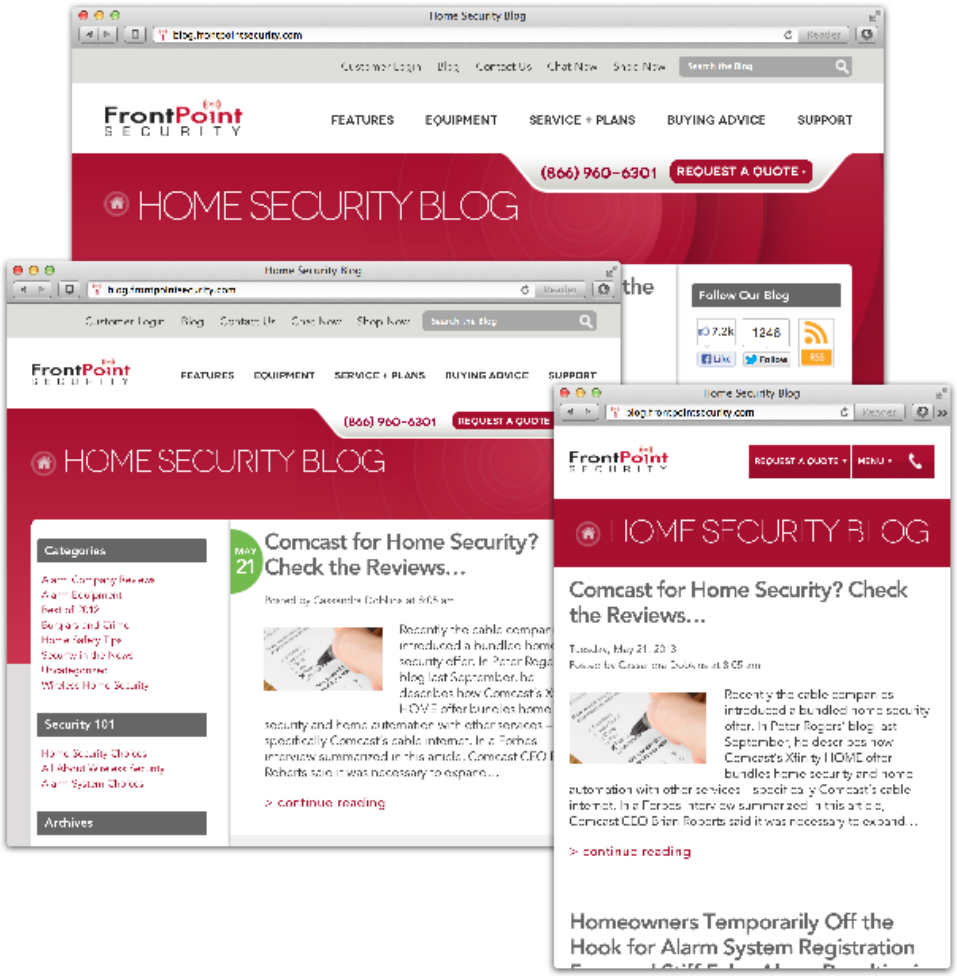


Figure 3.1 Virginia-based FrontPoint Security keeps all who are interested up-to-date on home security news via their responsive, WordPress-powered blog (http://blog.frontpointsecurity.com).

Knowing your client up front can save time in the development cycle as well. For example, the client probably has his or her logo already (or perhaps that's part of the overall project, but in that case it's still separate from the WordPress work), and so having to build in a mechanism to swap in a custom logo isn't going to be necessary. Similarly, other graphic and branding ingredients (things like background images, header images, typography, color scheme etc.) can be custom-tailored to the client's exact specifications. This takes a lot of the variables off your plate.

3.1.2 Building a Theme for Mass Distribution

Building a theme for mass distribution is a bit trickier prospect because you don't have any clients answering basic questions for you. Instead, you have to anticipate and build in options to allow an unknown installer of the theme to customize it to his or her needs. Fortunately, WordPress comes with many mechanisms to make these options easier and are yours for the activating.

But despite the uncertainty of an unknown client, don't think that the whole world necessarily has to be your audience. Plenty of themes find happy customers by being geared towards a particular purpose. There are themes for sale or download that are specifically geared towards niche markets such as photographers wishing to display their portfolios (figure 3.2, for example). Likewise, there are many newspaper and magazine-like themes that are used for sharing news on a particular topic or about a region. Ecommerce themes, college and university themes, and yes, even simple personal blogging themes still exist, and are rather plentiful. For all the talk of WordPress as a CMS, it's still awfully good, and awfully popular, as a blogging engine!

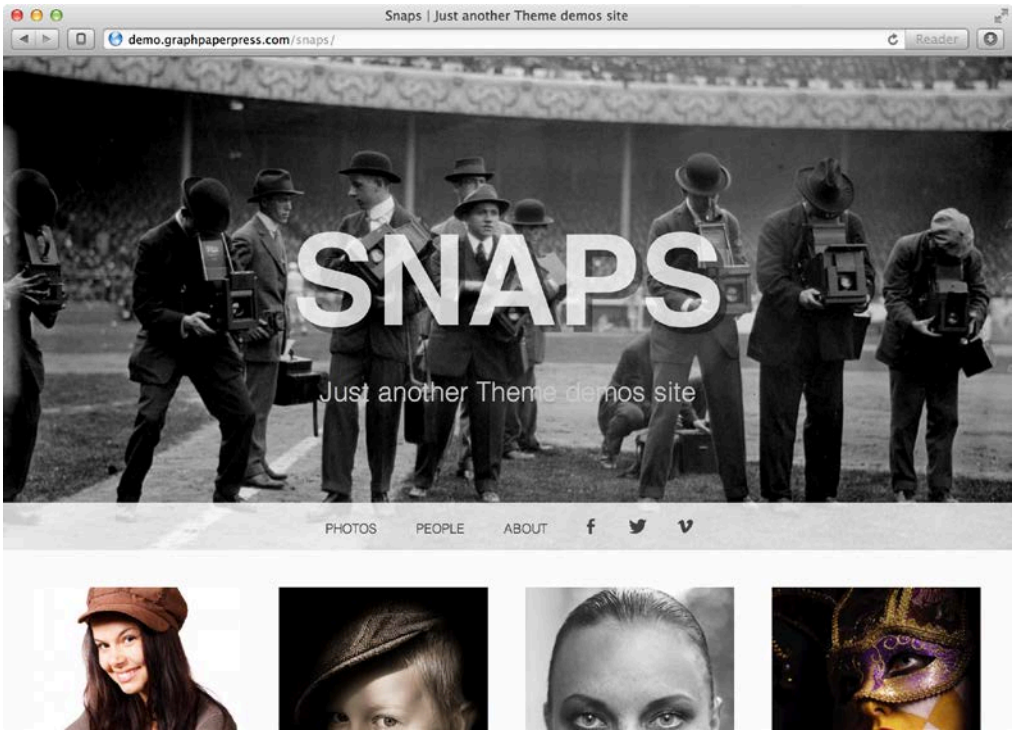


Figure 3.2 Snaps, a free responsive theme from Graph Paper Press, is designed especially for showcasing portrait images and galleries.

The theme we're going to build in this part of the book will be for mass distribution. Furthermore, we're going to build a basic blogging theme that can also hold some static pages as well. We're going to keep things nice and generic, so as to give you experience with as many of WordPress' standard features as possible.

Designing for Content First

It's becoming increasingly common to see the advice that websites should be designed with the content first and foremost in mind. On its face, this seems very logical, as delivering content to your visitors is the primary objective of your website and as such should drive many of your design decisions. Layout, typography, whitespace and even color schemes will likely be affected by the kind of content that will populate your site. As web standards guru Jeffrey Zeldman puts it, "Content precedes design. Design in the absence of content is not design, it's decoration." (https://twitter.com/zeldman/statuses/804159148)

If you're working on a WordPress site for a client, then you have the opportunity during the planning phases to work out the content that will appear on the site. You probably won't be able to nail down every word, of course, but you can develop a firm sense of the kind of content you will be designing for. A restaurant site might have menus, directions and information about reserving for private events. A blog will have articles, but also possibly photo galleries or video. The list goes on.

However, when building a generic theme, we're at a bit of a disadvantage because we don't have any way of contemplating the kind and quantity of the content the downloaders of the theme will have on their website. Ergo, when drawing up our wireframes, we need to build in as much flexibility as possible, and then hope that people will choose and populate our theme responsibly. I wouldn't use the Snaps theme for my everyday blog, and we wouldn't necessarily recommend the blogging-style theme we're about to create to someone looking to showcase their video portfolio. While it'll be able to handle the occasional large photo, video, etc., it's not really going to be geared to that sort of content, and thus wouldn't display it as optimally as another theme might.

Technical Requirements for a Mass-Distributed Theme

Fortunately, the fine folks at Automattic (the company founded by WordPress co-founder Matt Mullenweg and which serves as one of the guiding forces behind WordPress) have offered up some practical suggestions in things to think about when building a theme for distribution. According to their ThemeShaper blog, a theme destined for mass distribution should have the following characteristics:

- Make sure your theme can endure a wide range of content situations without breaking–long post titles, long widget titles, many menu items, etc.
- Make sure your theme is translation-ready.
- Add a right-to-left (RTL) stylesheet for those whose languages read from right-to-left.

- Make sure the fonts you use support characters from a wide number of languages.
- Avoid hard coding, and respect user settings as much as possible.
- Avoid using ornamental graphics (such as a post title or widget title background) that breaks if text gets too long or too short.

"Distributing Your WordPress Theme," http://themeshaper.com/2012/11/23/distributing-your-wordpress-theme/

These items, and more, will be covered in depth as we build our theme, but they're good things to think about now. The key ingredient is flexibility: since we don't know our ultimate end users or their particular needs, we need to design and build a theme that will fit the needs of a variety of installers and make adjusting our theme to those needs as easy as possible.

What Should Go Into Our Theme

With the world as our potential customer, we should think of what the customer is likely to want from our theme. Some mass-distributed themes make extensive use of theme options to allow the user to customize the site in a myriad of different ways. While it'll be necessary to offer some options such as customizing the logo and header graphic, there are some things that we're going to make executive decisions on, such as the number of sidebars and where they're placed.

Since there's no place like home, our home page is where we'll begin. Let's take a moment to determine which elements should be on our home page:

1. A header, containing a logo, a site name and a site description (also commonly referred to as a tagline), all set against a background image. If the user opts to not include any of these items, the site should adapt and not look broken in any way.
2. A horizontal primary navigation menu with a site search box against the right side of the page. Since we don't want our long horizontal navigation menu to wrap in narrower views, we'll need to figure out ways to allow it to collapse gracefully.
3. Sidebars to hold our widgets. We'll have two sidebars, and how they're laid out on the page will depend on the window width. Some WordPress themes will auto-populate a sidebar with some default widgets such as the search or archive widgets until the site owner puts widgets of their own there. We'll take this approach for one of our sidebars, but not the other; if it doesn't have any widgets, the second sidebar will simply not appear.
4. A footer with four widget areas. These widget areas can be used to hold the site's copyright information, a "fat" footer area with a more thorough site menu than our

primary navigation bar, a secondary or affiliate logo, etc. Footers with widget areas are becoming a standard practice in WordPress themes, a trend that started with the WordPress 3.0 default theme "Twenty Ten" and is now being adapted by other theme developers as well. It allows the theme user to control the content in the footer more easily and without having to edit the PHP template files.

Most of these elements will carry through to the interior pages, except that they shall have only one sidebar, not two and by extension will feature a wider main content area.

3.2 Designing Wireframes

Now that we've determined what type of theme we are building (as generic as it is) and what elements it will have, it's time to start plotting what it will look like. The first step in that process is building out wireframes, which depict websites in much the same way as blueprints depict houses. They are not meant to be accurate representations of the final product, but rather are intended to serve as a guide to the user interface. The details contained in wireframes can vary considerably from one designer to the next, but at the very least they will illustrate the layout of the site. Consideration can also be given to placement of text, images and navigational elements such as call to action buttons.

In the pre-responsive world, wireframes would only depict the context for which the site was being developed, either desktop or mobile. In our brave new responsive world, however, we need to design for the multiple contexts our site might be viewed in. So where once upon a time we could draw up one wireframe per template, we must now create several wireframes per template.

What to Call Our Views

I have a little confession to make. I hate the terms "mobile" and "desktop," even though I've been using them a lot to this point. These terms are at best imprecise, and at worst false and misleading. Is a site viewed on an iPhone necessarily mobile when the user is sitting on his couch at home? Likewise, what kind of desktop is a person using if he is outside on his office's roof garden with his laptop? And then there are tablets and even phablets (think the Samsung Galaxy Note II) that defy all attempts at such categorizations.

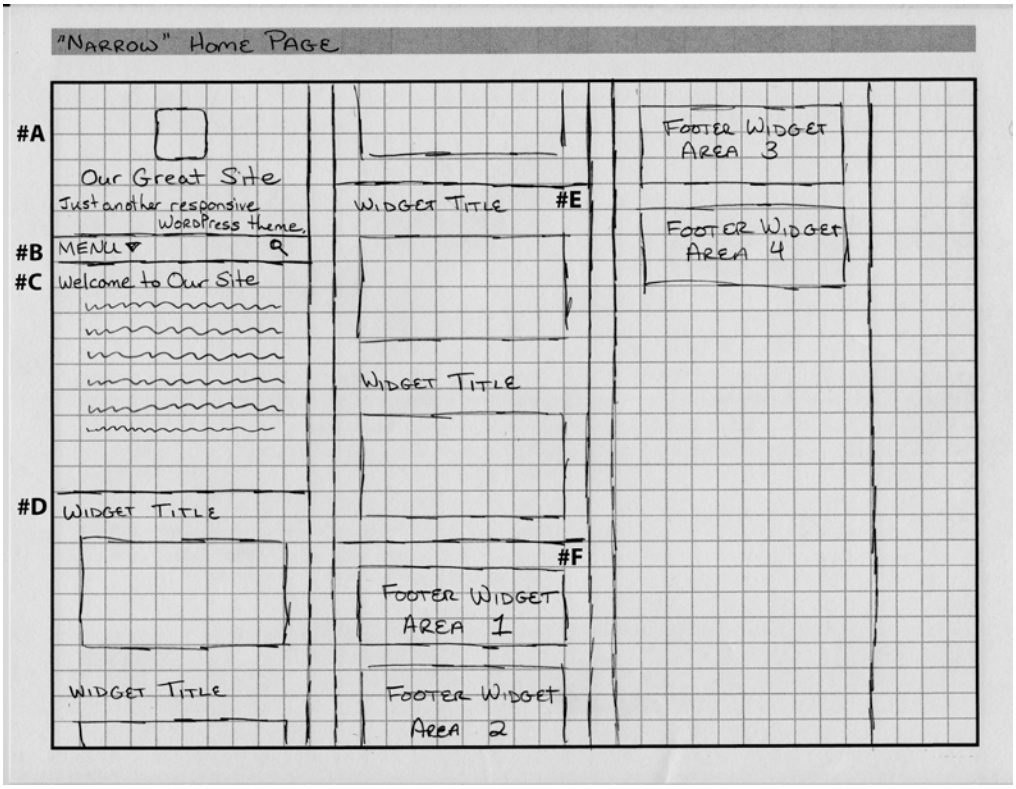
Instead of bogging ourselves down to the mobile versus desktop semantic debate, we're going to look purely at screen width and classify everything we do from here on to fall into one of three categories: narrow, medium and wide. At this point, we're not even going to worry about the breakpoints that will divide those three groupings; we'll let the design drive those decisions later. For now, it'll be the most useful to think in terms of what will our site look like to the users with the narrowest browsers, the widest browsers, and those that fall somewhere in between.

There are a number of wireframing tools out there that we can use in this earliest design phase. Some newer tools have been especially created for working with responsive websites. Others are more traditional, yet can be adapted to meet the needs of varying layouts. Let's start thinking about the structure and layout of our theme, and as we do we will try several different tools commonly used for wireframing.

3.2.1 The Old-Fashioned Way

It doesn't get any more old school than to build out your wireframes with simple pencil and paper. Certainly, no other method is more economical. For five dollars at your local office supply store, you can get a pack of pencils and a graph-lined notebook and get to work. If you can't be bothered with a trip to Staples, you can even download graph paper templates to print out at home. Paperkit (http://paperkit.net) is a site that allows you to customize graph, dotted and lined paper by adjusting the spacing, document size, stroke width, etc. Sneakpeekit (http://sneakpeekit.com) provides downloadable graph paper templates specifically for use in responsive design, including sheets for mocking up iPad and iPhone designs.

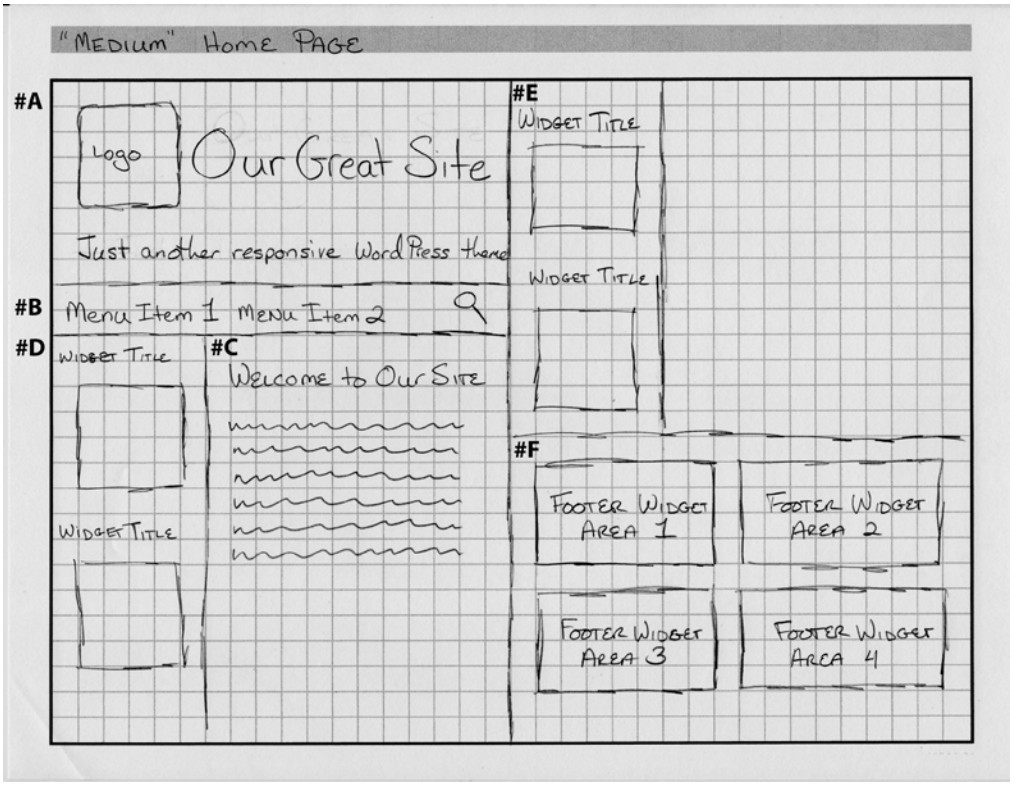
Using my rather crude drawing skills on just a basic sheet of graph paper, and with the elements of the theme home page that we listed out in section 3.1.2, I have sketched out several variations to what our home page might be beginning with the narrow view (figure 3.3):



#A Our site header.
#B Primary navigation. We'll examine different responsive navigation treatments in chapter x. Suffice for now that a full horizontal navigation in a narrow width screen won't be possible.
#C The main content area. We're assuming, for no reason in particular, that the user will choose a static page as their home page, but they could just as easily have a listing of recent blog posts here.
#D The first sidebar.
#E Stacked just underneath the first sidebar is the second one. On a mobile view, these two widget areas will appear to be contiguous, but rest assured we have big plans for this second sidebar.
#F Our footer with its four distinct widget areas.

Figure 3.3 We're creating our home page layout mobile first, so we're starting with the narrowest possible layout. The three columns you see are meant to represent one long column of content. Physical paper does have its limits, after all.

At this point, things are pretty simple. All the pieces of the site we described in section 3.1.2 are there, stacked together in one long column. The next step is to figure out how things change when we expand out the browser a bit, as in figure 3.4:



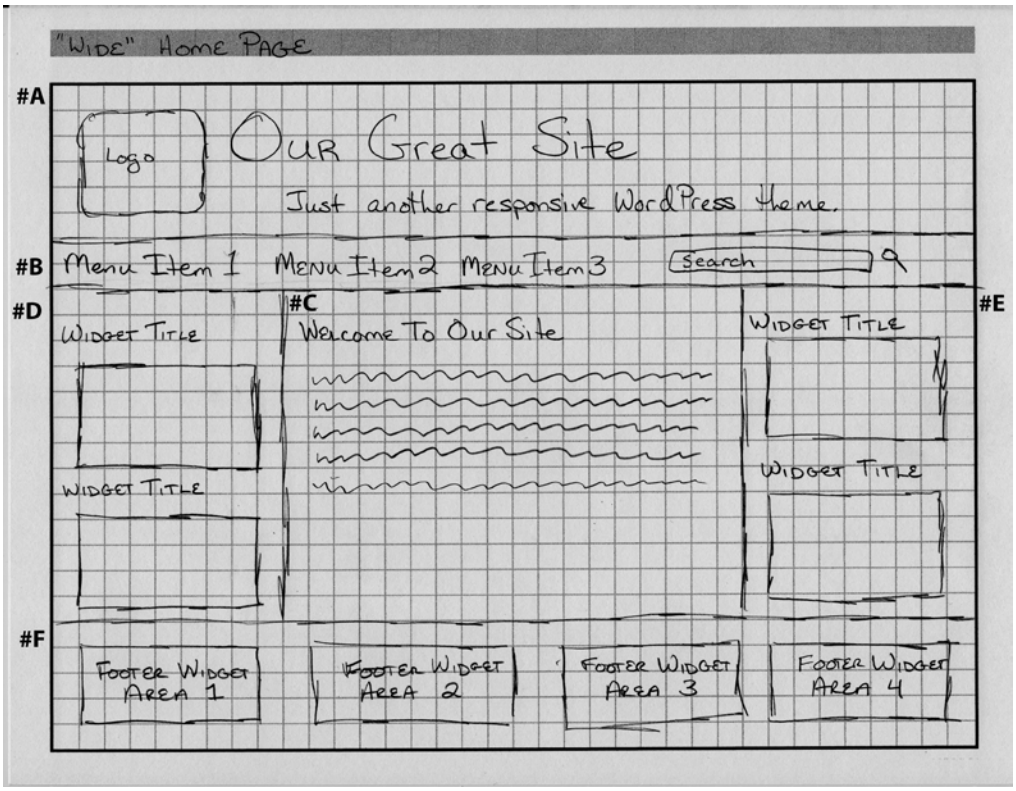
#A The site header.
#B Primary navigation and search.
#C The main content area.
#D The first sidebar.
#E The second sidebar, still stacked underneath the first.
#F The site footer.

Figure 3.4 While all of our elements are still there, the layout has changed slightly, making use of the wider browser. Still dealing with the limitations of paper, we've broken up the home page into two columns so we can see everything on one sheet.

All of the elements in our narrow view are in our medium view, but the layout has changed in order to make use of the greater screen real estate at our disposal. Key differences include the primary navigation is now shown expanded with some potential menu items and the footer widget areas are arranged in a two by two grid instead of all stacked one atop another.

But already, we're seeing one potential pitfall with our design in that the two sidebars, when filled with widgets, could potentially extend well beyond the main content that floats next to it. However, since we have no way of knowing how many widgets there will be, nor

how long the content in the main content area will be, we can't really stress about this too much. Later, as we work with test data, we'll see how much of an issue this actually becomes. In the meantime, let's move onto the third and final view of our home page, the wide view (figure 3.5):



#A The site header.
#B Primary navigation and search.
#C The main content area.
#D The first sidebar.
#E The second sidebar, now floating on the right side of the page.
#F The site footer.

Figure 3.5 Our third and final view is the wide view.

Again, all of our elements are there, but in a new layout. Our search field is now visible (in the narrow and medium views we only saw the magnifying glass icon). And most notably, our sidebars now sit on either side of our main content. To rearrange the sidebars

on the screen in this matter is going to take more than just CSS, it will involve some DOM manipulation with JavaScript. Don't worry about that now, we'll cover that when we get there in chapter x.

So even with the most basic, low-tech design tools possible, we've still managed to come out with a fairly decent representation of the structure and layout of our theme. If pencil and paper just isn't your thing, but you do like the idea of sketching out a site design by hand, there are sketching programs for tablets that will let you draw with a stylus.

3.2.2 Online Wireframing Tools

Since many of the elements in our theme will carry through to the other templates of our site, namely the header, primary navigation and footer, the interior page templates will contain a little more detail in the main content areas. For our next set of wireframes, we're going to look at some web-based wireframing tools and focus in on how the content for an archive page will look. All of the ones we're going to be working with have free trials available, so go ahead and work with them to get a feel of which one you like the most, if any.

Three of the most popular tools available are:

- MockFlow (http://www.mockflow.com/)
- Balsamiq (http://www.balsamiq.com/)
- HotGloo (http://www.hotgloo.com/)

There are pros and cons of using these online tools. One of the benefits of working with these online tools is the collaboration they allow. If the members of your team all have accounts on one of these services, you can share and contribute to mockups together, even if your team is widely distributed.

However, a downside in going the web-based route is the comfort of having possession of your wireframe documents on your own hard drive, under your control (although you can usually export them to a number of different formats such as PDF and images). This could backfire on you, of course, if your hard drive crashes and you don't have a backup. (You *do* back up your hard drive on a regular basis, right? Good, I knew you were cool like that.)

Lastly, the services we're going to look at in this section are all Flash-based applications (although one has a desktop application as well, but is not a "native" app, *per se*). I'm not going to pass along any judgments on whether Flash is a good thing or evil incarnate, but one does have to admit it has its quirks. If Flash isn't your cup of tea, you might want to stick with the pencil and paper route, or skip onto the next section. But if you're okay with it, continue on as we try out three of the most popular online wireframing options out there.

What Do We Mean by "Archive" Page?

Before we go much further, let's make sure we understand what is it we're going to be mocking up.

WordPress has a number of different templates that reflect different requests of data. One of these is known as an "archive" page, because it's a listing of archived posts and can be segmented in different ways. You can have an archive page for all posts, for posts in a particular category, by a certain author, or from a specific month and year, to name but a few examples.

We'll investigate the different types of template files further when we start setting up our theme in the next chapter.

We'll start by using MockFlow to wireframe our narrow view. When starting a new project in MockFlow, it gives you a selection of different project types to serve as a starting point, including WordPress (figure 3.6), which is handy because it saves you a lot of time populating your view with the same types of elements over and over on each new project.

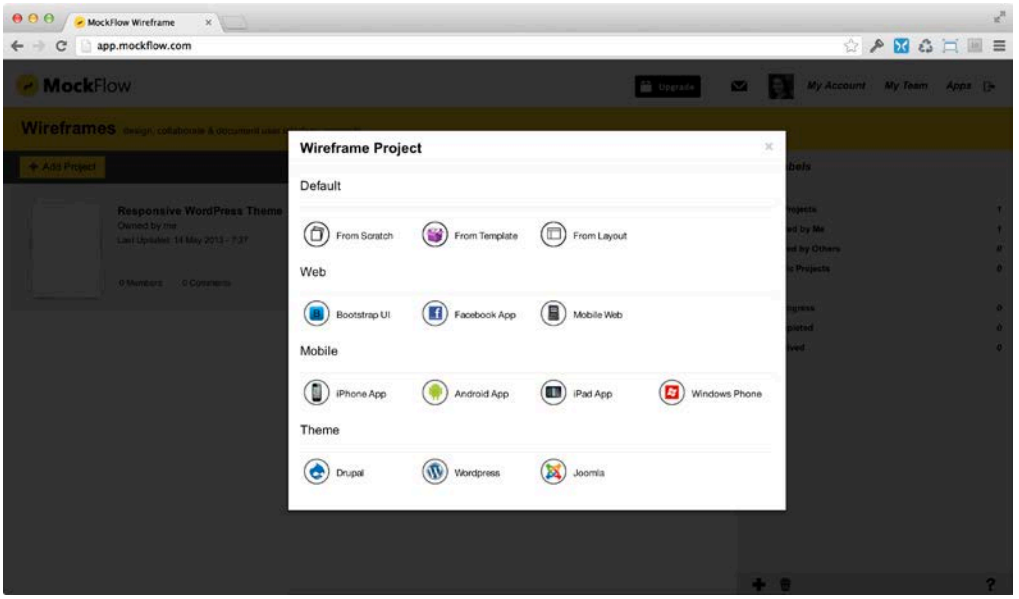


Figure 3.6 To get you started on your project quickly, MockFlow lets you choose from popular project types and then auto-creates a wireframe with items common to that type.

Since we're building a WordPress theme, logically we'll choose that. What we get is shown in figure 3.7:

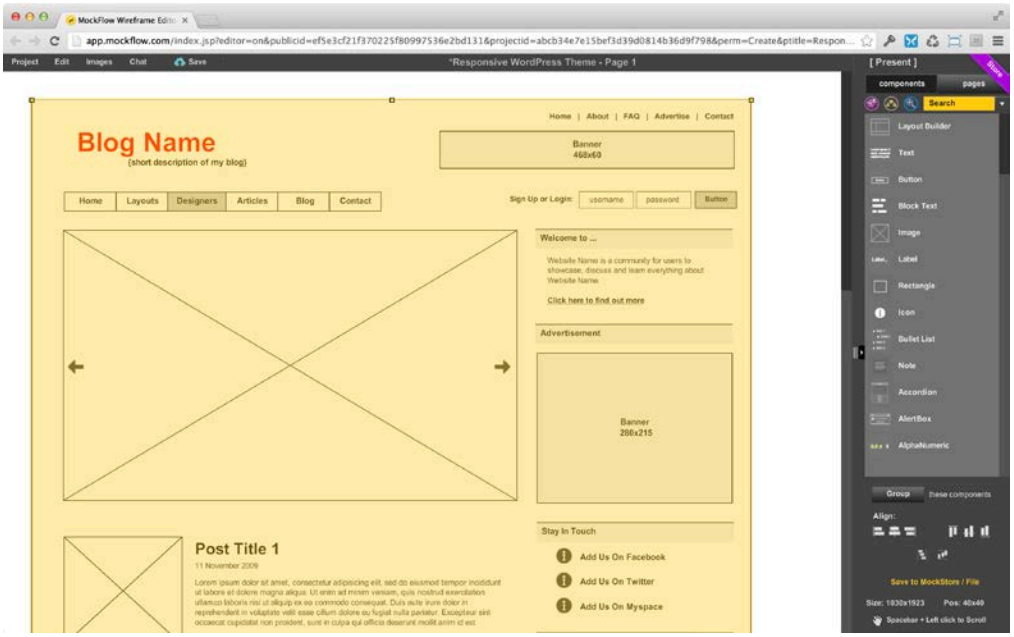
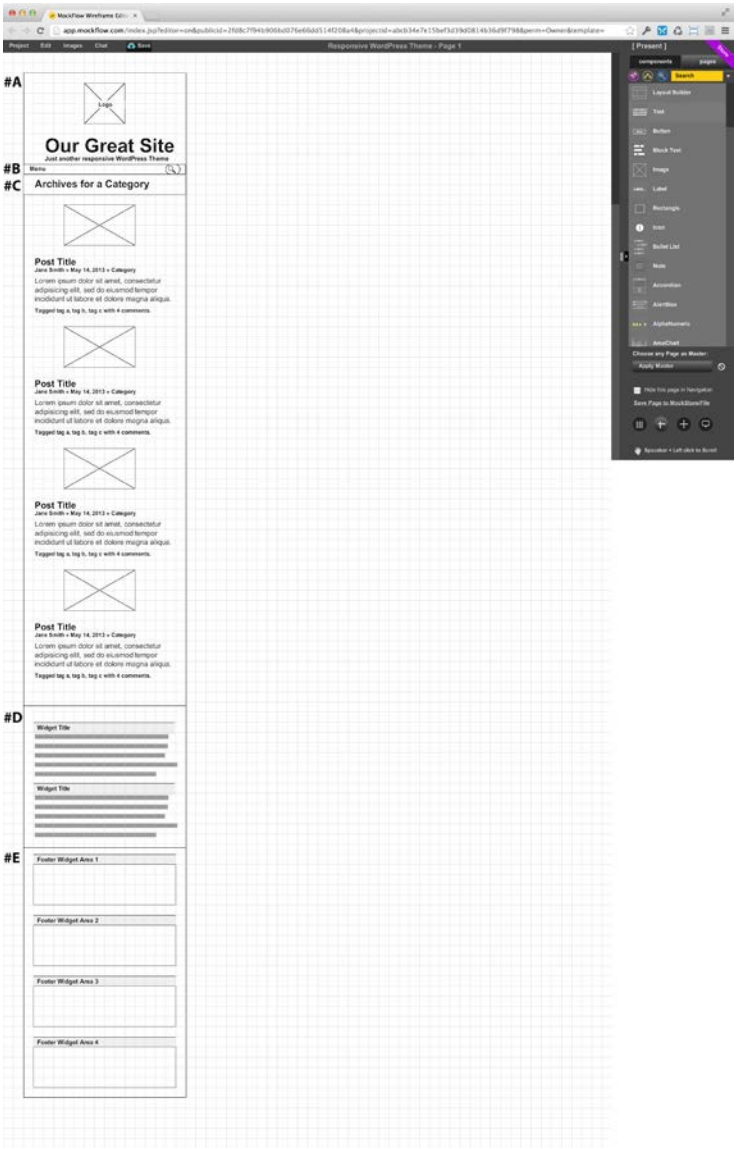


Figure 3.7 This certainly does look WordPress-y.

While the wireframe in figure 3.7 would be a terrific starting point for the wide view of a WordPress theme, it's not going to help us a whole lot in building our narrow template view. So some considerable editing will ensue.

Online wireframing tools come with pretty much every standard user interface element one can think of, including image placeholders, headers, form elements, etc. We just need to drag them to the point in our mockup we want them. After a bit of fiddling with our wireframe, we come up with the results in figure 3.8:



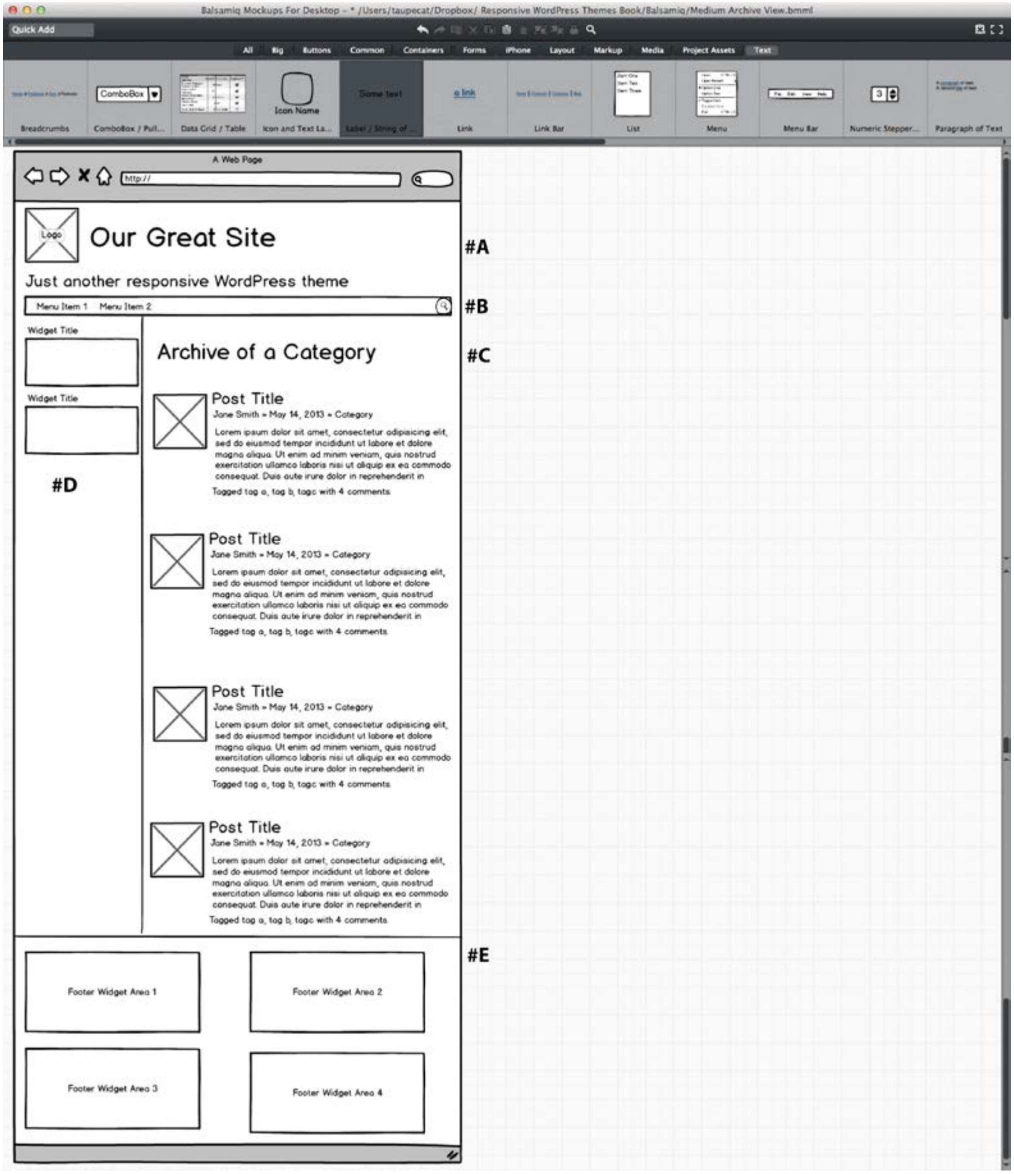
#A Our header is still intact in its original form.
#B Ditto for our primary navigation/search bar.
#C The meat of this page is a listing of posts, including featured images and meta data.
#D Instead of two sidebars as on the home page, interior pages will only have one.
#E The footer will remain the same throughout the site.

