Responsive DESIGN with WordPress

How to make great responsive themes and plugins

Joe Casabona

Responsive DESIGN with WordPress

How to make great responsive themes and plugins

Joe Casabona

Responsive Design with WordPress
Joe Casabona

New Riders
www.newriders.com
To report errors, please send a note to: errata@peachpit.com
New Riders is an imprint of Peachpit, a division of Pearson Education.
Copyright © 2014 by Joseph Casabona

Acquisitions Editor: Michael Nolan
Project Editor: Nancy Peterson
Development Editor: Eric Schumacher-Rasmussen
Copyeditor: Joanne Gosnell
Proofreader: Scout Festa
Technical Reviewer: Stephen N. Mekosh
Production Coordinator: David Van Ness
Compositor: Danielle Foster
Cover Designer: Aren Straiger
Interior Designer: Danielle Foster
Indexer: FireCrystal Communications

ISBN 13: 978-0-321-95741-2
ISBN 10: 0-321-95741-5

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

Dedication

To my parents, Louis and Marie, for their continued support. And to Joe and Jean Rizzi, whose advice, kindness, and patience helped me get to where I am today.

Acknowledgments

I'd like to thank the following people, without whom this book wouldn't be in your hands:

- Stephen Mekosh for not only being a fantastic tech editor and a good friend, but also being the first person to show me both CSS and WordPress.
- Michael Nolan for giving me the chance to write this book and welcoming me to Peachpit Press/New Riders.
- Nancy Peterson and Eric Schumacher-Rasmussen for advice and guidance, editing, keeping me on track, and our weekly talks.
- Joanne Gosnell and Scout Festa for copyediting and proofreading, respectively, and for making it seem like I have a good command over grammar and the English language.
- The design team at Peachpit Press for making this book look amazing.
- Jesse Friedman for the wonderful foreword and kind words.
- Jason Coleman, Stephanie Leary, Lisa Sabin-Wilson, and Pippin Williamson for letting me pick their brains on both WordPress and the art of writing a book.
- My family and friends, especially my parents; my brothers Phil, Mike, and Rob; Dave Redding; Rob McManus; Matt Wren; and my wonderful girlfriend, Erin Holman.

I'd also like to make a quick mention of my brother Phil's website, http://phil.casabona.org. He took the headshot used in this book, and I love his work.

Contents

Foreword

By Jesse Friedman

Today WordPress powers 20% of all websites, and we can look forward to it powering 1 in 4 sites launched in 2014. I launched my first WordPress-powered website in 2005, before all the "cool kids" were doing it. I am very proud to be a veteran developer and strategist for a product used by millions of people all over the world.

Since I launched that first site back in 2005, I have written my own book on developing for WordPress, and I have a few more coming out in spring 2014. I have also contributed to other books and written articles for online publications such as *Smashing Magazine* and *net Magazine*, and I teach both in universities and online. I have also spoken at conferences all over the world, including one where I met Joe Casabona.

I was honored when Joe asked me to write the foreword for this book, because I knew it was going to be great. Joe has a real talent for turning complicated solutions into very simple step-by-step directions. WordPress was built to be simple—simple to set up, simple to install, and simple to extend. Still, it can be somewhat challenging to understand for novice designers and developers who are looking to build on basic WordPress functionality.

These challenges prompted me to write my book *Web Designer's Guide to WordPress: Plan, Theme, Build, Launch* in 2012 and is exactly why Joe wrote his book this year. We are both veteran developers who want to help grow the WordPress community. The best way to do that is to help educate the community and share our experiences and knowledge around a product we use every day. Joe has done just that with *Responsive Design with WordPress*. This is a solid book with lots of great examples.

As a professor at two universities in Rhode Island, I know this book will compliment my class curriculum beautifully. The lessons, examples, and even questions at the end of each chapter help you build a great foundation on WordPress and Responsive Web Design. You also will develop a WordPress theme as you follow along with the book, so you'll be reinforcing the skills you're building as you read.

Not to mention you'll be learning two skills at the same time. You'll be learning WordPress and, at the same time, gaining experience specifically in Responsive Web Design. This approach will not only help to strengthen your skills in both areas but will also make you an expert in a very profitable niche.

As I mentioned earlier, WordPress will power 25% of all websites launched in 2014. This means that 1 in 4 new sites will need a developer who knows WordPress. What's more, as of this year more information is being consumed on mobile devices than on traditional computers. If you didn't have strong skills in Responsive Web Design in 2013, you're definitely going to need them in 2014 and beyond.

In my opinion, there is no better way to learn a skill than by doing it yourself. This book is the best way to learn both WordPress and Responsive Web Design at the same time. Great job, Joe!

Introduction

I got my first portable device when I was a freshman in high school. It was the Palm m100 and I loved it dearly. It didn't do much, and, well, at 13 or 14 I didn't have much to use it for. But having a computer in my pocket? Crazy! As a result, it went everywhere with me (and may have gotten taken away once or twice after I used it in class).

Then I moved to the Compaq iPAQ, which ran Windows and had a color screen. Even crazier, I thought. I could run *real* programs on this. I even asked about campus Wi-Fi when I was visiting colleges in the early 2000s, when it was just becoming popular. I thought of all the amazing things I could do with a tiny computer that came with a pen (stylus) and fit in my pocket. Still, I found myself wanting more after a while. This brings me to my first smartphone: the Palm Treo 650 (**Figure 0.1**).

Figure 0.1 Oh, Treo 650. I still miss you sometimes.

I would do everything on this phone—make calls, take photos, sync my Google Account to it. It even had a primitive browser called Blazer. I could visit websites from my phone!

Since then, of course, the mobile landscape has changed. The iPhone brought a full-featured browser to mobile devices, capable of everything from CSS to JavaScript. It didn't solve one problem, though: the problem of the small screen. That's where Responsive Web Design comes in.

Perhaps you've heard of it. It's apparently pretty popular right now. Lots of people—developers, designers, agencies, and users—are asking about it. And why shouldn't they? On top of catering to what is a quickly growing market, it's pretty cool. Responsive Web Design has become one of those things people check for when they visit a site (resizing a webpage is totally the new "check the source for table layouts").

If you're designing a website, you ultimately have no control over how it's viewed; you don't get to decide where it's viewed or what it's viewed on or the connection on which it's viewed. That might sound scary to some, but to me (and I bet to you, too) it's quite the contrary. I love solving that problem. That's not to say it's not a little daunting. I mean, you need to create a website that is easy to use on mobile but that totally "wows" on the desktop. That's what Responsive Web Design is all about.

WordPress is pretty great too. It powers millions of webpages. Hundreds of millions, even. As you read in the Foreword, it will run 1 of every 4 websites launched in 2014. It does a lot for us while allowing us to do a lot. So how does WordPress fit in with Responsive Web Design? Well, as it turns out, it can be really helpful when creating responsive themes; it has a lot of really great built-in features that we, as developers, can leverage to create better responsive sites. And that's just what I'm going to show you how to do.

Who Is This Book for?

I'd like to tell you that this book is for anyone looking to develop WordPress sites, but in order to get into the real heart of why I wrote this book, I need to make a few assumptions about you, dear reader.

First, I assume you have a working knowledge of HTML, CSS, PHP, JavaScript, and MySQL. I also assume you have some familiarity with WordPress—you've installed it, you use it, you've possibly even coded a theme for it. Finally, I assume you've used a server in some capacity; you should at least know the WordPress directory structure and how to use FTP/SFTP.

So this book is for web developers and WordPress developers who want to take advantage of what WordPress has to offer in order to create great responsive websites. In this book, we are going to cover a wide range of topics and techniques for converting website elements to responsive WordPress theme features.

