



Community Experience Distilled

Mobile First Design with HTML5 and CSS3

Roll out rock-solid, responsive, mobile first designs quickly and reliably

Jason Gonzales

[PACKT]
PUBLISHING

www.it-ebooks.info

Mobile First Design with HTML5 and CSS3

Roll out rock-solid, responsive, mobile first designs
quickly and reliably

Jason Gonzales

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Mobile First Design with HTML5 and CSS3

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1170913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-646-3

www.packtpub.com

Cover Image by Arvind Shetty (arvindshetty86@gmail.com)

Credits

Author

Jason Gonzales

Project Coordinator

Suraj Bist

Reviewers

Ahmad Alrousan

Daniel Blair

Martin Brennan

Proofreader

Stephen Copestake

Indexer

Rekha Nair

Acquisition Editor

Martin Bell

Owen Roberts

Production Coordinator

Aparna Bhagat

Commissioning Editor

Neil Alexander

Cover Work

Aparna Bhagat

Technical Editors

Chandni Maishery

Larissa Pinto

Copy Editors

Tanvi Gaitonde

Sayanee Mukherjee

Alfida Paiva

About the Author

Jason Gonzales has worked as a musician and an English teacher, but front-end engineering is his passion. He is a self-taught engineer, but is an obsessive learner and researcher. He's been working on front ends for over seven years, but also does full-stack work and lots of fretting over making sites that have awesome user experiences. This keeps him learning pretty much on a daily basis, which is how he likes it.

I'd like to thank my wife, kids, and friends for putting up with me while working on this book. I'd also like to thank Bear Republic Racer 5, coffee, and vim.

About the Reviewers

Ahmad Alrousan has been a professional developer for over seven years, specializing in building desktop, web, and mobile business applications for different industries.

He holds a bachelor's degree in Computer Engineering and he is a .NET Microsoft Certified Professional Developer (MCPD).

He spends a lot of time working on startups and learning new skills. He can be reached at <http://alrosan.net>.

Daniel Blair studied web development at Red River College in Canada. He is an independent web and mobile application developer. He specializes in Android where he has written several successful apps that do a wide range of tasks.

Dan also enjoys working with WordPress and regularly develops custom themes for clients that are both responsive and beautiful. A Linux enthusiast at heart, he often works with the Ubuntu desktop and server operating system and enjoys working with Linux compatibility issues.

Dan also runs a technology website that offers tutorials, reviews, and downloads. He also regularly blogs about the current open source, Linux, Android, and operating system news.

Martin Brennan is a web developer working in Brisbane, Australia who develops primarily in the ASP.NET platform and has been doing so for the past three years. He works regularly with ASP.NET, VB.NET, C#, JavaScript, and MSSQL, and loves to work with JavaScript MV* frameworks. He spends his spare time learning new programming languages and frameworks and blogging at <http://www.martin-brennan.com>. Martin also enjoys reading, obsessively organizing his music collection, and blogging about liquor and bars with his best friend at <http://www.imbibeblog.com>.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On-demand and accessible via web browsers

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

| | |
|---|-----------|
| Preface | 1 |
| Chapter 1: Mobile First – How and Why? | 5 |
| What is Responsive Web Design? | 5 |
| Prerequisites | 10 |
| Andy Clarke's site | 10 |
| GitHub | 10 |
| My GitHub Fork | 10 |
| Summary | 10 |
| Chapter 2: Building the Home Page | 11 |
| Preparing and planning your workspace | 11 |
| Planning ahead | 12 |
| Navigation | 14 |
| Hero/slider | 14 |
| Content panels | 14 |
| Footer | 15 |
| Let's build! | 19 |
| Header | 20 |
| Logo | 20 |
| Navigation | 21 |
| Hero | 29 |
| Content panels | 31 |
| Footer | 32 |
| Making our page responsive | 35 |
| Slider | 38 |
| Summary | 43 |
| Chapter 3: Building the Gallery Page | 45 |
| Creating the wireframe | 45 |
| The slim hero | 48 |
| Content panels | 54 |

| | |
|---|------------|
| The gallery detail | 57 |
| The back link | 64 |
| The gallery item JavaScript | 66 |
| Summary | 68 |
| Chapter 4: Building the Contact Form | 69 |
| Making a form plan | 69 |
| Handling mandatory fields | 70 |
| The form's layout | 71 |
| Input label magic | 73 |
| JS validation fallbacks | 79 |
| Summary | 80 |
| Chapter 5: Building the About Me Page | 81 |
| Justifying the About Me page | 81 |
| Making the wireframes | 83 |
| The markup | 84 |
| Awesome icon fonts | 87 |
| Summary | 91 |
| Appendix A: Anatomy of HTML5 Boilerplate | 93 |
| What is HTML5 Boilerplate? | 93 |
| Conditional comments | 94 |
| Many, many mobile meta tags | 95 |
| Including the scripts you'll need | 96 |
| The helper.js file | 97 |
| Appendix B: Using CSS Preprocessors | 99 |
| Why? | 100 |
| How | 100 |
| CodeKit | 100 |
| Compass | 101 |
| The Sass/LESS gem | 101 |
| Rails | 102 |
| What | 102 |
| Resources | 103 |
| Index | 105 |

Preface

Building websites that display well on everything from web-enabled smartphones, to tablets, to laptops and desktops, is a daunting challenge. The myriad permutations of screen sizes and browser types might be a reason enough to not even try. But if your business counts on getting web content to people on these devices and you need your business to look tech-savvy, you must put your best foot forward. In this book, you will see that with the help of some easy-to-understand principles and an open source framework you can build a mobile first responsive website fast.

What this book covers

Chapter 1, Mobile First – How and Why? gives a quick introduction to mobile first strategy.

Chapter 2, Building the Home Page, dives right in and builds the face of your site and the foundation for the rest of the site.

Chapter 3, Building the Gallery Page, builds a responsive page to show off your work.

Chapter 4, Building the Contact Form, lets prospective clients contact you from a device of any screen size.

Chapter 5, Building the About Me Page, makes an attractive, responsive page to help people get to know you.

Appendix A, Anatomy of HTML5 Boilerplate, gives an overview of HTML5 Boilerplate, including meta tags and scripts.

Appendix B, Using CSS Preprocessors, helps you learn the basics of CSS Preprocessors and how to use them.

What you need for this book

You should have Windows or Linux. The instructions in this book favor Mac OS X and Linux, but for the most part we will only be writing plain text and using very few command-line tools. In places where we do, I do my best to offer up resources for how to get similar results on a Windows computer. You should also have a text editor. It will be very helpful if you know how to launch and use a command-line tool.

Who this book is for

This book is for you if you are curious or excited about responsive design and how it can help provide usable web interfaces on everything from mobile phones to desktop computers.

In terms of technical skills, this book is targeted at both beginner to intermediate developers as well as designers. In other words, you should already know how to build an HTML page and style it with CSS by using a text editor of some kind. You don't have to be an expert at any of these things though. You also don't need to be a command-line expert, but hopefully you are open to using command-line tools. They are quite helpful.

Conventions


In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.


Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When the browser replies yes to both `screen` and `min-width 768px`, the conditions are met for applying the styles within that media query."

A block of code is set as follows:

```
<!DOCTYPE html>
<head>
  <link rel="stylesheet" href="css/main.css">
</head>
<body>
  <button class="big-button">Click Me!</button>
</body>
</html>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "I went ahead and created links to pages that don't exist yet, so if you click on them you will get a **404 file not found** message."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Mobile First – How and Why?

If you are in the business of building/maintaining a company's website or building web properties for an agency, you can and should be using mobile first strategies. Why? Because it is of value to your end product. You will end up with a website that is used by most of the people on all the devices and browsers possible.

This book is targeted at both beginner and intermediate developers as well as designers. It is also intended to be for those in business and management who want to gain a deeper understanding of what is possible (and, by extension, what may not be practical) with modern tools and strategies on the web. The code examples in this book, when used step-by-step, should help anyone with even basic development skills to get a deeper understanding of what is possible as well as how it is possible. Of course I love building things and I do it every day, but for those of us who also have to strategize and educate clients and coworkers, having procedural knowledge of how to make a mobile first website is qualitatively better than only having the knowledge of theory and concepts.

What is Responsive Web Design?

Responsive Web Design (RWD) is a set of strategies used to display web pages on screens of varying sizes. These strategies leverage, among other things, features available in modern browsers as well as a strategy of progressive enhancement (rather than graceful degradation). What's with all the buzzwords? Well, again, once we dig into the procedures and the code, it will all get a lot more meaningful. But here is a quick example to illustrate a two-way progressive enhancement that is used in RWD.

Let's say you want to make a nice button that is a large target and can be reliably pressed with big, fat clumsy thumbs on a wide array of mobile devices. In fact, you want that button to pretty much run the full spectrum of every mobile device known to humans. This is not a problem. The following code is how your (greatly simplified) HTML will look:

```
<!DOCTYPE html>
<head>
  <link rel="stylesheet" href="css/main.css">
</head>
<body>
  <button class="big-button">Click Me!</button>
</body>
</html>
```

The following code is how your CSS will look:

```
.big-button {
  width: 100%;
  padding: 8px 0;
  background: hotPink;
  border: 3px dotted purple;
  font-size: 18px;
  color: #fff;
  border-radius: 20px;
  box-shadow: #111 3px 4px 0px;
}
```

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

So this gets you a button that stretches the width of the document's body. It's also hot pink with a dotted purple border and thick black drop shadow (don't judge my design choices).

Here is what is nice about this code. Let's break down the CSS with some imaginary devices/browsers to illustrate some of the buzzwords in the first paragraph of this section:

- Device one (code name: Goldilocks): This device has a modern browser, with screen dimensions of 320 x 480 px. It is regularly updated, so is highly likely to have all the cool browser features you read about in your favorite blogs.
- Device two (code name: Baby Bear): This device has a browser that partially supports CSS2 and is poorly documented, so much so that you can only figure out which styles are supported through trial and error or forums. The screen is 320 x 240 px. This describes a device that predated the modern adoption levels of browsing the web on a mobile but your use case may require you to support it anyway.
- Device three (code name: Papa Bear): This is a laptop computer with a modern browser but you will never know the screen dimensions since the viewport size is controlled by the user.

Thus, Goldilocks gets the following display:

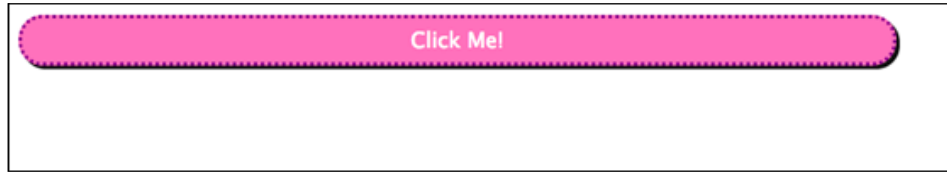


Because it is all tricked out with full CSS3 feature, it will render the rounded corners and drop shadow.



Baby Bear, on the other hand, will only get square corners and no drop shadow (as seen in the previous screenshot) because its browser can't make sense of those style declarations and will just do nothing with them. It's not a huge deal, though, as you still get the important features of the button; it stretches the full width of the screen, making it a big target for all the thumbs in the world (also, it's still pink).

Papa Bear gets the button with all the CSS3 goodies too.



That said, it stretches the full width of the browser no matter how absurdly wide a user makes his/her browser. We only need it to be about 480 px wide to make it big enough for a user to click and look reasonable within whatever design we are imagining. So in order to make that happen, we will take advantage of a nifty CSS3 feature called `@media` queries. We will use these extensively throughout this book and make your stylesheet look like this:

```
.big-button {
  width: 100%;
  padding: 8px 0;
  background: hotPink;
  border: 3px dotted purple;
  font-size: 18px;
  color: #fff;
  border-radius: 20px;
  box-shadow: #111 3px 3px 0px;
}

@media only screen and (min-width: 768px){
  .big-button {
    width: 480px;
  }
}
```

Now if you were coding along with me and have a modern browser (meaning a browser that supports most, if not all, features in the HTML5 specification, more on this later), you could do something fun. You can resize the width of your browser to see the start button respond to the `@media` queries. Start off with the browser really narrow and the button will get wider until the screen is 768 px wide; beyond that the button will snap to being only 480 px. If start off with your browser wider than 768 px, the button will stay 480 px wide until your browser width is under 768 px. Once it is under this threshold, the button snaps to being full width.

This happens because of the media query. This query essentially asks the browser a couple of questions. The first part of the query is about what type of medium it is (print or screen). The second part of the query asks what the screen's minimum width is. When the browser replies yes to both `screen` and `min-width 768px`, the conditions are met for applying the styles within that media query. To say these styles are applied is a little misleading. In fact, the approach actually takes advantage of the fact that the styles provided in the media query can override other styles set previously in the stylesheet. In our case, the only style applied is an explicit width for the button that overrides the percentage width that was set previously.

So, the nice thing about this is, we can make one website that will display appropriately for lots of screen sizes. This approach re-uses a lot of code, only applying styles as needed for various screen widths. Other approaches for getting usable sites to mobile devices require maintaining multiple codebases and having to resort to device detection, which only works if you can actually detect what device is requesting your website. These other approaches can be fragile and also break the **Don't Repeat Yourself (DRY)** commandment of programming.

This book is going to go over a specific way of approaching RWD, though. We will use the **320 and Up** framework to facilitate a mobile first strategy. In short, this strategy assumes that a device requesting the site has a small screen and doesn't necessarily have a lot of processing power. 320 and Up also has a lot of great helpers to make it fast and easy to produce features that many clients require on their sites. But we will get into these details as we build a simple site together.



Take note, there are lots of frameworks out there that will help you build responsive sites, and there are even some that will help you build a responsive, mobile first site. One thing that distinguishes 320 and Up is that it is a tad less opinionated than most frameworks. I like it because it is simple and eliminates the busy work of setting up things one is likely to use for many sites. I also like that it is open source and can be used with static sites as well as any server-side language.

Prerequisites

Before we can start building, you need to download the code associated with this book. It will have all the components that you will need and is structured properly for you. If you want 320 and Up for your own projects, you can get it from the website of *Andy Clarke* (he's the fellow responsible for 320 and Up) or his GitHub account. I also maintain a fork in my own GitHub repo.

Andy Clarke's site

<http://stuffandnonsense.co.uk/projects/320andup/>

GitHub

<https://github.com/malarkey/320andup>

My GitHub Fork

<https://github.com/jasongonzales23/320andup>

That said, the simplest route to follow along with this book is to get the code I've wrapped up for you from: https://github.com/jasongonzales23/mobilefirst_book

Summary

In this chapter, we looked at a simple example of how responsive web design strategies can serve up the same content to screens of many sizes and have the layout adjust to the screen it is displayed on. We wrote a simple example of that for a pink button and got a link to 320 and Up, so we can get started building an entire mobile first-responsive website.

2

Building the Home Page

In this chapter, we are going to start using the 320 and Up framework to immediately get started on building the home page of our example portfolio site. We will start off with some basics of where specific code goes and why. We will then quickly move on to building our page with many of the typical elements of a portfolio home page: navigation, hero/slider, and a triplet of content panels. If you don't know what these terms mean, don't worry, you will soon!

If you have successfully downloaded and unzipped all the code from the link at the end of *Chapter 1, Mobile First – How and Why?*, you are ready to go. If not, go back and use the link there to download the sample code and return.

Preparing and planning your workspace

Everyone has preferred methods for where they keep their code and how they organize it, and there are a lot of conventions in web development about organization that are great to know about. Ultimately, if you have a workflow you like for working with code, especially code from tutorials, please just go ahead and use it. But for those of you who don't, I suggest you place the code you download in some kind of working directory where you keep (or plan to keep) all web projects. I typically keep all my web code in a directory I call `work` in my `home` folder. So on a Unix or Mac OS X machine, it would look like this:

```
~/work/320-and-up
```

A few last notes about where to put your code. If you are using this book specifically for the purpose of building something you want to deploy and use, you may only want to use the sample code as a reference and build your project using only the 320 and Up framework files provided. However, ensure that you put all of it in a directory named something other than 320 and Up.

Regardless of how you proceed from your end, I will provide the before and after code in every chapter so that you can have a template of sorts to get started and also an example of the final product that we will have by the end of this chapter. If you're just getting started and all this confuses you, just copy the code and edit it. You can always download a fresh copy if you need it later.

If you look inside the `ch2` directory, you should see the two folders `before` and `after`. From here on, I am going to assume that you will take the simplest route and directly edit the `before` files. But please do carry on with your preferred way.

Go ahead and move to or look into the `before` directory. You will see the `320andup` folder that I cloned from Andy Clarke's GitHub repository (`repo`). All I did was change location into the `before` directory by typing the following command line:

```
$ cd before
```

Then I cloned the code from the repo:

```
git clone git@github.com:malarkey/320andup.git
```

If you don't want to mess around with any of this, just use the code I have provided. I just want you to know how I got the code there.

Once you look inside the `320andup` folder, you will see a lot of files. Don't stress out. I will explain what we are working with as we go. And some of the files we simply won't use. If you were going to deploy this code, I would encourage you to go through some kind of production process to deploy only the code you really need. That is beyond the scope of this book though, because we will focus exclusively on building.

Planning ahead

I know you are probably excited to get started on writing some code, but first we need to do a bit of planning on what it is we will be building. When I prepare to build a site, this is what I do first so that I have a reference for what I am building with code. It's good practice; you don't want to just wing it. But it also gets more complicated when you are building a responsive site.

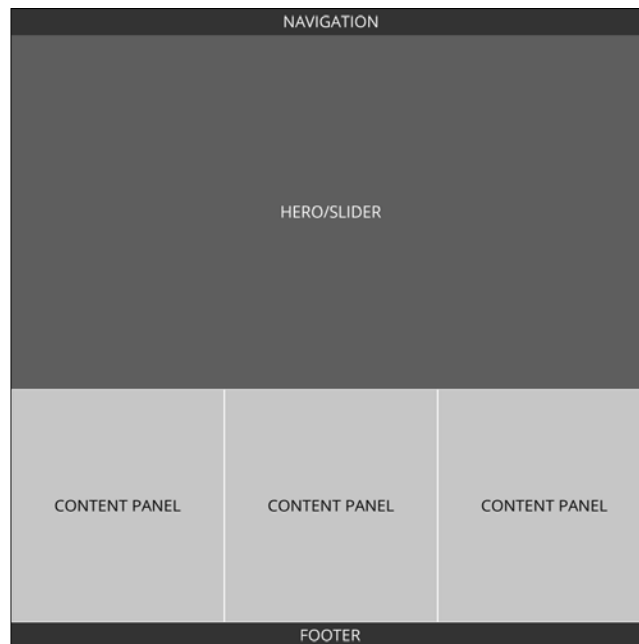
That said, here is the formula we will follow for each page that we will build:

1. Describe the elements we want on the page and their hierarchy.
2. Draw some simple pictures (called **wireframes**) of the elements on the page for all the different screen sizes we are coding to.
3. Write some code for a 320 px wide screen (with some thinking ahead).
4. Write some code for the other screen sizes we need to code for.

Let's start with step 1. Our portfolio site is going to have the following elements on the home page:

- Navigation menu
- Hero/slider
- Triad of content panels
- Footer

This is a fairly effective page layout for a portfolio site but it can work just as well for a company website. Before even designing a page, we should take a moment to plan out what the page content will look like in a really abstract way. Typically, the best way to represent this is with a wireframe. A wireframe should show where the content is placed on the page as well as the relative size. Here is what our site looks like as a desktop layout:



I quickly made that image in Photoshop, but you can easily do it in any image editor (in fact, many of my colleagues and I really like doing it with simple collaborative image editors, such as the one in Google Drive). You might want to take a moment right now to make your own image if you are making something that is different from this example.

The important thing in this phase is not to think about dimensions just yet (but that will come soon), and think instead about each kind of content and evaluate its importance with regard to the purpose of the site. The purpose of this portfolio site is to showoff our work so that we can get hired. To achieve that end, we've decided to have a home page, a gallery page, a contact form, and an **About Me** page. Not groundbreaking, but pretty effective. Next, let's examine how the home page can support the purpose of the site.

Navigation

On the home page, the navigation area will link to those pages I listed in the previous section:

- A logo
- Home
- Gallery
- Contact
- **About Me**

Hero/slider

This area is large and eye-catching. Let's plan to put some bold images and/or text here to drive people to the gallery work we want to highlight as well as the contact form.

Content panels

These areas should highlight the purpose of the site. I think that these areas are for those who will take the initiative to scroll down. In other words, those willing to scroll down are curious and we should supply them with more details about the purpose of the site. For example, my content might highlight three skill areas: frontend engineering, user experience, and visual design. Since I am mainly a frontend engineer, it is the highest priority; the next priority being user experience and the last being visual design. While all three will be visible at once on a desktop or a larger tablet, we can't comfortably fit all three in view on smaller tablets and mobiles.

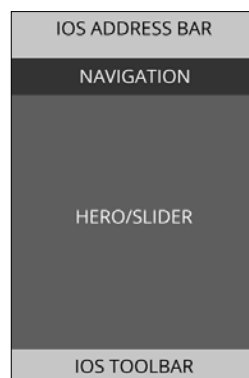
For yourself, think carefully about the three areas you want to highlight. It's common to dedicate a panel to social media integration as well. Whatever you decide on, make sure it gives more detail and doesn't just repeat the same content on the page.

Footer

The footer will have a short statement and a link at the top for the purpose of getting back to the main navigation. There's a really good reason to have a link to the top, especially on mobiles. On a mobile device, we need to provide an easy way for users to navigate from the top to the bottom of the page without having to manually scroll.

Ok, now we have our content prioritized and categorized, but you should have noticed a problem with the wireframe. I started with a desktop view but this book is mainly about designing for mobiles first, right? The reason I made that wireframe first is because I assume that most readers have designed a desktop page before moving on to mobile designs. In fact, it is common to only design for the desktop view! From here on, we will strictly be focusing on mobiles first. I promise!

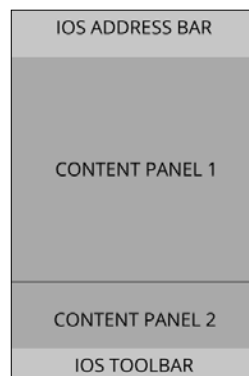
So knowing what our content is, we now need to make a layout that will work for mobiles. First, I'll show you what I think our layout should be and then explain the reason. Here it is:



Notice that we have to account for the address bar and the toolbar. Keep in mind that we aren't only designing for the iPhone. I just made that as a quick example, mostly because it is familiar to so many. The point is, on mobiles not only are you dealing with a small screen, you can't even count on getting all of the small screen since most mobile web browsers need some "chrome" for address and toolbar. There are some things we can do to try to reclaim that real estate, but more on that later. For now, we need to make a pessimistic assumption in order to plan our layout. And if we are using the currently very popular iPhone 4/4S's mobile Safari browser as an example, we only have 320 px by 376 px to work with because we use 60 px for the address bar and 44 px for the toolbar. The iPhone 5 is taller by about 88 px. To repeat though, we are not designing this just for the iPhone. We are looking at this example mainly to make a point – you can't necessarily fit a lot of content in the viewport.

In fact, it looks like we can only fit a navigation bar and the hero/slider. Better make sure that the content in the hero/slider counts for something! We won't focus too much on content strategy in this book, as there are a lot of other people who are far more experienced at it than me; nevertheless, let's do our best to put some well-chosen content there.

That said, we can still include all the other content; it's just out of view for now. If the users scroll down, they should still be able to see the three content panels just stacked rather than spread along the width of the page. A user who has scrolled down should see this:



If the users continue to scroll down, they will see the third panel and eventually the footer. To reiterate, by the time they get to the footer, it might be really helpful to have an easy access to site navigation from here.

Ok, so I bet you are eager to write some code and build! We can do it now that we know what we are building. Since a 320 px wide screen needs everything to fill the width of the screen, and all the main blocks are to be stacked, the HTML and CSS code will be quite simple!

Go ahead and open up the `index.html` file inside the `320andup` directory; or follow in the code sample and open up the file in this path:

```
ch2/before/320andup/index.html
```

We are going to take a quick look at this page in a browser and then we are going to change it to add our own content. Go ahead and view this file in a browser in your preferred manner. I prefer to use a Python simple HTTP server (see the following tip). But since we are only working with a static site, you can just double-click on the file or even drag it into a browser window.

Python simple HTTP server

I hate to be so Mac OS X-centric, but if you are using a Mac this will be easy. If you are using another *nix OS, it will still be pretty easy. If you are using Windows, it will be a little more work; nevertheless, it will probably be worth it.



To start a Python simple server on a Mac, you simply browse (via the command line) to the directory you want to serve up to a browser and type:

```
python -m SimpleHTTPServer
```

If using another *nix OS, you may need to install Python using your package manager and then run the preceding command. For Windows, you will need to install it from <http://www.python.org/getit/>. Follow the instructions to get it all going and then use the command line to run the same command.

For those of you familiar with WAMP/MAMP solutions, you may want to use those instead. You can find them at:

- <http://www.apachefriends.org/en/xampp.html>
- <http://www.mamp.info/en/index.html>

I highly recommend that you use a cutting edge browser, such as Chrome or Firefox, for the work we will be doing in this book, as they have really useful development tools that help you see what is going on with your code. Development tools make it easy for you to understand how things work as well as how to solve problems. In fact, many of the features we will be using are only available in modern browsers. So if you don't have one, go get one; they are all free and easy to install. For the record, my main development browser is Chrome.

Ok, once you have this in your browser, you should see what I have in the following image. Take a moment to read through it. You may have a lot of questions, and that is a good thing. By the time we build things, you will know a lot more.

320 and Up

This is the new '320 and Up'

A lot's happened since I wrote the original '320 and Up' — the 'tiny screen first' responsive web design boilerplate. Back then we were just getting started with responsive web design and many sites, including mine, and frameworks and boilerplates including HTML5 Boilerplate, structured CSS3 Media Queries from the desktop down, rather than from small screens up. (Oh how we laughed when we realised our mistake.) So to put things right, I wrote '320 and Up', to use as an extension to HTML5 Boilerplate or a set of standalone files.

Content first design's become the norm and HTML5 Boilerplate and its mobile cousin now both structure their stylesheets from small screens up. Twitter's Bootstrap and countless other frameworks include fluid grids, so what's left for '320 and Up'?

Upwardly mobile

I'm proud to say that '320 and Up' has been used by designers and developers all over the web. I've used versions of it on every website I've worked on since I wrote it. Small websites, medium-size websites and large websites including ISO, STV and UK Government websites that'll launch this year.

Along the way, '320 and Up' grown to include selected files and styles from Twitter's Bootstrap as well as responsive design libraries and polyfills. It's become my personal toolkit, somewhere I keep the files and styles that I use when I start every new project.

'320 and Up' contains:

- Five CSS3 Media Query increments: 480, 600, 768, 992 and 1382px
- Design 'atmosphere' (colour, texture and typography) separated from layout
- Styles for buttons, forms and tables compatible with Bootstrap.
- Font-based icons from Font Awesome
- Modernizr and Selectivizr

So, the first thing we need to do is edit this file (the one on the path `ch2/before/320andup/index.html`) to make it our own. Basically, we want to hollow out this page by removing the header, footer, and everything in between. In the `before` directory, I have provided an example called `index_stripped.html`. Feel free to compare your effort with that example file (if you are just beginning as a developer, don't be tempted to just change the name of `index_stripped.html` to `index.html` and use it; make the effort to edit the code successfully).

One more thing we will want to do right off the bat is make it so that we can pull in the JavaScript library jQuery from Google's servers. Google is very nice and hosts a ton of JavaScript and AJAX related libraries. So, many of us can use Google as a **Content Delivery Network (CDN)**. However, you may notice that the line of HTML that pulls it in from Google's service is missing something:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
```

It's missing the HTTP protocol, which is fancy talk for the first part of a URL, before the slash. I bet you're thinking why? The reason is that we need it to work within either `http` or `https` domains, depending on what our site is. Leaving it off essentially makes it so that it defaults to whatever the HTTP protocol is for the page this code lives in. If you specify it incorrectly as `http` within an `https` site (which is secure), it will throw a security warning to all well-made browsers because you can't serve up insecure content within the context of a secure site. Otherwise, `http` is just fine (you can also leave this out entirely and whatever protocol your site is using will apply).

For this project, I am using `http`; however, if you are building a secure site, by all means, make sure you make this secure as well. Here is what your code should look like now:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
```

Now if you refresh the page, you should not notice anything unless you look under the hood to see where your jQuery came from. If you don't know how to inspect whether site resources are downloading, don't worry about it too much right now. But if you are seeing errors, just double-check to make sure your code matches the example. You can check to see if you are getting JavaScript errors in any developer console, regardless of the browser you are using (even IE). Once this is working correctly, you can first have the page request that the jQuery library come from Google's service. If that fails, it will come from your site's server. Again, I won't go too much into the details of this boilerplate code, but it is good to know that the following line of HTML is a backup in case Google can't serve up the jQuery file when you request it:

```
<script>window.jQuery || document.write('<script src="js/jquery-1.7.2.min.js"></script>')</script>
```

Let's build!

OK! All the fundamentals are now in place. Let's build the components of the page for a small screen first. Let's go from the top of the page to the bottom. As I mentioned earlier, it typically makes sense for all the content to span the full width of small screens. Let's begin with the header and navigation.

Just below the opening body tag, let's put some HTML for our navigation. It should look like this:

```
<body class="clearfix">
<!-- PUT YOUR CONTENT HERE -->
<header>
  <nav class="navbar open">
    <div class="navbar-inner">
      <div class="container">
        <a class="logo" href=".">Logo</a>
        <ul class="nav">
          <li><a href="/index.html">Home</a></li>
          <li><a href="/gallery.html">Gallery</a></li>
          <li><a href="/contact.html">Contact</a></li>
          <li><a href="/about.html">About Me</a></li>
        </ul>
      </div>
    </div>
  </nav>
</header>
```

Header

We created a header block. We are using this for both semantic and layout reasons. The header will mainly contain the logo and navigation.

Logo

The logo will be contained in an `<a>` tag. This follows the unofficial web convention that the site logo should link back to the home page. We will still have an explicit link to the home page but it is helpful to offer both the links to users without being confusing. I use the shorthand `./` in order to have the page link back to the root of the current level of depth; for production, you may want to take the extra step of having it linked to your fully qualified root domain (for example, `www.yourdomain.com/index.html`).

Navigation

We create a semantic `<nav>` block and place some nested containers ending in an `` (unordered list) inside. Each `` (list item) will have a link to each page on our website. For this project, we will handcode each link, but if you were using some kind of framework, these links would be generated dynamically. I went ahead and created links to pages that don't exist yet, so if you click on them you will get a **404 file not found** message.

There are a few key things to notice about navigation. Right now, without any CSS applied, the basic layout is virtually what we want. Each link is stacked vertically and with some additional padding that will be a clear target for fat fingers the world around. This is all pretty ideal, since it's always good to know that your site will still function without CSS. This is good for many reasons. One being the case that your CSS fails to get served up for some reason. Another includes users who are using text-only browsers. You will also notice that there are a few relatively non-semantic containers here that function as utility containers. A few we will use soon.

One problem with this navigation is that once we style it properly, it will eat up a lot of screen real estate. The minimum area for an element that requires interaction on touch interfaces is roughly 50 px by 50 px so that it is wide enough for a fingertip. There is some leeway here though. For example, if the touch target is really wide, you can get away with making it about 40 px tall but that can get risky. Some usability experts recommend making your touch targets as wide as 60 px to accommodate the fattest finger – the thumb, since many users use it to get around on a mobile. For argument's sake though, let's make a compromise and assume each element to be 40 px tall and full-width, or at least 320 px wide. That means our navigation with the logo will be 200 px tall. We have potentially eaten up over half our screen real estate with just navigation and we do need to remember the potential chrome that we have to plan for. Greeting users with only navigation and no actual content is just plain bad.

We will need to do something about this!

Luckily, a convention has rapidly emerged to solve just this problem. Most mobile-friendly websites and mobile apps use an icon with a series of three parallel lines to signify a hidden navigation menu.



To the user, this should indicate that a touch or click of this element will reveal or hide the navigation. This assumes that the user knows the convention, of course. For this reason, there may be some situations where this is not appropriate, especially on sites where there is little navigation. That said, we are going to go ahead and build our navigation following this convention in order to save screen space and learn how to make this enhancement.

Here is the basic strategy we will use. We will hide and show the menu via CSS and use JavaScript to only change the class. This way, if users have no JS, they will still get the menu, but unfortunately it will be completely expanded.

So first things first; we will add a button. Add your button just below the `<a>` tag that will hold our logo. We will style the menu in a bit to organize things better, but let's get this working first. Here is what your navigation HTML should look like now:

```
<nav class="navbar">
  <div class="navbar-inner">
    <div class="container">
      <button class="menu-button">
        </button>
      <a class="logo" href=".">Logo</a>
```

If you refresh, you will now see a little nubbin of a button just to the left of your logo. It's not much to look at now, but be patient. We will write the JS code that will toggle some class to hide/show the navigation menu. Go ahead and open up the file in the path `ch2/before/320andup/js/script.js`. At this point, it should be an empty file. We are going to write some simple JavaScript that will hide and show the menu. Again, if a user doesn't have JS, the menu simply stays open. This is just one small example of progressive enhancement, there are more to come.

Next, we'll write this JS to assign a new class to the menu when a user touches the button. We are going to use some simple, elegant jQuery:

```
$(document).ready(function() {
  //all code that should run after the DOM loads goes here
  $(' .navbar').removeClass('open');
  $(' .menu-button').on('click', function(){
    $(' .navbar').toggleClass('open');
  });
});
```

Here is what this code does. The JS that appears first, `$(document).ready()`, is some jQuery that basically waits for the moment when the DOM has loaded, then executes all code placed within the `ready` function. It is typical to use this to make sure all the elements of the DOM are there so that the code that calls specific elements are actually all there.

The next line, `$('.navbar').removeClass('open')`, will remove the `open` class that we will use later to make the menu open and close with some CSS. If the device has no JS, then this class is never removed and the `open` style is the only one that will ever be applied to the menu!

The next line of code beginning with `$('.menu-button').on('click', function() {` attaches an event listener to the button that has a class of `.menu-button`. When a user clicks on the button, the code inside that function runs. Additionally, a touch event is translated into a click by mobile browsers, so both kinds of events are handled with this code. But getting back to the function—after a user touches or clicks, the function simply adds or removes the class `open` on the element with a class `navbar`. From here on, I won't go into too many details about the JavaScript we write. If you need more help in understanding it, that is beyond the scope of this book. But if you don't feel ready to dig into JavaScript, just follow along and you should learn something!

Now, if you save this code and reload your page, you can try this out. If you open your favorite developer tools and look at the `<nav>` tag when you click on the button, you should see the class `open` appear and disappear from that element. If it's not happening, or if you are getting errors, try retracing your steps and see if you missed some code. Also, try running the complete version of code from this chapter to see if it works properly. If the code I've provided you doesn't work, something other than the code is amiss.

If you don't see any errors, but at the same time don't see anything changing in your browser's inspector, just hang tight. It may not be updating the DOM for some reason. But we will soon see proof of its working once we add some styles.

Most of the CSS I will be writing can be written in plain CSS, SASS, or LESS. For a few reasons, I prefer to work with SASS. This subject too is outside the scope of this book. But for brevity, I will do my best to show you how to do all the CSS code examples both in SASS and plain CSS. Please read *Appendix B, Using CSS Preprocessors* and other preprocessors if you need to learn more. Otherwise, follow along and I will continue to show code examples of both CSS and SASS. The finished code samples are all in CSS and SASS/SCSS.

First things first, let's arrange the navigation menu so that things are laid out in a way that enhances usability and appearance. For example, let's get all those stacked elements to be 40 px tall.

If you are following and using SCSS, go ahead and open `_page.scss` inside the `scss` folder and make sure you change the name of the `css` file that is linked in the header of your page to:

```
<link rel="stylesheet" href="css/320andup-scss.css">
```

There are other ways you could handle this, of course, but let's keep it simple. If you are editing the plain CSS, just open the file in the path:

```
ch2/before/320andup/css/320andup.css
```

Again, you can always change the name of this file and the one linked to in your header if you wish, but I suggest we keep it simple for now and leave it as it is. Now, let's start styling this page. Just a quick note – for many of these styles, I am borrowing heavily from the great and powerful Twitter Bootstrap framework, which is a frontend framework that includes boilerplate CSS and HTML. You can include it with 320 and Up, but I decided not to include it in this book for simplicity. That said, if you decide to combine the two (and if you like building things well and quickly, I highly recommend you do), you will find that many of the styles I use are quite compatible with it. Now let's go!

First, let's get the button moved to where it should be and get it to look good:

```
.menu-button {  
  display: block;  
  float: right;  
  background: #444;  
  border: 1px solid #000;  
  padding: 7px 10px;  
  margin: 5px;  
}
```

The button is far away from all our links, so that users won't accidentally touch it when they are trying to open a link. It also looks a little better, but still needs those three lines that we discussed earlier. We won't need any images though.

If you are using any SASS or LESS, you can take advantage of one of the many handy mixins provided in 320 and Up. You should open up `_mixins.scss` and take a quick look at all of them. Again, if you are new to them, I will quickly give an example of what is so cool about them in just a moment; however, first a quick explanation of what mixins are in SASS and why they are so great.

In SASS, you can define mixins by typing `@mixin` followed by some CSS that you want to generate. This is great if you have a complicated task that you want to accomplish without repeated efforts. This harkens back to the concept of DRY; for example, we can make three rounded rectangles by using the rounded corners of CSS3 for the menu button. The trouble is that currently there are at least three different ways to declare rounded corners, thanks to vendor prefixes. For all rounded corners, we have to define them like so:

```
-moz-border-radius
-webkit-border-radius
border-radius
```

So, we could type the preceding code every time we need a rounded corner anywhere in our site styles. Or, we could save the effort and put these in a mixin. The rounded mixin does just that for you. Have a look at it in the `_mixins` file right now. Mixins in SASS do a lot of things, but this case alone is compelling. It essentially behaves like a callable function that executes when the code is compiled to CSS (read *Appendix B, Using CSS Preprocessors*, for more details). You code `@include rounded` and the CSS inside that mixin is rendered to your final CSS. In this case, you get all those ways of creating rounded corners without all the typing.

If you're already using SASS, here is all you need to do to see it in action on your site (if you're not, read *Appendix B, Using CSS Preprocessors*, to see how to get it going). First, we will add some new markup to our button.

