

Pro JavaScript™ RIA Techniques

Best Practices, Performance,
and Presentation



Den Odell

Pro JavaScript™ RIA Techniques: Best Practices, Performance, and Presentation

Copyright © 2009 by Den Odell

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1934-7

ISBN-13 (electronic): 978-1-4302-1935-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editors: Clay Andres and Jonathan Hassell

Technical Reviewer: Kunal Mittal

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editor: Marilyn Smith

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Esterman

Compositor: Lynn L'Heureux

Proofreader: Martha Whitt

Indexer: Carol Burbo

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.

For my family, friends, and loved ones

Contents at a Glance

About the Author.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi

PART 1 ■ ■ ■ Best Practices

■ CHAPTER 1	Building a Solid Foundation.....	3
■ CHAPTER 2	JavaScript for Rich Internet Applications	51

PART 2 ■ ■ ■ Performance

■ CHAPTER 3	Understanding the Web Browser.....	115
■ CHAPTER 4	Performance Tweaking.....	135
■ CHAPTER 5	Smoke and Mirrors: Perceived Responsiveness.....	179

PART 3 ■ ■ ■ Presentation

■ CHAPTER 6	Beautiful Typography.....	195
■ CHAPTER 7	Multimedia Playback	225
■ CHAPTER 8	Form Controls.....	249
■ CHAPTER 9	Offline Storage—When the Lights Go Out.....	307
■ CHAPTER 10	Binary Ajax	331
■ CHAPTER 11	Drawing in the Browser	357
■ CHAPTER 12	Accessibility in Rich Internet Applications	375
■ INDEX		403

Contents

About the Author.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi

PART 1 ■■■ Best Practices

■ CHAPTER 1	Building a Solid Foundation	3
	Best Practice Overview	3
	Who Put the “Best” in Best Practice?	4
	Who Benefits from Best Practices?	4
	General Best Practices	5
	Define the Project Goals	6
	Know the Basic Rules	6
	Markup Best Practice: Semantic HTML	14
	Learn the HTML Tags	14
	Start with a Document Type Definition	16
	How Do You Put the X in XHTML?	17
	Put Best Practice into Practice	19
	Accessibility Guidelines for Web Content	28
	Formatting Best Practice: CSS	30
	Regarding Pixel-Perfect Reproduction of Designs	30
	W3C CSS Standards	31
	Guidelines for Style Sheets	31
	Accessibility Guidelines for Styles	39
	Comment Blocks	41
	Browser Work-Arounds	42
	Localization Considerations	43

Structuring Your Folders, Files, and Assets	43
Readable URLs	43
File and Folder Naming	44
File Encoding	44
Organizing Assets	44
Setting Up Your Development Environment	46
Writing Your Files: Integrated Development Environments	46
Storing Your Files: Version Control Systems	47
Testing Your Pages: Browsers and Development Tools	47
Summary	49

CHAPTER 2 JavaScript for Rich Internet Applications

Coding Style Guidelines	51
Use Consistent Formatting	51
Use Braces and Brackets	52
Add Meaning with Letter Casing	53
Use Descriptive Variable and Function Names	53
Maintain Short Function Blocks	54
Use Comments As Documentation with ScriptDoc	56
Mark Remaining Tasks with TODO	57
Professional JavaScript Programming	57
Avoid Solving Nonexistent Problems	57
Use the Document Object Model	58
Don't Mix JavaScript and HTML	60
Separate Style from Code	60
Chain Function Calls	61
Write Bulletproof Code	61
Code with Localization in Mind	63
Object-Oriented JavaScript	64
Objects, Classes, and Constructors	64
Inheritance: Creating New Classes from Existing Ones	68
The this Keyword	71
Access to Properties and Methods	73
Object Literals and JavaScript Object Notation	75
Creating Namespaces and Hierarchies	77

Libraries and Frameworks	77
Selecting a Library	78
Building a JavaScript Library	79
Building RIAs	97
Structuring the Application	97
Managing Two Sets of HTML	100
Using Design Patterns	101
Testing and Test-Driven Development	107
Using Third-Party Scripts	110
Summary	111

PART 2 ■ ■ ■ Performance

CHAPTER 3	Understanding the Web Browser	115
	Engines: The Browser's Powerhouse	115
	The Rendering and JavaScript Engines	115
	JavaScript Engine Performance Benchmarking	116
	Anatomy of a Web Page Request	119
	HTTP: The Communication Standard Behind the Web	119
	HTTP Status Codes	125
	How Messages Are Transmitted	127
	Loading Order of an HTML Page	130
	Page Performance	131
	Viewing the Performance of a Page	131
	Identifying Potential Bottlenecks in Performance	132
	Summary	134
CHAPTER 4	Performance Tweaking	135
	Is Performance Really an Issue?	135
	Tweaking Your Web Server for Performance	137
	Use Separate Domain Names for External Assets	137
	Use a Content Delivery Network	137
	Send HTML to the Browser in Chunks	138
	Customize HTTP Headers to Force Browser Caching	140
	Compress the Output from the Server	141

Tweaking HTML for Performance	142
Shrink Your HTML File Size with HTML Tidy	143
Reference JavaScript Files at the End of Your HTML	143
Reduce the Number of HTTP Requests	144
Don't Load Every Asset from Your Home Page	146
Reduce Domain Name Lookups	146
Split Components Across Domains	147
Avoid Linking to Redirects	148
Reduce the Number of HTML Elements	148
Don't Link to Nonexistent Files	149
Reduce the Size of HTTP Cookies	149
Tweaking Your Style Sheets for Performance	150
Shrink Your CSS File Size with CSSTidy	150
Don't Use the @import Command	150
Speed Up Table Layouts	150
Avoid CSS Filters and Expressions in IE	151
Use Shorthand Values	151
Use the CSS Sprite Technique	155
Avoid Inefficient CSS Selectors	159
Tweaking Your Images for Performance	159
Understand Image File Formats	160
Optimize PNG Images	162
Don't Forget the Favicon	163
Tweaking Your JavaScript for Performance	163
Shrink Your JavaScript File Using Dojo ShrinkSafe	163
Access JavaScript Libraries Through CDNs	164
Timing Is Everything	164
Boost Core JavaScript Performance	166
Improve Ajax Performance	170
Improve DOM Performance	172
Summary	178