I will provide a bit of a primer, however. In the first chapter, we will take a closer look at Responsive Web Design: what it is, where it came from, and best practices for using it. Then, there will be a brief overview of WordPress theme development; this will go over some of the major parts of the WordPress theme—important files, the Loop, Custom Post Types, plugins, and more. Then, we'll get into the real fun part.

The real meat of the book—making a responsive WordPress theme—is divided into three parts. Chapter 3 will cover prominent responsive techniques and how to integrate them into the WordPress theme. Chapters 4 and 5 will look at specific components of a WordPress website, including navigation, images, comments, widgets, archives, and plugins.

We will wrap up the book by looking at responsive theme frameworks and child themes in Chapter 6, followed by a cookbook-style section full of tutorials for responsive development in Chapter 7.

Why Did I Write This Book?

When I came up with the idea for this book, there were a lot of things floating through my head. Responsive Web Design is always changing; WordPress is always changing. The best practices of a couple of years ago have changed in both fields, and it's important to get that information out.

There is a big movement in the web development community toward "doing responsive responsibly" (a phrase coined by Scott Jehl); this is the idea that responsive isn't just about screen sizes. There is another big movement in the WordPress community to remove functionality from themes (features such as sliders and Custom Post Types that rely on content). I wanted to create a single place that talks about these things, as a lot of web developers are likely working with both responsive design and WordPress.

Coding Conventions

First of all, any code you come across in the book will be presented in one of two ways. It could be inline, like this: `<?php echo "Hello World!"; ?>`, or it could be in its own block, which looks like this:

```
function hello_world(){
    $s= "Hello World";
    return $s;
}
print hello_world();
```

Either way, you should be able to recognize it pretty quickly. As far as standards, the WordPress Codex lays out quite a few (http://rwdwp.com/16). I will do my best to adhere to these coding standards.

To denote that the code on the page takes place in the middle of a block of code (that is, there was some code skipped between the first line listed and the next), look for an ellipses (`...`).

A couple of things I'd like to point out: I will be using HTML5 markup here, but we won't do anything with the more advanced facets of HTML5, like Web Sockets or Storage APIs.

In most cases, the CSS will use `.classes` instead of `#ids`. This should make for cleaner CSS while eliminating the need for really specific selectors. All of my CSS will be formatted like this:

```
.class-name{
    color: #FFFFFF;
    background: #000000;
    }
```

Notice the use of dashes (-) instead of camel case or underscores, and the fact that the closing bracket is also indented. This makes it easier to read the CSS, especially when there is a lot.

TIP
Look for text like this in the margins for Tips and Notes.

Conversely, my PHP function names will always use underscores (_) and be prefixed with mf_, like this: `mf_get_featured_images()`.

Finally, sometimes the layout limitations of a print publication mean that we have to stretch single lines of code over more than one line. In those cases, you'll see a light gray arrow (→) to indicate that these lines of code should not be broken when you use them. If you've got the digital version of this book, you may find that code breaks unpredictably in the text. In that case, it's important to rely on the downloadable files (www.rwdwp.com) for accuracy.

Other Book Notes

There is a lot of code in the book. Most of the time I will point out where you can find that code. If I don't, all of it is available on the book's website, www.rwdwp.com, as well as on GitHub. You will also find a list of all the short URLs and the sites they point to.

As you code throughout the book, you'll notice that I don't make much mention of testing before Chapter 6; it's important to test on at least a couple of devices, especially if you plan on using these techniques in production-ready sites (and I hope you do).

Finally, I tend to use a lot of acronyms, which are usually defined in context. In case they aren't, here are the most common ones:

- RWD: Responsive Web Design
- WP: WordPress
- RESS: Responsive Design + Server Side Components
- The Codex: the WordPress Codex (or documentation of the API)

Chapter 4

Making Your Theme Responsive: Core Features

So far, you've seen multiple ways to make a website responsive. Now it's time to really dig in and combine those techniques with WordPress' unique capabilities.

In Chapter 3, you learned how to apply your responsive CSS to a WordPress theme. You also got an in-depth look at WordPress' generated CSS classes and created some default styles for what are considered the "essential" ones. From here on out, you'll go even further, focusing on two types of features: core and blog.

This chapter looks at some of the core features of WordPress instrumental to a good User Experience. The next chapters are about using what WordPress provides developers and leveraging those tools to create better responsive sites. That said, you'll look at three areas of development in this chapter: navigation menus, images, and widgets.

Handling Navigation

Responsive navigation techniques are bountiful in books and blog posts; many developers have come up with their own implementations for navigation menus on responsive sites. Brad Frost outlines some of the most popular ones at http://rwdwp.com/32.

In this section, we will look at a few popular techniques and how they look on the Millennium Flights site, and then decide what's best.

A Look at Popular Techniques

If you looked at Brad Frost's post, you'll see that he also wrote a second one and linked to several others. In other words, there are lots of navigation menu patterns to choose from.

We will explore three possible options for the Millennium Flights navigation: Do Nothing (or Top Nav), Jump to (or Footer nav), and the Select box…plus a bonus technique that may surprise you. While we won't be implementing it, we will also look at the Off-Canvas technique, which is a bit more advanced

DO NOTHING (OR TOP NAV)

This is the easiest one to implement. You simply leave the navigation as is at the top of your site; you'll see what the "technique" looks like in **Figure 4.1**.

"Hide and Cry"

Frost mentions one technique in which you hide your navigation from mobile users, naming it "Hide and Cry." This reflects the all-too-prevalent thinking that users on mobile don't need as much functionality as those on the desktop, and it's wrong. 31% of American adults in 2012 used their phones for the majority of their Internet access. In essence, you're punishing mobile users by limiting their experience and making them download content that they'll never see. It might be a convenient way to handle other pages—saving on page height and adjusting the navigation—but it's a bad practice, for some of the reasons discussed in Chapter 1, primarily that you make users download sections of websites they will never see, increasing bandwidth required to download the site and slowing things down.

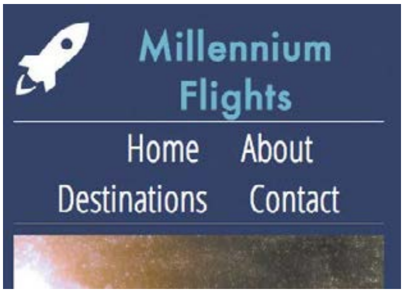


Figure 4.1 Millennium Flights with the "Do Nothing" responsive navigation technique.

As long as your navigation styles are not limited by a specific Media Query, they will apply to the entire site if you employ the Do Nothing technique. It is easy to apply because you don't have to make changes to your code, but there are some pitfalls. Luckily the navigation for Millennium Flights uses big text, but imagine a site with smaller text using this approach. Users may have difficulty selecting the menu items on smaller screens (**Figure 4.2**).

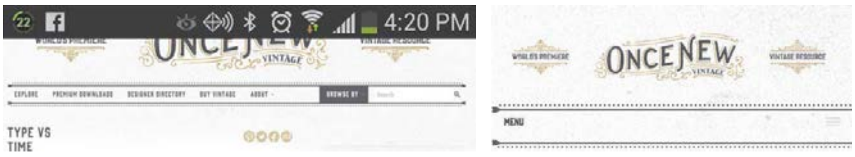


Figure 4.2 On the left is how this website navigation renders on an Android phone. On the right, you see what happens when you resize the browser. In both cases, the nav appears small and difficult to press without zooming.

The other big issue with the technique is that as a user adds more menu items, the height of the page increases. You don't want to have a situation where the user only sees a logo and menu items on page load on a smart phone.

"DO SOME THINGS"

I have a slight modification to this approach, which I've very cleverly called "Do Some Things." It's the idea that functionally, nothing about the navigation changes; it doesn't change locations or implementations. It's still an unordered list at the top of the page. That is, it will stay in the same place and it will be a list of menu items, exactly the same on both layouts. However, you do make slight tweaks and changes to the CSS, optimizing it a little bit for mobile. An example can be seen on my personal site, Casabona.org (**Figure 4.3**).

