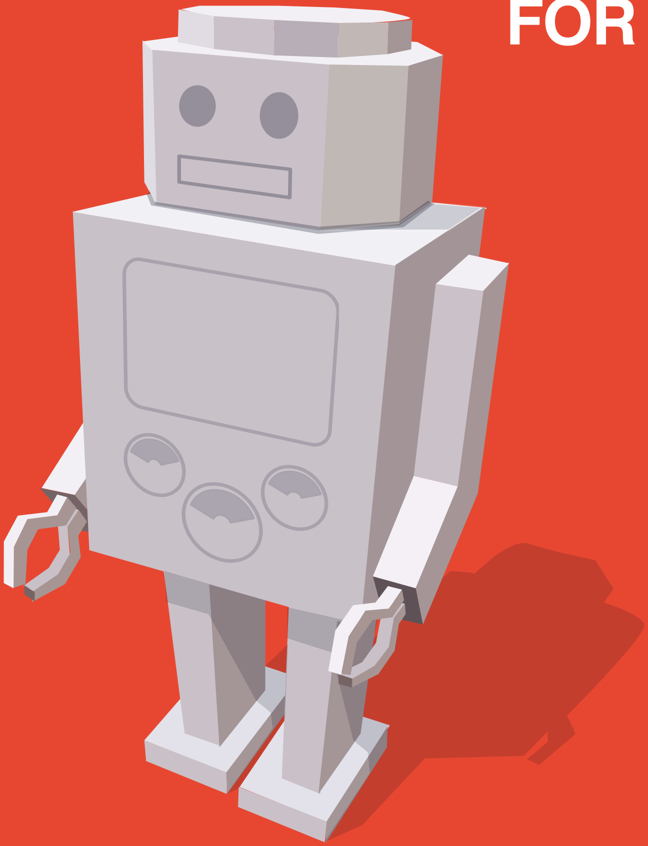
sitepoint

HTML5 & CSS3 FOR THE REAL WORLD

SECOND EDITION

BY ALEXIS GOLDSTEIN
LOUIS LAZARIS
& ESTELLE WEYL



POWERFUL HTML5 AND CSS3 TECHNIQUES YOU CAN USE TODAY!

Summary of Contents

sitepoint®

HTML5 & CSS3 FOR THE REAL WORLD

BY **ALEXIS GOLDSTEIN**
LOUIS LAZARIS
ESTELLE WEYL

HTML5 & CSS3 for the Real World

by Alexis Goldstein, Louis Lazaris, and Estelle Weyl

Copyright © 2015 SitePoint Pty. Ltd.

Product Manager: Simon Mackie
Technical Editor: Aurelio De Rosa
English Editor: Kelly Steele
Cover Designer: Alex Walker

Printing History:

First Edition: May 2011

Second Edition: March 2015

Notice of Rights

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 embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9874674-8-5 (print)

ISBN 978-0-9874674-9-2 (ebook)
Printed and bound in the United States of America

About Alexis Goldstein

Alexis Goldstein first taught herself HTML while a high school student in the mid-1990s, and went on to get her degree in Computer Science from Columbia University. She runs her own software development and training company, aut faciam LLC. Before striking out on her own, Alexis spent seven years in technology on Wall Street, where she worked in both the cash equity and equity derivative spaces at three major firms, and learned to love daily code reviews. She taught dozens of classes to hundreds of students as a teacher and co-organizer of Girl Develop It, a group that conducts low-cost programming classes for women. You can find Alexis at her website, http://alexisgo.com/.

About Louis Lazaris

Louis Lazaris is a freelance web designer and front-end developer based in Toronto, Canada who has been involved in the web design industry since 2000, when table layouts and one-pixel GIFs dominated the industry. In recent years he has transitioned to embrace web standards while endeavoring to promote best practices that help both developers and their clients reach practical goals for their projects. Louis is Managing Editor for SitePoint's HTML/CSS content, blogs about front-end code on his website Impressive Webs (http://www.impressivewebs.com/), and curates Web Tools Weekly (http://webtoolsweekly.com/), a weekly newsletter focused on tools for front-end developers.

About Estelle Weyl

Estelle Weyl is a front-end engineer from San Francisco who has been developing standards-based accessible websites since 1999. Estelle began playing with CSS3 when the iPhone was released in 2007, and after four years of web application development for mobile WebKit, she knows (almost) every CSS3 quirk on WebKit, and has vast experience implementing components of HTML5. She writes two popular technical blogs with tutorials and detailed grids of CSS3 and HTML5 browser support (http://www.standardista.com/). Estelle's passion is teaching web development, where you'll find her speaking on CSS3, HTML5, JavaScript, and mobile web development at conferences around the USA (and, she hopes, the world).

About the Technical Editor

Aurelio De Rosa is a (full-stack) web and app developer with more than 5 years' experience programming for the web using HTML, CSS, Sass, JavaScript, and PHP. He's an expert on JavaScript and HTML5 APIs, but his interests include web security, accessibility, performance, and SEO. He's also a regular writer for several networks, speaker, and author of the books *jQuery in Action, third edition* and *Instant jQuery Selectors*.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

To my mother, who always encourages and believes in me.

And to my father, who taught me so much about living up to my full potential. I miss you every day.

To Cakes, the most brilliant person I know. Thank you for everything you do for me. I'm so grateful for each and every day with you.

—Alexis

To Melanie, the best cook in the world.

And to my parents, for funding the original course that got me into this unique industry.

—Louis

To Amie, for putting up with me, and to Spazzo and Puppers, for snuggling with me as I worked away.

—Estelle

Table of Contents

Preface

Welcome to *HTML5 & CSS3 for the Real World*. We're glad you've decided to join us on this journey of discovering some of the latest and the greatest in front-end website building technology.

If you've picked up a copy of this book, it's likely that you've dabbled to some degree in HTML and CSS. You might even be a bit of a seasoned pro in certain areas of markup, styling, or scripting, and now want to extend those skills further by dipping into the features and technologies associated with HTML5 and CSS3.

Learning a new task can be difficult. You may have limited time to invest in poring over the official documentation and specifications for these web-based languages. You also might be turned off by some of the overly technical books that work well as references but provide little in the way of real-world, practical examples.

To that end, our goal with this book is to help you learn through hands-on, practical instruction that will assist you to tackle the real-world problems you face in building websites today—with a specific focus on HTML5 and CSS3.

But this is more than just a step-by-step tutorial. Along the way, we'll provide plenty of theory and technical information to help fill in any gaps in your understanding—the whys and hows of these new technologies—while doing our best not to overwhelm you with the sheer volume of cool new stuff. So let's get started!

Who Should Read This Book

This book is aimed at web designers and front-end developers who want to learn about the latest generation of browser-based technologies. You should already have at least intermediate knowledge of HTML and CSS, as we won't be spending any time covering the basics of markup and styles. Instead, we'll focus on teaching you what new powers are available to you in the form of HTML5 and CSS3.

The final two chapters of this book cover some of the new JavaScript APIs that have come to be associated with HTML5. These chapters, of course, require some basic familiarity with JavaScript—but they're not critical to the rest of the book. If you're unfamiliar with JavaScript, there's no harm in skipping over them for now, returning later when you're better acquainted with it.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout the book to signify different types of information. Look out for the following items:

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2015/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.learnable.com/books/htmlcss2/
The book's website, which contains links, updates, resources, and more.

https://github.com/spbooks/htmlcss2/
: The downloadable code archive for this book.

http://community.sitepoint.com/
: SitePoint's forums, for help on any tricky web problems.

`books@sitepoint.com`
: Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Acknowledgments

We'd like to offer special thanks to the following members of the SitePoint and Learnable community who made valuable contributions to this edition of the book:

Martin Ansdell-Smith, Ilya Bodrov, Jacob Christiansen, Ethan Glass, Gerard Konars, Dityo Nurasto, Thom Parkin, Guilherme Pereira, Jason Rogers, Bernard Savonet, and Julian Tancredi.

Alexis Goldstein

Thank you to Simon Mackie and Aurelio DeRosa. Simon, you always kept us on track and helped to successfully wrangle three co-authors, no small feat. And Aurelio, your incredible attention to detail, impressive technical expertise and catching of errors has made this book so much better than it would have been without your immense contributions. Thank you to my co-authors, Louis and Estelle, who never failed to impress me with their deep knowledge, vast experience, and uncanny ability to find bugs in the latest browsers. A special thank you to Estelle for the encouragement, for which I am deeply grateful.