Figure 3.8 After a bit of reworking, here is our archive template wireframe in narrow view.

There's not a huge difference between our archive page and our home page, especially in this narrow view. The nature of the main content is different in that it's a listing of posts instead of a single page of prose. Also, interior pages will have one sidebar, not the two that the home page sports.

We've added some detail where we need it the most, which is the details of the post that will be included in its listing. If a featured image is available, a thumbnail of it will appear at the top of the listing, followed by the post title, some meta data, an excerpt, a little more meta data and the number of comments. In these wireframes, I have four posts listed on each archive page. By default, WordPress lists ten posts per archive page, but the user can easily change this number to anything he would like.

Fairly satisfied with our narrow archive view, let's move onto the medium view with Balsamiq. Balsamiq is a little different in that it has downloadable desktop application counterparts for every major platform (Mac, Windows and Linux). Balsamiq's look and feel is much more of a hand-drawn aesthetic, with rough lines and text elements in a handwriting font. All of the usual web design elements are available again for dragging into place from a bar across the top of the window. After playing with the arrangement of the elements we need for our template, we can come up with something like figure 3.9.



#A Our heading is still just as it was on the home page.
#B As is our primary nav/search bar.
#C The single sidebar has moved to float left of the main content area.
#D The main content area has some slight changes from the narrow view.
#E As with the home page, a two by two grid of widget areas makes up the footer.

Figure 3.9 Again, we're not departing greatly from our medium width home page.

The only section of this view we need to focus on is the main content area, which has varied only slightly from the narrow view. All of the same content is there, but now the thumbnails of the featured images is floating to the left of the post entry rather than sitting above it.

Only the wide view is left for our archive renditions, so for that we're going to turn to our third and final online wireframing app, HotGloo. HotGloo's visual style is more of the clean, straight line similar to MockFlow. We don't have much more to do on our archive template wireframe other than to stretch things out for a full-width view, and doing so yields the wireframe we see in figure 3.10.

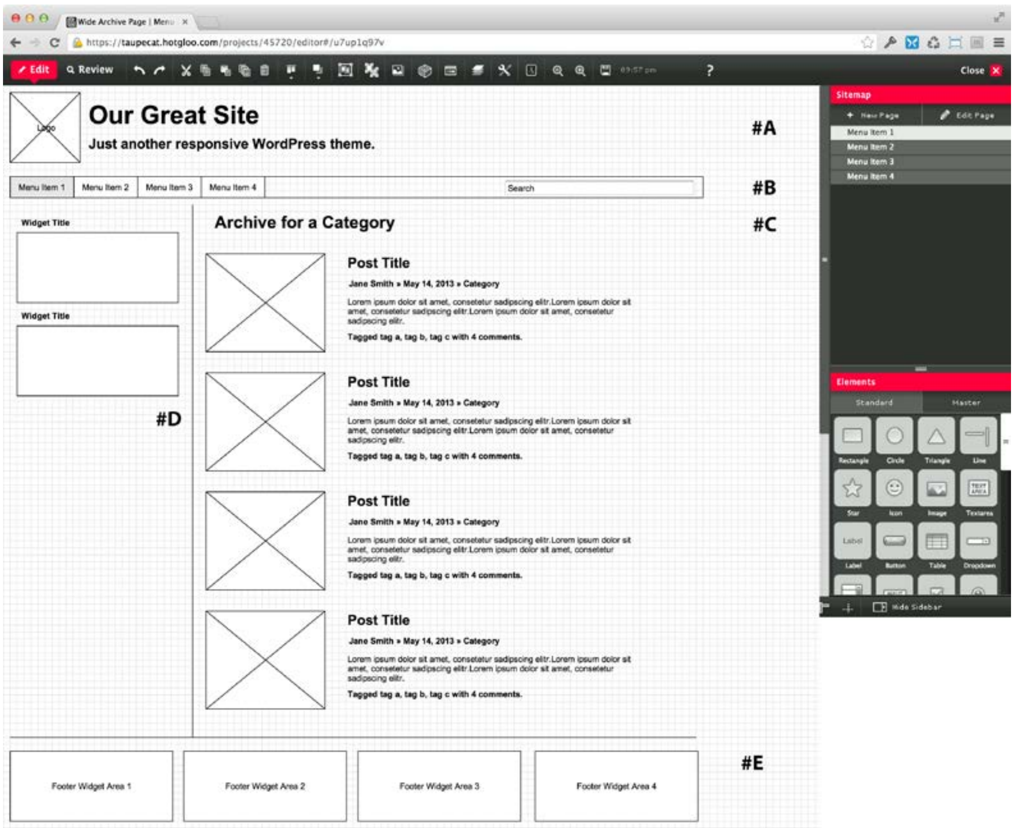


Figure 3.10 Our wide archive view, mocked up in HotGloo.

Now we have a good wireframe diagram of our archive template in our three different screen widths. Designing the wireframe for our single-article template isn't going to be hard

now, since so much of what we've done will carry through. So let's get a little crazy and try some more *experimental* tools.

3.2.3 Other Wireframing Tools

So far, we've looked at creating our wireframes with paper and pencil as well as a few online wireframing tools. Let's just look quickly at a couple of other design methods before moving onto our visual design.

ADOBE EDGE REFLOW

Since the web's beginnings, Adobe has had the reputation for building high-quality design tools that were firmly rooted in a print design past. Despite the ubiquity of Photoshop comps as a *lingua franca* for designing visual comps for web pages, it was often criticized for being tone deaf to the increasingly varied needs of modern web design (figure 3.11).

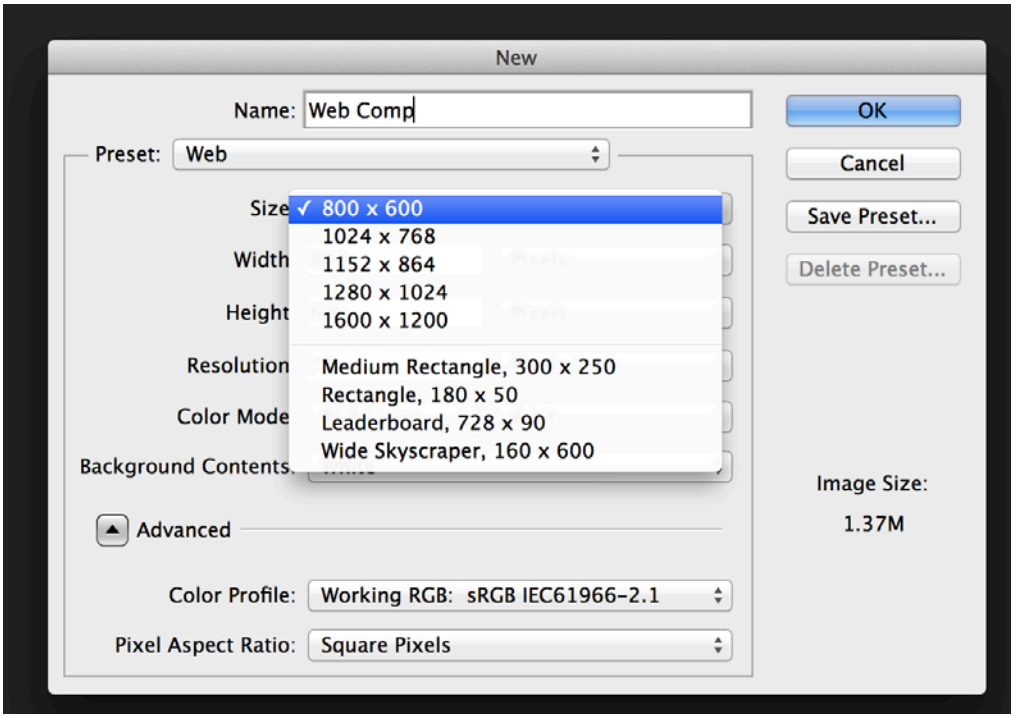


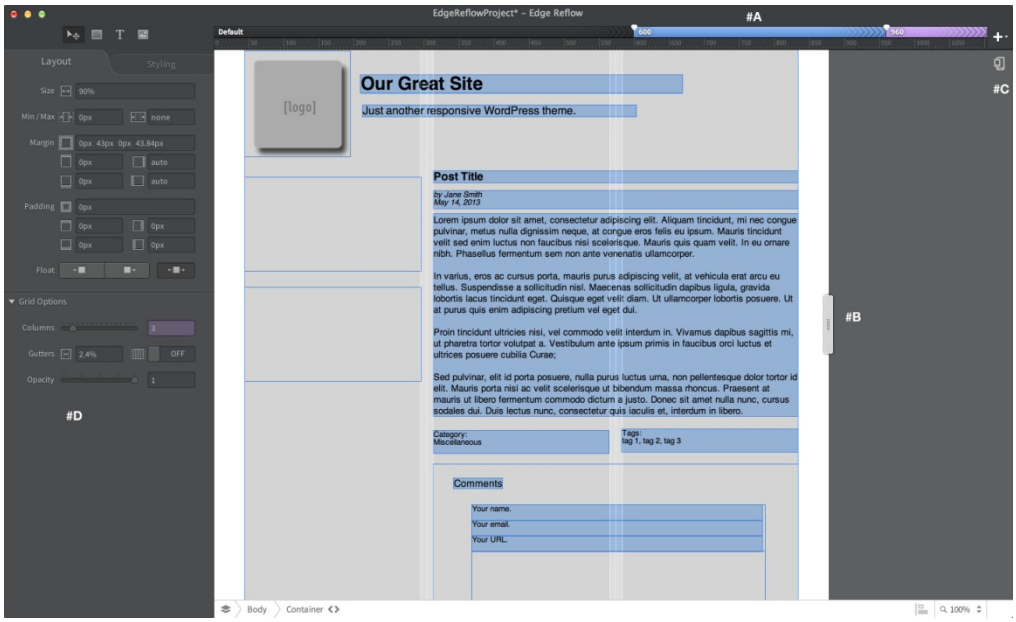
Figure 3.11 The dialog that appears when creating a new Photoshop document is somewhat ridiculous. Web pages just can't be squeezed into arbitrary dimensions anymore.

In recent years, however, Adobe has been seeking to change that. In 2011, they acquired Typekit (http://typekit.com), a popular online web font service that allows designers to

include custom fonts on their websites. They also announced a new suite of products called Adobe Edge, specifically designed for working with websites.

One of the products in this suite is Adobe Edge Reflow (http://html.adobe.com/edge/reflow/), a prototyping and design tool. Instead of being web-based, it's an application that you download to your computer. Also, it doesn't offer up every imaginable UI element, but rather requires you to draw the shapes you need on its canvas. But the real kicker of the app, and what makes it worthy of a look, is its responsive web design features.

The main canvas features a handle with which you can drag the width of page narrower and wider. You can set your default width to either be narrow or wide, then set breakpoints where the elements of your page will adjust. You can even change the number of columns you have available in your grid at different breakpoints (figure 3.12).



#A You can easily add breakpoints to shift your content around, starting with either mobile first or desktop-down.
#B Drag the handle left and right to narrow or widen your canvas.
#C With a separate application, Adobe Edge Inspect, you can preview your layout on an external device, such as your iPhone.
#D Layout and Styling tabs allow you adjust margins, font sizes, floats and a lot more.

Figure 3.12 Adobe Edge Reflow represents a shift in the company's offerings geared towards web development.

Once you have your elements laid out how you want them, you can easily preview the page in Google Chrome, complete with the responsiveness.

Adobe Edge Reflow is still in "preview" mode at the time of this writing, but promises deep integration with some of Adobe's other properties, such as Photoshop and Typekit, soon. When released, it will be part of Adobe's Creative Cloud offerings.

WIREFY: DESIGNING IN THE BROWSER

Another method of wireframing that's been gaining popularity is designing in the browser. The benefit of this is that no other means of design is going to give you the true feel of how something will work in the browser than actually using a web browser. The downside is that this method requires knowledge of HTML and CSS, and not all designers do.

Fortunately, there are tools to help with this method as well. For our single-page article template, we're going to look at Wirefy (http://getwirefy.com), figure 3.13, and see how that can help us build a living, breathing, browser-based wireframe.

Figure 3.13 Wirefy is a set of HTML, CSS and JavaScript files intended to allow quick wireframe design inside the web browser.

Wirefy is a tool that sits somewhere between wireframing and rapid prototyping and relies heavily on being able to code your own HTML, CSS and JavaScript. It bills itself as "a style agnostic wireframe tool built on the principles of content first."

So that you don't have to start from scratch, Wirefy comes with some pre-made templates to help you get going, including a blog post page, a gallery page, and a page with a sidebar. We'll use the sidebar page, and customize it to our needs, which gives us a page like figure 3.14.

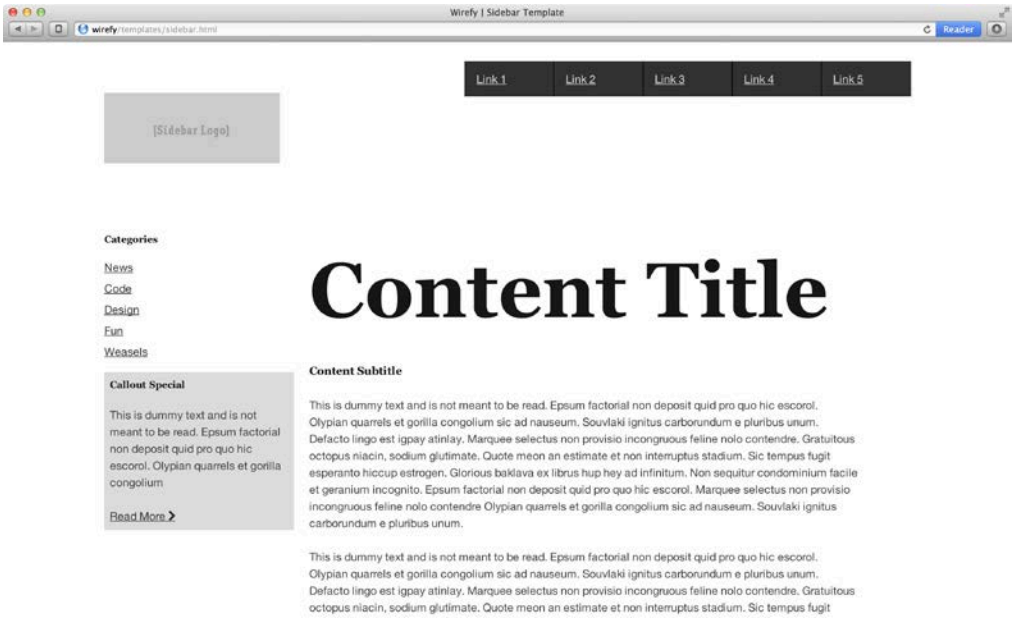


Figure 3.14 A sidebar page, right out of Wirefy's virtual box.

The sidebar template Wirefy provides is pretty close to what we want, but there are a couple of things that aren't quite right:

1. The menu isn't where we want it at all; we want that underneath the header and above the main content, not floating to the right of the site logo.
2. The "Content Title" header is way too large and out of proportion to the rest of the content.
3. When we make the page narrow, the sidebar will sit on top of the main content, rather than the other way around. This will require rearranging the content in the HTML, and adjusting the CSS floats accordingly.
4. We need to replace the stock contents of the footer with representations of our four widget areas.

Now that we know what needs to be done, let's get started by adjusting the primary navigation.

Listing 3.1 Default Header HTML

```
<!-- This is the header information. You can add things like logos and
navigation here. -->
<header role="banner" class="clearfix">
     <div class="logo">
     <h1><img src="http://placehold.it/250x100&text=[Sidebar Logo]"></h1>
     </div>
     <!-- Navigation. This navigation uses the three line approach for small
viewports. Check documentation for other examples -->
     <div class="menu-button"></div> #A
     <a href="#main" class="skip">Skip navigation</a>
     <nav role="navigation" id="nav" class="toggle nine columns">
     <ul>
            <li class="top-level"><a href="">Link 1</a></li>
            <li class="top-level"><a href="">Link 2</a></li>
            <li class="top-level"><a href="">Link 3</a></li>
            <li class="top-level"><a href="">Link 4</a></li>
            <li class="top-level"><a href="">Link 5</a></li>
     </ul>
</nav>
     <!-- End Of Navigation -->
</header>
     <!-- End Of Header -->
```

#A The default sidebar template in Wirefy floats the navigation to the right of the header elements. We'll adjust it so that it sits below the header instead.

We'll need to rearrange the HTML a little bit here, coming up with something like this:

Listing 3.2 Altered Header and Navigation HTML

```
<!-- This is the header information. You can add things like logos and
navigation here. -->
<header role="banner" class="clearfix">
     <div class="logo four columns"> #A
     <h1><img src="http://placehold.it/250x100&text=[Sidebar Logo]"></h1>
     </div>
     <div class="eleven columns">
            <h1>Our Great Site</h1>
            <div class="subtitle">Just another responsive WordPress
theme.</div>
     </div>
</header>
     <!-- End Of Header --> #B
     <!-- Navigation. This navigation uses the three line approach for small
viewports. Check documentation for other examples -->
     <div class="menu-button">Menu</div>
     <a href="#main" class="skip">Skip navigation</a>
     <nav role="navigation" id="nav" class="toggle sixteen columns">
     <ul>
            <li class="top-level"><a href="">Link 1</a></li>
            <li class="top-level"><a href="">Link 2</a></li>
            <li class="top-level"><a href="">Link 3</a></li>
            <li class="top-level"><a href="">Link 4</a></li>
            <li class="top-level"><a href="">Link 5</a></li>
     </ul>
```

```
</nav>
    <!-- End Of Navigation -->
```

#A By adding number of column classes to our logo and site name elements, we're telling Wirefy that we want those elements to fill up a minimum amount of space equal to the width of the page.
#B The other adjustment we're making here is pulling the <nav> element completely out of the <header>.

The results in the browser can be seen in figure 3.15:

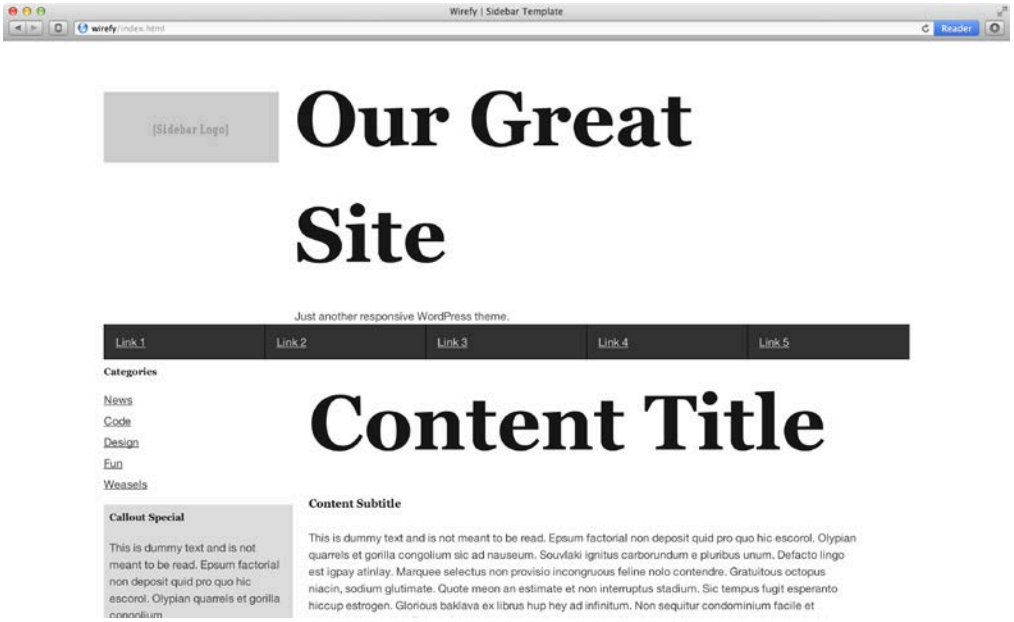


Figure 3.15 This is progress. Really, it is.

Okay, now our navigation is in the right place, but we really need to do something about the size of the headers. Wirefy uses Sass (which we'll talk a lot about in chapter x), but I don't want to throw something new like that at you right now. Suffice to say, we'll be changing a variable from the whopping size of 110 pixels to a more reasonable 72. We'll also knock "Content Title" down from an <h1>tag to an <h2>, just to make the sizes more proportional.

With those adjustments in place, we now have what's pictured in figure 3.16:

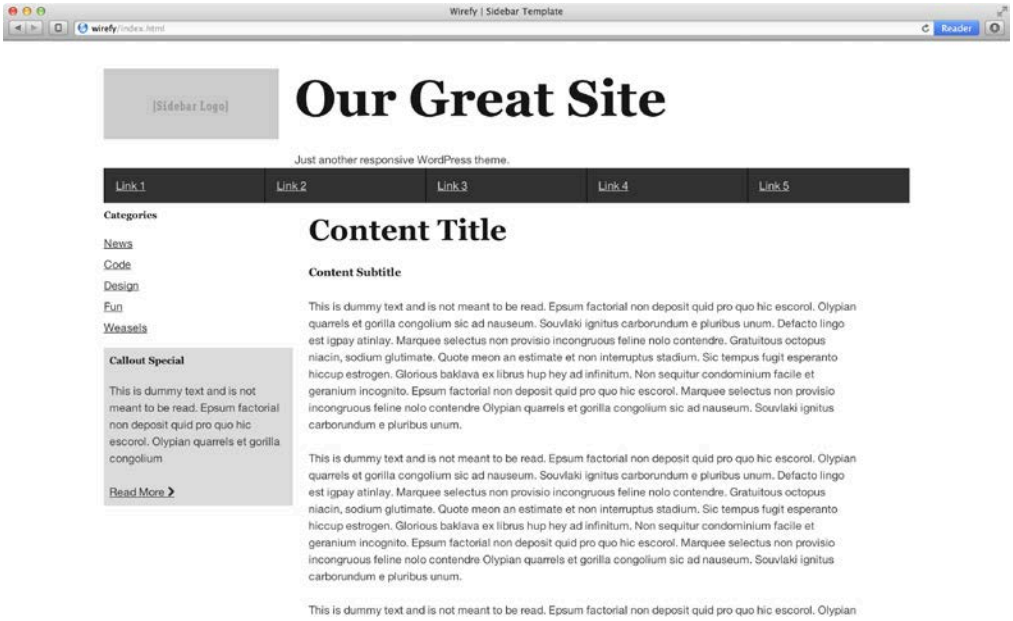


Figure 3.16 Finally, some sanity in our font sizes.

We're almost there. We still need to rearrange the main content and the sidebar, adjusting our CSS floats accordingly. To make sure that the main content sits *above* the sidebar in our narrow view, we'll need to move it above the sidebar in the HTML.

Listing 3.3Adjusted CSS for Sidebar and Main Content

```
@media screen and (min-width: 55.5em) {
  /* 888 ================================================ */
  section[role="content"] {
    float: right; #A
  }
```

#A By making sure our content floats to the right, we can make sure that it sits to the right of the sidebar in our wider views, yet still be on top of the sidebar in our narrowest view.

Now our main content and sidebar are perfectly in their place, as you can see in figure 3.17.

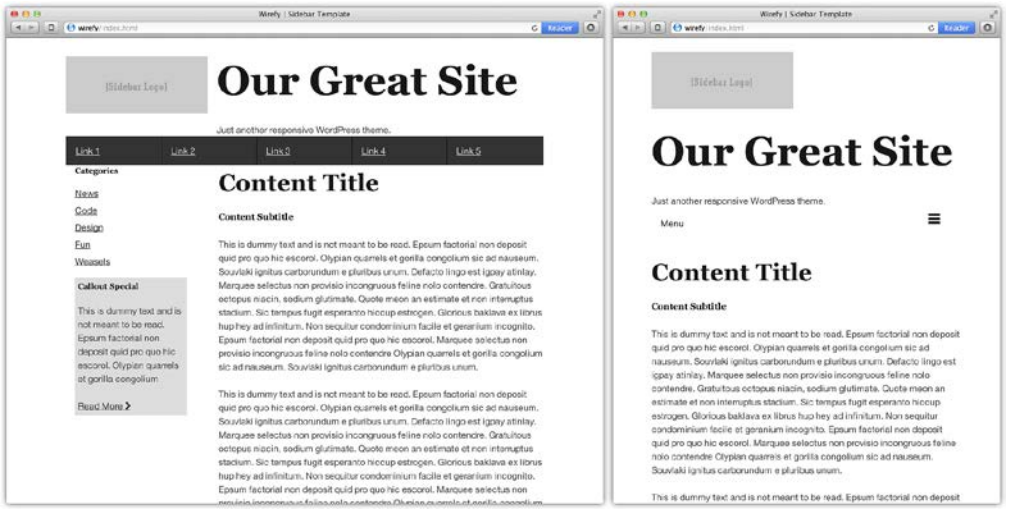


Figure 3.17 Wide view or narrow, our main content is right where we want it.

Our last adjustment is to the footer, which by default looks like figure 3.18:

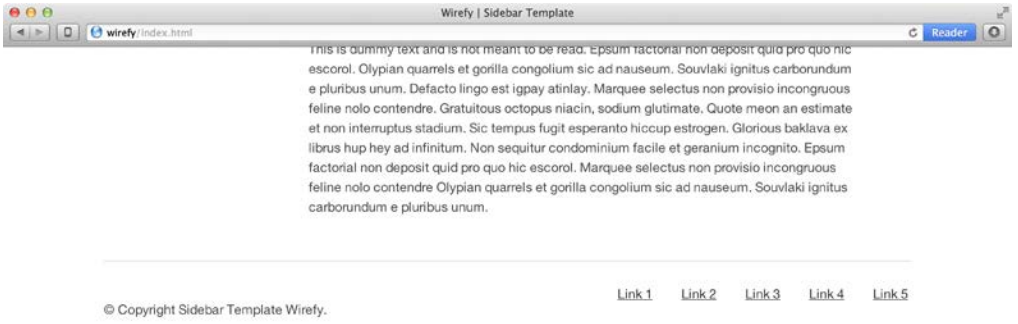


Figure 3.18 This is not what we want our footer to be.

We're going to replace the existing content of our footer element with the following:

Listing 3.4 Our New Footer Contents

```
<footer role="contentinfo" class="row">
<div class="four columns callout">
     Footer Widget Area 1
</div>
<div class="four columns callout">
     Footer Widget Area 2
</div>
```

```
<div class="four columns callout">
     Footer Widget Area 3
</div>
<div class="four columns callout">
     Footer Widget Area 4
</div>
</footer><!-- End Of Footer -->
```

And lastly, a little CSS adjustments:

Listing 3.5 Our New Footer CSS (in Sass form)

```
@media screen and (min-width: 55.5em) { #A
  /* 888 ================================================ */
  section[role="content"] {
    float: right;
  }

  footer[role="contentinfo"] .four.columns {
    width: 49%; #B

&:nth-of-type(3) { #C
      margin-left: 0;
    }
  }
}

@media only screen and (min-width: 61.5em) { #D
  /* 984 ================================================ */
  .logo {
     float:left;
     }
     nav{
     float:right;
     }

  footer[role="contentinfo"] .four.columns {
    width: 23.5%; #E
&:nth-of-type(3) {
      margin-left: 2%;
    }
  }
}
```

#A Everything in this block is set to a breakpoint equal to or greater than 888 pixels
#B This will set the width of our footer columns to 49%, leaving room for a little horizontal margin between them.
#C Since this is just a prototype, we're okay to use fancy CSS3 selectors like :nth-of-type here. Also, don't worry about the funny "&" notation, that's all part of Sass.
#D Next we'll set our styles for a breakpoint of 984 pixels or more.
#E 23.5%, plus requisite horizontal margins, will give us four evenly sized columns.

The results of all of this give us the three variations of the footer that we're looking for, in figure 3.19:

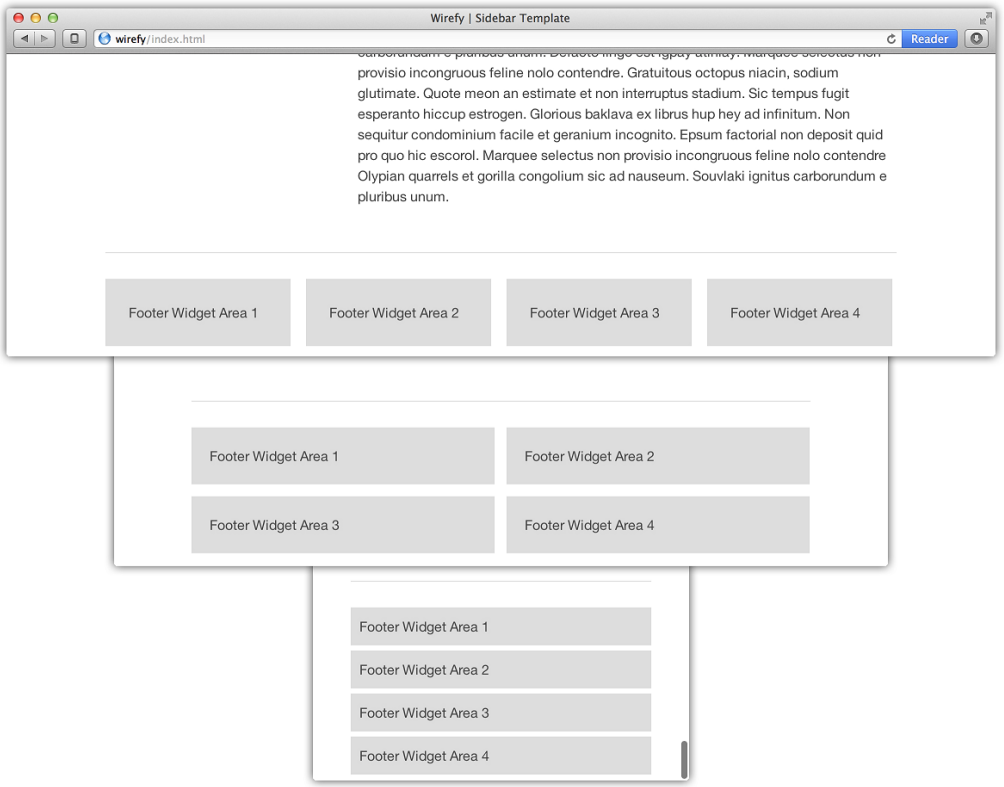


Figure 3.19 A footer full of widget areas.

If code is your thing, you might find Wirefy a more flexible solution to wireframing and prototyping than graphics programs or Flash-based wireframing tools. But a solid knowledge of HTML and CSS (preferably Sass) is required to work with it.

So far in this chapter, we've sketched out a home page by hand, and tried several various online wireframing tools to come up with an archives page. The only major template we have left is the single page template, for displaying a blog post or static content page. To be clear, you can have lots and lots of different templates for all sorts of arrangements of content, but as many of the templates share similar display properties, there's no need to wireframe every last one of them. Hitting the highlights of the three major types – home page, archive page, and single page – will give us enough of a roadmap once we're set to start building our theme.

3.3 Visual Design

Wireframes are useful and necessary and provide a crucial roadmap for building our theme, but no one is going to want to download a theme that is the web equivalent of stick figures. Certainly beauty is in the eye of the beholder, and that's just as true in web design as it is in other contexts, so what appeals to one person may not necessarily appeal to another. WordPress themes come in an astounding variety of visual designs, from the fun and flighty to the artistic and edgy, to the staid and business-like, and everywhere in between. See figure 3.20 for a few examples.

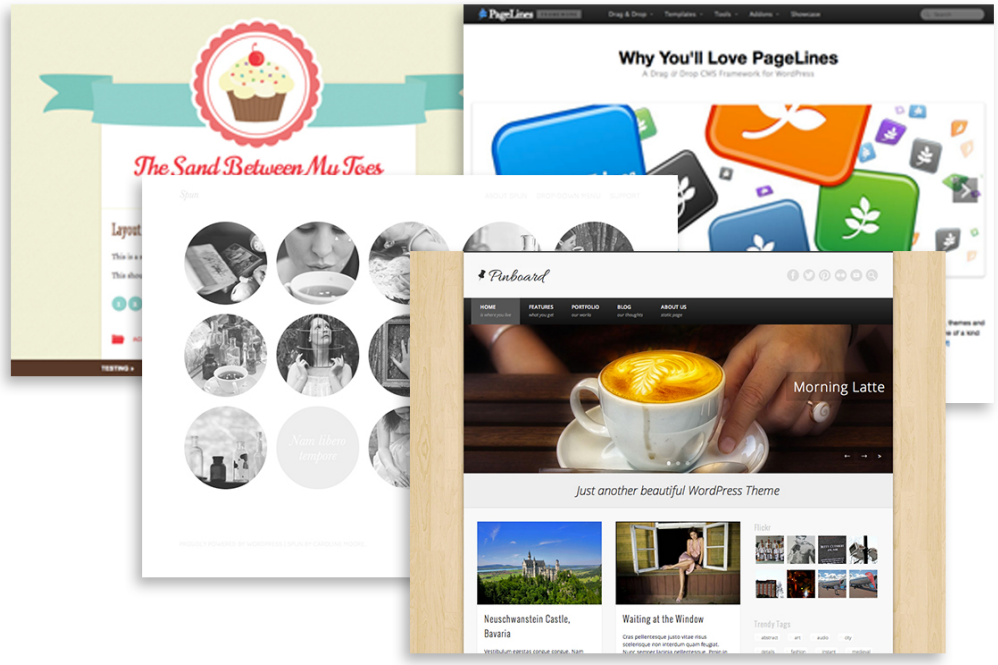


Figure 3.20 Themes pictured are, clockwise from top left, Buttercream, PageLines, Pinboard and Spun and are all available for free at the WordPress theme repository (http://wordpress.org/themes).

NOTE Actually, there are several popular themes that have little or no visual design to them. These usually fall into the category of "starter" themes, and serve as a basis on which to build a more stylish theme, either by altering the starter theme directly or creating a child theme. We'll examine these more in chapter x.

As the designer of our theme, we get to call the shots, and hopefully come up with something compelling that people will want to run on their WordPress sites. So let's set our tone. For our theme to have the broadest possible appeal, we'll want to keep the design

simple and clean. We'll use a neutral color scheme, highlighted with a strong accent color for one prominent feature of the site. Also, the typography should be simple and easy to read. And it'll help if we throw in a few options to allow people to customize their theme easily, without having to delve into code (we'll cover that in chapter x when we talk a bit about theme options).Now that we've established some of the criteria of our visual design, we'll take a look a couple of the tools and processes that we can employ to apply it to the wireframes we created in the previous section.

> **NEED MORE DESIGN INSPIRATION?** Media Queries (http://mediaqueri.es) is a showcase of the best responsive websites the Internet has to offer. Many of the sites featured are WordPress sites, although you'd have to actually click through to the site and view the source code to tell which as they aren't marked as such in the showcase.