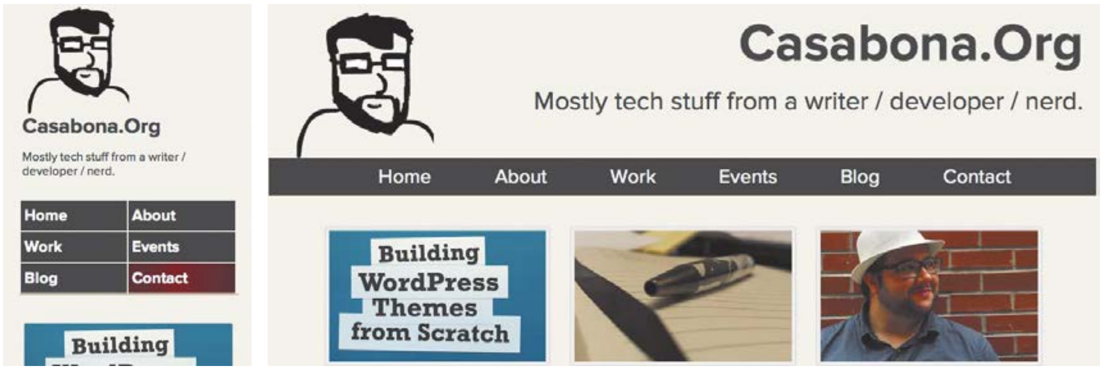

Casabona.Org
Mostly tech stuff from a writer / developer / nerd.
Home
About
Work
Events
Blog
Contact
Building
Casabona.Org
Mostly tech stuff from a writer / developer / nerd.
Home
About
Work
Events
Blog
Contact
Building WordPress Themes from Scratch

Figure 4.3 Casabona.org's navigation on both mobile and desktop layouts. The position of the navigation doesn't change, but the items become more button-like on mobile to make pressing them easier for the user.

NOTE
It should be mentioned that this section isn't going to present a single "right" approach; it all depends on the situation. These are just the options.

Not too much has changed in the overall webpage layout, but you'll see each menu item on smaller screens is its own individual button (as opposed to the contiguous grey bar on wider screens), and the font gets a little bolder on smaller screens.

This slightly modified approach still runs into some of the pitfalls of the "Do Nothing" approach, like taking up a lot of screen real estate on small screens.

JUMP TO OR FOOTER ANCHOR

In this technique, as the screen gets smaller, the navigation is replaced by a single link that jumps the user to navigation in the footer. This, like the "Do Nothing" approach, is easy to implement, but it saves on screen real estate, as the menu is no longer at the top to take up space (**Figure 4.4**).

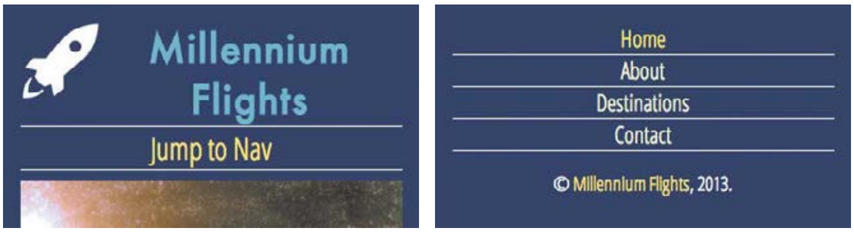


Figure 4.4 The Jump to navigation. The menu items are replaced with a single link that brings users to the bottom of the page.

There are a couple of ways to code this technique. The most common one is to use CSS's `display: none;` to hide the navigations that should not be in use. For example, if this is the header and footer markup:

Header Markup

```
<nav id="main">
<div class="full">
        <?php wp_nav_menu( array('menu' => 'Main' )); ?>
    </div>
    <div class="jump">
        <a href="#footernav">Jump to Nav</a>
    </div>
</nav>
```

Footer Markup

```
<nav id="footernav">
    <?php wp_nav_menu( array('menu' => 'Main' )); ?>
</nav>
```

The CSS would look like this:

```
nav#main .full{ display: none; }

#footernav ul{
    margin: 0;
    padding: 0;
    }

#footernav li{
    font-size: 1.5em;
```

```
        border-bottom: 1px solid #FFFFFF;
        }

    #footernav li a{
        padding: 10px;
        }

    ...
    @media screen and (min-width: 34.188em){

    ...

    nav#main .full{
        display: block;
        }
    #footernav, nav#main .jump{
        display: none;
        }

    ...
    }
```

This code works fine, but it's forcing the user to download code that he isn't going to see. Later in this section, you are going to see how this method can be improved using RESS.

Aside from the extra code, this experience might be a jarring one for the user. The page view would change from the top of the page to the bottom without any scrolling or animations, changing everything the user was looking at; it's not very smooth. A better solution, and one that takes into account the issues we've seen with the techniques we've looked at, would be to convert the navigation to a Select box for mobile.

THE SELECT BOX

The Select box is a clean, user-friendly method that places the navigation at the top of the page without taking up too much screen space. One drawback is that it's not as easy to implement as the previous methods because you actually need to code up two separate menus—one normal menu, and one inside of a select box (**Figure 4.5**).

Adding Smooth Scrolling to the "Jump to" Technique

You can make the transition a bit smoother using a jQuery scroll effect. After adding jQuery, add this function to your header (or functions.php using the wp_head action):

```
$(function() {
  $('a[href*=#]:not([href=#])').click(function() {
    if (location.pathname.replace(/^\//,'') ==
    →this.pathname.replace(/^\//,'')
        || location.hostname == this.hostname) {
      var target = $(this.hash);
      target = target.length ? target : $('[name=' +
      →this.hash.slice(1) +']');
      if (target.length) {
        $('html,body').animate({
          scrollTop: target.offset().top
        }, 1000);
        return false;
      }
    }
  });
});
```

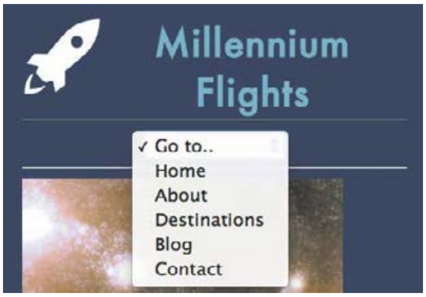


Figure 4.5 The Select box navigation approach clears up a lot of room while still keeping navigation at the top; however, the user still has to download two different navigation menus.

Let's think outside of WordPress for a second, and look at the HTML structure for a select menu navigation:

```
<select onchange="location=this.options[this.selectedIndex].
→value;">
    <option value="" selected="selected">Go to...</option>
    <option value="/home/">Home</option>
    <option value="/about/">About</option>
    <option value="/destinations/">Destinations</option>
    <option value="/contact/">Contact</option>
</select>
```

While this seems like a pretty standard implementation, it's actually going to take a little bit of work to get this properly working in WordPress because it changes the entire structure of the menu. Luckily, WordPress allows developers to do just that with the `Walker_Nav_Menu` class. A "Walker" in programming is a way to traverse, or process, tree-like data structures, which create hierarchical data. In this case, it's traversing HTML to create an unordered list of nav items. You will create a file called Select_Nav_Walker.php, where you will write a class that extends `Walker_Nav_Menu`, using the four functions it uses to display the menu:

```
class Select_Nav_Walker extends Walker_Nav_Menu {

    public function start_lvl(&$output, $depth){}
    public function end_lvl(&$output, $depth){}
    public function start_el(&$output, $item, $depth, $args){}
    public function end_el(&$output, $item, $depth){}
}
```

The functions `start_lvl` and `end_lvl` print the opening and closing elements for the menu; by default this is `<ul>` and `</ul>`. You will actually add the new markup (the `<select>` tags from above) in a different area, so both of those functions will remain as is: blank.