Louis Lazaris

Thank you to my wife for putting up with my odd work hours while I took part in this great project. Thanks to my talented co-authors, Estelle and Alexis, for gracing me with the privilege of having my name alongside theirs, and, of course, to our expert reviewer Aurelio De Rosa for always challenging me with his great technical insight. And special thanks to the talented staff at SitePoint for their super-professional handling of this project and everything that goes along with such an endeavor.

Estelle Weyl

Thank you to the entire open source community. With the option to “view source,” I have learned from every developer who opted for markup rather than plugins. I would especially like to thank Jen Mei Wu and Sandi Watkins, who helped point me in the right direction when I began my career. Thank you to Dave Gregory and Laurie Voss who have always been there to help me find the words when they escaped me. Thank you to Stephanie Sullivan for brainstorming over code into the wee hours of the morning. And thank you to my developer friends at Opera, Mozilla, and Google for creating awesome browsers, providing us with the opportunity to not just play with HTML5 and CSS, but also to write this book.

Want to Take Your Learning Further?

Thanks for buying this book—we appreciate your support. Do you want to continue learning? You can now gain unlimited access to courses and ALL SitePoint books at Learnable for one low price. Enroll now and start learning today! Join Learnable and you’ll stay ahead of the newest technology trends: http://www.learnable.com.

Chapter 1

Introducing HTML5 and CSS3

This chapter gives a basic overview of how the web development industry has evolved and why HTML5 and CSS3 are so important to modern websites and web apps. It will show how using these technologies will be invaluable to your career as a web professional.

Of course, if you'd prefer to just get into the meat of the project that we'll be building, and start learning how to use all the new bells and whistles that HTML5 and CSS3 bring to the table, you can always skip ahead to Chapter 2 and come back later.

What is HTML5?

What we understand today as HTML5 has had a relatively turbulent history. You probably already know that HTML is the predominant markup language used to describe content, or data, on the World Wide Web (another lesser-used markup language is XML). HTML5 is the latest iteration of the HTML5 language and includes new features, improvements to existing features, and JavaScript APIs.

That said, HTML5 is not a reformulation of previous versions of the language—it includes all valid elements from both HTML4 and XHTML 1.0. Furthermore, it's been designed with some principles in mind to ensure it works on just about every platform, is compatible with older browsers, and handles errors gracefully. A summary of the design principles that guided the creation of HTML5 can be found on the W3C's HTML Design Principles page.[1]

First and foremost, HTML5 includes redefinitions of existing markup elements in addition to new elements that allow web designers to be more expressive in describing the content of their pages. Why litter your page with `div` elements when you can use `article`, `section`, `header`, `footer`, and so on?

The term "HTML5" has also been used to refer to a number of other new technologies and APIs. Some of these include drawing with the `canvas` element, offline storage, the new `video` and `audio` elements, drag-and-drop functionality, Microdata, and embedded fonts. In this book, we'll be covering a number of those technologies, and more.

Application Programming Interface

API stands for Application Programming Interface. Think of an API in the same way you think of a graphical user interface or GUI—except that instead of being an interface for humans, it's an interface for your code. An API provides your code with a set of "buttons" (predefined methods) that it can press to elicit the desired behavior from the system, software library, or browser.

API-based commands are a way of abstracting the more complex workings that are done in the background (or sometimes by third-party software). Some of the HTML5-related APIs will be introduced and discussed in later sections of this book.

Overall, you shouldn't be intimidated if you've had little experience with JavaScript or other APIs. While it would certainly be beneficial to have some experience with JavaScript or other languages, it isn't mandatory. Whatever the case, we'll walk you through the scripting parts of our book gradually, ensuring that you're not left scratching your head!

At the time of writing, it's been a good 5-plus years since HTML5 has had wide use in terms of the semantic elements and the various APIs. So it's no longer correct to categorize HTML5 as a "new" set of technologies—but it is still maturing and there

[1] http://www.w3.org/TR/html-design-principles/

are ongoing issues that continue to be addressed (such as bugs in browsers, and inconsistent support across browsers and platforms).

It should also be noted that some technologies were never part of HTML5 (such as CSS3 and WOFF), yet have at times been lumped in under the same label. This has instigated the use of broad, all-encompassing expressions such as "HTML5 and related technologies." In the interest of brevity—and also at the risk of inciting heated arguments—we'll generally refer to these technologies collectively as "HTML5."

How did we get here?

The web development industry has evolved significantly in a relatively short time period. In the late 1990s, a website that included images and an eye-catching design was considered top of the line in terms of web content and presentation.

Today, the landscape is quite different. Simple performance-driven, Ajax-based websites (usually differentiated as "web apps") that rely on client-side scripting for critical functionality are becoming more and more common. Websites today often resemble standalone software applications, and an increasing number of developers are viewing them as such.

Along the way, web markup has evolved. HTML4 eventually gave way to XHTML, which is really just HTML4 with strict XML-style syntax. HTML5 has taken over as the most-used version of markup, and we now rarely, if ever, see new projects built with HTML4 or XHTML.

HTML5 originally began as two different specifications: Web Forms 2.0[2] and Web Apps 1.0.[3] Both were a result of the changed web landscape and the need for faster and more efficient maintainable web applications. Forms and app-like functionality are at the heart of web apps, so this was the natural direction for the HTML5 spec to take. Eventually, the two specs were merged to form what we now call HTML5.

For a short time, there was discussion about the production of XHTML 2.0,[4] but that project has long since been abandoned to allow focus on the much more practical HTML5.

[2] http://www.w3.org/TR/web-forms-2/
[3] https://whatwg.org/specs/web-apps/2005-09-01/
[4] http://www.w3.org/TR/xhtml2/

Would the real HTML5 please stand up?

Because the HTML5 specification is being developed by two different bodies (the WHATWG and the W3C), there are two versions of the spec. The W3C (or World Wide Web Consortium) you're probably familiar with: it's the organization that maintains the original HTML and CSS specifications, as well as a host of other web-related standards such as SVG (Scalable Vector Graphics) and WCAG (Web Content Accessibility Guidelines).

The WHATWG (aka the Web Hypertext Application Technology Working Group), on the other hand, was formed by a group of people from Apple, Mozilla, and Opera after a 2004 W3C meeting left them disheartened. They felt that the W3C was ignoring the needs of browser makers and users by focusing on XHTML 2.0, instead of working on a backwards-compatible HTML standard. So they went off on their own and developed the Web Apps and Web Forms specifications that we've discussed, which were then merged into a spec they called HTML5. On seeing this, the W3C eventually gave in and created its own HTML5 specification based on the WHATWG's spec.

This can seem a little confusing. Yes, there are some politics behind the scenes that we, as designers and developers, have no control over. But should it worry us that there are two versions of the spec? In short, no.

The WHATWG's version of the specification can be found at http://www.whatwg.org/html/, and in January 2011 was renamed "HTML" (dropping the "5"). It's now called a "living standard,"[5] meaning that it will be in constant development and will no longer be referred to using incrementing version numbers.

The WHATWG version contains information covering HTML-only features, including what's new in HTML5. Additionally, there are separate specifications being developed by WHATWG that cover the related technologies. These specifications include Microdata, Canvas 2D Context, Web Workers, Web Storage, and others.

[5] http://blog.whatwg.org/html-is-the-new-html5

The W3C's version of the spec can be found at ht-
tp://www.w3.org/html/wg/drafts/html/master/, and the separate specifications for the other technologies can be accessed through http://dev.w3.org/html5/.[6]

So what's the difference between the W3C spec and that of WHATWG? Besides the name ("Living Standard" versus "HTML5.1"), the WHATWG version is a little more informal and experimental (and, some might argue, more forward-thinking). But in most places they're identical, so either one can be used as a basis for studying new HTML5 elements and related technologies.[7]

Why should I care about HTML5?

As mentioned, at the core of HTML5 are a number of new semantic elements, as well as several related technologies and APIs. These additions and changes to the language have been introduced with the goal of allowing developers to build web pages that are easier to code, use, and access.

These new semantic elements, along with other standards such as WAI-ARIA and Microdata (which we cover in Appendix B and Appendix C respectively), help to make our documents more accessible to both humans and machines—resulting in benefits for both accessibility and search engine optimization.