CHAPTER 5	Smoke and Mirrors: Perceived Responsiveness	179
	Providing Prompt Visual Feedback	179
	Time It Right	179
	Use CSS Pseudo-Classes on Hyperlinks	181
	Let the User Know the Form Is Being Submitted	181
	Change the Mouse Pointer	182
	Use a Web 2.0–Style Animated Indicator	183
	Show a Progress Bar	183
	Handling Long-Running Scripts	184
	Divide Long-Running Scripts into Chunks	185
	Use a Timer to Run Code Blocks Multiple Times	187
	Anticipating Your Site Visitors' Needs	189
	Preload Content	189
	Load Navigation Levels Efficiently	190
	Catch Mouse Clicks Early	192
	Summary	192

PART 3 ■ ■ ■ Presentation

CHAPTER 6	Beautiful Typography	195
	The Challenge	195
	The Basic Anatomy of a Font	196
	Using Static Images for Text	198
	Generating Images for Text Dynamically	199
	Using CSS to Embed Font Files Directly	199
	Having the Server Generate Text Images	202
	Generating Text in Custom Typefaces Using Flash	210
	Generating Text Using Vector Graphics	211
	Using Reusable Custom Font Components	211
	Text2PNG	211
	Scalable Inman Flash Replacement	215
	Facelift Image Replacement	219
	Typeface.js	221
	Summary	223

CHAPTER 7	Multimedia Playback	225
	Handling Accessibility	225
	Using Reusable Audio Playback Components	226
	The SoundManager Component	226
	Playing Audio Files Without Flash	231
	Using Reusable Video Playback Components	232
	YouTube Chromeless Player	235
	JW FLV Player	241
	The Future: Audio and Video in HTML 5	246
	The <audio> and <video> Tags	246
	JavaScript API	247
	Current Adoption Level	248
	Summary	248
CHAPTER 8	Form Controls	249
	Customizing Existing Form Controls	249
	Buttons	249
	Text Fields	253
	File Upload Controls	255
	Adding New Types of Form Controls	259
	A Calendar Widget for Date Selection	259
	Slider Control	281
	Using Reusable Form Components	296
	SWFUpload: Multiple File Uploads with Progress Bars	296
	TinyMCE: Rich Text Editing	301
	Validating Forms	304
	Summary	305
CHAPTER 9	Offline Storage—When the Lights Go Out	307
	Using Cookies to Store Data	307
	Creating Cookies	308
	The Downside of Cookies	310
	Using Internet Explorer's Data Store	311

Introducing the Data Storage APIs	314
The Local Storage API	314
Mozilla's Global Storage API	315
Client-Side Database Storage API	317
Storing Data Using Flash Shared Objects	322
Creating a Cross-Browser Local Data Storage API	323
Using a Reusable Offline Storage Component	330
Summary	330
 CHAPTER 10 Binary Ajax	331
Plain Text Files vs. Binary Files	331
Reading Binary Files with Ajax	331
Extracting Image Data from Photo Files	339
Understanding the EXIF Format	340
Reading EXIF Data Using JavaScript	341
Displaying EXIF Data from a File	351
Summary	356
 CHAPTER 11 Drawing in the Browser	357
Creating Scalable Vector Graphics	357
Creating SVG Image Files	358
Specifying SVG Within HTML	359
Specifying SVG Through JavaScript	361
Drawing with Vector Markup Language	362
Building Dynamic Graphs with a Reusable Drawing Library	363
Using the HTML 5 <canvas> Tag	371
Summary	373
 CHAPTER 12 Accessibility in Rich Internet Applications	375
Whose Needs Are We Meeting?	375
Users Using Assistive Technology	375
Users on Mobile Devices	375
Users Without a Mouse	376
Accessibility for All	377

- Proper Navigation with the Back and Forward Buttons 377
- Device-Independent JavaScript 383
 - Device-Independent Events 383
 - Device-Independent Event Delegation 384
 - Updated Content Alerts and Focus 386
- Web Accessibility Initiative: Accessible Rich Internet
 - Applications (WAI-ARIA). 390
 - Roles. 390
 - States and Properties 391
 - Focus Management. 393
 - Keyboard Interaction with ARIA Widgets 394
 - WAI-ARIA Examples. 394
 - Validation. 399
 - Testing 400
 - Summary. 401

- INDEX 403

About the Author

DEN ODELL is a multidisciplined web developer with expert JavaScript skills. He is a web standards and accessibility advocate, with a special passion for user interface development.

As a front-end technical architect at the AKQA digital service agency in London, Den built and architected several large-scale web sites and rich Internet applications for a number of clients, including Ferrari, Nike, and Nokia. He now lives in Sweden, where he has been using his technical skills and passion for music to help record labels and artists develop their presence on the Web.

In his spare time, Den runs nightclub events, plays records at clubs across Europe, and has a keen eye for digital photography.



About the Technical Reviewer

■ **KUNAL MITTAL** serves as Executive Director of Technology at Sony Pictures Entertainment, where he is responsible for the SOA and Identity Management programs. He provides a centralized engineering service to different lines of business and consults on content management, collaboration, and mobile strategies.

Kunal is an entrepreneur who helps startups define their technology strategy, product road map, and development plans. With strong relationships with several development partners worldwide, he is able to help startups and even large companies build appropriate development partnerships. He generally works in an advisor or a consulting CTO capacity, and serves actively in the project management and technical architect functions.

Kunal has authored and edited several books and articles on J2EE, WebLogic, and SOA. He holds a Master's degree in Software Engineering and is an instrument-rated private pilot.