```
<button class="menu-button">
  <span class="icon-bar"></span>
  <span class="icon-bar"></span>
  <span class="icon-bar"></span>
</button>
```

Write this SCSS nested inside your `.menu-button` SCSS:

```
.icon-bar {
  display: block;
  width: 18px;
  height: 2px;
  margin-top: 3px;
  background-color: #fff;
  @include rounded(1px);
}
```

The rounded mixin will render the following CSS (or you can handcode this if you wish):

```
.menu-button .icon-bar {
  display: block;
  width: 18px;
  height: 2px;
  margin-top: 3px;
  background-color: #fff;
  -webkit-border-radius: 1px;
  -moz-border-radius: 1px;
  border-radius: 1px; }
```

The last three lines are generated by the mixin when the SCSS is processed. This is quite a time-saver. By now your button should be looking neat and floating over to the right!

Now, let's get all those links to look neat. Here is what your SCSS should look like:

```
.navbar {
  background: #1b1b1b;
  .navbar-inner {
    .logo{
      display: block;
      padding: 9px 15px;
      font-weight: bold;
      color: #999999;
      margin-bottom: 4px;
    }
    .nav {
      a {
        @extend .logo;
      }
    }
  }
}
```

And here is the CSS:

```
.navbar {
  background: #1b1b1b; }
.navbar .navbar-inner {
  background: #1b1b1b; }
.navbar .navbar-inner .logo, .navbar .navbar-inner .nav a {
  display: block; }
```

```
padding: 9px 15px;
font-weight: bold;
color: #999999;
margin-bottom: 4px; }
```

This will give a neat contrast and make the links 40 px tall. But now we need to do something to get that menu hiding and showing. My preference is to do it without JavaScript animation. Ok, it's more than a preference actually. CSS3 animations will be smoother for the most part, furthermore; this really comports with the ideology of progressive enhancement. If a device does not support CSS3 animations, it is quite possible that it isn't really powerful enough to deal with JavaScript animations either, so why are you forcing it to run JS loops just for a nice-to-have feature? On the other hand, most devices that support CSS3 animations optimize these animations by utilizing the GPU. Even if they don't, they will still play a JS animation as well.

I won't get too clever with my arguments, but this code essentially works well if you are on a slow device that doesn't support CSS3 animations and if you are on the slickest mobile out there.

First things first, we need to make one embarrassing concession here. CSS3 animations will not work when the height of an element is automatically calculated (yet!). This doesn't have to matter for us, since we can easily know the height of our navigation menu. But if you wanted to use this kind of animation on a menu of an unknown size, you could not use this approach. There are other approaches for that scenario; however, they are not included in this book. ☺

So, here is what your SCSS now needs to look like:

```
.navbar {
  background: #1b1b1b;
  overflow: hidden;
  max-height: 44px;
  @include transition(max-height .5s);
  &.open{
    max-height: 220px;
  }
  .navbar-inner {
    .logo{
      display: block;
      padding: 9px 15px;
      font-weight: bold;
      color: #999999;
      margin-bottom: 4px;
    }
    .nav {
```

```
        margin-bottom: 0;
        list-style:none;
        a {
            @extend .logo;
        }
    }
}
```

And the CSS:

```
.navbar {
  background: #1b1b1b;
  overflow: hidden;
  max-height: 44px;
  -webkit-transition: max-height 0.5s;
  -moz-transition: max-height 0.5s;
  -ms-transition: max-height 0.5s;
  -o-transition: max-height 0.5s;
  transition: max-height 0.5s; }
.navbar.open {
  max-height: 220px; }
.navbar .navbar-inner .logo, .navbar .navbar-inner .nav a {
  display: block;
  padding: 9px 15px;
  font-weight: bold;
  color: #999999;
  margin-bottom: 4px; }
.navbar .navbar-inner .nav {
  margin-bottom: 0; }
```

We set the maximum height of the open menu $5 \times 44 = 220\text{px}$. There are five stacked elements in `nav` and we know that they are each 44 px tall (I could tell by looking in my dev tools). By extension, the closed version, the version that has had the open class removed should have a max-height of 44 px. We need the overflow to be hidden so that the other elements aren't visible when the menu collapses to a smaller height.

You should also notice that the five different ways of creating the CSS3 transition animations were written with one line of SCSS (another mixin):

```
@include transition(max-height .5s);
```

Things are looking really nice now! Play around with it and enjoy. This was a pretty intense section. The rest will be a tad simpler, I promise!

Next, let's move on to our *Hero* section. For now, we will simply have a background with some placeholder text and a button. But I will provide some tips and suggestions for making a slide show later in this section.

Hero

Let's keep the markup simple for now. Later, we will come back and make this a simple slideshow.

```
<div class="hero">
  <div class="container">
    <h1>Big Headline</h1>
    <p>YOLO vero scenester, semiotics next level flannel Austin
shoreditch portland 3 wolf moon chillwave gentrify consequat tousled
retro. Umami tonx ennui cliché delectus pinterest, in excepteur
hashtag before they sold out.</p>
    <a href="/contact.html" class="btn btn-primary btn-
extlarge">Contact Me</a>
  </div>
</div>
```

The `hero` `div` acts as a container for some styles and content that we will add. For now, we will just stick to adding a headline, some text, and a button that will eventually take users to our contact page.

Here is what the SCSS should look like:

```
.hero {
  text-align: center;
  padding: 40px 20px;
  text-shadow: -1px 1px 0px #E0B78A;
  @include horizontal(#feb900, #cb790f);
  h1 {
    margin: 10px 0;
    font-size: 45px;
    font-weight: bold;
  }
  p {
    font-size: 18px;
    margin: 0 0 30px 0;
    font-weight: 200;
    line-height: 1.25;
  }
}
```



```
    }  
    .btn {  
      text-shadow: 1px 1px 0px #000000;  
    }  
  }  
}
```

...and the CSS:

```
.hero {  
  text-align: center;  
  padding: 40px 20px;  
  text-shadow: -1px 1px 0px #e0b78a;  
  background-color: #cb790f;  
  background-image: -webkit-gradient(linear, 0 0, 100% 0,  
from(#feb900), to(#cb790f));  
  background-image: -webkit-linear-gradient(left, #feb900, #cb790f);  
  background-image: -moz-linear-gradient(left, #feb900, #cb790f);  
  background-image: -ms-linear-gradient(left, #feb900, #cb790f);  
  background-image: -o-linear-gradient(left, #feb900, #cb790f);  
  background-image: linear-gradient(left, #feb900, #cb790f);  
  background-repeat: repeat-x; }  
.hero h1 {  
  margin: 10px 0;  
  font-size: 45px;  
  font-weight: bold; }  
.hero p {  
  font-size: 18px;  
  margin: 0 0 30px 0;  
  font-weight: 200;  
  line-height: 1.25; }  
.hero .btn {  
  text-shadow: 1px 1px 0px black; }
```

Again, you can see the use of a mixin. We used the gradient mixin, `@horizontal`, to create eight lines of plain 'ol CSS. Convinced you should be using SASS yet?

Everything else is relatively straightforward. You may notice that I had to override the text-shadow of the button with a black colored shadow, since the peachy-colored shadow would have looked pretty terrible behind white text on a black button. All the other choices are just some basic styles for this area, which you can feel free to adjust according to your taste.

Now, let's move on to the trio of content panels that will go at the bottom.

Content panels

Now below the hero, place this example code:

```
<!--panels -->
<div class="full clearfix">
  <div class="grids grids-three clearfix">
    <div class="header header-link clearfix">
      <h2 class="h2">Heading</h2>
    </div>
    <div class="grid grid-1 clearfix">
      <p class="grid-a"></p>
      <h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
    </div>
    <div class="grid grid-2 clearfix">
      <p class="grid-a"></p>
      <h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
    </div>
    <div class="grid grid-3 clearfix">
      <p class="grid-a"></p>
      <h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
    </div>
  </div><!-- / grids -->
</div>
```

Now, I have to confess at this juncture that all I did for this section was copy Andy's example from his panel `upstart`. These are darn useful. You can find his examples inside any of the preprocessor folders, but I got mine from `ch2/before/320andup/scss/320andup-panels/index.html`.

Not only are these automagically (that's silly developer speak for something that happens automatically but seems mysterious and magical) laid out for us but, as you will soon see, they are already responsive without us having to make any effort. This is a huge payoff!

The only change I want to make is to the background color of the `div` with a class of `full`. The bluish color doesn't go well with my orange theme. But if you look at the SCSS for the panel `upstart` (in `upstarts/320andup-panels/_upstart.scss`), you will notice that the color for the background is calculated from a `$basecolor` variable:

```
background-color : lighten($basecolor, 75%);
```

That means you need to assign the `$basecolor` variable to something. Let's just use one of the shades of orange from our hero gradient! Open up `_variables.scss` and change `$basecolor` to this:

```
$basecolor: rgb(203, 121, 15);
```

You will notice that our button in the hero changed color! Whoa! That's actually Ok, I planned for it. This is a powerful feature of tying your styles together with variables, but it can bite you if you don't pay attention.

Ok! Now things are looking really sharp! If you resize your browser, you can see the content panels change size and layout. We just need to make a footer, then we can add some responsive styles of our own for the things 320 and Up has not done for us.

Footer

Let's keep things simple again:

```
<footer>
  <div class="container">
    <h4>Let's build something awesome together.</h4>
    <p>Connect with me in any of the following ways</p>
    <ul class="social">
      <li><a class="icon-facebook-sign" href="http://faceylink.
html"></a></li>
      <li><a class="icon-twitter-sign" href="http://twitterlink.
html"></a></li>
      <li><a class="icon-envelope" href="mailto:something@yourmail.
com"></a></li>
    </ul>
    <div class="toplink">
      <a href="#">Top</a>
    </div>
  </div>
</footer>
```

We will style it like so. First the SCSS:

```
footer {
  padding: 15px 0;
  text-align: center;
  background: #1b1b1b;
  color: $white;
  overflow: auto;
  p {
    padding: 9px 15px;
  }
  .social {
    list-style: none;
    margin: 0 auto;
    width: 280px;
    li{
      float: left;
      height: 80px;
      width: 80px;
      list-style: none;
      border-radius: 50%;
      background: #000;
      margin-right: 20px;
      a {
        font-size: 42px;
        padding-top: 24px;
        color: $lightgrey;
        &:visited {
          color: $grey;
          text-decoration: none;
        }
        &:hover{
          text-decoration: none;
        }
      }
    }
    li:last-child {
      margin-right: 0;
    }
  }
  .toplink{
    clear:both;
    a {
      display:inline-block;
```

```
        padding: 30px;
        color: #fff;
    }
}
}
```

And the CSS:

```
footer {
  padding: 15px 0;
  text-align: center;
  background: #1b1b1b;
  color: white;
  overflow: auto; }
footer p {
  padding: 9px 15px; }
footer .social {
  list-style: none;
  margin: 0 auto;
  width: 280px; }
footer .social li {
  float: left;
  height: 80px;
  width: 80px;
  list-style: none;
  border-radius: 50%;
  background: #000;
  margin-right: 20px; }
footer .social li a {
  font-size: 42px;
  padding-top: 24px;
  color: #bfbfbf; }
footer .social li a:visited {
  color: gray;
  text-decoration: none; }
footer .social li a:hover {
  text-decoration: none; }
footer .social li:last-child {
  margin-right: 0; }
```

Hopefully, you've gotten the hang of this now. But you can once again see the use of some variables as well as the ampersand sign in the SCSS to help write code faster.

Next, we handle what happens to the layout on larger screens.

Making our page responsive

The best way to see when you need to add new styles is to take your browser window from its narrowest and gradually drag it wider. When the design starts to look weird or broken, it's time to restyle it.

In our case, the main things we will need to restyle for larger screens are the navigation, the hero, and the footer. The content panels are already taken care of for us. Let's start with the navigation.

In the case of the navigation, we implemented the hide/show functionality to save valuable screen space, but at some point we don't need to make users click to reveal the menu. We can simply leave the navigation fully displayed at all times like a desktop site navigation that we are used to. In order to find the point where that layout breaks, we could drag our browser width, which could quickly get tedious. Also, in reality, responsive websites aren't for wackos that resize their browsers spontaneously and repeatedly, like yours truly, but for devices of different sizes. Luckily, 320 and Up has a useful tool in its toolbox to help out.

If you open up an HTML file called `responsive.html` in the directory you're working in (to remind you it's `ch2/before/320andup/responsive.html`), it should just automatically load your `index.html` file. Now, by scrolling left and right, you can see your layout in five good layout breakpoints (not to be confused with breakpoints used in debugging code). Of course, there will be exceptions, but these breakpoints are a real time-saving place to start as they tend to hit the range of devices currently available. I encourage you to critique and question, but for now, let's take advantage of them as they are paired up with 320 and Up and will speed up development that is going to support a good design in almost all cases. If you open this page through your computer's filesystem, it won't load the pages. See my note earlier in the chapter to find a good way to open up this page. But to restate, my personal favorite for something this simple is the Python simple HTTP server.

Ok, so when you successfully get this page to load, what do you see? You should notice that the design works really well on the mobile and small tablet layouts. Based on the minimal amount of placeholder content I have in here, it doesn't look too sparse nor does it look too cramped. And as a bonus, those neat content panels expand to fill the extra real estate. The framework facilitates this via `@media` queries. More on this in a minute.

That said, what do you think about the tablet — portrait layout? It usually works, but we have more room in that hero area now. That doesn't mean we have to add more content but we can probably make that text a little bigger to fill it out. To heck with that, let's make it really bold and get people's attention. The nice thing is that 320 and Up already has all the structure in place to make it easy to change the size. First let's look at the code, then I will explain what goes on under the hood.

If you are using SASS, this is super easy. Open the `_768.sass` or `_768.scss` file and add this code:

```
.hero {
  h1 {
    font-size: 108px;
  }
  p{
    font-size: 40px;
  }
}
```

or in CSS, find the point in your file that says:

```
@media only screen and (min-width: 768px)
```

and within the curly braces add this code:

```
.hero h1 {
  font-size: 108px; }
.hero p, .hero footer a, footer .hero a {
  font-size: 40px; }
```

So, if you are new to SASS or `@media` queries, I will take a moment to help you understand what is happening here. First, I will explain the `@media` query. Quite simply, in this case all it does is tell the browser that once the screen is a minimum width of 768 px, the contained styles should be applied. You can set other dimensions and other conditions as well.

As for the magic in SASS that allows us to organize these styles in separate files, there is a similar syntax that is only in the `.sass` or `.scss` file (not the `.css` file) and is, in essence, an instruction to the preprocessor to pull in separate files. You may have noticed that the file you edited (and a bunch of others) has an underscore at the beginning of the name. That indicates that it is a partial file. If you look at the `320andup-sass.sass` file or the corresponding file for the language you chose, you will notice that inside all the `@media` queries, there are `@import` statements. For the file we just edited, there is an `@import 768` statement inside the same exact `@media` query you see in the plain CSS file:

```
// 768px
@media only screen and (min-width: 768px) {
  @import "768";
}
```

When it gets to this point in the file, it tells the SASS preprocessor to go find the file with the name `_768.sass` and render whatever code is there into this place. So, not exactly rocket science, but a lot of the busy work of setting all this up has been taken care of for you.

Ok, now back to getting this design to respond to this tablet size. The other thing you will notice is that we probably don't need to have the navigation elements hidden anymore. If we can keep the navigation convenient and show a lot of content as well, then we've accomplished some very important missions! So let's go back to that `_786.sass` file and add this above our previous chunk of code:

```
.menu-button {
  display: none;
}
.navbar {
  position: fixed;
  width: 100%;
  .logo {
    float: right
  }
  .nav li {
    float: left;
  }
}
```

You'll notice this structure mirrors the structure of our original `site.sass` file. This is just a good practice for maintenance reasons as well as making sure that the styles actually override the other ones.

Refresh your screen if necessary and now you'll see that the navigation elements extend from left to right. This is probably what you are used to seeing on a regular old desktop website. And there was much rejoicing. There is the possibility of moving this style into the `_480` file layout too, but it looks a bit crowded to my eye. That said, if you had less navigation and a small logo (or no logo), you might want to apply that style at 480 px instead.

There is one more neat little tweak that we should make at this point. All the content in the navigation and hero is sitting closer to the edge of the viewport than it needs to. We can definitely add some breathing space. In the markup, we have a nice utility class that we can use for this purpose (it's been something of a convention that frontend developers have been using for a while now). Add this code in the `_768` file of your choosing, above all the previous code we've written so far:

```
.container {
  width: 90%;
  margin: 0 auto;
}
```


This allows us to center these containers within the other elements that we want to visually fill the width of the screen, which makes our app have a huge visual impact without the content sprawling too much. This sizing and margin is dynamic and changes in a fluid manner as the browser gets wider in proportion to the content panels below. We could set explicit widths at various visual breakpoints; however, we wouldn't be taking advantage of the framework then. I would argue that fixed widths are a passing paradigm.

Let me explain what I mean by that. In the earlier days of web design, designers made their pages look more like... well, pages. But web design currently has an expanded idea that the page should be more flexible. I think this is a good thing, don't you? Just as an example, users with large displays don't get a narrow band of content in the middle of their page.

Along those lines, did you notice another convenience of the way 320 and Up is designed? Once we applied the styles for 768, those styles are applied to the larger screens too. Neat! Less code means faster and better work, and easier maintenance. It also means less CSS for a browser to download. That is the UI trifecta: good user experience, good performance, and maintainability.

Now, another thing you'll notice is that our footer is fine. I must admit that I've taken the easy way out with that, but the approach I've used here is still useful for content of this type. When there is little content for an area of the page, such as a footer, it pays to just tastefully center-align all the content. When done properly, it is easy to read and doesn't distract from what is clearly the more important content up above on the page. And if the footer has something really important, you should consider moving it up into the body of your page!

Next, let's revisit, the Hero area and discuss adding images in there and use some simple code to cycle through them.

Slider

So, before we get into making a slideshow, it will be useful to see how well 320 and Up facilitates making images responsive. If you look in the supplied code in the `index.html` file, you will see a block of code just below the hero markup for slider markup. I have left comments in there to make it easy to find.

For now, because I want you to see something already put in place for you in your own file, only add the following markup:

```
<div class="slider">
  <div class="container">
```

```
    
  </div>
</div>
```

Use this markup in place of the hero markup (either delete the hero stuff or comment it out; it's totally up to you).

Next, add this small bit of CSS to the `site.css` file you are working in (it will be identical in all file types):

```
.slider {
  text-align: center;
}
```

Refresh the page and play around with the width of your browser. You should see the image change without ever getting cropped. This is an elegant solution in that one image will work for all layouts. It is not a solution for all cases (and there are a lot of discussions right now about how to make images more responsive to screen size). But here is the situation it is good for: I have a small number of images that aren't too large and I don't need to have them cropped differently for different screen sizes. This situation is actually pretty common, so as long as you can get your images to be lightweight, it works well.

Now, let's add a little bit more complexity without recreating the wheel. For now, we will just add two more images and I will supply some simple JavaScript to cycle through and have the images fade-in.

Let's change the markup so that we can get ready for more JS and CSS:

```
<div class="slider">
  <div class="container">
    <div class="slide active"></div>
    <div class="slide"></div>
    <div class="slide"></div>
  </div>
</div>
```

Not a ton of code, but let's walk through this to understand it in depth. We need to wrap the images in `<div>` tags now (for other purposes, you could always put them in other block elements; however, right now this simple markup is totally appropriate for our purpose). These `div` containers allow us to assign classes and do block level styling to anything within a slide and not just images. For now, we are only placing a single image in these slides, but if we wanted to add captions or buttons or something that would become impractical. For the slideshow to display flexibly, we just need these wrappers around everything.

Now, let's look at some CSS to get this to display properly:

```
.slider {
  text-align: center;
  position: relative;
  .container {
    position: relative;
    .slide{
      position: relative;
      display: none;
      &.active{
        display: block;
      }
    }
  }
}
```

and the compiled CSS:

```
.slider {
  text-align: center;
  position: relative; }
.slider .container {
  position: relative; }
.slider .container .slide {
  position: relative;
  display: none; }
.slider .container .slide.active {
  display: block; }
```

This markup allows us to make sure that the first image is the only one visible without even running any JavaScript. The `slide` class by default is not visible and it only becomes visible when it gets the class `active` added to it. This not only works at the code level, but also reads nicely. You read the code and it says `class="active slide"` and you have a pretty good idea what that means.

To move on, let's add some JS to see if we can get a simple animation going. This will not be a fancy animation. Just to warn you; if you want something with cool controls and other bells and whistles, that is beyond the scope of this book. If you want a neat responsive slideshow, I recommend either the carousel included in Twitter Bootstrap or any other responsive slideshow. This sample code I am sharing below will simply cycle through some images.

Add this inside your document ready function:

```
var changeSlide = function(){
    //query the DOM for an active slide
    var $active = $('.slider .active');
    //if there are no active slides set the last one as active
    if ( $active.length === 0 ) {
        $active = $('.slide').last();
    }
    //get the next slide after the active one, if there is no next one,
    set next as the first slide
    var $next = $active.next().length ? $active.next() : $('.slide').
    first();
    //set classes on active and next slides so we can apply styles
    appropriately
    $active.addClass('last-active');
    $next.addClass('active');
    $active.removeClass('active last-active');
};
//this will kick off the slideshow code above
$(function() {
    setInterval( changeSlide, 5000 );
});
```

This code was adapted from <http://jonraasch.com/blog/a-simple-jquery-slideshow> to work with our 320 and Up layout. It will cycle through your images and append the active class to each one while removing it from the previous one. Then once it gets to the last one, it assigns it to the first one. Again, a very simple approach since the focus of this book is 320 and Up. If you want to use a slideshow, I suggest not reinventing the wheel since there are a lot of great components out there. If you are looking to choose a good component, look for one that is either designed to be responsive or at least does not interfere with it.. Another criterion for me is that it uses CSS3 animations with JS polyfills. CSS3 animations are likely to (though not in all cases) run smoother on mobiles than JS animations.

One limitation of the image we have used is that for really large screens, the image kind of gets swallowed up in all the negative space on the left and right of the slide. If this bothers you and is keeping your site from looking as good as you think it should, there are two strategies at your disposal: include larger images or put a full-width background in that area. I prefer the latter because a proportionally larger image is going to eat up the top of our layout and would also mean larger files that could definitely harm performance.

Remember that the ultimate goal here is to get content to the visitors on our site! The latter strategy requires some planning though. Either your images need to have some transparency around the edges or the background of all your images should match the background you use in your CSS or the slider area. I am going to show you a simple example of matching the background.

I happen to know that the sample images I created have a vertical gradient that goes from #383234 to #231F20. So now all I need to do is make a background that matches that. Using the SCSS mixin provided in 320 and Up is ridiculously easy. I just add this to my `.slider` styles:

```
@include vertical(#383234, #231f20);
```

and that is rendered to CSS as:

```
background-image: -webkit-gradient(linear, 0 0, 0 100%,
from(#383234), to(#231f20));
background-image: -webkit-linear-gradient(top, #383234, #231f20);
background-color: #231f20;
background-image: -moz-linear-gradient(top, #383234, #231f20);
background-image: -ms-linear-gradient(top, #383234, #231f20);
background-image: -o-linear-gradient(top, #383234, #231f20);
background-image: linear-gradient(top, #383234, #231f20);
```

The limitation of this approach is that devices that don't support gradients will get a solid color. If this is unacceptable to you, then it's time to go back to the drawing table and come up with a design that will work in all scenarios! In most cases, I have worked with designers who either throw their hands up in this situation or find ways to make their design work in all situations. If you ask me, it's a moving target and it's best to focus your energies on a design that you know will look spot-on to 80 percent of your audience and still decent to the remainder of your site's viewers.

Ok! Now you have the fundamentals for a home page that will display optimally on virtually any device! This was a lot of work, but now that we have laid the ground work, the other pages will go fast.

Summary

In this chapter, we created navigation that changes based on screen size so that users of small screens can expand or collapse it and users of large screens get the entire navigation menu. We even used CSS to create the icon that indicates a collapsible menu. We made a responsive hero area with a big call to action, leveraging mixins and variables to quickly get our design to come together with colors that complement one another. We used the panel `Upstart` to get the triad of content panels at the bottom of our page and we used the supplied icons and CSS framework to include social media and contact info icons in the footer. And best of all, this happened really fast. When you get the hang of it, you can pull a page like this together within an hour. Now let's move on to the next chapter!

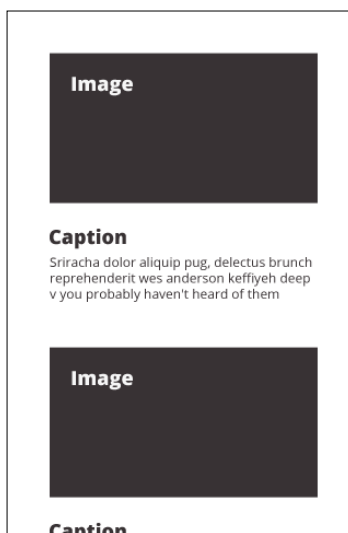
3

Building the Gallery Page

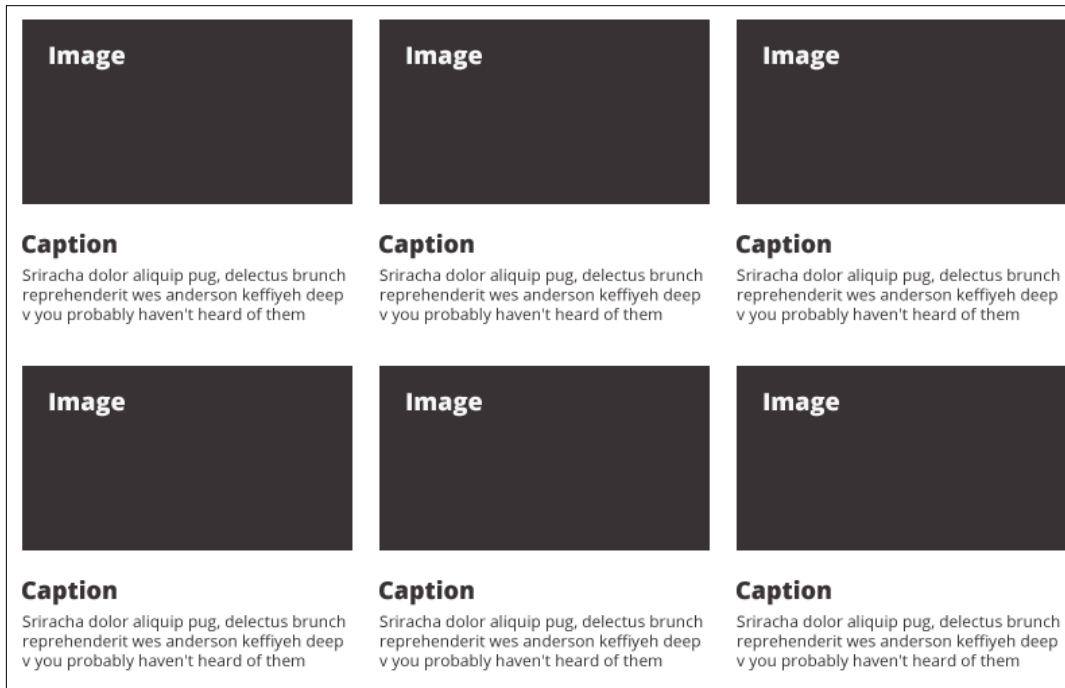
In the previous chapter, we did a lot of work and we built quite a base for the rest of our portfolio site. With the knowledge we now have and the small amount of code we wrote on top of the 320 and Up framework, we can really start to move fast. In this chapter, we are going to do just that. We are going to build a gallery of panels that will be stacked for narrow screens and tiled for wide screens. To do this, we are going to use the same basic approach that we did in the previous chapter for the triad of content panels at the bottom of our page.

Creating the wireframe

Before we jump into the code, let's take a look at some wireframes. The following screenshot shows how our screen should look on a small screen:



As the browser gets wider, we'd like those images to get bigger and change the layout from stacking to tiling for better use of the screen space. The following screenshot is the basic layout of a screen over 992 px wide:



There are a lot of visual breakpoints between the two that I have created above that 320 and Up facilitates. The only thing to keep in mind with 320 and Up is how to keep the rest of your page consistent with it. As we go along here, let's analyze what the layout is doing and get the rest of our page to play nice with it by either using styles that already exist in 320 and Up or creating our own.

The way I would create a **Gallery** page for myself is with some kind of hero at the top of the page, but not a slideshow. I feel it's important to orient users with a simple, bold statement at the top of most pages for this kind of site. The main reason is that you cannot count on the user coming to your **Home** page first, so you need to establish quite a bit of background about your website on every page. I guess I would compare this strategy to the one that writers of serial television use: you have to assume that the viewer may need a little background information in every episode or, in this case, page.

So, let's start the page off with a hero that is not quite as tall, but still has really bold text – a heading and a short sentence.

To get started, you can grab the `gallery.html` file from `ch3/before/320andup/gallery.html`. This file will have all the items we can re-use from our **Home** page, for example, the code in the head of the file, the navigation, the footer, and so on. Take a few moments to look at it and see what we are re-using for every page as we move forward. Go ahead and open this page in a browser to see what you have to start with. You should see the navigation part butted right up against the footer. Don't worry, we will fill the space in between them soon.

Before we move on, I just want to quickly outline what code we are keeping from the first page we made and why. Of course, we are keeping everything from the top of the file to the closing `</head>` tag. We are also re-using the code that begins at the footer and continues to the bottom of the file. This is the entire code that we need on every page to do essential work across devices and browsers as well as to include our styles, favicons, and JavaScript libraries. We also have code that will be the same on every page, such as the markup for navigation and the footer. In other words, we are repeating this code on every single page we are building. This is true but if, for example, you were using a framework such as Django or Rails, or some kind of templating language in another framework, you would separate the code that would be repeated on every page out to its own file so it could be re-used and shared in other files. This would be a great approach to solve the problem of requiring this code to appear on every page.

However, in order for me to make this book platform-agnostic, I have simply copied this code over from page to page as we progress. I really do not recommend making a site this way. Repeating the same code in different places is just an invitation to make a horrible mistake at some point (here's hoping I haven't made one).

Let me explain some of the risks of just manually copying this code over from page to page. For example, if you decide to make a change to the navigation part on one page, you have to remember to make the same change on every page and also make sure you execute it with precision. If you've been programming for a while, you will recognize this principle as **Don't Repeat Yourself (DRY)**. It is a bedrock principle of writing code. You should follow it. This principle is a major justification as to why I recommended using Sass (or LESS) in the previous chapter and why you should be using a framework like 320 and Up!

Ok, enough of me pontificating.

Other aspects of this page would also do well to be re-used in your framework of choice. The header, which contains the navigation, and the footer are highly likely to be identical on every page, so I would make these components re-usable, too. The last thing I will point out is that you should also re-use the JavaScript at the bottom. Many times, developers use strategies to make the inclusion of CSS and JavaScript dynamic, based on the page needs, but for our simple site that is unnecessary.

Now that we've got that business taken care of, let's move on to taking care of the content that will be unique to this page.

The slim hero

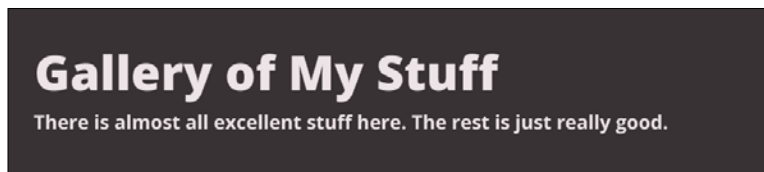
Now, we need a hero at the top of this page, but we don't want to distract too much from the gallery tiles. So, we don't want a big, splashy image. Instead we want some bold content that quickly sums up what is happening on the page and meets the following requirements (that I also mentioned previously); Assume that the visitor may have landed on this page without ever seeing the rest of the site while not insulting the intelligence of a person who has already been browsing the site.

What you actually say is up to you, of course. The real objective is to understand the strategy of the content and how it relates to our layout.

Here is a mock-up of what we are shooting for in a 320 px-wide device:



Here is how it should look on a desktop browser:



The major differences between the two mockups are as follows:

- The 320 px layout will need smaller font sizes than the wider layout
- The 320 px layout has the text aligned to the center whereas the wider layout has the text aligned left

This isn't mandatory or anything; it's just a design decision I have made that we can also support with the responsive design, as luck would have it.

Now, let's write some code for this. Place the following markup just below the closing `</header>` tag. Here is what the markup needs to look like:

```
<div class="hero">
  <div class="container">
    <h1>Gallery of My Stuff</h1>
    <p>There is almost all excellent stuff here. The rest is just
      really good.</p>
  </div>
</div>
```

Now, the cool thing is, if you have everything in its right place, your page will look close to how we want it to because we are re-using styles from the **Home** page. To be precise, it only looks right when we look at it on a 320 px wide screen. Take a moment to look at the **Home** page and this new **Gallery** page and get your head around how the styles are being re-used. You may notice one thing, now that our `<h1>` tag has text that goes over two lines, our `line-height` is too high. Let's tighten that up a bit. Edit the `.hero h1` style by adding the following line of code:

```
line-height: 1em;
```

Now, refresh the page and see how that keeps the headline nice and compact. It's a cozy look, don't you think?

Ok, so that was super easy! This just works.

Here's why that works: You'll notice I set the height to `1em`. An `em` is a unit of measurement different from pixels. Pixels set an explicit measurement and `ems` set a relative measurement. Why set a relative height? To make future changes easier. An `em` is equal to whatever the current font size is. So, in this case the `line-height` ends up being equal to the font size. This is the desired outcome, since we want there to be little to no extra white space generated by a `line-height` greater than the font size.

So, why bother to be so abstract? You don't always have to be but I like to use `ems` in places like this because it makes changing the font size less troublesome. If I come back later and need to adjust the font size, I won't need to also adjust the `line-height` to maintain the current styling effect. `Ems` will continue to render a `line-height` that is the same as the font size.

Next, let's look at how all this works for the desktop view. Using your preferred method (the responsive `.html` page or just resizing your browser), go ahead and take a look at the widest width we are styling for, 1382 px wide.

The trouble is that there are some fine points that are off by just a little bit. One thing you will notice is that the font size on the desktop layout is just a tad too big to keep the layout slim as we want. So, we need to override some of the styles we re-used from the **Home** page.

There are two ways to accomplish this. One way is to assign a class somewhere above the page elements that we want to style differently and then have some new style descend from that class that will override the existing class. For example, currently, the `<h1>` element in the hero gets this style:

```
.hero h1{
  font-size: 108px;
}
```

So, we could just add a new class to our hero as follows:

```
<div class="hero slimmer">
```

Then, somewhere further down on the stylesheet, have a style that has the following properties:

```
.hero.slimmer h1{
  font-size: 60px;
  text-align: left;
}
```

Then, this style would override the `.hero` styles applied above it. However, this is not ideal; now that we have two kinds of `hero`, the non-slim one is semantically vague. Instead, we can add a class to both kinds of `hero` to make it clear that the styles apply to a large version of a `hero`.

First, let's go back and change our **Home** page HTML as follows:

```
<div class="hero jumbo">
  <div class="container">
    <h1>Big Headline</h1>
    <p>YOLO vero scenester, semiotics next level flannel Austin
      shoreditch portland 3 wolf moon chillwave gentrify consequat
      tousled retro.</p>
    <a href="/contact.html" class="btn btn-primary btn-
      extlarge">Contact Me</a>
  </div>
</div>
```

Notice the addition of the class `jumbo`. Now, assuming you are using SCSS, we need to edit our stylesheet for screens with dimensions 768 px and above to match it. In order to do that, open up the `ch3/before/320andup/scss/_76s.scss` file. For future reference, I will just ask you to open up the file by its name rather than stating the whole path. So, for this file, I will ask you to open up the `76s` file. When we need to edit styles that apply to layouts 992 px and wider; I will ask you to open up your `992` file, and so on. With that established, let's continue with adding some code to the `76s` file. In this file, our SCSS used to read as follows:

```
.hero {
  h1 {
    font-size: 108px;
  }
  p {
    font-size: 40px;
  }
}
```

Now we will replace `.hero` with `.jumbo`.

So now, the whole section of SCSS file should look like the following code snippet (with CSS to follow):

```
.jumbo {
  h1 {
    font-size: 108px;
  }
  p {
    font-size: 40px;
  }
}
```

Then, add the following CSS in the previous code snippet:

```
@media only screen and (min-width: 768px) {

  .jumbo h1 {
    font-size: 108px; }
  .jumbo p {
    font-size: 40px; } }
```

So, the cool thing is that `.jumbo h1` and `.jumbo p` are much more re-usable now that they are decoupled from `.hero`, which has a pretty specific application.

Now we need to work on the styles for our **Gallery** page. Let's make the HTML code look as follows:

```
<div class="hero subhead">
  <div class="container">
    <h1>Gallery of My Stuff</h1>
    <p>There is almost all excellent stuff here. The rest is just
      really good.</p>
  </div>
</div>
```

We don't assign a subhead style for 320 styles, but let's add it for styles that need text to use up the available space better. The first step is the 480 px visual breakpoint. Take a look at this layout at 480 px; we can get away with pumping the font size up a bit. It seems like a small change but let's do this not only because we can, but because next year there will almost certainly be a tablet that is 520 px wide and your layout is going to be more likely to hold up at this resolution now that you took the time to do this!

If you are using SCSS (or another preprocessor), add the following code to your 480px file:

```
.subhead {
  h1 {
    font-size: 48px;
  }
  p {
    font-size: 24px;
  }
}
```

This will render CSS that is once again nested inside the query @media only screen and (min. width: 480px) and looks like this:

```
.subhead h1 {
  font-size: 60px; }
.subhead p {
  font-size: 24px; }
```

We are keeping the text centered at this point, since the rest of the layout is going to be centered too. More on this once we add the content panels.

Let's move on to the next visual breakpoint, that is, 600 px. The heading at 48px looks a bit puny. Let's go ahead and pump it all the way up to 60px. Add this to your 600px file:

```
.subhead {  
  h1 {  
    font-size: 60px;  
  }  
}
```

This previous code renders the following CSS:

```
.subhead h1 {  
  font-size: 60px; }
```

Now, moving on to the 768 px visual breakpoint—how does it look to you? I think this font size works here and up to the other breakpoints but, if you are so inclined, make changes to the larger sizes too. To a certain extent, it depends on how much you want to tailor the font size to your content or how much you want some safer, more generic styles to work well with the dynamic content. My goal with these layouts is to make a layout that is likely to work with a wide range of content.

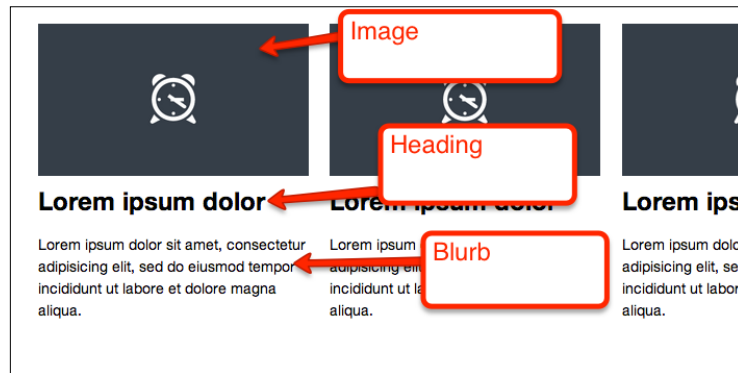
Now, we have the fonts in the `subhead hero` looking good at all sizes! Take a moment to re-size your browser and watch how everything changes and uses the existing screen space. One thing you may notice is how the space between the text and the edge of the viewing area gets dramatically narrower, up to somewhere between 600 px and 786 px. The reason this happens, you might recall, is because we don't style the `div` tag with the `container` class until we hit the 768px breakpoint. We will address that in a moment, but let's see how it plays with the content panels before we mess with it too much.

At this point, I should mention that my own approach to building responsive layouts, whether working alone or with a team, is always recursive like this. I try to build one component of a page until I feel like it is either exactly what I want or I find that I have questions about how it will play with the rest of the content on the page; this is one such juncture for me. While working with a team, I might start to code out this page and get feedback from a designer or another developer, then tweak it until we are all satisfied enough to ship the code or show the client, whatever the case may be. Since, we are making a site for ourselves we are just iterating alone (you and me together).

So, on that note, we are going to add our content panels, but then we are going to need to loop back around and make sure our `subhead hero` looks Ok.

Content panels

You may remember that, as with the **Home** page, each content panel will have an image, a heading, and a short blurb.



If I were doing this to show off my portfolio of work on websites, I would use screen grabs of each project I want to highlight and work on writing fairly short headings and blurbs for each. By default 320 and Up has each heading as a link to a corresponding page but you could link each panel, instead, if you are worried that people won't click on it. Later, we will make an example page to demonstrate where a user might land if they were to click the heading.

For this example page, we will continue to use the placeholder image and Lorem Ipsum, but feel free to make actual, meaningful content if you have some ready. Additionally, if you are hooking these layouts up to some kind of content management system or blog, you should think through how you will have to change your code for those kinds of approaches. For example, you may be building this layout via a loop in a template that relies on the number of `gallery` objects you have created.

For the panels themselves, all you need to do is use the same ones we put on the **Home** page; but instead of only three, you can add as many as you want to show off your awesome work.

Here is the HTML code you will need to make the first set of three panels:

```
<div class="full clearfix">
  <div class="grids grids-three clearfix">
    <div class="header header-link clearfix">
      <h2 class="h2">Heading</h2>
    </div>
    <div class="grid grid-1 clearfix">
```