The other two functions, `start_el` and `end_el`, will be used to print the individual menu items. `end_el` is incredibly simple:

```
public function end_el(&$output, $item, $depth){
      $output .= "</option>\n";
    }
```

The function is just one line—the markup that closes the menu item. By default, it is `</li>`, but since this is a select menu, you are changing it to `</option>`. The variable `$output` is what this class continually adds to before sending it back to be printed on the screen. Because of that, it's important to use "`.=`" and not just "`=`". If you don't, your menu will just be `</option>`.

The `start_el` function is a little more complicated than that, and it will make use of the `$item` argument passed to it:

```
public function start_el(&$output, $item, $depth, $args){
    $item->title = esc_attr($item->title);
    parent::start_el(&$output, $item, $depth, $args);
    $output .= '    <option value="'. $item->url .'">'.
    →$item->title;
}
```

The first line escapes the title, converting special characters to HTML entities. Then it calls the parent function (remember this class is extending the built-in `Walker_Nav_Menu` class), which will apply all CSS classes that would otherwise be applied. Finally, we send the `<option>` element to `$output`. `$item` is an array that has several values, including `url` and `title`.

That completes the class, with the entire thing looking like this:

```
class Select_Nav_Walker extends Walker_Nav_Menu {

    public function start_lvl(&$output, $depth){}

    public function end_lvl(&$output, $depth){}

    public function start_el(&$output, $item, $depth, $args){
        $item->title = esc_attr($item->title);
        parent::start_el(&$output, $item, $depth, $args);
        $output .= '    <option value="'. $item->url .'">'.
        →$item->title;
    }

    public function end_el(&$output, $item, $depth){
        $output .= "</option>\n";
    }
}
```

You can either place it directly into functions.php or in its own file, including it in functions.php.

After that, you'll need to attend to the header.php markup, which looks like this:

```
<nav id="main">
<div class="full">
        <?php wp_nav_menu( array('menu' => 'Main' )); ?>
    </div>
    <div class="select-menu">
        <?php
            wp_nav_menu(array(
                'menu' => 'Main',
                'walker'           => new Select_Nav_Walker(),
                'items_wrap'       => '<select ONCHANGE=
                →"location=this.options[this.selectedIndex].
                →value;"><option>Go to..</option>%3$s</select>',
                'container' => false
                )
            );
        ?>
    </div>
</nav>
```

You will show/hide the proper menu the same way you did with the Footer Anchor technique, but we will explore a better option later in this section.

What should be pointed out here are the new entries for the select-menu version of `wp_nav_menu()`. There are three new arguments being passed to it:

- `walker`: The value of the Walker is the instantiation of your `Select_Nav_Walker` class. This tells WordPress to use this walker instead of the default to construct the menu.
- `items_wrap`: By default, this is the `<ul>` wrapper for the normal walker. Since your walker is for a select menu, the appropriate markup should be sent. You'll also notice the `%3$s`. This is necessary to make sure `$output` is included.
- `container`: This by default wraps the entire list in a div, which we don't need to do.

That's everything. Now you should have the regular menu at full width and the select menu for smaller screens. This still has the issue of multiple menus being downloaded, but we will fix that later.

TIP
There are Word-Press plugins that achieve this same effect. It seems the most popular one is this one: http://rwdwp.com/33.

OFF CANVAS

This technique is by far the most advanced one we'll cover, but it's also the slickest. With this method, you'll have your main content on screen and then a button (or even swipe) to reveal more content off to the left or right. This is used by a lot of mobile applications to place the navigation on the left side, hidden away (**Figure 4.6**).

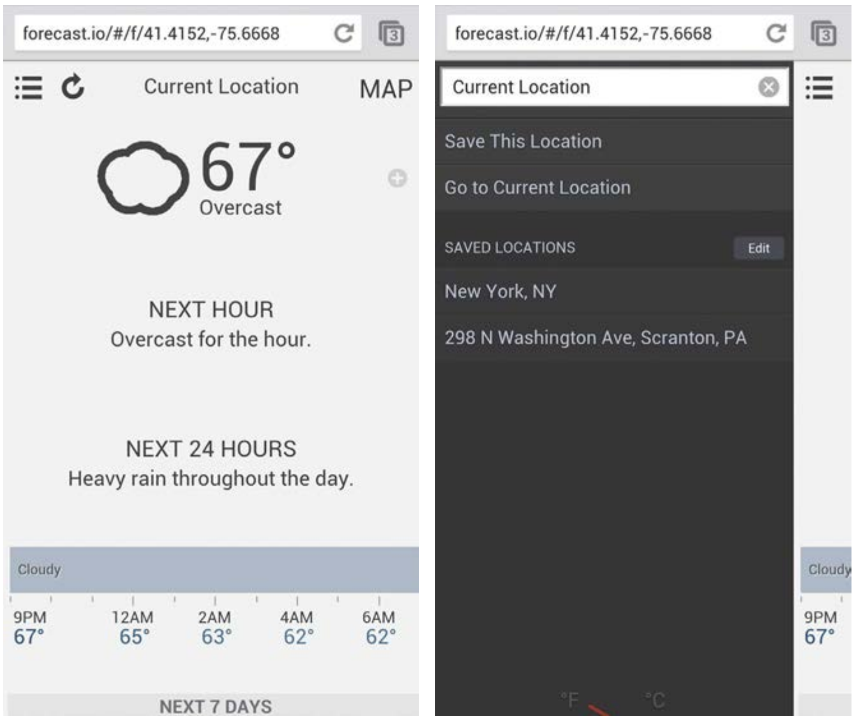
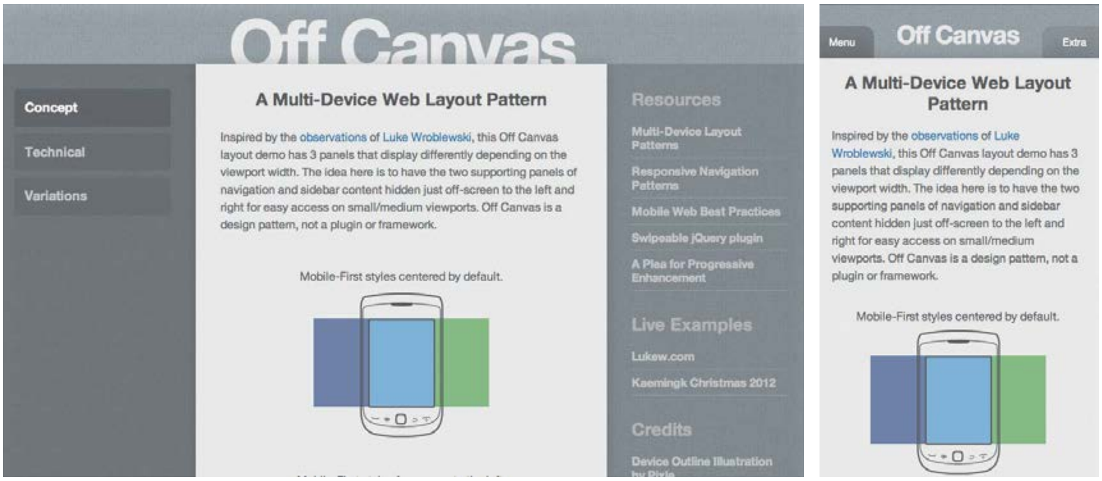


Figure 4.6 Forecast.io, an incredible mobile website, employs the off canvas technique.