3.3.1 Adobe Photoshop

Ah, Photoshop, the venerable old standard. Even in this age of changing processes – designing in the browser, fluid design, yaddayadda – a large amount of the work you do in web design will pass through Photoshop at some point in its lifecycle. For all the complaintsagainst it (such as being overcomplicated and rooted in the print design era of old) Photoshopremains the tool of choice for many graphic designers.

> **ALTERNATIVES TO PHOTOSHOP** Don't want to lay down the change for a subscription to Adobe's Creative Cloud? Low cost but capable alternatives exist on the Mac. Pixelmator (http://pixelmator.com) and Acorn (http://www.flyingmeat.com/acorn/) both have similar toolsets to Photoshop and handle layers. They are available on the Mac App Store and are less than the price of one month for a full Creative Cloud membership, combined.

But how does this fixed dimension tool fit into our responsive design process? With a little jury rigging we can use a single Photoshop document to contain the three variations of our design. We'll start with a large canvas, with plenty of room to stretch out both horizontally and vertically. Start a new document, but eschew the "web" size settings and go for something custom. 1,024 pixels wide by 2,000 pixels high (at 72 DPI) should be good (figure 3.21).

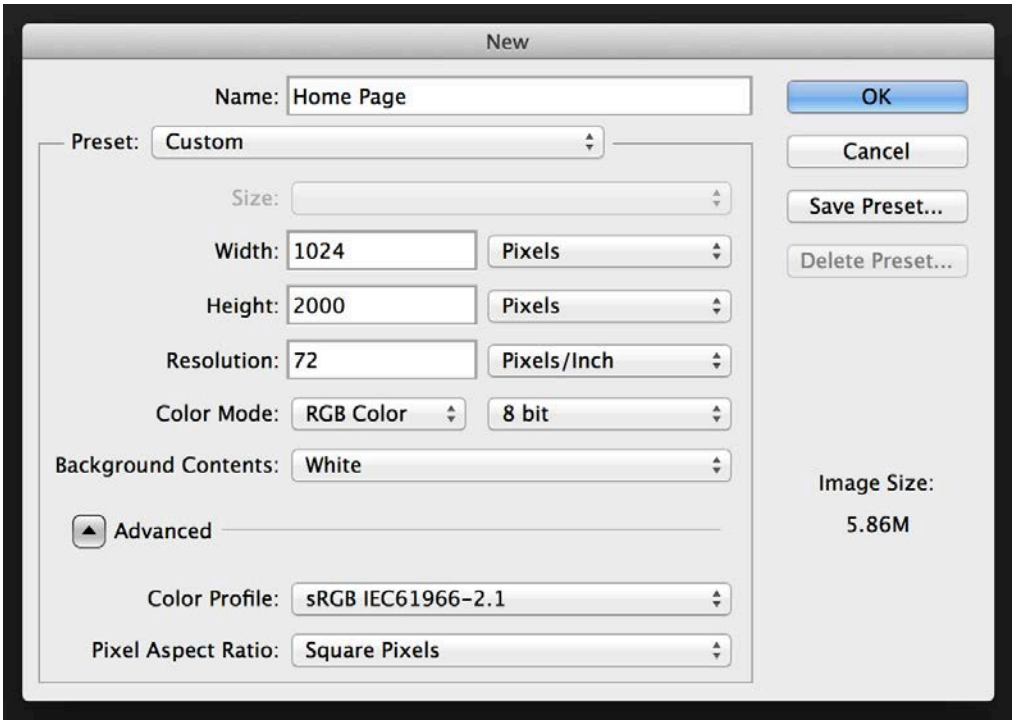


Figure 3.21 We all have to start somewhere, and setting up a nice, large canvas gives us room to play with.

Now that we have our canvas, we'll want to set up containers for each of the three different views we'll be working with. Go into your Layers window, and create four folders: Wide, Medium, Narrow and Background (for obvious reasons, make sure the Background folder is at the bottom of the window stack).Now, in each of the three width folders, create a layer called "Mask" (or create one and make copies). We're going to use these layers to "mask out" (just like Photoshop's native mask functions) the portion of the canvas we won't need for each particular view. At this point, we're just aiming for width and not worrying so much about height, so let's set a couple of vertical guides to help us out, one at 320 pixels and the other at 640 pixels. We should now have something that looks sort of like figure 3.22:

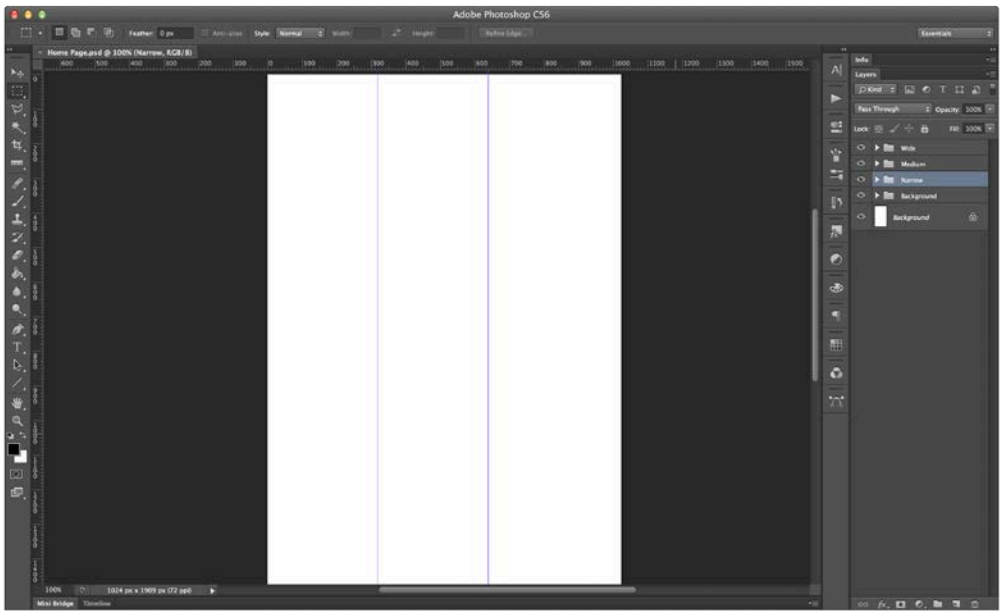


Figure 3.22We're setting the stage by setting our guides where we want our contexts to be.

Note that these are *not* our breakpoints! Figuring those out will come later. These are just the contexts from which we're going to base the dimensions of our design. Now, let's create those masks to hide what we don't want to see for each view.

On the "Mask" layers you created for Narrow and Medium, select the area to the right of the guide that sits at the appropriate context for that width. For example, we'll mask out the unused portion of the Narrow view by selecting everything to the right of the left guide, and filling it in with black, as in figure 3.23:

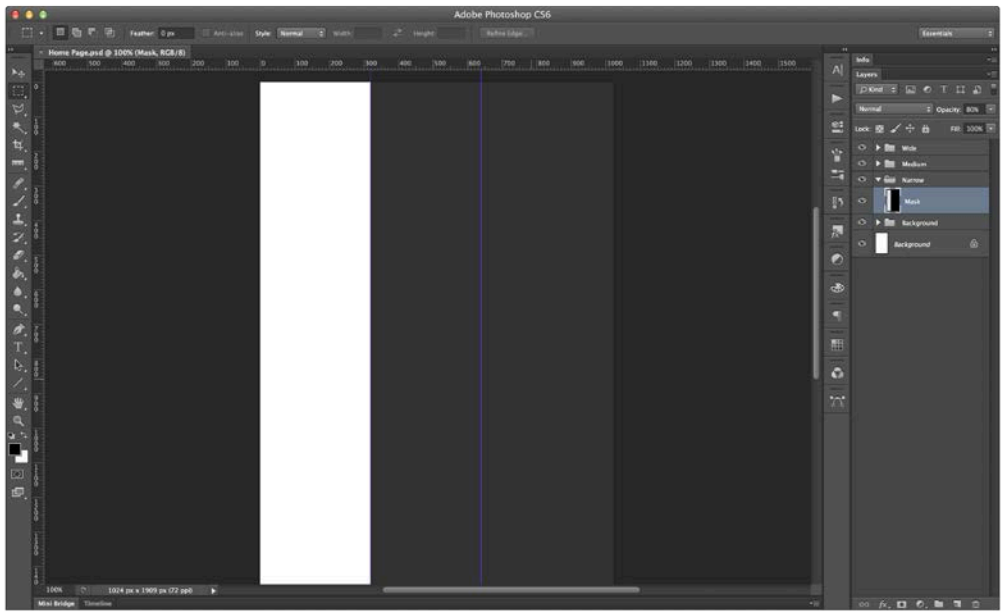


Figure 3.23 Now only our narrow canvas is visible.

What are we trying to accomplish, exactly? By constraining our visible canvas, yet keeping all of our views in one Photoshop document, we can more easily copy elements from one view to another, while keeping things organized. Also, we're not being as distracted by seeing the whole big canvas in front of us when we only need a portion of it. We're trying to give ourselves the best of both worlds by creating reusable objects but yet keeping everything in one neat and tidy package.

So next comes the design part. We already have our wireframes, so we know where everything goes. We just have to put it there now. For the sake of this book, let's assume that we've come up with something like figure 3.24:

Figure 3.24 The structure is the same as our sketched-out wireframes, but now it looks pretty (at least I hope it does).

Remember, we're going to provide a means for the downloader of the theme to alter some of the design elements, so don't get emotionally attached to the header image, the

logo or even the color scheme. Moving on, we want to come up with our medium view. Since we now have all of our elements in our narrow view, we can copy them *en masse* and place the copies in the Medium folder. With some repositioning, perhaps a bit of resizing, and cropping the header image slightly, we can come up with something that resembles figure 3.25:

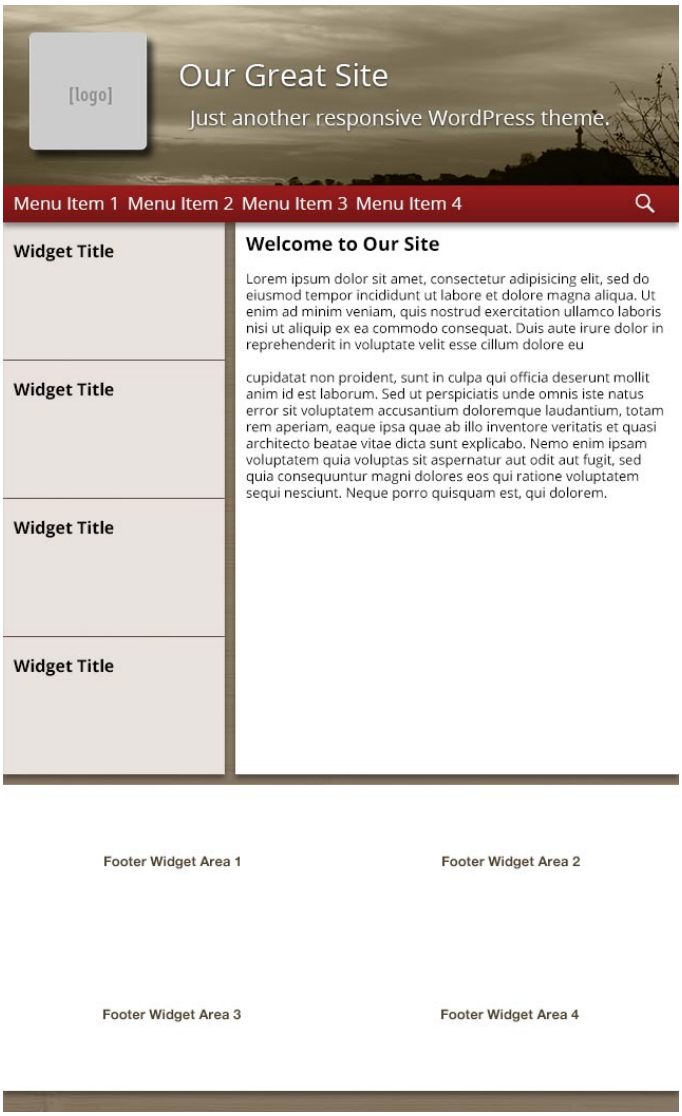


Figure 3.25 As with the wireframes, a lot of stuff stays the same, or at least similar, from view to view.

All we have left now is the wide view. By copying the elements in our medium view grouping to the Wide folder, we have all everything we need ready for their rearrangement. The results are in figure 3.26:

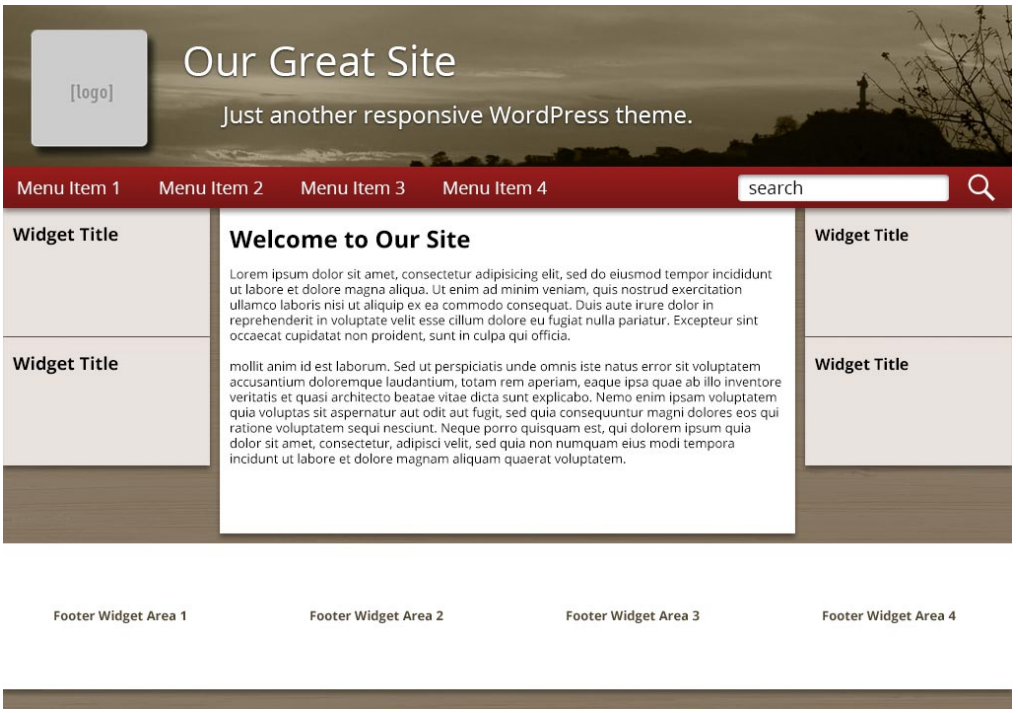


Figure 3.26 Our final view of the home page, spread out in all its wide glory.

A few more points: as the resolution changes, the font size will change as well, at least for some elements, especially headers. This is factored into the design. And you don't *have* to keep everything in one Photoshop document like I have here, but I think it does help with the comparison of how elements transition from one view to the next, while minimizing the amount of files necessary to go from design to development. The more files you have, the more difficult it can be to manage all the latest revisions and make sure everyone in your team stays on the same page.

3.3.2 Style Tiles

Now that we have a Photoshop comp of the home page in place, do we really need to comp out all the other views for the other templates in our theme? I think not, since a lot of stuff like headers and footers stay pretty much the same. Instead, let's use a different visual design tool for finishing off our template design.

Style tiles (http://styletil.es/) are similar to style boards used by interior designers. Interior designers present fabric swatches and paint swatches to their clients to give them a representation of what the finished room will look like. Style tiles do the same thing for web design by presenting the color scheme, font treatment, textures, etc. that will be used in the overall design. The concept was created by Twitter designer Samantha Warren and is more about creating a "design system" rather than endless piles of Photoshop comps.

By abstracting out the visual design elements from the layout of the page, style tiles offer greater flexibility in creating a responsive design than the full page comps we created earlier. With them, you can set the standards of the visual design, while referring to the wireframes you created earlier will show you how those elements should be arranged.

Style Tiles in the Client Process

The style tiles methodology was developed as a means of extracting a client's desires for a website and using that information to present variations that they can pick and choose from. It starts by presenting the client with a survey or other type of questionnaire where you can present contrasting options, with examples if necessary, and have the client stakeholders pick on a sliding scale where they want their site to fall. As Samantha Warren explains in her article for *A List Apart* (http://alistapart.com/article/style-tiles-and-how-they-work), "The answers you get are key to forming the emotional bond between a two-dimensional visual design concept and the passion that the client feels for their brand."

When it comes to applying style tiles to responsive design, Warren goes on to state, "Creating a mockup for every possible device or screen size is inefficient and confusing to a client. Style tiles are the cornerstone of a solid design system that sets client expectations and communicates the visual theme to all the project team members. Designing a system rather than site pages gives your team the tools to create a living, breathing website."

In our case, we are the client. We've already established what we want; we just need to flesh out the details a little more. With that in mind, let's create a style tile for the listing of articles in our archive template using the style tile Photoshop template Warren provides, customized to meet our needs (figure 3.27):

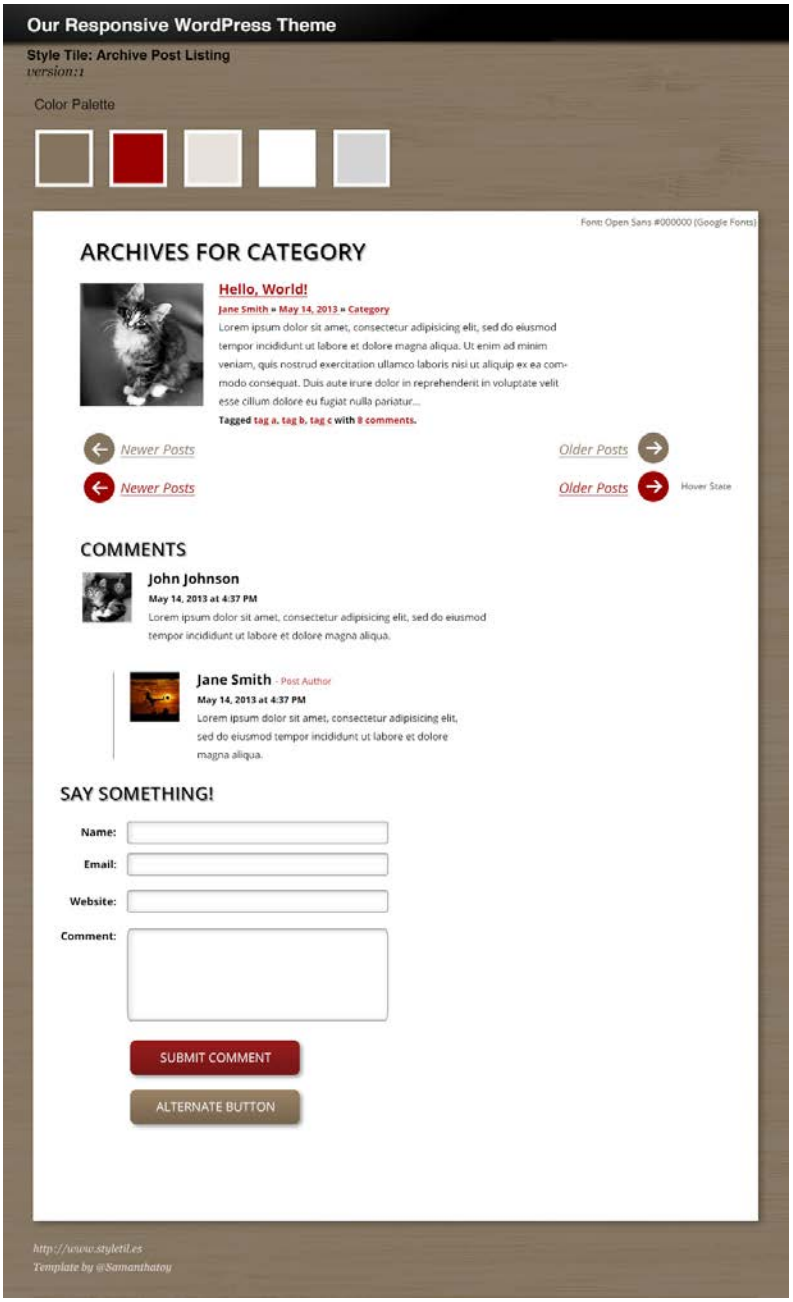


Figure 3.27 Applying our visual design as a style tile, we have a sample post entry, plus comment listing and a few other elements, that looks something like this.

Also in figure 3.27, I've added an example of what our comments form will look like, showing examples of form elements in the process. Obviously, we're not going to have comment forms after each article in the archive template, but that's not the point of style tiles. It's less about what elements go together on a particular page, but rather representing the elements that will make up the site as a whole.

In theory, style tiles are meant as an alternative to a full-blown comp, but with more specificity than a mood board, so you wouldn't necessarily create a comp as we did for the home page and then follow that up with style tiles. I'm presenting style tiles here as an alternative visual design tool that you can include in your responsive web design process.

In reality, you're likely to find that a combination of the traditional comp process along with style tiles will be necessary. For headers and footers especially, which can often be quite complex and change significantly from narrow to wide views, having full comps of these elements is essential before building them out in code. Other areas of your theme, however, change less from one width to another, and only need the guidance in font treatment and colors that style tiles provide.

As we move along in the process, we can add more elements to our style tile if we think we'll be reusing them in our theme. That way, we have a visual reference for them, and can envision them in the context of our theme before we go to code.

3.4 Summary

Every good carpenter knows that the first rule of carpentry is "measure twice, cut once." That is to say, plan out what you're going to do before you go and do it. In the web design world, that usually involves creating wireframes, or blueprints that map out the structure of your site, and some sort of visual design piece, whether it's style tiles or a full-fledged comp.

The end goal is the same as the carpenter's: plan out your theme before attempting to build it. Structurally and visually, see what works and what doesn't. Experiment with colors and typography where the cost of changing them is as low as clicking on a few options. This isn't to say that you can't change things as your move forward in your build process. Inevitably, something won't work right in the browser, or you'll decide that your color choice or image placement was sub-par and needs improving. But at least with planning you have a solid foundation from which to start, and a guide that you can refer to as your theme starts to take shape.

In our next chapter, we'll finally be heading to the browser to set up a WordPress environment custom-tailored towards theme development.

4

Building the Prototype

This chapter covers

- Building our site's three major templates as HTML prototypes
- Styling our home page using CSS

In chapter two, we learned about what every responsive web site needs to have in order to function properly: a properly defined meta viewport tag, a fluid grid, media queries, and responsive images. We also learned about the typical characteristics of many WordPress sites, such as the widget areas, primary navigation and WordPress' variety of image functionality.

Then in chapter three, we planned out what our theme will be. First, we decided on our theme's purpose. We then laid out its basic structure using a variety of wireframing tools. Finally, we created a visual design using Photoshop comps and style tiles.

Now, finally, we're ready to start our coding. But before we jump into coding the actual theme, we'll start with building a static prototype. Static prototypes help a developer flesh out the front-end issues of a theme before having to interact with the WordPress functionality.

I'm going to make a few assumptions now that we're ready to delve into code:

1. You have a development environment running a webserver. Usually, your personal computer (be that Mac, Windows or Linux) can be set up this way, but I'll leave you to figure out the best way for your needs. Searching Google for "setting up a webserver on [insert operating system name here]" will reveal instructions for how to do it.
2. You are familiar enough with HTML and CSS to follow along the code samples in this chapter. (In fact, being proficient in HTML5 would be best since this book isn't the place to cover the new elements and changes in syntax since HTML 4 and XHTML.) We won't get into much JavaScript yet, and won't touch PHP at all in the prototyping phase.

3. You have a program that will allow you to open up layered graphics files. Whether you're using the same theme we're building in this book or working off your own design, we'll need to reference the visual design as we build the prototype.
4. You have a code editor that outputs plain text. Sublime Text 2 is currently my editor of choice, but I tend to change with the seasons. There always seems to be a new text editor out there with cool new features.

Whenever I start a new WordPress theme project for a client, I *always* start with a static prototype. I don't consider this step to be optional. Usually, when creating a static prototype, we can build out 80-85% of the final CSS we'll use in the actual WordPress theme. That's not to say there won't be adjustments needed to the CSS down the road, but we can get much of it out of the way here.

Building a static prototype also allows us to see what will work and what won't in our fluid grid and, more importantly, we can start to get a good idea of where our breakpoints should go. We can also nail down typography, colors, and image placement and see how all the pieces work together.

Once we're done building our static prototype, we can take the same CSS file and use it to start our templating. We can even incorporate much of the HTML we code into our WordPress templates, breaking up the components – such as the header, main content, and widget areas – into the various template files and replacing our dummy copy with the WordPress codes to draw on the website's real information from the database.

4.1 The Markup

As Rogers and Hammerstein once wrote, "Let's start at the very beginning. A very good place to start." And so we shall, by creating the HTML markup for our three different template pages: the home page (index.html), the archive page (archive.html) and the single article page (single.html). Without any styling applied, these pages will look quite dull, but that's okay. We'll get to the CSS soon enough.

4.1.1 Home Page

With our prerequisites in section 4.1.1 established, fire up your text editor, and open the comp that we created in chapter three in your graphics program. We'll begin coding our home page prototype with a basic, minimal HTML5 document in listing 4.1.

Listing 4.1 Creating Our index.html Page

```
<!DOCTYPE html> #A
<html>
<head>
  <title>Responsive WordPress Theme Prototype | Home</title>
  <meta charset="utf-8"> #B
  <meta name="viewport" content="width=device-width, initial-scale=1"> #C
</head>

<body>
```

```
</body>
</html>
```

#A HTML document types have been simplified greatly with HTML5. This is all you need to create a valid, standards-compliant HTML5 web page.
#B HTML5 requires us to define a character set. Use "utf-8" for the Unicode character set.
#C Look familiar? Don't forget your meta viewport tag.

THE HEADER

Once our basic page is established, it's time to start filling in our content, starting with the header. Using our comp for reference, and using the latest HTML5 elements along with WAI-ARIA roles, we'll write out our header code in listing 4.2.

Listing 4.2 Adding Our Header HTML

```
<body>

  <header role="banner"> #A
    <a href="/"> #B
      <img src="http://placehold.it/123x123&amp;text=[logo]" class="logo">
#C
      <h1>Our Great Site</h1> #D
      <div class="subhead">Just another responsive WordPress theme.</div>
    </a>
  </header>

</body>
```

#A "role='banner'" is our WAI-ARIA role attribute for a site-wide header.
#B One of the main roles of a header is to provide quick access back to the home page. We can accomplish this by enclosing the entire header contents in an <a> tag, which is allowable in HTML5.
#C placehold.it is one of many sites that generate placeholder images for websites. We'll be using several over the course of the project.
#D Our site title, in the <h1> tag, and our subtitle will both be generated from WordPress data in our final theme. We're just using more placeholder values here, as we will through the rest of the prototype.

HTML5 and WAI-ARIA Roles

WAI-ARIA (Web Accessibility Initiative - Accessible Rich Internet Applications) roles provide semantics where there aren't any, mainly to web applications (hence the name). Their intent is to provide a meta language that users of assistive technology, such as screen readers, can access web pages and web applications when visual cues aren't possible.

Even though HTML5 provides more semantic markup elements than HTML 4, they still lack the uniqueness we need when used in certain contexts. For example, the `<header>` element can denote the main header of a web page, but it can also denote the header of a specific portion of the page, such as a widget. To tell screen readers that a `<header>` element is the banner header that will carry throughout the entire website, we can give it the ARIA landmark role "banner" which signifies that the content contained

within is site-specific, rather than page-specific. For a list of all ARIA landmark roles, which give screen readers information about the structure of a web page, see "Using WAI-ARIA Landmarks – 2013" at http://blog.paciellogroup.com/2013/02/using-wai-aria-landmarks-2013/.

Most of the ARIA landmark roles we'll be using are unique on a page, such as "main" and "banner." In that regard, they are like IDs that can also be assigned to elements. However, in CSS you target ARIA roles as you would any other non-class, non-ID HTML attribute, using the `[attribute="value"]` structure, which has the same level of CSS specificity as classes. We'll examine the implications of this when we start crafting our stylesheet. For more reading on this, I suggest Jeremy Keith's post on the subject at http://adactio.com/journal/4267/.

Now that we have a little (okay, *very* little) content on the page, we can see what it looks like in the browser, in figure 4.1:

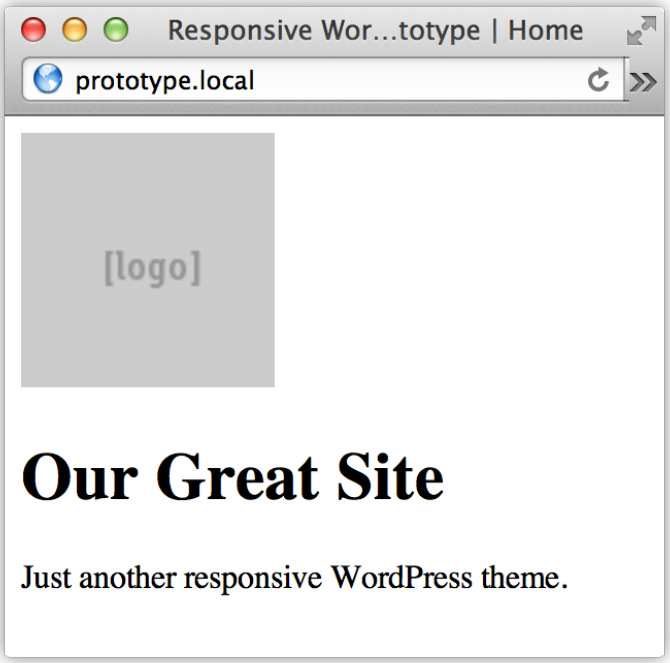


Figure 4.1 The beginnings of our prototype as viewed in a browser.

In figure 4.1, we're viewing the image in our "narrow" mode, both because we're building our site mobile-first, and because it doesn't really matter at this point. Let's move on.

NAVIGATION

Next, we have our navigation. Since we're building a prototype for a WordPress theme, we should be mindful of the code that WordPress is likely to generate, even though we're not using it quite yet. As we saw in chapter two, the default way that WordPress outputs menu code is shown in listing 4.3.

Listing 4.3 Default Navigation Menu Markup

```
<div class="menu-header">
  <ul id="menu-short" class="menu">
    <li id="menu-item-828" class="menu-item menu-item-type-custom menu-
item-object-custom menu-item-828"><a
href="http://wpthemetestdata.wordpress.com/">Home</a></li>
    <li id="menu-item-874" class="menu-item menu-item-type-post_type menu-
item-object-page menu-item-874"><a
href="http://localhost/?page_id=703">Blog</a></li>
    <li id="menu-item-875" class="menu-item menu-item-type-post_type menu-
item-object-page menu-item-875"><a
href="http://localhost/?page_id=832">About The Tests</a>
      <ul class="sub-menu">
        <li id="menu-item-876" class="menu-item menu-item-type-post_type
menu-item-object-page menu-item-876"><a
href="http://localhost/?page_id=501">Clearing Floats</a></li>
        <li id="menu-item-877" class="menu-item menu-item-type-post_type
menu-item-object-page menu-item-877"><a
href="http://localhost/?page_id=155">Page with comments</a></li>
      </ul>
    </li>
    <li id="menu-item-879" class="menu-item menu-item-type-post_type menu-
item-object-page menu-item-879"><a
href="http://localhost/?page_id=146">Lorem Ipsum</a></li>
  </ul>
</div>
```

That's a lot more than we need right now. It's also a bit old-fashioned in that it uses non-semantic `<div>` tags when we have an HTML5 element specifically for navigation. We're also going to need an element with which to toggle the visibility of the navigation when we're in narrow views, and we'd still like a "Skip to content" link for accessibility (because not all accessibility issues are vision-related). Fortunately, we can change that in WordPress. So with all of that in mind, let's pare down the navigation HTML for our prototype in listing 4.4.

Listing 4.4 Our Primary Navigation Prototype

```
<a class="screen-reader-text skip-link" href="#content">Skip to
content.</a>

<div class="navbar clearfix">

  <!-- Begin Primary Navigation -->
  <nav class="primary-navigation" role="navigation">
```

```
    <a href="#!" id="toggle-primary-nav" class="toggle-menu">Menu</a>

    <div class="primary-navigation-container">
      <ul>
        <li><a href="#!">Menu Option 1</a></li>
        <li class="current_page_item"><a href="#!">Menu Option 2</a></li>
        <li><a href="#!">Menu Option 3</a>
          <ul class="sub-menu">
            <li><a href="#!">Sub Option 1</a></li>
            <li><a href="#!">Sub Option 2</a></li>
          </ul>
        </li>
        <li><a href="#!">Menu Option 4</a></li>
      </ul>
    </div>
  </nav>
  <!-- End Primary Navigation -->

</div>
<nav class="primary-navigation" role="navigation"> #A
  <a href="#!" class="toggle-menu">Menu</a> #B
  <a class="screen-reader-text skip-link" href="#content">Skip to
content.</a>

  <div class="primary-navigation-container">
    <ul>
      <li><a href="#!">Menu Option 1</a></li> #C
      <li class="current_page_item"><a href="#!">Menu Option 2</a></li> #D
      <li><a href="#!">Menu Option 3</a>
        <ul class="sub-menu"> #E
          <li><a href="#!">Sub Option 1</a></li>
          <li><a href="#!">Sub Option 2</a></li>
        </ul>
      </li>
      <li><a href="#!">Menu Option 4</a></li>
    </ul>
  </div>
</nav>
```

#A "navigation" is the ARIA role a listing of links that provides navigation for the website.
#B We'll need an element to control the visibility of the menu in our narrow view. This <a> tag combined with a little JavaScript will do the job.
#C "#!" is just filler. If the link is clicked, nothing will happen. Yet we need *some* sort of href attribute in order to let the browser know that this is a link.
#D We're using the WordPress class that denotes which page of the menu options we're currently on, because we'll probably want this as a styling hook.
#E Let's include a submenu, since that's a likely situation users of our theme might encounter.

When we reload our prototype in the web browser, our main navigation appears (figure 4.2):

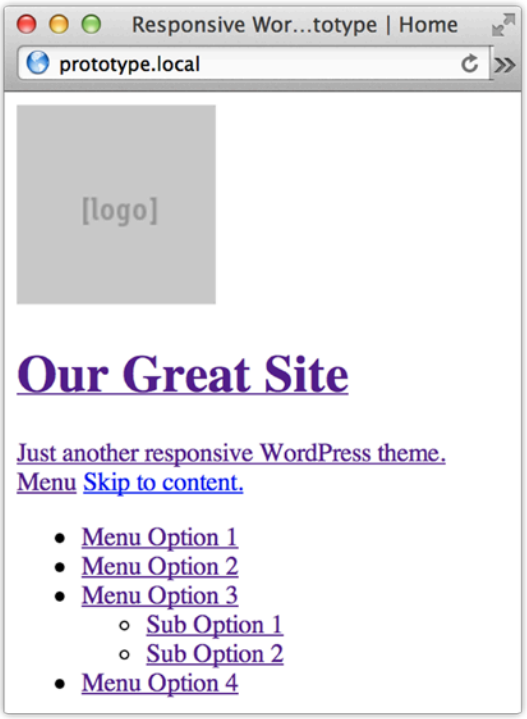


Figure 4.2 It doesn't look like the horizontal nav of our dreams, but we'll get there yet, I promise.

SEARCH BAR

Next, we need to build our search bar. This is another element for which WordPress has a default structure, although we can override it in our template later if we choose. Let's take a look at the WordPress default code first in listing 4.5.

Listing 4.5 Default WordPress Search Form

```
<form role="search" method="get" class="searchform"
action="http://prototype.local/"> #A
  <label>
    <span class="screen-reader-text">Search for:</span> #B
    <input type="search" placeholder="Search &hellip;" value="" name="s"
title="Search for:" />
  </label>
  <input type="submit" class="searchsubmit" value="Search" />
</form>
```

#A I've put in the URL for our prototype as a demonstration. Obviously, this would be the URL of whatever site WordPress was running on.
#B Never forget accessibility. WordPress automatically adds the "screen-reader-text" class so that if we want to hide it from regular browsers, but have it be accessible by screen readers, we have that option.

There are actually two versions of the default WordPress search form: an HTML5 version and a non-HTML5, XHTML version. The output in listing 4.x version is what browsers capable of rendering HTML5 will see.