```
<p class="grid-a"></p>
<h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
  sed do eiusmod tempor incididunt ut labore et dolore magna
  aliqua.</p>
</div>
<div class="grid grid-2 clearfix">
  <p class="grid-a"></p>
  <h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna
    aliqua.</p>
</div>
<div class="grid grid-3 clearfix">
  <p class="grid-a"></p>
  <h3 class="h2"><a href="#">Lorem ipsum dolor</a></h3>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna
    aliqua.</p>
</div>
</div><!-- / grids -->
</div>
```

Now, you just have to save and refresh your page and you will see it all work! Ridiculously efficient, isn't it? Now, of course, for your own content, it is critical that you use 410 x 230 pixel-images that you will crop with either image-editing software or a nifty editing tool in a CMS or blog. You will notice that there is an `<h2>` heading above this group of three panels. I would only include this if you have some kind of sensible groupings that will benefit from being titled. The layout will work fine with or without this particular heading. I would not recommend removing the `<h3>` headings because, along with the images, they will really help viewers scan the page and find information quickly.

So, at this point, you can either use the placeholder images in the code sample to build this page or start including your own content. If you are doing it statically, as opposed to building the page with a loop in some kind of template, you will just need to keep copying-and-pasting these panels.

Experiment a bit; however, I would like to give you just some quick thoughts about how these panels will work on the page. One thing to consider is how one gets the panels to lay out without being separated by headings or additional whitespace. All you really need to do is keep repeating the `grid-1`, `grid-2`, and `grid-3` blocks as needed (with the entire markup that is inside them, of course). I have also shared this in the example code.

Another thing to consider is what to do if you don't have panels in exact multiples of three. That is Ok, too! It just works. I have shared this in the example code as well. I stopped at just five blocks and it lays out just as you'd want it to.

There are only two problems left to solve with regard to this page. The first problem, you may notice, is that, in my initial wireframe, I wanted to left-align the text for larger layouts. Somewhat arbitrarily, I have decided that we will do that for all layouts above 600 px. Go into your 600px file and add the following code:

```
.subhead {
  text-align: left;
  h1 {
    font-size: 60px;
  }
}
```

Or, add the following code in the 600px @media query in your CSS:

```
.subhead {
  text-align: left; }
.subhead h1 {
  font-size: 60px; }
```

Now, the text is only centered for smaller devices and left-aligned for tablets and larger devices.

The second problem you might catch as you resize your browser between 600 px and 768 px. Hopefully, you've noticed that the text in the hero ends up being a lot closer to the edge of the viewing area than the rest of the layout. You may recall that we have all the content inside the `content` container, but that class doesn't get styled until the 768px @media query fires. Perhaps we should try to apply the styles within 768 @media query to 600px-width screens instead and see what this does to all layouts at all breakpoints above 600px. So, right now, go ahead and cut that style from the 768px file and paste it in the 600px file. Or, if you are using plain CSS, you will need to remove this code from the 768px @media query and paste into the 600px one.

Now, once you've done that, go back and play around with both the **Home** and **Gallery** pages by resizing your browser. The heading in the hero now stays nicely aligned with the panels beneath it. This change doesn't seem to adversely affect the **Home** page hero or the footer, so it looks like we are good to go.

To a certain extent, this is how I develop for responsive websites, I see what breaks and try to fix it, in a generic, elegant way where possible, without putting undue constraints on the content.

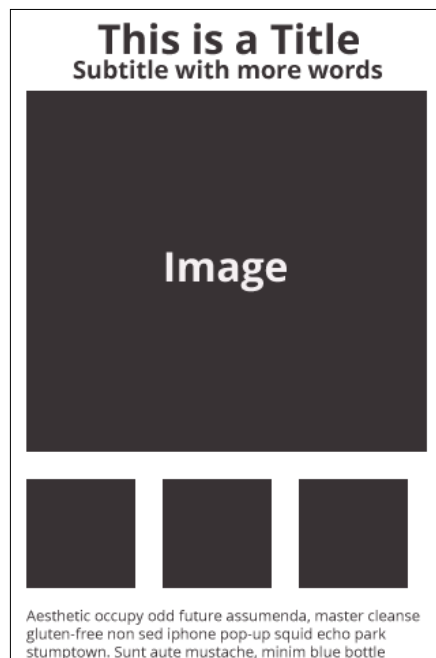
Now, the last big task we have for this chapter is to make the page that the users will land on when they click on the link for the corresponding content panel. We will call this task the gallery detail.

The gallery detail

So, let's have a look at the content we want on this page and how to strategize the layouts for different devices.

The main things, I think, most people want to see in a portfolio of any kind are a few key images and some lengthier, detailed text describing images.

Here is the layout we will need for mobile screens:



The smaller squares, shown in the screenshot, will be the thumbnails that users can touch or click on to show the larger image above the smaller images on the page. The first thumbnail will be the default image that will be displayed when the **Gallery** page loads. We will also highlight the thumbnail that is currently active, with a border. In order to do this, we will need to make both full-sized and thumbnail-sized images of all the images. The description for each image will be below the thumbnails (and that's completely appropriate in my book). If your images are compelling enough, people will scroll down to read it.

Now, let's look at the opposite end of the spectrum – the desktop view. We will want to orient things differently now that we have more screen real estate:



With the added space, we can put the text that you worked so hard to write right next to the big image below which the thumbnails are placed. Let's assume that, as with the other responsive layouts, these two layouts will cover all the breakpoints as long as we size key components of the page in percentages.

As before, let's start with the layout for mobile screens first. The HTML structure of the page should be fairly straightforward. We will need to put all the images on the page, hiding all but the first featured image, and then we will add our text. Again, let's not think too hard about what the desktop page will look like; let's just get what the mobile layout needs as simply as possible.

As before, you will need to make sure you re-use the basic structure of our page – the header, navigation, and footer. We will be inserting new content below the header after the closing `</header>` tag (to be precise). Please make sure to carefully investigate where the following code is going in the example code at `ch3/320andup/gallery-item.html` so you can follow along.

Here is what the HTML code we will add on the gallery item page will look like:

```
<header class="page-header">
  <h1>This is a Title</h1>
  <h2>Subtitle with more words</h2>
</header>
<div class="gallery-showcase">
  <div class="gallery-image-area">
    <ul class="featured-images">
      <li class="featured-image-item active"></li>
      <li class="featured-image-item"></li>
      <li class="featured-image-item"></li>
    </ul>
    <ul class="thumbnail-images">
      <li class="thumbnail actice"></li>
      <li class="thumbnail"></li>
      <li class="thumbnail"></li>
    </ul>
  </div>
</div>
<div class="gallery-description">
  <p>some text here...</p>
  <p>even more text if you want...</p>
</div>
```

You should notice in the basic structure of the code that we have an outer container that holds all our main images and all our thumbnail images. This container has the class `gallery-showcase`. Inside this class, there are containers for the big images and the smaller thumbnail images — `featured-images` and `thumbnail-images`, respectively — that users will click on or touch to see the corresponding larger images. The inner `gallery-image-area` container is there to help out with layout, mostly. As we did with the slideshow, we will load all the images onto the page while hiding the ones that aren't active using CSS. Later, we will hook up some simple, elegant JS to make it all interactive. The last bit, you will notice in the previous code, is the `gallery-description` container that will hold the text you write about the portfolio item.

I have included placeholder images for you and they are provided in the shared code. The size of the big images is 550 x 550 pixels and that of the thumbnails is 80 x 80 pixels. If you want to use these layouts for your own purposes without modifying the layout, you will need to edit some images to those sizes.

If you are feeling impatient, as I often do, you have already refreshed this page and can see that it is not ready for primetime. We have some work to do. Let's start off by hiding the large gallery images that won't be seen when the page loads. Much as we did for our hero slideshow, we will assign a class to the first image; this will make it the only visible image on the page. Go back to the HTML file we just made and notice that we have the class `active` assigned to the first large image and the first thumbnail.

Ok, let's get to styling!

We are going to add some styles to the page file; for me that is the `page.scss` file. But, if you are just directly editing CSS, you will need to just add these styles into your CSS file. The way they are rendered by the SCSS preprocessor includes these styles below the table styles in the code, fairly far down in the file. I recommend following this so that these styles are lower in the stylesheet than the more general, sitewide styles. I tend to think of the `site` and `page` files a little differently from *Andy* (no offense). I think of the `site` styles as elements on the page that will appear across the site on every (or nearly every) page. Then, I think of the `page` styles as being specific to particular pages, with less likelihood of being re-used on another page.

First, let's get the headings how we want them. You may notice that we have a second `<header>` on this page (yes that is allowed). We want to add some page-specific styles to the headings within the page. First, we need to get the text below the image to align to the center, so all we need in SCSS or CSS is the following code added to the appropriate spot (see the code samples that can be downloaded from the Packt Publishing website if you're not sure):

```
.page-header {
  text-align: center;
  margin: 12px 0;
}
```

Then, we need to apply styles for the fonts:

```
.page-header {
  text-align: center;
  margin: 12px 0;
  h1 {
    font-size: 30px;
    margin: 0;
  }
}
```

```

    }
    h2{
      font-size: 18px;
    }
  }
}

```

The previous code is rendered to this CSS:

```

.page-header {
  text-align: center;
  margin: 12px 0; }
.page-header h1 {
  font-size: 30px;
  margin: 0; }
.page-header h2 {
  font-size: 18px; }

```

Now, let's apply styles to hide all the big gallery images that aren't active. To do that, write the following SCSS code:

```

.gallery-image-area {
  .featured-image-item {
    display: none;
    &.active {
      display: block;
    }
  }
}
}

```

The previous code renders the following CSS:

```

.gallery-image-area .featured-image-item {
  display: none; }
.gallery-image-area .featured-image-item.active {
  display: block; }

```

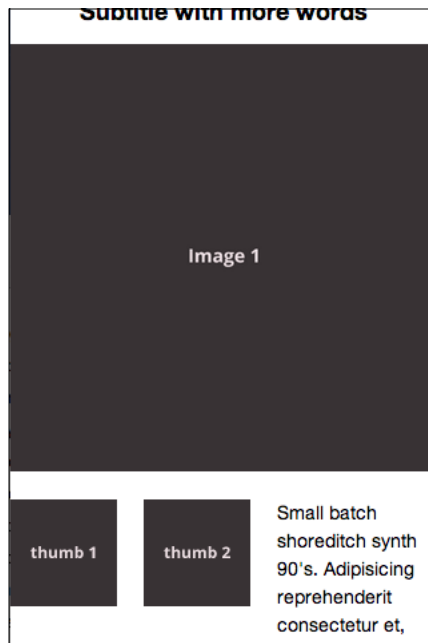
Refresh the page and now you should only see the first big gallery image. Progress!

You should be looking at our layout at 320 px (remember that this is mobile first) and will see that, so far, everything is stacked – absolutely everything. The only thing that we, for sure, don't want to stack is the thumbnail images. So, let's get those laid out correctly. Basically, all we need to do is get the ``s tags to float left and add some space and we will have most of what we need.

Add the following code to your stylesheet:

```
ul {
  list-style: none;
}
.thumbnail {
  float: left;
  margin: 0 20px 20px 0;
}
```

In this code, we first removed the browser's default bullets for lists (but only for lists inside the `gallery-image-container` block) and then we made the thumbnails float left. This way, you could potentially have as many thumbnails as you want, but I'd keep it down to about three to keep things simple for your site's visitors. The trouble, though, is that, if you have three in there as I do with the shared code, you will trick yourself into thinking that this layout works just fine. If you have three thumbnails in there, temporarily remove the entire HTML code for the third one so that you only have two left.



See what happens? The text we have below the thumbnails creeps up because the thumbnails `` are floats. If you don't know about all the idiosyncrasies of floats, I encourage you to read up on them. Here is the link to a great article about floats: <http://alistapart.com/article/css-floats-101>

But, for now, I will just show you one way to fix this. All we need to do is clear the container that holds the text. You do that by adding the following SCSS/CSS:

```
.gallery-description {
  clear: left;
}
```

Let's do just one more thing before we move on to the larger layouts: let's get all the content away from the edges of the screen and get it to a size similar to that of navigation. We can do this using a similar approach to what we used earlier with the hero.

Add the following SCSS/CSS code to your stylesheet:

```
.gallery-showcase, .gallery-description {
  width: 90%;
  margin: 0 auto;
}
```

Now, it's looking good! All we need is to make the layout appropriate for a larger layout. Once we are above 768 px, we can move the text up to the right of the large image. Go ahead and open up your `_768` file or find the `768 @media` query in your CSS and we will move the text up to right with minimal effort. Here is the SCSS/CSS code (again; it is the same as the one shown earlier in the chapter):

```
.gallery-showcase {
  width: 45%;
  float: left;
  margin: 0 2.5%;
}

.gallery-description {
  clear: none;
  width: 45%;
  float: left;
  margin: 0 2.5%;
}
```

Now, drag your browser's width around and enjoy. There is one last problem to solve: you may notice that the headings shift up under the navigation part once the size of navigation changes. You may recall that this is due to the navigation part being a fixed element. All we need to do is specify a different margin on the page-header as follows:

```
.page-header {  
  margin: 66px 0 12px 0;  
}
```

And, that should look right. Refresh the page and enjoy!

The back link

Let's add a simple enhancement to the page in order to make navigation easy. Since this Gallery item page isn't in the menu and it wouldn't necessarily be practical to add a menu item for every Gallery item, let's just add a back link to the top of the page. This makes it easy for users on any device to get back to the **Gallery** page.

First, let's add this link to the markup on `gallery-item.html`. Make the back link the last thing inside the header:

```
<a class="back btn" href="/gallery.html">&lt; Back</a>
```

Here is the code in context:

```
<header class="page-header">  
  <h1>This is a Title</h1>  
  <h2>Subtitle with more words</h2>  
  <a class="back btn" href="/gallery.html">&lt; Back</a>  
</header>
```

Refresh your browser and you will notice that you get some nice button styling for free, thanks to 320 and Up. We will need to do just a little bit more styling but first let me clear up something important. What we just built is one example Gallery item page linked from the `gallery.html` page. If you are building this as a hand-built, static site (despite my earlier advice not to), you will need to build this page manually for all your gallery items and give each page its own name that is not `gallery-item.html`. Instead, you might need to name it `company-site.html` or whatever the project you are showing off is. Furthermore, you will notice that I created the back link back to the **Gallery** page as follows:

```
...href="gallery.html"...
```

That is not a typical way to form links; in your own project you are more likely to make the link as follows:

```
...href="/gallery.html"...
```

In our example code, we need to use `gallery.html` because the more typical `/gallery.html` would take us to the root of our entire project. And, guess what? There is no `gallery.html` page at the root of this project because I have broken the project up into chapters. So, with this example project, you will get a **404** response (**Page Not Found**). Try it out.

To sum up, you will most likely want your link to look like `href="/gallery.html"` not like `href="gallery.html"`.

Now, let's add some styles to that button so that it isn't just sitting in the middle of the page. The simplest thing to do, for now, is to float it to the left. To do this, add the following code of styles to your equivalent of the `_page.scss` file:

```
.back {  
  float: left;  
}
```

I nested this inside the code for `.page-header`. So, in this context, the code looks as follows:

```
.page-header {  
  text-align: center;  
  margin: 12px 0;  
  h1 {  
    font-size: 30px;  
    margin: 0;  
  }  
  h2 {  
    font-size: 18px;  
  }  
  .back {  
    float: left;  
  }  
}
```

And the code in CSS will look as follows:

```
.page-header .back {  
  float: left; }
```

That isn't enough, though. Refresh the page after adding this style and you will notice that things look broken. That is because we need to clear that float. Simple! Add the following line of code to your SCSS or CSS for the `.gallery-showcase`, `.gallery-description` style:

```
clear: both;
```

This style will be applied to both elements, which doesn't affect anything adversely for our purposes. If you want to, though, you can always split your CSS code up into two separate styles if it bothers you. There is one last thing to do now for this button. It is sitting right up next to both the viewport and the main image. Let's go back and add a margin to push it away from everything. Here is what the updated `.page-header .back` style should look like:

```
float: left;
margin: 20px;
```

Next, let's get the gallery item JavaScript happening!

The gallery item JavaScript

Next, we need to write some JavaScript to meet our needs in this Gallery item page. Our needs are very simple; if a user clicks on a thumbnail, we want to show the corresponding larger image. There are lots of strategies for this but I am going to rely on two things to do this fast and easily: our page structure and jQuery's ability to index things easily. So, let me show you the code first and then I will explain how it works. Paste or type this code into your `script.js` file, anywhere within the ready function (look at the code sample that can be downloaded from the Packt Publishing website if you aren't sure):

```
$('.thumbnail').on('click', function(){
  var idx = $(this).index();
  $('.featured-images').children('.active').removeClass('active');
  $('.featured-image-item').eq(idx).addClass('active');
  $('.thumbnail-
    images').children('.active').removeClass('active');
  $('.thumbnail').eq(idx).addClass('active');
});
```

Here is what that code does line by line:

The following line of code attaches an event listener to the thumbnails so that, when it is clicked, the rest of the code inside that function is executed:

```
$('.thumbnail').on('click', function(){
```

The following line of code gets the zero-based index of the thumbnail you just clicked on. In other words, it finds out how many other thumbnails are alongside this one and which number it is in the sequence. Zero-based is the way computers count. So, if you clicked on the first thumbnail in the list, it will get an index of 0; the second will get an index of 1. Sorry if that is confusing, but that is how computers count many things. Anyway, we are going to use that number to target the corresponding image in the list of `featured-images` in a moment.