This technique completely frees up screen space so the user only sees content. The biggest drawback is the difficulty in implementing it. Depending on how you do it, you could be hiding content, and you will almost definitely use JavaScript (probably jQuery) for the animation effect. It would also, in essence, create an entirely new page section you would have to manage. If you want to see this technique in action or take a crack at it yourself, Jason Weaver and Luke Wroblewski have released code, resources, and examples at http://rwdwp.com/34 (**Figure 4.7**).


Off Canvas
Concept
Technical
Variations
A Multi-Device Web Layout Pattern
Mobile-First styles centered by default.
Resources
Multi-Device Layout Patterns
Responsive Navigation Patterns
Mobile Web Best Practices
Swipeable jQuery plugin
A Plea for Progressive Enhancement
Live Examples
Lukew.com
Kaemingk Christmas 2012
Credits
Device Outline Illustration
Menu
Off Canvas
Extra
A Multi-Device Web Layout Pattern
Mobile-First styles centered by default.

Inspired by the observations of Luke Wroblewski, this Off Canvas layout demo has 3 panels that display differently depending on the viewport width. The idea here is to have the two supporting panels of navigation and sidebar content hidden just off-screen to the left and right for easy access on small/medium viewports. Off Canvas is a design pattern, not a plugin or framework.

Inspired by the observations of Luke Wroblewski, this Off Canvas layout demo has 3 panels that display differently depending on the viewport width. The idea here is to have the two supporting panels of navigation and sidebar content hidden just off-screen to the left and right for easy access on small/medium viewports. Off Canvas is a design pattern, not a plugin or framework.

Figure 4.7 The Off Canvas website has a live demo, sample code, and resources for those interested in trying it out.

This is not the only implementation of the off canvas approach. If menus (or any content) slide down from the top or up from the bottom, it's also considered off canvas, which is really just the viewable area on the screen. Due to its popularity, an open source JavaScript plugin was created to easily implement it. It's called responsive-nav.js. Let's look at implementing that into the Millennium Flights theme.

RESPONSIVE-NAV.JS

In Chapter 3, we saw some questions that should be asked when considering using some piece of JavaScript; questions like, "Do I need this?" and "How large is the file?" Truth be told, we really don't *need* to use responsive-nav.js. However, it is one of the best and most lightweight implementations of the off canvas approach out there. At 1.7 KB, it's considered small by anyone's standard.

Everything you'll need to implement this can be found at http://rwdwp.com/26 (**Figure 4.8**).

RESPONSIVE NAV
Responsive navigation plugin without library dependencies and with fast touch screen support. Try it out by resizing this window.
DOWNLOAD
Prototype

Figure 4.8 The Responsive Nav home page, illustrating how the JavaScript works.

A few things need to happen in order to make responsive-nav.js work in the Millennium Flights theme. First, you'll need to enqueue the script (after downloading the files). Since there is already a function to do that in the theme, you can just add a new line to it:

```
function mf_scripts() {
    wp_enqueue_style( 'googlewebfonts',
    →'http://fonts.googleapis.com/css?family=
    →Open+Sans+Condensed:300' );
    wp_enqueue_script( 'responsivenav',
    →TEMPPATH.'/js/responsive-nav.min.js', array());
    echo '<!--[if lt IE 9]>';
    echo '<script src="http://html5shim.googlecode.com/
    →svn/trunk/html5.js"></script>';
    echo '<script src="'. TEMPPATH .'/js/respond.min.js">
    →</script>';
    echo '![endif]-->';

}
```

Since the script requires the containing div to have a CSS ID, you'll have to modify the nav function slightly. In header.php, change the `wp_nav_menu` function to this:

```
<nav id="main">
<?php wp_nav_menu( array('menu' => 'Main',
→'container_id' => 'top-menu')); ?>
</nav>
```

You're adding one more argument, `container_id`, which will add `id="top-menu"` to the div containing the nav menu. Then it's time to add the CSS.

From the downloaded files, you'll also find a style sheet called responsive-nav.css. You can copy that CSS into your theme's style.css file, but you'll need to make some modifications. This will go before any Media Queries:

```
#top-menu ul {
    margin: 0;
    padding: 0;
    width: 100%;
    display: block;
    list-style: none;
    }

#top-menu li {
    width: 100%;
    }

.js #top-menu {
    clip: rect(0 0 0 0);
    max-height: 0;
    position: absolute;
    display: block;
    overflow: hidden;
    zoom: 1;
    }
```

```
nav.opened {
    max-height: 9999px;
    }
```

This is a slight modification of the default CSS, changed only to match our new menu selector. The same thing goes for the CSS added after the first Media Query:

```
@media screen and (min-width: 34.188em){
    .js #top-menu {
        position: relative;
        }
    .js #top-menu {
        max-height: none;
        }
    #nav-toggle {
        display: none;
        }
...
}
```

Finally, there is a bit of JavaScript that should go before the `</body>` tag. To insert it into the theme, we can use the `wp_footer` action:

```
function mf_responsive_nav(){
echo '<script>
            var navigation = responsiveNav("#top-menu");
        </script>';
}

add_action('wp_footer', 'mf_responsive_nav');
```

This will tell responsive-nav.js to look for the div with the ID top-menu and apply the changes to the menu in that container. Once that is done, you should be able to refresh your page and see the results (**Figure 4.9**).

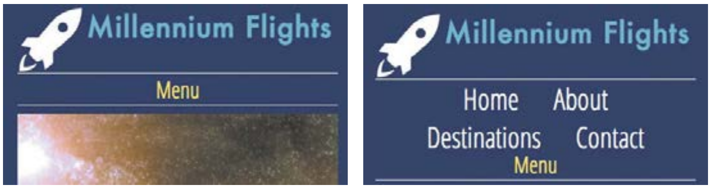


Figure 4.9 Millennium Flights with the default settings for responsive-nav.js. Most of what's seen here is customizable through JavaScript or CSS.

Most of what can be seen in the screenshot is customizable using CSS or the arguments you pass to the `responsiveNav()` function. In the final version of the theme, instead of the text, "Menu," the common menu icon (often called the "hamburger" for some unfortunate reason) is used.

Changing Navigation on the Server Side

Up until this point, the techniques you've been using relied completely on CSS—namely showing and hiding navigation based on breakpoints. However, in Chapter 3, you saw a function that could be especially helpful for navigation: `mf_is_mobile_device()`. Using this, you can choose to conditionally show one version of the site's navigation over another instead of loading both. Let's first look at the select box navigation. Instead of having this:

```
<nav id="main">
<div class="full">
        <?php wp_nav_menu( array('menu' => 'Main' )); ?>
    </div>
    <div class="select-menu">
        <?php
            wp_nav_menu(array(
                'menu' => 'Main',
                'walker'         => new Select_Nav_Walker(),
                'items_wrap'     => '<select ONCHANGE=
                →"location=this.options[this.selectedIndex].
                →value;"><option>Go to..</option>%3$s</select>',
                'container' => false
                )
            );
        ?>
    </div>
</nav>
```

...we can have something much cleaner, and requiring much less CSS:

```
<nav id="main">
<?php
if(!ISMOBILE){
        wp_nav_menu( array('menu' => 'Main', 'container_id' =>
        →'top-menu'));
    }else{
        wp_nav_menu(array(
                'menu' => 'Main',
                'walker'         => new Select_Nav_Walker(),
                'items_wrap'     => '<select ONCHANGE=
                →"location=this.options[this.selectedIndex].
                →value;"><option>Go to..</option>%3$s</select>',
                'container' => false
                )
            );
    }
?>
</nav>
```

Notice there isn't an extra class to hide one menu over the other. It simply says, "If the user is not on a mobile device, print the normal menu. Otherwise, print the select box menu." This means the user isn't downloading two sets of navigation, and you don't have to write extra CSS to hide or show that navigation based on breakpoints.

NOTE
When testing a layout using RESS, you won't be able to just resize your browser window; you will have to do device testing.