Acknowledgments

Throughout the course of my career, I have met and worked alongside many incredibly intelligent people who have inspired me to improve my technical skills, and to varying degrees, have had an impact on this book and its material. There are way too many people to name, but I would like to thank you all—you know who you are.

Thanks to Clay Andres for seeing the potential in my book and allowing me to run with it. I'd also like to offer my sincere thanks to Kunal, Sofia, Jon, Marilyn, Laura, and the rest of the team at Apress who worked so diligently and effectively to run a tight ship for delivering such a high-quality product from my source material.

I want to offer massive thanks to Maria for supporting me when I was busy for what must have seemed like endless evenings and weekends as I wrote this book. Thank you for calming my stress, keeping me together, encouraging me to keep on when times were tough, and going above and beyond what anyone could expect. You are an amazing, beautiful, insightful, and intelligent person; I love you, and I can't imagine my life without you.

Thanks most of all to you, my readers, for taking the time to read and study this book. I hope you are able to understand, learn from, and put into practice its contents and build better web applications, and to advance your career as a result.

Introduction

Rich Internet applications (RIAs), or web applications, are those web sites that blur the boundary between the web browser and standard desktop applications. Managing your e-mail through web sites such as Google Gmail, Yahoo! Mail, and Microsoft Windows Live Hotmail is every bit as simple and intuitive as using a desktop e-mail client such as Microsoft Outlook or Apple Mail. Web page refreshes are not expected when performing actions, and if a new message is received by the mail server, we expect to see it appear in our inbox immediately.

Building web sites that behave in this way is seen as a departure from the traditional model on the Web, where performing actions such as submitting a form or clicking a link to refresh an online forum to see the latest posts were considered the norm. It is this difference that has led some to label these RIAs as Web 2.0, as if an upgrade to the Web were taking place.

In some respects an upgrade has been taking place, but not an upgrade of the Web itself. The improvements are actually in the web browsers we use to browse our favorite sites. Gradually over the past few years, features have been added to each of the major web browsers. Additionally, some attempts at conformance among browser manufacturers have meant that finally, through the power of JavaScript and standardized Document Object Model (DOM) scripting, live page updates are possible using data loaded dynamically from the web server. The Web is no longer a static place.

I have written this book primarily to help you harness the power of JavaScript to add dynamic components to your pages and to create entire RIAs of your own. (I assume you already have some knowledge of HTML, CSS, and JavaScript.) With great power comes great responsibility, however. I put emphasis on ensuring that you understand the importance of creating a responsive user experience that excites, rather than frustrates, your site visitors. I also stress that you have the ability to apply creativity through your design, to make your application look and behave superior to any static web site. You'll see how you can use custom-built user interface components that don't sacrifice usability or accessibility,

By the end of this book, you should have the confidence to build your own web site or RIA, safe in the knowledge that it has been constructed in a robust, reliable, efficient, beautiful, and highly accessible manner.

PART 1



Best Practices

In this first part of the book, I will present some tried-and-tested guidelines for building rich Internet applications (RIAs). Applying these guidelines will allow you to build the foundations of a web site structure that's scalable from a single page with a few lines of code up to many thousands of pages and thousands of lines of code. I will show you how to follow best practices in a sensible, pragmatic way that won't make the tasks of application maintenance and bug fixing daunting—during construction or in the future.



Building a Solid Foundation

If you're reading this book, chances are that you have felt the proud sense of achievement that comes with building and releasing a web site. Perhaps you completed the project solo; perhaps you built it as part of a team. Maybe it's a simple site, with just a few pages you're using to establish a presence for yourself on the Internet for an audience of a few of your friends, or maybe it's a cutting-edge rich Internet application (RIA) with social networking features for a potential audience of millions. In any case, congratulations on completing your project! You deserve to feel proud.

Looking back to the start of your project with the knowledge and experience you have garnered, I bet you can think of at least one thing that, if done differently, would have saved you from bashing your head against the wall. If you're just starting out in the web development industry, it might be that you wish you had kept a backup of a previous version of your files, because it cost you precious time trying to recover your changes after an unexpected power outage. Or it might be that you wish you hadn't decided to rely on that third-party software library that seemed like it would be up to the task at the start of the project, but soon proved itself to be a huge waste of time and effort. In the course of my own career, I've been in exactly these situations and come out the other side a little wiser. I've learned from those mistakes and fed that new knowledge back into the next project.

Based on my experiences and what I've learned from others, I've developed an effective, sensible approach to web development. This approach, along with a handful of smart techniques thrown in the mix, should minimize those head-bashing moments and ensure things run more smoothly right from the get-go all the way through to the launch of your next web site or application.

Best Practice Overview

Let's start by considering what is meant by the term *best practice*. If you've been in the development profession for long, you'll have heard this expression being tossed around quite a lot to justify a particular coding technique or approach. It is a bit of a loaded phrase, however, and should be treated with caution. I'll explain why.

Who Put the “Best” in Best Practice?

The landscape of web development is constantly changing. Browsers rise and fall in popularity, feature adoption between them is not always in parallel, and the technologies we use to construct web sites for display in such browsers are still fairly immature, constantly undergoing revisions and updates. In an environment that is in flux, what we might consider to be a best-practice solution to a problem right now could be obsolete in six months' time.

The use of the word *best* implies that a benchmark exists for comparison or that some kind of scientific testing has been adopted to make the distinction. However, very rarely have such tests been undertaken. You should consider carefully any techniques, technologies, and components that have been labeled as *best practice*. Evaluate them for yourself and decide if they meet a set of criteria that benefit you as a developer, the end users of your site, and if relevant, the client for whom you are undertaking the work.

The guidelines, rules, and techniques I set out in this chapter are ones that I have personally tried out and can attest to their suitability for real-world web development. I consider them to be the best we have right now. Of course, some of these could be irrelevant by the time you are reading this book, so my advice to you is to stay up-to-date with changes in the industry. Read magazines, subscribe to blog feeds, chat with other developers, and scour the Web for knowledge. I will maintain a comprehensive list of sources I recommend on my personal web site at <http://www.denode11.com/> to give you a place to start.