The semantic elements, in particular, have been designed with the dynamic Web in mind, with a particular focus on making pages more accessible and modular. We'll go into more detail on this in later chapters.

Finally, the APIs associated with HTML5 help improve on a number of techniques that web developers have been using for years. Many common tasks are now simplified, putting more power in developers' hands. Furthermore, the introduction of HTML5 audio and video means that there will be less dependence on third-party software and plugins when publishing rich media content on the Web.

[6] Technically, the W3C's version has now been upgraded to a new version: "HTML5.1" [http://www.w3.org/TR/html51/]. For simplicity we'll continue to refer to both versions as "HTML5". In addition, the W3C's website has a wiki page dedicated to something called "HTML.next [http://www.w3.org/wiki/HTML/next]", which discusses some far-future features of HTML that we won't cover in this book.
[7] There's a document published by the W3C [http://www.w3.org/wiki/HTML/W3C-WHATWG-Differences] that details many of the differences between the two specs, but most of the differences aren't very relevant or useful.

Overall, there are good reasons to start looking into HTML5's new features and APIs, and we'll discuss more of those reasons as we go through this book.

What is CSS3?

Another separate—but no less important—part of creating web pages is Cascading Style Sheets (CSS). As you probably know, CSS is a style language that describes how HTML markup is presented to the user. CSS3 is the latest version of the CSS specification.

CSS3 contains just about everything that's included in CSS2.1, the previous version of the spec. It also adds new features to help developers solve a number of presentation-related problems without resorting to scripting plugins or extra images.

New features in CSS3 include support for additional selectors, drop shadows, rounded corners, updated layout features, animation, transparency, and much more.

CSS3 is distinct from HTML5. In this publication, we'll be using the term CSS3 to refer to the current level of the CSS specification, with a particular focus on what's been added since CSS2.1. Thus, CSS3 is separate from HTML5 and its related APIs.

One final point should be made here regarding CSS and the current "version 3" label. Although this does seem to imply that there will one day be a "CSS4," Tab Atkins, a member of the CSS Working Group, has noted that there are no plans for it.[8] Instead, as he explains, the specification has been divided into separate modules, each with its own version number. So you might see something like "CSS Color Module Level 4"[9]—but that does not refer to "CSS4." No matter what level an individual module is at, it will still technically be under the umbrella of "CSS3," or better yet, simply "CSS." For the purposes of this book, we'll still refer to it as "CSS3," but just understand that this is likely to be the last version number for the language as a whole.

[8] http://www.xanthir.com/b4Ko0
[9] http://dev.w3.org/csswg/css-color/

Why should I care about CSS3?

Later in this book, we'll look in greater detail at many of the new features in CSS. In the meantime, we'll give you a taste of why CSS3's new techniques are so exciting to web designers.

Some design techniques find their way into almost every project. Drop shadows, gradients, and rounded corners are three good examples. We see them everywhere. When used appropriately, and in harmony with a site's overall theme and purpose, these enhancements can make a design flourish. Perhaps you're thinking: we've been creating these design elements using CSS for years now. But have we?

In the past, in order to create gradients, shadows, and rounded corners, web designers have had to resort to a number of tricky techniques. Sometimes extra HTML elements were required. In cases where the HTML is kept fairly clean, scripting hacks were required. In the case of gradients, the use of extra images was inevitable. We put up with these workarounds, because there was no other way of accomplishing those designs. CSS3 allows you to include these and other design elements in a forward-thinking manner that leads to so many benefits: cleaner markup, maintainable code, fewer extraneous images, and faster-loading pages.

A Short History on Vendor Prefixes

Ever since experimental features in CSS3 have begun to be introduced, developers have had to use prefixes in their CSS to target those features in various browsers. Browsers add vendor prefixes to features that might still be experimental in the specification (that is, they're not very far along in the standards process).[10] For example, at one time it was common to see something like this for a simple CSS transition:

```
a {
    color: #3381d6;
    -webkit-transition: color 0.4s ease;
    -moz-transition: color 0.4s ease;
```

[10] For more info, see: http://www.sitepoint.com/web-foundations/vendor-specific-properties/

```
    -o-transition: color 0.4s ease;
    transition: color 0.4s ease;
}
```

This would seem counterproductive to what was just mentioned, namely that CSS3 makes the code cleaner and easier to maintain. Fortunately, many prefixes are no longer needed. Additionally, we highly recommend that developers use a tool that will add prefixing automatically to your CSS.

One such tool is called Autoprefixer.[11] Autoprefixer can be included as part of your Grunt[12] workflow to post-process your CSS. With this, you need to include only the standard version of any CSS feature, and Autoprefixer will look through the Can I use... database[13] to determine if any vendor prefixes are needed. It will then build your CSS automatically, with all necessary prefixes. You also have the option to manually process your CSS using an online tool such as pleeease.[14] Whatever the case, in many places in this book we will include vendor prefixes, however be sure to use an online resource for up-to-date information on which features still require prefixes.

What do we mean by "the Real World"?

In the real world, we create web applications and we update them, fine-tune them, test them for potential performance problems, and continually tweak their design, layout, and content.

In other words, in the real world, we don't write code that we have no intention of revisiting. We write code using the most reliable, maintainable, and effective methods available, with every intention of returning to work on that code again to make any necessary improvements or alterations. This is evident not only in websites and web apps that we build and maintain as personal projects, but also in those we create and maintain for our clients.

We need to continually search out new and better ways to write our code. HTML5 and CSS3 are a big step in that direction.

[11] https://github.com/postcss/autoprefixer
[12] http://gruntjs.com/
[13] http://caniuse.com/
[14] http://pleeease.io/play/

The Current Browser Market

Although HTML5 is still in development, presenting significant changes in the way content is marked up, it's worth noting that those changes won't cause older browsers to choke, nor result in layout problems or page errors.

What this means is that you could take any old project containing valid HTML4 or XHTML markup, change the doctype to HTML5 (which we'll cover in Chapter 2), and the page will appear in the browser the same as it did before. The changes and additions in HTML5 have been implemented into the language in such a way as to ensure backwards-compatibility with older browsers—even older versions of Internet Explorer! Of course, this is no guarantee that the new features will work, it simply means they won't break your pages or cause any visible problems.

Even with regards to the more complex new features (for example, the APIs), developers have come up with various solutions to provide the equivalent experience to non-supporting browsers, all while embracing the exciting new possibilities offered by HTML5 and CSS3. Sometimes this is as simple as providing fallback content, such as a Flash video player to browsers without native video support. At other times, though, it's been necessary to use scripting to mimic support for new features.

These "gap-filling" techniques are referred to as **polyfills**. Relying on scripts to emulate native features isn't always the best approach when building high-performance web apps, but it's a necessary growing pain as we evolve to include new enhancements and features, such as the ones we'll be discussing in this book. Fortunately, as of writing, older browsers such as Internet Explorer 6 through 9 that fail to support many of the new features in HTML5 and CSS3, are used by less than 10% of web visitors today.[15] More and more people are using what has been termed evergreen browsers;[16] that is, browsers that automatically update. This means that new features will be functional to a larger audience, and eventually to all, as older browser shares wane.

[15] http://gs.statcounter.com/#browser_version-ww-monthly-201502-201502-bar
[16] http://tomdale.net/2013/05/evergreen-browsers/

In this book we may occasionally recommend fallback options or polyfills to plug the gaps in browser incompatibilities; we'll also try to do our best in warning you of potential drawbacks and pitfalls associated with using these options.

Of course, it's worth noting that sometimes no fallbacks or polyfills are required at all; for example, when using CSS3 to create rounded corners on boxes in your design, there's often no harm in users of really old browsers seeing square boxes instead. The functionality of the site has no degradation, and those users will be none the wiser about what they're missing.

As we progress through the lessons and introduce new subjects, if you plan on using one of these in a project we strongly recommend that you consult a browser-support reference such as the aforementioned Can I use...[17] That way, you'll know how and whether to provide fallbacks or polyfills. Where necessary, we'll occasionally discuss ways you can ensure that non-supporting browsers have an acceptable experience, but the good news is that it's becoming less and less of an issue as time goes on.

The Growing Mobile Market

Another compelling reason to start learning and using HTML5 and CSS3 today is the exploding mobile market. According to one source, in 2009 less than 1% of all web usage was on mobile devices and tablets.[18] By the middle of 2014, that number had risen to more than 35%![19] That's an astounding growth rate in a little more than five years. So what does this mean for those learning HTML5 and CSS3?