The same thing can be done with the Jump to or Footer Anchor approach. In header.php, you'll have:

```
<nav id="main">
<?php
        if(!ISMOBILE){
            wp_nav_menu( array('menu' => 'Main',
            →'container_id' => 'top-menu'));
        }else{
            echo '<a href="#footernav">Jump to Nav</a>';
        }
    ?>
</nav>
```

This is constructed just like the Select box code above, except there is a jump link to the footer. And here is what the footer.php markup looks like:

```
<?php if(ISMOBILE){ ?>
<nav id="footernav">
        <?php wp_nav_menu( array('menu' => 'Main' )); ?>
    </nav>
<?php } ?>
```

TIP
If you are having issues implementing RESS into your theme or need some added functionality, Jesse Friedman's WP Mobile Detect plugin is available in the Plugin Repository http://rwdwp.com/35.

In the footer, you are checking to make sure the user is on a mobile device before printing out the navigation.

Moving forward, we will be able to employ this technique in several other areas of the theme. So with all of these techniques and more, how do you decide which one to use?

Ask What's Best for the Users

As with just about anything in the field of web development (nay, design and development in general), it's best to ask what's best for your users. In many cases, that might be whatever takes up the least amount of screen real estate.

However, maybe your users would understand the navigation better if they saw it all laid out in front of them. Maybe your users don't realize what the "hamburger" icon is. The best thing to do is some research into how your users use your website. Do some A/B testing, or at the very least, get the opinion of friends and family. Ask your users for feedback. As with a lot of things in coding, simple is probably better than clever.

To determine what's best for your users, ask yourself a few questions:

- From what class/type of devices do my users access the site?
- Are my users there to see what's on the home page, or do they usually go to a different page?
- What information do *I* want to present as most important?
- How tech-savvy are my users?

And, as mentioned earlier, give them some options and see what works out best.

Using SVG Images

We won't discuss Scalable Vector Graphics (SVG) extensively here because it's not specific to WordPress, but it is a very good technique to use when it comes to scalable images. SVGs have no set pixel widths or heights associated with them (though there is a default width and height). Because of this, they scale well without losing quality, and even look great on Apple Retina displays.

Using them is as easy as using the `<img>` tag, though there are some support issues, namely for IE 8 and below and Android 2.3 and below. Chris Coyier has a really great tutorial for getting started here: http://rwdwp.com/36. In order to allow SVG uploads in WordPress, you will also need this function (also courtesy of Chris Coyier):

```
function mf_allow_svg( $mimes ){
    $mimes['svg'] = 'image/svg+xml';
    return $mimes;
}
add_filter( 'upload_mimes', 'mf_allow_svg' );
```

Handling Images

Images are one of the biggest sticking points for making a website truly responsive; it's not good enough to just resize them along with the layout or container. There are a lot of tools and techniques out there to aid in better resizing, smaller file sizes, faster loading, and more. What really helps is the way WordPress processes uploaded images.

How WordPress Uploads Images

By default, when you upload an image, the WordPress media uploader creates several different sizes: thumbnail (150x150px max), medium (300x300px max), large (1024x1024px max), and full, which is the original size of the uploaded image. You can also specify different sizes using `the_post_thumbnail()` or `get_the_post_thumbnail()`, but this will only return an `<img>` tag with width and height specified. If you want just an image URL, you can use `wp_get_attachment_image_src()`, which returns an array with URL, width, height, and a Boolean; the Boolean tells you if the image is a resized version (true) or the original (false).

TIP
This is also really useful if you want to create a more diverse image range for the technique we are going to implement in the next section.

You can add new image sizes associated with keywords like "thumbnail," or "medium," for example, which will get resized. That function is `add_image_size()`; if you want to create a feature Destination image for the Millennium Flights CPT, you would use this code:

```
add_image_size('mf_destinations_featured', 650, 300, true);
```

In order, the parameters are: `$name` (which can be used in functions like `the_post_thumbnail()`), `$width`, `$height`, and `$crop`. `$crop` (which is false by default) tells WordPress if it should do a hard crop. A hard crop will crop to the exact dimensions specified, regardless of aspect ratio. If `$crop` is false, it will do a soft or proportional crop. The image's width and height are treated as maximum dimensions.

So when you upload an image, at least three new images (or different sizes) are created. Because of this, the file size is affected; smaller images will have smaller files sizes. We can take advantage of these images by calling them using a lightweight JavaScript library called picturefill.js to show the appropriately sized images based on screen size.

Using picturefill.js

picturefill.js was created by Scott Jehl to mimic functionality for a proposed `<picture>` HTML element that would nicely handle responsive (and even Retina-ready) images. All of the information about it can be found at http://rwdwp.com/23.

To use it, you list several lines in this format:

```
<span data-src="image.jpg" data-media="(min-width: 400px)"></span>
```

The `data-src` is the image source, and the `data-media` is the Media Query at which the image should be used. A full block might look like this example on GitHub:

```
<span data-picture data-alt="A giant stone face at The Bayon
temple in Angkor Thom, Cambodia">
<span data-src="small.jpg"></span>
<span data-src="medium.jpg"
→data-media="(min-width: 400px)"></span>
<span data-src="large.jpg"
→data-media="(min-width: 800px)"></span>
```

```
<span data-src="extralarge.jpg"
→data-media="(min-width: 1000px)"></span>

        <!-- Fallback content for non-JS browsers. Same img src
        →as the initial, unqualified source element. -->
<noscript>
<img src="external/imgs/small.jpg" alt="A giant stone face at
→The Bayon temple in Angkor Thom, Cambodia">
</noscript>
</span>
```

The first one is assumed to be for the smallest screens, and you can have as many entries/Media Queries as you like. The GitHub page talks about various uses before the basic example here, but this will serve us well.

As you might imagine, we can use this script along with the multiple image sizes produced by WordPress' Media Manager to automatically generate a picturefill object that can be called in your themes:

```
function mf_get_featured_image($html, $aid=false){
    $sizes= array('thumbnail', 'medium', 'large', 'full');

    $img= '<span data-picture data-alt="'.get_the_title().'">';
    $ct= 0;
    $aid= (!$aid) ? get_post_thumbnail_id() : $aid;

    foreach($sizes as $size){
        $url= wp_get_attachment_image_src($aid, $size);

        $width= ($ct < sizeof($sizes)-1) ? ($url[1]*0.66) :
        →($width/0.66)+25;

        $img.= '
            <span data-src="'. $url[0] .'"';
        $img.= ($ct > 0) ? ' data-media="(min-width: '.
        →$width .'px)"></span>' :'></span>';

        $ct++;
    }
```

```
        $url= wp_get_attachment_image_src( $aid, $sizes[1]);
        $img.=  '<noscript>
                    <img src="'.$url[0] .'" alt="'.
                    →get_the_title().'">
                </noscript>
            </span>';
        return $img;
    }
```

There are a few things going on here. The first is that the function has an array of all the default sizes in WordPress. If you have your own sizes defined, you will have to add them here. This is so the picturefill element is accurately populated. After some setup (defining the image sizes, opening the picturefill element, initializing a counter), it moves through the `$sizes`, printing an image entry for each size.

For each entry, `wp_get_attachment_image_src()` is called to grab the URL of the image based on the image's ID (which `get_post_thumbnail_id()` returns based on the post ID) and the size. `wp_get_attachement_image_src()` returns an array that includes the image, the width, the height, and whether or not it's cropped. The first time through the Loop, we don't need to specify a minimum width, since that image will be a default. That's where the counter comes in. For the rest of the iterations, the width is calculated using a simple formula and an assumption. Let's look at that line again:

```
$width= ($ct < sizeof($sizes)-1) ? ($url[1]*0.66) :
→($width/0.66)+25;
```

What's happening here in most cases is that we start showing the next image size up at 66% the width of the image; so if the image is 1000px, it will start being shown at 660px. However, if it is the last image in the array, the assumption is that this is the biggest image (the image at full width). There are some strange results returned in some cases with this image, so you can't rely on the width and height returned with the full width image. We simply take the previous image's width and add 25px to it.