Since we're doing the whole "expand to see the form in narrow" thing, we'll need a hook to toggle the visibility of the search form similar to what we did for the navigation. For that, we'll add the following line just above our form:

```
<a href="#!" class="toggle-search">□</a>
```

What's that funny looking mark inside our anchor tag? That's actually a UTF-8 character (a "glyph," to be precise), copied from the icon font Genericons, found at http://genericons.com. We'll be using the search icon, which appears as a magnifying glass, to indicate where to click in order to make the search form appear. When the Genericons font is applied to that glyph, the result will look something like figure 4.3:

Figure 4.3 Using the icon font "Genericons", we can create a search icon that is completely scalable without having to use rasterized images.

Introducing Icon Fonts

Icon fonts are a relatively new means of introducing symbolism into a web page in a responsive way. Common web symbols, such as the search icon, the three line "menu" icon, social media service icons, and many, many more are increasingly available as glyphs in various available icon fonts.

Genericons (http://genericons.com), an open-source icon font produced by Automattic, contains, not surprisingly, a large number of icons related to common blogging tasks, such as RSS, commenting, categorizing and tagging. All of the icon font symbols we'll be using on our project will come from Genericons.

One of the main benefits of using icon fonts is how they can easily scale by simply using CSS. A drawback is that their characters often look like gibberish when included in the code of plain text editors because their characters have no meaning unless the proper font is applied. So while our source code shows nothing but a plain box, once we tell the web browser, via CSS, that this is a search icon, our little magnifying glass will magically appear.

We'll cover more pros, cons and features of icon fonts later in this book.

This looks pretty good as-is, so we'll add it to our prototype document, just under our primary navigation (figure 4.4).

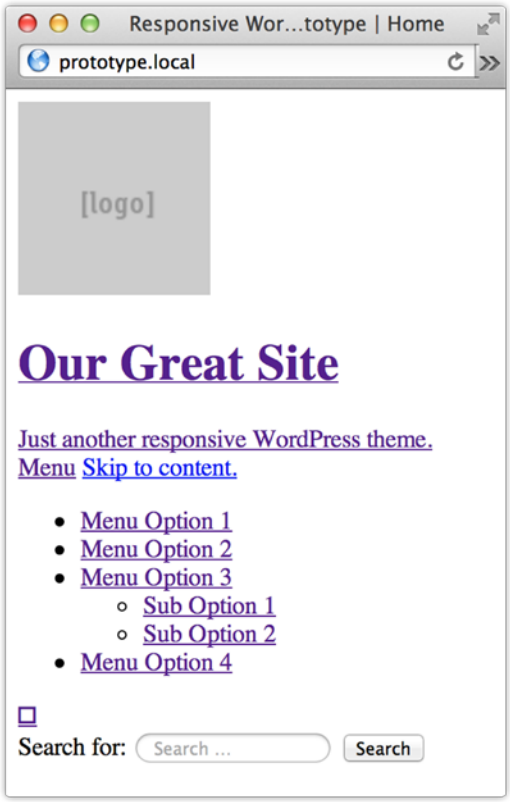


Figure 4.4 Our search form appears. Our funny-little box is still there, waiting to be turned into a beautiful swan, er, magnifying glass.

Main Content and Sidebars

Now we've come to the heart of our page, our main content with sidebars on either side. Since we're building our prototype mobile first, we should build the content in the order we want to see them on the mobile site. Referring to our comp and wireframes, that order should be:

1. Main content
2. First sidebar
3. Second sidebar

The first sidebar is the one that will sit to the left of the main content in the widest view while the second sidebar will sit to the right. Hmm, that sounds like we're going to have to move stuff around later, but don't worry about that yet. Just keep it in the back of your mind. Let's add our placeholder content now (listing 4.6).

Listing 4.6 Main Content and Sidebars Markup

```
<!-- Begin Main Content -->
<main id="content" role="main"> #A
  <h1>Welcome to Our Site</h1>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu</p> #B
  <p>cupidatat non proident, sunt in culpa qui officia deserunt mollit anim
id est laborum. Sed ut perspiciatis unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa
quae ab illo inventore veritatis et quasi architecto beatae vitae dicta
sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut
odit aut fugit, sed quia consequuntur magni dolores eos qui ratione
voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem.</p>
</main>
<!-- End Main Content -->
<!-- Begin Primary Sidebar -->
<section id="sidebar-primary" class="sidebar"> #C
  <!-- Begin First Sample Widget -->
  <aside class="widget widget_text"> #D
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End First Sample Widget -->
  <!-- Begin Second Sample Widget -->
  <aside class="widget widget_text">
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End Second Sample Widget -->
</section>
<!-- End Primary Sidebar -->
<!-- Begin Secondary Sidebar -->
<section id="sidebar-secondary" class="sidebar">
  <!-- Begin Third Sample Widget -->
  <aside class="widget widget_text">
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End Third Sample Widget -->
  <!-- Begin Fourth Sample Widget -->
```

```
  <aside class="widget widget_text">
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End Fourth Sample Widget -->
</section>
<!-- End Secondary Sidebar -->
```

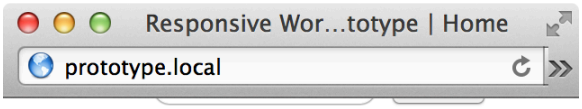
#A Using the new HTML5 <main> element with the ARIA "main" landmark role may seem redundant, but they serve different purposes, so we'll include them both. Our "content" ID serves as a target for our earlier "skip navigation" link.

#B Lorem ipsum (or Greek, or placeholder) copy can be generated by Photoshop or from a number of different websites such as http://lipsum.org.

#C Yes, I referred to these widget containers in chapter two as "widget areas," but since they are sidebars in the truest sense of the word, that's how we'll assign the IDs and classes.

#D The end-user of this theme can put whatever kind of widget he wants in here, so we'll just fill it with some more lorem ipsum copy.

We're nearly done! Let's take a look at our results so far in a narrow-sized browser in figure 4.5:



Welcome to Our Site

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu

cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem.

Widget Title

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris

Widget Title

Figure 4.5 It's not looking any prettier, but our prototype is definitely looking wordier.

FOOTER WIDGET AREAS

This leaves only one major section of the page: the footer widget areas. Let's get to it in listing 4.7.

Listing 4.7 Finally, Our Footer Widget Areas

```
<!-- Begin Footer -->
<footer role="contentinfo">
<!-- Begin Footer Widget Area 1 -->
<section id="footer-widget-1" class="footer-widget-area"> #A
  <aside class="widget widget_text">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
</section>
<!-- End Footer Widget Area 1 -->
<!-- Begin Footer Widget Area 2 -->
<section id="footer-widget-2" class="footer-widget-area">
  <aside class="widget widget_text">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
</section>
<!-- End Footer Widget Area 2 -->
<!-- Begin Footer Widget Area 3 -->
<section id="footer-widget-3" class="footer-widget-area">
  <aside class="widget widget_text">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
</section>
<!-- End Footer Widget Area 3 -->
<!-- Begin Footer Widget Area 4 -->
<section id="footer-widget-4" class="footer-widget-area">
  <aside class="widget widget_text">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
</section>
<!-- End Footer Widget Area 4 -->
</footer>
<!-- End Footer -->
```

#A Whereas in listing 4.6 we had true sidebars, you can't call our footer widget areas sidebars. So we'll call them, well, footer widget areas.

One more refresh of the browser and *voilà*! Our prototype is complete (well, the markup is, at least). See figure 4.6:

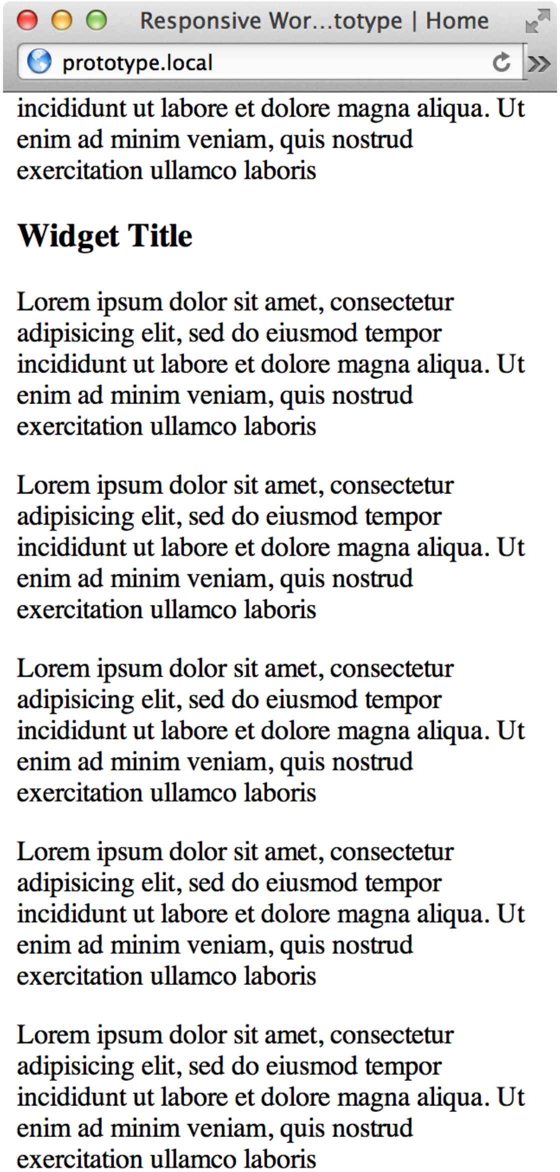


Figure 4.6 Our markup is complete, but still lacks a certain, *je ne sais quoi*.

4.1.2 Archive Page

So with our sample content in place for the home page, it should be quick work to create our archive and single-article pages as well, since much of the markup will be the same. Let's tackle the archive page first. Nearly everything on our home page conveys to the archive page, except for the secondary sidebar and the main content itself. Instead, our `<main>` element will have an archive listing similar to what we mocked up in our style tile, including an article listing and links to the previous and next pages of the archive. With those considerations in place, we'll copy index.html into a new document called "archive.html" and our main content and sidebar sections will look like listing 4.8.

Listing 4.8 Our Main Content for archive.html

```
<!-- Begin Main Content -->
<main id="content" role="main">
  <!-- Archive Page Header -->
  <h1>Archives for "Category"</h1> #A
  <!-- Begin First Post Excerpt -->
  <article>
    <img src="http://placekitten.com/g/150/150" alt="Adorable
placekitten."> #B
    <h1>Posting Number One</h1>
    <!-- Begin Top Meta Information -->
    <div class="content-meta">
      <span class="content-meta-author"><a href="#!">Jane Smith</a></span>
» #C
      <span class="content-meta-date"><a href="#!">May 14, 2013</a></span>
»
      <span class="content-meta-cateogry"><a href="#!">Category</a></span>
    </div>
    <!-- End Top Meta Information -->
    <!-- Begin Post Excerpt Content -->
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu…</p>
    <!-- End Post Excerpt Content -->
    <!-- Begin Bottom Meta Information -->
    <div class="content-meta">
      <span class="content-meta-tags">Tagged: <a href="#!">tag 1</a>, <a
href="#!">tag 2</a>, <a href="#!">tag 3</a></span> with
      <span class="content-meta-comments"><a href="#!">8
comments</a></span>.
    </div>
    <!-- End Bottom Meta Information -->
  </article>
  <!-- End First Post Excerpt -->
[… repeat three more times, with subtle variations …]
</main>
<!-- End Main Content -->
<!-- Begin Bottom Newer/Older Posts Links --> #D
<nav class="navigation">
  <div class="nav-previous"><a href="http://wordpress36/?paged=3">Older
```

```
posts</a></div>
  <div class="nav-next"><a href="http://wordpress36/">Newer posts</a></div>
</nav><!-- #nav-below -->
<!-- End Bottom Newer/Older Posts Links -->
<!-- Begin Primary Sidebar -->
<section id="sidebar-primary" class="sidebar">
  <!-- Begin First Sample Widget -->
  <aside class="widget widget_text">
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End First Sample Widget -->
  <!-- Begin Second Sample Widget -->
  <aside class="widget widget_text">
    <h1>Widget Title</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris</p>
  </aside>
  <!-- End Second Sample Widget -->
</section>
<!-- End Primary Sidebar --> #E
```

#A If this were an actual WordPress template file, we'd have code here to replace the word "Category" with the actual category, keyword, author, etc. that this archive was for.
#B http://placekitten.com is the cutest darned site on the Interwebs.
#C Good thing we set our character set to UTF-8 in listing 4.x so that our web page can handle funky characters like these European-style double quotes.
#D We need some means for visitors to page through the lists of archived posts. This navigation structure is a common way to do that.
#E Our prototype primary sidebar is exactly the same as on the home page, but our secondary sidebar is gone, baby, gone.

As I mentioned in chapter three, we're including four posts in our sample archive page, which is just long enough to give us a good sampling of what our archive will look like as content repeats down the length of the page. Varying the length of the excerpt (in case we have really short posts that don't reach the length needed to be truncated), omitting featured images, adjusting the number of categories and tags, etc. will help us get a feel for the different scenarios our theme might face with a stranger's data.

The results of the changes can be seen in figure 4.7.

Figure 4.7 Some sample content thrown into a prototype archive page.

4.1.3 Single Article Page

Now all we have to do is copy our archive.html page to single.html for our single article page. The sidebars on this page will stay the same as archive.html, but our main content

will be a single post or page content. On this template we'll also have our comments and comment form. Let's dummy that up in listing 4.9.

Listing 4.9 The Main Content and Comment Form for single.html

```
<main id="content" role="main">
  <article>
    <section class="content">
      <!-- Post Title -->
      <h1>Post Number One</h1>
      <div class="content-meta">
        <span class="content-meta-author">Posted by <a href="#!">Jane
Smith</a></span> on <span class="content-meta-date"><a href="#!">May 14,
2013</a></span><br>
        <span class="content-meta-cateogry">Category: <a
href="#!">Category</a></span> | <span class="content-meta-tags">Tags: <a
href="#!">tag 1</a>, <a href="#!">tag 2</a>, <a href="#!">tag 3</a></span>
      </div>
      <div class="featured-image"><img
src="http://placekitten.com/g/150/150" alt="Adorable placekitten."></div>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu</p>

[ …more content here… ]

      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu</p>
    </section>
    <!-- Begin Navigation to Previous and Next Articles -->
    <nav class="navigation">
      <div class="nav-previous"><a href="#!" rel="prev">Previous Article
Title Here</a></div>
      <div class="nav-next"><a href="#!" rel="next">Next Article Title
Here</a></div>
    </nav>
    <!-- End Navigation to Previous and Next Articles -->
    <section id="comments" class="comments">
      <h1 id="comments-title">Comments</h1>
      <!-- Begin Comments -->
      <ol class="commentlist">
        <li class="comment">
          <div class="comment-author">
            <img
src="http://0.gravatar.com/avatar/ad516503a11cd5ca435acc9bb6523536">
            <cite><a href="#!">John Johnson</a></cite>
          </div>
          <div class="comment-meta commentmetadata">
            May 17, 2013 at 2:01 AM
          </div>
```

```
          <div class="comment-body">
            <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
          </div>
          <div class="reply">
            <a href="#!">Reply</a>
          </div>
          <!-- Begin Comment Reply -->
          <ul class="children">
            <li>
              <div class="comment-author">
                <img
src="http://0.gravatar.com/avatar/ad516503a11cd5ca435acc9bb6523536">
                <cite><a href="#!">Tracy Rotton</a></cite> <span
class="author-comment">- Post Author</span>
              </div>
              <div class="comment-meta commentmetadata">
                June 3, 2013 at 10:58 PM
              </div>
              <div class="comment-body">
                <p>Lorem ipsum dolor sit amet, consectetur adipisicing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua.</p>
              </div>
              <div class="reply">
                <a href="#!">Reply</a>
              </div>
            </li>
          </ul>
          <!-- End Comment Reply -->
        </li>
      </ol>
      <!-- End Comments -->
      <!-- Begin Comment Form -->
      <div class="comment-form">
        <h2>Say Something!</h2>
        <form action="#!" method="post">
          <p class="comment-notes">Your email address will not be
published. Required fields are marked <span class="required">*</span></p>
          <p class="comment-form-author">
            <label for="author">Name <span
class="required">*</span></label>
            <input id="author" name="author" type="text" value="" aria-
required='true'>
          </p>
          <p class="comment-form-email">
            <label for="email">Email <span
class="required">*</span></label>
            <input id="email" name="email" type="email" value="" aria-
required='true'>
          </p>
          <p class="comment-form-url">
            <label for="url">Website</label>
            <input id="url" name="url" type="url" value="">
          </p>
          <p class="comment-form-comment">
```

```
            <label for="comment">Comment</label>
            <textarea id="comment" name="comment" cols="45" rows="8" aria-
required="true"></textarea>
          </p>
          <p class="form-allowed-tags">
            You may use these <abbr title="HyperText Markup
Language">HTML</abbr> tags and attributes:
            <code>&lt;a href=&quot;&quot; title=&quot;&quot;&gt; &lt;abbr
title=&quot;&quot;&gt; &lt;acronym title=&quot;&quot;&gt; &lt;b&gt;
&lt;blockquote cite=&quot;&quot;&gt; &lt;cite&gt; &lt;code&gt; &lt;del
datetime=&quot;&quot;&gt; &lt;em&gt; &lt;i&gt; &lt;q cite=&quot;&quot;&gt;
&lt;strike&gt; &lt;strong&gt;</code>
          </p>
          <p class="form-submit">
            <input name="submit" type="submit" id="submit" value="Post
Comment">
          </p>
        </form>
      </div>
      <!-- End Comment Form -->
    </section>
  </article>
</main>
```

There's a lot going on in the previous listing, but it all comes down to putting in the elements we need, or are likely to find, in order to apply styling to them. I'll admit that I didn't make up this entire markup on my own; the comments section was copied (and then modified) from the output of the default WordPress themes. That's what they're there for: to use as examples of typical WordPress markup.

4.2 The Stylesheet

So now we have our prototype's markup in place, but it's looking awfully bare. It's time to fix that by building our stylesheet.

The first step in that is to create our CSS file, which we'll call prototype.css. And in order for our prototype to recognize it we'll need to add a link to it in the <head> section of our prototype pages. So in each of our prototype files, add the following line after our meta tags:

```
<link rel="stylesheet" href="prototype.css" type="text/css">
```

4.2.1 Reset

Now we're ready to start styling our prototype. We'll start by applying a reset CSS to bring all of our elements to one baseline styling and strip away all of the normal defaults that web browsers would otherwise apply. Eric Meyer's reset CSS (at http://meyerweb.com/eric/tools/css/reset/) is one of the most popular and is the one we'll be using in this project, albeit with a couple of small edits (listing 4.10).

Listing 4.10 The Eric Meyer Reset, version 2.0

```
/* http://meyerweb.com/eric/tools/css/reset/ #A
   v2.0 | 20110126
   License: none (public domain)
*/

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, main,
menu, nav, output, ruby, section, summary,
time, mark, audio, video { #B
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, main, menu, nav, section {
  display: block; #C
}
body {
  line-height: 1.2;
}
ol, ul {
  list-style: none;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
  content: '';
  content: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}
```

#A You can find Eric Meyer's reset on his website at http://meyerweb.com/eric/tools/css/reset/, but for the love of all that is good, don't link directly to it. It's meant to be copied and pasted into your own CSS.

#B Take almost every HTML tag imaginable and strip out all margins, padding, font weights, font sizes, etc. so that pretty much every element is exactly alike.
#C Then take our newish HTML5 elements and explicitly set them to appear as block elements because some browsers may not be aware of them yet.

My version of the reset here has a few differences from Meyer's original. First, I've gotten rid of references to the `<hgroup>` element and added the `<main>` element to reflect the current HTML5 recommendation. Also, I added a little bit of line spacing (1.2, up from the default 1) so that we get a little nicer vertical rhythm by default on our elements.

Meyer Reset versus Normalize

Mac or Windows? "*jif*" or "*gif*"? Reset or normalize? There are many holy wars in the web development world.

Reset stylesheets, such as the popular Eric Meyer reset, take all of the elements and strips out any default styling that the user agent (a.k.a., web browser) would normally apply to them. This will leave all of your headings, paragraphs, list items, etc. without any margins or paddings, not bolded, and unadorned. In essence, a clean state.

Normalize (http://necolas.github.io/normalize.css/), on the other hand, uses a different approach. Instead of stripping out all user agent styling, it seeks to find common ground and make up the differences between the various user agents out in the world. So if one browser added padding underneath a paragraph, but another used margins, normalize would take a stand and apply a sensible default for all browsers.

Whenever possible, I usually prefer to use the Meyer reset. Often, designers give me designs so far removed from the default styling a browser would apply that it doesn't make sense to me to keep working against the normalized defaults. The needs of your projects, or your personal preferences, may dictate the use of one or the other.

For a more thorough comparison of reset and normalize, see "CSS: reset or normalize?" by Oli Studholme on *The Pastry Box Project* (http://the-pastry-box-project.net/oli-studholme/2013-june-3/).

With all the native browser styling stripped from our home page, it has become even blander in its appearance, if that's even possible. (Figure 4.8)

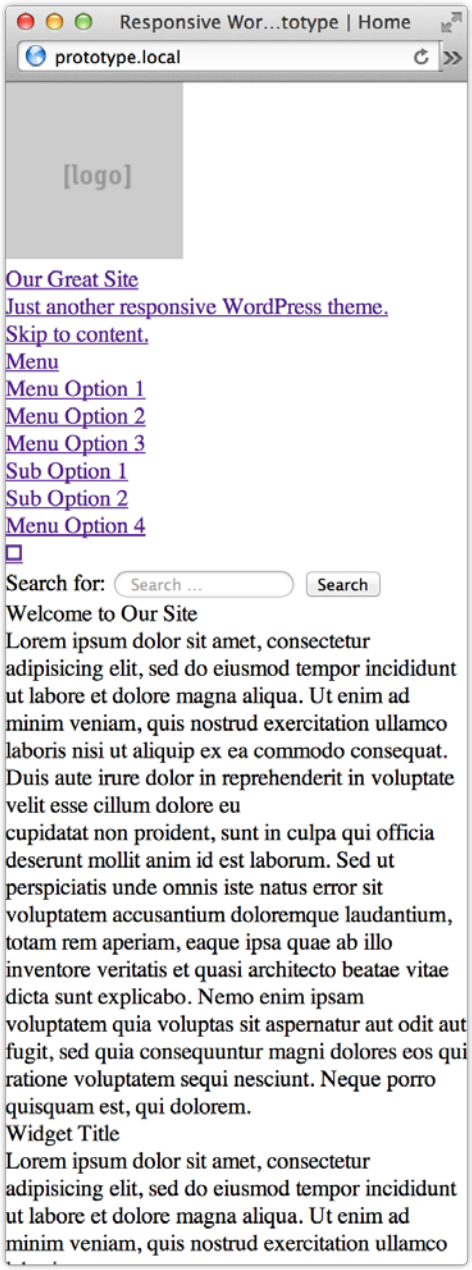


Figure 4.8 Our stylistic clean slate.

4.2.2 Site-Wide Properties

Once we've completely removed the default user agent styling from our prototype, we're going to want to establish some other defaults that we'll use throughout the site. These include things like font treatments, and establishing the `border-box` property we looked at in chapter two. Let's set them up in listing 4.11.

Listing 4.11 Setting Some Global CSS Properties

```
/**
 * Include "Open Sans" font from Google Webfonts #A
 */
@import
url(http://fonts.googleapis.com/css?family=Open+Sans:400italic,400,700);

/**
 * Include "Genericons" icon font from Automattic #B
 */
@import url(fonts/genericons/genericons.css);

[ … The Eric Meyer Reset … ]

/**
 * Global Properties
 */

/* Box Sizing */ #C
*, *:before, *:after {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

/* Fonts */ #D
html {
  font: 81.25% "Open Sans", sans-serif; /* 13px / 16px */
}

/* Body */ #E
body {
  background: #877865 url(/images/background.jpg);
}

/* Accessibility */ #F
.screen-reader-text {
  clip: rect(1px, 1px, 1px, 1px);
  position: absolute;
}

/* Micro Clearfix (from Nicolas Gallagher) */ #G
.clearfix:before,
.clearfix:after {
  content: '';
  display: table;
}
```

```
.clearfix:after {
  clear: both;
}

.clearfix {
  zoom: 1; /* For IE 6/7 (trigger hasLayout) */
}
```

#A As with all @import directives in CSS, getting the "Open Sans" font from Google Fonts must be done before declaring any style properties.
#B We're also linking to the icon font "Genericons," which we've downloaded from http://genericons.com and set up in our prototype's directory.
#C box-sizing: border-box; establishes that the CSS should include the padding and border in measuring the overall size of a box. See Paul Irish, "* { Box-sizing: Border-box } FTW" (http://www.paulirish.com/2012/box-sizing-border-box-ftw/)
#D Right off the bat we're starting with our relative units. In this case, we're setting the base font size for our documents as a percentage of the browser default, which is usually 16px. From here on out, we'll use either "em" or "rem" units.
#E We're going to set our background color to a deep, neutral color with a textured background image.
#F For accessibility, we're going to make any element with the class "screen-reader-text" hidden from the view of regular browsers, but visible to screen readers. Also, we'll apply some styling to them so that keyboard-only users can see them when the elements have focus.
#G This is standard "clearfix" code, as developed by Nicolas Gallagher, so that any element with the class "clearfix" that contains floating children will expand to the height of the children.

4.2.3 Site Header

Now we can work on individual parts of the page, starting with the header in listing 4.12.

Listing 4.12 Styling the Header

```
/**
 * Site Header
 */
header[role="banner"] {
  background: url(http://lorempixel.com/320/255/nature/2) no-repeat center
center;
  background-size: cover;
  line-height: 1.2;
  padding: 2em 18.75 %; /* 26px / 13px, 60px / 320px */ #A
  text-align: center;
}

header[role="banner"] a {
  color: #FFFFFF;
  text-decoration: none;
}

header[role="banner"] .logo {
  display: block;
  margin: 0 auto 2.30769230769231em; /* 30px / 13px */
}
```

```
header[role="banner"] h1,
header[role="banner"] .subhead {
  text-shadow: 0 1px 3px rgba(0, 0, 0, 0.75);
}

header[role="banner"] h1 {
  font-size: 2em; /* 26px / 13px */
  margin-bottom: 0.34615384615385em; /* 9px / 26px */
}

header[role="banner"] .subhead {
  font-size: 1.23076923076923em; /* 16px / 13px */
}
```

#A As a general rule, we'll express all relative vertical measurements relative to font size (that is, using "ems" or "rems") and horizontal measurements as percentages.

For the first time since we started our prototype, we have something that we might actually want to look at (figure 4.9):

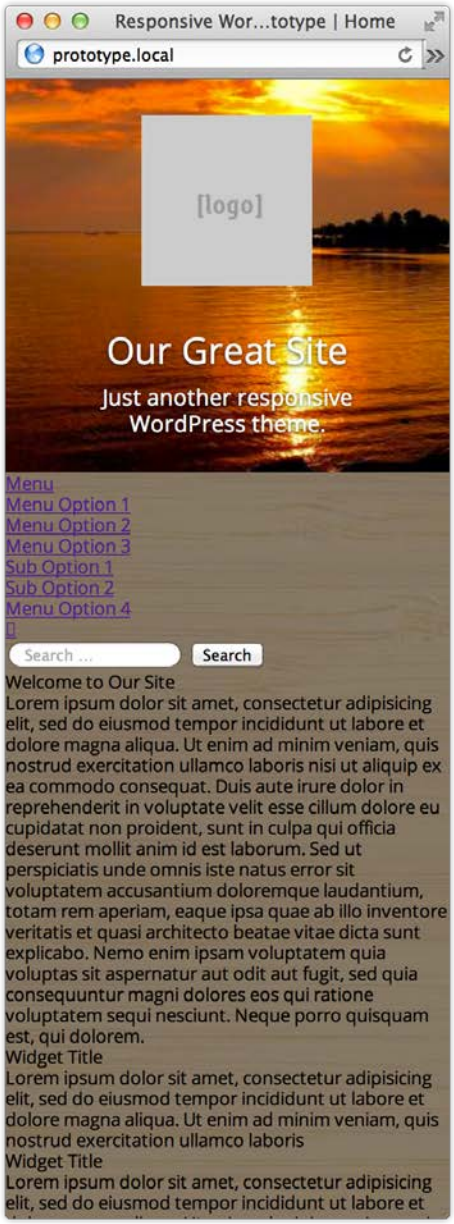


Figure 4.9 We have a styled header. Progress!

4.2.4 Primary Navigation and Search

Let's move onto our navigation. All of our options and sub-options are visible, as is the full search form, but that's not what we want according to the comp. So what we'll do in listing 4.13 is style the menu toggle "button" (for lack of a better term; we won't actually be using a button) and the menu list items, and then hide all of the list items with a "display: none;" property. Later, we'll use JavaScript to control the visibility of the navigation via the toggle button.

Listing 4.13 Styling Our Primary Navigation

```
/**
 * Primary Navigation
 */
.screen-reader-text.skip-link:focus { #A
  background-color: #771717;
  clip: auto;
  color: #FFFFFF;
  display: block;
  height: 2.46153846153846em; /* 32px / 13px */
  line-height: 2.46153846153846em; /* Equal to height for vertical
centering */
  padding: 0 3.75%; /* 12px / 320px */
  position: static;
  text-decoration: none;
}

.navbar {
  background-color: #771717;
  border-bottom: 1px solid rgba(255, 255, 255, 0.1);
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  position: relative;

  background-image: -moz-linear-gradient(top, #951c1c 0%, #771717 100%); #B
  background-image: -webkit-gradient(linear, left top, left bottom, color-
stop(0%,#951c1c), color-stop(100%,#771717));
  background-image: -webkit-linear-gradient(top, #951c1c 0%, #771717 100%);
  background-image: -o-linear-gradient(top, #951c1c 0%, #771717 100%);
  background-image: -ms-linear-gradient(top, #951c1c 0%, #771717 100%);
  background-image: linear-gradient(to bottom,  #951c1c 0%, #771717 100%);
  filter: progid:DXImageTransform.Microsoft.gradient(
startColorstr='#951c1c', endColorstr='#771717',GradientType=0 );
 }

nav[role="navigation"] a {
  color: #FFFFFF;
  display: block;
  height: 2.46153846153846em; /* 32px / 13px */
  line-height: 2.46153846153846em; /* Equal to height for vertical
centering */
  padding: 0 3.75%; /* 12px / 320px */
  text-decoration: none;
}
```

```
nav[role="navigation"] .toggle-menu {
  font-size: 1.38461538461538em; /* 18px / 13px */
  height: 1.77777777777778em; /* 32px / 18px */
  line-height: 1.77777777777778em; /* 32px / 18px */
}

nav[role="navigation"] .toggle-menu:after {
  content: "\f431";
  display: inline-block;
  font-family: Genericons; #C
  font-size: 1.23076923076923em;
  margin-left: 4px;
  vertical-align: bottom;
}

nav[role="navigation"] ul {
  background-color: #771717;
}

nav[role="navigation"] li {
  border-top: 1px solid rgba(255, 255, 255, 0.1);
}

nav[role="navigation"] ul a:hover,
nav[role="navigation"] ul a:active,
nav[role="navigation"] ul a:focus,
nav[role="navigation"] li.current_page_item a {
  background-color: rgba(255, 255, 255, 0.1);
}

nav[role="navigation"] ul ul a { #D
  padding: 0 7.5%;
}

nav[role="navigation"] ul ul ul a {
  padding: 0 11.25%;
}
```

#A For keyboard-only users, the "skip to content" link will appear in a way that's consistent with the style of the navigation when the user tabs to it. Always think accessibility!
#B CSS gradients are a blessing and a curse. We no longer have to cut our gradient background from Photoshop, but we have mounds of vendor prefixes to deal with. And I didn't even put the IE9-compatible version here because it's a heinously long base-64 string! If only there were a tool that could do all this stuff for us…
#C Yet another way to apply icon fonts: as the content of pseudo-elements. In this case, we're adding a little expand icon after the word "Menu" in our navigation visibility toggle.
#D In the case of submenus and sub-submenus, we're increasing the indentation to show their place in the hierarchy. But let's not go nuts; three levels are deep enough.

I've omitted the important "display: none" property so that we can see the results of the expanded navigation in figure 4.10:

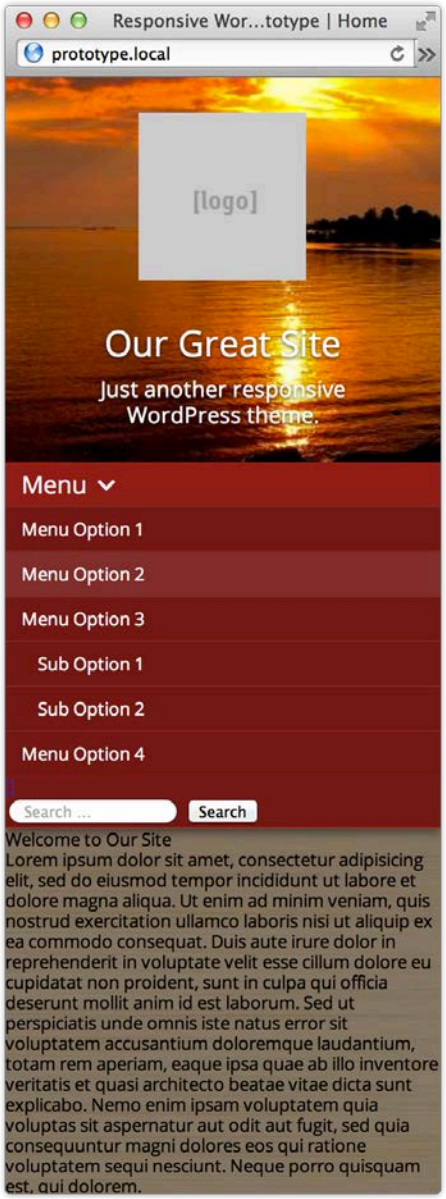


Figure 4.10 A look at our expanded navigation.

Our search bar looks completely out of place, but in the finished theme we'll never have the search form and menu both expanded at the same time. Let's hide the menu by adding:

```
display: none;
```

to the `nav[role="navigation"] ul` selector, and style the search form with the CSS in listing 4.14.

Listing 4.14 Styling the Search Form

```
.toggle-search {
  color: #FFFFFF;
  display: block;
  font-family: Genericons;
  font-size: 1.38461538461538em; /* 18px / 13px */
  height: 1.77777777777778em; /* 32px / 18px */
  line-height: 1.77777777777778em; /* 32px / 18px */
  overflow: hidden;
  position: absolute;
  right: 3.75%;
  text-decoration: none;
  top: 0;
  white-space: nowrap;
}

form[role="search"] {
  background-color: #771717;
  padding: 0 3.75% 0.25em 0;
  position: absolute;
  text-align: right;
  top: 100%;
  width: 100%;
}

form[role="search"] label {
  display: inline-block;
}

form[role="search"] input {
  border: none;
  box-shadow: 2px 2px 10px -2px rgba(0, 0, 0, 0.4) inset;
  display: inline-block;
  float: right;
}

form[role="search"] input[type="search"] { #A
  border-radius: 100em;
  margin-right: 1em;
  padding: 4px 8px;
}

form[role="search"] input[type="submit"] {
  background-color: #dfdfdf;
  border: 1px solid #d0d0d0;
  border-radius: 100em;
  box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.5),
    inset 0 -2px 3px rgba(0, 0, 0, 0.1);
  cursor: pointer;
```

```
  padding: 2px 8px;

  -webkit-transition: all 0.2s ease-out;
  -moz-transition: all 0.2s ease-out;
  -ms-transition: all 0.2s ease-out;
  -o-transition: all 0.2s ease-out;
  transition: all 0.2s ease-out;
}

form[role="search"] input[type="submit"]:hover,
form[role="search"] input[type="submit"]:active,
form[role="search"] input[type="submit"]:focus {
  background-color: #d0d0d0;
}
```

#A WebKit browsers (Chrome and especially Safari) go a little heavy on the styling for the "search" input type, to the point where it's usually easier to set the input type as "text" and style it from there. I've opted to work with what we got, and come up with a style for the search field that works with both WebKit and Mozilla. See Chris Coyier, "WebKit HTML5 Search Inputs" (http://css-tricks.com/webkit-html5-search-inputs/) for more on this.