HTML5, CSS3, and related cutting-edge technologies are very well supported in many mobile web browsers. For example, mobile Safari on iOS devices such as the iPhone and iPad, Opera Mobile, Android Browser, and UC Browser all provide strong levels of HTML5 and CSS3 support. New features and technologies supported by some of those browsers include CSS3 animations, CSS flexbox, the Canvas API, Web Storage, SVG, Offline Web Apps, and more.

In fact, some of the new technologies we'll be introducing in this book have been specifically designed with mobile devices in mind. Technologies such as Offline Web Apps and Web Storage have been designed, in part, because of the growing

[17] http://caniuse.com/
[18] http://gs.statcounter.com/#desktop+mobile+tablet-comparison-ww-monthly-200901-200901-bar
[19] http://gs.statcounter.com/#desktop+mobile+tablet-comparison-ww-monthly-201408-201408-bar

number of people accessing web pages with mobile devices. Such devices can often have limitations with online data usage, and thus benefit greatly from the ability to access web applications offline.

We'll be touching on those subjects in Chapter 11, as well as others throughout the course of the book, providing the tools you'll need to create web pages for a variety of devices and platforms.

On to the Real Stuff

It's unrealistic to push ahead into new technologies and expect to author pages and apps for only one level of browser. In the real world, and in a world where we desire HTML5 and CSS3 to make further inroads, we need to be prepared to develop pages that work across a varied landscape. That landscape includes modern browsers, any remaining older versions of Internet Explorer, and an exploding market of mobile devices.

Yes, in some ways, supplying a different set of instructions for different user agents resembles the early days of the Web with its messy browser sniffing and code forking. But this time around, the new code is much more future-proof: when older browsers fall out of general use, all you need to do is remove any fallbacks and polyfills, leaving only the code base that's aimed at modern browsers.

HTML5 and CSS3 are the leading technologies that have ushered in a much more exciting world of web page authoring. Because all modern browsers provide excellent levels of support for a number of HTML5 and CSS3 features, creating powerful and simple-to-maintain future-proof web pages is easier than ever before.

So, enough about the "why," let's start digging into the "how"!

Chapter 2

Markup, HTML5 Style

Now that we've given you a bit of a history primer, along with some compelling reasons to learn HTML5 and start using it in your projects today, it's time to introduce you to the sample site that we'll be progressively building in this book.

After we briefly cover what we'll be building, we'll discuss some HTML5 syntax basics, along with some suggestions for best-practice coding. We'll follow that with some important info on cross-browser compatibility, and the basics of page structure in HTML5. Lastly, we'll introduce some specific HTML5 elements and see how they'll fit into our layout.

So let's get into it!

Introducing *The HTML5 Herald*

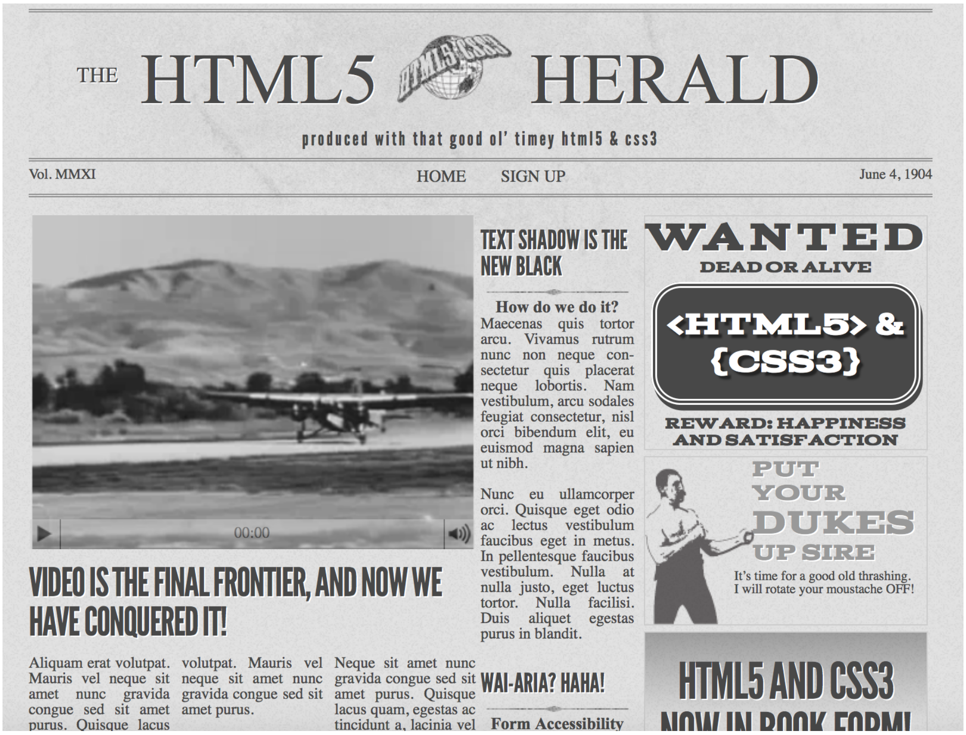
For the purpose of this book, we've put together a sample website project that we'll be building from scratch. The website is already built—you can check it out now at thehtml5herald.com.[1] It's an old-time newspaper-style website called *The HTML5 Herald*. The home page of the site contains some media in the form of video, images,

[1] http://thehtml5herald.com/

articles, and advertisements. There's also another page comprising a registration form.

Go ahead and view the source, and try some of the functionality if you like. As we proceed through the book, we'll be working through the code that went into making the site. We'll avoid discussing every detail of the CSS involved, as most of it should already be familiar to you—float layouts, absolute and relative positioning, basic font styling, and the like. We'll primarily focus on the new HTML5 elements, along with the APIs, plus all the new CSS3 techniques being used to add styles and interactivity to various elements.

Figure 2.1 shows a bit of what the finished product looks like.



THE HTML5 HERALD

produced with that good ol' timey html5 & css3

Vol. MMXI HOME SIGN UP June 4, 1904

00:00

VIDEO IS THE FINAL FRONTIER, AND NOW WE HAVE CONQUERED IT!

Aliquam erat volutpat. Mauris vel neque sit amet nunc gravida congue sed sit amet purus. Quisque lacus

volutpat. Mauris vel neque sit amet nunc gravida congue sed sit amet purus.

Neque sit amet nunc gravida congue sed sit amet purus. Quisque lacus quam, egestas ac tincidunt a, lacinia vel

TEXT SHADOW IS THE NEW BLACK

How do we do it?

Maecenas quis tortor arcu. Vivamus rutrum nunc non neque consectetur quis placerat neque lobortis. Nam vestibulum, arcu sodales feugiat consectetur, nisl orci bibendum elit, eu euismod magna sapien ut nibh.

Nunc eu ullamcorper orci. Quisque eget odio ac lectus vestibulum faucibus eget in metus. In pellentesque faucibus vestibulum. Nulla at nulla justo, eget luctus tortor. Nulla facilisi. Duis aliquet egestas purus in blandit.

WAI-ARIA? HAHA!

Form Accessibility

WANTED

DEAD OR ALIVE

<HTML5> & {CSS3}

REWARD: HAPPINESS AND SATISFACTION

PUT YOUR DUKES UP SIRE

It's time for a good old thrashing. I will rotate your moustache OFF!

HTML5 AND CSS3 NOW IN BOOK FORM!

Figure 2.1. The front page of *The HTML5 Herald*

While we build the site, we'll do our best to explain the new HTML5 elements, APIs, and CSS3 features, and aim to recommend some best practices. Of course, some of these technologies are still new and in development, so we'll try not to be too dogmatic about what you can and can't do.

A Basic HTML5 Template

As you learn HTML5 and add new techniques to your toolbox, you're likely to want to build yourself a boilerplate, from which you can begin all your HTML5-based projects. We encourage this, and you may also consider using one of the many online sources that provide a basic HTML5 starting point for you.[2]

In this project, however, we want to build our code from scratch and explain each piece as we go along. Of course, it would be impossible for even the most fantastical and unwieldy sample site we could dream up to include *every* new element or technique, so we'll also explain many new features that don't fit into the project. This way, you'll be familiar with a wide set of options when deciding how to build your HTML5 and CSS3 websites and applications, enabling you to use this book as a quick reference for a number of features and techniques.