The last thing this function does before returning the picturefill code is set a default image in case JavaScript is disabled. The medium image is the default.

Since this plugin requires picturefill, one more task needs to be performed, and that's to actually add picturefill.js to the rest of the JavaScript loaded on the site. Looking at the `mf_scripts()` we've used throughout the book, you'll notice that the line `wp_enqueue_script( 'picturefill', TEMPPATH.'/js/picturefill.js', array());` has already been added.

If you'd rather continue to use `the_post_thumbnail()` instead of a new function, or if you want this to apply to all features images/instances of `the_post_thumbnail()`, you can easily do that by adding this filter to your functions.php file:

```
add_filter( 'post_thumbnail_html', 'mf_get_featured_image');
```

It's important to note that this function will not automatically run for all images on pages and blog posts; this is strictly for getting featured post images. To replace all post images would require content filters, as well as some regex magic to replace the `<img>` tag with the picturefill script.

NOTE
To support featured images, you will need to add `add_theme_support( 'post-thumbnails' );` to your functions.php file.

There is a plugin available that will replace content images with picturefill, located at http://rwdwp.com/37. Based on my testing, it works fairly well, but you may see performance issues. That said, this might be your best bet as what I was experimenting with returned worse performance than the plugin.

Moving forward, you can also use a shortcode, along with the above function. I didn't touch on the arguments the function accepts, but the first is `$html`; this is the HTML send by the `post_thumbnail_html` filter. The second is `$aid`, for attachment ID. This will allow you to call the function on any image you want, not just featured ones.

The shortcode you're going to create is `[mf_image src='path/to/image']`. This is a shortcode that will accept a URL for an argument and print out the picturefill markup for that image. The function that does the heavy lifting is a modified version of one by wpmu.org (http://rwdwp.com/38) and is listed below, but first let's look at the function used for the actual shortcode:

```
function mf_responsive_image($atts, $content=null){
    extract( shortcode_atts( array(
        'src' => false
    ), $atts ) );

    if(!$src){
        return '';
    }else{
        $aid= mf_get_attachment_id_from_src($src);
        $img= mf_get_featured_image('', $aid);
    }
```

```
        return $img;

    }

    add_shortcode('mf_image', 'mf_responsive_image');
```

The function will check to make sure a URL is passed, then grab the ID for that URL (that's where wpmu.org's function comes in) before passing that ID off to the `mf_get_featured_image` function. It will then return the HTML generated. Here is the function that grabs the attachment ID based on the URL:

```
    function mf_get_attachment_id_from_src($url) {
        global $wpdb;
        $prefix = $wpdb->prefix;
        $attachment = $wpdb->get_col($wpdb->prepare("SELECT ID
        →FROM " . $prefix . "posts" . " WHERE guid='%s';", $url ));
        return $attachment[0];
    }
```

NOTE Unfortunately, the shortcode means you won't be able to use the media uploader to insert images into posts.

Between the shortcode and the featured image function, you have two good methods for using picturefill and responsive images moving forward. Hopefully an efficient way to do all images will emerge in the near future. In the mean-time, there is another method that can be used to make images a bit more responsive-friendly.

Overriding Set Width and Height

This is a technique that Jesse Friedman put forth in his book, *Web Designer's Guide to WordPress*. jQuery would allow us to search for all `<img>` tags in the content and remove the set width and height attributes applied to images. This will, at the very least, ensure that the images resize properly:

```
    $(function(){
        $(".post img).removeAttr("width").removeAttribute("height");
    }
```

You can add this function to your header (or even better, using add_action) and enqueue jQuery, and you're all set.

CSS may also be used for the technique, though results may vary based on custom posts, images, and other CSS rules; because of that, this code may need to be tweaked:

```
img[class*="align"],
img[class*="wp-image-"] {
  height: auto; }

img.size-full {
  width: auto; }
```

Handling Widgets

Sidebars and widgets are a staple of many sites, including those powered by WordPress. The WordPress admin makes it incredibly easy for end users to manage their own sidebars; however, it's up to you to ensure that the sidebars and widgets don't break when it comes to responsive design. First, let's look at responsive sidebars.

The Sidebar

It's worth noting right off the bat that using the term *sidebar* (at least in this book) doesn't necessarily mean the physical location of this content. The sidebar can be any auxiliary content. It just so happens that the main example of our sidebar is on the right side.

TIP
Proper page structure also helps with search engine optimization. By placing the main content first, it tells search engines, "This is more important."

Responsive sidebar development starts right at the beginning of the coding process, in determining where in the markup the sidebar will go. It's important to structure and stack each column properly, especially when taking a Mobile First approach to development; if we don't, it might be the case that on small screens, the sidebar shows first, and then the main content area.

The way content should be structured or stacked is Header > Main Content > Sidebars > Footer (**Figure 4.10**).

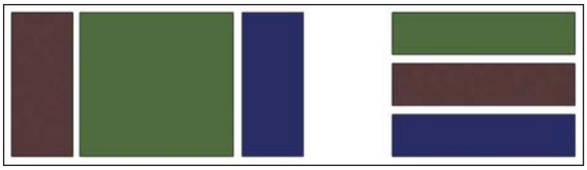


Figure 4.10
Here is the Content stack, illustrated for a three-column layout.

The Millennium Flights site actually only has a right sidebar, so the structure of things in the HTML will look like this:

```
<div id="content" class="group">
    <div class="entry">
        <!--Content Goes Here-->
    </div>
    <aside class="sidebar">
        <!--Widgets Go Here-->
    </aside>
</div>
```

Notice that the div with the id “content” serves as a container for both the main content area and the sidebar. The div “entry” will be for the site’s main content, and the aside element will house the sidebar.

Since the sections are in the proper order, there is actually no CSS required for smaller screens, layout wise. Any pre–Media Query styles will be strictly for some padding, font adjustments, image aligning, and so on. However, once the screen reaches a certain width, it’s time to move the entire sidebar up to the right, next to the main content area. The CSS looks like this:

```
@media screen and (min-width: 40.625em){
    #wrapper #content .entry{
        float: left;
        width: 66%;
        }

    #wrapper #content aside{
        margin-left: 68%;
        width: 30%;
        }
```

If you’ve worked with non-fluid (or fixed) layouts, this should look familiar to you. The main content area, `.entry`, is being floated to the left and limited to a width of 66%. The sidebar is then given a slightly larger margin and a width that accounts for the extra spacing. As users expand out even more, we may want the sidebar and content area to adjust further so the content and sidebar look proportionally better compared to the width of the screen:

```
@media screen and (min-width: 55.652em){
    #wrapper #content .entry{
        float: left;
        width: 75%;
        }

    #wrapper #content aside{
        margin-left: 78%;
        width: 22%;
        }
}
```

As you develop, it's important to test and try different layouts along the way. Remember—the breakpoints are based on content, not device! Do what's best for your own content.

That's all we need as far as structural CSS goes for the sidebar. Of course, now there is the question of handling widgets...

The .group Class

In order to ensure there is no runaway floating content without using the common .clearfix class, the .group class can be used. This technique was put forth by Dan Cederholm as a better alternative that cuts down on markup. The CSS looks like this:

```
.group:after {
    content: ".";
    display: block;
    height: 0;
    clear: both;
    visibility: hidden;
    }
```

This will make the containing class "self-clearing" and alleviate the need for an empty container. It's worth noting that the :after pseudo-class does not work in IE7 and below.