By staying abreast of changes to these best practices, you should be able to remain at the forefront of the web development industry, armed with a set of tools and techniques that will help you make your day-to-day work more efficient, constructive, and rewarding.

Finally, don't be afraid to review, rewrite, or refactor the code you write as you build your sites. No one has built a web site from scratch without needing to make code alterations. Don't believe for a second that any code examples you see on the Web, or in this or any other book, were written in a way that worked perfectly the first time. With that said, knowledge and experience make things easier, so practice every chance you get to become the best web developer you can be.

Who Benefits from Best Practices?

The truth is that everyone should be able to benefit from the use of best practices in your code. Take a look at the following lists, and use these criteria to assess any guidelines, techniques, or technologies you come across for their suitability for your site.

Web Developers

Best practice starts at home. A site structure and code that work well for you and your web developer colleagues will make all your lives a lot easier, and reduce the pain that can be caused by poor coding.

- Will my code adhere to World Wide Web Consortium recommendations?
- Will my site be usable if a proprietary technology or plug-in is unavailable?
- Will my code pass testing and validation?

- Is my code easily understood, well structured, and maintainable?
- Can extra pages, sections, and assets be added to the site without significant unnecessary effort?
- Can my code be localized for different languages and world regions without a lot of extra effort?

Search Engines and Other Automated Systems

Believe it or not, a large percentage of site traffic is from automated machines and scripts, such as search engines, screen scrapers, and site analysis tools. Designing for these robots is every bit as important as for any other group of users.

- Will my code appear appropriately in search engine results according to sensible search terms used to find it?
- Can my code be read simply and easily by a machine or script that wishes to read or parse its contents for whatever reason?

End Users

The most important users of your code are your site visitors, so making your code work effectively for them is the number one priority.

- Will my code be usable in and accessible to any web browser or device, regardless of its age, screen size, or input method?
- If my site were read aloud by screen reader software, would the content and its order make sense to the listener?
- Can I be confident my code will not demonstrate erroneous behavior or display error messages when used in a certain way I have not anticipated?
- Can my site be found through search engines or other online tools when using appropriate search terms?
- Can my users access a localized version of my site easily if one is available?

General Best Practices

If you're like most developers, you probably want to spend as much of your time as possible constructing attractive user interface components and great-looking web sites, rather than refactoring your code base because of an unfortunate architectural decision. It's very important to keep your code well maintained. Without sensible structure and readability, it will become harder and harder to maintain your code as time passes. Bear in mind that all the guidelines in this chapter have been put together with a view to making things as easy on you, the developer, as possible.

Define the Project Goals

The following are the two most important things to consider while coding a web page:

- How will end users want to use this?
- How will other developers want to make changes to this?

Bear in mind that the end users may not be human. If you were to check the server request logs for one of your existing sites, you would discover that many of your site visitors are actually search engine spiders, RSS readers, or other online services capable of reading your raw page content and transforming it into something else.

This kind of machine-based access is likely to become more widespread over the coming years, as automatic content syndication, such as RSS feeds, becomes more commonplace. For example, content from the popular knowledge-sharing site Wikipedia (<http://www.wikipedia.org/>) is already being used in other places around the Web, including within Google Maps (<http://maps.google.com/>), where articles are placed according to the geographical position of the content described in each article.

Yahoo! and other search engine companies have been pushing for some time for web developers to incorporate extra context-specific markup within pages, so that they can better understand the content and perhaps present the results in their search engine in a different way. Recipes could be presented with images and ingredients, for example; movie-related results could contain reviews and a list of where the movie is showing near you. The possibilities of connecting your code together with other developers' code online like this are vast. By marking up your content in the correct way, you ensure the whole system fits together in a sensible, coherent, connected way, which helps users get the information they are looking for faster.

As for ensuring other developers (including yourself, if only for your own sanity when you return to a project after a long break) can follow your code, you need to consider what the usual site maintenance tasks might be. These usually fall into the following four categories:

- Making alterations to existing pages
- Adding new pages
- Redesigning or modifying the page layout
- Adding support for end users who need the page in other languages, or in region- or country-specific versions

By thinking about these tasks up-front, you reduce the likelihood of needing to refactor your code or rearrange and split up files, so the job of maintenance is made easier. Welcome back, sanity!

Know the Basic Rules

So how do we go about making sure that we get it right in the first place? The following seven rules of thumb seem to sum it up succinctly:

- Always follow mature, open, and well-supported web standards.
- Be aware of cross-browser differences between HTML, CSS, and JavaScript implementations, and learn how to deal with them.

- Assume HTML support, but allow your site to be usable regardless of whether any other technologies—such as CSS, JavaScript, or any plug-ins—are present in the browser.
- Name your folders and files consistently, and consider grouping files together according to purpose, site structure, and/or language.
- Regularly purge redundant code, files, and folders for a clean and tidy code base.
- Design your code for performance.
- Don't overuse technology for its own sake.

Let's go through each of these basic rules in order.

Follow Mature, Open, and Well-Supported Web Standards

Back in the early 1990s, a very clever man who worked at the technology research organization CERN (European Organization for Nuclear Research, <http://www.cern.ch/>), Tim Berners-Lee, invented what we know today as the World Wide Web. He developed the concepts of home pages, Hypertext Markup Language (HTML), and interconnected hyperlinks that form the foundation of web browsing. He also created the world's first web browser to demonstrate his invention.

The project became quite large and eventually took up many resources at CERN. When the decision was made to redirect funding and talent toward building the recently completed Large Hadron Collider project instead, Tim Berners-Lee made the decision to create a separate organization to manage the continuation of standards development for HTML and its related technologies. This new organization, the World Wide Web Consortium (W3C, <http://www.w3.org/>), was born in October 1994.