```
var idx = $(this).index();
```

The following line of code removes the `active` class from the `featured-image-item` that currently has the class `active` on it:

```
$('.featured-images').children('.active').removeClass('active');
```

The following line of code adds the `active` class to the `featured-image-item` container that is in the same place in the list as the corresponding thumbnail:

```
$('.featured-image-item').eq(idx).addClass('active');
```

The following two lines of code function the same as the two we just looked at and also remove and add the `active` class on thumbnails instead:

```
$('.thumbnail-
  images').children('.active').removeClass('active');
$('.thumbnail').eq(idx).addClass('active');
```

To oversimplify it a bit, the previous code says that, when a user clicks on the *n*th thumbnail, the `active` class makes it active, and then makes the *n*th featured image active.

Now that you have the `active` class, another nice enhancement is to add a border to highlight the active thumbnail.

Update your `_page.scss` file (or its equivalent) to the following code:

```
.thumbnail {
  float: left;
  margin: 0 20px 20px 0;
  &.active {
    border: 3px solid $basecolor;
    margin: 0 14px 14px 0;
  }
}
```

The CSS code will look as follows:

```
.gallery-image-area .thumbnail.active {  
  border: 3px solid #cb790f;  
  margin: 0 14px 14px 0; }  
}
```

I added a 3 px-wide border and chose our theme's base color (choose whatever color works best for you, though). Since the border will make each thumb take up more space, I reduced the margins by a corresponding amount. It makes the thumbnail jump a little, but I don't mind because it gives the user some feedback. If you do mind, I encourage you to find a strategy that makes it not do that!

Summary

We've covered a lot of ground, yet again! In this chapter, we made a gallery overview and a gallery detail that will work equally well for devices ranging from mobile phones to desktops. We re-used some of 320 and Up's upstarts so that we didn't have to build responsive, three-columned layouts from scratch. The columns stack nicely on small screens and arrange themselves horizontally to fill the width on wider screens. We made a slightly modified hero for the **Gallery** page without having to write a ton of override styles and we even wrote some elegant JavaScript to make the **Gallery** detail page interactive. In the next chapter, we will build a page so site visitors can contact us.

4

Building the Contact Form

In *Chapter 3, Building the Gallery Page*, we built pages to show off our work. Hopefully, the quality of the work you show off on these pages is so compelling that site visitors will want to contact you to hire you for your amazing work.

Let's make the ability to do that easy and attractive!

Making a form plan

I know forms aren't exactly exciting, but we must get user info somehow, so we might as well make them look nice and not stodgy and cold. A clean and friendly form will be easy and minimal, gathering only the info we need. We also need to make the process of filling out a form as clear and free of frustration as possible. The 320 and Up framework is built to facilitate quite a bit of this, but we will still need to do the requisite planning to make sure it is just so.

Luckily, this isn't going to be too tough for our rather simple needs. Let's think about the bare minimum we need to collect in order to follow up with a potential client. Here are the things we need:

- The prospect's name
- The company
- The e-mail address
- The phone number
- A message

A few important things to keep in mind are to make sure that the labels for all fields let users know what to put in which field. I think that the most compelling argument for a usable form goes like this:

People read from left-to-right and top-to-bottom. Therefore, the label should appear above the input that it is describing, as the user will read the label first and then see the input. This of course is an assumption that our users understand the visual cues that define an input in a form. If we have users that don't know what form fields are, we are probably out of luck. That said, it's probably worth thinking about the fact that our user interfaces heavily rely on people understanding conventions!

There are other conventions we can use for our purpose on this form. A common one is the use of placeholders to show users examples of the kind of content that is expected in each input. Again, this convention is well-known to anyone who has been using the Internet for any significant amount of time. Hopefully, it could be a useful cue to someone less familiar with these conventions too.

Handling mandatory fields

The last things we need to let users know are the required fields that we want them to enter input into. There are two schools of thought on this; I will introduce both, mostly because I see the merit in both and it really depends on what you are doing.

One convention is to place the required * next to all the fields that are required. This convention again works for most visitors but the problem with this approach is that it might prevent gathering some information if we essentially annotate a few fields that we have as being optional. This argument basically claims that if we don't require a message and don't mark it as required, there is an increased likelihood of users skipping over this. Our form should absolutely require the user to submit a name and an e-mail; otherwise, we cannot respond at all. It's customary to not require a phone number or a message. Leaving the **Phone** field as optional is a courtesy in most cases for those who prefer not to be contacted via the phone. Leaving the **Message** field as optional is a courtesy to users. We don't want to make it mandatory as we can always respond once we have a name and e-mail, although our response will have to be really generic. It is helpful for our prospective clients to get a context for our next conversation. It saves everyone the time and energy.

With that in mind, I want to introduce the argument against following the convention of marking fields as required. Here is how the argument goes:

If we, as creators of the site, put on the form only the input fields that we absolutely need to follow up, then the form should be simple enough to not discourage a user. In our case, we have five fields, which is quite parsimonious. Then, by not marking any fields as required, we suggest that we want all the information we are asking for, but we don't actually require it. We can use form validation to then make sure we get the bare minimum, which in our case will be the name and e-mail.

Ultimately, these decisions take a lot of other things into account. Since our portfolio site is most likely to be for digital media work, our audience should be familiar with web conventions, and we can use that to everyone's advantage and present them with a clean, simple form. On other projects, you will certainly have to cater to different audiences or gather more data. Hopefully, walking you through my planning on this form will be of some help in the future decisions you make.

The form's layout

Ok, now let's move on to how we want this form to look at our breakpoints. This one will be easy because all our focus on this page is on getting some information from the prospective client. For this reason, we can get away with nearly having the exact layout from the mobile to the desktop.

Here's the mobile's layout:



A vertical form layout for mobile devices. It consists of five input fields stacked vertically, each with a label above it: 'Name', 'Company', 'Email', 'Phone', and 'Message'. The 'Message' field is a larger text area at the bottom.

And here's an example of a layout wider than 992 px:



A horizontal form layout for wide screens. It consists of five input fields arranged horizontally, each with a label above it: 'Name', 'Company', 'Email', 'Phone', and 'Message'. The 'Message' field is a larger text area at the bottom.

It's pretty hard to tell them apart! I know, the input fields are going to get unnecessarily wide if someone has this form open at full width on his/her brand new Thunderbolt. But really, we don't need or want any other content on this page to interfere, so we will make a small effort to still make the page look pleasing.

Ok, enough of talk! Let's write some code.

First, let's put one of those small hero areas just above the form. I always like the opportunity to add a human touch to things that I create on the Web, so just under the header, let's place this markup:

```
<!--hero markup -->
  <div class="hero subhead">
    <div class="container">
      <h1>Say Hello!</h1>
      <p>I just met you and this is crazy. Leave your number, I'd love
to work with you.</p>
    </div>
  </div> <!--end hero markup -->
```

You should probably put your own message in there, but you get the idea.

After that, let's put the markup we'll need for the form. The markup we'll be using here goes after the hero:

```
<!--form --> <div class="full row clearfix">
  <h2 class="h2">Hello! Is it me you're looking for?</h2>
  <p>Reach out to me for your new projects.</p>
  <form method="post" action="#" class="contact">
  <p>
  <label for="name">Name</label>
  <input id="name" name="name" placeholder="Firstname Lastname"
type="text" required/>
  </p>
  <p>
  <label for="company">Company</label>
  <input id="company" name="company" placeholder="Widgets Inc."
type="text"/>
  </p>
  <p>
  <label for="email">Email</label>>>
  <input id="email" name="email" placeholder="firstname@somename.com"
type="email" required/>
  </p>
  <p>
  <label for="phone">Phone</label>
```

```
<input id="phone" name="phone" placeholder="123-456-7890" type="tel"/>
</p>
<p>
<label for="message">Message</label>
<textarea id="message" name="message"></textarea>
</p>
<p>
<input type="submit" class="btn btn-primary btn-extlarge" value="Send
It!" />
</p>
</form>
</div>
<!--end form -->
```

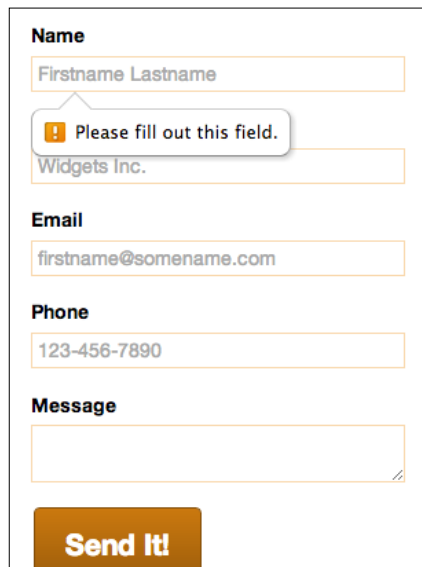
This markup is mostly straightforward, but I have used markup that is slightly opinionated and is good practices. First, you will notice that I have not supplied a value for the `action` parameter for the form that is used to post form data to a server. I will leave that to you, as we will not be making a backend to handle this data (alternatively, you can use one of the many nifty services out there that will handle contact and e-mail forms for you).

Moving down the code, you will notice that I have wrapped every label and input pairing in `<p>` `</p>` tags. This isn't uncommon but it is an opinionated way to handle the laying out of the form. I prefer not to style inputs and form controls if I can avoid it. For sites that grow, these can lead to a lot of work that isn't reusable. You can eliminate or reduce this by relying on styling some elements that wrap the form controls. As always, keep these elements semantically appropriate. I would argue that a label and an input form something of a paragraph, as they share the same subject and are a break in the subject of the content that follows.

Input label magic

Also, especially for mobiles, always take advantage of the `for` attribute in the label that works only if you set the value of that parameter to mirror the value of the ID of the input you want to associate with it. In other words, if your label is the **E-mail** input, give that input an ID of `email` (`id=email`) and set the label to `email` as well (`name=email`). This practice is not just semantic, otherwise, I probably wouldn't bother. Once you have paired an input and a label in this way, some magic happens. The label now gets magical powers—when a user clicks or touches the label, the input it is paired with will get focus. This standard has been around long before the practice of browsing the web with touch interfaces was common, but what an awesome feature for touch! Now, users with fat fingers, shaky hands, or imprecise movements will be more likely to hit their mark. If you never knew this, test it out. If you knew about it already, I hope you skipped this paragraph; time is precious.

I still have a few more things to point out. I have placed the `required` attribute on all fields that I want to be mandatory. This attribute is new to the HTML 5 spec and does some nice magic under the hood. We will need to make a fallback for browsers that don't support this feature, but you can temporarily enjoy the fantasy that HTML 5 will make your work as a web developer easier than it was before (don't worry, you still need to write some JavaScript to help validate this form). Still, this feature will be a time saver once you get to the point where nearly all your potential users are using modern browsers (the question is, when will that be?). Anyway, go ahead and try it out. Start up a simple server, such as the Python Simple HTTP server, and visit your contact form in its current state. Don't bother filling out the form, then hit **Send It**. If you are using Chrome, you will get a nice validation error message, **Please fill out this field**, in a tooltip:



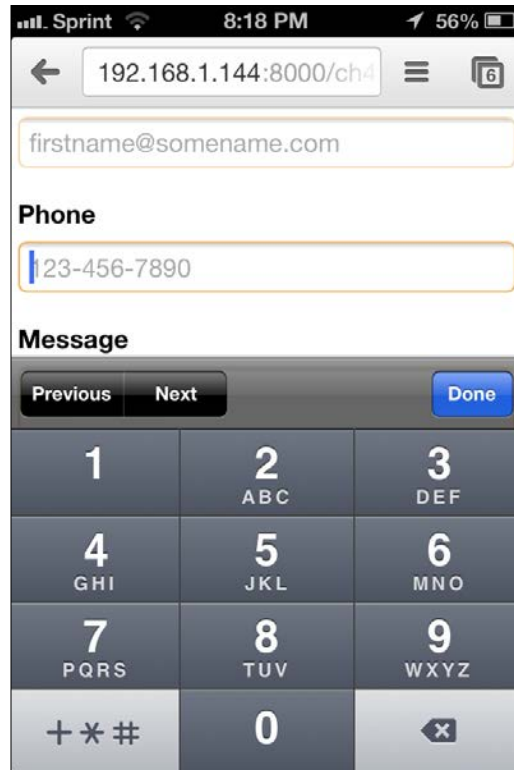
The image shows a contact form with the following fields and a validation error:

- Name**: A text input field containing "Firstname Lastname". A tooltip with an orange exclamation mark icon and the text "Please fill out this field." is positioned over this field.
- Company**: A text input field containing "Widgets Inc."
- Email**: A text input field containing "firstname@somename.com".
- Phone**: A text input field containing "123-456-7890".
- Message**: A text area field.
- Send It!**: A large orange button at the bottom of the form.

Modern versions of other browsers (Firefox, Safari, and Internet Explorer) will do something similar. Try it on your mobile too; it's pretty nice!

Okay, the fun will be over soon enough, as we will need to make fallbacks for browsers that don't support this feature. But we still have a few fun enhancements to add, thanks to the HTML 5 spec. Next, I'd like to point out the absolutely painless enhancements you get with some HTML 5 form field attributes. You will notice that the input for an e-mail is set to `type="email"`. That attribute gets you two kinds of special sauces. On both the desktop and mobile (in browsers that support it, of course), you get the kind of validation for e-mail addresses that we've been writing in JavaScript for years. It looks for an @ and all that. On a mobile, it should open a soft keyboard that features a prominent @ as well.

The other field we use with a nifty attribute introduced in HTML 5 is the `type="tel"` attribute. The only benefit to this is that on mobiles, it will pull up a numeric keyboard rather than an alpha keyboard.

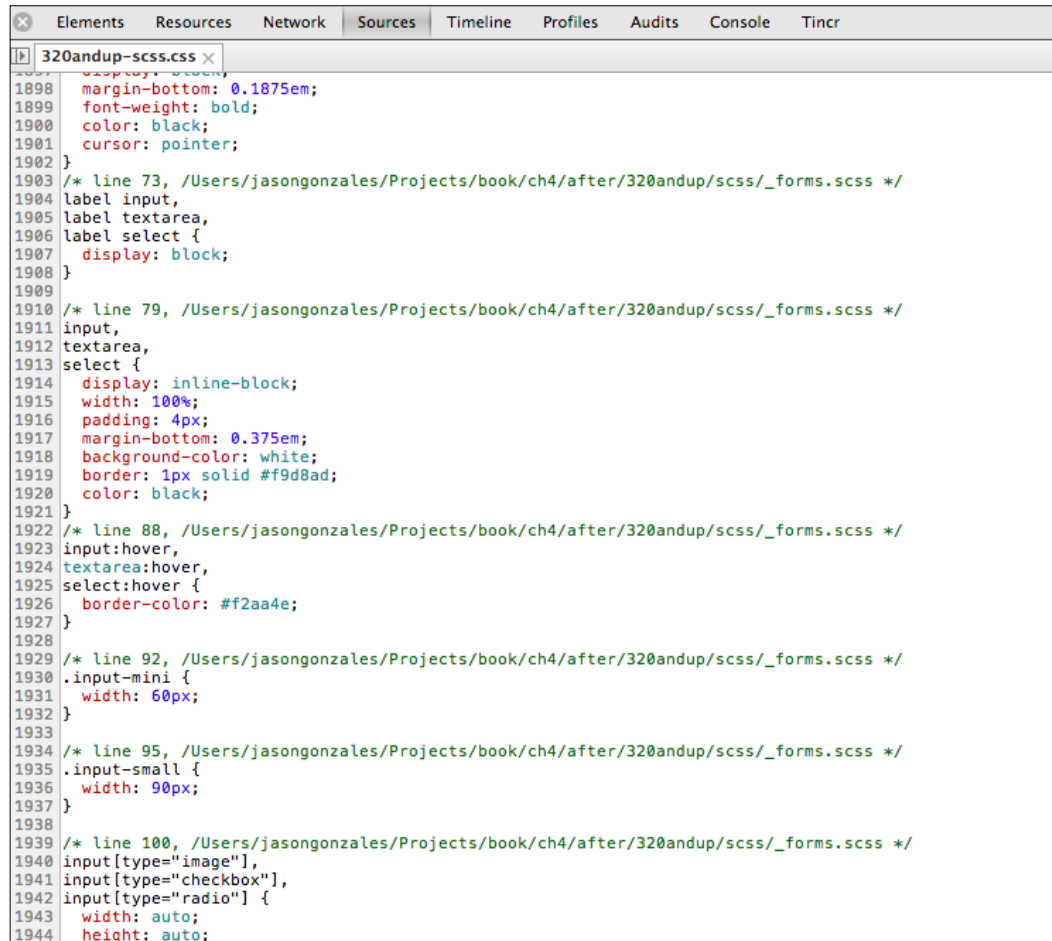


This is just a really nice thing to do for those poor souls filling out a form on a small screen. Your users will thank you.

Now, let's add the minimal styling that we will need to get this to look consistent with the rest of our apps. Everything looks pretty great; the only exception is that the borders on the inputs are orange. If you are using SASS, and your compiler writes the line numbers of the styles, this is really easy to debug. I've been encouraging you to use SASS all along, but one thing I really like is the ability to print the line numbers of all the style selectors while you are in development. You really need to compile the compressed CSS for production, but for development, always switch over to code that is friendly for debugging. I use CodeKit for this and for a handful of other reasons. I should add that I generally use open source, Command-Line Tools. For example, I use `tmux` and `vim` to write code, not a standalone text editor. But Codekit has so many useful features that are effortless to configure and I am really addicted to it. I only wish it had a command-line version.

CodeKit makes my whole day easier when I have to solve problems in CSS.

Here is what I see in Chrome development tools when I look at what is going on with these wacky orange borders:



```
320andup-scss.css x
1898 margin-bottom: 0.1875em;
1899 font-weight: bold;
1900 color: black;
1901 cursor: pointer;
1902 }
1903 /* line 73, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1904 label input,
1905 label textarea,
1906 label select {
1907   display: block;
1908 }
1909
1910 /* line 79, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1911 input,
1912 textarea,
1913 select {
1914   display: inline-block;
1915   width: 100%;
1916   padding: 4px;
1917   margin-bottom: 0.375em;
1918   background-color: white;
1919   border: 1px solid #f9d8ad;
1920   color: black;
1921 }
1922 /* line 88, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1923 input:hover,
1924 textarea:hover,
1925 select:hover {
1926   border-color: #f2aa4e;
1927 }
1928
1929 /* line 92, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1930 .input-mini {
1931   width: 60px;
1932 }
1933
1934 /* line 95, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1935 .input-small {
1936   width: 90px;
1937 }
1938
1939 /* line 100, /Users/jasongonzales/Projects/book/ch4/after/320andup/scss/_forms.scss */
1940 input[type="image"],
1941 input[type="checkbox"],
1942 input[type="radio"] {
1943   width: auto;
1944   height: auto;
```

I see that the border properties are defined at line 79 in `_forms.scss`. Super helpful, yes?

Unfortunately, that is not the end of the story. When I get to that line of code, here is what I see:

```
input,
textarea,select {
  display : inline-block;
```

```
width : 100%;
padding : 4px;
margin-bottom : $baselineheight / 4;
background-color : $inputbackground;
border : $inputborderwidth $inputborderstyle $inputborder;
color : $textcolor;

&:hover {
border-color : $inputhover; }
}
```

Right off I notice two things. The border color is defined with the variable `$inputborder` and the hover color for that border is defined with `$inputhover`. In my opinion, these are poorly named variables as they are not semantically accurate, but to be fair, I've done worse in the past. At any rate, if I could pick an improvement here, it would be to name these variables something in order to point out the fact that they are actually variables for color—something such as `$inputbordercolor` and `$inputborderhovercolor`. Sure, those are long names, but they are precise.

Okay, moving on. We need to go to the `_variables.scss` partial to see what is going on. Why are these borders orange, for goodness sake? Don't panic, help is on the way. Going into the `_variables.scss` file, I do a quick search for `$inputborder` and here is what I see:

```
$inputborder : lighten($basecolor, 40%);
```

Let's think about what is going on in the code. For many designs, using the base color for input borders can create a harmonious design. But, in my example, I have chosen a yellowish orange color, which makes for pretty low-contrast borders. To be honest, they look annoying to me; I can't imagine what they look like to someone who has visual challenges. But, furthermore, I would guess that 90 percent of the time, I want my input borders to be some shade of grey. Why? Well, with something as critical as creating a form, I want to make sure that the fields are well defined with a high-contrast color, and with a white background, the greatest contrast comes with black. If #000 black looks too stark, we can always choose a darker shade of gray that is near to black. At this point, I think it's best for this design (and perhaps others in the future) to go ahead and redefine this variable as some shade of gray.

Let's try this:

```
$inputborder : $lightgrey;
```


I actually experimented with all the greys defined in this variable file, and I prefer the lighter grey. It helps the rows look more organized. Another thing you may have noticed by now is that the border still changes to an orange color when you hover over the input. Let's make that a darker grey instead. Similar to what we followed previously, you will notice that the `input:hover` style is defined in the `_forms.scss` file. It looks like this:

```
&:hover {
  border-color : $inputhover; }
```

So now we go to the `_variables.scss` file to redefine `$inputhover`!

Let's make it like this:

```
$inputhover : $grey;
```

Looks good!

We just need a few more things to tune up the styles on this page. Let's make the inputs look nicer and more consistent in terms of how inputs are rendered in a mobile browser. You will notice in the previous screenshot on the form of an iPhone (or on your own mobile, if you are checking your work there) that the inputs automatically get rounded corners on the mobile. Let's set a style to do that in our forms page.

I want to change all inputs across the site, so I am going to go and edit the `_forms.scss` file.

Update this style:

```
input,
textarea,
select {
  display : inline-block;
  width : 100%;
  padding : 4px;
  margin-bottom : $baselineheight / 4;
  background-color : $inputbackground;
  border : $inputborderwidth $inputborderstyle $inputborder;
  color : $textcolor;
  @include rounded(6px); //this is the new bit

  &:hover {
    border-color : $inputhover; }
}
```

While using `rounded` mixin, I like 6 px, but feel free to round it to your taste. One more thing that I'd like to change with these inputs is the padding. Having a big target is nice, but they could also do with a little breathing room between the words and the border of the input. Just below that `rounded` mixin, let's add some padding:

```
padding: 10px;
```

Looking much better!

The last styling task we have on this page is to contain the width of the form. Let's keep the form from getting any wider than 992 px and keep it centered.

We can do that without having to use any @ media queries actually. Let's go back to the `site.scss` file and add a style that will work the same if we want to reuse it:

```
.row {
  max-width: 992px;
  margin: 0 auto;
}
```

This does exactly what I described previously. This is actually a great example of how to think about being responsive without necessarily relying on newer standards.

Okay, now the last thing we need to do is go hook up the validation that will work for browsers that don't yet support the HTML 5 `required` attribute.

JS validation fallbacks

Well, we could write all our fallbacks. Knowing how to write fallbacks is super useful but that is beyond the scope of this book. Also, there is a really awesome way to make fallbacks that have been made already. It is called **webshims** and you can find it here: <http://afarkas.github.io/webshim/demos/index.html>.

This library makes it super easy to take advantage of a lot of HTML 5 features without writing a ton of support for older browsers. In our case, we will have to do very little to support the HTML 5 validation in our form.

Download the lib from the site I listed previously. Once you do that, copy the `js-webshim` folder to your project. I've already done that in the `after` folder for this chapter.

Now, we need to do two more things and we will be good to go.

Include the `polyfiller` script from the `webshims` lib at the bottom of the `contact.html` page:

```
<script src="js/js-webshim/minified/polyfiller.js"></script>
```

You must put this after `jQuery` but before the scripts you write.

Now in `script.js`, add this line to instantiate the `polyfiller` script:

```
$.webshims.polyfill();
```

I have put this inside the `ready` function to make sure that all the form elements are present in the **Document Object Model (DOM)** before it fires.

Now, we're done polyfilling our form validation and it should work in browsers that don't support HTML 5 validation. Enjoy!

Summary

So, in this chapter, we planned our way to a much simpler layout than any of our other pages, but for good reason. No one likes filling out forms much, but if we can keep the noise down on pages with forms, we can encourage users to give us the information to better facilitate communication. Or at the very least, we won't discourage people from filling out our form.

Probably, the biggest challenge here is the cross-browser support for client-side validation. Until it is known that the majority of users use modern browsers, we still need to shim and polyfill, but as we saw, well-written code makes that fairly easy too, unless our requirements are complex.

Next, let's move on to the **About Me** page.

5

Building the About Me Page

In the previous chapter, we built a form potential for the clients to contact us. This is just one part of the formula of the current model for a portfolio site, either for personal business or for any other business use. The last aspect of this convention is the **About Me** page. This is arguably the least important page I have often seen and have had to consult with clients who do pretty weird stuff in this corner of their websites. That said, I think it is usually well-intentioned.

But before we get back to designing and coding, I want to put forward an argument for and against the **About Me** page.

Justifying the About Me page

Everyone wants their website to meet what can seem like two conflicting goals, which are as follows:

- Goal 1: We are professionals. Here is our work and our track record. We are different from our competitors.
- Goal 2: We are just people! And I am just like you! We have warm blood flowing through our veins just as you do!

So, rhetorically speaking, this actually makes a lot of sense, right? You want to make an emotional connection with your audience. This is called pathos in the study of rhetoric. If you don't make an emotional connection, the likelihood of getting a client or a sale decreases for sure. But, believe it or not, I just genuinely enjoy connecting with people, hearing about their experiences, and I really enjoy helping them solve problems. Furthermore, I find it gratifying to communicate (whether via the web or by any other means) in ways that convey my warmth and humanity (hope I'm right about that one). It's pretty awesome that I get to make money while doing it.

I am not a rare breed. Despite some silly stereotypes, I find that web developers are very social people who also enjoy technology. So, why is it that so much of the software we make lacks personality and a human touch? No idea. But I am here to argue for more of it. As a user of any technology, anything that is designed to acknowledge my humanity, and the actual conditions of my living, makes me happy. They can give you a moment of delight or deeply gratifying experiences.

All of this is to say that the **About Me** page is a nice place to shed light on yourself (either individually or collectively) and is but one strategy for connecting with your audience. But if you haven't connected with your audience by the time you get to the **About Me** page, in the current vernacular "ur doin' it wrong". And if your clients ask you to do this, you are still doing it wrong. To me it's important to imbue a site with the right tone as well as to make the site really usable. For a portfolio site, this is deceptively easy because the conventions are pretty mature. And for this book we've just used the conventions to make the **About Me** page quickly. But ultimately, this is just a framework. It's up to you to be creative without being overwrought, by adding surprising things, or maybe just one surprising memorable thing to your site. Hopefully using a fast, easy framework gives you more time and brainspace to do this. My worst fear about frameworks is that they facilitate rapid production of sites that are all nearly indistinguishable.

So, my real point is, do have an **About Me** page, do it well, but make sure the whole site is also about you. It takes more work, and there are risks, but ultimately I think it's worth it. The other thing is this book isn't really about this; I am not going to teach you how to create the **About Me** page though this is the main topic of this chapter. That really warrants the attention of a whole book, not to mention a book that is less technical. So I apologize in advance that this chapter is going to be mainly about how to lay this page out.

With that said, let's get started on designing this page. The kind of content we will need on this page will be quite simple. It will include the following:

- Some details about the services we're going to offer.
- Images of the company members. We will assume a small number for this scenario but include a strategy for dealing with a larger number. Obviously if this is just you, you're only going to need one photo. Make it good; some text for bio(s), something snappy, but also meaningful.

Making the wireframes

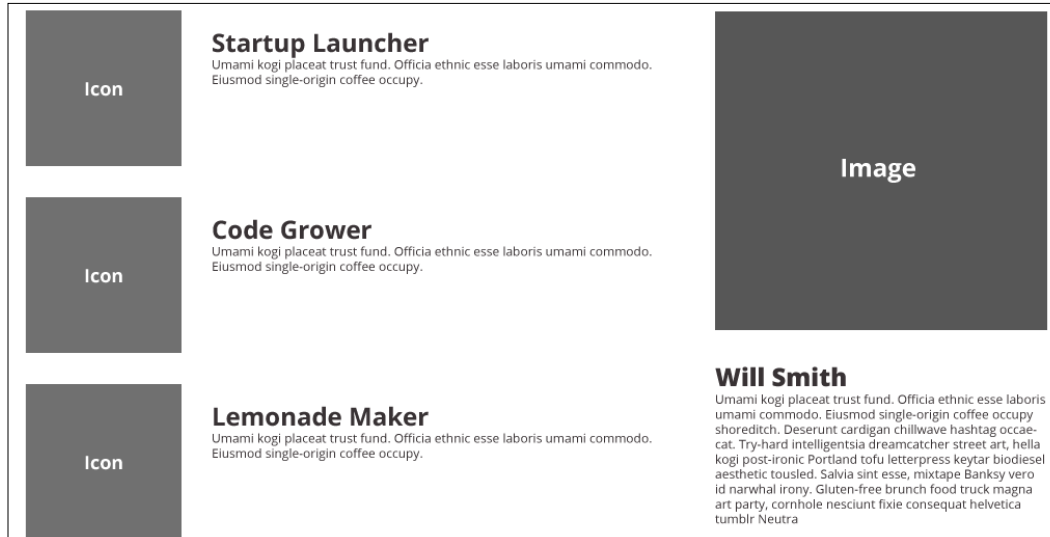
Let's take a look at some wireframes for this page. For these examples, I am going to assume that this is a portfolio site for a one person. But if you had a small group, you would probably want to reconsider this layout. We will discuss more on this later. For now, here are the wireframes for the mobile view. First, the top part of the page (under the marquee, which will remain consistent):



Notice that we have three services listed and stacked. We're going to use the set of included icons to make big, eye-catching icons to the left of each service/skill described. Below the icon we will put the obligatory headshot and bio:



Now here is how we will use the same content on a wider desktop layout:



We'll use the wider layout of this page to put some of the content on the sidebar. This has the benefit of letting users see more content without scrolling.

The markup

Let's start with the markup for this page. This page will have a special technical challenge, since neither does 320 and Up ship a way to make this kind of sidebar, nor was it really meant to. Other frameworks such as Bootstrap and Foundation have that facility. We will roll our own solution though. No need to introduce an entire framework to solve this single problem. With that said, I will be taking a cue from how these frameworks do this.

To work this problem through, let's start off with some pretty basic markup for the mobile layout. First, let's add another, our last, marquee (don't shed a tear, though; you can make as many as you want in the future).

```
<!--hero markup -->
<div class="hero subhead">
  <div class="container">
    <h1>Let's Talk About Me.</h1>
    <p>Read on to learn about my special powers.</p>
  </div>
</div>
<!--end hero markup -->
```

Hopefully this all makes sense by now. Now let's add our first bit of markup for the main area, the one that describes all the stuff we have to offer. We know we need the following things:

- A wrapper for all the content, so we can pad and set width on the content of the page as necessary for various screen widths
- A wrapper for the main content and the sidebar content, for similar reasons as the previous point
- Various containers and markup for all the actual content

So here it goes. Let's put this markup below the marquee:

```
<!--main content -->
<div class="full summary">
  <div class="main-content">
    <div class="content-item">
      <div class="circle pull-left">
        <i class="icon-fire big-icon"></i>
      </div>
      <div class="content-body">
        <h4 class="content-heading">Startup Igniter</h4>
        <p>Umami kogi placeat trust fund. Officia ethnic esse
          laboris umami commodo. Eiusmod single-origin coffee
          occupy.</p>
      </div>
    </div> <!--end content-item -->
    <div class="content-item">
      <div class="circle pull-left">
        <i class="icon-leaf big-icon"></i>
      </div>
      <div class="content-body">
        <h4 class="content-heading">Code Grower</h4>
        <p>Umami kogi placeat trust fund. Officia ethnic esse
          laboris umami commodo. Eiusmod single-origin coffee
          occupy.</p>
      </div>
    </div><!--end content-item -->
    <div class="content-item">
      <div class="circle pull-left">
        <i class="icon-lemon big-icon"></i>
      </div>
      <div class="content-body">
        <h4 class="content-heading">Lemonade Maker</h4>
```



```
        <p>Umami kogi placeat trust fund. Officia ethnic esse
          laboris umami commodo. Eiusmod single-origin coffee
          occupy.</p>
      </div>
    </div><!--end content-item -->
</div>
  <div class="sidebar-content">

    </div>
  </div>
<!--end main content -->
```

We've re-used the `.full` class that applies the same styles on all our main content, which is mainly padding and margin. I've created a lot of new classes too. Right now there are no styles for them but let me explain my thinking before I go ahead and style them.

The next container after `.full` is the `.main-content` container. It will not do much in the mobile view, but as the layout gets wider we will assign a size to it and float it so there is room to float the `.sidebar-container` to the right (spoiler alert).

Within the `.main-content` container, we are going to have three chunks of content consisting of an icon, a heading, and some text. I gave each chunk the class `.content-item`. Inside the class, there is a `div` tag that I will change into a circle (well, for modern browsers anyway) that will frame icons. That is followed by the `content-body` `div` class, which will hold a heading and a short blurb about my special prowess that I want to describe. Rinse and repeat the `content-items` class as much as you like, but I like sets of three. Three is a magic number after all.

Notice that I use the `icon` classes that are in the framework. They utilize the `font-awesome` font. They are super easy to implement and are super flexible as you will see when we start to style them. All you need to do to get them to appear in your markup is add the appropriate class to your markup. I add these classes to the `<i>` tag, which is something of a convention, but you can just easily use them on a `` tag, `<a>`, or whatever as long as the markup makes sense. You will notice that I also add the class `.big-icon` on each one. That is because I anticipate that I will need an additional style to make these big and add some other styling for larger layouts. Next, let's take advantage of the set of icon fonts that ship with 320 and Up.

Awesome icon fonts

For future reference, take a look at the `_font-awesome.scss` (or similar) file and you will see a list of all the icons created by the styles to facilitate the use of the `font-awesome` icon fonts. These are just prerolled for us but if you need a new icon that has been added to `font-awesome`, you will need to add it to this list (or one of your own making). You will notice that the icon is actually specified with the `content` attribute. For example, the lemon icon CSS looks as follows:

```
. icon-lemon:before { content: "\f094"; }
```

This is because the icon is specified with the Unicode character `F094`. You can look this up at <http://fontawesome.github.io/Font-Awesome/icons/>. Just click on each icon to get to know a little more about each one.

While we're here, let's take a quick look at what else is going on to support these icon fonts. At the top of the `_font-awesome.scss` sheet, you will see that anything that has a class beginning with `icon-` gets some style by default. That is specified with this style:

```
[class^="icon-"],
[class*=" icon-"] {
  display : inline;
  width : auto;
  height : auto;
  line-height : inherit;
  vertical-align : baseline;
  background-image : none;
  background-position : 0 0;
  background-repeat : repeat; }
```

The previous code targets anything beginning with or containing `icon-`. This is done with the regex `^` and `*`. Regex is the shorthand for regular expressions. Regular expressions are a utility for searching strings; they use various symbols to accomplish this task. Regex is a huge topic beyond the scope of this book but just know that, by using these symbols in CSS, the CSS engine searches selector strings in your markup. Not all the available regex symbols can be used in CSS, but the use of `^` and `*` can be quite powerful.

Keep reading and you will see that there are additional styles applied if you use these classes on the `` or `<a>` elements. There are even special styles defined for putting elements inside buttons or the `` tags. We won't be using these but please do experiment with them.

If you go ahead and refresh your page, this won't look too great; so let's get on with styling this for the 320 and Up layout. The first thing I want to do is make the circles that will hold our icons. I want all these circles to be of the same size, so I am going to set a uniform size as well as a few other styles that I will explain in a moment:

```
.circle {
  background: #FFA500;
  @include rounded(28px);
  height: 56px;
  width: 56px;
  position: relative;
}
```

I actually arrived at these dimensions through a little trial and error but I won't bore you with that. The border radius is set to half of the height and width of the element. That is the recipe for making a circle. Obviously, someone using an old browser will get a square. If you're not okay with that, you can put some kind of `polyfill` or fallback in place. Notice that I used the mixin to make the vendor-specific border radius, but if you're using plain CSS you will need to type all of these.

Lastly, I set the position to `relative` on these icons so I can absolutely position each icon within the circle. They all have different dimensions, so they will each get a unique position to accommodate that.

Let's move on to styling and positioning them. Put these styles below the `.circle` style:

```
.big-icon {
  font-size: 2em;
  color: #FFF;
  text-shadow: -1px -1px #999;
  position: absolute;
  &.icon-fire {
    top: 15px;
    left: 18px;
  }
  &.icon-leaf {
    top: 16px;
    left: 13px;
  }
  &.icon-lemon {
    top: 15px;
    left: 16px;
  }
}
```

The previous code is the SCSS code. Here is what regular CSS looks like:

```
.big-icon {
  font-size: 2em;
  color: #FFF;
  text-shadow: -1px -1px #999999;
  position: absolute; }
.big-icon.icon-fire {
  top: 15px;
  left: 18px; }
.big-icon.icon-leaf {
  top: 16px;
  left: 13px; }
.big-icon.icon-lemon {
  top: 15px;
  left: 16px; }
```

The `.big-icon` styles just make the icons bigger, white and with a little shadow that makes them look like they've been subtly embossed into the circle. Kind of a cool effect I think. I also set the `position` to `absolute` here for all the icons. I arrived at all the positions of the icons through a combination of math and eyeballing it.

Here's the math method. Looking at the icons in the developer tools in my browser I get the dimensions. Using the fire icon, for example, I see that it measures 22 x 25 pixels. So to get the left positioning, I subtract the width of the icon from the width of the circle, that is, $56 - 22 = 34$. Divide that by 2 to get the left position because the left position is on the top-left of the icon, so we need the distance from the left edge of the icon to the center of the icon. This gives us a left position of 17 px. But then I eyeballed it and liked 18 px better, go figure. Rinse and repeat for the remaining icons.

Now, we need to position the text to the right of each icon. Here's how that will look:

```
.content-body {
  overflow: hidden;
  .content-heading {
    margin: 0 0 5px;
  }
}
```

This is pretty simple. The only thing that may seem weird is the `overflow: hidden` business. All that it does is make sure the text stays in a tidy little box, rather than flowing around the icon `div`. To learn more about this, you should consult Google, but you can refer a fabulous article at <http://alistapart.com/article/css-floats-101>.

Go ahead and refresh the page and take a look. Looks pretty great, but the icon is too close to the text. Let's fix that. Add the following code to your circle styles:

```
.circle {
  margin-right: 12px; /* this is the new bit */
  background: #FFA500;
  @include rounded(28px);
  height: 56px;
  width: 56px;
  position: relative;
}
```

Ah! Now it looks good! Let's move on to adding our picture and bio. Here is the markup we will need:

```
<!-- sidebar content -->
<div class="full bio">
  <div class="sidebar-content">
    <div class="image-container">
      
    </div>
    <div class="bio-container">
      <h3>Will Smith</h3>
      <p>Seitan gastropub jean shorts DIY, shabby chic scenester
      flannel umami. Keffiyeh freegan small batch Neutra before
      they sold out, literally salvia 8-bit. Flannel trust fund
      swag Austin, locavore sustainable irony. Fingerstache pop-
      up readymade Schlitz try-hard. Roof party 3 wolf moon
      forage Schlitz, butcher squid Pinterest cardigan seitan.
      Cray YOLO helvetica, cliché tattooed single-origin coffee
      selvage food truck gastropub. Disrupt McSweeney's ugh put
      a bird on it.</p>
    </div>
  </div>
</div>
<!-- sidebar content -->
```

Notice that we are using the large placeholder image we used for the gallery page, but of course you need to use your own favorite glamor shot. Speaking of the gallery page, we will need to use some similar styles here to get the image and text to look right. Since we have re-used the `full` class on both sections, there is an appropriate amount of space between the bio content and the edge of the viewport. We just need to put a margin below the image to push that header down. Add this style:

```
.image-container {
  margin-bottom: 2em;
}
```

Feel free to adjust the margin according to your tastes.

Take a look at this layout at 320 px wide and everything is hunky-dory. Let's get this to layout as two columns when the screen is 992 px wide or greater. Go ahead and resize your browser to 992 px (or just look at it on a tablet or something). It looks pretty weird. Luckily, all we should need to do is assign the appropriate percentage widths to these and float them. Try adding these styles:

```
.summary {  
  width: 55%;  
  float: left;  
}  
  
.bio {  
  width: 35%;  
  float: right;  
}
```

We can use the classes `summary` and `bio` on each block of content now. Go ahead and refresh and you will notice that the footer is now trying to squeeze into the tiny area between the columns. Easily fixed. Just add this style to your footer styles in the `_site.scss` file:

```
clear: both;
```

Fixed!

Ok, that is all!

Summary

In this chapter, we learned to use icon fonts and style them so they look every bit as good as bitmap images; however, they are far more flexible since we can resize, color, and add simple effects such as shadows via CSS. We also whipped up a custom layout very quickly to accommodate our content needs. Awesome! Now go out there and use what we've done to build great stuff for yourself and your clients!



Anatomy of HTML5 Boilerplate

This appendix is to help those who have no experience with HTML5 Boilerplate. If you know all about it, there is no need to read any further (thus the reason this is an appendix). But if you are new to HTML5 Boilerplate, this appendix will help you get started with 320 and Up with a deeper knowledge of what is going on. In this appendix, we will look at the structure and choices of HTML5 Boilerplate and understand the implications of its choices to the further web pages you may develop.

What is HTML5 Boilerplate?

First of all, you can find the home page for the project at <http://html5boilerplate.com/>. The site offers a quick overview of HTML5, but does not provide much context to why HTML5 Boilerplate is useful and why it was created at all.

I won't go extensively into the history of HTML5 Boilerplate. This book is mainly focused on how to do stuff, not how stuff happened, so I will try to explain just enough of the background to let you know how it works, with the hope that your future use of HTML5 Boilerplate with 320 and Up (as well as any other framework) will be with enough understanding to help you solve future problems.

In essence, HTML5 Boilerplate was a project started with the intention of creating an HTML page that had all the components one would need to make an effective, cross-browser web page; it also utilizes all the goodness available in modern browsers that support the modern HTML5 specification.

If you want to know more about what I mean by HTML5 specification or what the difference is between a modern browser and an old browser, then I encourage you to do some searching and reading. Any links I leave for you are at the risk of being out-of-date soon. But, briefly, HTML5 is a specification for what browsers should do. It is an effort to make all browsers support the same or similar features so as to make it possible for web developers to make a web page and have it behave the same in all the browsers. This is not a reality yet, nor do I think it will ever be; to be fair, though, things are a lot better than they used to be.

So, currently, as a web developer, in most situations, you will most likely have to support, at a minimum, **Internet Explorer (IE)** Version 8 and up, Firefox 4 and up, and the current release of Chrome (Version matters less with Chrome, since it has always encouraged users to update). Note the challenge here. The current versions of all the browsers, except Chrome, are higher than those you must support. Also, chances are, if you are making a site at the time of writing, you may actually need to support older versions of these browsers as well as a few more browsers, depending on your user base. And those are just the desktop versions; forget about all the mobile versions of those browsers as well as the Android flavor of WebKit, which has a bazillion versions out there.

Knowing this, you can see why I and so many other frontend engineers evangelize for simplicity in frontend design. A simple design has a high likelihood of giving a good experience to the end user as well as something approaching consistency.

But, you don't get that experience and consistency without some effort. HTML5 Boilerplate goes a long way towards making that effort on your behalf. Let's walk through it for a few pages to better understand how it works. As you read along, I suggest that you pull up the `index.html` file from the `before` directory in *Chapter 1, Mobile First – How and Why*. This is the boilerplate version before we added anything to it. Let's start at the top!

Conditional comments

The code for conditional comment looks as follows:

```
<!--[if lt IE 7]><html class="no-js lt-ie9 lt-ie8 lt-ie7"
  lang="en"> <![endif]-->
```

This code is nested inside comments but has some conditional logic. Only Internet Explorer has the super powers to use this logic. The previous example only renders the HTML code inside the comments if the browser executing it is a version less than IE7, thus the syntax `[if lt IE 7]`. You can infer that different HTML tags are rendered based on the version of IE. You can use the classes in this HTML tag to make special styles that are typically necessary to deal with the shortcomings of these older browsers.

If you jump ahead a bit, you will see the following conditional comment:

```
<!--[if (lt IE 9) & (!EMobile)]>
  <script src="js/selectivizr-min.js"></script>
<![endif]-->
```

This includes a JavaScript library included in the 320 and Up framework that allows CSS3 selectors to work in browsers older than IE9.

Many, many mobile meta tags

So next, within the head tags, you will see some meta tags that are used by vendors to perform a magic trick. The first one to take note of is as follows:

```
<!-- http://t.co/dKP3o1e -->
<meta name="HandheldFriendly" content="True">
<meta name="MobileOptimized" content="320">
```

You can visit the link provided in the comment if you want further details on meta tags. I encourage you to do so.

To summarize, these meta tags help mobile browsers know that they can render a page that isn't only intended for desktops. Get used to this as there is a lot more of this kind of thing coming ahead.

After this, there is a bunch of comments that allow you to create icons for the Apple devices. These icons are cool if a user wants to create a shortcut for your website. It creates an icon for your website just like an app would have. If you want to take advantage of this, you need icons of all the dimensions, and you need to either place them in the path already specified or edit the path so it loads your icon files.

There are still more Apple-specific meta tags. For many, as you can see, you just need to fill in the blank field (for example, `apple-mobile-web-app-title`).