When expanded, our search form will look like figure 4.11:

Figure 4.11 Look! Our mysterious glyph has grown up to be a magnifying glass which, when clicked or tapped, will reveal our slide-out search form (once the JavaScript to do so has been applied).

4.2.5 Main Content and Widget Areas

We're almost done. All that's left is the main content and widget areas, and that's not really as much as it sounds since right now we're still dealing with a lot of dummy content. Let's throw in a few styles to give our main content area a little definition in listing 4.15.

Listing 4.15 Our Main Content CSS

```
/**
 * Main Content
 */
main {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1px;
  padding: 1.38461538461538em 3.75%;
}

main h1 {
  font-size: 1.38461538461538em; /* 18px / 13px */
  font-weight: bold;
  margin-bottom: 1em;
}

main p {
  margin: 0.5em 0;
}
```

See? I told you there wouldn't be much to it. Let's take a look at the results in figure 4.12:

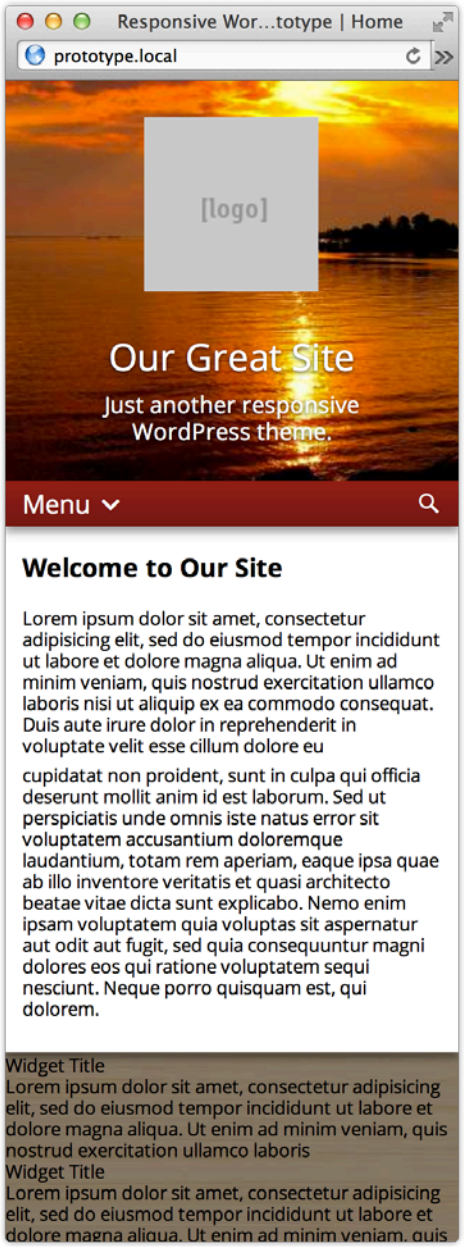


Figure 4.12 Simple and clean (for now), our home page is starting to really take shape.

Last, but not least, let's tackle our two types of widget areas, the sidebars and the footer widget areas, together since they will be sharing some of the same qualities (listing 4.16).

Listing 4.16 Styling the Widget Areas

```
/**
 * Widgets
 */
.widget {
  padding: 1em 3.75%;
}

/**
 * Primary and Secondary Sidebars
 */
.sidebar .widget {
  background-color: #e9e2dd;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1px;
}

.sidebar .widget h1 {
  color: #514530;
  font-size: 1.23076923076923em; /* 16px / 13px */
  font-weight: bold;
}

/**
 * Footer Widget Areas
 */
footer[role="contentinfo"] {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1.53846153846154em; /* 20px / 13px */
}
```

With our last bit of CSS in place (for now), we have a fully-realized prototype for what the home page for our responsive WordPress theme will look like, at least in its narrow view (figure 4.13).

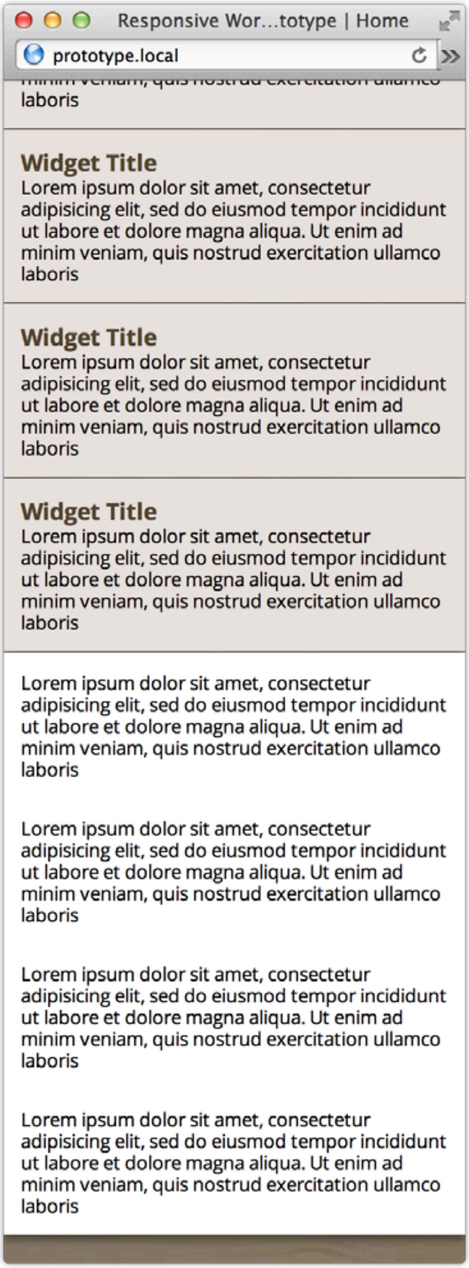


Figure 4.13 Every element on the page now has styling of some sort.

Once we get into building the live WordPress theme, and incorporating more complete test data, we'll certainly have to revisit the widget areas to add some more styling. But for now, we're cool with what we got here.

Our work is by no means done, but the pieces are certainly starting to come together. We have something in a web browser that looks like the designs we created in Photoshop. Furthermore, thanks to CSS3 and icon fonts, there was mercifully little we had to cut and use from Photoshop to make this design work.

4.3 Summary

In this chapter, we've taken the home page that we designed in Photoshop and turned it into a living, breathing web page with HTML and straight CSS. This may not have been the sexiest chapter to work through, but building a static prototype gives you a solid foundation that will make the actual WordPress theming process go a lot more smoothly.

If you've been playing along at home, you may have noticed some glaring omissions from our prototype: there was not a single media query to be had, and we haven't even attempted styling the main content areas of the archive and single article templates. Where's the responsiveness? Where's the styling of meta content and featured images?

Worry not, fair reader. Just as in chapter three we designed different parts of our wireframe and visual design using different tools, we're going to do the same with our prototype. In chapter five we're going to meet one of the most miraculous, mind-blowing tools to come to CSS since the `border-radius` and `box-shadow` properties. Ladies and gentlemen, I give you the wonderful world of CSS preprocessors.

5

Using CSS Preprocessors

This chapter covers

- What CSS preprocessors are
- The most widely used preprocessors
- Tools for using CSS preprocessors in your workflow
- Using Sass in responsive web design

In chapter four, we spent a good amount of time building a working HTML and CSS prototype of our theme's home page in its narrow view. But we only got that far; there's still the matter of the other pages, not to mention the responsiveness as the viewport expands.

On my first major responsive web project, coding the front end for the Robert Wood Johnson Foundation's site redesign (http://rwjf.org), figure 5.1, we didn't use CSS preprocessors. At one point, I lamented on Twitter how much of a pain it was to change a color that was used throughout the site, and someone replied that I should give preprocessors a try. But I was reluctant, partly because they weren't part of the workflow where I was working at the time, but also because I thought that preprocessors were a fringe tool used only by the über-geeks of the CSS world. In short, I was too afraid to try them.

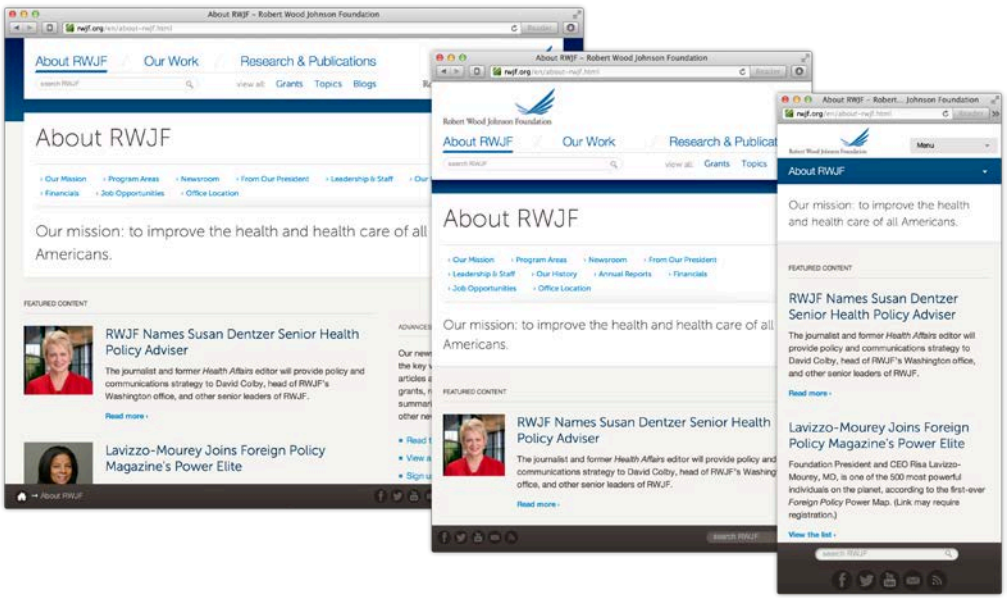


Figure 5.1 The Robert Wood Johnson Foundation website was built without the benefit of CSS preprocessors, but I wouldn't recommend that approach today.

Further down the road, I started dabbling with LESS, and finally started to understand the appeal. The nesting and variable features alone not only saved a tremendous amount of time, but also made my code more organized and easier to maintain. There were still problems, however, as I started working on more and more responsive projects, I found that LESS had its limitations (some of which have since been addressed), and that Sass provided a more powerful solution.

Despite my initial reluctance (and, admittedly, my "Get off my lawn!" attitude towards the growing preprocessor movement), I've come to the conclusion that they are invaluable tools that can save hours of development time. I've also decided that I won't ever work on a responsive project again without one.

Whatever CSS preprocessor you ultimately decide to use, there is one important thing to remember: there is *nothing* you can do with CSS preprocessors that you couldn't do in plain CSS. The difference is in the shortcuts, time savers, and organization that preprocessors provide. In this chapter, we'll continue the work we started in chapter four, but this time we'll use CSS preprocessors to help us along.

5.1 What Are CSS Preprocessors, Exactly?

In a nutshell, CSS preprocessors are an alternative way of writing CSS that gets passed through another language (such as JavaScript or Ruby) and then converted into the standard

CSS that browsers expect. In the process, they provide a number of useful tools meant to facilitate the mundane and repetitive tasks so often found in writing plain CSS.

> **"COMPILING" CSS** It's common to say that preprocessors "compile" their code into pure CSS.This isn't compilation in the same way a computer compiles code into ones and zeroes that the CPU understands. In this context, it's more akin to conversion. The results can be filled with debugging code, or compressed into a one-line minified state, all depending on the user's settings.

In many ways, preprocessors make CSS writing more like a true programming language (which pure CSS is *not*, in any way, shape or form). If you've worked with languages such as PHP and JavaScript, then you might appreciate being able to use variables and functions in your CSS. If not, it's really not hard to pick up on these features. The extent to which you use any preprocessor's features is entirely up to you, as most of them understand plain CSS just as readily as their own unique syntax.

There are lots of preprocessors out there, but two of them – LESS and Sass– seem to get most of the press. Let's take a look at them and some of their key characteristics.

5.1.1 LESS

LESS (http://lesscss.org) was started by Alexis Sellier in 2009. It uses JavaScript as its compilation engine, meaning that you can compile your CSS on the client side by including the LESS JavaScript file in your HTML, or you can compile it on the server side by passing your LESS stylesheet through a Node.js module. Because LESS can be compiled on the client side, it's popular with users new to preprocessors who appreciate the low overhead to getting started. To change our stylesheet to LESS, all we need to do is rename "prototype.css" to "prototype.less" and change the <head> of our document to listing 5.1.

Listing 5.1 Compiling LESS on the Client Side

```
<head>
<title>Responsive WordPress Theme Prototype | Home</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet/less" type="text/css" href="prototype.less"> #A
<script src="js/less-1.3.3.min.js"></script> #B
</head>
```

#A Notice the change to the "rel" attribute in our <link> tag. Also, we've renamed our original prototype.css to prototype.less. Other than that, everything is the same.
#B In order for the LESS stylesheet to compile to regular CSS, we need to link the stylesheet *before* we call the JavaScript parser. Also, the JavaScript parser must be called in the header, not at the bottom of the document in order for our CSS to be processed as much before the DOM loads as possible.

When we load the revised page in our web browser, we get a result that looks a lot like what we had by the end of chapter four (figure 5.2):

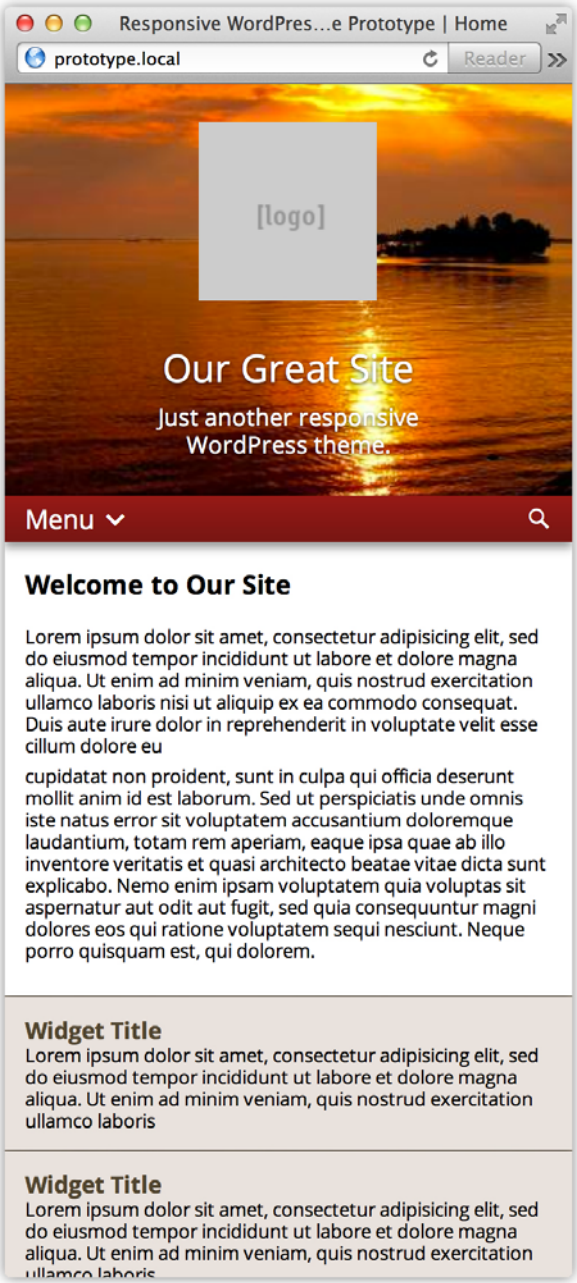


Figure 5.2 Notice anything different? Neither do I. All that's changed is the way our CSS is handled, nothing else.

LESS still enjoys an enthusiastic following, and is used by such popular frameworks as Twitter Bootstrap. However, for a long while its development stagnated, and it lacked capabilities that its chief rival (for lack of a better term) had. These deficiencies were particularly noticeable in terms of responsive design, so I stopped using LESS myself and turned to Sass.

5.1.2 Sass and Compass

Sass (http://sass-lang.com) was originally developed by Hampton Catlin and Nathan Weizenbaum in 2007 and is now maintained by Weizenbaum and Chris Eppstein. Sass is my personal choice for CSS preprocessors these days,and one of the main reasons I switched was Sass' "media query bubbling" which I'll demonstrate in a moment. That was a feature that LESS didn't have when I switched, although it has since been added.

Sass has a companion project called Compass (http://compass-style.org)that shares a maintainer in Eppstein. Compass is an extra framework that adds a selection of ready-to-use mixins and functions beyond those that come with Sass alone. It also has several advanced utilities, such as automatic CSS sprite generation, that can be very powerful and timesaving.

One reason why WordPress people in particular may find Sass and Compass somewhat daunting is the fact that they are Ruby projects, and thus go particularly well in a Ruby on Rails environment. Obviously, that's not the world that WordPress people usually live in,but don't let that scare you off. It's pretty simple to install and run Sass and Compass on OS X machines, and even simpler still to use a GUI tool to do your processing for you.

There are actually two ways of writing Sass. The first is the original Sass way, using indentation and an absence of punctuation like curly brackets and semicolons that one is used to seeing in CSS code. This format is based on Haml (http://haml.info/), an HTML abstraction tool also created by Catlin. The goal of this format is conciseness: saving as many keystrokes as possible in order to make your work go faster, as can be seen in listing 5.2.

Listing 5.2 Our Header CSS Using Sass' Indented Syntax

```
header[role="banner"] #A
  background: url(http://lorempixel.com/320/255/nature/2) no-repeat center
center #B
  background-size: cover
  line-height: 1.2
  padding: 2em 18.75% /* 26px / 13px; 60px / 320px */
  text-align: center

header[role="banner"] a
  color: #FFFFFF
  text-decoration: none

header[role="banner"] .logo
  display: block
  margin: 0 auto 2.30769230769231em /* 30px / 13px */
```

```
header[role="banner"] h1,
header[role="banner"] .subhead
  text-shadow: 0 1px 3px rgba(0, 0, 0, 0.75)

header[role="banner"] h1
  font-size: 2em /* 26px / 13px */
  margin-bottom: 0.34615384615385em /* 9px / 26px */

header[role="banner"] .subhead
  font-size: 1.23076923076923em /* 16px / 13px */
 #A Look Ma! No curly brackets!
 #B Hey, no semicolons either.  What gives?
```

The second, newer Sass syntax is known as SCSS (for Sassy CSS) and looks more like the traditional CSS you've grown to know and love. Indentation isn't important at all, and our code has the curly brackets and semicolons we're used to. The name of the SCSS game is compatibility: You can take any valid CSS file and turn it into an SCSS file simply by changing the extension to ".scss". For that reason, SCSS is often referred to as a "superset" of regular CSS.My preference is for the SCSS format,therefore all of the listings from here on out will be in that syntax.

But What About Stylus?

If I don't at least mention Stylus, I feel I am doing a disservice to you, the reader, when talking about CSS preprocessors. You've probably heard of it, or at least seen it mentioned in a Google search on CSS preprocessors.

The fact is, Stylus is completely dependent upon using Node.js for your project (unlike LESS which also runs fine on Node.js, but can be used without it as well). That simply isn't where WordPress development happens, so we're going to continue on.

5.2 Workflows

Before we get into the nuts and bolts of using Sass, we have to establish a means by which our Sass code will be converted into normal CSS that the browser can understand. Since we're still in the prototyping phase, we don't yet have to worry about how to integrate our Sass file into our working WordPress theme, although we'll cover that when the time comes. For now we have two options: command line tools and GUI tools.

5.2.1 Command Line Tools

Smack-dab at the top of every page on Sass' website are the instructions for installing Sass, creating a file, and processing it in real time. If you're not familiar with Ruby or gems, then even those simple lines can seem a bit daunting, which is why there are GUI options. But if you'd prefer to work directly on the command line (or are using a platform that doesn't have a GUI option), then you'll find it's not hard to get started.

The first step is to make sure that you have Ruby installed on your development system. If you're using Mac OS X, then you're already good. Most Linux flavors either come with Ruby as a default package, or it can be easily added with a package manager; check documentation for your distribution for further details. Windows users don't have Ruby preinstalled on their systems so if that's your situation, you might want to skip down to the GUI tools. (If you're dead set on installing Sass on Windows via Ruby, *Impressive Webs* wrote instructions on how to do it that you can find at http://www.impressivewebs.com/sass-on-windows/.)

Since you'll need root privileges to install a Ruby gem, we're going to modify Sass' instructions, just a little. Bring up the command line for your system and run the following:

```
sudo gem install sass
```

You'll need to enter your password in order for the command to execute. Once done, you should see output like so:

```
taupecat@tracy-rmbp-1 10:52:38 $ sudo gem install sass
Password:
Successfully installed sass-3.2.9
1 gem installed
Installing ri documentation for sass-3.2.9...
Installing RDoc documentation for sass-3.2.9...
```

Now we're ready to rock and roll! Staying in the command line, change your directory to where your prototype files are. Doing a little thinking ahead, we're going to need more than one Sass file (trust me on this, we'll get there soon), so let's create a new directory called **scss** and move our **prototype.css** file to **scss/prototype.scss**:

```
/usr/local/web/prototype/htdocs
taupecat@tracy-rmbp-1 23:04:23 $ mkdir scss
taupecat@tracy-rmbp-1 23:04:49 $ mv prototype.css scss/prototype.scss
```

Note that not only are we moving it into our new directory, but we're also changing its extension to denote that it is a Sass file using SCSS syntax.

Now all that's left is to tell Sass to watch the file, and recompile on every change. In our revised file structure, the command for that will be:

```
/usr/local/web/prototype/htdocs
taupecat@tracy-rmbp-1 23:07:23 $ sass --watch
scss/prototype.scss:prototype.css
>>> Sass is watching for changes. Press Ctrl-C to stop.
  overwrite prototype.css
```

Now every single time we make a change to Sass file (or files, as we shall see), Sass will detect that change on save and create a new CSS file to reflect those changes. If there are

errors at any point in our Sass stylesheet, an error message will display in the console describing the problem and where to find it (listing 5.x):

```
/usr/local/web/prototype/htdocs
taupecat@tracy-rmbp-1 13:08:16 $  sass --watch
scss/prototype.scss:prototype.css
>>> Sass is watching for changes. Press Ctrl-C to stop.
  overwrite prototype.css #A
>>> Change detected to: /usr/local/web/prototype/htdocs/scss/prototype.scss
#B
      error scss/prototype.scss (Line 120: Invalid CSS after "  line-
height": expected ";", was ": 1.2;") #C
```

#A The first thing Sass does when you launch it is compile the stylesheet. If the compiled stylesheet already exists, it'll overwrite it.
#B Every time you save a change to your Sass files, Sass will report it in the console.
#C In this change, we introduced an error. Apparently I removed a semicolon from line 120 of scss/prototype.scss.

5.2.2 GUI Tools

If the command line isn't your thing, there are some GUI tools that will handle the compilation process for you. The nice thing about these is you don't need to install Sass via the whole Ruby gem thing.

CodeKit

CodeKit (http://incident57.com/codekit/) (figure 5.3) has become the *de facto* app for compiling Sass and Compass on the Mac, as well as handling a bunch of other things like JavaScript linting, Coffeescript processing, and more.

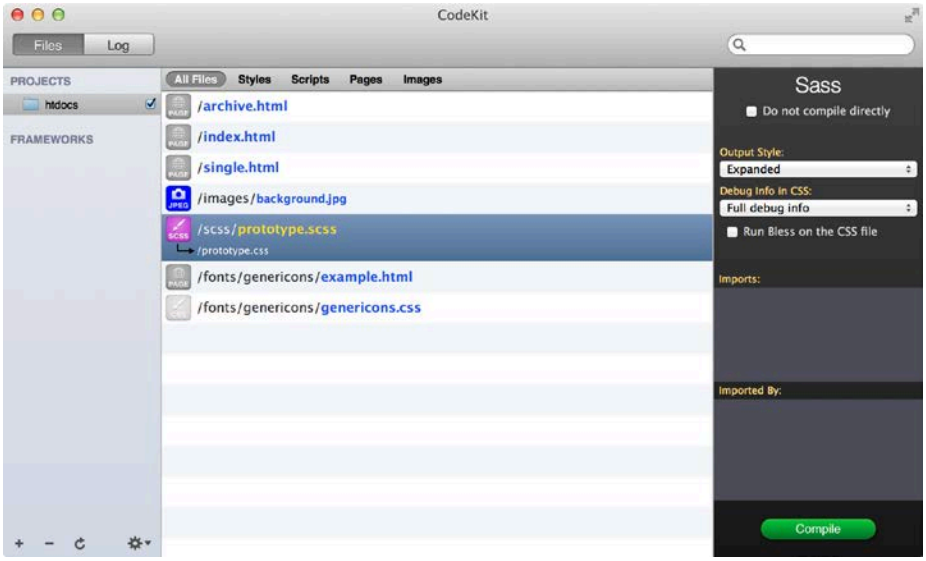


Figure 5.3 CodeKit has become a popular choice for Mac-based web developers looking to work in Sass.

CodeKit is extremely easy to use. Just download and launch it, and drop the directories you want to watch on its window. You may have to define where you want your Sass files to be compiled to (its *output path*, in CodeKit jargon), but once that's done, it'll do all the watching for you. You don't even need to download and install Sass from Ruby (or Compass, for that matter), as it has built-in parsers for handling that. Would you rather use LESS? CodeKit has you covered there, too.

CodeKit also touts the ability to further losslessly compress images, although my results trying this have been mixed. It also allows you to break up your static HTML files into templates (so that you're not copying the header of your document three different times in your prototype files, for example).

SCOUT

Scout (http://mhs.github.io/scout-app/) is a good alternative if you either don't want to pay the nominal shareware price for CodeKit, don't need all its features, or are running a development environment other than a Mac.

Scout is a free processor that only works with Sass and Compass (sorry LESS users). It has the added benefit of being cross platform, with Windows and Mac versions of the software. It works just like a wrapper around the command line Sass functionality we covered earlier, and lacks a lot of CodeKit's GUI polish (figure 5.4). Still, if you're hard core about avoiding the command line, Scout is a good alternative.

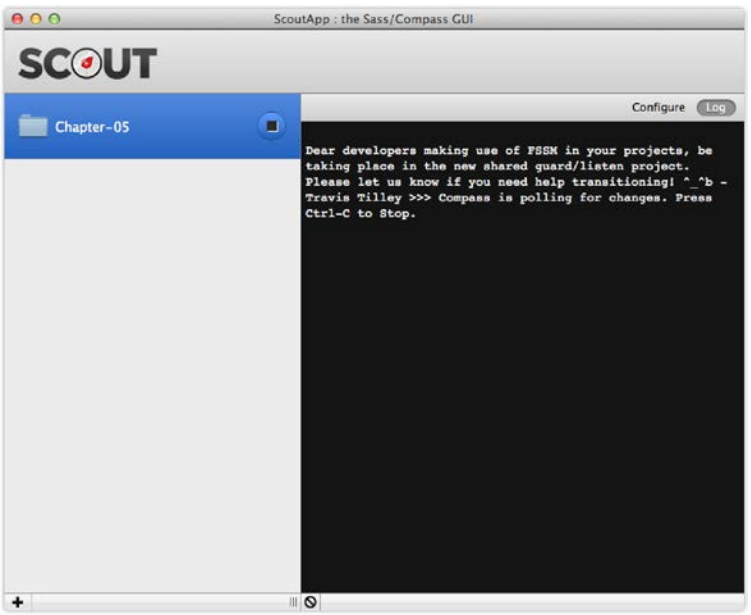


Figure 5.4 Scout is a free and cross-platform alternative to compiling Sass and Compass, but lacks the polish of CodeKit.

5.3 Using Sass

As I indicated before, Sass is my CSS preprocessor of choice, and so that is what we'll explore in depth and ultimately use in our project. Many of the features listed below are also available in LESS, but not always, and LESS may have features that Sass does not. It also might have slight differences in syntaxto achieve the same thing.

5.3.1 Imports and Partials

There are probably harder things to debug than a ridiculously long CSS file, but not many. Many developers attempt to get past this by chopping up their CSS into different files and importing them individually with the CSS @import directive. But traditional CSS imports have twobig drawbacks:

- A regular CSS import will increase the number of HTTP requests your page makes, causing a significant drain on your page's performance.
- You can only use the @import directive in traditional CSS before any other property is declared, potentially causing inheritance nightmares.

Fortunately, Sass' version of imports addresses both these problems. Since in Sass the @import directive is actually merging the imported code into the document requesting it, there is no extra HTTP request hit to worry about. Also, Sass doesn't particularly care where in the file your @import directives are, and will honor them anywhere in the document. If your @import directive is to an external file (a Google font request, for example), then it will move that directive to the top of the compiled CSS file. Yes, you will get the HTTP request hit in that case, but there's only so much one can do about that.

Partials are special kinds of Sass files that are not compiled directly, but are intended specifically for importation into another document. They are designated by an underscore ("_") at the beginning of the filename. When importing them into your Sass file, you need only to specify the name of the partial, minus the underscore and extension. For example, if we had a partial called "_header.scss", our Sass import directive would look like this:

```
@import "header";
```

Let's break out the parts of our **prototype.scss**file into separate partials. When we're done, our **prototype.scss** file will look something like listing 5.3.

Listing 5.3 Our prototype.scss, Broken Into Imports

```
/**
 * Fonts
 */
@import "fonts"; #A

/**
 * Eric Meyer Reset
 */
```

```
@import "reset";

/**
 * Global Properties
 */
@import "global";

/**
 * Site Header
 */
@import "header";

/**
 * Primary Navigation and Site Search
 */
@import "navigation";

/**
 * Main Content
 */
@import "main";

/**
 * Widgets
 */
@import "widgets";
```

#A Our fonts file will have import directives to our two external font files, one on Google and one that's local. Yes, you can have imports that call other imports.

In the meantime, our directory listing for the **scss** directory now looks like so:

```
taupecat@tracy-rmbp-1 22:34:48 $ ls –l
total 72
-rw-r--r--  1 taupecat  staff   246 Jun 30 22:29 _fonts.scss
-rw-r--r--  1 taupecat  staff   608 Jun 30 22:31 _global.scss
-rw-r--r--  1 taupecat  staff   725 Jun 30 22:31 _header.scss
-rw-r--r--  1 taupecat  staff   279 Jun 30 22:32 _main.scss
-rw-r--r--  1 taupecat  staff  4431 Jun 30 22:32 _navigation.scss
-rw-r--r--  1 taupecat  staff  1105 Jun 30 22:29 _reset.scss
-rw-r--r--  1 taupecat  staff   503 Jun 30 22:32 _widgets.scss
-rw-r--r--  1 taupecat  staff   332 Jun 30 22:32 prototype.scss
```

Everything that will be written out to our final stylesheet is now in a partial. There is actually no code in **prototype.scss** that will write out CSS on its own. Think of your main SCSS file as a holding company, while its subsidiaries (in the form of partials) do all the real work.

In both the command line Sass compiler and the GUI helper applications, a change to any included partial file will trigger a recompilation of the overall main Sass file with those changes included.

5.3.2 Variables

Easily one of CSS preprocessors' greatest attractions is the use of honest-to-goodness variables in your CSS documents. No longer must you have the same cryptic hexadecimal

color code scattered throughout your file; you can define all of your colors in one place and reference them by easy to remember variables from that point on.

Variables in CSS preprocessors can be any stringand are useful for defining colors as well as font stacks, responsive breakpoints, and common contexts.Looking through our code so far, let's find every color that gets repeated and make them variables. We'll want to add the following lines to the top of our **prototype.scss** file.

```
$red: #771717;
$gray: #D0D0D0;
```

Both of those values appear exclusively in our **_navigation.scss** partial, so let's go ahead and swap out the hex codes for the variables in listing 5.4.

Listing 5.4Using Our Sass Variables

```
.screen-reader-text.skip-link:focus {
  background-color: $red;
  […] #A
}

.navbar {
  background-color: $red;
 […]
  background-image: -moz-linear-gradient(top, #951c1c 0%, $red 100%); /*
FF3.6+ */
  background-image: -webkit-gradient(linear, left top, left bottom, color-
stop(0%,#951c1c), color-stop(100%,$red)); /* Chrome,Safari4+ */
  background-image: -webkit-linear-gradient(top, #951c1c 0%, $red 100%); /*
Chrome10+,Safari5.1+ */
  background-image: -o-linear-gradient(top, #951c1c 0%, $red 100%); /*
Opera 11.10+ */
  background-image: -ms-linear-gradient(top, #951c1c 0%, $red 100%); /*
IE10+ */
  background-image: linear-gradient(to bottom,  #951c1c 0%, $red 100%); /*
W3C */
  filter: progid:DXImageTransform.Microsoft.gradient(
startColorstr='#951c1c', endColorstr='#{$red}',GradientType=0 ); /* IE6-8
*/ #B
}

nav[role="navigation"] ul {
  background-color: $red;
  display: none;
}

form[role="search"] {
  background-color: $red;
  […]
}

form[role="search"] input[type="submit"] {
  background-color: #dfdfdf;
  border: 1px solid $gray;
```

```
  […]
}

form[role="search"] input[type="submit"]:hover,
form[role="search"] input[type="submit"]:active,
form[role="search"] input[type="submit"]:focus {
  background-color: $gray;
}
```

#A Obviously, I'm truncating a lot of stuff that we covered earlier.
#B In some cases, your variable will need to be interpolated.

It's easy to guess what the processed CSS output will be, but let's take a look at it in listing 5.5.

Listing 5.5 Results of Using Sass Variables

```
.screen-reader-text.skip-link:focus {
  background-color: #771717;
  […]
}

.navbar {
  background-color: #771717;
  […]
  background-image: -moz-linear-gradient(top, #951c1c 0%, #771717 100%);
  /* FF3.6+ */ #A
  background-image: -webkit-gradient(linear, left top, left bottom, color-
stop(0%, #951c1c), color-stop(100%, #771717));
  /* Chrome,Safari4+ */
  background-image: -webkit-linear-gradient(top, #951c1c 0%, #771717 100%);
  /* Chrome10+,Safari5.1+ */
  background-image: -o-linear-gradient(top, #951c1c 0%, #771717 100%);
  /* Opera 11.10+ */
  background-image: -ms-linear-gradient(top, #951c1c 0%, #771717 100%);
  /* IE10+ */
  background-image: linear-gradient(to bottom, #951c1c 0%, #771717 100%);
  /* W3C */
  filter:
progid:DXImageTransform.Microsoft.gradient(startColorstr='#951c1c',
endColorstr='#771717',GradientType=0 );
  /* IE6-8 */
}

nav[role="navigation"] ul {
  background-color: #771717;
  display: none;
}

form[role="search"] {
  background-color: #771717;
  […]
}

form[role="search"] input[type="submit"] {
  background-color: #dfdfdf;
```

```
  border: 1px solid #D0D0D0;
  […]
}

form[role="search"] input[type="submit"]:hover,
form[role="search"] input[type="submit"]:active,
form[role="search"] input[type="submit"]:focus {
  background-color: #D0D0D0;
}
```

#A In the "expanded" output style, comments are preserved, but there are settings that compress code into a single, uncommented and minified line, and other settings that allow for Sass-specific debugging information.

Variable Interpolation

In certain cases, Sass variables need to be *interpolated* in order to render. These cases include when the variable isin the name of a selector or property or if the variable is enclosed in single or double quotes (as in our Internet Explorer filter in listing 5.4).Interpolated syntax in Sass takes the form of `#{$variable-name}`.

5.3.3 Nesting

Along with variables, nesting in Sass makes it worth the price of admission. The concept is quite simple: When defining selectors that are the descendants of previously defined selectors, you can nest their definitions in order to avoid repetition. Let's move on with our prototype, using nesting as we build the archive page main content area in listing 5.6.

Listing 5.6 Using Nesting in Our Archive Main Content

```
/**
 * Archive Page Styling
 */

.archive-title {
  font-size: 1.53846153846154em; /* 20px / 13px */
  font-weight: 600;
  margin-bottom: 2.6em; /* 52px / 20px */
  text-shadow: 2px 2px 0 rgba(0, 0, 0, 0.25);
  text-transform: uppercase;
}

.post { #A
  margin-bottom: 4em;

  /* Featured Image */
  .attachment-post-thumbnail { #B
    display: block;
    margin: 1em auto 0.5em;
  }

  .entry-title {
    margin: 0.5em 0;
```

```
  }

  .content-meta {
    font-size: 0.92307692307692em; /* 12px / 13px */
    font-weight: 700;
    margin: 0.5em 0;
  }
}
```

#A Every article in the archive page will be contained inside of an element with the class of "post."
#B "attachment-post-thumbnail," ".entry-title," and ".content-meta" are all classes of child elements inside the ".post" element.