Since its inception, the W3C organization has documented more than 110 recommended standards and practices relating to the Web. The three that are most useful to readers of this book are those pertaining to HTML (including XHTML), Cascading Style Sheets (CSS), and Domain Object Model (DOM) scripting with JavaScript (also known as ECMAScript, since the JavaScript name is trademarked by Sun Microsystems).

Two popular browsers emerged in those early days of the Web: Netscape Navigator, released in December 1994, and Microsoft's Internet Explorer (IE), released in August 1995. Both browsers were based on similar underlying source code and rendered web pages in a similar way. Of course, a web page at the time was visibly very different from what we see today, so this wasn't particularly difficult for both to achieve.

Roll on a year to 1996, and things get a little more interesting. Microsoft introduced basic support for a new W3C recommendation, CSS level 1, in IE 3. This recommendation defined a way for web developers to apply font and color formatting; text alignment; and margins, borders, and padding to most page elements. Netscape soon followed suit, and competition began to intensify between the two browser manufacturers. They both were attempting to implement new and upcoming W3C recommendations, often before those recommendations were ready for the mainstream.

Naturally, such a variation in browser support for standards led to confusion for web developers, who often tended to design for an either/or scenario. This resulted in end users facing web sites that displayed the message "This web site works only in Internet Explorer. Please upgrade your browser."

Of course, the W3C recommendations are just that: recommendations for browser manufacturers and developers to follow. As developers, we must consider them only as useful as their actual implementation in common web browsers. Over time, browsers have certainly made strides toward convergence on their implementations of these web standards. Unfortunately, older versions of browsers with poorer quality of standards adoption in their rendering of web pages still exist, and these must be taken into account by web developers.

The principle here is to ensure you are up-to-date on common standards support in browsers, rather than just on the latest recommendations to emerge from the W3C. If the standard is well supported, you should use it. If not, it is best avoided.

Deal with Cross-Browser Issues

Web browsers are regularly updated, and they quite often feature better support for existing W3C recommendations and some first attempts at implementations of upcoming recommendations.

Historically, browsers have varied in their implementations of existing recommendations. This is also true of browser support for the newer recommendations. This means that developers must aim to stay up-to-date with changes made to browser software and be aware of the features and limitations of different browsers.

Most browser users, on the other hand, tend not to be quite as up-to-date with new browser releases as developers would wish. Even browsers that are capable of automatically updating themselves to the latest version often require the user to authorize the upgrade first. Many users actually find these notifications distracting to what they're trying to achieve in their web browser then and there, and so they tend to put off the upgrade.

As developers, we must be aware and acknowledge that there are many different web browsers and versions of web browsers in the world (some 10,000 different versions in total, and counting). We have no control over which particular piece of software the end user is using to browse our pages, nor should we.

What we do know from browser statistics sites, such as Net Applications' Market Share (<http://marketshare.hitslink.com/>), is that the five main web browsers in the world today are Microsoft's IE, Mozilla's Firefox, Apple's Safari, Opera Software's Opera, and Google's Chrome. These five browsers account for around 99% of all access to web pages through the desktop. However, just relying on testing in these browsers misses out on the burgeoning market in mobile web browsing, for example, so it is worth staying up-to-date with the latest progress in the web browser market.

Testing your pages across a multitude of browsers and operating systems allows you to locate the portions of your code that cause different browsers to interpret it in different ways. Minimizing these differences is one of the hardest tasks for any web developer and separates this role from most other software-related professions. This is a task that needs to be attacked from the get-go of a new project, as leaving it until too late can result in frantic midnight coding sessions and missed deadlines—never fun!

The smartest approach is to build the HTML, CSS, and JavaScript code that form the basic template or outline of the site before writing any page-specific code. Then test this bare-bones structure in as wide a range of browsers on as many different operating systems, and with as varied a range of monitor and window sizes, as possible. Tweak the code to ensure the template displays correctly before adding any page-specific code or content.

A particular source of variation is in the different interpretations of color within browsers. Some support the reading of color profile information from image files; some don't support this. Some apply a gamma correction value; some don't apply this value. Consequently, the same image or color can appear slightly different in various browsers, so it's worth checking that your design doesn't cause color mismatching to occur between objects on your page.

You should build and test individual page components one at a time in as many browsers as possible during development. Again, by bringing most of the testing up-front to coincide with development, you will experience fewer problems later on and have fewer bugs to squish. By the end of a project, developers are often feeling the pressure of last-minute client requests for changes, so minimizing bugs by this stage in the proceedings is a smart idea.

Assume Support for HTML Only

Your HTML markup must be visible and operate functionally in any available browser, device, or user agent without reliance on CSS, JavaScript, or plug-ins. Where CSS, JavaScript, or plug-ins provide additional content, layout, or functionality over and above the HTML, the end users should be able to access the content and a functional equivalent of the behavior in a sensible way, without reliance on these technologies. For example, if you're using a Flash movie to provide an animated navigation menu for your site, you need to ensure the same navigation is available through HTML; otherwise, you are preventing a whole group of users from accessing your site.

Obviously, this has a massive impact on the way you develop your web pages. You will build from the HTML foundations upward, ensuring no functionality gets lost when certain browser features are switched off or are nonexistent. Each "layer" of code should be unobtrusive; that is to say that no CSS style rules or JavaScript code should exist within the HTML markup—each should be in a separate file and stand alone.

In the context of modern web applications, which are often written in such a way so that communication between the browser and the server is handled via JavaScript, this means that those communication points must exist when JavaScript is switched off in the browser. For example, modern JavaScript allows data to be sent to and received from a web server without the need for the page to refresh when sending a form. In this case, you must ensure that the form can be submitted without the need for JavaScript—treat it like an optional extra, rather than a requirement.

You might hear this principle called *progressive enhancement*, referring to the adding or layering of extra functionality on top of the HTML, or *graceful degradation*, referring to the fact that the removal of features from the browser always results in a working web page. It is the central principle of what is termed *accessibility*, which refers to providing access to a web page regardless of browser or device.