Let's start simple, with a bare-bones HTML5 page:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <title>The HTML5 Herald</title>
    <meta name="description" content="The HTML5 Herald">
    <meta name="author" content="SitePoint">

    <link rel="stylesheet" href="css/styles.css">

    <!--[if lt IE 9]>
      <script src="js/html5shim.js"></script>
    <![endif]-->

  </head>
  <body>

    <script src="js/scripts.js"></script>
```

[2] A few you might want to look into can be found at html5boilerplate.com [https://html5boilerplate.com/] and https://github.com/murtaugh/HTML5-Reset.

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

With that basic template in place, let's now examine some of the significant parts of the markup and how these might differ from how HTML was written prior to HTML5.

The Doctype

First, we have the Document Type Declaration, or **doctype**. This is simply a way to tell the browser—or any other parser—what type of document it's looking at. In the case of HTML files, it means the specific version and flavor of HTML. The doctype should always be the first item at the top of any HTML file. Many years ago, the doctype declaration was an ugly and hard-to-remember mess. For XHTML 1.0 Strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www
➥.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

And for HTML4 Transitional:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http
➥://www.w3.org/TR/html4/loose.dtd">
```

Although that long string of text at the top of our documents hasn't really hurt us (other than forcing our sites' viewers to download a few extra bytes), HTML5 has done away with that indecipherable eyesore. Now all you need is this:

```
<!DOCTYPE html>
```

Simple, and to the point. The doctype can be written in uppercase, lowercase, or mixed case. You'll notice that the "5" is conspicuously missing from the declaration. Although the current iteration of web markup is known as "HTML5," it really is just an evolution of previous HTML standards—and future specifications will simply be a development of what we have today.

Because browsers are usually required to support all existing content on the Web, there's no reliance on the doctype to tell them which features should be supported in a given document. In other words, the doctype alone is not going to make your pages HTML5-compliant. It's really up to the browser to do this. In fact, you can use one of those two older doctypes with new HTML5 elements on the page and the page will render the same as it would if you used the new doctype.

The `html` Element

Next up in any HTML document is the `html` element, which has not changed significantly with HTML5. In our example, we've included the `lang` attribute with a value of `en`, which specifies that the document is in English. In XHTML-based markup, you were required to include an `xmlns` attribute. In HTML5, this is no longer needed, and even the `lang` attribute is unnecessary for the document to validate or function correctly.

So here's what we have so far, including the closing `html` tag:

```
<!DOCTYPE html>
<html lang="en">

</html>
```

The `head` Element

The next part of our page is the `head` section. The first line inside the `head` is the one that defines the character encoding for the document. This is another element that's been simplified since XHTML and HTML4, and is an optional feature, but recommended. In the past, you may have written it like this:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

HTML5 improves on this by reducing the character-encoding `meta` tag to the bare minimum:

```
<meta charset="utf-8">
```

In nearly all cases, `utf-8` is the value you'll be using in your documents. A full explanation of character encoding is beyond the scope of this book, and it probably won't be that interesting to you, either. Nonetheless, if you want to delve a little deeper, you can read up on the topic on W3C[3] or WHATWG.[4]

Encoding Declaration

To ensure that all browsers read the character encoding correctly, the entire character-encoding declaration must be included somewhere within the first 512 characters of your document. It should also appear before any content-based elements (such as the `title` element that follows it in our example site).

There's much more we could write about this subject, but we want to keep you awake—so we'll spare you those details! For now, we're content to accept this simplified declaration and move on to the next part of our document:

```
<title>The HTML5 Herald</title>
<meta name="description" content="The HTML5 Herald">
<meta name="author" content="SitePoint">

<link rel="stylesheet" href="css/styles.css">
```

In these lines, HTML5 barely differs from previous syntaxes. The page title (the only mandatory element inside the `head`) is declared the same as it always was, and the meta tags we've included are merely optional examples to indicate where these would be placed; you could put as many valid `meta` elements[5] here as you like.

The key part of this chunk of markup is the stylesheet, which is included using the customary `link` element. There are no required attributes for `link` other than `href` and `rel`. The `type` attribute (which was common in older versions of HTML) is not necessary, nor was it ever needed to indicate the content type of the stylesheet.

Leveling the Playing Field

The next element in our markup requires a bit of background information before it can be introduced. HTML5 includes a number of new elements, such as `article` and `section`, which we'll be covering later on. You might think this would be a major problem for older browser support for unrecognized elements, but you'd be

[3] http://www.w3.org/html/wg/drafts/html/master/infrastructure.html#encoding-terminology
[4] https://html.spec.whatwg.org/multipage/infrastructure.html#encoding-terminology
[5] https://html.spec.whatwg.org/multipage/semantics.html#the-meta-element

wrong. This is because the majority of browsers don't actually care what tags you use. If you had an HTML document with a `recipe` tag (or even a `ziggy` tag) in it, and your CSS attached some styles to that element, nearly every browser would proceed as if this were totally normal, applying your styling without complaint.

Of course, such a hypothetical document would fail to validate and may have accessibility problems, but it *would* render correctly in *almost* all browsers—the exception being old versions of Internet Explorer (IE). Prior to version 9, IE prevented unrecognized elements from receiving styling. These mystery elements were seen by the rendering engine as "unknown elements," so you were unable to change the way they looked or behaved. This includes not only our imagined elements, but also any elements that had yet to be defined at the time those browser versions were developed. That means (you guessed it) the new HTML5 elements.

The good news is, at the time of writing, most people still using a version of IE are using version 9 or higher, and very few are on version 9, so this is not a big problem for most developers anymore; however, if a big chunk of your audience is still using IE8 or earlier, you'll have to take action to ensure your designs don't fall apart.

Fortunately, there's a solution: a very simple piece of JavaScript originally developed by John Resig.[6] Inspired by an idea by Sjoerd Visscher, it can make the new HTML5 elements styleable in older versions of IE.

We've included this so-called "HTML5 shiv"[7] in our markup as a script tag surrounded by conditional comments. Conditional comments are a proprietary feature implemented in Internet Explorer in version 9 and earlier. They provide you with the ability to target specific versions of that browser with scripts or styles.[8] The following conditional comment is telling the browser that the enclosed markup should only appear to users viewing the page with Internet Explorer prior to version 9:

[6] http://ejohn.org/blog/html5-shiv/

[7] You might be more familiar with its alternative name: the HTML5 shim. Whilst there are identical code snippets out there that go by both names, we'll be referring to all instances as the HTML5 shiv, its original name.

[8] For more information see the SitePoint Reference [http://www.sitepoint.com/web-foundations/internet-explorer-conditional-comments/].

```
<!--[if lt IE 9]>
  <script src="js/html5shim.js"></script>
<![endif]-->
```

It should be noted that if you're using a JavaScript library that deals with HTML5 features or the new APIs, it's possible that it will already have the HTML5-enabling script present; in this case, you could remove reference to the script. One example of this would be Modernizr,[9] a JavaScript library that detects modern HTML and CSS features. Modernizr gives you the option to include code that enables the HTML5 elements in older versions of IE, so the shiv would be redundant. We take a closer look at Modernizr in Appendix A.

Not Everyone Can Benefit from the HTML5 Shiv

Of course, there's still a group of users unable to benefit from the HTML5 shiv: those who have for one reason or another disabled JavaScript and are using IE8 or lower. As web designers, we're constantly told that the content of our sites should be fully accessible to all users, even those without JavaScript. But it's not as bad as it seems. A number of studies have shown that the number of users who have JavaScript disabled is low enough to be of little concern, especially when you factor in how few of those will be using IE8 or lower.

In a study published in October, 2013, the UK Government Digital Service determined that users browsing government web services in the UK with JavaScript disabled or unavailable was 1.1%.[10] In another study conducted on the Yahoo Developer Network[11] (published in October 2010), users with JavaScript disabled amounted to around 1% of total traffic worldwide.

The Rest Is History

Looking at the rest of our starting template, we have the usual `body` element along with its closing tag and the closing `html` tag. We also have a reference to a JavaScript file inside a `script` element.