Our compiled code will appear as listing 5.7.

Listing 5.7 Results of Nesting

```
/**
 * Archive Page Styling
 */
.archive-title {
  font-size: 1.53846153846154em;
  /* 20px / 13px */
  font-weight: 600;
  margin-bottom: 2.6em;
  /* 52px / 20px */
  text-shadow: 2px 2px 0 rgba(0, 0, 0, 0.25);
  text-transform: uppercase;
}

.post {
  margin-bottom: 4em;
  /* Featured Image */
}
.post .attachment-post-thumbnail { #A
  display: block;
  margin: 1em auto 0.5em;
}
.post .entry-title {
  margin: 0.5em 0;
}
.post .content-meta {
  font-size: 0.92307692307692em;
  /* 12px / 13px */
  font-weight: 700;
  margin: 0.5em 0;
}
```

#A Sass has put the necessary descendant selectors in a valid, recognizable CSS way.

In nesting, the ampersand (&) is used to reference the selector it's inside. It's useful when you need to modify the parent selector, such as an element that gets different CSS treatment when it has a different class or pseudo-class.

Let's take a look at our new "archive" prototype (figure 5.5):

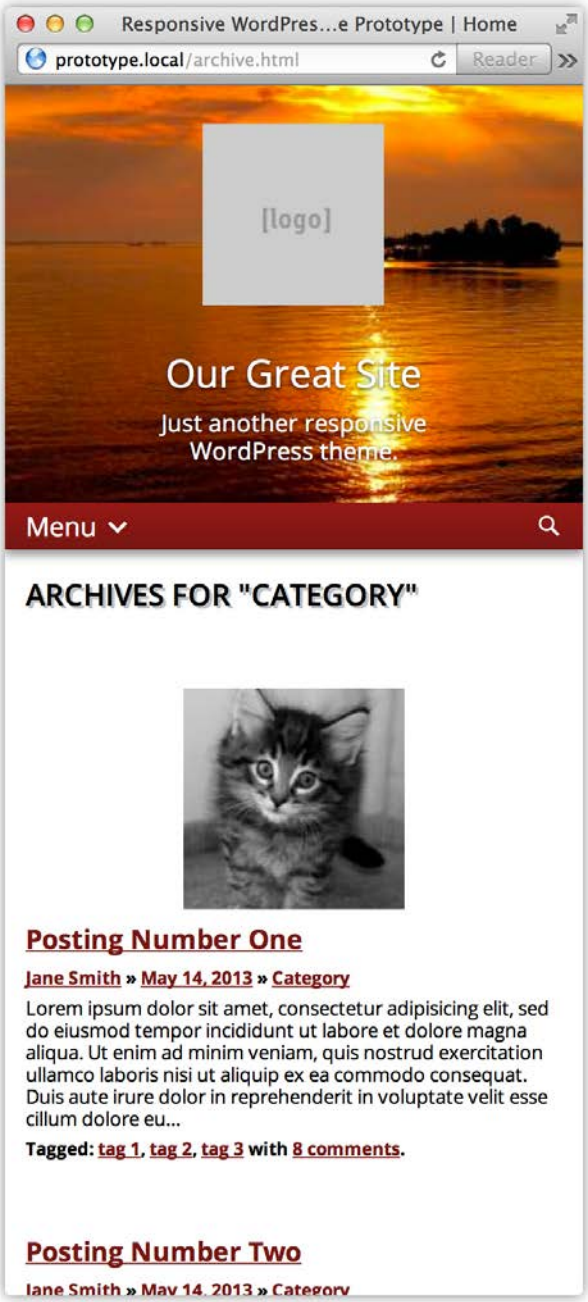


Figure 5.5 Our archive-style page begins to take shape.

The *Inception* Rule

Critics of CSS preprocessors often point to nesting as a major source of code bloat, and a major cause of this can be overzealous nesting. For this reason, there is the *Inception* rule, derived from the movie: Don't go more than four levels deep. A good explanation and demonstration of this rule can be found on the blog *The Sass Way* at http://thesassway.com/beginner/the-inception-rule.

WordPress developers should be especially careful of overnesting. WordPress doesn't contribute to divitis (http://www.zeldman.com/2010/12/05/divitis-we-fall/) as much as some other CMS platforms (ahem, Drupal), but you still don't always have as much control over markup as you'd like. When nesting your Sass based on WordPress markup, always think to yourself "Does this element really need to be expressed as a descendant as it is in the HTML, or can it be expressed next to it without compromising specificity?"

It's not a hard and fast rule. The hand of Zeus is not going to smite you if you *on the rare occasion* hit that fifth level. Just don't go crazy with it!

5.3.4 Extends and Placeholder Selectors

So we've built the main content archive listing, but we also need some buttons to navigate through the various pages of our archive listing. These usually appear at the bottom of each page with wording like "Next Posts" and "Previous Posts." Looking at the sample on our style tile from chapter three, the two links share a lot in common such as the arrows (for which we will use pseudo-elements). We want to style these semantically, without the use of "display only" classes. First let's take a look at our HTML markup in listing 5.8.

Listing 5.8 Our Older Posts/Newer Posts Markup

```
<!-- Begin Bottom Newer/Older Posts Links -->

<nav class="nav-next-previous"> #A
<a class="nav-previous" href="http://wordpress36/?paged=3">Older posts</a>
#B
<a class="nav-next" href="http://wordpress36/">Newer posts</a>
</nav><!-- #nav-below -->

<!-- End Bottom Newer/Older Posts Links -->
```

#A There's going to be a lot of floating going on inside this <nav> element. Gee, maybe we should add a "clearfix" class? Or perhaps there's another way…
#B Who needs s when we can make our <a>s block elements?

The @extend directive is a Sass feature for handling inheritance The Right Way™. Instead of copying the same properties over and over, Sass will find selectors that include the same @extend calls and group them together around the applicable properties. Any class can be extended, so let's take a look by extending our clearfix class in listing 5.9.

Listing 5.9 Extending .clearfix On Our Container Element

```
/**
 * Archive Next/Previous Navigation
 */

.nav-next-previous {
  @extend .clearfix;
}
```

So instead of including all those clearfix properties we defined in the last chapter *again* here under the .nav-next-previous (or, for that matter, adding the "clearfix" class to our markup), we're going to add .nav-next-previous to where we've already defined .clearfix. The resulting CSS is shown in listing 5.10.

Listing 5.10 Result of Extending .clearfix In the Container Element

```
/* Micro Clearfix (from Nicolas Gallagher) */
.clearfix:before, .nav-next-previous:before,
.clearfix:after,
.nav-next-previous:after { #A
  content: '';
  display: table;
}

.clearfix:after, .nav-next-previous:after {
  clear: both;
}

.clearfix, .nav-next-previous {
  zoom: 1;
  /* For IE 6/7 (trigger hasLayout) */
}
```

#A When you're extending a selector in Sass, it also preserves any pseudo-elements and descendantsthat the selector *that is being extended* has.

PLACEHOLDER SELECTORS

There are going to be times where there isn't an existing class that meets our needs exactly. In this application, the "Older Posts" and "Newer Posts" links share a lot of common characteristics (such as the icon font, the circle shape, the hover state, etc.), but what they *don't* share is a selector structure that will make inheritance feasible. Sass has an answer for this too in the form of placeholder selectors. Just like a dot denotes a class, and a hash (or pound sign) denotes an ID (which shouldn't be used as a CSS selector, but I digress), a percentage sign creates a placeholder selector. Let's create a placeholder selector for the link pseudo-elements and apply them in our Sass in listing 5.11.

Listing 5.11 Creating a Placeholder Selector

```
/**
 * Archive Next/Previous Navigation
 */

%nav-next-previous-a-psuedo-elements { #A
  background-color: $taupe; #B
  border-radius: 100em; #C
  color: #FFFFFF;
  display: block;
  font-family: genericons;
  font-size: 2em; /* 36px / 18px */
  font-style: normal;
  height: 1.11111111111111em; /* 40px / 36px */
  line-height: 1.11111111111111em; /* 40px / 36px */
  text-align: center;
  width: 1.11111111111111em; /* 40px / 36px */ #D
}

.nav-next-previous {
  @extend .clearfix;

  a {
    color: $taupe;
    display: block;
    font-size: 1.38461538461538em; /* 18px / 13px */
    font-style: italic;
    line-height: 2.22222222222222em; /* 40px / 18px */

&.nav-previous { #E
      float: right;

&:after {
        @extend %nav-next-previous-a-psuedo-elements; #F

        content: '\f429'; #G
        float: right;
        margin-left: 10px; #H
      }
    }

&.nav-next {
      float: left;

&:before {
        @extend %nav-next-previous-a-psuedo-elements;

        content: '\f430';
        float: left;
        margin-right: 10px;
      }
    }

&:hover, &:active, &:focus {
      color: $red;
```

```
&:before, &:after {
        background-color: $red;
      }
    }
  }
}
```

#A I'm lousy at naming things.
#B We're going to call this taupe color a couple of times here, so we'll define it in our variables section in prototype.scss as #85745F.
#C By setting our border radius to a ridiculously large number, our pseudo-element will be a circle, so long as the width and height are equal. Which brings us to…
#D We're making an exception to our normal rule of using percentages for widths here so that the width is exactly the same value as the height, forming a square. Thanks to our ridiculously large border-radius value (see #C), it's now a circle.
#E Here's an example of the & (ampersand) in Sass nesting. Here it is saying that this selector *modifies* the a above it, and is not a descendant of it.
#F We can extend our placeholder selector just like we did our regular class selector.
#G Once we've done our extend, we can declare the properties that the two elements *don't* have in common.
#H Gasp! A fixed pixel measurement! Not *every* single definition of space has to be in relative terms. If you need 10 pixels, no more and no less, then go ahead and use pixels.

The results of our extend are shown in Listing 5.12.

Listing 5.12 Creating a Placeholder Selector

```
/**
 * Archive Next/Previous Navigation
 */
.nav-next-previous a.nav-previous:after, .nav-next-previous a.nav-
next:before { #A
  background-color: #85745f;
  border-radius: 100em;
  color: #FFFFFF;
  display: block;
  font-family: genericons;
  font-size: 2em;
  /* 36px / 18px */
  font-style: normal;
  height: 1.11111111111111em;
  /* 40px / 36px */
  line-height: 1.11111111111111em;
  /* 40px / 36px */
  text-align: center;
  width: 1.11111111111111em;
  /* 40px / 36px */
}

.nav-next-previous a {
  color: #85745f;
  display: block;
  font-size: 1.38461538461538em;
  /* 18px / 13px */
  font-style: italic;
```

```
  line-height: 2.22222222222222em;
  /* 40px / 18px */
}
.nav-next-previous a.nav-previous {
  float: right;
}
.nav-next-previous a.nav-previous:after {
  content: '\f429';
  float: right;
  margin-left: 10px;
}
.nav-next-previous a.nav-next {
  float: left;
}
.nav-next-previous a.nav-next:before {
  content: '\f430';
  float: left;
  margin-right: 10px;
}
.nav-next-previous a:hover, .nav-next-previous a:active, .nav-next-previous
a:focus {
  color: #771717;
}
.nav-next-previous a:hover:before, .nav-next-previous a:hover:after, .nav-
next-previous a:active:before, .nav-next-previous a:active:after, .nav-
next-previous a:focus:before, .nav-next-previous a:focus:after {
  background-color: #771717;
}
```

#A Notice that the name of our placeholder selector isn't written, only the selectors that are extending it.

The results of our page navigation can be seen in figure 5.6:

Figure 5.6 Creating next page/previous page navigation by identifying the traits in common, and using @extend with placeholder selectors to apply them.

When working with @extend, they should be included *before* any unique properties that you're going to add on. If there is a property in the extend that you need to override, you can do that easily through the cascade.

Unfortunately, @extend has some issues with media queries because of the way Sass merges the selectors together in other parts of the document. I find that it's usually best to use @extend only in the base version of your styles (before any media queries are applied), and use other techniques (such as mixins, which we'll look at below) to carry your properties inside media queries through to multiple elements. Mixins aren't as efficient as @extend, but are more flexible in where they can be used.

But hey, did somebody mention media queries?

5.3.5 Media Query Bubbling

Media query bubbling is the awesome sauce that is going to make our responsive design *so* much easier to implement, and one of the major reasons why using a CSS preprocessor in your responsive theme is simply a good idea. In pure CSS, the media queries are usually created after we've set the default properties for an element, as we saw in chapter two. With media query bubbling, however, the media queries are set *inside* the element we're adjusting in a manner that looks a lot like nesting.

We've come pretty far in our responsive prototype without actually doing anything that would make it responsive. Let's fix that right now, by adding the "medium" view to our site header and primary navigation. As we saw in our wireframes in chapter three, the narrow view of the site header featured the site logo, site title and site description stacked one atop another. In the medium view, the logo will float to the left, while the site title and description will sit beside it. Our primary navigation will change from a toggle on and off button to a horizontal list of menu choices. Let's use media query bubbling in listings 5.13and 5.14to make this happen.

Listing 5.13 Media Query Bubbling on the Site Header in _header.scss

```
header[role="banner"] {
  @extend .clearfix; #A

  background: url(http://lorempixel.com/320/255/nature/2) no-repeat center
center;
  background-size: cover;
  line-height: 1.2;
  padding: 2em 18.75%; /* 26px / 13px; 60px / 320px */
  text-align: center;

  @media only all and (min-width: $bp-medium) { #B
    background-image: url(http://lorempixel.com/640/255/nature/2/); #C
    padding: 0;
    text-align: left;
  }
}

header[role="banner"] a {
```

```
  color: #FFFFFF;
  text-decoration: none;
}

header[role="banner"] .logo {
  display: block;
  margin: 0 auto 2.30769230769231em; /* 30px / 13px */

    @media only all and (min-width: $bp-medium) {
      float: left;
      margin: 1.92307692307692em 3.90625%;
    }
}

header[role="banner"] h1,
header[role="banner"] .subhead {
  text-shadow: 0 1px 3px rgba(0, 0, 0, 0.75);
}

header[role="banner"] h1 {
  font-size: 2em; /* 26px / 13px */
  margin-bottom: 0.34615384615385em; /* 9px / 26px */

  @media only all and (min-width: $bp-medium) {
    font-size: 2.30769230769231em; /* 30px / 13px */
    margin-top: 1.46666666666667em; /* 44px / 30px */
  }
}

header[role="banner"] .subhead {
  font-size: 1.23076923076923em; /* 16px / 13px */

  @media only all and (min-width: $bp-medium) {
    font-size: 1.53846153846154em; /* 20px / 13px */
  }
}
```

#A We only need to extend the clearfix selector here when the contents inside the header start to float in wider views, but since we've already said how @extend doesn't play nice with media queries, we'll put it in the base style for the header. Don't worry, it won't hurt anything,nor will it add any weight to our page.
#B We haven't quite determined exactly what our breakpoint will be, so we'll set it as a variable on our prototype.scss page, underneath the color variables. I've initially set it to 600px, but we can fine tune that if we need to, and only have to change it in one place.
#C Our first taste of responsive imagery: as the site gets wider, we'll be fetching a larger background image from our placeholder image site.

Listing 5.14 Media Query Bubbling on the Primary Navigation in _navigation.scss

```
nav[role="navigation"] .toggle-menu {
  font-size: 1.38461538461538em; /* 18px / 13px */
  height: 1.77777777777778em; /* 32px / 18px */
  line-height: 1.77777777777778em; /* 32px / 18px */

  @media only all and (min-width: $bp-medium) { #A
    display: none;
```

```
  }
}

[…]

nav[role="navigation"] ul {
  background-color: $red;
  display: none;

  @media only all and (min-width: $bp-medium) {
    display: block;
  }
}

nav[role="navigation"] li {
  border-top: 1px solid rgba(255, 255, 255, 0.1);

  @media only all and (min-width: $bp-medium) {
    float: left;
    position: relative;
    white-space: nowrap;

&:hover > ul { #B
      display: block;
    }

    a {
      display: block;
      padding: 0 1em;
    }
  }
}

nav[role="navigation"] ul a:hover,
nav[role="navigation"] ul a:active,
nav[role="navigation"] ul a:focus,
nav[role="navigation"] li.current_page_item > a {
  background-color: rgba(255, 255, 255, 0.1);
}

@media only all and (min-width: $bp-medium) { #C
  nav[role="navigation"] ul ul {
    box-shadow: -2px 0 2px rgba(0, 0, 0, 0.2);
    display: none;
    position: absolute;

    li {
      float: none;

      a {
        display: block;
        min-width: 180px;
      }
    }

    ul {
```

```
        position: absolute;
        top: 0;
        left: 100%;
      }
    }
  }

  nav[role="navigation"] ul ul a {
    padding: 0 7.5%;

    @media only all and (min-width: 600px) {
      padding: 0 1em;
    }
  }

  nav[role="navigation"] ul ul ul a {
    padding: 0 11.25%;

    @media only all and (min-width: 600px) {
      padding: 0 1em;
    }
  }
```

#A Once again, we're employing our breakpoint variable so that it can be easily changed if need-be.
#B You can include descendent selectors inside your media query.
#C The ability to use media query bubbling doesn't preclude being able to use ordinary media queries in the same document.

Aside from being *inside* our declarations, and the fact that we're using a variable to set the breakpoint, these media queries look just like the ones we saw in chapter two. Now let's take a look at how Sass handles this code in listings 5.15 and 5.16.

Listing 5.15 Our Header with Media Query Bubbling Applied

```
header[role="banner"] {
  background: url(http://lorempixel.com/320/255/nature/2) no-repeat center
center;
  background-size: cover;
  line-height: 1.2;
  padding: 2em 18.75%;
  /* 26px / 13px; 60px / 320px */
  text-align: center;
}
@media only all and (min-width: 600px) { #A
  header[role="banner"] {
    background-image: url(http://lorempixel.com/640/255/nature/2/);
    padding: 0;
    text-align: left;
  }
}
header[role="banner"] a {
  color: #FFFFFF;
  text-decoration: none;
}
header[role="banner"] .logo {
  display: block;
```

```
  margin: 0 auto 2.30769230769231em;
  /* 30px / 13px */
}
@media only all and (min-width: 600px) { #B
  header[role="banner"] .logo {
    float: left;
    margin: 1.92307692307692em 3.90625%;
  }
}
header[role="banner"] h1, header[role="banner"] .subhead {
  text-shadow: 0 1px 3px rgba(0, 0, 0, 0.75);
}
header[role="banner"] h1 {
  font-size: 2em;
  /* 26px / 13px */
  margin-bottom: 0.34615384615385em;
  /* 9px / 26px */
}
@media only all and (min-width: 600px) {
  header[role="banner"] h1 {
    font-size: 2.30769230769231em;
    /* 30px / 13px */
    margin-top: 1.46666666666667em;
    /* 44px / 30px */
  }
}
header[role="banner"] .subhead {
  font-size: 1.23076923076923em;
  /* 16px / 13px */
}
@media only all and (min-width: 600px) {
  header[role="banner"] .subhead {
    font-size: 1.53846153846154em;
    /* 20px / 13px */
  }
}
```

#A Sass has taken the media query that was inside the declaration and moved it outside, where it belongs, with the media query declaration properly inside.
#B It works on nested elements too.

Listing 5.16 Our Header with Media Query Bubbling Applied

```
nav[role="navigation"] .toggle-menu {
  font-size: 1.38461538461538em;
  /* 18px / 13px */
  height: 1.77777777777778em;
  /* 32px / 18px */
  line-height: 1.77777777777778em;
  /* 32px / 18px */
}
@media only all and (min-width: 600px) {
  nav[role="navigation"] .toggle-menu {
    display: none;
  }
}
```

```
[…]

nav[role="navigation"] ul {
  background-color: #771717;
  display: none;
}
@media only all and (min-width: 600px) {
  nav[role="navigation"] ul {
    display: block;
  }
}

nav[role="navigation"] li {
  border-top: 1px solid rgba(255, 255, 255, 0.1);
}
@media only all and (min-width: 600px) {
  nav[role="navigation"] li {
    float: left;
    position: relative;
    white-space: nowrap;
  }
  nav[role="navigation"] li:hover > ul {
    display: block;
  }
  nav[role="navigation"] li a {
    display: block;
    padding: 0 1em;
  }
}

nav[role="navigation"] ul a:hover,
nav[role="navigation"] ul a:active,
nav[role="navigation"] ul a:focus,
nav[role="navigation"] li.current_page_item > a {
  background-color: rgba(255, 255, 255, 0.1);
}

@media only all and (min-width: 600px) {
  nav[role="navigation"] ul ul {
    box-shadow: -2px 0 2px rgba(0, 0, 0, 0.2);
    display: none;
    position: absolute;
  }
  nav[role="navigation"] ul ul li {
    float: none;
  }
  nav[role="navigation"] ul ul li a {
    display: block;
    min-width: 180px;
  }
  nav[role="navigation"] ul ul ul {
    position: absolute;
    top: 0;
    left: 100%;
  }
```

```
}
nav[role="navigation"] ul ul a {
  padding: 0 7.5%;
}
@media only all and (min-width: 600px) {
  nav[role="navigation"] ul ul a {
    padding: 0 1em;
  }
}

nav[role="navigation"] ul ul ul a {
  padding: 0 11.25%;
}
@media only all and (min-width: 600px) {
  nav[role="navigation"] ul ul ul a {
    padding: 0 1em;
  }
}
```

Aside from being positioned inside the declaration, media query bubbling in Sass works just like media queries without Sass. All the same inheritance and specificity rules that apply to CSS apply here, and media queries in of themselves add no specificity or importance to the CSS contained within.

We can see, visually, the results of these media queries when we compare our narrow and medium views of our header and primary navigation in figure 5.7:

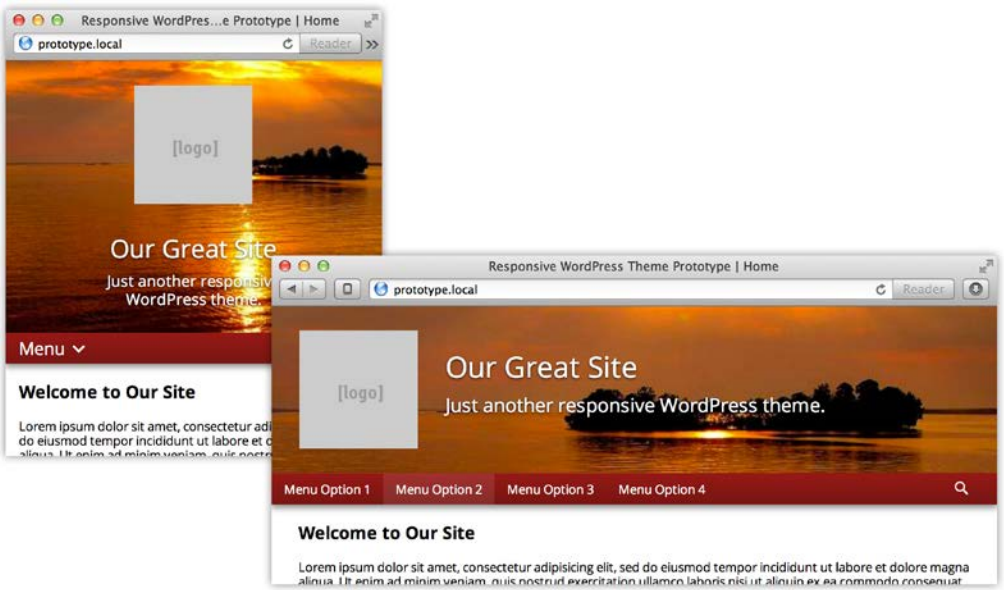


Figure 5.7 Thanks to variables and media query bubbling, we were able to make our header and navigation responsive without touching our markup.

Code Bloating Redux

Haters of code bloat point to the overuse of media query bubbling as the second major source of bloat. Every time you bubble your media query, the CSS preprocessor is going to write out the whole query again, giving you kilobyte after kilobyte of "@media only all and (min-width: 600px)" strewn throughout your document. It's much more efficient, they argue, to put everything that needs to be expressed at a certain breakpoint in *one* media query, at the end of the document.

This, I believe, is a maintainability nightmare. Sure, partials could be used to ease the mess and make it easier to find the code you need to tweak, but at this point, you're micro-optimizing.

If you're minifying (which Sass can do for you) and Gzipping (which you should be doing anyway) your final CSS file, then repeating the same media queries are not going to have a significant impact in file size. Gzipping eats repetitive strings for breakfast (credit to Chris Coyier for saying that), and so the file size hit isn't going to be nearly as large as you might initially think.

5.3.6 Functions

We're on a roll now, but one thing we've spent an inordinate amount of time on is calculating all of those relative measurements. Fortunately, Sass gives us the ability to create functions to which we pass along some values and they, in turn, return a result. So far in our prototype we have two basic calculations going on: the percentages and ems.

Aside from being an incredible time saver by not having to calculate all these values ourselves each and every time, functions can serve a second, more subtle purpose: self-documenting our code. Since when we call these functions we call them with the same target and context values that we used when we were calculating their result manually, there's no need to add all those comments afterwards to describe the calculation you were applying. Bonus!

Both calculating percentages and ems can easily be turned into functions, so let's define them and apply that to creating the medium view of our home page's main content in listing 5.17. In our prototype, we'll add these functions to our **prototype.scss** document.

Listing 5.17 Building Functions for Relative Calculations

```
@function calc-percent($target, $context) { #A
  @return ($target / $context) * 100%; #B
}

@function cp($target, $context) { #C
  @return calc-percent($target, $context);
}

@function calc-em($target, $context) { #D
  @return ($target / $context) * 1em;
}
```

```
@function ce($target, $context) {
  @return calc-em($target, $context);
}
```

#A This function does exactly what it says it does: calculates a percentage based on a given target and context.
#B Multiplying the result of the target divided by the context by 100% returns our value in percent.
#C Less typing, please! This "cp" function is essentially an alias that calls our "calc-percent" function.
#D It works so well for percentages, let's do the same thing with ems.

We can create a simple fluid grid to the main content of our home page, using these functions to do all the percentage and em calculations for us (listings 5.18 and 5.19).When the CSS is compiled, it's still going to have the same crazy multi-decimal percentages it had before, but now it's the computer's problem, not ours.

Listing 5.18 Using Our New "cp" and "ce" Functions in _main.scss

```
main {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1px;
  padding: 1.38461538461538em 3.75%;

  @media only all and (min-width: $bp-medium) {
    float: right;
    margin-bottom: ce(10px, 13px); #A
    width: cp(420px, 640px); #B
  }

[…]

}
```

#A When calculating ems, we have to know what the applicable context is. Usually it's going to be your default font size, but keep an eye out for nested elements whose font size is different or the results won't be what you expect.
#B We're using 640px for our percentage context since that's what our Photoshop comp used as its medium view size.

Listing 5.19Using Our New "cp" and "ce" Functions in _widgets.scss

```
/**
 * Footer Widget Areas
 */

footer[role="contentinfo"] {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1.53846153846154em; /* 20px / 13px */

  @media only all and (min-width: $bp-medium) {
    clear: both; #A
  }
}
```

```
/**
 * Home page
 * (where the two sidebars are grouped together for narrow &
 * medium views)
 */

.sidebars {

  @media only all and (min-width: $bp-medium) {
    float: left;
    margin: 0 cp(10px, 640px) ce(10px, 13px) 0; #B
    width: cp(210px, 640px);
  }
}
```

#A Since we're going to have elements before it in the DOM floating every which way, we need to make sure our footer area will clear them.
#B Yes, you can use two functions in one declaration if it makes to do so, as it does here.

When we run this code through Sass, we see that all those tedious calculations have been done for us (listing 5.20).

Listing 5.20 Result of Our "ce" and "cp" Functions

```
main {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1px;
  padding: 1.38461538461538em 3.75%;
}
@media only all and (min-width: 600px) {
  main {
    float: right;
    margin-bottom: 0.76923em;
    width: 65.625%;
  }
}

[…]

footer[role="contentinfo"] {
  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1.53846153846154em;
  /* 20px / 13px */
}
@media only all and (min-width: 600px) {
  footer[role="contentinfo"] {
    clear: both;
  }
}

/**
 * Home page
```

```
   * (where the two sidebars are grouped together for narrow &
   * medium views)
   */
  @media only all and (min-width: 600px) {
    .sidebars {
      float: left;
      margin: 0 1.5625% 0.76923em 0;
      width: 32.8125%;
    }
  }
```

Now we have a responsive, fluid grid for our medium view on our main content and sidebar. Let's take a look in figure 5.8:

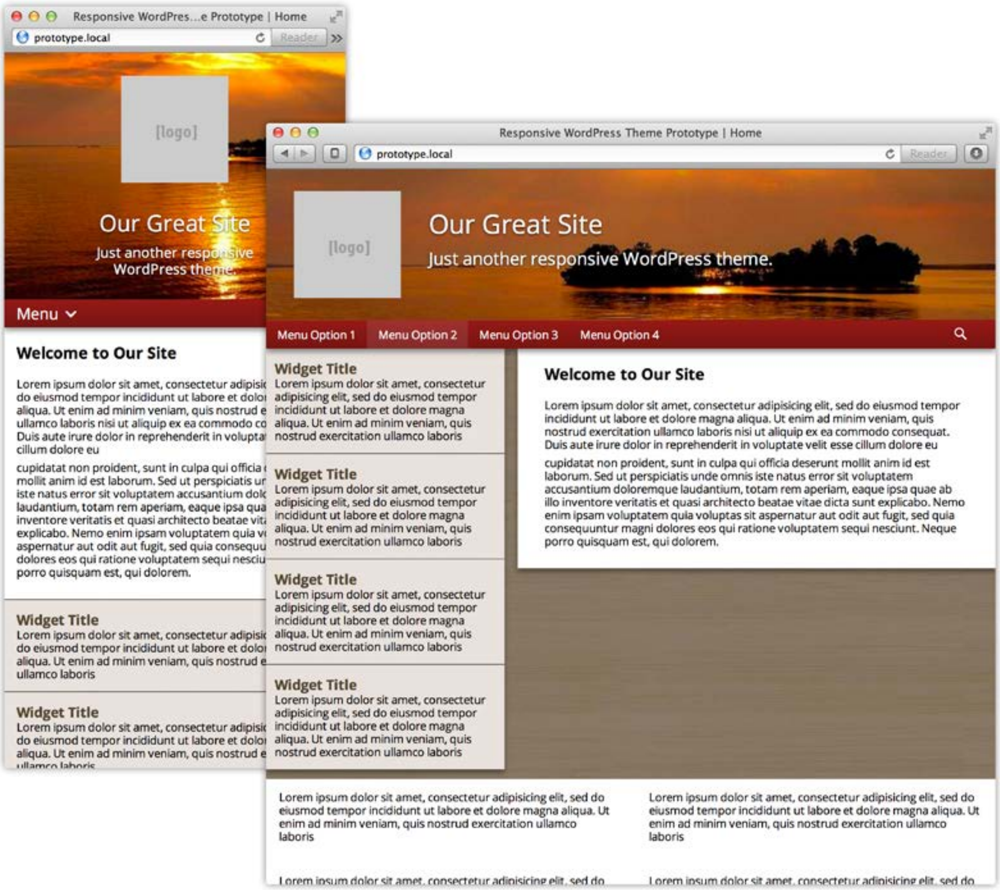


Figure 5.8 Now as the browser window expands, the sidebar slides up and floats to the left of the main content area.

We do have an issue where the main content comes short of the sidebar's length. We can fix that with some JavaScript, or just leave it as-is. In the future, CSS flexbox will fix this for us, but as of the writing of this book, that CSS feature isn't ready for prime time.

5.3.7 Mixins

Functions are great, but sometimes they're not enough to get the output we're looking for. Similar in structure and purpose to functions are *mixins*, but the result of a mixin is written CSS code that goes to the browser, for those times when we're looking for more than just a value.

So far in our stylesheet, we've been using ems as our unit for vertical measurements. But what if we wanted to use rems instead? Rems are great, but they come with the drawback that they're not supported in versions of Internet Explorer below 9. This is problematic, since there are quite a few Internet Explorer 8 (and even some IE7) users left in the world.

What Is a REM?

You're probably familiar with the relative unit of measurement known as "ems." In modern web typography, one em is equal to the height of the current font in pixels.

We love relative units in responsive web design, but ems have one significant drawback: Since they're based on the font size of the parent element, nested elements tend to be somewhat unpredictable as the element could inherit, *ad naseum*, the font size of its parents straight through to the top of the DOM. This means that if you set the font size of a list item to 0.8em, intending it to be about 11px, any list items nested inside the original list would end up being 9px (11px x 0.8 = about 9px). And so on, and so forth.

Enter the rem, or *rootem*. Instead of taking the parent element as its context, it uses the font size for the <html> element itself. Since there is only ever one HTML element, font sizes defined in rems will act consistently, no matter where the element appears in the DOM or how nested it is. If you then decide to change the default font size in the <html> element, all the elements whose font sizes are given in rems will follow suit.

Rems are preferable to ems, but if we want to use them, we'll need to provide a fallback by also providing the font size in pixels. For that, we can employ a mixin, such as the one in listing 5.21.

Listing 5.21 Creating a "rem" Mixin

```
@function calc-rem( $target, $context: $default-font-size ) { #A
  @return $target / $context * 1rem;
}

@function cr( $target, $context: $default-font-size ) { #B
  @return calc-rem( $target, $context );
}
```

```
@mixin rem($property, $value, $context: $default-font-size) { #C
     #{$property}: $value; #D
     #{$property}: cr($value, $context); #E
}
```

#A We'll start by defining our function to calculate our rems just as we did for the percentages and ems. Since the context is pretty much always going to be the font size we've set for the <HTML> element, we'll probably never pass that along as a variable. Instead, we'll set it to whatever our site's default font size is, which we've declared where all our other variables are. For the purposes of this site, the default font size is 13px.
#B Again with the shortcut to our shortcut.
#C In addition to our target and context values (and ditto on probably never passing along the context value), we need to know what CSS property this will apply to.
#D The property is expressed here via interpolation. The value on this line is the unmodified value we fed into the mixin.
#E Here we have the line repeated, except we're calling for the calculated rem value. Browsers that don't know what rems are will ignore this line and follow the pixel-based property above it.

We have one major piece left to our prototype that we haven't yet made responsive: the footer widget areas. They're going to need margins around them to keep from bumping into each other, and I'd like the top and bottom margins to be expressed in rems, where possible. Let's do that in listing 5.22.

Listing 5.22 Making Our Footer Widget Areas Responsive

```
footer[role="contentinfo"] {
  @extend .clearfix; #A

  background-color: #FFFFFF;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.4);
  margin-bottom: 1.53846153846154em; /* 20px / 13px */

  @media only all and (min-width: $bp-medium) {
    clear: both;

    .footer-widget-area {
      float: left;
      margin-right: cp(10px, 640px);
      width: cp(315px, 640px);

      @include rem(margin-bottom, 10px); #B

&:nth-of-type(3) { #C
        clear: left;
      }

&:nth-of-type(2), &:nth-of-type(4) {
        margin-right: 0;
      }
    }
  }
}
```

#A Again, since we're getting a bit floaty in here, we'll want to extend the "clearfix" class to our footer container element.

#B The rem mixin called for three parameters, but we've only passed along two. The third will default to the value we set for it (in this case, the default font size) unless we choose to override it for some reason.
#C We're using some fancy CSS3 selectors here to remove margins from the widget areas that don't need them. Use with caution, as they're not supported in IE8 or below.

We now have a responsive set of footer widget areas (figure 5.9):

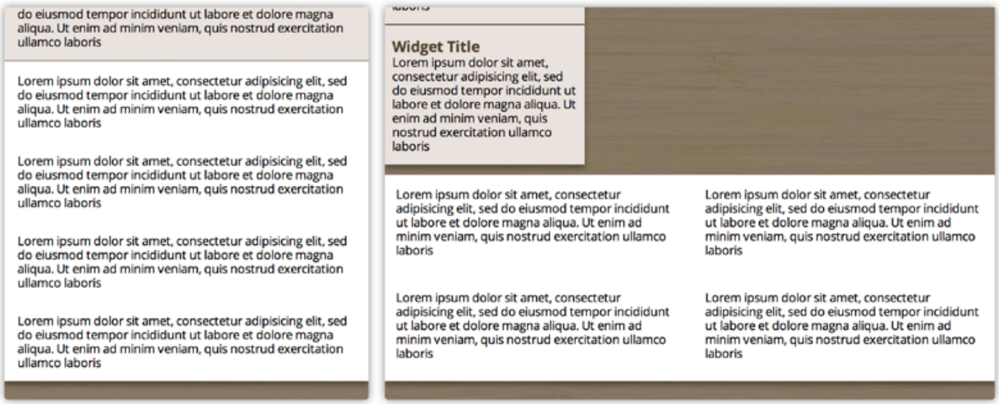


Figure 5.9 A stacked footer for the narrow view, and a two-by-two grid of footer widget areas in the medium view.