This principle is best understood through real-life examples, so let's go through two of them now.

First, suppose that in your web application, you have a button that, when clicked, launches a login modal dialog box within the page, as shown in Figure 1-1. After the user fills in the form and clicks the submit button, JavaScript is used to send the supplied login credentials to the server, and then to perform a refresh of certain page elements, instead of the entire page, based on the user's logged-in status as shown in Figure 1-2.

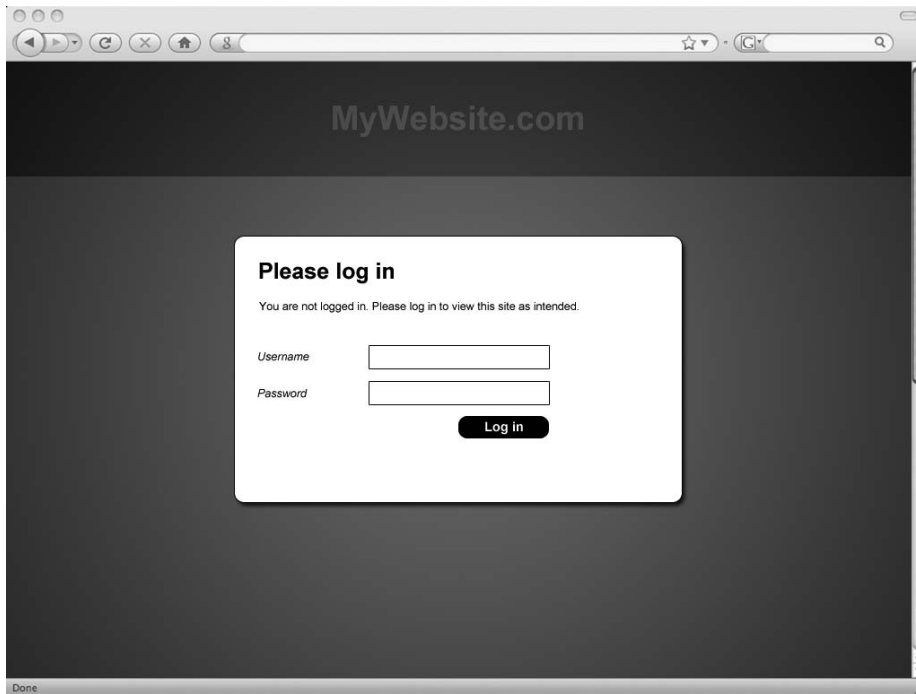


Figure 1-1. A modal-style login box

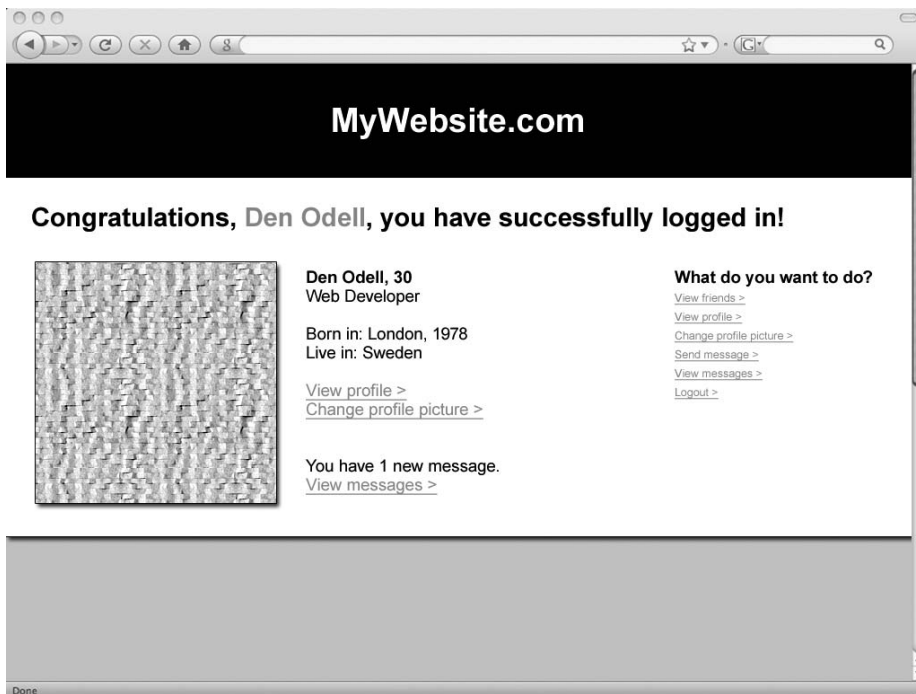


Figure 1-2. Successful login, loaded without a refresh if JavaScript is enabled

But what if JavaScript is disabled? You would need to ensure that your HTML code was structured such that the user would be taken to a separate page with the login form. Submitting this form would post the data back to the server, causing a refresh, and the server-side code would decide which page to send according to the user's status—either successfully logged in or not logged in. In this way, both scenarios are made to be functionally equivalent, although their user flow and creative treatment could potentially be different.

As another example, suppose you have a page that contains a form used for collecting payment information for an online booking system. Within this form, depending on the type of credit card selected, you would like certain fields to display only if the user selects a credit card, as shown in Figure 1-3, rather than a debit card, as shown in Figure 1-4. For instance, the Issue Number field is applicable only to debit cards, and perhaps you want to display the Valid from Date fields only for cards from certain suppliers. You probably also want to make it impossible for the user to submit an incorrect date, such as February 30.

As web developers, we use JavaScript to make this happen. JavaScript fires events when the user performs certain actions within the browser, and we are able to assign code to execute when these events are fired. We even have the power to cancel the event, meaning that if the user attempted to submit a form, for example, we could cancel that submission if we decided that form wasn't suitable for submission because it had failed some validation tests.