[9] http://www.modernizr.com/
[10] https://gds.blog.gov.uk/2013/10/21/how-many-people-are-missing-out-on-javascript-enhancement/
[11] https://developer.yahoo.com/blogs/ydn/many-users-javascript-disabled-14121.html

Much like the `link` tag discussed earlier, the `script` tag does not require that you declare a `type` attribute. If you ever wrote XHTML, you might remember your `script` tags looking like this:

```
<script src="js/scripts.js" type="text/javascript"></script>
```

Since JavaScript is, for all practical purposes, the only real scripting language used on the Web, and since all browsers will assume that you're using JavaScript even when you don't explicitly declare that fact, the `type` attribute is unnecessary in HTML5 documents:

```
<script src="js/scripts.js"></script>
```

We've put the `script` element at the bottom of our page to conform to best practices for embedding JavaScript. This has to do with the page-loading speed; when a browser encounters a script, it will pause downloading and rendering the rest of the page while it parses the script. This results in the page appearing to load much more slowly when large scripts are included at the top of the page before any content. It's why most scripts should be placed at the very bottom of the page, so that they'll only be parsed after the rest of the page has loaded.

In some cases, however, (such as with the HTML5 shiv) the script may *need* to be placed in the head of your document, because you want it to take effect before the browser starts rendering the page.

HTML5 FAQ

After this quick introduction to HTML5 markup, you probably have a bunch of questions swirling in your head. Here are some answers to a few of the likely ones.

Why do these changes still work in older browsers?

To understand why this isn't a problem, we can compare HTML5 to some of the new features added in CSS3, which we'll be discussing in later chapters.

In CSS, when a new feature is added (for example, the `border-radius` property that adds rounded corners to elements), that feature also has to be added to browsers' rendering engines, so older browsers will fail to recognize it. If a user is viewing the page on a browser with no support for `border-radius`, the rounded corners will appear square. This happens because square corners are the default and the browser will ignore the `border-radius` declaration. Other CSS3 features behave similarly, causing the experience to be degraded to some degree.

Many developers expect that HTML5 will work in a similar way. While this might be true for some of the advanced features and APIs we'll be considering later in the book, it's not the case with the changes we've covered so far; that is, the simpler syntax, fewer superfluous attributes, and the new doctype.

HTML5's syntax was more or less defined after a careful study of what older browsers can and can't handle. For example, the 15 characters that comprise the doctype declaration in HTML5 are the minimum characters required to make every browser display a page in standards mode.

Likewise, while XHTML required a lengthier character-encoding declaration and an extra attribute on the `html` element for the purpose of validation, browsers never actually required them in order to display a page correctly. Again, the behavior of older browsers was carefully examined, and it was determined that the character encoding could be simplified and the `xmlns` attribute removed—and browsers would still see the page the same way.

Unlike changes to CSS3 and JavaScript, where additions are only supported when browser makers actually implement them, there's no need to wait for new browser versions to be released before using HTML5's markup syntax. And when it comes to using the new semantic elements, a small snippet of JavaScript is all that's required to bring any really old browsers into line.

Standards Mode versus Quirks Mode

When standards-based web design was in its infancy, browser makers were faced with a problem: supporting emerging standards would, in many cases, break backwards compatibility with existing web pages that were designed to older, nonstandard browser implementations. Browser makers needed a signal indicating that a given page was meant to be rendered according to the standards. They found such a signal in the doctype: new standards-compliant pages included a correctly formatted doctype, while older nonstandard pages generally didn't. Using the doctype as a signal, browsers could switch between standards mode (in which they try to follow standards to the letter in the way they render elements) and quirks mode (where they attempt to mimic the “quirky” rendering capabilities of older browsers to ensure that the page renders how it was intended).

It’s safe to say that in the current development landscape, nearly every web page has a proper doctype, and thus will render in standards mode; it’s therefore unlikely that you’ll ever have to deal with a page rendered in quirks mode. Of course, if a user is viewing a web page using a very old browser (such as IE4), the page will be rendered using that browser’s rendering mode. This is what quirks mode mimics, and it will do so regardless of the doctype being used.

Although the XHTML and older HTML doctypes include information about the exact version of the specification they refer to, browsers have never actually made use of that information. As long as a seemingly correct doctype is present, they’ll render the page in standards mode. Consequently, HTML5’s doctype has been stripped down to the bare minimum required to trigger standards mode in any browser. Further information, along with a chart that outlines what will cause different browsers to render in quirks mode, can be found on Wikipedia.[12] You can also read a good overview of standards and quirks mode on SitePoint’s CSS reference.[13]

Shouldn’t all tags be closed?

In XHTML, all elements were required to be closed—either with a corresponding closing tag (such as `html`) or in the case of void elements, a forward slash at the end of the tag. Void elements are elements that *can’t* contain child elements (such as `input`, `img`, or `link`).

You can still use that style of syntax in HTML5—and you might prefer it for consistency and maintainability reasons—but adding a trailing slash on void elements is no longer required for validation. Continuing with the theme of “cutting the fat,” HTML5 allows you to omit the trailing slash from such elements, arguably leaving your markup cleaner and less cluttered.

It’s worth noting that in HTML5, most elements that *can* contain nested elements—but simply happen to be empty—still need to be paired with a corresponding closing tag. There are exceptions to this rule (such as `p` tags and `li` tags), but it’s simpler to assume that it’s universal.

[12] http://en.wikipedia.org/wiki/Quirks_mode/
[13] http://reference.sitepoint.com/css/doctypesniffing/

What about other XHTML-based syntax customs?

While we're on the subject, omitting closing slashes is just one aspect of HTML5-based syntax that differs from XHTML. In fact, syntax style issues are completely ignored by the HTML5 validator, which will only throw errors for code mistakes that threaten to disrupt your document in some way.

What this means is that through the eyes of the validator, the following five lines of markup are identical:

```
<link rel="stylesheet" href="css/styles.css" />
<link rel="stylesheet" href="css/styles.css">
<LINK REL="stylesheet" HREF="css/styles.css">
<Link Rel="stylesheet" Href="css/styles.css">
<link rel=stylesheet href=css/styles.css>
```

In HTML5, you can use lowercase, uppercase, or mixed-case tag names or attributes, as well as quoted or unquoted attribute values (as long as those values don't contain spaces or other reserved characters), and it will all validate just fine.

In XHTML, all attributes were required to have values, even if those values were redundant. For example, in XHTML you'd often see markup like this:

```
<input type="text" disabled="disabled" />
```

In HTML5, attributes that are either "on" or "off" (called Boolean attributes) can simply be specified with no value. So, the aforementioned `input` element can be written as follows:

```
<input type="text" disabled>
```

Hence, HTML5 has very loose requirements for validation, at least as far as syntax is concerned. Does this mean you should just go nuts and use whatever syntax you want on any given element? No, we certainly don't recommend that.

We encourage developers to choose a syntax style and stick to it—especially if you are working in a team environment where code maintenance and readability are crucial. We also recommend (though this is optional) that you choose a minimalist coding style while staying consistent.

Here are some guidelines for you to consider using:

- Use lowercase for all elements and attributes as you would in XHTML.
- Despite some elements not requiring closing tags, we recommend that all elements containing content be closed (as in `<p>Text</p>`).
- Although you can leave attribute values unquoted, it's highly likely that you'll have attributes that require quotes (for example, when declaring multiple classes separated by spaces, or when appending a query string value to a URL). As a result, we suggest that you always use quotes for the sake of consistency.
- Omit the trailing slash from void elements (such as `meta` or `input`).
- Avoid providing redundant values for Boolean attributes (for instance, use `<input type="checkbox" checked>` rather than `<input type="checkbox" checked="checked">`).

Again, these recommendations are by no means universally accepted; however, we believe that they're reasonable syntax suggestions for achieving clean, easy-to-read maintainable markup.

If you do run amok with your code style, including too much that's unnecessary, you're just adding extra bytes for no reason. You're also potentially making your code harder to maintain, especially if you work with other developers on the same code base.

Defining the Page's Structure

Now that we've broken down the basics of our template, let's start adding some meat to the bones and give our page some structure.

Later in the book, we're going to specifically deal with adding CSS3 features and other HTML5 goodness; for now, we'll consider what elements we want to use in building our site's overall layout. We'll be covering a lot in this section and throughout the coming chapters about semantics. When we use the term "semantics," we're referring to the way a given HTML element describes the meaning of its content.

If you look back at the screenshot of *The HTML5 Herald* (or view the site online), you'll see that it's divided up as follows:

- header section with a logo and title
- navigation bar
- body content divided into three columns
- articles and ad blocks within the columns
- footer containing some author and copyright information

Before we decide which elements are appropriate for these different parts of our page, let's consider some of our options. First of all, we'll introduce you to some of the new HTML5 semantic elements that could be used to help divide our page and add more meaning to our document's structure.