5.4 Summary

In this chapter we explored CSS preprocessors and found how they can make our responsive web development go much more smoothly and quickly. The material we covered will get you started and help you make your CSS code more maintainable, efficient and easy to work with. There's much more we can do with Sass (and Compass along with it); we've only scratched the surface. As we work on our project, we'll uncover more of Sass' handy features, such as control logic (that's if/else statements), looping, color functions and more! If this chapter has made you hungry for more information on how to use Sass and Compass, you might want to check out *Sass and Compass in Action*, written by the project maintainers themselves.

We didn't complete every line in our prototype in the code samples in these last two chapters, but the code that you'll find on the book's site will have the complete and final prototype code, including the CSS with Sass.In the meantime, we're going to continue on with our project and shift our focus to WordPress itself. In our next chapter, we're going to set up our WordPress development environment, looking at the various plugins and other tools we can use to help us create our living, working theme.

6

Setting Up Our WordPress Development Environment

This chapter covers

- Basic WordPress theme structure
- Creating a child theme
- Theme frameworks
- Popular WordPress “starter” themes
- Helpful plugins for WordPress theme development
- Using CSS preprocessors in WordPress

We’ve done a lot of preparation just to get to this point. We’ve planned, sketched, designed and coded our way to a perfect prototype. Now it’s time to take it to the next step: create a home for developing ourtheme, and equip ourselves with the tools that will help us build it.

In this chapter, we’re going to set up an environment in which we will turn our prototype into a living, breathing WordPress theme. We’ll take a look at the nuts and bolts of how a WordPress theme does what it does: pull in content from the database and push it to the browser in an (ideally) attractive way.We’ll look at how to leverage the efforts of the WordPress community by creating child themes, working with theme frameworks, and building off of starter themes. Finally, we’ll take a look at some indispensable plugins and other tools that will help us along in the process.

Since this book is not a basic “get started with WordPress” kind of title, it’s assumed that you already know how to set up your own WordPress development environment. If this is not the case, I’ve provided instructions in appendix A on one way of installing a WordPress

instance on a Mac OS X environment, and listed some other resources for installing WordPress on other platforms.

6.1 Basics of WordPress Theme Structure

Back in chapter two, we covered the bare essentials we needed in order for a theme to be considered a theme. All you really, *really* need is an **index.php** file and a **style.css** stylesheet, without either of which WordPress will complain and not display your site.We alsodiscussed how every WordPress theme you're likely to encounter also contains a **functions.php** file that holds all the custom template tags and other functionality unique to the theme. But practically speaking, that's just the tip of the iceberg. Let's examine some of the other template files we'll be building for our theme, and what purpose they serve.

6.1.1 style.css

Somehow we need to tell WordPress about our theme, and that "how" is through the **style.css** file. *Note that it actually has to be called* ***style.css****!* Yeah, it's kinda*that* important.

The **style.css** file must start with a bank of specifically-formatted comments that gives WordPress all the details about the theme, otherwise known as the meta data. Only the theme name is absolutely required, but it helps out users of your theme if you give them some other information such as the author, your website, the theme's website, and a bit of a description. You should also state your licenseof preference(see the sidebar, below) and version number. If you want to put this in the WordPress theme repository, then you also need to assign it tags from a list of predefined choices. These tags are one-word descriptions used to denote the dominant colors, theme features, column options, etc. The list of acceptable tags can be found at http://wordpress.org/themes/about/.

An example of such a comment block, the one for the default theme Twenty Thirteen, is shown in listing 6.1. Most of the items should be self-explanatory, but see the annotations for a description of some of them.

Listing 6.1 Twenty Thirteen's style.css Meta Data

```
/*
Theme Name: Twenty Thirteen #A
Theme URI: http://wordpress.org/themes/twentythirteen
Author: the WordPress team
Author URI: http://wordpress.org/
Description: The 2013 theme for WordPress takes us back to the blog,
featuring a full range of post formats, each displayed beautifully in their
own unique way. Design details abound, starting with a gorgeous color
scheme and matching header images, optional display fonts for beautiful
typography, and a wide layout that looks great on large screens yet remains
device-agnostic and is readable on any device.
Version: 0.1
License: GNU General Public License v2 or later #B
License URI: http://www.gnu.org/licenses/gpl-2.0.html
Tags: black, brown, orange, tan, white, yellow, light, one-column, two-
```

```
columns, right-sidebar, flexible-width, custom-header, custom-menu, editor-
style, featured-images, microformats, post-formats, rtl-language-support,
sticky-post, translation-ready #C
Text Domain: twentythirteen

This theme, like WordPress, is licensed under the GPL.
Use it to make something cool, have fun, and share what you've learned with
others.
*/
```

#A You gotta have a *name* for this thing.
#B Choose your licenses carefully. See the sidebar below.
#C You can't just choose tags willy-nilly. There's a list of acceptable tags at http://wordpress.org/themes/about/.

A Word About Licensing

The subject of licensing comes up on a regular basis in the WordPress community. The fact that WordPress is licensed under the GNU Public License, version 2 (GPLv2 for short) is something that Matt Mullenweg and the lead maintainers take very seriously. At times they have taken to task purveyors of commercial themes that have adopted more stringent, non-open, and non-compatible licensing for their work.

Just because a theme is licensed under GPLv2 doesn't mean that the author can't charge money for it. On the contrary, there are many successful commercial theme shops and individual theme authors whose products are sold with a GPLv2 license. The heart of the GPLv2 license is freedom of use, modification and distribution, leading to the catch phrase "free as in speech, not free as in beer."

If you want your theme to go onto the theme repository, it *must* be compatible with GPLv2, but it doesn't have to *be* GPLv2. You can view WordPress' specific licensing requirements at http://make.wordpress.org/themes/guidelines/guidelines-license-theme-name-credit-links-up-sell-themes/.

There are a number of open source licenses that are compatible, but differ in their terms. My personal favorite is the M.I.T. license (http://opensource.org/licenses/MIT) that states, in a nutshell, do whatever you want with this code, but don't come blaming me if your program gains sentience and tries to kill us all. Frankly, I'm not one to get hung up on restrictions, forcing my personal licensing preferences on others, or requiring contributions back to the community, so I tend to go with the most permissive license out there.

6.1.2 Template Files

There are a slew of specialized files that can handle different parts of our web page and different types of pages. Commonly in WordPress themes, you'll see the three files – **style.css**, **index.php** and **functions.php** – listed above as well as those listed in table 6.1.

Table 6.1 Common WordPress Template Files

Template File Name	Purpose
header.php	Site-wide informational header and primary navigation
footer.php	Site-wide informational footer
sidebar.php	Widget areas, commonly displayed as sidebars
archive.php	Listing of posts
single.php	Individual post
page.php	Static page
404.php	A page that displays when the user gets a “File Not Found” error
comments.php	List of comments for a post or page, as well as the comments form
search.php	Search results
searchform.php	Search form

There are many more possibilities, but these are the most basic ones, and the ones that we'll need for our theme. With them, you have the means to define almost any kind of content the site is likely to contain.

The WordPress core software uses a series of tests to determine which template file it should use in order to display the requested information. For this reason, template files must follow a particular naming convention in order to be recognized by the system. Figure 6.1, the WordPress Template Hierarchy, shows the flow chart by which WordPress decides which template file to use. If the optimal, specific template file doesn't exist, WordPress will simply fail over to the next best choice, straight down until it hits **index.php** (which is why all themes are required to have an **index.php** page).

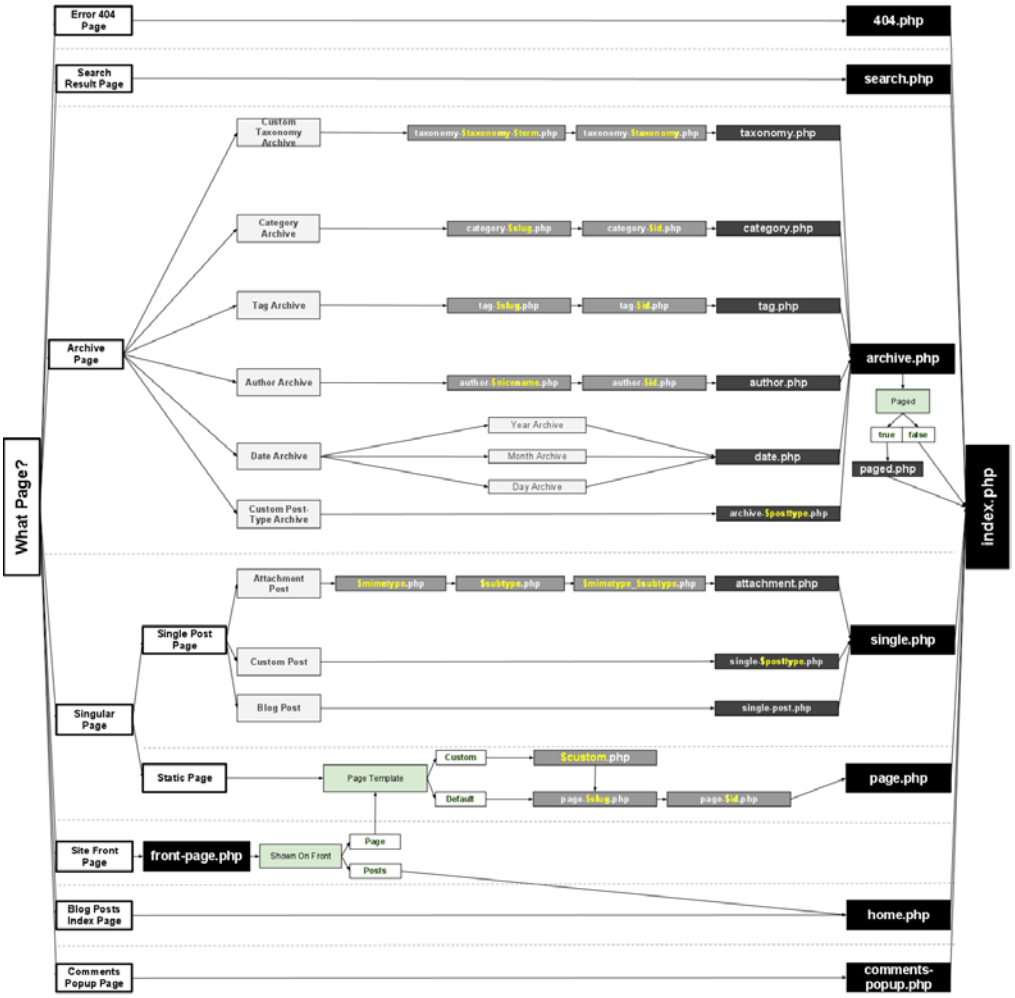


Figure 6.1 Depending on the type of page being requested, WordPress will keep looking for the proper template file until it finds one that qualifies. This graphic lives at http://codex.wordpress.org/Template_Hierarchy, and it might be a good idea to bookmark it.

You can get very, very specific when creating template files. You can have template files that only serve posts in a particular category or written by a particular author. You can even have templates that only are called for one particular post or page! We won't be doing anything so fancy here;rather we'll keep things as generic as possible.

6.1.3 The Loop

No discussion of how a WordPress theme works would be complete without mentioning the Loop. At its heart, the Loop is just the means of iterating (or looping, get it?) through the content that is to be displayed on the page, whether that be a long list of posts in an archive, or just the content for one particular post or page. Listing 6.2 shows an example of the Loop in the **page.php** template of Twenty Twelve.

Listing 6.2 The Loop in page.php

```
<div id="primary" class="site-content">
<div id="content" role="main">

<?php while ( have_posts() ) : the_post(); ?> #A
<?php get_template_part( 'content', 'page' ); ?> #B
<?php comments_template( '', true ); ?>
<?php endwhile; // end of the loop. ?> #C

</div><!-- #content -->
</div><!-- #primary -->
```

#A This is the code that makes the Loop happen.
#B In Twenty Twelve, most of the work that goes on inside the Loop happens in another file. get_template_part() calls that file and inserts the results here.
#C Our "endwhile" statement closes the loop. If there is more content to be gone through, WordPress will go back to the beginning of the loop and start the process again.

The Loop is initiated by the statement

```
while ( have_posts() ) : the_post();
```

which states that so long as we haven't exhausted our supply of items (posts, pages, or other custom post types), take the next one in the queue and work with that. If this were a single post page, or a page of static content, there would be one item to work with and then the process would exit. If this were an archive listing, then the Loop would iterate through the entire list one by one, exiting out when the last one was finished.

Code that is inside this `while`/`endwhile` construction is said to be "inside the Loop," and certain template tags will only work if they're inside the Loop. Conversely, code not inside this construction is said to be "outside the Loop." If code referring to a specific post or page is outside the Loop, we'll need to specifically identify it.

6.2 Working from Existing Sources

While it's possible to create an excellent WordPress theme from scratch, and many people do so for their day-to-day work, it's often not necessary, or even advisable to do so if you're new to WordPress theming. There are a number of ways to build your theme off a solid foundation that was started by other members of the WordPress community. Whether through the parent and child theme architecture, using a theme framework, or taking a

starter theme and customizing to meet your needs, there's really no need to start from a blank slate in order to create the perfect theme of your own.

6.2.1 Parent and Child Themes

Parent and child themes are best suited for those times where you find a theme that's really close to what you want your site to look like, save for a few details. Child themes inherit many of the properties of its parent, including template files and the code in the **functions.php** file, but allow you to easily override any of the information (be it in the template files or the stylesheet) that you want.

Why not modify the parent theme directly? In a word: updates. If we're using a popular theme that is likely to receive lots of updates, we have the choice of either overriding all of our changes every time we update, or not updating at all (and door number three, trying to port the updates, manually, to our altered version of the code is just a bag of hurt). Given that themes are subject to security holes and fixes as any other collection of code, we want to make sure we have the ability to make updates without destroying our work.

If you were using a parent/child theme setup, it's likely that you wouldn't have gone through the whole design process that we covered in chapter three. More likely, you're looking to make small changes that don't need a whole lot of mapping out.

Let's go ahead and make a child theme of the WordPress 3.6 default theme, Twenty Thirteen. We'll start off by creating a new directory in our **wp-content/themes/** directory to hold our theme, calling it **twentythirteen-child**. Inside, we'll create the **style.css** file that will tell WordPress what it needs to know about this theme (listing 6.3).

Listing 6.3 Our Child Theme's style.css

```
/*
Theme Name: Twenty Thirteen Child #A
Description: A child theme of Twenty Thirteen
Author: Tracy F. Rotton
Author URI: http://www.taupecat.com/

Template: twentythirteen #B
*/
```

#A We need to give our new theme a unique name, even if it *is* entirely derivative.
#B This is new. We're telling WordPress that this theme uses another theme as a template.

In addition to the standard information we're telling WordPress, we've added a new item called "Template." This tells WordPress the theme (that exists in our WordPress instance) that we're basing this child theme off of. The parameter we give it is the *directory name* of the parent theme, as you see it when you look in **wp-content/themes/**. The directory name for Twenty Thirteen is "twentythirteen," so that's our value here.

From this point on, WordPress is going to be very lax in what it needs from us in our child theme, because it knows that whatever it can't find here it can find in the parent theme. We can see the result when we go to the "Themes" page in the admin (figure 6.2):

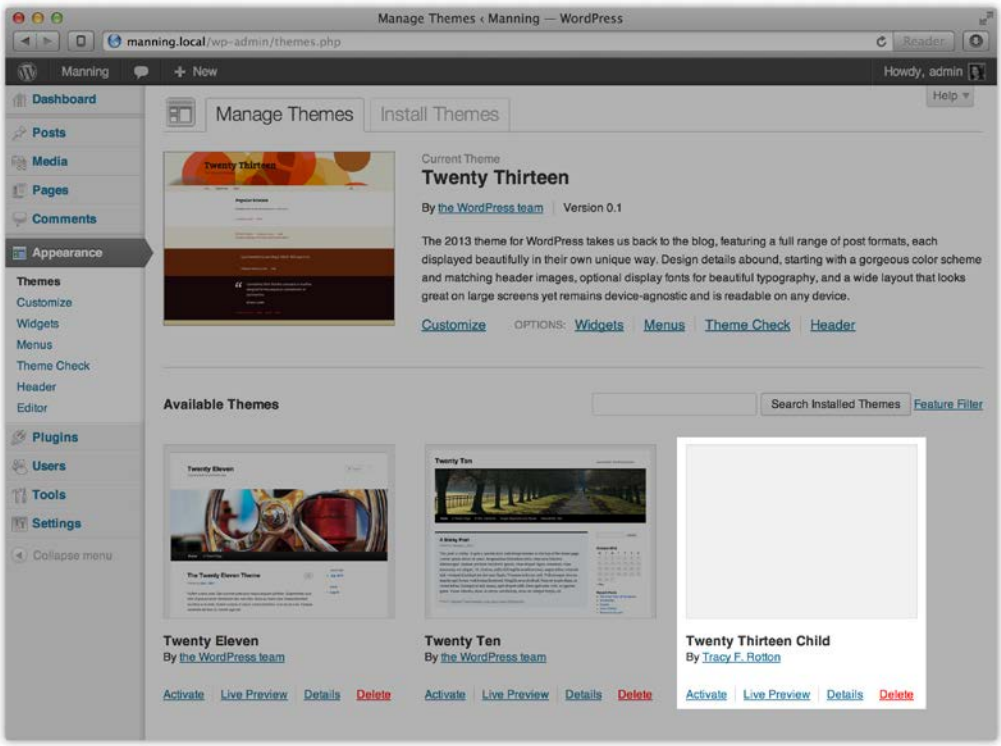


Figure 6.2 The Themes section of the admin now shows us our child theme.

Because we haven't provided a screenshot (not that we have one to provide yet), there's a big blank square above our child theme's name. No matter. Go ahead and click "Activate" and we'll begin our brief child theming exercise. We're going to take Twenty Thirteen's distinctive orange color scheme and cool it down into a softer, blue-green palette. Oh, and we'll make one small change to the header markup while we're at it.

The first thing we need to do is to create our **style.css** file and bring in the styles from our parent theme. Child themes *do not* automatically inherit the CSS of their parent themes; for that we must do so explicitly. The simplest way to do so is to use a standard CSS `@import` directive in Twenty Thirteen Child's **style.css**, like so:

```
@import "../twentythirteen/style.css";
```

Yes, yes, I know that's adding another HTTP request, and thus adding to our load time. While we *could* copy the CSS from the parent theme directly into the child theme, we're back to losing any updates to the parent's CSS trickling down into our child theme. Sometimes, performance and maintainability have to do a balancing act.

Now when we load our site's home page, we'll see something that looks very much like Twenty Thirteen (figure 6.3):

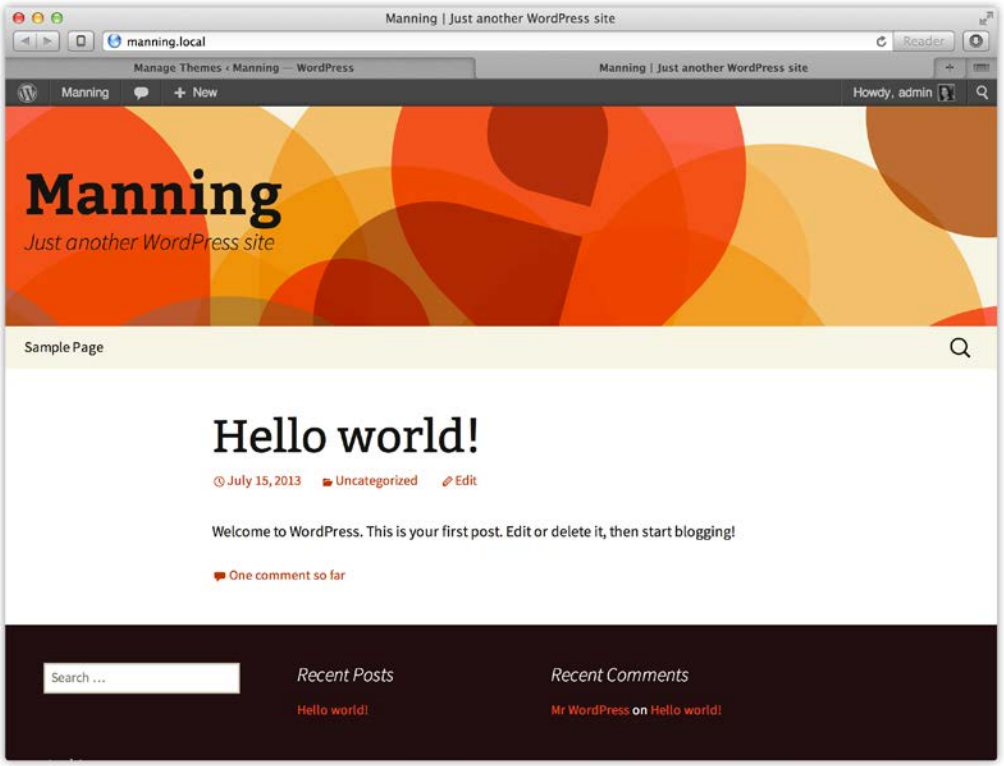


Figure 6.3 It's like they say, the apple doesn't fall very far from the tree. Our child theme looks *just* like its parent!

All we want to change is some colors, not reinvent the whole wheel. Since cascading stylesheets are very good at, well, cascading, we need only to override the colors in our child's **style.css**. Listing 6.4 shows us the declarations we need.

Listing 6.4 Twenty Thirteen Child's CSS

```
a {
  color: #ca3c08;
}

a:visited {
  color: #1e854a;
}
```

```
a:active,
a:hover {
  color: #27ae60;
}

.navbar {
  background-color: #ecf0f1;
}

.entry-title a:hover {
  color: #27ae60;
}

.entry-meta a {
  color: #1e854a;
}

.entry-meta a:hover {
  color: #27ae60;
}

.site-footer {
  background-color: #ecf0f1;
}

.site-footer .sidebar-container {
  background-color: #34495e;
}

.site-footer .widget a {
  color: #ecf0f1;
}
```

Note that these are only some of the colors we need. Twenty Thirteen is a complex theme with lots of different variations for different post formats, and I don't have the space here to go into each one individually. Still, listing 6.4 should give you some idea of how you can use the stylesheet of a child theme to alter the properties defined by the parent theme. The results of these changes can be seen in figure 6.4:

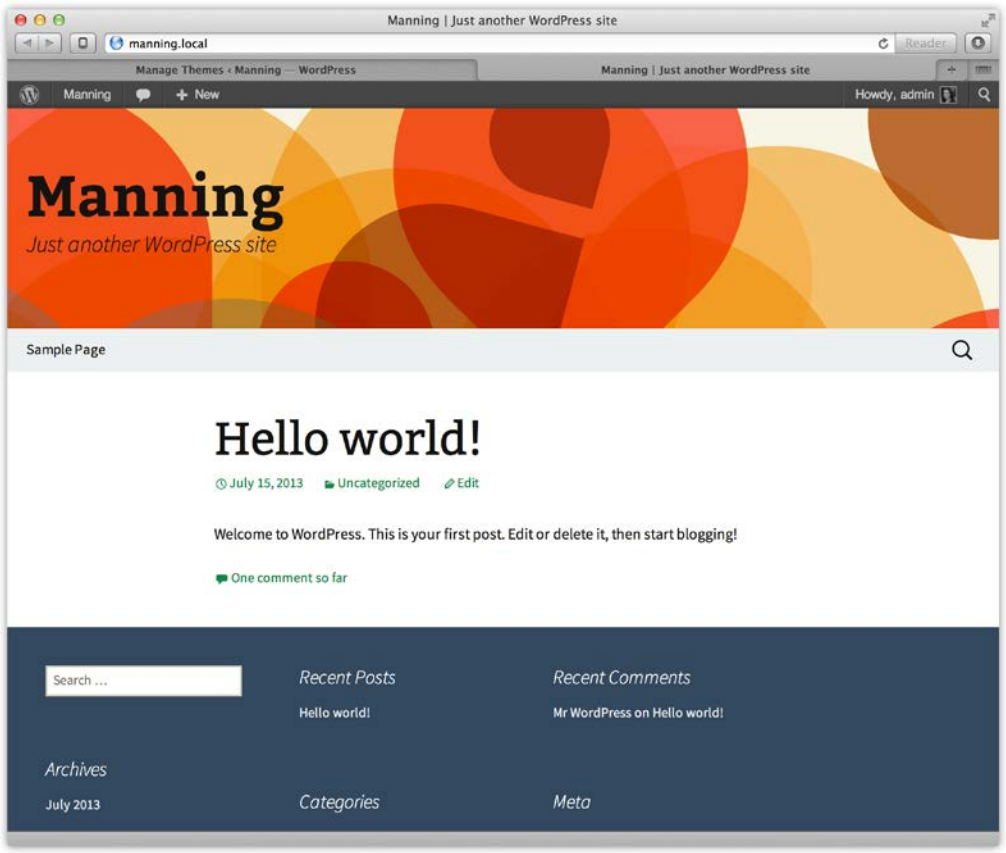


Figure 6.4 Everything is now nice and blue and green and serene… except for the header image!

With our cooler color scheme, the bright orange and red header image really stands out in a not-too-pleasing sort of way. Since that is changeable through the theme admin, all we need to do is take that header, adjust its hue in the image processor software of our choice, and upload it through the admin (figure 6.5).

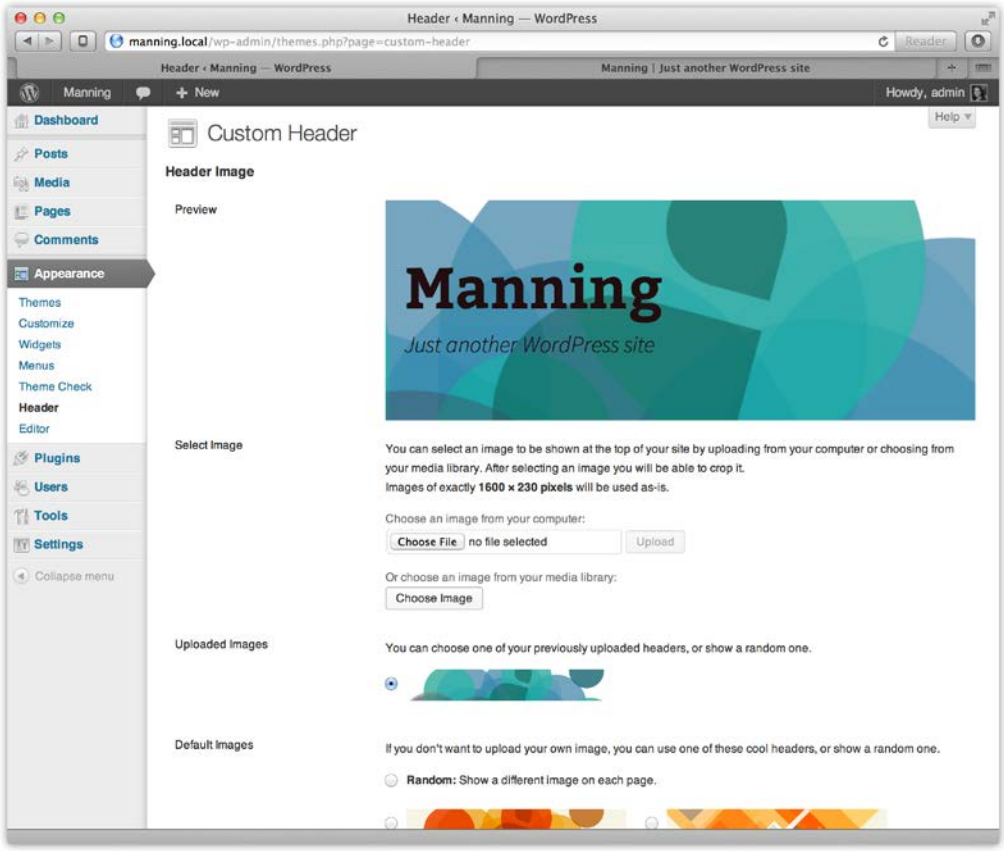


Figure 6.5The admin interface allows us to upload any header we want to use for our site.

A quick adjustment, and voilà! We have a cooler, more calming variant of Twenty Thirteen (figure 6.6):

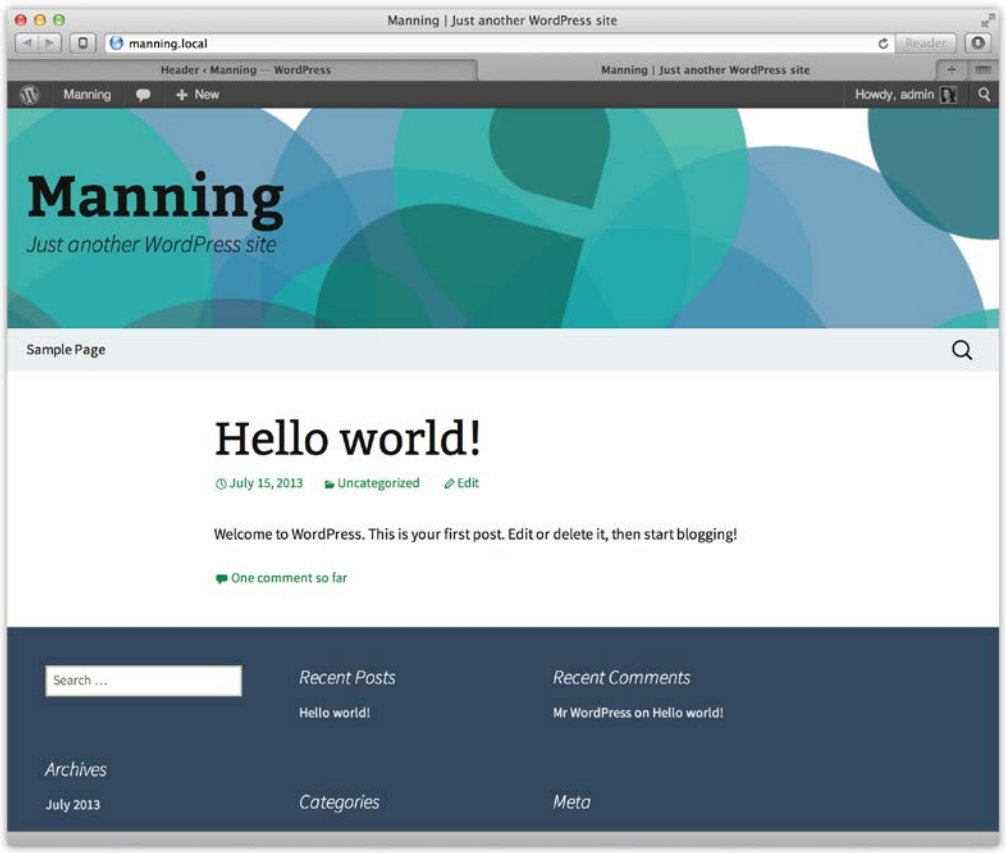


Figure 6.6Blue is the new orange.

Now all we need is to make one small header change. As I indicated before, WordPress will happily read the template files for a parent theme if the child theme does not have them, which is why our theme is working perfectly fine with zero template files at the moment. Any template the child them *does* possess, however, takes precedence over the same file in the parent.

Let's copy our parent's **header.php** file to our child theme's directory and make a small change in listing 6.5.

Listing 6.5 Twenty Thirteen Child's header.php

```
<?php
/**
 * The Header for our theme.
 *
```

```
 * Displays all of the <head> section and everything up till <div
id="main">
 *
 * @package WordPress
 * @subpackage Twenty_Thirteen
 * @since Twenty Thirteen 1.0
 */
?><!DOCTYPE html>
<!--[if IE 7]>
<html class="ie ie7" <?php language_attributes(); ?>>
<![endif]-->
<!--[if IE 8]>
<html class="ie ie8" <?php language_attributes(); ?>>
<![endif]-->
<!--[if !(IE 7) | !(IE 8)  ]><!-->
<html <?php language_attributes(); ?>>
<!--<![endif]-->
<head>
<meta charset="<?php bloginfo( 'charset' ); ?>">
<meta name="viewport" content="width=device-width">
<title><?php wp_title( '|', true, 'right' ); ?></title>
<link rel="profile" href="http://gmpg.org/xfn/11">
<link rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>">
<!--[if lt IE 9]>
<script src="<?php echo get_template_directory_uri();
?>/js/html5.js"></script>
<![endif]-->
<?php wp_head(); ?>
</head>

<body <?php body_class(); ?>>
<div id="page" class="hfeed site">
<header id="masthead" class="site-header" role="banner">
<div id="navbar" class="navbar"> #A
<nav id="site-navigation" class="navigation main-navigation"
role="navigation">
<h3 class="menu-toggle"><?php _e( 'Menu', 'twentythirteen' ); ?></h3>
<a class="screen-reader-text skip-link" href="#content" title="<?php
esc_attr_e( 'Skip to content', 'twentythirteen' ); ?>"><?php _e( 'Skip to
content', 'twentythirteen' ); ?></a>
<?php wp_nav_menu( array( 'theme_location' => 'primary', 'menu_class' =>
'nav-menu' ) ); ?>
<?php get_search_form(); ?>
</nav><!-- #site-navigation -->
</div><!-- #navbar -->
<a class="home-link" href="<?php echo esc_url( home_url( '/' ) ); ?>"
title="<?php echo esc_attr( get_bloginfo( 'name', 'display' ) ); ?>"
rel="home">
<h1 class="site-title"><?php bloginfo( 'name' ); ?></h1>
<h2 class="site-description"><?php bloginfo( 'description' ); ?></h2>
</a>
</header><!-- #masthead -->

<div id="main" class="site-main">
```

#A The only difference we made is switching the positions of the navbar (which is now on top) and the site header (which is now underneath).

The result of this header/navbar swapping is shown in figure 6.7:

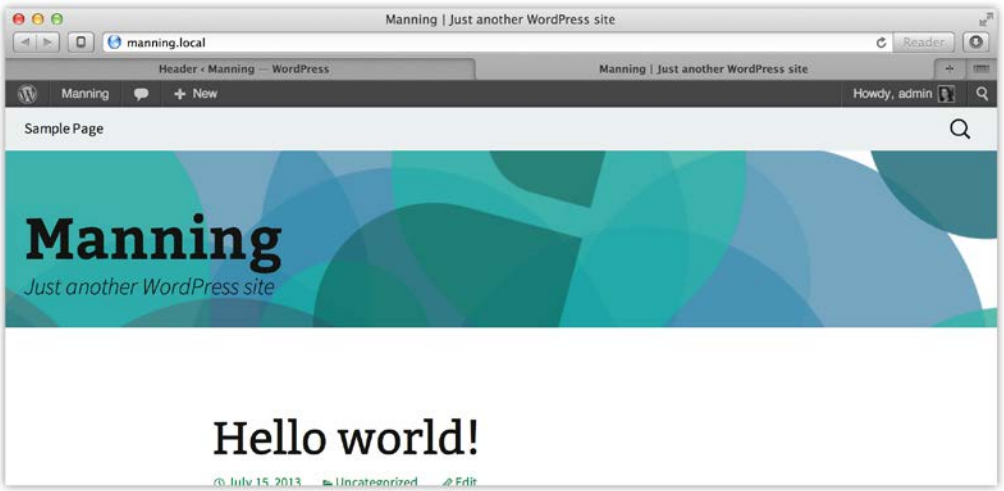


Figure 6.7 It's doubtful as to whether we'd want our site navigation above the header in this theme, but remember that this is just an exercise.

Now that our child theme has a **header.php** file of its own, WordPress will use that to display the header content. Since that is our child theme's *only* template, everything else will fall back to the parent theme.

The exception to the fallback pattern is the **functions.php** file. Since we probably want to leverage all of the goodness our parent theme has imparted to it, WordPress will actually combine the two **functions.php** files of the child and parent themes, and in that order. This means that if the parent theme has a function you want to override with your own code, WordPress will use the child theme's function instead of the parent. Otherwise, every function defined by the parent is available to you in the child.