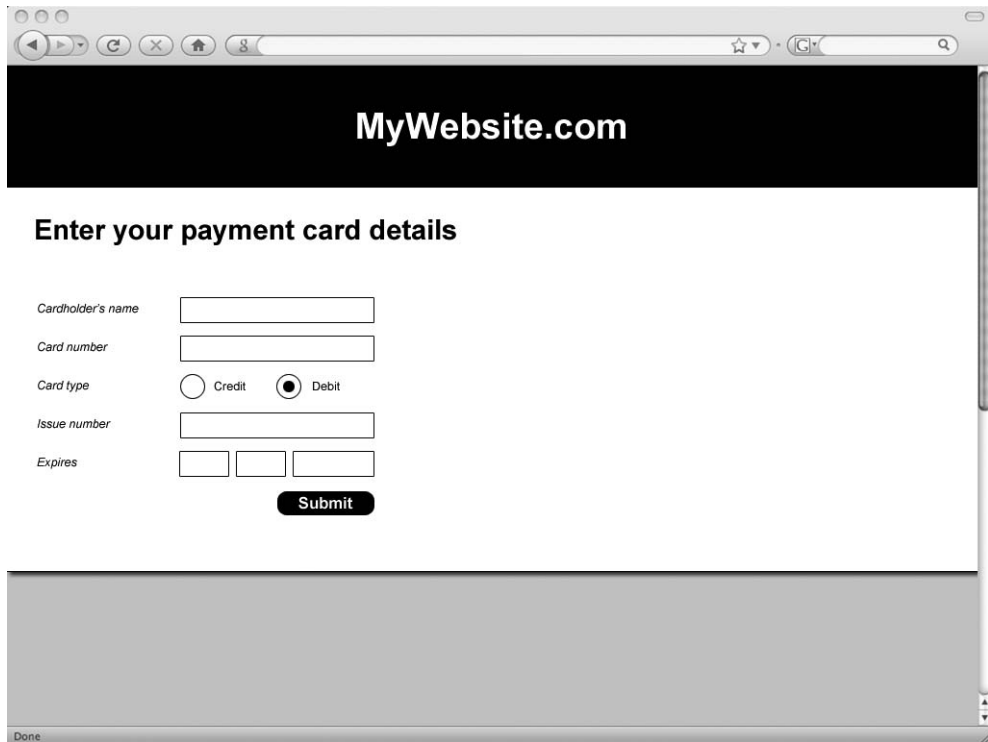
We use JavaScript to “listen” for changes to the Card Type field. This event gets fired when the user selects a different radio button option. When this event is fired, we can execute a piece of code that, depending on the card type selected, shows or hides the desired fields.

The screenshot shows a web browser window with the address bar displaying 'MyWebsite.com'. The main content area has a black header with the text 'MyWebsite.com' in white. Below the header, the form is titled 'Enter your payment card details'. The form contains the following fields and controls:

- Cardholder's name:** A single-line text input field.
- Card number:** A single-line text input field.
- Card type:** Two radio buttons labeled 'Credit' and 'Debit'. The 'Credit' radio button is selected.
- Valid from:** Three adjacent text input fields for the day, month, and year.
- Expires:** Three adjacent text input fields for the day, month, and year.
- Submit:** A black button with the text 'Submit' in white.

The browser's status bar at the bottom shows 'Done'.

Figure 1-3. A payment card form showing credit card fields



The image shows a web browser window with the address bar displaying 'MyWebsite.com'. The main content area has a black header with the text 'MyWebsite.com' in white. Below the header, the text 'Enter your payment card details' is displayed in bold. The form consists of the following fields and controls:

- Cardholder's name:** A single-line text input field.
- Card number:** A single-line text input field.
- Card type:** Two radio buttons labeled 'Credit' and 'Debit'. The 'Debit' radio button is selected.
- Issue number:** A single-line text input field.
- Expires:** Three separate single-line text input fields for the month, year, and day.
- Submit:** A black button with the text 'Submit' in white.

The browser's status bar at the bottom shows 'Done'.

Figure 1-4. A payment card form showing debit card fields

We also listen for the submit event of the form to fire and, when it does, we run a small JavaScript routine to check that the date fields contain valid values. We can force the form submission to fail if we decide the values entered are not up to scratch.

Now what happens when someone visits your web page with a browser that doesn't support JavaScript? She selects her card type using the radio button, but nothing changes. In fact, in order to instantiate a change to the appearance of the page, the form must be submitted to the server to allow the server to perform the kind of processing you had been performing using JavaScript.

In terms of usability, you might consider it odd to ask the users to submit the form after they have selected their card type, as the fields are already displayed below. Probably the ideal way to structure your page in this case is to have all of the fields existing in the page's HTML, and simply allow the users to fill in the information they have available on their card. When they finally submit the form, the processing that exists on the server can validate their card data and check whether they have entered a valid date, and if there is an error, reload the page displaying an error message.

Name and Group Folders and Files Consistently

By establishing rules and conventions regarding the naming of folders, files, and their contents, you make the task of locating files and code a lot easier for yourself and other developers. The task of maintenance and future additions is made simpler with a consistent naming convention, ensuring developers always know how to name their assets. See the "Structuring

Your Folders, Files, and Assets” section later in this chapter for some examples of directory structures you might adopt.

Maintain a Tidy Code Base

You should ensure that the files and code associated with a project are the only ones necessary for the web site to do its job—no more and no less. Over time, certain files may be superseded by others, and certain CSS style rules or JavaScript files by others.

I recommend that you purge all redundant files, folders, and code from your code base on a regular basis during development. This reduces the size of the project, which aids comprehension of the code by other developers and ensures the end users of your site are not downloading files that are never used, consuming bandwidth that they could potentially be paying for.

To avoid problems with the accidental deletion of files or the situation where you later require files you’ve deleted, you should consider using a source code management system. Such a system will keep backups of changes made to project files and ensure you can always revert to a previous version of a particular folder or file—even a deleted one. See the “Storing Your Files: Version Control System” section later in this chapter for more information.