The header Element

Naturally, the first element we'll look at is the `header` element. The spec describes it succinctly as "a group of introductory or navigational aids."[14]

Contrary to what you might normally assume, you can include a new header element to introduce each section of your content. It's not just reserved for the page header (which you might normally mark up with `<div id="header">`). When we use the word "section" here, we're not limiting ourselves to the actual `section` element described in the next part; technically, we're referring to what HTML5 calls "sectioning content." This means any chunk of content that might need its own header, even if that means there are multiple such chunks on a single page.

A `header` element can be used to include introductory content or navigational aids that are specific to any single section of a page, or apply to the entire page, or both.

While a `header` element will frequently be placed at the top of a page or section, its definition is independent from its position. Your site's layout might call for the title of an article or blog post to be off to the left, right, or even below the content; regardless of which, you can still use `header` to describe this content.

[14] https://html.spec.whatwg.org/multipage/semantics.html#the-header-element

The `section` Element

The next element you should become familiar with is HTML5's `section` element. The spec defines `section` as follows:[15]

> The `section` element represents a generic section of a document or application. A section, in this context, is a thematic grouping of content, typically with a heading.

It further explains that a `section` shouldn't be used as a generic container that exists for styling or scripting purposes only. If you're unable to use `section` as a generic container—for example, in order to achieve your desired CSS layout—then what should you use? Our old friend, the `div` element, which is semantically meaningless.

Going back to the definition from the spec, the `section` element's content should be "thematic," so it would be incorrect to use it in a generic way to wrap unrelated pieces of content.

Some examples of acceptable uses for `section` elements include:

- individual sections of a tabbed interface
- segments of an "About" page; for example, a company's "About" page might include sections on the company's history, its mission statement, and its team
- different parts of a lengthy "terms of service" page
- various sections of an online news site; for example, articles could be grouped into sections covering sports, world affairs, and economic news

Using `section` Correctly

Every time new semantic markup is made available to web designers, there will be debate over what constitutes correct use of these elements, what the spec's intention was, and so on. You may remember discussions about the appropriate use of the `dl` element in previous HTML specifications. Unsurprisingly, HTML5 has not been immune to this phenomenon, particularly when it comes to the `section` element. Even Bruce Lawson, a well-respected authority on HTML5, has admitted

[15] https://html.spec.whatwg.org/multipage/semantics.html#the-section-element

to using `section` incorrectly in the past. For a bit of clarity, it's well worth reading Bruce's post explaining his error.[16]

In short:

- `section` is *generic*, so if a more specific semantic element is appropriate (such as `article`, `aside`, or `nav`), use that instead.

- `section` *has semantic meaning*; it implies that the content it contains is related in some way. If you're unable to succinctly describe all the content you're trying to put in a `section` using just a few words, it's likely you need a semantically neutral container instead: the humble `div`.

That said, as is always the case with semantics, it's open to interpretation in some instances. If you feel you can make a case for why you're using a given element rather than another, go for it. In the unlikely event that anyone ever calls you on it, the resulting discussion can be both entertaining and enriching for everyone involved, and might even contribute to the wider community's understanding of the specification.

Keep in mind, also, that you're permitted to nest `section` elements inside existing `section` elements, if it's appropriate. For example, for an online news website, the World News section might be further subdivided into a section for each major global region.

The `article` Element

The `article` element is similar to the `section` element, but there are some notable differences. Here's the definition according to the spec:[17]

> The article element represents a complete, or self-contained, composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g. in syndication.

The key terms in that definition are *self-contained composition* and *independently distributable*. Whereas a `section` can contain any content that can be grouped thematically, an `article` must be a single piece of content that can stand on its

[16] http://html5doctor.com/the-section-element/
[17] https://html.spec.whatwg.org/multipage/semantics.html#the-article-element

own. This distinction can be hard to wrap your head around, so when in doubt, try the test of syndication: if a piece of content can be republished on another site without being modified, or if it can be pushed out as an update via RSS, or on social media sites such as Twitter or Facebook, it has the makings of an `article`.

Ultimately, it's up to you to decide what constitutes an article, but here are some suggestions in line with recommendations in the spec:

- a forum post
- a magazine or newspaper article
- a blog entry
- a user-submitted comment on a blog entry or article

Finally, just like `section` elements, `article` elements can be nested inside other `article` elements. You can also nest a `section` inside an `article`, and vice versa. It all depends on the content you're marking up.

The nav Element

It's safe to assume that the `nav` element will appear in virtually every project. `nav` represents exactly what it implies: a group of navigation links. Although the most common use for `nav` will be for wrapping an unordered list of links, there are other options. For example, you could wrap the `nav` element around a paragraph of text that contained the major navigation links for a page or section of a page.

In either case, the `nav` element should be reserved for navigation that is of primary importance. So, it's recommended that you avoid using `nav` for a brief list of links in a footer, for example.

Skip Navigation Links

A design pattern you may have seen implemented on many sites is the "skip navigation" link. The idea is to allow users of screen readers to quickly skip past your site's main navigation if they've already heard it—after all, there's no point listening to a large site's entire navigation menu every time you click through to a new page! The `nav` element has the potential to eliminate this need; if a screen reader sees a `nav` element, it could allow its users to skip over the navigation without requiring an additional link. The specification states: "User agents (such as screen readers) that are targeted at users who can benefit from navigation information being omitted in the initial rendering, or who can benefit from navigation information being immediately available, can use this element as a way to determine what content on the page to initially skip or provide on request (or both)."

Although not all assistive devices recognize `nav` as of this writing, by building to the standards now you ensure that as screen readers improve, your page will become more accessible over time.

User Agents

You'll encounter the term "user agent" a lot when browsing through specifications. Really, it's just a fancy term for a browser—a software "agent" that a user employs to access the content of a page. The reason the specs don't simply say "browser" is that user agents can include screen readers or any other technological means to read a web page.

You can use `nav` more than once on a given page. If you have a primary navigation bar for the site, this would call for a `nav` element. Additionally, if you had a secondary set of links pointing to different parts of the current page (using in-page anchors or "local" links), this too could be wrapped in a `nav` element.

As with `section`, there's been some debate over what constitutes acceptable use of `nav` and why it isn't recommended in some circumstances (such as in a footer). Some developers believe this element is appropriate for pagination or breadcrumb links, or for a search form that constitutes a primary means of navigating a site (as is the case on Google).

This decision will ultimately be up to you, the developer. Ian Hickson, the primary editor of WHATWG's HTML5 specification, responded to the question directly: "use [it] whenever you would have used class=nav".[18]

[18] See http://html5doctor.com/nav-element/#comment-213. [http://html5doctor.com/nav-element/#comment-213]

The `aside` Element

This element represents a part of the page that's "tangentially related to the content around the `aside` element, and which could be considered separate from that content."[19]

The `aside` element could be used to wrap a portion of content that is tangential to:

- a specific standalone piece of content (such as an `article` or `section`).
- an entire page or document, as is customarily done when adding a sidebar to a page or website.

The `aside` element should never be used to wrap sections of the page that are part of the primary content; in other words, `aside` is not meant to be parenthetical. The `aside` content could stand on its own, but it should still be part of a larger whole.

Some possible uses for `aside` include a sidebar, a secondary list of links, or a block of advertising. It should also be noted that the `aside` element (as in the case of `header`) is not defined by its position on the page. It could be on the side, or it could be elsewhere. It's the content itself, and its relation to other elements, that defines it.

The `footer` Element

The final element we'll discuss in this chapter is the `footer` element. As with `header`, you can have multiple `footer` elements on a single page, and you should use `footer` instead of something generic such as `<div id="footer">`.

A footer element, according to the spec, represents a footer for the section of content that is its nearest ancestor. The section of content could be the entire document, or it could be a `section`, `article`, or `aside` element.

Often a `footer` will contain copyright information, lists of related links, author information, and similar information that you normally think of as coming at the end of a block of content; however, much like `aside` and `header`, a footer element is not defined in terms of its position on the page, so it does not have to appear at the end

[19] https://html.spec.whatwg.org/multipage/semantics.html#the-aside-element

of a section, or at the bottom of a page. Most likely it will, but this is not required. For example, information about the author of a blog post might be displayed above the post instead of below it, and will still be considered footer information.

How did we get here?