Pluggable Functions

Some functions are considered "pluggable," that is, able to be overridden by means of declaring the function in the child theme, and then testing for its existence (via the `if ( !function_exists() ) { }` test) in the parent theme. If you're creating a theme that could potentially be used as parent theme, make sure you test for the existence of any theme that you would consider a good candidate to be overridden.

If you find a theme that you love that is responsive, as Twenty Thirteen is, but just want a few adjustments, then using a child theme is a great way to save yourself a lot of work.

One last thing about child themes: there's no such thing as a "grandchild" theme. That is to say, you can't make a child theme from a theme that's already a child theme of some other parent theme.

6.2.2 Theme Frameworks

Theme frameworks are aspecial case of the parent and child theme structure. While any theme, in theory, can serve as a parent theme for the child you want to build, theme frameworks are built expressly for that purpose. Often developed by theme shops as a means of creating a unified system for their products, theme frameworks provide many more bells and whistles than any "ordinary" theme you might be using as a parent. And, just as often, theme frameworks provide little or know styling of their own, relying on the child themes that use them to do all the stylistic heavy lifting.

What theme frameworks ultimately provide is extra features such as custom action and filter hooks, templates with a bevy of different layouts, and possibly expanded options pages in the admin to let you customize more of your theme without having to write code. While many users of theme frameworks laud them for their enhanced capabilities, they can also be difficult to learn if you're just starting out. WordPress at its core has a huge number of action and filter hooks, as well as template tags; when you work with a theme framework you have more of these to learn.

Another bonus of theme frameworks is they usually have excellent support, but that support usually comes with a monetary cost. In fact, most of the most popular theme frameworks out there are commercial. Some, such as StartBox (http://wpstartbox.com) can be downloaded for free, but charge for support. Other theme frameworks, such as Genesis (http://my.studiopress.com/themes/genesis/, figure 6.8) charge to download.

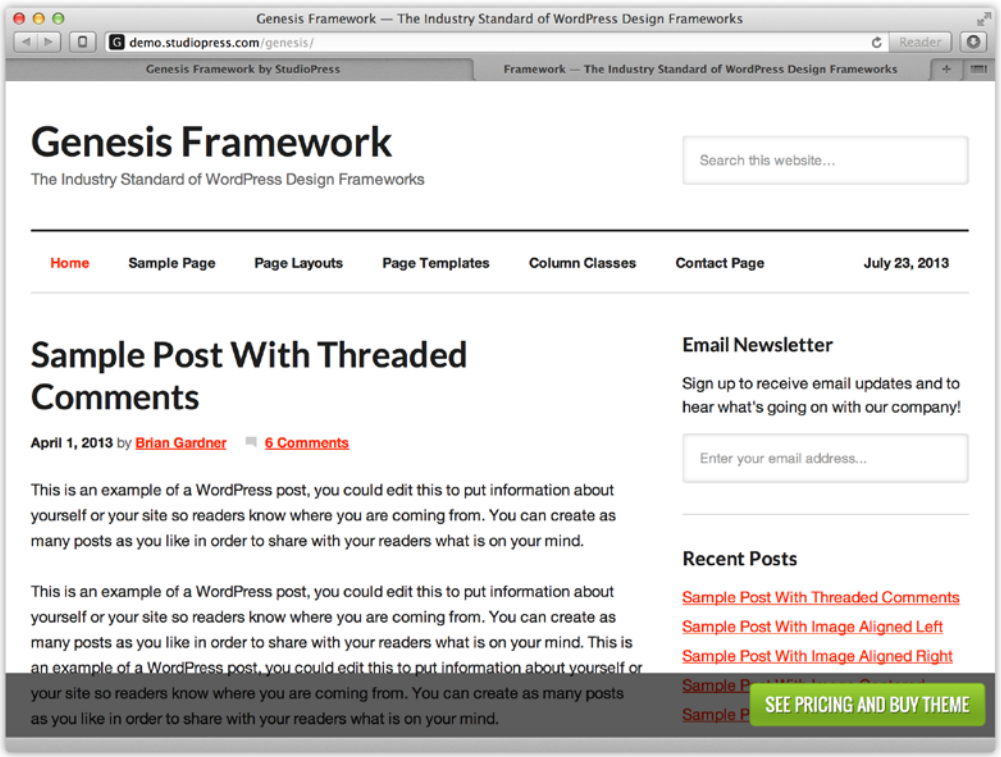


Figure 6.8StudioPress' Genesis is one of the most popular commercial theme frameworks available, and is responsive.

But essentially, these are all parent themes, and the downside is that not all of them are responsive. If you want to use a theme framework that isn't responsive, bear in mind that you'll have to do all the heavy lifting – and a bit of responsive retrofitting – to get your child theme to be responsive.

6.2.3 Starter Themes

The last category in our list of existing theme options is starter themes. These are different than parent themes in the sense that the developers of these themes don't intend that you build child themes off of them, but rather take the original theme files and modify them to your specifications.

I can hear you already. "But what about updates? Won't my changes be destroyed?" When working with a starter theme, all the meta information (the theme name, description, author, etc.) should be changed in the **style.css** file so that it no longer resembles the original theme, at least in WordPress' eyes. That way, there will never be updates to

overwrite your work. And as starter themes tend to have less going on with them in terms of functionality and styling (than, say, thefully-realized parent themes and theme frameworks we discussed above), you won't be missing out on much if you never update your starter theme once you make your changes.

FOUNDATION AND BOOTSTRAP

If you have a favorite CSS grid system or user interface library, something like Foundation by Zurb (http://foundation.zurb.com) or Twitter Bootstrap (http://twitter.github.io/bootstrap), chances are someone's made a WordPress starter theme out of it. This is especially handy if you use these systems outside of WordPress projects, and like to have a consistent grid system or UI library across all of your work. Foundation and Bootstrap, two of the most popular systems today, are also responsive, so that work is done for you. Another bonus of these systems is you can often build your static prototype using the static versions of these projects, allowing you to port your CSS over to your new WordPress theme with minimal changes.

However, one criticism of these two systems in particular is that they're *opinionated*. Opinionated here is a term of art meaning that these systems have a distinct style for elements such as form elements, typography, etc. If you're design calls for a different look and feel than what comes with these projects, you may end up having to strip out a whole lot of CSS in order to get the look you want. This may end up defeating the purpose of using these systems in the first place.

_S (A.K.A. UNDERSCORES)

One of the most popular starter themes in use today is called _s (pronounced "underscores," and found at http://underscores.me), made by the theme folks at Automattic and contributed to by WordPress developers across the Internet. _scalls itself the "1,000 hour head start," and seeks to save us time type by giving us a very basic, stripped down stylesheet (starting with the same Eric Meyer reset that we used in our prototype), and pre-coding useful options such as the custom headers option.

I like _s because of the strong support behind it (it's what the Theme Wranglers at Automattic base all of their projects on) and because of all of the thought that went into creating it. It's got a lot of features without feeling bloated, yet doesn't have a lot of extraneous things that need stripping out or fighting against.

_s, on its own, has extremely little responsiveness built into it. In fact, the only responsive feature of _s is a collapsing primary navigation menu at screen widths up to 600 pixels. But that's okay. Styling and responsiveness are not jobs for the starter theme;they're jobs for us as we take _s and mold it into our own creation.

We'll be using _s as our starter theme for the remainder of this project. But before we set it all up, I want to go over some of the plugins and other tools we'll be working with as we continue on in the theme development process.

6.3 WordPress Development Plugins and Tools

You'd be hard-pressed to find any installation of WordPress anywhere that isn't running at least a few plugins. There are complex plugins, lightweight plugins, and everything in between. There are plugins for Google Analytics, search engine optimization, and building the better slideshow. There are plugins meant expressly for developers, and those that are meant for end users and content providers. At present, there are over 26,000 plugins in the WordPress plugin repository, and that doesn't even count the array of commercial plugins available.

If you develop WordPress for any given length of time, you are bound to come up with a list of your favorites, the go-to plugins you install as soon as your core WordPress instance is up and running. In this section, we'll look at some of the essential plugins I rely on when developing WordPress themes. None of these make it to the final, production level of the site, but rather exist in my development environment to help me debug my theme and make it better.

Installing Plugins

We're going to be installing a bunch of plugins in the coming pages, so let's review how to do so. There are actually many different ways to install a plugin, and everyone has their own preferred technique. I find the following way the most straightforward when developing on my local machine:

Download the desired plugin from the WordPress plugin repository at http://wordpress.org/plugins/. Unzip the plugin and place the expanded directory inside the **wp-content/plugins/** directory of your WordPress instance.

On the admin screen for your WordPress instance in your browser, navigate to the "Plugins" page. If you were already on that page, you'll have to refresh your browser. Find the plugin you just installed, and click the "Activate" link.

If you're a power WordPress user who's comfortable with the command line, and these four steps are too time consuming to want to do over and over (I don't blame you), you might want to check out wp-cli, a command line interface for performing common WordPress tasks such as installing, activating and updating plugins. I wrote an introduction to wp-cli for the Treehouse blog which you can find at http://blog.teamtreehouse.com/tame-wordpress-from-the-command-line-with-wp-cli. I find wp-cli to be an *incredible* time saver, and I sing its praises whenever I get the opportunity.

6.3.1 Show Template

Our first plugin, Show Template (http://wordpress.org/plugins/show-template/) (figure 6.9), is also the simplest and most basic. Once installed and activated, all it does it display a comment near the bottom of our browser's source code that tells us which template the page

is using. That's it. It may not seem like much, but if you're ever coding late at night and not able to figure out why the changes to your template are not displaying on the page, it's extremely useful to be able to check which template WordPress is *actually* using, not just what you *think* it's using.

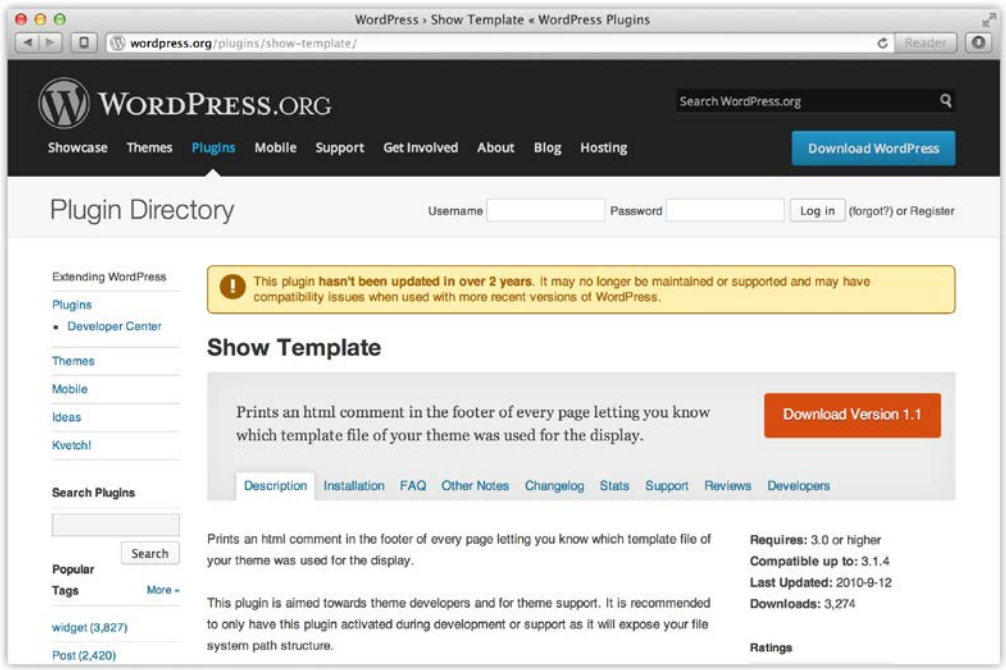


Figure 6.9 Normally, I wouldn't recommend a plugin that hasn't been updated in over two years, but this plugin is so simple and still does the job that I wouldn't worry about it here.

6.3.2 Theme-Check

Eventually, we'll want this theme to find a good home on the Internet. Even if we decide to not submit it to the WordPress theme repository (http://wordpress.org/themes), it's a good idea to make sure that it does all the basic stuff that we want a WordPress theme to do. For that, we need Theme-Check (http://wordpress.org/plugins/theme-check) which will run our theme through the same series of tests that the WordPress.org theme review team uses to evaluate themes for inclusion in the repository. In fact, if your theme *does* have aspirations of submission to the repository, it *must* pass the Theme-Check tests without generating any warnings or required notices (figure 6.10).

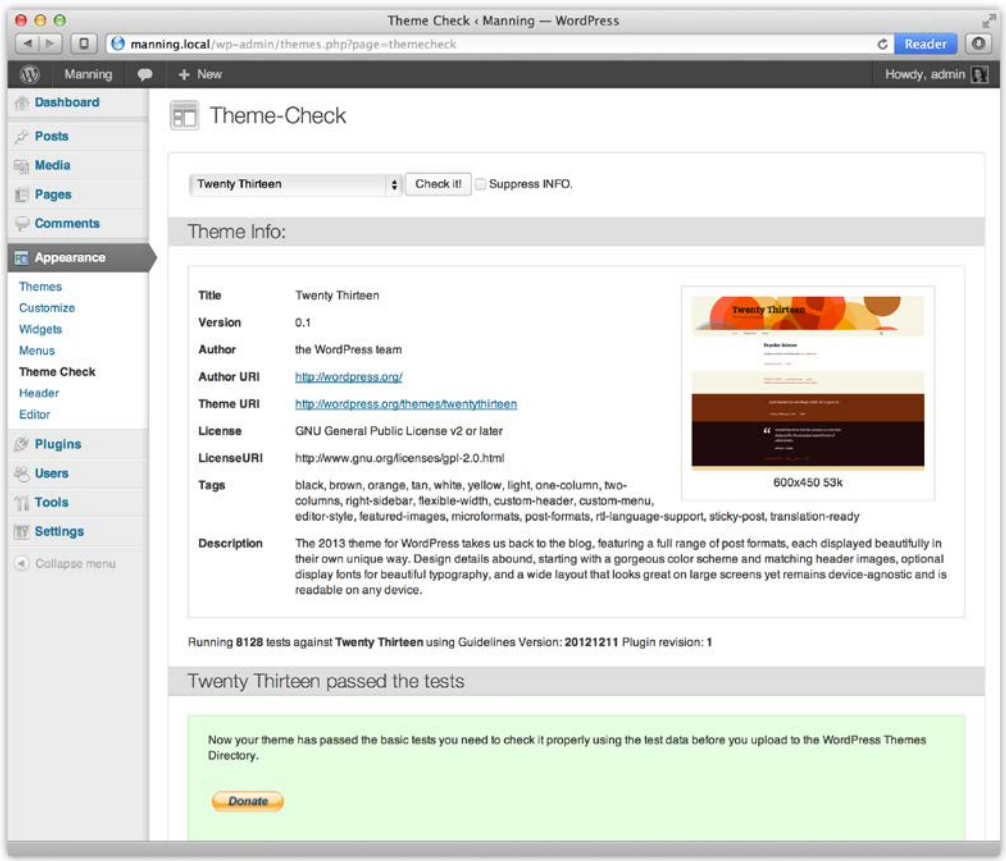


Figure 6.10Twenty Thirteen passes the Theme-Check, which really shouldn't come as any great surprise.

Theme-Check is a good tool for making sure you remember to add in those easy-to-forget classes like `.sticky` and `.bypostauthor`. It will also remind you to remove the "development" stuff out of your theme – things like the **.git** directory and **.gitignore** files, along with our **.sass-cache** hidden directory – before moving this to production.

6.3.3 Debug Bar

Debug bar is a tool that provides diagnostic information on your WordPress site with the click of a button in your administration toolbar (the dark gray bar at the top of your browser when you're logged into WordPress). Once installed, you also need to turn WordPress' native debugging efforts on in order to reap its full benefits. To do so, open up your **wp-config.php** file (usually located in the root directory of your WordPress instance), and set debugging to true, like so:

```
define('WP_DEBUG', true);
```

This line is towards the bottom of the **wp-config.php** file. With that enabled, Debug Bar will now keep you informed on the request being made for the page we're on, how that translates into a query string (when we have some sort of pretty permalinks enabled), the matched rewrite rule (which is going to look indecipherable unless you understand regular expressions), and the resulting query as understood by WordPress. As we get into our theme development (and inevitably introduce typos and another annoying bugs), Debug Bar will be there to let us know where we went wrong. Figure 6.11 shows an example of the kind of data we'll get when we click the "Debug" button:

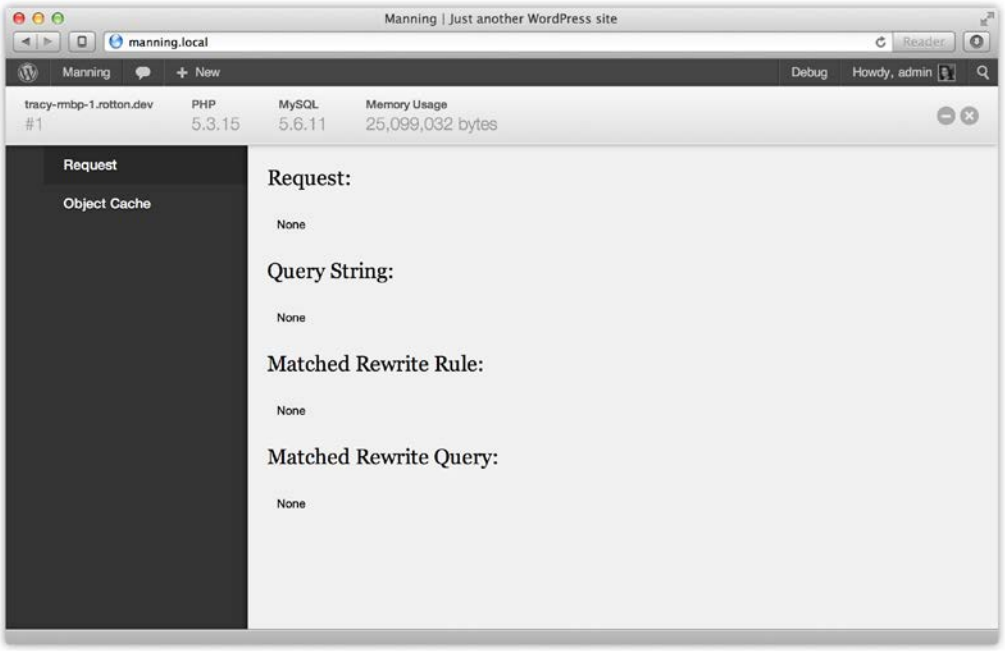


Figure 6.11 There's not a ton of information when your page is behaving itself, but if you have a problem in your code, Debug Bar can help you find it.

6.3.4 Theme Unit Test Data

A blog is nothing without its content. But if we're developing a general-purpose theme for unseen, unknown users, how can we possibly know what kind of content will end up on that blog?We don't. It's impossible to know that. So instead, we need to plan for the widest possible range of content a "typical" blog is likely to see.

Enter the Theme Unit Test found at http://codex.wordpress.org/Theme_Unit_Test. This XML file contains a number of different types of content, from "sticky" posts (posts that are

meant to stick to the top of an archive page, even when newer content is posted), paginated posts, static pages, different post formats, etc.

As with the Theme-Check plugin, the WordPress Theme Review team uses the test data to look for problems with the theme; usually deficiencies where the theme developer didn't account for a certain scenario with the content.

Once you download the unit test file from the URL above, we need to bring it into our test environment, and to do that we need the WordPress importer plugin (http://wordpress.org/plugins/wordpress-importer/). This plugin allows you to migrate the contents of one blog to another. Aside from importing the unit test file, you can use it to migrate from a WordPress.com blog to one that is self-hosted, or transfer content you developed in staging to your production site upon launch.

1. Install the WordPress Importer plugin via your preferred method.
2. Download the Theme Unit Test from WordPress.org.
3. Navigate to the Tools > Import section of the admin. Click "WordPress" from the choices.
4. Choose the theme unit test XML file you downloaded and click "Upload file and import."
5. On the next screen, click the checkbox for "Download and import file attachments," and click "Submit." This will make sure that images associated with the test posts are downloaded and available on your site.
6. Wait.

Once done, your development site will be filled with a whole lot of sample content (but sadly, no place kittens). If you go to your site's home page, you'll see something like figure 6.12:

Figure 6.12There are a whole lot more words on this page than when we started this chapter.

6.3.5 Using CSS Preprocessors with WordPress

In our last chapter, we talked extensively about how CSS preprocessors can save a tremendous amount of time by automating mundane tasks with a combination of nesting, functions, mixins and other benefits. You might think that given the constraints of WordPress theming that these tools would be available to us only during the prototype phase. But good news! We can continue to develop our theme without having to abandon them.

While there are plugins that will process your LESS or Sass for you (see sidebar), I prefer to continue with the same CodeKit(or Scout, or command line, etc.) workflow we used in chapter five. For one thing, if we are having WordPress process the stylesheets through a plugin, it's more work that has to be done by WordPress, and if the Sass files are particularly large or complex, then it could take a while for the CSS to be compiled and load (unless you're using some sort of caching mechanism, which is beyond the scope of this book). Also,

it's not really WordPress' job. WordPress should be all about delivering content to the browser; the styling should be in place when it gets there.

CSS Preprocessor Plugins

Whether your preference is for LESS or Sass, there are WordPress plugins that let you compile your CSS on the fly.

WP-LESS (http://wordpress.org/plugins/wp-less/) uses WordPress' native CSS enqueuing hooks to take ***.less** files and convert them to CSS. The compiled file is created in the **wp-content/uploads** directory, and the stylesheet location is written as an ordinary CSS link in the head of the web page.

For Sass aficionados, WordPress Sass (http://wordpress.org/plugins/wordpress-sass/) adds its own function, `wpsass_define_stylesheet()`which takes the place of the WordPress native function to enqueue your stylesheet in your theme. To make sure that you preserve the theme's identifying comments, it will not let you use **style.scss** or **style.sass** so as not to overwrite the original **style.css** file, so a good alternative would be to name your stylesheet**[theme-name].scss**.

While performance isn't of major concern when we're just building the theme on our development environment, it's still a complication we don't need, since all of the plugins have their own way of doing things. We had a workflow that worked in the prototyping phase; we can continue using it now in the theme development phase. We can compile the CSS with all the Sass debugging our heart desires while we're developing, and then switch over to a fully compressed, minified and comment-less version when we're ready to upload to production.

But uh-oh, there's a gotcha there. If there are no comments in our **style.css** file, then there will be no informational comments describing the theme, and WordPress won't even recognize it as being a theme. Everything will be ruined!

Relax. We can handle this by telling Sass that no matter how crazy minified we want our CSS to be, don't delete *these* particular comments. You can dothat by adding exclamation marks to our CSS comment openers, like in listing 6.6.

Listing 6.6 Force Sass to Preserve Comments

```
/*! #A
No matter how hard you minify this file, this comment will never go away!
#B
*/
```

#A Use the exclamation mark when you really, really want your comments to be there!
#B Anything entered into that comment block will be preserved.

In the next section, we'll set up our starter theme using Sass and Compass compiled outside of the WordPress ecosystem.

6.4 Setting Up Our _s Starter Theme

We'll be using _s as the starter theme for the rest of this project, but since we want to use Sass and Compass for our stylesheets, we'll want a version where our styling has been converted to a Compass project. Currently, _s can be customized to your project's name by going to its website at http://underscores.me, giving it the desired theme name (and any other information you wish to provide), and clicking the "Generate" button. The copy of _s you'll download will have all of the text domain and other theme name-based prefixes (we'll discuss these in a later chapter) already set for you, so you won't have to change them automatically.

Unfortunately, this customized copy of _s *does not* contain the necessary files to use Sass and Compass. While any ordinary CSS file can be converted into an SCSS file by simply changing the file extension from **.css** to **.scss**, I've already gone ahead and created a fork of _s on my GitHub account which not only breaks the native _s **style.css** file into logical Sass partials, but also contains a **config.rb** file for using with Compass.

According to Konstantin Obenland —a Theme Wrangler for Automattic and one of the chief developers of _s— it is the goal to serve up Sass-ready or LESS-ready versions of _s from the customizer at http://underscores.me, but as of now there is no timetable on when that feature will be available.

With all of that in mind, let's set ourselves up with our starter theme now.

1. Download (or clone in Git, if you're like that) my fork of _s from https://github.com/taupecat/_s.
2. Put the expanded directory into your development WordPress instance's theme directory at **wp-content/themes**.
3. Change the name to something unique to your project. For this book, we'll call it Manning, and set its directory name to **manning**.
4. Open the **sass/style.scss** file in your favorite text area so we can edit the theme's meta information (in listing 6.7) to something more reflective of our needs (in listing 6.8).

Listing 6.7 _s Meta Information in style.scss

```
/*! #A
Theme Name: _s #B
Theme URI: http://underscores.me
Author: Automattic
Author URI: http://automattic.com/
Description: Hi. I'm a starter theme called <code>_s</code>, or
<em>underscores</em>, if you like. I'm a theme meant for hacking so don't
use me as a <em>Parent Theme</em>. Instead try turning me into the next,
most awesome, WordPress theme out there. That's what I'm here for.
Version: 1.3-wpcom
License: GNU General Public License v2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html
Tags:
```

```
This theme, like WordPress, is licensed under the GPL.
Use it to make something cool, have fun, and share what you've learned with
others.

 _s is underscoresd on Underscores http://underscores.me/, (C) 2012-2013
Automattic, Inc.

Resetting and rebuilding styles have been helped along thanks to the fine
work of
Eric Meyer http://meyerweb.com/eric/tools/css/reset/index.html
along with Nicolas Gallagher and Jonathan Neal
http://necolas.github.com/normalize.css/
and Blueprint http://www.blueprintcss.org/
*/
```

#A Here's our exclamation mark to preserve our comments.
#B We'll need to change the theme name and other information.

Listing 6.8 Our New "Manning" Meta Information

```
/*!
Theme Name: Manning #A
Theme URI: http://manning.com/rotton/
Author: Tracy F. Rotton
Author URI: http://www.taupecat.com/
Description: A responsive WordPress theme.
Version: 1.0
[…] #B
```

#A Provide your theme's new name and your other information to make this theme your own.
#B We're going to leave all the other information in the opening comments the same.

Since we're using Sass and Compass, we need to do one more thing before we can really get started, and that's fire up a tool to process our stylesheets. I provided a few options in chapter five, but as my preference is for CodeKit, we'll look at that.

1. Launch CodeKit (figure 6.13).
2. Drop the theme's folder (in this case, **manning**, onto the application window). CodeKit will automatically recognize it as a Compass project and set its output paths automatically based on the settings in the theme's **config.rb** file.

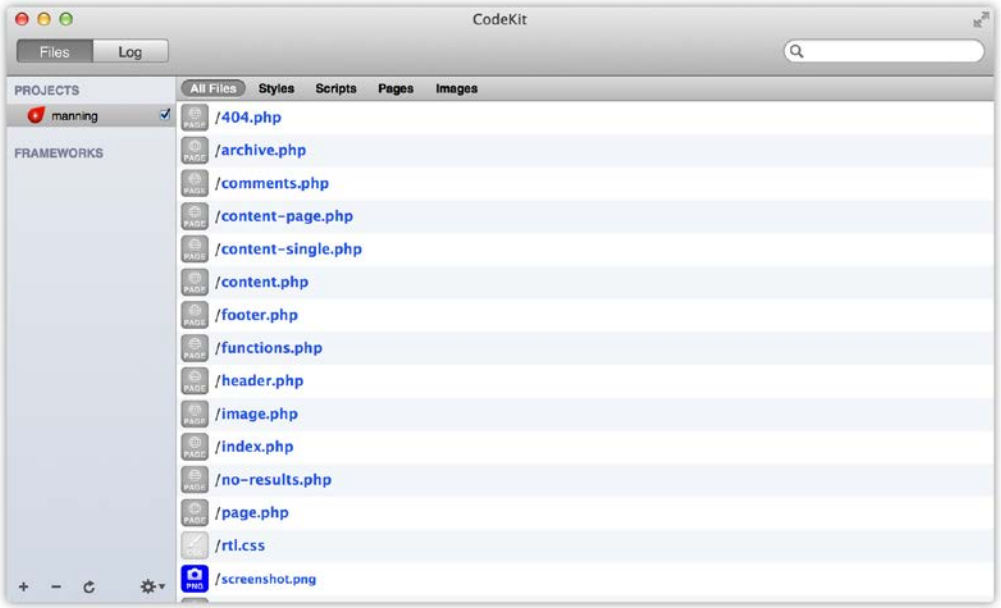


Figure 6.13 Once dropped into our CodeKit window, all of our theme's files are listed, and CodeKit will continuously watch for changes we make.

Remember, we only need to be watching the *theme's* directory, not our entire WordPress instance. Everything of importance happens within the confines of the theme's files, so that's all that CodeKit (or other preprocessing program) needs to be aware of.

Last, but certainly not least, we need to tell WordPress to use our new theme for its front end. Warning, it's not going to look like much right now, but we'll aim to change that in the coming chapters. Navigate to your Theme selection page in the admin (figure 6.14) and activate "Manning" (or whatever you opted to name your theme).

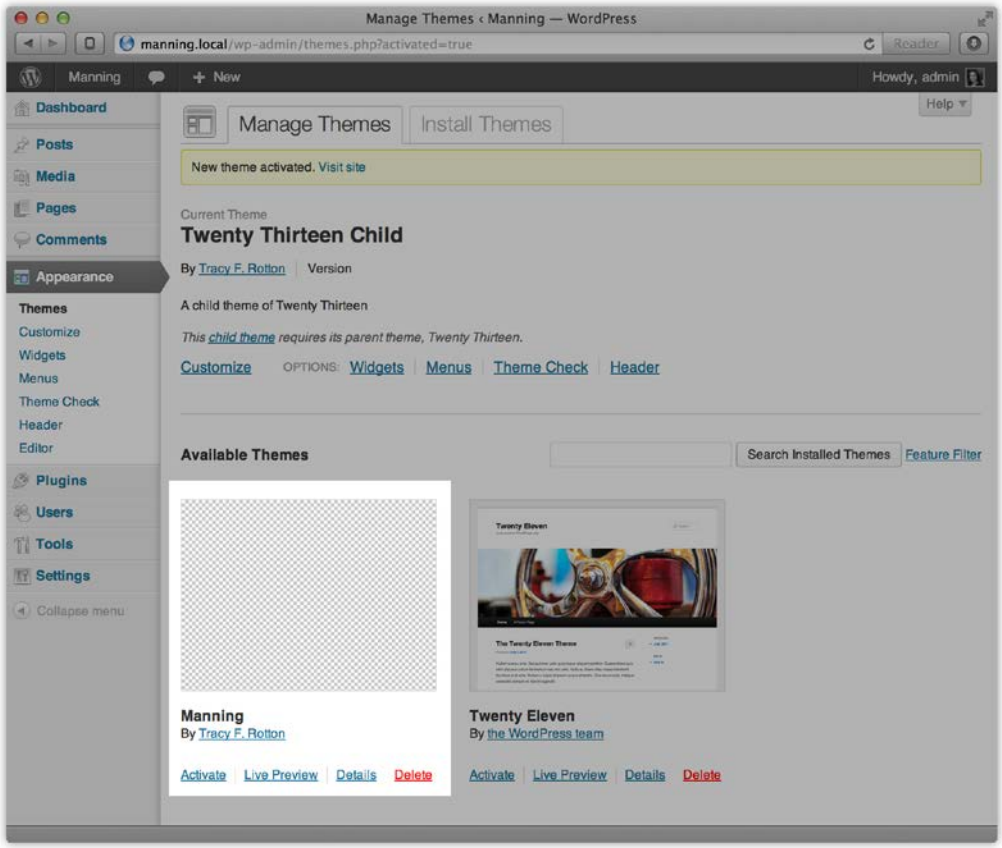


Figure 6.14 Our “Manning” theme is a new option available to us, and its screenshot is the characteristic “transparency grid” that comes with the _s theme.

Once we’ve selected our new theme, go ahead and visit the home page of your site (figure 6.15). You might be underwhelmed, but that’s okay. What’s important is that we’ve established our foundation and are ready to begin construction. Soon.

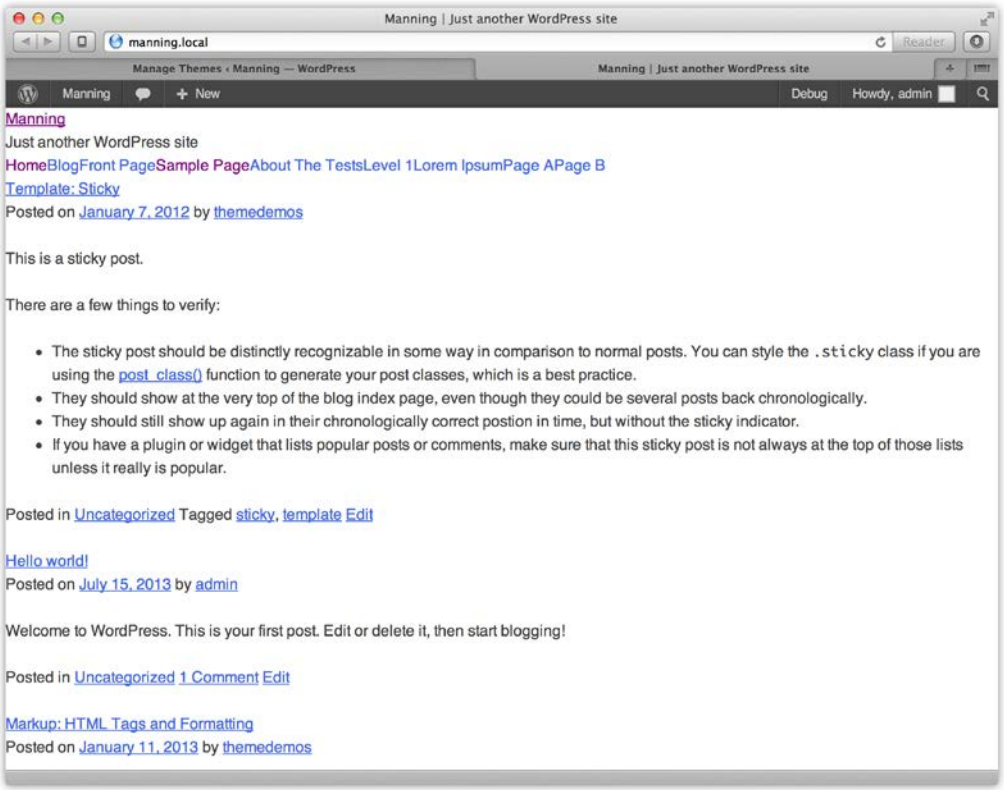


Figure 6.15 All _s gives us is a blank slate, which is all we need.

6.5 Summary

It feels good to finally be getting down to business in WordPress, doesn't it? We've done a lot of prep work in this chapter to establish a development environment that will allow us to create a solid responsive WordPress theme. From choosing the right starter theme, to downloading plugins and other testing and debugging tools, we now have everything to move the prototype we built in the last few chapters into a dynamic, functioning theme.

There are always choices to make when starting out with a new theming project, and there's never a "one size fits all" solution to anything in the WordPress (or web development) world. Become familiar with as many tools as you can, really master the ones that appeal to you most, and you'll be on your way to becoming an expert WordPressthemer.

It's now time to take our static prototype and start the theming process in earnest. Let's get to it!

about the author

Tracy Rotton is a front end web developer and WordPress themer with nearly 20 years experience developing websites. Starting out in the earliest days of the World Wide Web, she has worked for companies such as Marriott International and Discovery Communications. As an interface engineer for vivid studios, she worked on projects for American Century Investments, the Webby Awards and Intel.

Tracy began working with WordPress while setting up a personal blog in 2009. Her first professional WordPress development projects came in 2010 when she developed custom WordPress themes and plugins for the U.S. federal government. She has also developed WordPress themes for the Washington Theological Consortium, the Center for Safe Internet Pharmacies, the FrontPoint Security blog, and Long and Foster real estate (the latter two were responsive WordPress theme projects). In addition to her responsive WordPress projects, she has built responsive websites on other platforms for the Robert Wood Johnson Foundation.

Tracy has spoken on responsive web design for WordPress at both the WordPress D.C. meetup group and WordCamp Baltimore in 2012. She also writes on WordPress and front end development topics for the Treehouse Blog. She is also a minor contributor to the WordPress 3.6 default theme, Twenty Thirteen.