Design Your Code for Performance

Your site visitors, whether they realize it or not, demand a responsive user interface on the Web. If a button is clicked, the users expect some kind of reaction to indicate that their action was recognized and is being acted upon.

HTML, CSS, and JavaScript code run within the browser and are reliant on the power of the end user’s machine or device. Your code needs to be lightweight and efficient so it downloads quickly, displays correctly, and reacts promptly. Part 2 of this book focuses on performance and explains how you can make your code lighter, leaner, and faster for the benefit of your end users.

Don’t Use Technology for Its Own Sake

Within the wider web development community, you will often hear hype about new technologies that will make your web pages better in some way. Most recently, this hype has focused around the Asynchronous JavaScript and XML (Ajax) technique, which is the practice of communicating with the server using JavaScript within the browser, meaning that page refreshes can be less frequent. This became the favorite technique to be used by web developers on any new project.

The problem is that sites were built so that they worked only with the Ajax technique, and so relied exclusively upon JavaScript. Those users without this capability in their browsers—users with some mobile web browsers, users with restrictions in place in their office environment, users with special browser requirements due to a disability, and external robots such as search engine spiders—could not access the information that would normally have been provided through HTML pages, connected together through hyperlinks and buttons. Conversely, some users with capable browsers were finding that if they remained on certain sites that relied heavily on the Ajax technique, eventually their browser would become slow or unresponsive. Some web developers, keen to jump onboard the new craze, forgot to code in a way that would prevent memory leaks from occurring in the browser.

Build your sites on sound foundations and solid principles, ensuring you test and push new technologies to usable limits before deciding they are a good choice for your project. You'll learn about the Ajax technique in Chapter 2, and how to deal with memory leaks in browsers in Chapter 4.

Markup Best Practice: Semantic HTML

HTML or XHTML forms the basic foundation of every web page. Technically, these are the only web standards that need to be supported by all web browsers and user agents out there in the wild. The term *semantic* in this context refers to applying the correct tags to match the meaning behind the content (according to the dictionary, the word *semantic* literally means *meaning*).

Knowledge of as many of the HTML/XHTML tags and attributes as possible will put you in good stead. Make sure that your content is marked up with exactly the right tag for the content it encompasses: table tags for tabular data, heading tags for section headlines, and so on. The more meaning you are able to give your content, the more capable web browsers, search engine spiders, and other software will be at interpreting your content in the intended way.

It is advisable to include all semantic information in your markup for a page, even if you chose to use CSS style rules to hide some elements visually within the browser. A useful guideline is that you should code your markup according to how it would sound if read aloud. Imagine the tag name were read aloud, followed by the contents of that tag. In fact, this is how most screen reader browsers work, providing audio descriptions of web pages for those with visual impairments.

For example, suppose you've built a web site for movie reviews, and you want to display an image that denotes the movie has scored four out of five possible stars. Now consider how you would want this information to be read aloud—something like, “rated four out of a possible five stars.” Say you put this text within the HTML, so that everyone can access it. But you don't want this text to be displayed on the page; you want only the image of four stars to appear. This is where CSS comes into play. You can apply a `class` attribute to the tag surrounding this text, and target this class using CSS to specify that the text be hidden and an image displayed according to a specified size. The style rules for hiding portions of text in a way that works for all browsers, including screen readers, are covered in the “Formatting Best Practice: CSS” section later in this chapter. The HTML for this part of the page might look like this:

```
<div class="rated-four-out-of-five">  
  This movie was rated four out of a possible five stars.  
</div>
```

Learn the HTML Tags

If you're an experienced web developer who has worked on multiple sites, and you've been marking up your content semantically, you're already familiar with a whole host of tags: `<h1>`, `<h2>`, `<p>`, ``, ``, and ``, to name a few. However, a number of less common tags are rarely at the forefront of developers' minds. Without some of these tags, you risk marking up your documents in the wrong way, missing an opportunity to add meaning to your content for the benefit of your users, search engines, and others.

The following are a few tags that add important meaning for the browser or end user, but are commonly forgotten:

`<abbr>`: Abbreviation, used for marking up inline text as an abbreviation. In many browsers, hovering the mouse over the text reveals the unabbreviated version.

```
<abbr title="et cetera">etc.</abbr>
```

`<acronym>`: Acronym, used for marking up inline text as an acronym. In many browsers, hovering the mouse over the text reveals the elongated version.

```
<acronym title="World Wide Web">WWW</acronym>
```

`<address>`: Contact information for page. At first glance, you may think this tag should be used to mark up postal addresses listed on the page. However, that is an incorrect usage of the tag. It should be used only to mark up the contact details of the page author. (Of course, a postal address could be part of that information.)

```
<address>
  Author: Den Odell<br />
  <a href="mailto:me@denodell.com">Email the author</a>
</address>
```

`<blockquote>`: Long quotation. An important point to note about block quotes that often gets missed is that the tag may contain only block-level elements. Therefore, the quote itself must, at the very least, be enclosed by a paragraph or other block-level element.

```
<blockquote>
  <p>If music be the food of love, play on,<br />
  Give me excess of it, that surfeiting,<br />
  The appetite may sicken, and so die.</p>
</blockquote>
```

`<ins>` and ``: Inserted and deleted copy. `` is used to show that one piece of content has been deleted. `<ins>` shows that another piece has been inserted into a page. For example, these tags might be used on a blog post where the author has, after publication, returned to the piece and edited it to alter a particular sentence. The tags can be used to show this in a semantic way. Often, content within a `` tag will be rendered in the browser as struck through with a line.

There are `50` `<ins>60</ins>` million inhabitants of the UK

Keep these tags in mind as you code your pages. See if you can spot opportunities to work them into your markup to denote the correct meaning of your content.

Tip Keep a reference list of tags and attributes on hand when developing, and revise that list occasionally. A great online resource for XHTML tags and attributes can be found at <http://www.w3schools.com/tags/>.