If you're wondering a little bit about the path to HTML5 and how we ended up with the tags that we did, you might want to check out Luke Stevens' book called *The Truth about HTML5*.[20] Currently in its 2nd edition, Luke's book is somewhat controversial. In addition to covering many of the HTML5 technologies such as video and canvas, he also goes in-depth in his coverage of the history of HTML5, explaining some of the semantic and accessibility problems inherent in the new elements and providing some recommendations on how to handle these issues.

Structuring *The HTML5 Herald*

Now that we've covered the basics of page structure and the elements in HTML5 that will assist in this area, it's time to start building the parts of our page that will hold the content.

Let's start from the top, with a `header` element. It makes sense to include the logo and title of *The Herald* in here, as well as the tagline. We can also add a `nav` element for the site navigation.

After the `header`, the main content of our site is divided into three columns. While you might be tempted to use `section` elements for these, stop and think about the content. If each column contained a separate "section" of information (such as a sports section and an entertainment section), that would make sense. As it is, though, the separation into columns is really only a visual arrangement, so we'll use a plain old `div` for each column.

Inside those `div`s, we have newspaper articles; these, of course, are perfect candidates for the `article` element.

The column on the far right, though, contains three ads in addition to an article. We'll use an `aside` element to wrap the ads, with each ad placed inside an `article` element. This may seem odd, but look back at the description of `article`: "a self-

[20] http://www.truthabouthtml5.com/

contained composition […] that is, in principle, independently distributable or reusable." An ad fits the bill almost perfectly, as it's usually intended to be reproduced across a number of websites without modification.

Next up, we'll add another `article` element for the final article that appears below the ads. That final article will be *excluded* from the `aside` element that holds the three ads. To belong in the `aside`, the `article` needs to be tangentially related to the page's content. This isn't the case: this article is part of the page's main content, so it would be wrong to include it in the `aside`.

Now the third column consists of two elements: an `aside` and an `article`, stacked one on top of the other. To help hold them together and make them easier to style, we'll wrap them in a `div`. We're not using a `section`, or any other semantic markup, because that would imply that the `article` and the `aside` were somehow topically related. They're not—it's just a feature of our design that they happen to be in the same column together.

The New `main` Element

At this point, it's probably a good time to introduce another major structural element that's been introduced in HTML5: the `main` element. This element was not originally part of the HTML5 spec, but has been added since the first edition of this book was published.

Unfortunately, defining the `main` element and how it can be used is a little tricky. But let's start with where the element originated. In some HTML documents, developers were wrapping their primary content in a generic element, like this:

```
<body>
  <header>
    ...
  </header>

  <div id="main">
    ...
  </div>

  <footer>
```

```
    ...
  </footer>
</body>
```

Notice the generic `div` element used here as a sibling to the `header` and `footer` elements. Notice also the ID attribute with a value of `"main"`. In addition to this, many developers were adding an ARIA `role` to this element:

```
<div id="main" role="main">
  ...
</div>
```

We'll avoid going into the details of ARIA here—that's covered in Appendix B —but basically, the new `main` element is meant to replace this practice.

The W3C spec defines `main` as follows:[21] “The `main` element represents the main content of the body of a document or application. The main content area consists of content that is directly related to or expands upon the central topic of a document or central functionality of an application.”

The WHATWG spec defines it similarly; however, the two specs have very different definitions beyond that. The WHATWG spec says[22]:

> “There is no restriction as to the number of `main` elements in a document. Indeed, there are many cases where it would make sense to have multiple `main` elements. For example, a page with multiple `article` elements might need to indicate the dominant contents of each such element.”

But uncharacteristically, in complete contradiction to that, the W3C spec says:

> “Authors must not include more than one `main` element in a document. Authors must not include the `main` element as a descendant of an `article`, `aside`, `footer`, `header`, or `nav` element.”

In addition, the W3C spec adds the recommendation to use the `role="main"` attribute on the `main` element until the `main` element is fully recognized by user agents.

[21] http://www.w3.org/html/wg/drafts/html/master/grouping-content.html#the-main-element
[22] https://html.spec.whatwg.org/multipage/semantics.html#the-main-element

Having that knowledge, we're going to adopt the W3C's recommendation, and use only a single `main` element on our page, using an ARIA role as a fallback.

Going back to our *Herald* markup, this is how it will look after we've added the `main` element inside the `body` tag:

```
<body>
  <header>
    ...
  </header>
  <main role="main">

  </main>
  <footer>
    ...
  </footer>
  <script src="js/scripts.js"></script>
</body>
```

As you can see, the `main` element exists outside the `header` and `footer`. Inside the `main` is where we'll put the three columns we discussed, which make up the layout and primary content for *The HTML5 Herald*.

Continuing to Structure *The Herald*

The last part of our layout we'll consider here is the footer, which you can see in *The Herald* screenshot in its traditional location—at the bottom of the page. Because the footer contains a few different chunks of content, each of which forms a self-contained and topically related unit, we've split these out into `section` elements inside the footer. The author information will form one `section`, with each author sitting in their own nested `section`. Then there's another `section` for the copyright and additional information.

Let's add the new elements to our page so that we can see where our document stands:

```
<body>
  <header>
    <nav></nav>
  </header>
  <main role="main">
    <div class="primary">
      <article></article>
    </div>

    <div class="secondary">
      <article></article>
    </div>

    <div class="tertiary">
      <aside>
        <article></article>
      </aside>

      <article>
      </article>

    </div>
  </main><!-- main -->

  <footer>
    <section id="authors">
      <section></section>
    </section>
    <section id="copyright">
    </section>
  </footer>

  <script src="js/scripts.js"></script>
</body>
```

Figure 2.2 shows a screenshot that displays our page with some labels indicating the major structural elements we've used.

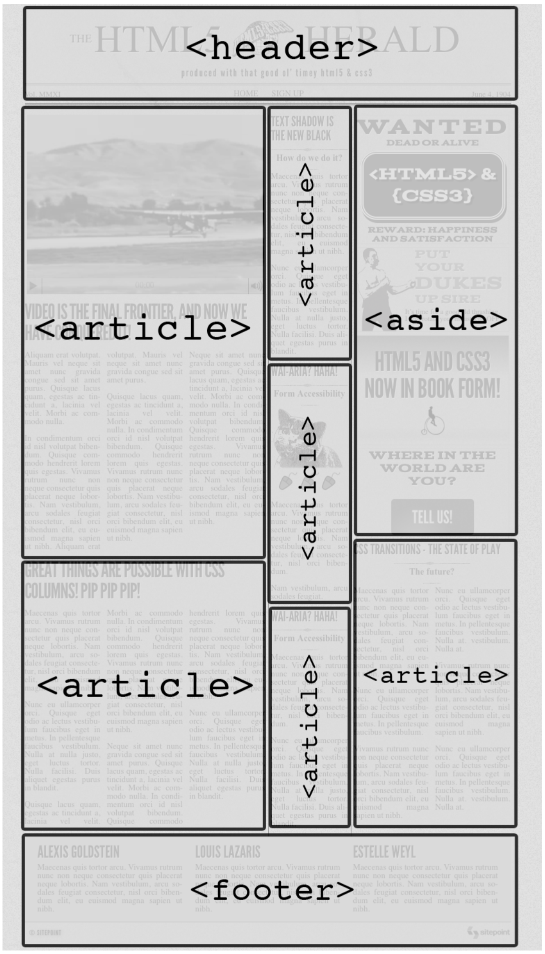


Figure 2.2. *The HTML5 Herald* broken into structural HTML5 elements

We now have a structure that can serve as a solid basis for the content of our website.

What if I use the wrong element?

Although it can be confusing at times to remember which elements to use in which situations, we encourage you to avoid stressing or spending too much time making decisions on semantics. While it is good to be consistent, there are few repercussions from using the wrong elements. If your pages are accessible, that's what is important. Of course, there are cases where the correct semantic element will be beneficial to accessibility, so we encourage you to research this and make sure your choice of element won't cause your pages to become inaccessible. A good place to start might be HTML5 Accessibility[23] or The Accessibility Project.[24]

Wrapping Things Up

That's it for this chapter. We've learned some of the basics of content structure in HTML5, and we've started to build our sample project using the knowledge we've gained.

In the next chapter, we'll have a more in-depth look at HTML5 content, and continue to add semantics to our page when we deal with some of the other elements available in HTML5.

[23] http://www.html5accessibility.com/
[24] http://a11yproject.com/