```
<meta name="viewport" content="initial-scale=1.0">
```

The previous line of code makes sure that the page doesn't get zoomed in as long as you have the `content` field in the next code line set to `yes`:

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

In the next section, the one labeled `startup images` gives your web page more app-like functionalities. When users launch your site from their home screen, these images will fill their screen until the page loads. Again, you will need to provide images of all the dimensions listed and put them in the correct path. However, you should know that this particular block of markup can potentially be removed and applied with cleaner code. You will learn about this in a moment when we go over the `helper.js` file together.

Mercifully, the next set of tags for Windows 8 adds almost no new work! This tag can and should share the icon you created for Apple, and will appear in those nifty Windows 8 tiles. You can set the color of your tile in the following tag:

```
<meta name="msapplication-TileColor" content="#000">
```

Please, please, I beg you. Set it to hot pink.

Whew! We're done with the head!

Now, you can skip to the bottom of the file.

Including the scripts you'll need

The rest isn't terribly surprising. We include the following scripts below the footer, and in order:

- jQuery (from a CDN first, then locally as a fallback)
- A plugins file
- A script file
- A helper file
- Google Analytics

There is no problem with making your site this way but I often combine plugins, scripts, and helpers into one file.

The `helper.js` file should come before the script file; otherwise you can't call the functions in it. If they are correct, just rearrange them and you'll be good to go.

The last thing is I want to give you an overview of the helper file, since that is a part of the HTML5 Boilerplate.

The helper.js file

The code in this file is really helpful. The comments in it explain what each function does for the most part. Nonetheless, I'd like to highlight a few things and make sure you know how to implement them.

Basically, to call any function in here, just find one in the file you need (or just want to try) and put `()`; after it. That is JavaScript's way of executing a function. For example, the first usable function defined in this file (you can tell that they are functions because they have the word `function` in them) is as follows:

```
MBP.scaleFix = function() {...
```

If you want to use this function, just add this to your script file:

```
MBP.scaleFix();
```

Et voilà! You just called this function. Now let's go through a quick rundown of what the most useful functions in here do. Keep in mind that many of the items in this script are used by the functions themselves; so, if you try to use them, they might not do anything, especially the ones that don't have the word `function` in them.

- `MBP.scaleFix`: This function stops an annoying bug that happens in iOS. This bug manifests itself when a user rotates the phone from portrait to landscape. In landscape, your lovely web page will end up running off the edge of the screen. But, no worries; this script fixes it. So you should use it.
- `MBP.hideUrlBar` and `MBP.hideUrlBarOnLoad`: These two functions are callable but you are more likely to use `MBP.hideUrlBarOnLoad` to do pretty much what it says. This function is useful on mobiles because, once the page loads, it scrolls the URL bar up out of the view, thus saving precious screen real-estate. This is super useful for users who use Safari on an iPhone. I suppose you could call `MBP.hideUrlBar` but I have a hard time imagining a scenario where you want to directly call it without freaking out users. `MBP.hideUrlBarOnLoad` calls `MBP.hideUrlBar`.
- `MBP.fastButton`: This is a function to get around a feature of WebKit browsers that introduces a slight delay when users touch a link or button. Use this with caution.

- `MBP.splash`: This script can replace the `Startup images` block of commented-out code that we were discussing previously. It is provided in the head of the boilerplate that we were previously examining. If you've forgotten about it already, go back and read it over again. I really like the cleaner page when using this JavaScript to replace all that markup in the page, especially considering that only a few users will ever see this splash screen. In fact, if you go grab the most current version of HTML5 Mobile Boilerplate, instead of the index page provided in the current (as of this writing) version of 320 and Up, it won't have that block of markup with all the splash images.
- `MBP.autogrow`: This feature is great if you have forms on your responsive site. It makes `<textarea>` grow as a user fills it.
- `MBP.enableActive`: This is another awesome enhancement that enables the active pseudo class in Mobile Safari and is nice for user feedback on those buttons that tend to lag a bit (unless you are brave enough to use `FastButton`).
- `MBP.preventZoom`: This does what it says. The default behavior of Mobile Safari is to zoom in when an on-focus event happens. This is really inconvenient for users as they have to then manually zoom out after they are done inputting text to an input field.

Now you know enough to go experiment with these in your own apps. For the most part, you will want to fire these functions when the page is loaded and ready, so only use those that you need in order to prevent bogging down small devices with loads of JavaScript.

B

Using CSS Preprocessors

If you have already read any part of the book, you already know that I am begging you to use a CSS preprocessor, such as Sass or LESS. I prefer Sass for several reasons, but I won't really go into the details too much. Instead, I prefer to focus on the similarities between the two, and I will leave it to you to decide which framework you prefer. I am not being coy or disingenuous; I honestly find such arguments exhausting, especially when they come down to individual contexts.

I will say for the nth time that I prefer Sass. For me, some of it is the syntax but the differences from LESS are fairly minor. For me, another part of it is that I work mainly with Ruby on Rails, and Rails supports Sass right out of the box. Furthermore, I find the syntax of SCSS (rather than the older Sass that came earlier) to be so similar to CSS, which I've been using for about 25 percent of my life, so I also prefer it because it's less of a cognitive shift from CSS, which I know so well.

I also find some of the syntax of LESS to be confusing. The two main examples are how they use the @ symbol for global variables, as opposed to the \$ symbol that Sass uses, and how LESS uses the notation for a CSS class, .(period), to call a mixin, whereas Sass uses the more explicit @include. These are admittedly small quibbles.

But, there, those are my reasons. I hope you find them helpful in figuring out what works best for you.

Now, let's move on to understanding how all of these preprocessors help you work fast and efficiently. Hopefully, this is enough to whet your appetite so you can learn more. I will end the chapter with a list of resources, where you can go to learn more independently.

Why?

Why do we need something to preprocess CSS? Here are a few simple reasons:

- There are no variables in CSS
- When styling nested elements, your code will not be DRY, that is, you will type a lot of classes and/or IDs repeatedly
- It isn't convenient to re-use code in CSS, so you end up with code that isn't DRY
- There is no logic at all to CSS
- Preprocessors allow us to manipulate color relationships in a dynamic way rather than statically assigning all color values; this is especially powerful when coupled with the ability to use variables

These are broad explanations. This appendix, and essentially this whole book should give you lots of detailed examples of when these are useful.

How

Ok, so you've never tried a preprocessor. How do you get started? Of course, that depends on whether you choose LESS over Sass. I will walk you through three simple ways to use either of the two.

CodeKit

The easiest way to get started is with a Mac OS X application called CodeKit. I don't get kickbacks from the maker of this app. It's just a solid, simple app that does tons of stuff that used to be kind of a pain. It's not that expensive and you can get it at <http://incident57.com/codekit/>.

If you use anything other than a Mac, you are sadly out of luck as this app is for Mac only.

Once you've downloaded and launched it, it is trivial to add a project. The app is smart enough to find all the files in your project; more precisely, the files that fit into these categories are stylesheets, scripts, pages, and images.

CodeKit will preprocess all of these things in various ways; it will even optimize your images for you. As much as I am a fan of the command line, the convenience of this app has really won me over. I am trying to avoid gushing here but it will also do things such as preprocess haml and run JSLint or JSHint; it concatenates and minifies all your JavaScript, optimizes the images, and has many features, which I won't go deep into.

But, we are not here right now to use all those things; we are here to discuss how it facilitates the preprocessing of LESS and Sass. I am going to continue to describe how to set this up with the assumption that you don't own this app yet but just want to know more about how it works and, more specifically, how it works with 320 and Up. If you are using 320 and Up, and already have all your project files where they should be, you don't have to do much of anything else to get started. Just make sure that you select the appropriate file that essentially pulls in all the other files, and make sure it outputs the right file to the right location. Since CodeKit has an easy UI, all you have to do is right-click on `320andup-scss.scss` (for example, there is an equivalent file for LESS and the others) and set the output path, filename, and so on. That's just about it. Now let's look at some command-line tools.

Compass

Compass is a command-line tool that is community-driven. There are also GUIs for it. I have less experience with it, but there are lots of tutorials and guides on their site if you want to give it a go: <http://compass-style.org/>. Compass is a Ruby gem, so you can install and run it easily on the command line. Compass won't process LESS. But the LESS preprocessor is pretty simple to set up with the Node package manager.

The Sass/LESS gem

This solution is similar to using Compass. You install a gem on the command line with a simple configuration. Just as with the previously mentioned Compass and CodeKit, it will look for file changes and process your preprocessed code. I've had issues with LESS for successfully looking out for any code changes.

Rails

So, this is technically a fourth way and it's a bit redundant to mention Rails as a way to preprocess CSS but you can use any of the previously mentioned gems within a Rails project. The Sass gems will watch for code changes and process them without any further interaction. Again, in the past I've had issues getting LESS to watch for file changes and I had to restart the server to get it to process the code. This is unacceptable to me, since it just gobbles up time. On the other hand, things may have been fixed by now but I have moved on to Sass for additional reasons, some of which I previously outlined.

We will move on to what is happening in the preprocessed code itself in a moment, but I just want to tell you that, if you are intimidated by command-line tools, don't be. I entered this profession starting out mainly as a graphics guy, and have come to love the simplicity and elegance of the command line. There are many simple beginner courses that are free or are very cheap online and that will help you to get over your fears or confusion. I am a fan of the *Learn the Hard Way* tutorials, but there are tons more, and there will continue to be more. Once you know your way around the command line, I can assure that you will be able to work more efficiently than before.

What

Let's look at the sample project we are working with to see how it all hangs together. In this book, I focused on the SCSS variant of 320 and Up, so I will continue using the same through this appendix. For the most part, LESS is similar but has some syntactic differences. I will point out a few key examples along the way.

Let's look at the `before` project file from *Chapter 2, Building the Home Page*. Take a peek inside the `320andup` directory and look at the file structure for the moment. The main things I want to focus on are the `css` directory and the `scss` directory. The other siblings such as `less`, `sass-compass`, `sass`, and `scss-compass` hold the code to skin this cat in a different way.

Moving on to the `scss` folder, you will notice the file `320andup-scss.scss` and a bunch of files with underscores in front of them. The files with the underscores in front of them, for example, `_1382.scss`, are called *partials*. The underscore lets the preprocessor know not to turn these individual files into CSS. But they will have to get processed eventually though, right? That happens after they get imported to the one and only file that doesn't have an underscore in front of it. (LESS, on the other hand, does not use this underscore convention. For me, this is another small advantage I give to Sass. With Sass, I can make a quick visual scan of the file tree and know which files are *partials* and which aren't.)

Using the `320andup-scss.scss` file as an example, think of this file as the mother ship. All the other little ships dock there and unload their cargo. Once it's all there, things begin to happen. To be specific, CSS happens.

To learn how this comes together, let's look at the mother ship — `320andup-scss.scss`.

You will notice that the file is just a nice clean file that orders `imports`. Notice that the partials don't have the underscore in front of them in the `import` statements. Also, the ordering is important as, for example, you want to define all your variables and mixins before you try to use them. The other `imports` are placed inside the `@media` queries so that those files preceded by underscores (partials) are only used inside those queries.

What's so great about this? It keeps your code super tidy — easy to work with and maintain. This is the benefit of the 320 and Up framework. It takes care of the busy work of organizing all of this.

Lastly, I want to list some resources for you to check out to learn more about the CSS preprocessors and their helpers. Enjoy!

Resources

The following list is a list of resources for you to learn more about CSS preprocessors and their helpers:

- Sass: <http://sass-lang.com/>
- LESS: <http://lesscss.org/>
- Compass: <http://compass-style.org/>
- CodeKit: <http://incident57.com/codekit/>

Index

Symbols

320andup directory 16, 102
320 and Up framework 9
 benefit 103
\$basecolor variable 32
\$inputbordercolor variable 77
\$inputborderhovercolor variable 77
\$inputborder variable 77
\$inputhover variable 77
.big-icon styles 89
@media queries feature
 about 35, 36
 using 8
<nav> block 21

A

About Me page
 font-awesome icon fonts 87
 goals 81
 justifying 81, 82
 markup 84
 wireframes, creating 83
action parameter 73
active class 40
Andy Clarke's site
 URL 10

B

Baby Bear
 about 7
 display 7
before directory 12, 18

C

ch2 directory 12
CodeKit
 about 100
 URL 100, 103
 used, for CSS preprocessing 100, 101
Compass
 URL 101, 103
 used, for CSS preprocessing 101
container class 53
Content Delivery Network (CDN) 19
content panels
 building 31, 32
CSS
 preprocessing, CodeKit used 100, 101
 preprocessing, Compass used 101
 preprocessing, LESS gem used 101
 preprocessing, Rails used 102
 preprocessing, reasons 100
 preprocessing, sample project 102, 103
 preprocessing, Sass gem used 101
css directory 102
CSS preprocessor
 resources list 103
 types 99
 types, LESS 99
 types, Sass 99
 using 99
 using, reasons 100

D

div method 39
Document Object Model (DOM) 80

document ready function 41
Don't Repeat Yourself (DRY) 9, 47

E

em 49

F

font-awesome icon fonts 87-91

footer

about 15
building 32
designing 15-19
styling 33, 34

for attribute 73

form

layout 71-73

form plan

creating 69, 70

form plan, creating

form layout 71-73
input label 73-79

JS validation, fallbacks 79, 80

mandatory fields, managing 70, 71

G

gallery item JavaScript 67, 68

gallery page

back link, adding 64, 65
building 45, 48, 54, 57
content panel 54-57
desktop view 58
gallery item JavaScript 66, 67
layout 57
mock-up, viewing 48-53
structure 58-63
wireframe, creating 45-47

GitHub

URL 10

Goldilocks

about 7
display 7

H

helper.js file 97

hero

about 14, 29
building 29, 30

HTML5 Boilerplate

about 93, 94
conditional comments 95
helper.js file 97, 98
mobile meta tags 95, 96
scripts 96

I

input label

about 73
working 74-79

Internet Explorer (IE) Version 94

J

JS validation fallbacks

creating 79, 80

L

LESS

URL 103
versus Sass 99

LESS gem

used, for CSS preprocessing 101

M

mandatory fields

managing 70, 71

markup

about 84, 85
putting 85, 86

MBP.autogrow function 98

MBP.enableActive function 98

MBP.fastButton function 97

MBP.hideUrlBar function 97

MBP.hideUrlBarOnLoad function 97

MBP.preventZoom function 98

MBP.scaleFix function 97

MBP.splash function 98

Message field 70

My GitHub Fork

URL 10

N

navbar class 23

navigation

about 14, 21

building 22-29

O

open class 23, 28

P

page components

building 19-28

page components, building

header 20

logo 20

navigation 21-29

page responsiveness

creating 35-38

Papa Bear

display 8

Papa Bear device 7

Phone field 70

prerequisites, RWD

Andy Clarke's site 10

GitHub 10

My GitHub Fork 10

Python

URL 17

R

Rails gem

used, for CSS preprocessing 102

ready function 23, 66, 80

Regex 87

required attribute 74, 79

Responsive Web Design. *See* RWD

RWD

about 5

example 6

prerequisites 10

working 6-9

S

Sass

URL 103

versus LESS 99

Sass gem

used, for CSS preprocessing 101

scss directory 102

slide class 40

slider

about 14, 38

creating 39-41

T

type=tel attributes 75

W

webshims

URL 79

wireframes

about 12

creating 83, 84

workspace

preparing 11-13

workspace preparation

content panels 14, 31, 32

footer 15-19, 32-34

hero 14, 29, 30

navigation 14

page building, steps 12, 13

page responsiveness, creating 35-38

planning 12-14

slider 14, 38-42



Thank you for buying
**Mobile First Design with Html5
and CSS3**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Responsive Web Design by Example

ISBN: 978-1-849695-42-8 Paperback: 338 pages

Discover how you can easily create engaging, responsive websites with minimum hassle!

1. Rapidly develop and prototype responsive websites by utilizing powerful open source frameworks
2. Focus less on the theory and more on results, with clear step-by-step instructions, previews, and examples to help you along the way
3. Learn how you can utilize three of the most powerful responsive frameworks available today: Bootstrap, Skeleton, and Zurb Foundation



HTML5 and CSS3 Responsive Web Design Cookbook

ISBN: 978-1-849695-44-2 Paperback: 204 pages

Learn the secrets of developing responsive websites capable of interfacing with today's mobile Internet devices

1. Learn the fundamental elements of writing responsive website code for all stages of the development lifecycle
2. Create the ultimate code writer's resource using logical workflow layers
3. Full of usable code for immediate use in your website projects
4. Written in an easy-to-understand language giving knowledge without preaching

Please check www.PacktPub.com for information on our titles

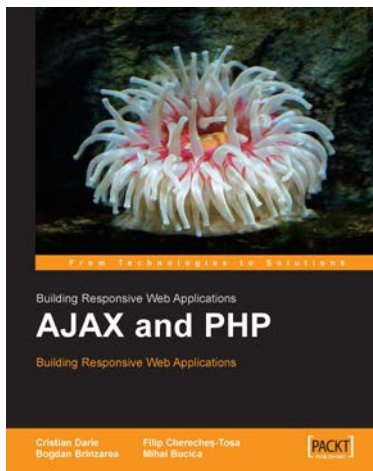


Responsive Web Design with HTML5 and CSS3

ISBN: 184-9-693-18-8 Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size
2. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions and transformations
3. Real world examples show how to progressively enhance a responsive design while providing fall backs for older browsers



AJAX and PHP: Building Responsive Web Applications

ISBN: 978-1-904811-82-4 Paperback: 284 pages

Enhance the user experience of your PHP website using AJAX with this practical tutorial featuring detailed case studies

1. Build a solid foundation for your next generation of web applications
2. Build a solid foundation for your next generation of web applications
3. Leverage the power of PHP and MySQL to create powerful back-end functionality and make it work in harmony with the smart AJAX client

Please check www.PacktPub.com for information on our titles