Sidebar Widgets

A widget in WordPress is defined as adding features and content to sidebars. Generally speaking, they are boxes placed outside the main content area that can contain any kind of content—text, images, forms, and so on. As designers and developers, we should plan to support these elements and more, as there are hundreds (perhaps thousands) of widgets out there.

TIP
When developing your own plugins or widgets for release, use the least amount of CSS possible and make it easily available to theme editors who might want to change it.

First, you must properly register your sidebars to use containers and class names you assign. Here's what's added in the Millennium Flights theme:

```
register_sidebar( array (
    'name' => __( 'Sidebar', 'main-sidebar' ),
    'id' => 'primary-widget-area',
    'description' => __( 'The primary widget area', 'wpbp' ),
    'before_widget' => '<div class="widget">',
    'after_widget' => "</div>",
    'before_title' => '<h3 class="widget-title">',
    'after_title' => '</h3>',
) );

register_sidebar( array (
    'name' => __( 'Sidebar2', 'secondary-sidebar' ),
    'id' => 'secondary-widget-area',
    'description' => __( 'The secondary widget area', 'wpbp' ),
    'before_widget' => '<div class="widget">',
    'after_widget' => "</div>",
    'before_title' => '<h3 class="widget-title">',
    'after_title' => '</h3>',
) );
```

You can see that the same structure is used for both sidebars. Each widget gets the class "widget," and the titles are `<h3>` tags with the class "widget-title." This will allow you to apply general styles to HTML elements without them being overridden by other plugins.

In the Millennium Flights theme, before any Media Queries, there are some basic styles applied to elements you might find in a sidebar widget. These will also optimize certain elements (like form fields) for touchscreen devices.

```
aside .widget h3{
    margin-top: 0px;
    color: #8E6DD7;
    }

aside .widget{
    border-bottom: 2px solid #FFFFFF;
    margin: 10px 1.5%;
    text-align: left;
    padding: 5px 0.4%;
    }

aside .widget p, aside .widget ul,
aside .widget ol, aside .widget dl{
    margin: 0;
    padding: 0;
    font-size: 1.2em;
    }

aside .widget li{
    padding: 0;
    margin: 0 0 0 1em;
    list-style-type: none;
    }

aside .widget input, aside .widget select{
    margin: 1%;
    padding: 7px;
    font-size: 1.6em;
    width: 100%;
    border: 1px solid #CFCFCF;
    -moz-border-radius: 10px;
    -webkit-border-radius: 10px;
    -o-border-radius: 10px;
    border-radius: 10px;
    }
```

Then, as users move to larger screens, only slight adjustments are needed:

```
@media screen and (min-width: 40.625em){
aside .widget input[type=submit], aside .widget
input[type=button]{
        margin: 0 auto;
        padding: 3px;
        font-size: 1.25em;
        width: 40%;
        }

    footer aside .widget{
        width: 30%;
        margin: 10px 1.2%;
        float: left;
        text-align: left;
        }
}
```

As a matter of fact, very little needs to be done here. There is a small adjustment for submits and buttons, and some styles for the widgets found in the footer, which should be placed three per line. However, there is a small issue with this approach.

I ran into a situation where a breakpoint was already added to a previous layout I was using, so I kept it in. If you look at the page with a screen width of 40em (650px), the widgets are a bit scrunched (**Figure 4.11**).

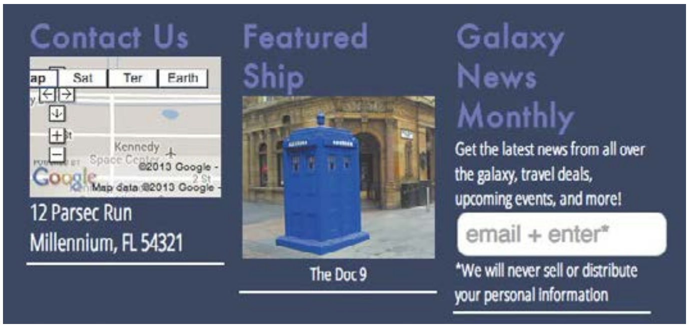
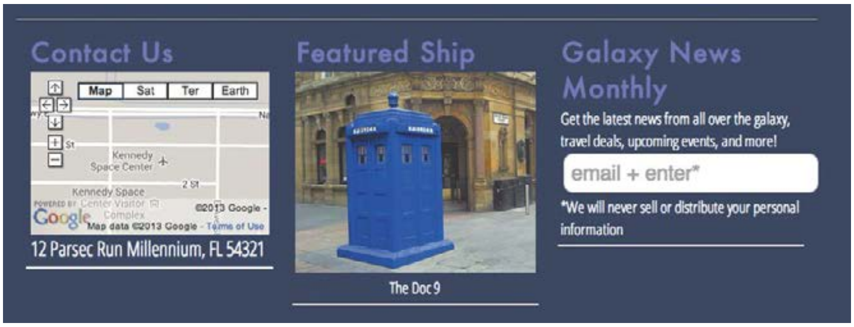

Contact Us
12 Parsec Run
Millennium, FL 54321
Featured Ship
The Doc 9
Galaxy News Monthly
Get the latest news from all over the galaxy, travel deals, upcoming events, and more!
email + enter*
*We will never sell or distribute your personal information

Figure 4.11 The footer widgets at 650px are a bit too scrunched to be three per line.

Because of this, I needed to determine new breakpoints. Here's what the new adjustments look like in the CSS:

```
@media screen and (min-width: 34.188em){
    ...
    footer aside .widget{
        width: 46%;
        margin: 10px 1.2%;
        float: left;
        text-align: left;
        }
}
@media screen and (min-width: 51em){
    footer aside .widget{
        width: 30%;
        }
}
```

I created two new breakpoints—one for a two-column widget layout and one for a three-column layout. Things look much better with the 3-column layout now (**Figure 4.12**).



Get the latest news from all over the galaxy, travel deals, upcoming events, and more!

*We will never sell or distribute your personal information

Figure 4.12 With the adjusted breakpoints, the widgets look much better.

As you continue to add widgets to your theme, it's important to consider the following:

- Is the widget properly coded? Will it use the HTML structure you defined when registering the sidebars?
- Does it come with its own complicated CSS? If it does, overriding styles (especially if they are inline) might be a pain in the neck due to the need to nest classes and be incredibly specific with your style selectors.
- Does the author make the class names intuitive and easy to find (through documentation)? If so, you'll have an easier time styling for them.

If you properly plan and code consistently, you shouldn't run into issues with your widgets. Between the reset, default element styles, and default widget styles discussed in this chapter, most bases should be covered.

Wrapping Up

We explored quite a bit in this chapter, from several navigation techniques using Media Queries, RESS, and JavaScript to making our images respond to both screen resolutions and connection speeds, as well as making the most out of our sidebar widgets.

Images proved to be a tough task to tackle, but we can now replace featured images automatically and post images with a shortcode. While images still remain a bit of a question mark as far as the best way to replace in-content images on a large scale due to some possible performance issues, what we looked at will at least help you and your users moving forward. Trying to fix all images at load time puts a lot of stress on the website that results in it being visibly slower. Hopefully we will see this improved upon in the near future.

Since this chapter focused primarily on layout elements, in the next chapter we will go back to WordPress' roots and look at making elements of the blog portion of a site responsive. Specifically, we'll look at comments, archives, and other developers' plugins.

Questions

1. What is the biggest pitfall of the "Do Nothing" navigation approach?
2. When should RESS be employed in regard to navigation?
3. What does picturefill.js do?
4. How do you ensure content integrity as far as multicolumn layouts are concerned?

Answers

1. It will take up a lot of vertical space, pushing the content down the page.
2. RESS should be employed when we are hiding navigation/markup based on screen width.
3. It replaces a single `<img>` with multiple image options, to be displayed based on screen width.
4. Structure the columns so the main content is on top, followed by the sidebars.

This page intentionally left blank

Index